CUDA Math API Reference Manual

Release 13.0

NVIDIA Corporation

Oct 02, 2025

Contents

1 FP4 Intrinsics **3**
- 1.1 C++ struct for handling fp4 data type of e2m1 kind. 3
- 1.2 C++ struct for handling vector type of four fp4 values of e2m1 kind. 3
- 1.3 C++ struct for handling vector type of two fp4 values of e2m1 kind. 4
- 1.4 FP4 Conversion and Data Movement 4
  - 1.4.1 Enumerations 7
  - 1.4.2 Functions 7
  - 1.4.3 Typedefs 10

2 FP6 Intrinsics **13**
- 2.1 C++ struct for handling fp6 data type of e2m3 kind. 13
- 2.2 C++ struct for handling fp6 data type of e3m2 kind. 13
- 2.3 C++ struct for handling vector type of four fp6 values of e2m3 kind. 14
- 2.4 C++ struct for handling vector type of four fp6 values of e3m2 kind. 14
- 2.5 C++ struct for handling vector type of two fp6 values of e2m3 kind. 14
- 2.6 C++ struct for handling vector type of two fp6 values of e3m2 kind. 14
- 2.7 FP6 Conversion and Data Movement 15
  - 2.7.1 Enumerations 19
  - 2.7.2 Functions 20
  - 2.7.3 Typedefs 23

3 FP8 Intrinsics **25**
- 3.1 C++ struct for handling fp8 data type of e4m3 kind. 25
- 3.2 C++ struct for handling fp8 data type of e5m2 kind. 25
- 3.3 C++ struct for handling vector type of four fp8 values of e4m3 kind. 26
- 3.4 C++ struct for handling vector type of four fp8 values of e5m2 kind. 26
- 3.5 C++ struct for handling vector type of four scale factors of e8m0 kind. 26
- 3.6 C++ struct for handling vector type of two fp8 values of e4m3 kind. 26
- 3.7 C++ struct for handling vector type of two fp8 values of e5m2 kind. 27
- 3.8 C++ struct for handling vector type of two scale factors of e8m0 kind. 27
- 3.9 FP8 Conversion and Data Movement 27
  - 3.9.1 Enumerations 36
  - 3.9.2 Functions 37
  - 3.9.3 Typedefs 42

4 Half Precision Intrinsics **45**
- 4.1 Half Arithmetic Constants 45
  - 4.1.1 Macros 46
- 4.2 Half Arithmetic Functions 46
  - 4.2.1 Functions 48
- 4.3 Half Comparison Functions 55
  - 4.3.1 Functions 56
- 4.4 Half Math Functions 62

4.4.1 Functions 63
4.5 Half Precision Conversion and Data Movement 68
4.5.1 Functions 76
4.6 Half2 Arithmetic Functions 106
4.6.1 Functions 107
4.7 Half2 Comparison Functions 114
4.7.1 Functions 117
4.8 Half2 Math Functions 131
4.8.1 Functions 132
4.9 Typedefs 138

5 Bfloat16 Precision Intrinsics **139**
5.1 Bfloat16 Arithmetic Constants 139
5.1.1 Macros 140
5.2 Bfloat16 Arithmetic Functions 141
5.2.1 Functions 142
5.3 Bfloat16 Comparison Functions 148
5.3.1 Functions 149
5.4 Bfloat16 Math Functions 155
5.4.1 Functions 156
5.5 Bfloat16 Precision Conversion and Data Movement 160
5.5.1 Functions 168
5.6 Bfloat162 Arithmetic Functions 196
5.6.1 Functions 198
5.7 Bfloat162 Comparison Functions 204
5.7.1 Functions 207
5.8 Bfloat162 Math Functions 220
5.8.1 Functions 221
5.9 Typedefs 226

6 Single Precision Mathematical Functions **227**
6.1 Functions 231

7 Single Precision Intrinsics **263**
7.1 Functions 266

8 Double Precision Mathematical Functions **287**
8.1 Functions 292

9 Double Precision Intrinsics **323**
9.1 Functions 324

10 FP128 Quad Precision Mathematical Functions **339**
10.1 Functions 342

11 Type Casting Intrinsics **361**
11.1 Functions 364

12 Integer Mathematical Functions **379**
12.1 Functions 381

13 Integer Intrinsics **385**
13.1 Functions 387

14 SIMD Intrinsics **395**
14.1 Functions 401

15 Structs **421**
15.1 __half . . . 421
15.2 __half2 . . . 425
15.3 __half2_raw . . . 426
15.4 __half_raw . . . 427
15.5 __nv_bfloat16 . . . 427
15.6 __nv_bfloat162 . . . 430
15.7 __nv_bfloat162_raw . . . 431
15.8 __nv_bfloat16_raw . . . 432
15.9 __nv_fp4_e2m1 . . . 432
15.10 __nv_fp4x2_e2m1 . . . 434
15.11 __nv_fp4x4_e2m1 . . . 435
15.12 __nv_fp6_e2m3 . . . 436
15.13 __nv_fp6_e3m2 . . . 437
15.14 __nv_fp6x2_e2m3 . . . 438
15.15 __nv_fp6x2_e3m2 . . . 439
15.16 __nv_fp6x4_e2m3 . . . 440
15.17 __nv_fp6x4_e3m2 . . . 441
15.18 __nv_fp8_e4m3 . . . 442
15.19 __nv_fp8_e5m2 . . . 445
15.20 __nv_fp8_e8m0 . . . 447
15.21 __nv_fp8x2_e4m3 . . . 450
15.22 __nv_fp8x2_e5m2 . . . 451
15.23 __nv_fp8x2_e8m0 . . . 452
15.24 __nv_fp8x4_e4m3 . . . 453
15.25 __nv_fp8x4_e5m2 . . . 454
15.26 __nv_fp8x4_e8m0 . . . 455

16 Notices **459**
16.1 Notice . . . 459
16.2 OpenCL . . . 460
16.3 Trademarks . . . 460

CUDA mathematical functions are always available in device code.

Host implementations of the common mathematical functions are mapped in a platform-specific way to standard math library functions, provided by the host compiler and respective host libm where available. Some functions, not available with the host compilers, are implemented in crt/math_functions.hpp header file. For example, see *erfinv()*. Other, less common functions, like *rhypot()*, *cyl_bessel_i0()* are only available in device code.

CUDA Math device functions are no-throw for well-formed CUDA programs.

Note that many floating-point and integer functions names are overloaded for different argument types. For example, the *log()* function has the following prototypes:

```
double log(double x);
float log(float x);
float logf(float x);
```

Note also that due to implementation constraints, certain math functions from std:: namespace may be callable in device code even via explicitly qualified std:: names. However, such use is discouraged, since this capability is unsupported, unverified, undocumented, not portable, and may change without notice.

Chapter 1. FP4 Intrinsics

This section describes fp4 intrinsic functions.

To use these functions, include the header file `cuda_fp4.h` in your program.

The following macros are available to help users selectively enable/disable various definitions present in the header file:

- `__CUDA_NO_FP4_CONVERSIONS__` - If defined, this macro will prevent any use of the C++ type conversions (converting constructors and conversion operators) defined in the header.
- `__CUDA_NO_FP4_CONVERSION_OPERATORS__` - If defined, this macro will prevent any use of the C++ conversion operators from `fp4` to other types.

Note: Most of the operations defined here benefit from native HW support when compiled for specific GPU targets (e.g. devices of compute capability 10.0a), other targets use emulation path.

1.1. C++ struct for handling fp4 data type of e2m1 kind.

Structs

__nv_fp4_e2m1
__nv_fp4_e2m1 datatype

1.2. C++ struct for handling vector type of four fp4 values of e2m1 kind.

Structs

__nv_fp4x4_e2m1
__nv_fp4x4_e2m1 datatype

1.3. C++ struct for handling vector type of two fp4 values of e2m1 kind.

Structs

__nv_fp4x2_e2m1
__nv_fp4x2_e2m1 datatype

1.4. FP4 Conversion and Data Movement

To use these functions, include the header file `cuda_fp4.h` in your program.

Enumerations

__nv_fp4_interpretation_t
Enumerates the possible interpretations of the 4-bit values when referring to them as `fp4` types.

Functions

__host__ __device__ __nv_fp4x2_storage_t *__nv_cvt_bfloat16raw2_to_fp4x2*(const __nv_bfloat162_raw x, const __nv_fp4_interpretation_t fp4_interpretation, const enum cudaRoundMode rounding)
Converts input vector of two `nv_bfloat16` precision numbers packed in `__nv_bfloat162_raw` `x` into a vector of two values of `fp4` type of the requested kind using specified rounding mode and saturating the out-of-range values.

__host__ __device__ __nv_fp4_storage_t *__nv_cvt_bfloat16raw_to_fp4*(const __nv_bfloat16_raw x, const __nv_fp4_interpretation_t fp4_interpretation, const enum cudaRoundMode rounding)
Converts input `nv_bfloat16` precision `x` to `fp4` type of the requested kind using specified rounding mode and saturating the out-of-range values.

__host__ __device__ __nv_fp4x2_storage_t *__nv_cvt_double2_to_fp4x2*(const double2 x, const __nv_fp4_interpretation_t fp4_interpretation, const enum cudaRoundMode rounding)
Converts input vector of two `double` precision numbers packed in `double2` `x` into a vector of two values of `fp4` type of the requested kind using specified rounding mode and saturating the out-of-range values.

__host__ __device__ __nv_fp4_storage_t *__nv_cvt_double_to_fp4*(const double x, const __nv_fp4_interpretation_t fp4_interpretation, const enum cudaRoundMode rounding)
Converts input `double` precision `x` to `fp4` type of the requested kind using specified rounding mode and saturating the out-of-range values.

__host__ __device__ __nv_fp4x2_storage_t *__nv_cvt_float2_to_fp4x2*(const float2 x, const __nv_fp4_interpretation_t fp4_interpretation, const enum cudaRoundMode rounding)
Converts input vector of two `single` precision numbers packed in `float2` `x` into a vector of two values of `fp4` type of the requested kind using specified rounding mode and saturating the out-of-range values.

__host__ __device__ __nv_fp4_storage_t *__nv_cvt_float_to_fp4*(const float x, const __nv_fp4_interpretation_t fp4_interpretation, const enum cudaRoundMode rounding)
Converts input `single` precision `x` to `fp4` type of the requested kind using specified rounding mode and saturating the out-of-range values.

__host__ __device__ __half_raw *__nv_cvt_fp4_to_halfraw*(const __nv_fp4_storage_t x, const __nv_fp4_interpretation_t fp4_interpretation)
Converts input `fp4` `x` of the specified kind to `half` precision.

__host__ __device__ __half2_raw *__nv_cvt_fp4x2_to_halfraw2*(const __nv_fp4x2_storage_t x, const __nv_fp4_interpretation_t fp4_interpretation)
Converts input vector of two `fp4` values of the specified kind to a vector of two `half` precision values packed in `__half2_raw` structure.

__host__ __device__ __nv_fp4x2_storage_t *__nv_cvt_halfraw2_to_fp4x2*(const __half2_raw x, const __nv_fp4_interpretation_t fp4_interpretation, const enum cudaRoundMode rounding)
Converts input vector of two `half` precision numbers packed in `__half2_raw` `x` into a vector of two values of `fp4` type of the requested kind using specified rounding mode and saturating the out-of-range values.

__host__ __device__ __nv_fp4_storage_t *__nv_cvt_halfraw_to_fp4*(const __half_raw x, const __nv_fp4_interpretation_t fp4_interpretation, const enum cudaRoundMode rounding)
Converts input `half` precision `x` to `fp4` type of the requested kind using specified rounding mode and saturating the out-of-range values.

__host__ __device__ *__nv_fp4_e2m1::__nv_fp4_e2m1*()
Constructor by default.

__host__ __device__ *__nv_fp4_e2m1::__nv_fp4_e2m1*(const unsigned long int val)
Constructor from `unsigned long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp4_e2m1::__nv_fp4_e2m1*(const double f)
Constructor from `double` data type, relies on `__NV_SATFINITE` behavior for out-of-range values and `cudaRoundNearest` rounding mode.

__host__ __device__ *__nv_fp4_e2m1::__nv_fp4_e2m1*(const long int val)
Constructor from `long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp4_e2m1::__nv_fp4_e2m1*(const float f)
Constructor from `float` data type, relies on `__NV_SATFINITE` behavior for out-of-range values and `cudaRoundNearest` rounding mode.

__host__ __device__ *__nv_fp4_e2m1::__nv_fp4_e2m1*(const int val)
Constructor from `int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp4_e2m1::__nv_fp4_e2m1*(const unsigned short int val)
Constructor from `unsigned short int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp4_e2m1::__nv_fp4_e2m1*(const long long int val)
Constructor from `long long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp4_e2m1::__nv_fp4_e2m1*(const short int val)
Constructor from `short int` data type.

__host__ __device__ *__nv_fp4_e2m1::__nv_fp4_e2m1*(const __nv_bfloat16 f)
Constructor from `__nv_bfloat16` data type, relies on `__NV_SATFINITE` behavior for out-of-range values and `cudaRoundNearest` rounding mode.

__host__ __device__ ***__nv_fp4_e2m1::__nv_fp4_e2m1*** **(const unsigned int val)**
Constructor from `unsigned int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ ***__nv_fp4_e2m1::__nv_fp4_e2m1*** **(const unsigned long long int val)**
Constructor from `unsigned long long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ ***__nv_fp4_e2m1::__nv_fp4_e2m1*** **(const __half f)**
Constructor from `__half` data type, relies on `__NV_SATFINITE` behavior for out-of-range values and `cudaRoundNearest` rounding mode.

__host__ __device__ ***__nv_fp4x2_e2m1::__nv_fp4x2_e2m1*** **(const double2 f)**
Constructor from `double2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ ***__nv_fp4x2_e2m1::__nv_fp4x2_e2m1*** **(const __nv_bfloat162 f)**
Constructor from `__nv_bfloat162` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ ***__nv_fp4x2_e2m1::__nv_fp4x2_e2m1*** **(const __half2 f)**
Constructor from `__half2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ ***__nv_fp4x2_e2m1::__nv_fp4x2_e2m1*** **(const float2 f)**
Constructor from `float2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ ***__nv_fp4x2_e2m1::__nv_fp4x2_e2m1*** **()**
Constructor by default.

__NV_SILENCE_DEPRECATION_END __host__ __device__ ***__nv_fp4x4_e2m1::__nv_fp4x4_e2m1*** **(const double4_16a f)**
Constructor from `double4_16a` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ ***__nv_fp4x4_e2m1::__nv_fp4x4_e2m1*** **()**
Constructor by default.

__host__ __device__ ***__nv_fp4x4_e2m1::__nv_fp4x4_e2m1*** **(const __nv_bfloat162 flo, const __nv_bfloat162 fhi)**
Constructor from a pair of `__nv_bfloat162` data type values, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ ***__nv_fp4x4_e2m1::__nv_fp4x4_e2m1*** **(const float4 f)**
Constructor from `float4` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ ***__nv_fp4x4_e2m1::__nv_fp4x4_e2m1*** **(const __half2 flo, const __half2 fhi)**
Constructor from a pair of `__half2` data type values, relies on `__NV_SATFINITE` behavior for out-of-range values.

__NV_SILENCE_DEPRECATION_BEGIN __host__ __device__ ***__nv_fp4x4_e2m1::__nv_fp4x4_e2m1*** **(const double4 f)**
Constructor from `double4` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ ***__nv_fp4x4_e2m1::__nv_fp4x4_e2m1*** **(const double4_32a f)**
Constructor from `double4_32a` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

Typedefs

__nv_fp4_storage_t
8-bit `unsigned integer` type abstraction used for `fp4` floating-point numbers storage.

__nv_fp4x2_storage_t
8-bit `unsigned integer` type abstraction used for storage of pairs of `fp4` floating-point numbers.

__nv_fp4x4_storage_t
16-bit `unsigned integer` type abstraction used for storage of tetrads of `fp4` floating-point numbers.

1.4.1. Enumerations

enum **`__nv_fp4_interpretation_t`**

Enumerates the possible interpretations of the 4-bit values when referring to them as `fp4` types.

Values:

enumerator **`__NV_E2M1`**

Stands for `fp4` numbers of `e2m1` kind.

1.4.2. Functions

__host__ __device__ *__nv_fp4x2_storage_t* **`__nv_cvt_bfloat16raw2_to_fp4x2`**(const *__nv_bfloat162_raw* x, const *__nv_fp4_interpretation_t* fp4_interpretation, const enum cudaRoundMode rounding)

Converts input vector of two `nv_bfloat16` precision numbers packed in *`__nv_bfloat162_raw`* `x` into a vector of two values of `fp4` type of the requested kind using specified rounding mode and saturating the out-of-range values.

Converts input vector `x` to a vector of two `fp4` values of the kind specified by `fp4_interpretation` parameter, using rounding mode specified by `rounding` parameter. Large out-of-range values saturate to MAXNORM of the same sign. NaN input values result in positive MAXNORM.

Returns

- The `__nv_fp4x2_storage_t` value holds the result of conversion.

__host__ __device__ *__nv_fp4_storage_t* **__nv_cvt_bfloat16raw_to_fp4**(const *__nv_bfloat16_raw* x, const *__nv_fp4_interpretation_t* fp4_interpretation, const enum cudaRoundMode rounding)

Converts input `nv_bfloat16` precision `x` to `fp4` type of the requested kind using specified rounding mode and saturating the out-of-range values.

Converts input `x` to `fp4` type of the kind specified by `fp4_interpretation` parameter, using rounding mode specified by `rounding` parameter. Large out-of-range values saturate to MAXNORM of the same sign. NaN input values result in positive MAXNORM.

Returns

- The `__nv_fp4_storage_t` value holds the result of conversion.

__host__ __device__ *__nv_fp4x2_storage_t* **__nv_cvt_double2_to_fp4x2**(const double2 x, const *__nv_fp4_interpretation_t* fp4_interpretation, const enum cudaRoundMode rounding)

Converts input vector of two `double` precision numbers packed in `double2 x` into a vector of two values of `fp4` type of the requested kind using specified rounding mode and saturating the out-of-range values.

Converts input vector `x` to a vector of two `fp4` values of the kind specified by `fp4_interpretation` parameter, using rounding mode specified by `rounding` parameter. Large out-of-range values saturate to MAXNORM of the same sign. NaN input values result in positive MAXNORM.

Returns

- The `__nv_fp4x2_storage_t` value holds the result of conversion.

__host__ __device__ *__nv_fp4_storage_t* **__nv_cvt_double_to_fp4**(const double x, const *__nv_fp4_interpretation_t* fp4_interpretation, const enum cudaRoundMode rounding)

Converts input `double` precision `x` to `fp4` type of the requested kind using specified rounding mode and saturating the out-of-range values.

Converts input `x` to `fp4` type of the kind specified by `fp4_interpretation` parameter, using rounding mode specified by `rounding` parameter. Large out-of-range values saturate to MAXNORM of the same sign. NaN input values result in positive MAXNORM.

Returns

- The `__nv_fp4_storage_t` value holds the result of conversion.

__host__ __device__ *__nv_fp4x2_storage_t* **__nv_cvt_float2_to_fp4x2**(const float2 x, const *__nv_fp4_interpretation_t* fp4_interpretation, const enum cudaRoundMode rounding)

Converts input vector of two `single` precision numbers packed in `float2` `x` into a vector of two values of `fp4` type of the requested kind using specified rounding mode and saturating the out-of-range values.

Converts input vector `x` to a vector of two `fp4` values of the kind specified by `fp4_interpretation` parameter, using rounding mode specified by `rounding` parameter. Large out-of-range values saturate to MAXNORM of the same sign. NaN input values result in positive MAXNORM.

Returns

- The `__nv_fp4x2_storage_t` value holds the result of conversion.

__host__ __device__ *__nv_fp4_storage_t* **__nv_cvt_float_to_fp4**(const float x, const *__nv_fp4_interpretation_t* fp4_interpretation, const enum cudaRoundMode rounding)

Converts input `single` precision `x` to `fp4` type of the requested kind using specified rounding mode and saturating the out-of-range values.

Converts input `x` to `fp4` type of the kind specified by `fp4_interpretation` parameter, using rounding mode specified by `rounding` parameter. Large out-of-range values saturate to MAXNORM of the same sign. NaN input values result in positive MAXNORM.

Returns

- The `__nv_fp4_storage_t` value holds the result of conversion.

__host__ __device__ *__half_raw* **__nv_cvt_fp4_to_halfraw**(const *__nv_fp4_storage_t* x, const *__nv_fp4_interpretation_t* fp4_interpretation)

Converts input `fp4` `x` of the specified kind to `half` precision.

Converts input `x` of `fp4` type of the kind specified by `fp4_interpretation` parameter to `half` precision.

Returns

- The *`__half_raw`* value holds the result of conversion.

__host__ __device__ *__half2_raw* **__nv_cvt_fp4x2_to_halfraw2**(const *__nv_fp4x2_storage_t* x, const *__nv_fp4_interpretation_t* fp4_interpretation)

Converts input vector of two `fp4` values of the specified kind to a vector of two `half` precision values packed in *`__half2_raw`* structure.

Converts input vector `x` of `fp4` type of the kind specified by `fp4_interpretation` parameter to a vector of two `half` precision values and returns as *`__half2_raw`* structure.

Returns

- The *__half2_raw* value holds the result of conversion.

__host__ __device__ *__nv_fp4x2_storage_t* **__nv_cvt_halfraw2_to_fp4x2**(const *__half2_raw* x, const *__nv_fp4_interpretation_t* fp4_interpretation, const enum cudaRoundMode rounding)

Converts input vector of two `half` precision numbers packed in *__half2_raw* x into a vector of two values of `fp4` type of the requested kind using specified rounding mode and saturating the out-of-range values.

Converts input vector x to a vector of two `fp4` values of the kind specified by `fp4_interpretation` parameter, using rounding mode specified by `rounding` parameter. Large out-of-range values saturate to MAXNORM of the same sign. NaN input values result in positive MAXNORM.

Returns

- The `__nv_fp4x2_storage_t` value holds the result of conversion.

__host__ __device__ *__nv_fp4_storage_t* **__nv_cvt_halfraw_to_fp4**(const *__half_raw* x, const *__nv_fp4_interpretation_t* fp4_interpretation, const enum cudaRoundMode rounding)

Converts input `half` precision x to `fp4` type of the requested kind using specified rounding mode and saturating the out-of-range values.

Converts input x to `fp4` type of the kind specified by `fp4_interpretation` parameter, using rounding mode specified by `rounding` parameter. Large out-of-range values saturate to MAXNORM of the same sign. NaN input values result in positive MAXNORM.

Returns

- The `__nv_fp4_storage_t` value holds the result of conversion.

1.4.3. Typedefs

typedef *__nv_fp8_storage_t* **__nv_fp4_storage_t**

8-bit `unsigned integer` type abstraction used for `fp4` floating-point numbers storage.

typedef *__nv_fp8_storage_t* **__nv_fp4x2_storage_t**

8-bit `unsigned integer` type abstraction used for storage of pairs of `fp4` floating-point numbers.

typedef *__nv_fp8x2_storage_t* **__nv_fp4x4_storage_t**

16-bit `unsigned integer` type abstraction used for storage of tetrads of `fp4` floating-point numbers.

Groups

C++ struct for handling fp4 data type of e2m1 kind.

C++ struct for handling vector type of four fp4 values of e2m1 kind.

C++ struct for handling vector type of two fp4 values of e2m1 kind.

FP4 Conversion and Data Movement

To use these functions, include the header file `cuda_fp4.h` in your program.

Chapter 2. FP6 Intrinsics

This section describes fp6 intrinsic functions.

To use these functions, include the header file `cuda_fp6.h` in your program.

The following macros are available to help users selectively enable/disable various definitions present in the header file:

- `__CUDA_NO_FP6_CONVERSIONS__` - If defined, this macro will prevent any use of the C++ type conversions (converting constructors and conversion operators) defined in the header.
- `__CUDA_NO_FP6_CONVERSION_OPERATORS__` - If defined, this macro will prevent any use of the C++ conversion operators from `fp6` to other types.

Note: Most of the operations defined here benefit from native HW support when compiled for specific GPU targets (e.g. devices of compute capability 10.0a), other targets use emulation path.

2.1. C++ struct for handling fp6 data type of e2m3 kind.

Structs

__nv_fp6_e2m3

__nv_fp6_e2m3 datatype

2.2. C++ struct for handling fp6 data type of e3m2 kind.

Structs

__nv_fp6_e3m2

__nv_fp6_e3m2 datatype

2.3. C++ struct for handling vector type of four fp6 values of e2m3 kind.

Structs

__nv_fp6x4_e2m3
__nv_fp6x4_e2m3 datatype

2.4. C++ struct for handling vector type of four fp6 values of e3m2 kind.

Structs

__nv_fp6x4_e3m2
__nv_fp6x4_e3m2 datatype

2.5. C++ struct for handling vector type of two fp6 values of e2m3 kind.

Structs

__nv_fp6x2_e2m3
__nv_fp6x2_e2m3 datatype

2.6. C++ struct for handling vector type of two fp6 values of e3m2 kind.

Structs

__nv_fp6x2_e3m2
__nv_fp6x2_e3m2 datatype

2.7. FP6 Conversion and Data Movement

To use these functions, include the header file `cuda_fp6.h` in your program.

Enumerations

__nv_fp6_interpretation_t
Enumerates the possible interpretations of the 8-bit values when referring to them as `fp6` types.

Functions

__host__ __device__ __nv_fp6x2_storage_t *__nv_cvt_bfloat16raw2_to_fp6x2*(const __nv_bfloat162_raw x, const __nv_fp6_interpretation_t fp6_interpretation, const enum cudaRoundMode rounding)
Converts input vector of two `nv_bfloat16` precision numbers packed in `__nv_bfloat162_raw` `x` into a vector of two values of `fp6` type of the requested kind using specified rounding mode and saturating the out-of-range values.

__host__ __device__ __nv_fp6_storage_t *__nv_cvt_bfloat16raw_to_fp6*(const __nv_bfloat16_raw x, const __nv_fp6_interpretation_t fp6_interpretation, const enum cudaRoundMode rounding)
Converts input `nv_bfloat16` precision `x` to `fp6` type of the requested kind using specified rounding mode and saturating the out-of-range values.

__host__ __device__ __nv_fp6x2_storage_t *__nv_cvt_double2_to_fp6x2*(const double2 x, const __nv_fp6_interpretation_t fp6_interpretation, const enum cudaRoundMode rounding)
Converts input vector of two `double` precision numbers packed in `double2` `x` into a vector of two values of `fp6` type of the requested kind using specified rounding mode and saturating the out-of-range values.

__host__ __device__ __nv_fp6_storage_t *__nv_cvt_double_to_fp6*(const double x, const __nv_fp6_interpretation_t fp6_interpretation, const enum cudaRoundMode rounding)
Converts input `double` precision `x` to `fp6` type of the requested kind using specified rounding mode and saturating the out-of-range values.

__host__ __device__ __nv_fp6x2_storage_t *__nv_cvt_float2_to_fp6x2*(const float2 x, const __nv_fp6_interpretation_t fp6_interpretation, const enum cudaRoundMode rounding)
Converts input vector of two `single` precision numbers packed in `float2` `x` into a vector of two values of `fp6` type of the requested kind using specified rounding mode and saturating the out-of-range values.

__host__ __device__ __nv_fp6_storage_t *__nv_cvt_float_to_fp6*(const float x, const __nv_fp6_interpretation_t fp6_interpretation, const enum cudaRoundMode rounding)
Converts input `single` precision `x` to `fp6` type of the requested kind using specified rounding mode and saturating the out-of-range values.

__host__ __device__ __half_raw *__nv_cvt_fp6_to_halfraw*(const __nv_fp6_storage_t x, const __nv_fp6_interpretation_t fp6_interpretation)
Converts input `fp6` `x` of the specified kind to `half` precision.

__host__ __device__ __half2_raw *__nv_cvt_fp6x2_to_halfraw2*(const __nv_fp6x2_storage_t x, const __nv_fp6_interpretation_t fp6_interpretation)
Converts input vector of two `fp6` values of the specified kind to a vector of two `half` precision values packed in `__half2_raw` structure.

__host__ __device__ __nv_fp6x2_storage_t *__nv_cvt_halfraw2_to_fp6x2*(const __half2_raw x, const __nv_fp6_interpretation_t fp6_interpretation, const enum cudaRoundMode rounding)
Converts input vector of two `half` precision numbers packed in `__half2_raw` x into a vector of two values of `fp6` type of the requested kind using specified rounding mode and saturating the out-of-range values.

__host__ __device__ __nv_fp6_storage_t *__nv_cvt_halfraw_to_fp6*(const __half_raw x, const __nv_fp6_interpretation_t fp6_interpretation, const enum cudaRoundMode rounding)
Converts input `half` precision `x` to `fp6` type of the requested kind using specified rounding mode and saturating the out-of-range values.

__host__ __device__ *__nv_fp6_e2m3::__nv_fp6_e2m3*()
Constructor by default.

__host__ __device__ *__nv_fp6_e2m3::__nv_fp6_e2m3*(const float f)
Constructor from `float` data type, relies on `__NV_SATFINITE` behavior for out-of-range values and `cudaRoundNearest` rounding mode.

__host__ __device__ *__nv_fp6_e2m3::__nv_fp6_e2m3*(const long long int val)
Constructor from `long long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp6_e2m3::__nv_fp6_e2m3*(const long int val)
Constructor from `long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp6_e2m3::__nv_fp6_e2m3*(const unsigned short int val)
Constructor from `unsigned short int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp6_e2m3::__nv_fp6_e2m3*(const double f)
Constructor from `double` data type, relies on `__NV_SATFINITE` behavior for out-of-range values and `cudaRoundNearest` rounding mode.

__host__ __device__ *__nv_fp6_e2m3::__nv_fp6_e2m3*(const unsigned long long int val)
Constructor from `unsigned long long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp6_e2m3::__nv_fp6_e2m3*(const short int val)
Constructor from `short int` data type.

__host__ __device__ *__nv_fp6_e2m3::__nv_fp6_e2m3*(const __nv_bfloat16 f)
Constructor from `__nv_bfloat16` data type, relies on `__NV_SATFINITE` behavior for out-of-range values and `cudaRoundNearest` rounding mode.

__host__ __device__ *__nv_fp6_e2m3::__nv_fp6_e2m3*(const int val)
Constructor from `int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp6_e2m3::__nv_fp6_e2m3*(const __half f)
Constructor from `__half` data type, relies on `__NV_SATFINITE` behavior for out-of-range values and `cudaRoundNearest` rounding mode.

__host__ __device__ *__nv_fp6_e2m3::__nv_fp6_e2m3*(const unsigned int val)
Constructor from `unsigned int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp6_e2m3::__nv_fp6_e2m3*(const unsigned long int val)
Constructor from `unsigned long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp6_e3m2::__nv_fp6_e3m2*(const long long int val)
Constructor from `long long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp6_e3m2::__nv_fp6_e3m2*(const __half f)
Constructor from `__half` data type, relies on `__NV_SATFINITE` behavior for out-of-range values and `cudaRoundNearest` rounding mode.

__host__ __device__ *__nv_fp6_e3m2::__nv_fp6_e3m2*(const int val)
Constructor from `int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp6_e3m2::__nv_fp6_e3m2*(const float f)
Constructor from `float` data type, relies on `__NV_SATFINITE` behavior for out-of-range values and `cudaRoundNearest` rounding mode.

__host__ __device__ *__nv_fp6_e3m2::__nv_fp6_e3m2*(const short int val)
Constructor from `short int` data type.

__host__ __device__ *__nv_fp6_e3m2::__nv_fp6_e3m2*(const long int val)
Constructor from `long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp6_e3m2::__nv_fp6_e3m2*(const unsigned long long int val)
Constructor from `unsigned long long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp6_e3m2::__nv_fp6_e3m2*(const unsigned short int val)
Constructor from `unsigned short int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp6_e3m2::__nv_fp6_e3m2*(const __nv_bfloat16 f)
Constructor from `__nv_bfloat16` data type, relies on `__NV_SATFINITE` behavior for out-of-range values and `cudaRoundNearest` rounding mode.

__host__ __device__ *__nv_fp6_e3m2::__nv_fp6_e3m2*(const double f)
Constructor from `double` data type, relies on `__NV_SATFINITE` behavior for out-of-range values and `cudaRoundNearest` rounding mode.

__host__ __device__ *__nv_fp6_e3m2::__nv_fp6_e3m2*(const unsigned int val)
Constructor from `unsigned int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp6_e3m2::__nv_fp6_e3m2*()
Constructor by default.

__host__ __device__ *__nv_fp6_e3m2::__nv_fp6_e3m2*(const unsigned long int val)
Constructor from `unsigned long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp6x2_e2m3::__nv_fp6x2_e2m3*(const __nv_bfloat162 f)
Constructor from `__nv_bfloat162` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp6x2_e2m3::__nv_fp6x2_e2m3*(const float2 f)
Constructor from `float2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp6x2_e2m3::__nv_fp6x2_e2m3*(const double2 f)
Constructor from `double2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp6x2_e2m3::__nv_fp6x2_e2m3*(const __half2 f)
: Constructor from __half2 data type, relies on __NV_SATFINITE behavior for out-of-range values.

__host__ __device__ *__nv_fp6x2_e2m3::__nv_fp6x2_e2m3*()
: Constructor by default.

__host__ __device__ *__nv_fp6x2_e3m2::__nv_fp6x2_e3m2*(const __half2 f)
: Constructor from __half2 data type, relies on __NV_SATFINITE behavior for out-of-range values.

__host__ __device__ *__nv_fp6x2_e3m2::__nv_fp6x2_e3m2*(const float2 f)
: Constructor from float2 data type, relies on __NV_SATFINITE behavior for out-of-range values.

__host__ __device__ *__nv_fp6x2_e3m2::__nv_fp6x2_e3m2*(const __nv_bfloat162 f)
: Constructor from __nv_bfloat162 data type, relies on __NV_SATFINITE behavior for out-of-range values.

__host__ __device__ *__nv_fp6x2_e3m2::__nv_fp6x2_e3m2*(const double2 f)
: Constructor from double2 data type, relies on __NV_SATFINITE behavior for out-of-range values.

__host__ __device__ *__nv_fp6x2_e3m2::__nv_fp6x2_e3m2*()
: Constructor by default.

__host__ __device__ *__nv_fp6x4_e2m3::__nv_fp6x4_e2m3*(const double4_32a f)
: Constructor from double4_32a vector data type, relies on __NV_SATFINITE behavior for out-of-range values.

__host__ __device__ *__nv_fp6x4_e2m3::__nv_fp6x4_e2m3*(const __half2 flo, const __half2 fhi)
: Constructor from a pair of __half2 data type values, relies on __NV_SATFINITE behavior for out-of-range values.

__host__ __device__ *__nv_fp6x4_e2m3::__nv_fp6x4_e2m3*(const float4 f)
: Constructor from float4 vector data type, relies on __NV_SATFINITE behavior for out-of-range values.

__NV_SILENCE_DEPRECATION_END __host__ __device__ *__nv_fp6x4_e2m3::__nv_fp6x4_e2m3*(const double4_16a f)
: Constructor from double4_16a vector data type, relies on __NV_SATFINITE behavior for out-of-range values.

__NV_SILENCE_DEPRECATION_BEGIN __host__ __device__ *__nv_fp6x4_e2m3::__nv_fp6x4_e2m3*(const double4 f)
: Constructor from double4 vector data type, relies on __NV_SATFINITE behavior for out-of-range values.

__host__ __device__ *__nv_fp6x4_e2m3::__nv_fp6x4_e2m3*(const __nv_bfloat162 flo, const __nv_bfloat162 fhi)
: Constructor from a pair of __nv_bfloat162 data type values, relies on __NV_SATFINITE behavior for out-of-range values.

__host__ __device__ *__nv_fp6x4_e2m3::__nv_fp6x4_e2m3*()
: Constructor by default.

__NV_SILENCE_DEPRECATION_END __host__ __device__ *__nv_fp6x4_e3m2::__nv_fp6x4_e3m2*(const double4_16a f)
: Constructor from double4_16a vector data type, relies on __NV_SATFINITE behavior for out-of-range values.

__host__ __device__ *__nv_fp6x4_e3m2::__nv_fp6x4_e3m2*()
Constructor by default.

__host__ __device__ *__nv_fp6x4_e3m2::__nv_fp6x4_e3m2*(const double4_32a f)
Constructor from `double4_32a` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp6x4_e3m2::__nv_fp6x4_e3m2*(const __half2 flo, const __half2 fhi)
Constructor from a pair of `__half2` data type values, relies on `__NV_SATFINITE` behavior for out-of-range values.

__NV_SILENCE_DEPRECATION_BEGIN __host__ __device__ *__nv_fp6x4_e3m2::__nv_fp6x4_e3m2*(const double4 f)
Constructor from `double4` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp6x4_e3m2::__nv_fp6x4_e3m2*(const float4 f)
Constructor from `float4` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp6x4_e3m2::__nv_fp6x4_e3m2*(const __nv_bfloat162 flo, const __nv_bfloat162 fhi)
Constructor from a pair of `__nv_bfloat162` data type values, relies on `__NV_SATFINITE` behavior for out-of-range values.

Typedefs

__nv_fp6_storage_t
8-bit `unsigned integer` type abstraction used for `fp6` floating-point numbers storage.

__nv_fp6x2_storage_t
16-bit `unsigned integer` type abstraction used for storage of pairs of `fp6` floating-point numbers.

__nv_fp6x4_storage_t
32-bit `unsigned integer` type abstraction used for storage of tetrads of `fp6` floating-point numbers.

2.7.1. Enumerations

enum **`__nv_fp6_interpretation_t`**

Enumerates the possible interpretations of the 8-bit values when referring to them as `fp6` types.

Values:

enumerator **`__NV_E2M3`**

Stands for `fp6` numbers of `e2m3` kind.

enumerator **`__NV_E3M2`**

Stands for `fp6` numbers of `e3m2` kind.

2.7.2. Functions

__host__ __device__ *__nv_fp6x2_storage_t* **__nv_cvt_bfloat16raw2_to_fp6x2**(const *__nv_bfloat162_raw* x, const *__nv_fp6_interpretation_t* fp6_interpretation, const enum cudaRoundMode rounding)

Converts input vector of two `nv_bfloat16` precision numbers packed in *__nv_bfloat162_raw* `x` into a vector of two values of `fp6` type of the requested kind using specified rounding mode and saturating the out-of-range values.

Converts input vector `x` to a vector of two `fp6` values of the kind specified by `fp6_interpretation` parameter, using rounding mode specified by `rounding` parameter. Large out-of-range values saturate to MAXNORM of the same sign. NaN input values result in positive MAXNORM.

Returns

- The `__nv_fp6x2_storage_t` value holds the result of conversion.

__host__ __device__ *__nv_fp6_storage_t* **__nv_cvt_bfloat16raw_to_fp6**(const *__nv_bfloat16_raw* x, const *__nv_fp6_interpretation_t* fp6_interpretation, const enum cudaRoundMode rounding)

Converts input `nv_bfloat16` precision `x` to `fp6` type of the requested kind using specified rounding mode and saturating the out-of-range values.

Converts input `x` to `fp6` type of the kind specified by `fp6_interpretation` parameter, using rounding mode specified by `rounding` parameter. Large out-of-range values saturate to MAXNORM of the same sign. NaN input values result in positive MAXNORM.

Returns

- The `__nv_fp6_storage_t` value holds the result of conversion.

__host__ __device__ *__nv_fp6x2_storage_t* **__nv_cvt_double2_to_fp6x2**(const double2 x, const *__nv_fp6_interpretation_t* fp6_interpretation, const enum cudaRoundMode rounding)

Converts input vector of two `double` precision numbers packed in `double2 x` into a vector of two values of `fp6` type of the requested kind using specified rounding mode and saturating the out-of-range values.

Converts input vector `x` to a vector of two `fp6` values of the kind specified by `fp6_interpretation` parameter, using rounding mode specified by `rounding` parameter. Large out-of-range values saturate to MAXNORM of the same sign. NaN input values result in positive MAXNORM.

Returns

- The `__nv_fp6x2_storage_t` value holds the result of conversion.

__host__ __device__ *__nv_fp6_storage_t* **`__nv_cvt_double_to_fp6`**(const double x, const *__nv_fp6_interpretation_t* fp6_interpretation, const enum cudaRoundMode rounding)

Converts input `double` precision `x` to `fp6` type of the requested kind using specified rounding mode and saturating the out-of-range values.

Converts input `x` to `fp6` type of the kind specified by `fp6_interpretation` parameter, using rounding mode specified by `rounding` parameter. Large out-of-range values saturate to MAXNORM of the same sign. NaN input values result in positive MAXNORM.

Returns

- The `__nv_fp6_storage_t` value holds the result of conversion.

__host__ __device__ *__nv_fp6x2_storage_t* **`__nv_cvt_float2_to_fp6x2`**(const float2 x, const *__nv_fp6_interpretation_t* fp6_interpretation, const enum cudaRoundMode rounding)

Converts input vector of two `single` precision numbers packed in `float2 x` into a vector of two values of `fp6` type of the requested kind using specified rounding mode and saturating the out-of-range values.

Converts input vector `x` to a vector of two `fp6` values of the kind specified by `fp6_interpretation` parameter, using rounding mode specified by `rounding` parameter. Large out-of-range values saturate to MAXNORM of the same sign. NaN input values result in positive MAXNORM.

Returns

- The `__nv_fp6x2_storage_t` value holds the result of conversion.

__host__ __device__ *__nv_fp6_storage_t* **`__nv_cvt_float_to_fp6`**(const float x, const *__nv_fp6_interpretation_t* fp6_interpretation, const enum cudaRoundMode rounding)

Converts input `single` precision `x` to `fp6` type of the requested kind using specified rounding mode and saturating the out-of-range values.

Converts input `x` to `fp6` type of the kind specified by `fp6_interpretation` parameter, using rounding mode specified by `rounding` parameter. Large out-of-range values saturate to MAXNORM of the same sign. NaN input values result in positive MAXNORM.

Returns

- The `__nv_fp6_storage_t` value holds the result of conversion.

__host__ __device__ *__half_raw* **__nv_cvt_fp6_to_halfraw**(const *__nv_fp6_storage_t* x, const *__nv_fp6_interpretation_t* fp6_interpretation)

Converts input `fp6` x of the specified kind to `half` precision.

Converts input x of `fp6` type of the kind specified by `fp6_interpretation` parameter to `half` precision.

Returns

- The *__half_raw* value holds the result of conversion.

__host__ __device__ *__half2_raw* **__nv_cvt_fp6x2_to_halfraw2**(const *__nv_fp6x2_storage_t* x, const *__nv_fp6_interpretation_t* fp6_interpretation)

Converts input vector of two `fp6` values of the specified kind to a vector of two `half` precision values packed in *__half2_raw* structure.

Converts input vector x of `fp6` type of the kind specified by `fp6_interpretation` parameter to a vector of two `half` precision values and returns as *__half2_raw* structure.

Returns

- The *__half2_raw* value holds the result of conversion.

__host__ __device__ *__nv_fp6x2_storage_t* **__nv_cvt_halfraw2_to_fp6x2**(const *__half2_raw* x, const *__nv_fp6_interpretation_t* fp6_interpretation, const enum cudaRoundMode rounding)

Converts input vector of two `half` precision numbers packed in *__half2_raw* x into a vector of two values of `fp6` type of the requested kind using specified rounding mode and saturating the out-of-range values.

Converts input vector x to a vector of two `fp6` values of the kind specified by `fp6_interpretation` parameter, using rounding mode specified by `rounding` parameter. Large out-of-range values saturate to MAXNORM of the same sign. NaN input values result in positive MAXNORM.

Returns

- The `__nv_fp6x2_storage_t` value holds the result of conversion.

__host__ __device__ *__nv_fp6_storage_t* **__nv_cvt_halfraw_to_fp6**(const *__half_raw* x, const *__nv_fp6_interpretation_t* fp6_interpretation, const enum cudaRoundMode rounding)

Converts input `half` precision x to `fp6` type of the requested kind using specified rounding mode and saturating the out-of-range values.

Converts input x to `fp6` type of the kind specified by `fp6_interpretation` parameter, using rounding mode specified by `rounding` parameter. Large out-of-range values saturate to MAXNORM of the same sign. NaN input values result in positive MAXNORM.

Returns

- The `__nv_fp6_storage_t` value holds the result of conversion.

2.7.3. Typedefs

typedef *__nv_fp8_storage_t* **`__nv_fp6_storage_t`**

8-bit `unsigned integer` type abstraction used for `fp6` floating-point numbers storage.

typedef *__nv_fp8x2_storage_t* **`__nv_fp6x2_storage_t`**

16-bit `unsigned integer` type abstraction used for storage of pairs of `fp6` floating-point numbers.

typedef *__nv_fp8x4_storage_t* **`__nv_fp6x4_storage_t`**

32-bit `unsigned integer` type abstraction used for storage of tetrads of `fp6` floating-point numbers.

Groups

C++ struct for handling fp6 data type of e2m3 kind.

C++ struct for handling fp6 data type of e3m2 kind.

C++ struct for handling vector type of four fp6 values of e2m3 kind.

C++ struct for handling vector type of four fp6 values of e3m2 kind.

C++ struct for handling vector type of two fp6 values of e2m3 kind.

C++ struct for handling vector type of two fp6 values of e3m2 kind.

FP6 Conversion and Data Movement

To use these functions, include the header file `cuda_fp6.h` in your program.

Chapter 3. FP8 Intrinsics

This section describes fp8 intrinsic functions.

To use these functions, include the header file `cuda_fp8.h` in your program. The following macros are available to help users selectively enable/disable various definitions present in the header file:

- `__CUDA_NO_FP8_CONVERSIONS__` - If defined, this macro will prevent any use of the C++ type conversions (converting constructors and conversion operators) defined in the header.
- `__CUDA_NO_FP8_CONVERSION_OPERATORS__` - If defined, this macro will prevent any use of the C++ conversion operators from `fp8` to other types.

3.1. C++ struct for handling fp8 data type of e4m3 kind.

Structs

__nv_fp8_e4m3
    __nv_fp8_e4m3 datatype

3.2. C++ struct for handling fp8 data type of e5m2 kind.

Structs

__nv_fp8_e5m2
    __nv_fp8_e5m2 datatype

3.3. C++ struct for handling vector type of four fp8 values of e4m3 kind.

Structs

__nv_fp8x4_e4m3
__nv_fp8x4_e4m3 datatype

3.4. C++ struct for handling vector type of four fp8 values of e5m2 kind.

Structs

__nv_fp8x4_e5m2
__nv_fp8x4_e5m2 datatype

3.5. C++ struct for handling vector type of four scale factors of e8m0 kind.

Structs

__nv_fp8x4_e8m0
__nv_fp8x4_e8m0 datatype

3.6. C++ struct for handling vector type of two fp8 values of e4m3 kind.

Structs

__nv_fp8x2_e4m3
__nv_fp8x2_e4m3 datatype

3.7. C++ struct for handling vector type of two fp8 values of e5m2 kind.

Structs

__nv_fp8x2_e5m2

__nv_fp8x2_e5m2 datatype

3.8. C++ struct for handling vector type of two scale factors of e8m0 kind.

Structs

__nv_fp8x2_e8m0

__nv_fp8x2_e8m0 datatype

3.9. FP8 Conversion and Data Movement

To use these functions, include the header file `cuda_fp8.h` in your program.

Enumerations

__nv_fp8_interpretation_t

Enumerates the possible interpretations of the 8-bit values when referring to them as `fp8` types.

__nv_saturation_t

Enumerates the modes applicable when performing a narrowing conversion to `fp8` destination types.

Functions

__host__ __device__ __nv_fp8x2_storage_t *__nv_cvt_bfloat162raw_to_e8m0x2*(const __nv_bfloat162_raw x, const __nv_saturation_t saturate, const enum cudaRoundMode rounding)

Converts a pair of `bfloat16` values into a pair of scaling factors of `e8m0` kind.

__host__ __device__ __nv_fp8x2_storage_t *__nv_cvt_bfloat16raw2_to_fp8x2*(const __nv_bfloat162_raw x, const __nv_saturation_t saturate, const __nv_fp8_interpretation_t fp8_interpretation)

Converts input vector of two `nv_bfloat16` precision numbers packed in `__nv_bfloat162_raw x` into a vector of two values of `fp8` type of the requested kind using round-to-nearest-even rounding and requested saturation mode.

__host__ __device__ __nv_fp8_storage_t *__nv_cvt_bfloat16raw_to_e8m0*(const __nv_bfloat16_raw x, const __nv_saturation_t saturate, const enum cudaRoundMode rounding)
Converts input `bfloat16` input into a scaling factor of `e8m0` kind.

__host__ __device__ __nv_fp8_storage_t *__nv_cvt_bfloat16raw_to_fp8*(const __nv_bfloat16_raw x, const __nv_saturation_t saturate, const __nv_fp8_interpretation_t fp8_interpretation)
Converts input `nv_bfloat16` precision `x` to `fp8` type of the requested kind using round-to-nearest-even rounding and requested saturation mode.

__host__ __device__ __nv_fp8x2_storage_t *__nv_cvt_double2_to_e8m0x2*(const double2 x, const __nv_saturation_t saturate, const enum cudaRoundMode rounding)
Converts a pair of `double` values into a pair of scaling factors of `e8m0` kind.

__host__ __device__ __nv_fp8x2_storage_t *__nv_cvt_double2_to_fp8x2*(const double2 x, const __nv_saturation_t saturate, const __nv_fp8_interpretation_t fp8_interpretation)
Converts input vector of two `double` precision numbers packed in `double2` `x` into a vector of two values of `fp8` type of the requested kind using round-to-nearest-even rounding and requested saturation mode.

__host__ __device__ __nv_fp8_storage_t *__nv_cvt_double_to_e8m0*(const double x, const __nv_saturation_t saturate, const enum cudaRoundMode rounding)
Converts input `double` value into a scaling factor of `e8m0` kind.

__host__ __device__ __nv_fp8_storage_t *__nv_cvt_double_to_fp8*(const double x, const __nv_saturation_t saturate, const __nv_fp8_interpretation_t fp8_interpretation)
Converts input `double` precision `x` to `fp8` type of the requested kind using round-to-nearest-even rounding and requested saturation mode.

__host__ __device__ __nv_bfloat16_raw *__nv_cvt_e8m0_to_bf16raw*(const __nv_fp8_storage_t x)
Converts input scaling factor value of `e8m0` kind into `bfloat16` .

__host__ __device__ __nv_bfloat162_raw *__nv_cvt_e8m0x2_to_bf162raw*(const __nv_fp8x2_storage_t x)
Converts input pair of scaling factors of `e8m0` kind into a pair of `bfloat16` values.

__host__ __device__ __nv_fp8x2_storage_t *__nv_cvt_float2_to_e8m0x2*(const float2 x, const __nv_saturation_t saturate, const enum cudaRoundMode rounding)
Converts a pair of `float` values into a pair of scaling factors of `e8m0` kind.

__host__ __device__ __nv_fp8x2_storage_t *__nv_cvt_float2_to_fp8x2*(const float2 x, const __nv_saturation_t saturate, const __nv_fp8_interpretation_t fp8_interpretation)
Converts input vector of two `single` precision numbers packed in `float2` `x` into a vector of two values of `fp8` type of the requested kind using round-to-nearest-even rounding and requested saturation mode.

__host__ __device__ __nv_fp8_storage_t *__nv_cvt_float_to_e8m0*(const float x, const __nv_saturation_t saturate, const enum cudaRoundMode rounding)
Converts input `float` value into a scaling factor of `e8m0` kind.

__host__ __device__ __nv_fp8_storage_t *__nv_cvt_float_to_fp8*(const float x, const __nv_saturation_t saturate, const __nv_fp8_interpretation_t fp8_interpretation)
Converts input `single` precision `x` to `fp8` type of the requested kind using round-to-nearest-even rounding and requested saturation mode.

__host__ __device__ __half_raw *__nv_cvt_fp8_to_halfraw*(const __nv_fp8_storage_t x, const __nv_fp8_interpretation_t fp8_interpretation)
Converts input `fp8` `x` of the specified kind to `half` precision.

__host__ __device__ __half2_raw *__nv_cvt_fp8x2_to_halfraw2*(const __nv_fp8x2_storage_t x, const __nv_fp8_interpretation_t fp8_interpretation)

Converts input vector of two fp8 values of the specified kind to a vector of two half precision values packed in __half2_raw structure.

__host__ __device__ __nv_fp8x2_storage_t *__nv_cvt_halfraw2_to_fp8x2*(const __half2_raw x, const __nv_saturation_t saturate, const __nv_fp8_interpretation_t fp8_interpretation)
Converts input vector of two half precision numbers packed in __half2_raw x into a vector of two values of fp8 type of the requested kind using round-to-nearest-even rounding and requested saturation mode.

__host__ __device__ __nv_fp8_storage_t *__nv_cvt_halfraw_to_fp8*(const __half_raw x, const __nv_saturation_t saturate, const __nv_fp8_interpretation_t fp8_interpretation)
Converts input half precision x to fp8 type of the requested kind using round-to-nearest-even rounding and requested saturation mode.

__host__ __device__ *__nv_fp8_e4m3::__nv_fp8_e4m3*(const int val)
Constructor from int data type, relies on __NV_SATFINITE behavior for out-of-range values.

__host__ __device__ *__nv_fp8_e4m3::__nv_fp8_e4m3*(const unsigned long long int val)
Constructor from unsigned long long int data type, relies on __NV_SATFINITE behavior for out-of-range values.

__host__ __device__ *__nv_fp8_e4m3::__nv_fp8_e4m3*(const __nv_bfloat16 f)
Constructor from __nv_bfloat16 data type, relies on __NV_SATFINITE behavior for out-of-range values.

__host__ __device__ *__nv_fp8_e4m3::__nv_fp8_e4m3*(const long int val)
Constructor from long int data type, relies on __NV_SATFINITE behavior for out-of-range values.

__host__ __device__ *__nv_fp8_e4m3::__nv_fp8_e4m3*(const long long int val)
Constructor from long long int data type, relies on __NV_SATFINITE behavior for out-of-range values.

***__nv_fp8_e4m3::__nv_fp8_e4m3*()=default**
Constructor by default.

__host__ __device__ *__nv_fp8_e4m3::__nv_fp8_e4m3*(const unsigned short int val)
Constructor from unsigned short int data type, relies on __NV_SATFINITE behavior for out-of-range values.

__host__ __device__ *__nv_fp8_e4m3::__nv_fp8_e4m3*(const float f)
Constructor from float data type, relies on __NV_SATFINITE behavior for out-of-range values.

__host__ __device__ *__nv_fp8_e4m3::__nv_fp8_e4m3*(const __half f)
Constructor from __half data type, relies on __NV_SATFINITE behavior for out-of-range values.

__host__ __device__ *__nv_fp8_e4m3::__nv_fp8_e4m3*(const unsigned long int val)
Constructor from unsigned long int data type, relies on __NV_SATFINITE behavior for out-of-range values.

__host__ __device__ *__nv_fp8_e4m3::__nv_fp8_e4m3*(const double f)
Constructor from double data type, relies on __NV_SATFINITE behavior for out-of-range values.

__host__ __device__ *__nv_fp8_e4m3::__nv_fp8_e4m3*(const short int val)
Constructor from short int data type, relies on __NV_SATFINITE behavior for out-of-range values.

__host__ __device__ *__nv_fp8_e4m3::__nv_fp8_e4m3*(const unsigned int val)
Constructor from unsigned int data type, relies on __NV_SATFINITE behavior for out-of-range values.

__host__ __device__ *__nv_fp8_e4m3::operator __half*() const
Conversion operator to `__half` data type.

__host__ __device__ *__nv_fp8_e4m3::operator __nv_bfloat16*() const
Conversion operator to `__nv_bfloat16` data type.

__host__ __device__ *__nv_fp8_e4m3::operator bool*() const
Conversion operator to `bool` data type.

__host__ __device__ *__nv_fp8_e4m3::operator char*() const
Conversion operator to an implementation defined `char` data type.

__host__ __device__ *__nv_fp8_e4m3::operator double*() const
Conversion operator to `double` data type.

__host__ __device__ *__nv_fp8_e4m3::operator float*() const
Conversion operator to `float` data type.

__host__ __device__ *__nv_fp8_e4m3::operator int*() const
Conversion operator to `int` data type.

__host__ __device__ *__nv_fp8_e4m3::operator long int*() const
Conversion operator to `long int` data type.

__host__ __device__ *__nv_fp8_e4m3::operator long long int*() const
Conversion operator to `long long int` data type.

__host__ __device__ *__nv_fp8_e4m3::operator short int*() const
Conversion operator to `short int` data type.

__host__ __device__ *__nv_fp8_e4m3::operator signed char*() const
Conversion operator to `signed char` data type.

__host__ __device__ *__nv_fp8_e4m3::operator unsigned char*() const
Conversion operator to `unsigned char` data type.

__host__ __device__ *__nv_fp8_e4m3::operator unsigned int*() const
Conversion operator to `unsigned int` data type.

__host__ __device__ *__nv_fp8_e4m3::operator unsigned long int*() const
Conversion operator to `unsigned long int` data type.

__host__ __device__ *__nv_fp8_e4m3::operator unsigned long long int*() const
Conversion operator to `unsigned long long int` data type.

__host__ __device__ *__nv_fp8_e4m3::operator unsigned short int*() const
Conversion operator to `unsigned short int` data type.

__host__ __device__ *__nv_fp8_e5m2::__nv_fp8_e5m2*(const __half f)
Constructor from `__half` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp8_e5m2::__nv_fp8_e5m2*(const long long int val)
Constructor from `long long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp8_e5m2::__nv_fp8_e5m2*(const unsigned int val)
Constructor from `unsigned int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp8_e5m2::__nv_fp8_e5m2*(const float f)
Constructor from `float` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp8_e5m2::__nv_fp8_e5m2*(const unsigned short int val)
Constructor from `unsigned short int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

***__nv_fp8_e5m2::__nv_fp8_e5m2*()=default**
Constructor by default.

__host__ __device__ *__nv_fp8_e5m2::__nv_fp8_e5m2*(const int val)
Constructor from `int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp8_e5m2::__nv_fp8_e5m2*(const long int val)
Constructor from `long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp8_e5m2::__nv_fp8_e5m2*(const double f)
Constructor from `double` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp8_e5m2::__nv_fp8_e5m2*(const unsigned long int val)
Constructor from `unsigned long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp8_e5m2::__nv_fp8_e5m2*(const short int val)
Constructor from `short int` data type.

__host__ __device__ *__nv_fp8_e5m2::__nv_fp8_e5m2*(const __nv_bfloat16 f)
Constructor from `__nv_bfloat16` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp8_e5m2::__nv_fp8_e5m2*(const unsigned long long int val)
Constructor from `unsigned long long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp8_e5m2::operator __half*() const
Conversion operator to `__half` data type.

__host__ __device__ *__nv_fp8_e5m2::operator __nv_bfloat16*() const
Conversion operator to `__nv_bfloat16` data type.

__host__ __device__ *__nv_fp8_e5m2::operator bool*() const
Conversion operator to `bool` data type.

__host__ __device__ *__nv_fp8_e5m2::operator char*() const
Conversion operator to an implementation defined `char` data type.

__host__ __device__ *__nv_fp8_e5m2::operator double*() const
Conversion operator to `double` data type.

__host__ __device__ *__nv_fp8_e5m2::operator float*() const
Conversion operator to `float` data type.

__host__ __device__ *__nv_fp8_e5m2::operator int*() const
Conversion operator to `int` data type.

__host__ __device__ *__nv_fp8_e5m2::operator long int*() const
Conversion operator to `long int` data type.

__host__ __device__ *__nv_fp8_e5m2::operator long long int*() const
Conversion operator to `long long int` data type.

__host__ __device__ *__nv_fp8_e5m2::operator short int*() const
Conversion operator to `short int` data type.

__host__ __device__ *__nv_fp8_e5m2::operator signed char*() const
Conversion operator to signed char data type.

__host__ __device__ *__nv_fp8_e5m2::operator unsigned char*() const
Conversion operator to unsigned char data type.

__host__ __device__ *__nv_fp8_e5m2::operator unsigned int*() const
Conversion operator to unsigned int data type.

__host__ __device__ *__nv_fp8_e5m2::operator unsigned long int*() const
Conversion operator to unsigned long int data type.

__host__ __device__ *__nv_fp8_e5m2::operator unsigned long long int*() const
Conversion operator to unsigned long long int data type.

__host__ __device__ *__nv_fp8_e5m2::operator unsigned short int*() const
Conversion operator to unsigned short int data type.

__host__ __device__ *__nv_fp8_e8m0::__nv_fp8_e8m0*(const long int val)
Constructor from long int data type, relies on cudaRoundPosInf rounding.

__host__ __device__ *__nv_fp8_e8m0::__nv_fp8_e8m0*(const int val)
Constructor from int data type, relies on cudaRoundPosInf rounding.

***__nv_fp8_e8m0::__nv_fp8_e8m0*()=default**
Constructor by default.

__host__ __device__ *__nv_fp8_e8m0::__nv_fp8_e8m0*(const unsigned int val)
Constructor from unsigned int data type, relies on cudaRoundPosInf rounding.

__host__ __device__ *__nv_fp8_e8m0::__nv_fp8_e8m0*(const float f)
Constructor from float data type, relies on __NV_SATFINITE behavior behavior for large input values and cudaRoundPosInf for rounding.

__host__ __device__ *__nv_fp8_e8m0::__nv_fp8_e8m0*(const unsigned long long int val)
Constructor from unsigned long long int data type, relies on cudaRoundPosInf rounding.

__host__ __device__ *__nv_fp8_e8m0::__nv_fp8_e8m0*(const double f)
Constructor from double data type, relies on __NV_SATFINITE behavior for large input values and cudaRoundPosInf for rounding.

__host__ __device__ *__nv_fp8_e8m0::__nv_fp8_e8m0*(const __half f)
Constructor from __half data type, relies on __NV_SATFINITE behavior for large input values and cudaRoundPosInf for rounding.

__host__ __device__ *__nv_fp8_e8m0::__nv_fp8_e8m0*(const __nv_bfloat16 f)
Constructor from __nv_bfloat16 data type, relies on __NV_SATFINITE behavior for large input values and cudaRoundPosInf for rounding.

__host__ __device__ *__nv_fp8_e8m0::__nv_fp8_e8m0*(const unsigned long int val)
Constructor from unsigned long int data type, relies on cudaRoundPosInf rounding.

__host__ __device__ *__nv_fp8_e8m0::__nv_fp8_e8m0*(const unsigned short int val)
Constructor from unsigned short int data type, relies on cudaRoundPosInf rounding.

__host__ __device__ *__nv_fp8_e8m0::__nv_fp8_e8m0*(const long long int val)
Constructor from long long int data type, relies on cudaRoundPosInf rounding.

__host__ __device__ *__nv_fp8_e8m0::__nv_fp8_e8m0*(const short int val)
Constructor from short int data type, relies on cudaRoundPosInf rounding.

__host__ __device__ *__nv_fp8_e8m0::operator __half*() const
Conversion operator to __half data type.

__host__ __device__ *__nv_fp8_e8m0::operator __nv_bfloat16*() const
Conversion operator to `__nv_bfloat16` data type.

__host__ __device__ *__nv_fp8_e8m0::operator bool*() const
Conversion operator to `bool` data type.

__host__ __device__ *__nv_fp8_e8m0::operator char*() const
Conversion operator to an implementation defined `char` data type.

__host__ __device__ *__nv_fp8_e8m0::operator double*() const
Conversion operator to `double` data type.

__host__ __device__ *__nv_fp8_e8m0::operator float*() const
Conversion operator to `float` data type.

__host__ __device__ *__nv_fp8_e8m0::operator int*() const
Conversion operator to `int` data type.

__host__ __device__ *__nv_fp8_e8m0::operator long int*() const
Conversion operator to `long int` data type.

__host__ __device__ *__nv_fp8_e8m0::operator long long int*() const
Conversion operator to `long long int` data type.

__host__ __device__ *__nv_fp8_e8m0::operator short int*() const
Conversion operator to `short int` data type.

__host__ __device__ *__nv_fp8_e8m0::operator signed char*() const
Conversion operator to `signed char` data type.

__host__ __device__ *__nv_fp8_e8m0::operator unsigned char*() const
Conversion operator to `unsigned char` data type.

__host__ __device__ *__nv_fp8_e8m0::operator unsigned int*() const
Conversion operator to `unsigned int` data type.

__host__ __device__ *__nv_fp8_e8m0::operator unsigned long int*() const
Conversion operator to `unsigned long int` data type.

__host__ __device__ *__nv_fp8_e8m0::operator unsigned long long int*() const
Conversion operator to `unsigned long long int` data type.

__host__ __device__ *__nv_fp8_e8m0::operator unsigned short int*() const
Conversion operator to `unsigned short int` data type.

***__nv_fp8x2_e4m3::__nv_fp8x2_e4m3*()=default**
Constructor by default.

__host__ __device__ *__nv_fp8x2_e4m3::__nv_fp8x2_e4m3*(const __nv_bfloat162 f)
Constructor from `__nv_bfloat162` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp8x2_e4m3::__nv_fp8x2_e4m3*(const double2 f)
Constructor from `double2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp8x2_e4m3::__nv_fp8x2_e4m3*(const __half2 f)
Constructor from `__half2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp8x2_e4m3::__nv_fp8x2_e4m3*(const float2 f)
Constructor from `float2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp8x2_e4m3::operator __half2*() const
Conversion operator to `__half2` data type.

__host__ __device__ *__nv_fp8x2_e4m3::operator float2*() const
Conversion operator to `float2` data type.

__host__ __device__ *__nv_fp8x2_e5m2::__nv_fp8x2_e5m2*(const __nv_bfloat162 f)
Constructor from `__nv_bfloat162` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp8x2_e5m2::__nv_fp8x2_e5m2*(const double2 f)
Constructor from `double2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp8x2_e5m2::__nv_fp8x2_e5m2*(const __half2 f)
Constructor from `__half2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

***__nv_fp8x2_e5m2::__nv_fp8x2_e5m2*()=default**
Constructor by default.

__host__ __device__ *__nv_fp8x2_e5m2::__nv_fp8x2_e5m2*(const float2 f)
Constructor from `float2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp8x2_e5m2::operator __half2*() const
Conversion operator to `__half2` data type.

__host__ __device__ *__nv_fp8x2_e5m2::operator float2*() const
Conversion operator to `float2` data type.

__host__ __device__ *__nv_fp8x2_e8m0::__nv_fp8x2_e8m0*(const __half2 f)
Constructor from `__half2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp8x2_e8m0::__nv_fp8x2_e8m0*(const float2 f)
Constructor from `float2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp8x2_e8m0::__nv_fp8x2_e8m0*(const __nv_bfloat162 f)
Constructor from `__nv_bfloat162` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp8x2_e8m0::__nv_fp8x2_e8m0*(const double2 f)
Constructor from `double2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

***__nv_fp8x2_e8m0::__nv_fp8x2_e8m0*()=default**
Constructor by default.

__host__ __device__ *__nv_fp8x2_e8m0::operator __half2*() const
Conversion operator to `__half2` data type.

__host__ __device__ *__nv_fp8x2_e8m0::operator __nv_bfloat162*() const
Conversion operator to `__nv_bfloat162` data type.

__host__ __device__ *__nv_fp8x2_e8m0::operator float2*() const
Conversion operator to `float2` data type.

__NV_SILENCE_DEPRECATION_END __host__ __device__ *__nv_fp8x4_e4m3::__nv_fp8x4_e4m3*(const double4_16a f)
Constructor from `double4_16a` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp8x4_e4m3::__nv_fp8x4_e4m3*(const __nv_bfloat162 flo, const __nv_bfloat162 fhi)
Constructor from a pair of `__nv_bfloat162` data type values, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp8x4_e4m3::__nv_fp8x4_e4m3*(const double4_32a f)
Constructor from `double4_32a` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__NV_SILENCE_DEPRECATION_BEGIN __host__ __device__ *__nv_fp8x4_e4m3::__nv_fp8x4_e4m3*(const double4 f)
Constructor from `double4` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

***__nv_fp8x4_e4m3::__nv_fp8x4_e4m3*()=default**
Constructor by default.

__host__ __device__ *__nv_fp8x4_e4m3::__nv_fp8x4_e4m3*(const float4 f)
Constructor from `float4` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp8x4_e4m3::__nv_fp8x4_e4m3*(const __half2 flo, const __half2 fhi)
Constructor from a pair of `__half2` data type values, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp8x4_e4m3::operator float4*() const
Conversion operator to `float4` vector data type.

***__nv_fp8x4_e5m2::__nv_fp8x4_e5m2*()=default**
Constructor by default.

__host__ __device__ *__nv_fp8x4_e5m2::__nv_fp8x4_e5m2*(const double4_32a f)
Constructor from `double4_32a` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__NV_SILENCE_DEPRECATION_END __host__ __device__ *__nv_fp8x4_e5m2::__nv_fp8x4_e5m2*(const double4_16a f)
Constructor from `double4_16a` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__NV_SILENCE_DEPRECATION_BEGIN __host__ __device__ *__nv_fp8x4_e5m2::__nv_fp8x4_e5m2*(const double4 f)
Constructor from `double4` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp8x4_e5m2::__nv_fp8x4_e5m2*(const float4 f)
Constructor from `float4` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp8x4_e5m2::__nv_fp8x4_e5m2*(const __nv_bfloat162 flo, const __nv_bfloat162 fhi)
Constructor from a pair of `__nv_bfloat162` data type values, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp8x4_e5m2::__nv_fp8x4_e5m2*(const __half2 flo, const __half2 fhi)
Constructor from a pair of `__half2` data type values, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp8x4_e5m2::operator float4*() const
Conversion operator to `float4` vector data type.

__NV_SILENCE_DEPRECATION_BEGIN __host__ __device__ *__nv_fp8x4_e8m0::__nv_fp8x4_e8m0*(const double4 f)
Constructor from `double4` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__NV_SILENCE_DEPRECATION_END __host__ __device__ *__nv_fp8x4_e8m0::__nv_fp8x4_e8m0*(const double4_16a f)
Constructor from `double4_16a` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp8x4_e8m0::__nv_fp8x4_e8m0*(const __half2 flo, const __half2 fhi)
Constructor from a pair of `__half2` data type values, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp8x4_e8m0::__nv_fp8x4_e8m0*(const float4 f)
Constructor from `float4` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp8x4_e8m0::__nv_fp8x4_e8m0*(const __nv_bfloat162 flo, const __nv_bfloat162 fhi)
Constructor from a pair of `__nv_bfloat162` data type values, relies on `__NV_SATFINITE` behavior for out-of-range values.

***__nv_fp8x4_e8m0::__nv_fp8x4_e8m0*()=default**
Constructor by default.

__host__ __device__ *__nv_fp8x4_e8m0::__nv_fp8x4_e8m0*(const double4_32a f)
Constructor from `double4_32a` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ *__nv_fp8x4_e8m0::operator float4*() const
Conversion operator to `float4` vector data type.

Typedefs

__nv_fp8_storage_t
8-bit `unsigned integer` type abstraction used for `fp8` floating-point numbers storage.

__nv_fp8x2_storage_t
16-bit `unsigned integer` type abstraction used for storage of pairs of `fp8` floating-point numbers.

__nv_fp8x4_storage_t
32-bit `unsigned integer` type abstraction used for storage of tetrads of `fp8` floating-point numbers.

3.9.1. Enumerations

enum **`__nv_fp8_interpretation_t`**

Enumerates the possible interpretations of the 8-bit values when referring to them as `fp8` types.

Values:

enumerator **`__NV_E4M3`**

Stands for `fp8` numbers of `e4m3` kind.

enumerator **__NV_E5M2**

Stands for `fp8` numbers of `e5m2` kind.

enum **__nv_saturation_t**

Enumerates the modes applicable when performing a narrowing conversion to `fp8` destination types.

Values:

enumerator **__NV_NOSAT**

Means no saturation to finite is performed when conversion results in rounding values outside the range of destination type.

NOTE: for fp8 type of e4m3 kind, the results that are larger than the maximum representable finite number of the target format become NaN.

enumerator **__NV_SATFINITE**

Means input larger than the maximum representable finite number MAXNORM of the target format round to the MAXNORM of the same sign as input.

3.9.2. Functions

__host__ __device__ *__nv_fp8x2_storage_t* **__nv_cvt_bfloat162raw_to_e8m0x2**(const *__nv_bfloat162_raw* x, const *__nv_saturation_t* saturate, const enum cudaRoundMode rounding)

Converts a pair of `bfloat16` values into a pair of scaling factors of `e8m0` kind.

See also:

__nv_cvt_bfloat16raw_to_e8m0() for details of conversion.

Returns

- The `__nv_fp8x2_storage_t` value holds the result of conversion.

__host__ __device__ *__nv_fp8x2_storage_t* **__nv_cvt_bfloat16raw2_to_fp8x2**(const *__nv_bfloat162_raw* x, const *__nv_saturation_t* saturate, const *__nv_fp8_interpretation_t* fp8_interpretation)

Converts input vector of two `nv_bfloat16` precision numbers packed in *`__nv_bfloat162_raw`* x into a vector of two values of fp8 type of the requested kind using round-to-nearest-even rounding and requested saturation mode.

Converts input vector x to a vector of two fp8 values of the kind specified by fp8_interpretation parameter, using round-to-nearest-even rounding and saturation mode specified by saturate parameter.

Returns

- The __nv_fp8x2_storage_t value holds the result of conversion.

__host__ __device__ *__nv_fp8_storage_t* **__nv_cvt_bfloat16raw_to_e8m0**(const *__nv_bfloat16_raw* x, const *__nv_saturation_t* saturate, const enum cudaRoundMode rounding)

Converts input bfloat16 input into a scaling factor of e8m0 kind.

Input number's absolute value is rounded to the closest power of two in the direction specified via rounding parameter. Rounded results that are smaller than the smallest representable target format number 2^-127 are then clipped to 2^-127. Results that are larger than the largest representable target format number 2^127 are either clipped to 2^127 if saturate equals to __NV_SATFINITE, or convert to NaN otherwise. NaN inputs convert into NaN output, encoded as 0xFF in the target format.

Returns

- The __nv_fp8_storage_t value holds the result of conversion.

__host__ __device__ *__nv_fp8_storage_t* **__nv_cvt_bfloat16raw_to_fp8**(const *__nv_bfloat16_raw* x, const *__nv_saturation_t* saturate, const *__nv_fp8_interpretation_t* fp8_interpretation)

Converts input nv_bfloat16 precision x to fp8 type of the requested kind using round-to-nearest-even rounding and requested saturation mode.

Converts input x to fp8 type of the kind specified by fp8_interpretation parameter, using round-to-nearest-even rounding and saturation mode specified by saturate parameter.

Returns

- The __nv_fp8_storage_t value holds the result of conversion.

__host__ __device__ *__nv_fp8x2_storage_t* **__nv_cvt_double2_to_e8m0x2**(const double2 x, const *__nv_saturation_t* saturate, const enum cudaRoundMode rounding)

Converts a pair of double values into a pair of scaling factors of e8m0 kind.

See also:

__nv_cvt_bfloat16raw_to_e8m0() for details of conversion.

Returns

- The `__nv_fp8x2_storage_t` value holds the result of conversion.

__host__ __device__ *__nv_fp8x2_storage_t* **`__nv_cvt_double2_to_fp8x2`**(const double2 x, const *__nv_saturation_t* saturate, const *__nv_fp8_interpretation_t* fp8_interpretation)

Converts input vector of two `double` precision numbers packed in `double2` x into a vector of two values of `fp8` type of the requested kind using round-to-nearest-even rounding and requested saturation mode.

Converts input vector `x` to a vector of two `fp8` values of the kind specified by `fp8_interpretation` parameter, using round-to-nearest-even rounding and saturation mode specified by `saturate` parameter.

Returns

- The `__nv_fp8x2_storage_t` value holds the result of conversion.

__host__ __device__ *__nv_fp8_storage_t* **`__nv_cvt_double_to_e8m0`**(const double x, const *__nv_saturation_t* saturate, const enum cudaRoundMode rounding)

Converts input `double` value into a scaling factor of `e8m0` kind.

See also:

__nv_cvt_bfloat16raw_to_e8m0() for details of conversion.

Returns

- The `__nv_fp8_storage_t` value holds the result of conversion.

__host__ __device__ *__nv_fp8_storage_t* **`__nv_cvt_double_to_fp8`**(const double x, const *__nv_saturation_t* saturate, const *__nv_fp8_interpretation_t* fp8_interpretation)

Converts input `double` precision `x` to `fp8` type of the requested kind using round-to-nearest-even rounding and requested saturation mode.

Converts input `x` to `fp8` type of the kind specified by `fp8_interpretation` parameter, using round-to-nearest-even rounding and saturation mode specified by `saturate` parameter.

Returns

- The `__nv_fp8_storage_t` value holds the result of conversion.

__host__ __device__ *__nv_bfloat16_raw* **__nv_cvt_e8m0_to_bf16raw**(const *__nv_fp8_storage_t* x)

Converts input scaling factor value of `e8m0` kind into `bfloat16`.

Input scales are exact powers of two or a `NaN` value, also representable in the target format.

Returns

- The *__nv_bfloat16_raw* value holds the result of conversion.

__host__ __device__ *__nv_bfloat162_raw* **__nv_cvt_e8m0x2_to_bf162raw**(const *__nv_fp8x2_storage_t* x)

Converts input pair of scaling factors of `e8m0` kind into a pair of `bfloat16` values.

Returns

- The *__nv_bfloat162_raw* value holds the result of conversion.

__host__ __device__ *__nv_fp8x2_storage_t* **__nv_cvt_float2_to_e8m0x2**(const float2 x, const *__nv_saturation_t* saturate, const enum cudaRoundMode rounding)

Converts a pair of `float` values into a pair of scaling factors of `e8m0` kind.

See also:

__nv_cvt_bfloat16raw_to_e8m0() for details of conversion.

Returns

- The `__nv_fp8x2_storage_t` value holds the result of conversion.

__host__ __device__ *__nv_fp8x2_storage_t* **__nv_cvt_float2_to_fp8x2**(const float2 x, const *__nv_saturation_t* saturate, const *__nv_fp8_interpretation_t* fp8_interpretation)

Converts input vector of two `single` precision numbers packed in `float2` x into a vector of two values of `fp8` type of the requested kind using round-to-nearest-even rounding and requested saturation mode.

Converts input vector `x` to a vector of two `fp8` values of the kind specified by `fp8_interpretation` parameter, using round-to-nearest-even rounding and saturation mode specified by `saturate` parameter.

Returns

- The `__nv_fp8x2_storage_t` value holds the result of conversion.

__host__ __device__ *__nv_fp8_storage_t* **__nv_cvt_float_to_e8m0**(const float x, const *__nv_saturation_t* saturate, const enum cudaRoundMode rounding)

Converts input `float` value into a scaling factor of `e8m0` kind.

See also:

__nv_cvt_bfloat16raw_to_e8m0() for details of conversion.

Returns

- The `__nv_fp8_storage_t` value holds the result of conversion.

__host__ __device__ *__nv_fp8_storage_t* **`__nv_cvt_float_to_fp8`**(const float x, const *__nv_saturation_t* saturate, const *__nv_fp8_interpretation_t* fp8_interpretation)

Converts input `single` precision `x` to `fp8` type of the requested kind using round-to-nearest-even rounding and requested saturation mode.

Converts input `x` to `fp8` type of the kind specified by `fp8_interpretation` parameter, using round-to-nearest-even rounding and saturation mode specified by `saturate` parameter.

Returns

- The `__nv_fp8_storage_t` value holds the result of conversion.

__host__ __device__ *__half_raw* **`__nv_cvt_fp8_to_halfraw`**(const *__nv_fp8_storage_t* x, const *__nv_fp8_interpretation_t* fp8_interpretation)

Converts input `fp8` `x` of the specified kind to `half` precision.

Converts input `x` of `fp8` type of the kind specified by `fp8_interpretation` parameter to `half` precision.

Returns

- The *`__half_raw`* value holds the result of conversion.

__host__ __device__ *__half2_raw* **`__nv_cvt_fp8x2_to_halfraw2`**(const *__nv_fp8x2_storage_t* x, const *__nv_fp8_interpretation_t* fp8_interpretation)

Converts input vector of two `fp8` values of the specified kind to a vector of two `half` precision values packed in *`__half2_raw`* structure.

Converts input vector `x` of `fp8` type of the kind specified by `fp8_interpretation` parameter to a vector of two `half` precision values and returns as *`__half2_raw`* structure.

Returns

- The *`__half2_raw`* value holds the result of conversion.

__host__ __device__ *__nv_fp8x2_storage_t* **`__nv_cvt_halfraw2_to_fp8x2`**(const *__half2_raw* x, const *__nv_saturation_t* saturate, const *__nv_fp8_interpretation_t* fp8_interpretation)

Converts input vector of two `half` precision numbers packed in *__half2_raw* x into a vector of two values of `fp8` type of the requested kind using round-to-nearest-even rounding and requested saturation mode.

Converts input vector x to a vector of two `fp8` values of the kind specified by `fp8_interpretation` parameter, using round-to-nearest-even rounding and saturation mode specified by `saturate` parameter.

Returns

- The `__nv_fp8x2_storage_t` value holds the result of conversion.

__host__ __device__ *__nv_fp8_storage_t* **__nv_cvt_halfraw_to_fp8**(const *__half_raw* x, const *__nv_saturation_t* saturate, const *__nv_fp8_interpretation_t* fp8_interpretation)

Converts input `half` precision x to `fp8` type of the requested kind using round-to-nearest-even rounding and requested saturation mode.

Converts input x to `fp8` type of the kind specified by `fp8_interpretation` parameter, using round-to-nearest-even rounding and saturation mode specified by `saturate` parameter.

Returns

- The `__nv_fp8_storage_t` value holds the result of conversion.

3.9.3. Typedefs

typedef unsigned char **__nv_fp8_storage_t**

8-bit `unsigned integer` type abstraction used for `fp8` floating-point numbers storage.

typedef unsigned short int **__nv_fp8x2_storage_t**

16-bit `unsigned integer` type abstraction used for storage of pairs of `fp8` floating-point numbers.

typedef unsigned int **__nv_fp8x4_storage_t**

32-bit `unsigned integer` type abstraction used for storage of tetrads of `fp8` floating-point numbers.

Groups

C++ struct for handling fp8 data type of e4m3 kind.

C++ struct for handling fp8 data type of e5m2 kind.

C++ struct for handling vector type of four fp8 values of e4m3 kind.

C++ struct for handling vector type of four fp8 values of e5m2 kind.

C++ struct for handling vector type of four scale factors of e8m0 kind.

C++ struct for handling vector type of two fp8 values of e4m3 kind.

C++ struct for handling vector type of two fp8 values of e5m2 kind.

C++ struct for handling vector type of two scale factors of e8m0 kind.

FP8 Conversion and Data Movement

To use these functions, include the header file `cuda_fp8.h` in your program.

Chapter 4. Half Precision Intrinsics

This section describes half precision intrinsic functions.

To use these functions, include the header file `cuda_fp16.h` in your program. All of the functions defined here are available in device code. Some of the functions are also available to host compilers, please refer to respective functions' documentation for details.

NOTE: Aggressive floating-point optimizations performed by host or device compilers may affect numeric behavior of the functions implemented in this header.

The following macros are available to help users selectively enable/disable various definitions present in the header file:

- `CUDA_NO_HALF` - If defined, this macro will prevent the definition of additional type aliases in the global namespace, helping to avoid potential conflicts with symbols defined in the user program.
- `__CUDA_NO_HALF_CONVERSIONS__` - If defined, this macro will prevent the use of the C++ type conversions (converting constructors and conversion operators) that are common for built-in floating-point types, but may be undesirable for `half` which is essentially a user-defined type.
- `__CUDA_NO_HALF_OPERATORS__` and `__CUDA_NO_HALF2_OPERATORS__` - If defined, these macros will prevent the inadvertent use of usual arithmetic and comparison operators. This enforces the storage-only type semantics and prevents C++ style computations on `half` and `half2` types.

4.1. Half Arithmetic Constants

To use these constants, include the header file `cuda_fp16.h` in your program.

Macros

CUDART_INF_FP16
Defines floating-point positive infinity value for the `half` data type.

CUDART_MAX_NORMAL_FP16
Defines a maximum representable value for the `half` data type.

CUDART_MIN_DENORM_FP16
Defines a minimum representable (denormalized) value for the `half` data type.

CUDART_NAN_FP16
Defines canonical NaN value for the `half` data type.

CUDART_NEG_ZERO_FP16
Defines a negative zero value for the `half` data type.

CUDART_ONE_FP16
Defines a value of 1.0 for the `half` data type.

CUDART_ZERO_FP16
Defines a positive zero value for the `half` data type.

4.1.1. Macros

CUDART_INF_FP16 *__ushort_as_half*((unsigned short)0x7C00U)

Defines floating-point positive infinity value for the `half` data type.

CUDART_MAX_NORMAL_FP16 *__ushort_as_half*((unsigned short)0x7BFFU)

Defines a maximum representable value for the `half` data type.

CUDART_MIN_DENORM_FP16 *__ushort_as_half*((unsigned short)0x0001U)

Defines a minimum representable (denormalized) value for the `half` data type.

CUDART_NAN_FP16 *__ushort_as_half*((unsigned short)0x7FFFU)

Defines canonical NaN value for the `half` data type.

CUDART_NEG_ZERO_FP16 *__ushort_as_half*((unsigned short)0x8000U)

Defines a negative zero value for the `half` data type.

CUDART_ONE_FP16 *__ushort_as_half*((unsigned short)0x3C00U)

Defines a value of 1.0 for the `half` data type.

CUDART_ZERO_FP16 *__ushort_as_half*((unsigned short)0x0000U)

Defines a positive zero value for the `half` data type.

4.2. Half Arithmetic Functions

To use these functions, include the header file `cuda_fp16.h` in your program.

Functions

__host__ __device__ __half *__habs*(const __half a)
Calculates the absolute value of input `half` number and returns the result.

__host__ __device__ __half *__hadd*(const __half a, const __half b)
Performs `half` addition in round-to-nearest-even mode.

__host__ __device__ __half *__hadd_rn*(const __half a, const __half b)
Performs `half` addition in round-to-nearest-even mode.

__host__ __device__ __half *__hadd_sat*(const __half a, const __half b)
Performs `half` addition in round-to-nearest-even mode, with saturation to [0.0, 1.0].

__host__ __device__ __half *__hdiv*(const __half a, const __half b)
Performs `half` division in round-to-nearest-even mode.

__device__ __half *__hfma*(const __half a, const __half b, const __half c)
Performs `half` fused multiply-add in round-to-nearest-even mode.

__device__ __half *__hfma_relu*(const __half a, const __half b, const __half c)
Performs `half` fused multiply-add in round-to-nearest-even mode with relu saturation.

__device__ __half *__hfma_sat*(const __half a, const __half b, const __half c)
Performs `half` fused multiply-add in round-to-nearest-even mode, with saturation to [0.0, 1.0].

__host__ __device__ __half *__hmul*(const __half a, const __half b)
Performs `half` multiplication in round-to-nearest-even mode.

__host__ __device__ __half *__hmul_rn*(const __half a, const __half b)
Performs `half` multiplication in round-to-nearest-even mode.

__host__ __device__ __half *__hmul_sat*(const __half a, const __half b)
Performs `half` multiplication in round-to-nearest-even mode, with saturation to [0.0, 1.0].

__host__ __device__ __half *__hneg*(const __half a)
Negates input `half` number and returns the result.

__host__ __device__ __half *__hsub*(const __half a, const __half b)
Performs `half` subtraction in round-to-nearest-even mode.

__host__ __device__ __half *__hsub_rn*(const __half a, const __half b)
Performs `half` subtraction in round-to-nearest-even mode.

__host__ __device__ __half *__hsub_sat*(const __half a, const __half b)
Performs `half` subtraction in round-to-nearest-even mode, with saturation to [0.0, 1.0].

__device__ __half *atomicAdd*(__half *const address, const __half val)
Adds `val` to the value stored at `address` in global or shared memory, and writes this value back to `address` .

__host__ __device__ __half *operator(const __half &lh, const __half &rh)**
Performs `half` multiplication operation.

__host__ __device__ __half & *operator*=*(__half &lh, const __half &rh)
Performs `half` compound assignment with multiplication operation.

__host__ __device__ __half *operator+*(const __half &h)
Implements `half` unary plus operator, returns input value.

__host__ __device__ __half *operator+*(const __half &lh, const __half &rh)
Performs `half` addition operation.

__host__ __device__ __half & *operator++*(__half &h)
Performs `half` prefix increment operation.

__host__ __device__ __half *operator++*(__half &h, const int ignored)
Performs `half` postfix increment operation.

__host__ __device__ __half & *operator+=*(__half &lh, const __half &rh)
Performs `half` compound assignment with addition operation.

__host__ __device__ __half *operator-*(const __half &lh, const __half &rh)
Performs `half` subtraction operation.

__host__ __device__ __half *operator-*(const __half &h)
Implements `half` unary minus operator.

__host__ __device__ __half *operator--*(__half &h, const int ignored)
Performs `half` postfix decrement operation.

__host__ __device__ __half & *operator--*(__half &h)
Performs `half` prefix decrement operation.

__host__ __device__ __half & *operator-=*(__half &lh, const __half &rh)
Performs `half` compound assignment with subtraction operation.

__host__ __device__ __half *operator/*(const __half &lh, const __half &rh)
Performs `half` division operation.

__host__ __device__ __half & *operator/=*(__half &lh, const __half &rh)
Performs `half` compound assignment with division operation.

4.2.1. Functions

__host__ __device__ *__half* **__habs**(const *__half* a)

Calculates the absolute value of input `half` number and returns the result.

Calculates the absolute value of input `half` number and returns the result.

Parameters

a – **[in]** - half. Is only being read.

Returns

half

- The absolute value of a.
- __habs (± 0) returns +0.
- __habs $(\pm\infty)$ returns $+\infty$.
- __habs(NaN) returns NaN.

__host__ __device__ *__half* **__hadd**(const *__half* a, const *__half* b)

Performs `half` addition in round-to-nearest-even mode.

Performs `half` addition of inputs a and b, in round-to-nearest-even mode.

Parameters

- **a** – **[in]** - half. Is only being read.
- **b** – **[in]** - half. Is only being read.

Returns

half

- The sum of a and b.

__host__ __device__ *__half* **__hadd_rn**(const *__half* a, const *__half* b)

Performs `half` addition in round-to-nearest-even mode.

Performs `half` addition of inputs a and b, in round-to-nearest-even mode. Prevents floating-point contractions of mul+add into fma.

Parameters

- **a** – **[in]** - half. Is only being read.
- **b** – **[in]** - half. Is only being read.

Returns

half

- The sum of a and b.

__host__ __device__ *__half* **__hadd_sat**(const *__half* a, const *__half* b)

Performs `half` addition in round-to-nearest-even mode, with saturation to [0.0, 1.0].

Performs `half` add of inputs a and b, in round-to-nearest-even mode, and clamps the result to range [0.0, 1.0]. NaN results are flushed to +0.0.

Parameters

- **a** – **[in]** - half. Is only being read.
- **b** – **[in]** - half. Is only being read.

Returns

half

- The sum of a and b, with respect to saturation.

__host__ __device__ *__half* **__hdiv**(const *__half* a, const *__half* b)

Performs `half` division in round-to-nearest-even mode.

Divides `half` input a by input b in round-to-nearest-even mode.

Parameters

- **a** – **[in]** - half. Is only being read.
- **b** – **[in]** - half. Is only being read.

Returns

half

- The result of dividing a by b.

__device__ *__half* **__hfma**(const *__half* a, const *__half* b, const *__half* c)

Performs `half` fused multiply-add in round-to-nearest-even mode.

Performs `half` multiply on inputs a and b, then performs a `half` add of the result with c, rounding the result once in round-to-nearest-even mode.

Parameters

- **a** – **[in]** - half. Is only being read.
- **b** – **[in]** - half. Is only being read.
- **c** – **[in]** - half. Is only being read.

Returns
half

- The result of fused multiply-add operation on a, b, and c.

__device__ *__half* **__hfma_relu**(const *__half* a, const *__half* b, const *__half* c)

Performs `half` fused multiply-add in round-to-nearest-even mode with relu saturation.

Performs `half` multiply on inputs a and b, then performs a `half` add of the result with c, rounding the result once in round-to-nearest-even mode. Then negative result is clamped to 0. NaN result is converted to canonical NaN.

Parameters

- **a** – **[in]** - half. Is only being read.
- **b** – **[in]** - half. Is only being read.
- **c** – **[in]** - half. Is only being read.

Returns
half

- The result of fused multiply-add operation on a, b, and c with relu saturation.

__device__ *__half* **__hfma_sat**(const *__half* a, const *__half* b, const *__half* c)

Performs `half` fused multiply-add in round-to-nearest-even mode, with saturation to [0.0, 1.0].

Performs `half` multiply on inputs a and b, then performs a `half` add of the result with c, rounding the result once in round-to-nearest-even mode, and clamps the result to range [0.0, 1.0]. NaN results are flushed to +0.0.

Parameters

- **a** – **[in]** - half. Is only being read.
- **b** – **[in]** - half. Is only being read.
- **c** – **[in]** - half. Is only being read.

Returns
half

- The result of fused multiply-add operation on a, b, and c, with respect to saturation.

__host__ __device__ *__half* **__hmul**(const *__half* a, const *__half* b)

Performs `half` multiplication in round-to-nearest-even mode.

Performs `half` multiplication of inputs a and b, in round-to-nearest-even mode.

Parameters

- **a** – **[in]** - half. Is only being read.
- **b** – **[in]** - half. Is only being read.

Returns
half

- The result of multiplying a and b.

__host__ __device__ *__half* **__hmul_rn**(const *__half* a, const *__half* b)

Performs `half` multiplication in round-to-nearest-even mode.

Performs `half` multiplication of inputs a and b, in round-to-nearest-even mode. Prevents floating-point contractions of mul+add or sub into fma.

Parameters

- **a** – **[in]** - half. Is only being read.
- **b** – **[in]** - half. Is only being read.

Returns
half

- The result of multiplying a and b.

__host__ __device__ *__half* **__hmul_sat**(const *__half* a, const *__half* b)

Performs `half` multiplication in round-to-nearest-even mode, with saturation to [0.0, 1.0].

Performs `half` multiplication of inputs a and b, in round-to-nearest-even mode, and clamps the result to range [0.0, 1.0]. NaN results are flushed to +0.0.

Parameters

- **a** – **[in]** - half. Is only being read.
- **b** – **[in]** - half. Is only being read.

Returns
half

- The result of multiplying a and b, with respect to saturation.

__host__ __device__ *__half* **__hneg**(const *__half* a)

Negates input `half` number and returns the result.

Negates input `half` number and returns the result.

Parameters
a – **[in]** - half. Is only being read.

Returns
half

- Negated input a.
- __hneg (± 0) returns ∓ 0.
- __hneg $(\pm\infty)$ returns $\mp\infty$.
- __hneg(NaN) returns NaN.

__host__ __device__ *__half* **__hsub**(const *__half* a, const *__half* b)

Performs `half` subtraction in round-to-nearest-even mode.

Subtracts `half` input b from input a in round-to-nearest-even mode.

Parameters

- **a** – **[in]** - half. Is only being read.
- **b** – **[in]** - half. Is only being read.

Returns
half

- The result of subtracting b from a.

__host__ __device__ *__half* **__hsub_rn**(const *__half* a, const *__half* b)

Performs `half` subtraction in round-to-nearest-even mode.

Subtracts `half` input b from input a in round-to-nearest-even mode. Prevents floating-point contractions of mul+sub into fma.

Parameters

- **a** – **[in]** - half. Is only being read.
- **b** – **[in]** - half. Is only being read.

Returns
half

- The result of subtracting b from a.

__host__ __device__ *__half* **__hsub_sat**(const *__half* a, const *__half* b)

Performs `half` subtraction in round-to-nearest-even mode, with saturation to [0.0, 1.0].

Subtracts `half` input b from input a in round-to-nearest-even mode, and clamps the result to range [0.0, 1.0]. NaN results are flushed to +0.0.

Parameters

- **a** – **[in]** - half. Is only being read.
- **b** – **[in]** - half. Is only being read.

Returns
half

- The result of subtraction of b from a, with respect to saturation.

__device__ *__half* **atomicAdd**(*__half* *const address, const *__half* val)

Adds `val` to the value stored at `address` in global or shared memory, and writes this value back to `address`.

This operation is performed in one atomic operation.

The location of `address` must be in global or shared memory. This operation has undefined behavior otherwise. This operation is only supported by devices of compute capability 7.x and higher.

Note: For more details about this function, see the Atomic Functions section in the CUDA C++ Programming Guide.

Parameters

- **address** – **[in]** - half*. An address in global or shared memory.
- **val** – **[in]** - half. The value to be added.

Returns
half

- The old value read from address.

__host__ __device__ *__half* **operator***(const *__half* &lh, const *__half* &rh)
Performs half multiplication operation.

See also:

__hmul(__half, __half)

__host__ __device__ *__half* &**operator*=**(*__half* &lh, const *__half* &rh)
Performs half compound assignment with multiplication operation.

See also:

__hmul(__half, __half)

__host__ __device__ *__half* **operator+**(const *__half* &h)
Implements half unary plus operator, returns input value.

__host__ __device__ *__half* **operator+**(const *__half* &lh, const *__half* &rh)
Performs half addition operation.

See also:

__hadd(__half, __half)

__host__ __device__ *__half* &**operator++**(*__half* &h)
Performs half prefix increment operation.

See also:

__hadd(__half, __half)

__host__ __device__ *__half* **operator++**(*__half* &h, const int ignored)
Performs half postfix increment operation.

See also:

__hadd(__half, __half)

__host__ __device__ *__half* &**operator+=**(*__half* &lh, const *__half* &rh)
Performs half compound assignment with addition operation.

See also:

__hadd(__half, __half)

__host__ __device__ *__half* **operator-**(const *__half* &lh, const *__half* &rh)

Performs `half` subtraction operation.

See also:

__hsub(__half, __half)

__host__ __device__ *__half* **operator-**(const *__half* &h)

Implements `half` unary minus operator.

See also:

__hneg(__half)

__host__ __device__ *__half* **operator--**(*__half* &h, const int ignored)

Performs `half` postfix decrement operation.

See also:

__hsub(__half, __half)

__host__ __device__ *__half* &**operator--**(*__half* &h)

Performs `half` prefix decrement operation.

See also:

__hsub(__half, __half)

__host__ __device__ *__half* &**operator-=**(*__half* &lh, const *__half* &rh)

Performs `half` compound assignment with subtraction operation.

See also:

__hsub(__half, __half)

__host__ __device__ *__half* **operator/**(const *__half* &lh, const *__half* &rh)

Performs `half` division operation.

See also:

__hdiv(__half, __half)

__host__ __device__ *__half* &**operator/=**(*__half* &lh, const *__half* &rh)

Performs `half` compound assignment with division operation.

See also:

__hdiv(__half, __half)

4.3. Half Comparison Functions

To use these functions, include the header file `cuda_fp16.h` in your program.

Functions

__host__ __device__ bool *__heq*(const __half a, const __half b)
Performs `half` if-equal comparison.

__host__ __device__ bool *__hequ*(const __half a, const __half b)
Performs `half` unordered if-equal comparison.

__host__ __device__ bool *__hge*(const __half a, const __half b)
Performs `half` greater-equal comparison.

__host__ __device__ bool *__hgeu*(const __half a, const __half b)
Performs `half` unordered greater-equal comparison.

__host__ __device__ bool *__hgt*(const __half a, const __half b)
Performs `half` greater-than comparison.

__host__ __device__ bool *__hgtu*(const __half a, const __half b)
Performs `half` unordered greater-than comparison.

__host__ __device__ int *__hisinf*(const __half a)
Checks if the input `half` number is infinite.

__host__ __device__ bool *__hisnan*(const __half a)
Determine whether `half` argument is a NaN.

__host__ __device__ bool *__hle*(const __half a, const __half b)
Performs `half` less-equal comparison.

__host__ __device__ bool *__hleu*(const __half a, const __half b)
Performs `half` unordered less-equal comparison.

__host__ __device__ bool *__hlt*(const __half a, const __half b)
Performs `half` less-than comparison.

__host__ __device__ bool *__hltu*(const __half a, const __half b)
Performs `half` unordered less-than comparison.

__host__ __device__ __half *__hmax*(const __half a, const __half b)
Calculates `half` maximum of two input values.

__host__ __device__ __half *__hmax_nan*(const __half a, const __half b)
Calculates `half` maximum of two input values, NaNs pass through.

__host__ __device__ __half *__hmin*(const __half a, const __half b)
Calculates `half` minimum of two input values.

__host__ __device__ __half *__hmin_nan*(const __half a, const __half b)
Calculates `half` minimum of two input values, NaNs pass through.

__host__ __device__ bool *__hne*(const __half a, const __half b)
Performs `half` not-equal comparison.

__host__ __device__ bool *__hneu*(const __half a, const __half b)
Performs `half` unordered not-equal comparison.

__host__ __device__ bool *operator!=*(const __half &lh, const __half &rh)
Performs `half` unordered compare not-equal operation.

__host__ __device__ bool *operator<*(const __half &lh, const __half &rh)
Performs `half` ordered less-than compare operation.

__host__ __device__ bool *operator<=*(const __half &lh, const __half &rh)
Performs `half` ordered less-or-equal compare operation.

__host__ __device__ bool *operator==*(const __half &lh, const __half &rh)
Performs `half` ordered compare equal operation.

__host__ __device__ bool *operator>*(const __half &lh, const __half &rh)
Performs `half` ordered greater-than compare operation.

__host__ __device__ bool *operator>=*(const __half &lh, const __half &rh)
Performs `half` ordered greater-or-equal compare operation.

4.3.1. Functions

__host__ __device__ bool **__heq**(const *__half* a, const *__half* b)
Performs `half` if-equal comparison.

Performs `half` if-equal comparison of inputs a and b. NaN inputs generate false results.

Parameters

- **a** – **[in]** - half. Is only being read.
- **b** – **[in]** - half. Is only being read.

Returns
bool

- The boolean result of if-equal comparison of a and b.

__host__ __device__ bool **__hequ**(const *__half* a, const *__half* b)
Performs `half` unordered if-equal comparison.

Performs `half` if-equal comparison of inputs a and b. NaN inputs generate true results.

Parameters

- **a** – **[in]** - half. Is only being read.
- **b** – **[in]** - half. Is only being read.

Returns
bool

- The boolean result of unordered if-equal comparison of a and b.

__host__ __device__ bool **__hge**(const *__half* a, const *__half* b)
Performs `half` greater-equal comparison.

Performs `half` greater-equal comparison of inputs a and b. NaN inputs generate false results.

Parameters

- **a** – **[in]** - half. Is only being read.
- **b** – **[in]** - half. Is only being read.

Returns
bool

- The boolean result of greater-equal comparison of a and b.

__host__ __device__ bool **__hgeu**(const *__half* a, const *__half* b)

Performs `half` unordered greater-equal comparison.

Performs `half` greater-equal comparison of inputs a and b. NaN inputs generate true results.

Parameters

- **a** – **[in]** - half. Is only being read.
- **b** – **[in]** - half. Is only being read.

Returns
bool

- The boolean result of unordered greater-equal comparison of a and b.

__host__ __device__ bool **__hgt**(const *__half* a, const *__half* b)

Performs `half` greater-than comparison.

Performs `half` greater-than comparison of inputs a and b. NaN inputs generate false results.

Parameters

- **a** – **[in]** - half. Is only being read.
- **b** – **[in]** - half. Is only being read.

Returns
bool

- The boolean result of greater-than comparison of a and b.

__host__ __device__ bool **__hgtu**(const *__half* a, const *__half* b)

Performs `half` unordered greater-than comparison.

Performs `half` greater-than comparison of inputs a and b. NaN inputs generate true results.

Parameters

- **a** – **[in]** - half. Is only being read.
- **b** – **[in]** - half. Is only being read.

Returns
bool

- The boolean result of unordered greater-than comparison of a and b.

__host__ __device__ int **__hisinf**(const *__half* a)

Checks if the input `half` number is infinite.

Checks if the input `half` number a is infinite.

Parameters
a – **[in]** - half. Is only being read.

Returns

int

- -1 if a is equal to negative infinity,
- 1 if a is equal to positive infinity,
- 0 otherwise.

__host__ __device__ bool **__hisnan**(const *__half* a)

Determine whether `half` argument is a NaN.

Determine whether `half` value a is a NaN.

Parameters

a – **[in]** - half. Is only being read.

Returns

bool

- true if argument is NaN.

__host__ __device__ bool **__hle**(const *__half* a, const *__half* b)

Performs `half` less-equal comparison.

Performs `half` less-equal comparison of inputs a and b. NaN inputs generate false results.

Parameters

- **a** – **[in]** - half. Is only being read.
- **b** – **[in]** - half. Is only being read.

Returns

bool

- The boolean result of less-equal comparison of a and b.

__host__ __device__ bool **__hleu**(const *__half* a, const *__half* b)

Performs `half` unordered less-equal comparison.

Performs `half` less-equal comparison of inputs a and b. NaN inputs generate true results.

Parameters

- **a** – **[in]** - half. Is only being read.
- **b** – **[in]** - half. Is only being read.

Returns

bool

- The boolean result of unordered less-equal comparison of a and b.

__host__ __device__ bool **__hlt**(const *__half* a, const *__half* b)

Performs `half` less-than comparison.

Performs `half` less-than comparison of inputs a and b. NaN inputs generate false results.

Parameters

- **a** – **[in]** - half. Is only being read.

- **b** – **[in]** - half. Is only being read.

Returns
bool

- The boolean result of less-than comparison of a and b.

__host__ __device__ bool **__hltu**(const *__half* a, const *__half* b)

Performs `half` unordered less-than comparison.

Performs `half` less-than comparison of inputs a and b. NaN inputs generate true results.

Parameters

- **a** – **[in]** - half. Is only being read.
- **b** – **[in]** - half. Is only being read.

Returns
bool

- The boolean result of unordered less-than comparison of a and b.

__host__ __device__ *__half* **__hmax**(const *__half* a, const *__half* b)

Calculates `half` maximum of two input values.

Calculates `half` max(a, b) defined as (a > b) ? a : b.

- If either of inputs is NaN, the other input is returned.
- If both inputs are NaNs, then canonical NaN is returned.
- If values of both inputs are 0.0, then +0.0 > -0.0

Parameters

- **a** – **[in]** - half. Is only being read.
- **b** – **[in]** - half. Is only being read.

Returns
half

__host__ __device__ *__half* **__hmax_nan**(const *__half* a, const *__half* b)

Calculates `half` maximum of two input values, NaNs pass through.

Calculates `half` max(a, b) defined as (a > b) ? a : b.

- If either of inputs is NaN, then canonical NaN is returned.
- If values of both inputs are 0.0, then +0.0 > -0.0

Parameters

- **a** – **[in]** - half. Is only being read.
- **b** – **[in]** - half. Is only being read.

Returns
half

__host__ __device__ *__half* **__hmin**(const *__half* a, const *__half* b)

Calculates `half` minimum of two input values.

Calculates `half` min(a, b) defined as (a < b) ? a : b.

- If either of inputs is NaN, the other input is returned.
- If both inputs are NaNs, then canonical NaN is returned.
- If values of both inputs are 0.0, then +0.0 > -0.0

Parameters

- **a** – **[in]** - half. Is only being read.
- **b** – **[in]** - half. Is only being read.

Returns

half

__host__ __device__ *__half* **__hmin_nan**(const *__half* a, const *__half* b)

Calculates `half` minimum of two input values, NaNs pass through.

Calculates `half` min(a, b) defined as (a < b) ? a : b.

- If either of inputs is NaN, then canonical NaN is returned.
- If values of both inputs are 0.0, then +0.0 > -0.0

Parameters

- **a** – **[in]** - half. Is only being read.
- **b** – **[in]** - half. Is only being read.

Returns

half

__host__ __device__ bool **__hne**(const *__half* a, const *__half* b)

Performs `half` not-equal comparison.

Performs `half` not-equal comparison of inputs a and b. NaN inputs generate false results.

Parameters

- **a** – **[in]** - half. Is only being read.
- **b** – **[in]** - half. Is only being read.

Returns

bool

- The boolean result of not-equal comparison of a and b.

__host__ __device__ bool **__hneu**(const *__half* a, const *__half* b)

Performs `half` unordered not-equal comparison.

Performs `half` not-equal comparison of inputs a and b. NaN inputs generate true results.

Parameters

- **a** – **[in]** - half. Is only being read.
- **b** – **[in]** - half. Is only being read.

Returns
bool

- The boolean result of unordered not-equal comparison of a and b.

__host__ __device__ bool **operator!=**(const *__half* &lh, const *__half* &rh)
Performs `half` unordered compare not-equal operation.

See also:

__hneu(__half, __half)

__host__ __device__ bool **operator<**(const *__half* &lh, const *__half* &rh)
Performs `half` ordered less-than compare operation.

See also:

__hlt(__half, __half)

__host__ __device__ bool **operator<=**(const *__half* &lh, const *__half* &rh)
Performs `half` ordered less-or-equal compare operation.

See also:

__hle(__half, __half)

__host__ __device__ bool **operator==**(const *__half* &lh, const *__half* &rh)
Performs `half` ordered compare equal operation.

See also:

__heq(__half, __half)

__host__ __device__ bool **operator>**(const *__half* &lh, const *__half* &rh)
Performs `half` ordered greater-than compare operation.

See also:

__hgt(__half, __half)

__host__ __device__ bool **operator>=**(const *__half* &lh, const *__half* &rh)
Performs `half` ordered greater-or-equal compare operation.

See also:

__hge(__half, __half)

4.4. Half Math Functions

To use these functions, include the header file `cuda_fp16.h` in your program.

Functions

__device__ __half *hceil*(const __half h)
Calculate ceiling of the input argument.

__device__ __half *hcos*(const __half a)
Calculates `half` cosine in round-to-nearest-even mode.

__device__ __half *hexp*(const __half a)
Calculates `half` natural exponential function in round-to-nearest-even mode.

__device__ __half *hexp10*(const __half a)
Calculates `half` decimal exponential function in round-to-nearest-even mode.

__device__ __half *hexp2*(const __half a)
Calculates `half` binary exponential function in round-to-nearest-even mode.

__device__ __half *hfloor*(const __half h)
Calculate the largest integer less than or equal to h .

__device__ __half *hlog*(const __half a)
Calculates `half` natural logarithm in round-to-nearest-even mode.

__device__ __half *hlog10*(const __half a)
Calculates `half` decimal logarithm in round-to-nearest-even mode.

__device__ __half *hlog2*(const __half a)
Calculates `half` binary logarithm in round-to-nearest-even mode.

__device__ __half *hrcp*(const __half a)
Calculates `half` reciprocal in round-to-nearest-even mode.

__device__ __half *hrint*(const __half h)
Round input to nearest integer value in half-precision floating-point number.

__device__ __half *hrsqrt*(const __half a)
Calculates `half` reciprocal square root in round-to-nearest-even mode.

__device__ __half *hsin*(const __half a)
Calculates `half` sine in round-to-nearest-even mode.

__device__ __half *hsqrt*(const __half a)
Calculates `half` square root in round-to-nearest-even mode.

__device__ __half *htanh*(const __half a)
Calculates `half` hyperbolic tangent function in round-to-nearest-even mode.

__device__ __half *htanh_approx*(const __half a)
Calculates approximate `half` hyperbolic tangent function.

__device__ __half *htrunc*(const __half h)
Truncate input argument to the integral part.

4.4.1. Functions

__device__ *__half* **hceil**(const *__half* h)

Calculate ceiling of the input argument.

Compute the smallest integer value not less than h.

Parameters

h – **[in]** - half. Is only being read.

Returns

half

- The smallest integer value not less than h.
- hceil(± 0) returns ± 0.
- hceil($\pm\infty$) returns $\pm\infty$.
- hceil(NaN) returns NaN.

__device__ *__half* **hcos**(const *__half* a)

Calculates `half` cosine in round-to-nearest-even mode.

Calculates `half` cosine of input a in round-to-nearest-even mode.

Parameters

a – **[in]** - half. Is only being read.

Returns

half

- The cosine of a.
- hcos (± 0) returns 1.
- hcos $(\pm\infty)$ returns NaN.
- hcos(NaN) returns NaN.

__device__ *__half* **hexp**(const *__half* a)

Calculates `half` natural exponential function in round-to-nearest-even mode.

Calculates `half` natural exponential function of input: e^a in round-to-nearest-even mode.

Parameters

a – **[in]** - half. Is only being read.

Returns

half

- The natural exponential function on a.
- hexp (± 0) returns 1.
- hexp $(-\infty)$ returns +0.
- hexp $(+\infty)$ returns $+\infty$.
- hexp(NaN) returns NaN.

__device__ *__half* **hexp10**(const *__half* a)

Calculates `half` decimal exponential function in round-to-nearest-even mode.

Calculates `half` decimal exponential function of input: 10^a in round-to-nearest-even mode.

Parameters
: **a** – **[in]** - half. Is only being read.

Returns
: half

- The decimal exponential function on a.
- hexp10 (± 0) returns 1.
- hexp10 $(-\infty)$ returns +0.
- hexp10 $(+\infty)$ returns $+\infty$.
- hexp10(NaN) returns NaN.

__device__ *__half* **hexp2**(const *__half* a)

Calculates `half` binary exponential function in round-to-nearest-even mode.

Calculates `half` binary exponential function of input: 2^a in round-to-nearest-even mode.

Parameters
: **a** – **[in]** - half. Is only being read.

Returns
: half

- The binary exponential function on a.
- hexp2 (± 0) returns 1.
- hexp2 $(-\infty)$ returns +0.
- hexp2 $(+\infty)$ returns $+\infty$.
- hexp2(NaN) returns NaN.

__device__ *__half* **hfloor**(const *__half* h)

Calculate the largest integer less than or equal to h.

Calculate the largest integer value which is less than or equal to h.

Parameters
: **h** – **[in]** - half. Is only being read.

Returns
: half

- The largest integer value which is less than or equal to h.
- hfloor(± 0) returns ± 0.
- hfloor($\pm\infty$) returns $\pm\infty$.
- hfloor(NaN) returns NaN.

__device__ *__half* **hlog**(const *__half* a)

Calculates `half` natural logarithm in round-to-nearest-even mode.

Calculates `half` natural logarithm of input: $\ln(a)$ in round-to-nearest-even mode.

Parameters
: **a** – **[in]** - half. Is only being read.

Returns
: half

- The natural logarithm of a.
- hlog (± 0) returns $-\infty$.
- hlog(1) returns +0.
- hlog(x), x < 0 returns NaN.
- hlog $(+\infty)$ returns $+\infty$.
- hlog(NaN) returns NaN.

__device__ *__half* **hlog10**(const *__half* a)

Calculates `half` decimal logarithm in round-to-nearest-even mode.

Calculates `half` decimal logarithm of input: $\log_{10}(a)$ in round-to-nearest-even mode.

Parameters

a – **[in]** - half. Is only being read.

Returns

half

- The decimal logarithm of a.
- hlog10 (± 0) returns $-\infty$.
- hlog10(1) returns +0.
- hlog10(x), x < 0 returns NaN.
- hlog10 $(+\infty)$ returns $+\infty$.
- hlog10(NaN) returns NaN.

__device__ *__half* **hlog2**(const *__half* a)

Calculates `half` binary logarithm in round-to-nearest-even mode.

Calculates `half` binary logarithm of input: $\log_2(a)$ in round-to-nearest-even mode.

Parameters

a – **[in]** - half. Is only being read.

Returns

half

- The binary logarithm of a.
- hlog2 (± 0) returns $-\infty$.
- hlog2(1) returns +0.
- hlog2(x), x < 0 returns NaN.
- hlog2 $(+\infty)$ returns $+\infty$.
- hlog2(NaN) returns NaN.

__device__ *__half* **hrcp**(const *__half* a)

Calculates `half` reciprocal in round-to-nearest-even mode.

Calculates `half` reciprocal of input: $\frac{1}{a}$ in round-to-nearest-even mode.

Parameters

a – **[in]** - half. Is only being read.

Returns

half

- The reciprocal of a.
- hrcp (± 0) returns $\pm\infty$.
- hrcp $(\pm\infty)$ returns ± 0.
- hrcp(NaN) returns NaN.

__device__ *__half* **hrint**(const *__half* h)

Round input to nearest integer value in half-precision floating-point number.

Round h to the nearest integer value in half-precision floating-point format, with halfway cases rounded to the nearest even integer value.

Parameters

h – **[in]** - half. Is only being read.

Returns

half

- The nearest integer to h.
- hrint(± 0) returns ± 0.
- hrint($\pm\infty$) returns $\pm\infty$.
- hrint(NaN) returns NaN.

__device__ *__half* **hrsqrt**(const *__half* a)

Calculates `half` reciprocal square root in round-to-nearest-even mode.

Calculates `half` reciprocal square root of input: $\frac{1}{\sqrt{a}}$ in round-to-nearest-even mode.

Parameters

a – **[in]** - half. Is only being read.

Returns

half

- The reciprocal square root of a.
- hrsqrt (± 0) returns $\pm\infty$.
- hrsqrt $(+\infty)$ returns +0.
- hrsqrt $(x), x < 0.0$ returns NaN.
- hrsqrt(NaN) returns NaN.

__device__ *__half* **hsin**(const *__half* a)

Calculates `half` sine in round-to-nearest-even mode.

Calculates `half` sine of input a in round-to-nearest-even mode.

Parameters

a – **[in]** - half. Is only being read.

Returns

half

- The sine of a.
- hsin (± 0) returns (± 0).
- hsin $(\pm\infty)$ returns NaN.
- hsin(NaN) returns NaN.

__device__ *__half* **hsqrt**(const *__half* a)

Calculates `half` square root in round-to-nearest-even mode.

Calculates `half` square root of input: $\sqrt{a}$ in round-to-nearest-even mode.

Parameters

a – **[in]** - half. Is only being read.

Returns

half

- The square root of a.
- hsqrt $(+\infty)$ returns $+\infty$.
- hsqrt (± 0) returns ± 0.
- hsqrt $(x), x < 0.0$ returns NaN.
- hsqrt(NaN) returns NaN.

__device__ *__half* **htanh**(const *__half* a)

Calculates `half` hyperbolic tangent function in round-to-nearest-even mode.

Calculates `half` hyperbolic tangent function: tanh(a) in round-to-nearest-even mode.

Parameters

a – **[in]** - half. Is only being read.

Returns

half

- The hyperbolic tangent function of a.
- htanh (± 0) returns (± 0).
- htanh $(\pm\infty)$ returns (± 1).
- htanh(NaN) returns NaN.

__device__ *__half* **htanh_approx**(const *__half* a)

Calculates approximate `half` hyperbolic tangent function.

Calculates approximate `half` hyperbolic tangent function: tanh(a). This operation uses HW acceleration on devices of compute capability 7.5 and higher.

Parameters

a – **[in]** - half. Is only being read.

Returns

half

- The approximate hyperbolic tangent function of a.
- htanh_approx (± 0) returns (± 0).
- htanh_approx $(\pm\infty)$ returns (± 1).
- htanh_approx(NaN) returns NaN.

__device__ *__half* **htrunc**(const *__half* h)

Truncate input argument to the integral part.

Round h to the largest integer value that does not exceed h in magnitude.

Parameters

h – **[in]** - half. Is only being read.

Returns

half

- The truncated value.
- htrunc(± 0) returns ± 0.
- htrunc($\pm\infty$) returns $\pm\infty$.
- htrunc(NaN) returns NaN.

4.5. Half Precision Conversion and Data Movement

To use these functions, include the header file `cuda_fp16.h` in your program.

Functions

__host__ __device__ __half *__double2half*(const double a)
Converts double number to half precision in round-to-nearest-even mode and returns `half` with converted value.

__host__ __device__ __half2 *__float22half2_rn*(const float2 a)
Converts both components of `float2` number to half precision in round-to-nearest-even mode and returns `half2` with converted values.

__host__ __device__ __half *__float2half*(const float a)
Converts float number to half precision in round-to-nearest-even mode and returns `half` with converted value.

__host__ __device__ __half2 *__float2half2_rn*(const float a)
Converts input to half precision in round-to-nearest-even mode and populates both halves of `half2` with converted value.

__host__ __device__ __half *__float2half_rd*(const float a)
Converts float number to half precision in round-down mode and returns `half` with converted value.

__host__ __device__ __half *__float2half_rn*(const float a)
Converts float number to half precision in round-to-nearest-even mode and returns `half` with converted value.

__host__ __device__ __half *__float2half_ru*(const float a)
Converts float number to half precision in round-up mode and returns `half` with converted value.

__host__ __device__ __half *__float2half_rz*(const float a)
Converts float number to half precision in round-towards-zero mode and returns `half` with converted value.

__host__ __device__ __half2 *__floats2half2_rn*(const float a, const float b)
Converts both input floats to half precision in round-to-nearest-even mode and returns `half2` with converted values.

__host__ __device__ float2 *__half22float2*(const __half2 a)
Converts both halves of `half2` to `float2` and returns the result.

__host__ __device__ *__half2::__half2*(const __half2_raw &h2r)
Constructor from `__half2_raw`.

__host__ __device__ constexpr *__half2::__half2*(const __half &a, const __half &b)
Constructor from two `__half` variables.

__host__ __device__ *__half2::__half2*(const __half2 &&src)
Move constructor, available for C++11 and later dialects.

__host__ __device__ *__half2::__half2*(const __half2 &src)
Copy constructor.

***__half2::__half2*()=default**
Constructor by default.

__host__ __device__ *__half2::operator __half2_raw*() const
Conversion operator to `__half2_raw`.

__host__ __device__ __half2 & *__half2::operator=*(const __half2_raw &h2r)
Assignment operator from `__half2_raw`.

__host__ __device__ __half2 & *__half2::operator=*(const __half2 &&src)
Move assignment operator, available for C++11 and later dialects.

__host__ __device__ __half2 & *__half2::operator=*(const __half2 &src)
Copy assignment operator.

__host__ __device__ signed char *__half2char_rz*(const __half h)
Convert a half to a signed char in round-towards-zero mode.

__host__ __device__ float *__half2float*(const __half a)
Converts `half` number to float.

__host__ __device__ __half2 *__half2half2*(const __half a)
Returns `half2` with both halves equal to the input value.

__device__ int *__half2int_rd*(const __half h)
Convert a half to a signed integer in round-down mode.

__device__ int *__half2int_rn*(const __half h)
Convert a half to a signed integer in round-to-nearest-even mode.

__device__ int *__half2int_ru*(const __half h)
Convert a half to a signed integer in round-up mode.

__host__ __device__ int *__half2int_rz*(const __half h)
Convert a half to a signed integer in round-towards-zero mode.

__device__ long long int *__half2ll_rd*(const __half h)
Convert a half to a signed 64-bit integer in round-down mode.

__device__ long long int *__half2ll_rn*(const __half h)
Convert a half to a signed 64-bit integer in round-to-nearest-even mode.

__device__ long long int *__half2ll_ru*(const __half h)
Convert a half to a signed 64-bit integer in round-up mode.

__host__ __device__ long long int *__half2ll_rz*(const __half h)
Convert a half to a signed 64-bit integer in round-towards-zero mode.

__device__ short int *__half2short_rd*(const __half h)
Convert a half to a signed short integer in round-down mode.

__device__ short int ***__half2short_rn*****(const __half h)**
Convert a half to a signed short integer in round-to-nearest-even mode.

__device__ short int ***__half2short_ru*****(const __half h)**
Convert a half to a signed short integer in round-up mode.

__host__ __device__ short int ***__half2short_rz*****(const __half h)**
Convert a half to a signed short integer in round-towards-zero mode.

__host__ __device__ unsigned char ***__half2uchar_rz*****(const __half h)**
Convert a half to an unsigned char in round-towards-zero mode.

__device__ unsigned int ***__half2uint_rd*****(const __half h)**
Convert a half to an unsigned integer in round-down mode.

__device__ unsigned int ***__half2uint_rn*****(const __half h)**
Convert a half to an unsigned integer in round-to-nearest-even mode.

__device__ unsigned int ***__half2uint_ru*****(const __half h)**
Convert a half to an unsigned integer in round-up mode.

__host__ __device__ unsigned int ***__half2uint_rz*****(const __half h)**
Convert a half to an unsigned integer in round-towards-zero mode.

__device__ unsigned long long int ***__half2ull_rd*****(const __half h)**
Convert a half to an unsigned 64-bit integer in round-down mode.

__device__ unsigned long long int ***__half2ull_rn*****(const __half h)**
Convert a half to an unsigned 64-bit integer in round-to-nearest-even mode.

__device__ unsigned long long int ***__half2ull_ru*****(const __half h)**
Convert a half to an unsigned 64-bit integer in round-up mode.

__host__ __device__ unsigned long long int ***__half2ull_rz*****(const __half h)**
Convert a half to an unsigned 64-bit integer in round-towards-zero mode.

__device__ unsigned short int ***__half2ushort_rd*****(const __half h)**
Convert a half to an unsigned short integer in round-down mode.

__device__ unsigned short int ***__half2ushort_rn*****(const __half h)**
Convert a half to an unsigned short integer in round-to-nearest-even mode.

__device__ unsigned short int ***__half2ushort_ru*****(const __half h)**
Convert a half to an unsigned short integer in round-up mode.

__host__ __device__ unsigned short int ***__half2ushort_rz*****(const __half h)**
Convert a half to an unsigned short integer in round-towards-zero mode.

__host__ __device__ constexpr ***__half::__half*****(const __half_raw &hr)**
Constructor from `__half_raw`.

__host__ __device__ ***__half::__half*****(const unsigned short val)**
Construct `__half` from `unsigned short` integer input using default round-to-nearest-even rounding mode.

__host__ __device__ ***__half::__half*****(const unsigned int val)**
Construct `__half` from `unsigned int` input using default round-to-nearest-even rounding mode.

__host__ __device__ ***__half::__half*****(const short val)**
Construct `__half` from `short` integer input using default round-to-nearest-even rounding mode.

__host__ __device__ *__half::__half*(const double f)
Construct `__half` from `double` input using default round-to-nearest-even rounding mode.

__host__ __device__ *__half::__half*(const unsigned long val)
Construct `__half` from `unsigned long` input using default round-to-nearest-even rounding mode.

__host__ __device__ *__half::__half*(const float f)
Construct `__half` from `float` input using default round-to-nearest-even rounding mode.

__host__ __device__ *__half::__half*(const int val)
Construct `__half` from `int` input using default round-to-nearest-even rounding mode.

__host__ __device__ *__half::__half*(const long val)
Construct `__half` from `long` input using default round-to-nearest-even rounding mode.

__host__ __device__ *__half::__half*(const long long val)
Construct `__half` from `long long` input using default round-to-nearest-even rounding mode.

***__half::__half*()=default**
Constructor by default.

__host__ __device__ *__half::__half*(const __nv_bfloat16 f)
Construct `__half` from `__nv_bfloat16` input using default round-to-nearest-even rounding mode.

__host__ __device__ *__half::__half*(const unsigned long long val)
Construct `__half` from `unsigned long long` input using default round-to-nearest-even rounding mode.

__host__ __device__ *__half::operator __half_raw*() const volatile
Type cast to `__half_raw` operator with `volatile` input.

__host__ __device__ *__half::operator __half_raw*() const
Type cast to `__half_raw` operator.

__host__ __device__ constexpr *__half::operator bool*() const
Conversion operator to `bool` data type.

__host__ __device__ *__half::operator char*() const
Conversion operator to an implementation defined `char` data type.

__host__ __device__ *__half::operator float*() const
Type cast to `float` operator.

__host__ __device__ *__half::operator int*() const
Conversion operator to `int` data type.

__host__ __device__ *__half::operator long*() const
Conversion operator to `long` data type.

__host__ __device__ *__half::operator long long*() const
Conversion operator to `long long` data type.

__host__ __device__ *__half::operator short*() const
Conversion operator to `short` data type.

__host__ __device__ *__half::operator signed char*() const
Conversion operator to `signed char` data type.

__host__ __device__ *__half::operator unsigned char*() const
Conversion operator to `unsigned char` data type.

__host__ __device__ *__half::operator unsigned int*() const
Conversion operator to `unsigned int` data type.

__host__ __device__ *__half::operator unsigned long*() const
Conversion operator to `unsigned long` data type.

__host__ __device__ *__half::operator unsigned long long*() const
Conversion operator to `unsigned long long` data type.

__host__ __device__ *__half::operator unsigned short*() const
Conversion operator to `unsigned short` data type.

__host__ __device__ __half & *__half::operator=*(const float f)
Type cast to `__half` assignment operator from `float` input using default round-to-nearest-even rounding mode.

__host__ __device__ volatile __half & *__half::operator=*(const volatile __half_raw &hr) volatile
Assignment operator from `volatile __half_raw` to `volatile __half`.

__host__ __device__ __half & *__half::operator=*(const long long val)
Type cast from `long long` assignment operator, using default round-to-nearest-even rounding mode.

__host__ __device__ volatile __half & *__half::operator=*(const __half_raw &hr) volatile
Assignment operator from `__half_raw` to `volatile __half`.

__host__ __device__ __half & *__half::operator=*(const unsigned int val)
Type cast from `unsigned int` assignment operator, using default round-to-nearest-even rounding mode.

__host__ __device__ __half & *__half::operator=*(const unsigned short val)
Type cast from `unsigned short` assignment operator, using default round-to-nearest-even rounding mode.

__host__ __device__ __half & *__half::operator=*(const short val)
Type cast from `short` assignment operator, using default round-to-nearest-even rounding mode.

__host__ __device__ __half & *__half::operator=*(const double f)
Type cast to `__half` assignment operator from `double` input using default round-to-nearest-even rounding mode.

__host__ __device__ __half & *__half::operator=*(const __half_raw &hr)
Assignment operator from `__half_raw`.

__host__ __device__ __half & *__half::operator=*(const unsigned long long val)
Type cast from `unsigned long long` assignment operator, using default round-to-nearest-even rounding mode.

__host__ __device__ __half & *__half::operator=*(const int val)
Type cast from `int` assignment operator, using default round-to-nearest-even rounding mode.

__host__ __device__ short int *__half_as_short*(const __half h)
Reinterprets bits in a `half` as a signed short integer.

__host__ __device__ unsigned short int *__half_as_ushort*(const __half h)
Reinterprets bits in a `half` as an unsigned short integer.

__host__ __device__ __half2 *__halves2half2*(const __half a, const __half b)
Combines two `half` numbers into one `half2` number.

__host__ __device__ float *__high2float*(const __half2 a)
Converts high 16 bits of `half2` to float and returns the result.

__host__ __device__ __half *__high2half*(const __half2 a)
Returns high 16 bits of `half2` input.

__host__ __device__ __half2 *__high2half2*(const __half2 a)
Extracts high 16 bits from `half2` input.

__host__ __device__ __half2 *__highs2half2*(const __half2 a, const __half2 b)
Extracts high 16 bits from each of the two `half2` inputs and combines into one `half2` number.

__host__ __device__ __half *__int2half_rd*(const int i)
Convert a signed integer to a half in round-down mode.

__host__ __device__ __half *__int2half_rn*(const int i)
Convert a signed integer to a half in round-to-nearest-even mode.

__host__ __device__ __half *__int2half_ru*(const int i)
Convert a signed integer to a half in round-up mode.

__host__ __device__ __half *__int2half_rz*(const int i)
Convert a signed integer to a half in round-towards-zero mode.

__device__ __half2 *__ldca*(const __half2 *const ptr)
Generates a `ld.global.ca` load instruction.

__device__ __half *__ldca*(const __half *const ptr)
Generates a `ld.global.ca` load instruction.

__device__ __half *__ldcg*(const __half *const ptr)
Generates a `ld.global.cg` load instruction.

__device__ __half2 *__ldcg*(const __half2 *const ptr)
Generates a `ld.global.cg` load instruction.

__device__ __half *__ldcs*(const __half *const ptr)
Generates a `ld.global.cs` load instruction.

__device__ __half2 *__ldcs*(const __half2 *const ptr)
Generates a `ld.global.cs` load instruction.

__device__ __half2 *__ldcv*(const __half2 *const ptr)
Generates a `ld.global.cv` load instruction.

__device__ __half *__ldcv*(const __half *const ptr)
Generates a `ld.global.cv` load instruction.

__device__ __half2 *__ldg*(const __half2 *const ptr)
Generates a `ld.global.nc` load instruction.

__device__ __half *__ldg*(const __half *const ptr)
Generates a `ld.global.nc` load instruction.

__device__ __half *__ldlu*(const __half *const ptr)
Generates a `ld.global.lu` load instruction.

__device__ __half2 *__ldlu*(const __half2 *const ptr)
Generates a `ld.global.lu` load instruction.

__host__ __device__ __half *__ll2half_rd*(const long long int i)
Convert a signed 64-bit integer to a half in round-down mode.

__host__ __device__ __half *__ll2half_rn*(const long long int i)
Convert a signed 64-bit integer to a half in round-to-nearest-even mode.

__host__ __device__ __half *__ll2half_ru*(const long long int i)
Convert a signed 64-bit integer to a half in round-up mode.

__host__ __device__ __half *__ll2half_rz*(const long long int i)
Convert a signed 64-bit integer to a half in round-towards-zero mode.

__host__ __device__ float *__low2float*(const __half2 a)
Converts low 16 bits of `half2` to float and returns the result.

__host__ __device__ __half *__low2half*(const __half2 a)
Returns low 16 bits of `half2` input.

__host__ __device__ __half2 *__low2half2*(const __half2 a)
Extracts low 16 bits from `half2` input.

__host__ __device__ __half2 *__lowhigh2highlow*(const __half2 a)
Swaps both halves of the `half2` input.

__host__ __device__ __half2 *__lows2half2*(const __half2 a, const __half2 b)
Extracts low 16 bits from each of the two `half2` inputs and combines into one `half2` number.

__device__ __half *__shfl_down_sync*(const unsigned int mask, const __half var, const unsigned int delta, const int width=warpSize)
Exchange a variable between threads within a warp.

__device__ __half2 *__shfl_down_sync*(const unsigned int mask, const __half2 var, const unsigned int delta, const int width=warpSize)
Exchange a variable between threads within a warp.

__device__ __half2 *__shfl_sync*(const unsigned int mask, const __half2 var, const int srcLane, const int width=warpSize)
Exchange a variable between threads within a warp.

__device__ __half *__shfl_sync*(const unsigned int mask, const __half var, const int srcLane, const int width=warpSize)
Exchange a variable between threads within a warp.

__device__ __half2 *__shfl_up_sync*(const unsigned int mask, const __half2 var, const unsigned int delta, const int width=warpSize)
Exchange a variable between threads within a warp.

__device__ __half *__shfl_up_sync*(const unsigned int mask, const __half var, const unsigned int delta, const int width=warpSize)
Exchange a variable between threads within a warp.

__device__ __half2 *__shfl_xor_sync*(const unsigned int mask, const __half2 var, const int laneMask, const int width=warpSize)
Exchange a variable between threads within a warp.

__device__ __half *__shfl_xor_sync*(const unsigned int mask, const __half var, const int laneMask, const int width=warpSize)
Exchange a variable between threads within a warp.

__host__ __device__ __half *__short2half_rd*(const short int i)
Convert a signed short integer to a half in round-down mode.

__host__ __device__ __half *__short2half_rn*(const short int i)
Convert a signed short integer to a half in round-to-nearest-even mode.

__host__ __device__ __half *__short2half_ru*(const short int i)
Convert a signed short integer to a half in round-up mode.

__host__ __device__ __half *__short2half_rz*(const short int i)
Convert a signed short integer to a half in round-towards-zero mode.

__host__ __device__ __half *__short_as_half*(const short int i)
Reinterprets bits in a signed short integer as a `half`.

__device__ void *__stcg*(__half2 *const ptr, const __half2 value)
Generates a `st.global.cg` store instruction.

__device__ void *__stcg*(__half *const ptr, const __half value)
Generates a `st.global.cg` store instruction.

__device__ void *__stcs*(__half2 *const ptr, const __half2 value)
Generates a `st.global.cs` store instruction.

__device__ void *__stcs*(__half *const ptr, const __half value)
Generates a `st.global.cs` store instruction.

__device__ void *__stwb*(__half2 *const ptr, const __half2 value)
Generates a `st.global.wb` store instruction.

__device__ void *__stwb*(__half *const ptr, const __half value)
Generates a `st.global.wb` store instruction.

__device__ void *__stwt*(__half *const ptr, const __half value)
Generates a `st.global.wt` store instruction.

__device__ void *__stwt*(__half2 *const ptr, const __half2 value)
Generates a `st.global.wt` store instruction.

__host__ __device__ __half *__uint2half_rd*(const unsigned int i)
Convert an unsigned integer to a half in round-down mode.

__host__ __device__ __half *__uint2half_rn*(const unsigned int i)
Convert an unsigned integer to a half in round-to-nearest-even mode.

__host__ __device__ __half *__uint2half_ru*(const unsigned int i)
Convert an unsigned integer to a half in round-up mode.

__host__ __device__ __half *__uint2half_rz*(const unsigned int i)
Convert an unsigned integer to a half in round-towards-zero mode.

__host__ __device__ __half *__ull2half_rd*(const unsigned long long int i)
Convert an unsigned 64-bit integer to a half in round-down mode.

__host__ __device__ __half *__ull2half_rn*(const unsigned long long int i)
Convert an unsigned 64-bit integer to a half in round-to-nearest-even mode.

__host__ __device__ __half *__ull2half_ru*(const unsigned long long int i)
Convert an unsigned 64-bit integer to a half in round-up mode.

__host__ __device__ __half *__ull2half_rz*(const unsigned long long int i)
Convert an unsigned 64-bit integer to a half in round-towards-zero mode.

__host__ __device__ __half *__ushort2half_rd*(const unsigned short int i)
Convert an unsigned short integer to a half in round-down mode.

__host__ __device__ __half *__ushort2half_rn*(const unsigned short int i)
Convert an unsigned short integer to a half in round-to-nearest-even mode.

__host__ __device__ __half *__ushort2half_ru*(const unsigned short int i)
Convert an unsigned short integer to a half in round-up mode.

__host__ __device__ __half *__ushort2half_rz*(const unsigned short int i)
Convert an unsigned short integer to a half in round-towards-zero mode.

__host__ __device__ __half *__ushort_as_half*(const unsigned short int i)
Reinterprets bits in an unsigned short integer as a `half` .

__host__ __device__ __half2 *make_half2*(const __half x, const __half y)
Vector function, combines two `__half` numbers into one `__half2` number.

4.5.1. Functions

__host__ __device__ *__half* **`__double2half`**(const double a)

Converts double number to half precision in round-to-nearest-even mode and returns `half` with converted value.

Converts double number a to half precision in round-to-nearest-even mode.

Parameters
a – **[in]** - double. Is only being read.

Returns
half

- a converted to half precision using round-to-nearest-even mode.
- __double2half (± 0) returns ± 0.
- __double2half $(\pm\infty)$ returns $\pm\infty$.
- __double2half(NaN) returns NaN.

__host__ __device__ *__half2* **`__float22half2_rn`**(const float2 a)

Converts both components of `float2` number to half precision in round-to-nearest-even mode and returns `half2` with converted values.

Converts both components of `float2` to half precision in round-to-nearest-even mode and combines the results into one `half2` number. Low 16 bits of the return value correspond to `a.x` and high 16 bits of the return value correspond to `a.y`.

See also:

__float2half_rn(float) for further details.

Parameters
a – **[in]** - float2. Is only being read.

Returns
half2

- The `half2` which has corresponding halves equal to the converted `float2` components.

__host__ __device__ *__half* **`__float2half`**(const float a)

Converts float number to half precision in round-to-nearest-even mode and returns `half` with converted value.

Converts float number a to half precision in round-to-nearest-even mode.

See also:

__float2half_rn(float) for further details.

Parameters
a – **[in]** - float. Is only being read.

Returns
half

- a converted to half precision using round-to-nearest-even mode.

__host__ __device__ *__half2* **`__float2half2_rn`**(const float a)

Converts input to half precision in round-to-nearest-even mode and populates both halves of `half2` with converted value.

Converts input a to half precision in round-to-nearest-even mode and populates both halves of `half2` with converted value.

See also:

__float2half_rn(float) for further details.

Parameters
a – **[in]** - float. Is only being read.

Returns
half2

- The `half2` value with both halves equal to the converted half precision number.

__host__ __device__ *__half* **`__float2half_rd`**(const float a)

Converts float number to half precision in round-down mode and returns `half` with converted value.

Converts float number a to half precision in round-down mode.

Parameters
a – **[in]** - float. Is only being read.

Returns
half

- a converted to half precision using round-down mode.
- __float2half_rd (± 0) returns ± 0.
- __float2half_rd $(\pm\infty)$ returns $\pm\infty$.
- __float2half_rd(NaN) returns NaN.

__host__ __device__ *__half* **`__float2half_rn`**(const float a)

Converts float number to half precision in round-to-nearest-even mode and returns `half` with converted value.

Converts float number a to half precision in round-to-nearest-even mode.

Parameters
a – **[in]** - float. Is only being read.

Returns
half

- a converted to half precision using round-to-nearest-even mode.
- __float2half_rn (± 0) returns ± 0.
- __float2half_rn $(\pm\infty)$ returns $\pm\infty$.
- __float2half_rn(NaN) returns NaN.

__host__ __device__ *__half* **`__float2half_ru`**(const float a)

Converts float number to half precision in round-up mode and returns `half` with converted value.

Converts float number a to half precision in round-up mode.

Parameters

a – **[in]** - float. Is only being read.

Returns

half

- a converted to half precision using round-up mode.
- __float2half_ru (± 0) returns ± 0.
- __float2half_ru $(\pm\infty)$ returns $\pm\infty$.
- __float2half_ru(NaN) returns NaN.

__host__ __device__ *__half* **`__float2half_rz`**(const float a)

Converts float number to half precision in round-towards-zero mode and returns `half` with converted value.

Converts float number a to half precision in round-towards-zero mode.

Parameters

a – **[in]** - float. Is only being read.

Returns

half

- a converted to half precision using round-towards-zero mode.
- __float2half_rz (± 0) returns ± 0.
- __float2half_rz $(\pm\infty)$ returns $\pm\infty$.
- __float2half_rz(NaN) returns NaN.

__host__ __device__ *__half2* **`__floats2half2_rn`**(const float a, const float b)

Converts both input floats to half precision in round-to-nearest-even mode and returns `half2` with converted values.

Converts both input floats to half precision in round-to-nearest-even mode and combines the results into one `half2` number. Low 16 bits of the return value correspond to the input a, high 16 bits correspond to the input b.

See also:

__float2half_rn(float) for further details.

Parameters

- **a** – **[in]** - float. Is only being read.
- **b** – **[in]** - float. Is only being read.

Returns

half2

- The `half2` value with corresponding halves equal to the converted input floats.

__host__ __device__ float2 **`__half22float2`**(const *__half2* a)

Converts both halves of `half2` to `float2` and returns the result.

Converts both halves of `half2` input a to `float2` and returns the result.

See also:

__half2float(__half) for further details.

Parameters

a – **[in]** - half2. Is only being read.

Returns

float2

- a converted to `float2`.

__host__ __device__ signed char **`__half2char_rz`**(const *__half* h)

Convert a half to a signed char in round-towards-zero mode.

Convert the half-precision floating-point value h to a signed char integer in round-towards-zero mode. NaN inputs are converted to 0.

Parameters

h – **[in]** - half. Is only being read.

Returns

signed char

- h converted to a signed char using round-towards-zero mode.
- __half2char_rz (± 0) returns 0.
- __half2char_rz $(x), x > 127$ returns SCHAR_MAX = `0x7F`.
- __half2char_rz $(x), x < -128$ returns SCHAR_MIN = `0x80`.
- __half2char_rz(NaN) returns 0.

__host__ __device__ float **`__half2float`**(const *__half* a)

Converts `half` number to float.

Converts half number a to float.

Parameters

a – **[in]** - float. Is only being read.

Returns

float

- a converted to float.
- __half2float (± 0) returns ± 0.
- __half2float $(\pm\infty)$ returns $\pm\infty$.
- __half2float(NaN) returns NaN.

__host__ __device__ *__half2* **__half2half2**(const *__half* a)

Returns `half2` with both halves equal to the input value.

Returns `half2` number with both halves equal to the input a `half` number.

Parameters

a – **[in]** - half. Is only being read.

Returns

half2

- The vector which has both its halves equal to the input a.

__device__ int **__half2int_rd**(const *__half* h)

Convert a half to a signed integer in round-down mode.

Convert the half-precision floating-point value h to a signed integer in round-down mode. NaN inputs are converted to 0.

Parameters

h – **[in]** - half. Is only being read.

Returns

int

- h converted to a signed integer using round-down mode.
- __half2int_rd (± 0) returns 0.
- __half2int_rd $(+\infty)$ returns INT_MAX = `0x7FFFFFFF`.
- __half2int_rd $(-\infty)$ returns INT_MIN = `0x80000000`.
- __half2int_rd(NaN) returns 0.

__device__ int **__half2int_rn**(const *__half* h)

Convert a half to a signed integer in round-to-nearest-even mode.

Convert the half-precision floating-point value h to a signed integer in round-to-nearest-even mode. NaN inputs are converted to 0.

Parameters

h – **[in]** - half. Is only being read.

Returns

int

- h converted to a signed integer using round-to-nearest-even mode.
- __half2int_rn (± 0) returns 0.
- __half2int_rn $(+\infty)$ returns INT_MAX = `0x7FFFFFFF`.
- __half2int_rn $(-\infty)$ returns INT_MIN = `0x80000000`.
- __half2int_rn(NaN) returns 0.

__device__ int **__half2int_ru**(const *__half* h)

Convert a half to a signed integer in round-up mode.

Convert the half-precision floating-point value h to a signed integer in round-up mode. NaN inputs are converted to 0.

Parameters

h – **[in]** - half. Is only being read.

Returns

int

- h converted to a signed integer using round-up mode.
- __half2int_ru (± 0) returns 0.
- __half2int_ru $(+\infty)$ returns INT_MAX = `0x7FFFFFFF`.
- __half2int_ru $(-\infty)$ returns INT_MIN = `0x80000000`.
- __half2int_ru(NaN) returns 0.

__host__ __device__ int **__half2int_rz**(const *__half* h)

Convert a half to a signed integer in round-towards-zero mode.

Convert the half-precision floating-point value h to a signed integer in round-towards-zero mode. NaN inputs are converted to 0.

Parameters

h – **[in]** - half. Is only being read.

Returns

int

- h converted to a signed integer using round-towards-zero mode.
- __half2int_rz (± 0) returns 0.
- __half2int_rz $(+\infty)$ returns INT_MAX = `0x7FFFFFFF`.
- __half2int_rz $(-\infty)$ returns INT_MIN = `0x80000000`.
- __half2int_rz(NaN) returns 0.

__device__ long long int **__half2ll_rd**(const *__half* h)

Convert a half to a signed 64-bit integer in round-down mode.

Convert the half-precision floating-point value h to a signed 64-bit integer in round-down mode. NaN inputs return a long long int with hex value of `0x8000000000000000`.

Parameters

h – **[in]** - half. Is only being read.

Returns

long long int

- h converted to a signed 64-bit integer using round-down mode.
- __half2ll_rd (± 0) returns 0.
- __half2ll_rd $(+\infty)$ returns LLONG_MAX = `0x7FFFFFFFFFFFFFFF`.
- __half2ll_rd $(-\infty)$ returns LLONG_MIN = `0x8000000000000000`.
- __half2ll_rd(NaN) returns `0x8000000000000000`.

__device__ long long int **__half2ll_rn**(const *__half* h)

Convert a half to a signed 64-bit integer in round-to-nearest-even mode.

Convert the half-precision floating-point value h to a signed 64-bit integer in round-to-nearest-even mode. NaN inputs return a long long int with hex value of `0x8000000000000000`.

Parameters

h – **[in]** - half. Is only being read.

Returns

long long int

- h converted to a signed 64-bit integer using round-to-nearest-even mode.
- __half2ll_rn (± 0) returns 0.
- __half2ll_rn $(+\infty)$ returns LLONG_MAX = `0x7FFFFFFFFFFFFFFF`.
- __half2ll_rn $(-\infty)$ returns LLONG_MIN = `0x8000000000000000`.
- __half2ll_rn(NaN) returns `0x8000000000000000`.

__device__ long long int **__half2ll_ru**(const *__half* h)

Convert a half to a signed 64-bit integer in round-up mode.

Convert the half-precision floating-point value h to a signed 64-bit integer in round-up mode. NaN inputs return a long long int with hex value of `0x8000000000000000`.

Parameters

h – **[in]** - half. Is only being read.

Returns

long long int

- h converted to a signed 64-bit integer using round-up mode.
- __half2ll_ru (± 0) returns 0.
- __half2ll_ru $(+\infty)$ returns LLONG_MAX = `0x7FFFFFFFFFFFFFFF`.
- __half2ll_ru $(-\infty)$ returns LLONG_MIN = `0x8000000000000000`.
- __half2ll_ru(NaN) returns `0x8000000000000000`.

__host__ __device__ long long int **__half2ll_rz**(const *__half* h)

Convert a half to a signed 64-bit integer in round-towards-zero mode.

Convert the half-precision floating-point value h to a signed 64-bit integer in round-towards-zero mode. NaN inputs return a long long int with hex value of `0x8000000000000000`.

Parameters

h – **[in]** - half. Is only being read.

Returns

long long int

- h converted to a signed 64-bit integer using round-towards-zero mode.
- __half2ll_rz (± 0) returns 0.
- __half2ll_rz $(+\infty)$ returns LLONG_MAX = `0x7FFFFFFFFFFFFFFF`.
- __half2ll_rz $(-\infty)$ returns LLONG_MIN = `0x8000000000000000`.
- __half2ll_rz(NaN) returns `0x8000000000000000`.

__device__ short int **__half2short_rd**(const *__half* h)

Convert a half to a signed short integer in round-down mode.

Convert the half-precision floating-point value h to a signed short integer in round-down mode. NaN inputs are converted to 0.

Parameters

h – **[in]** - half. Is only being read.

Returns

short int

- h converted to a signed short integer using round-down mode.
- __half2short_rd (± 0) returns 0.
- __half2short_rd $(x), x > 32767$ returns SHRT_MAX = `0x7FFF`.
- __half2short_rd $(x), x < -32768$ returns SHRT_MIN = `0x8000`.
- __half2short_rd(NaN) returns 0.

__device__ short int **__half2short_rn**(const *__half* h)

Convert a half to a signed short integer in round-to-nearest-even mode.

Convert the half-precision floating-point value h to a signed short integer in round-to-nearest-even mode. NaN inputs are converted to 0.

Parameters

h – **[in]** - half. Is only being read.

Returns

short int

- h converted to a signed short integer using round-to-nearest-even mode.
- __half2short_rn (± 0) returns 0.
- __half2short_rn $(x), x > 32767$ returns SHRT_MAX = `0x7FFF`.
- __half2short_rn $(x), x < -32768$ returns SHRT_MIN = `0x8000`.
- __half2short_rn(NaN) returns 0.

__device__ short int **__half2short_ru**(const *__half* h)

Convert a half to a signed short integer in round-up mode.

Convert the half-precision floating-point value h to a signed short integer in round-up mode. NaN inputs are converted to 0.

Parameters

h – **[in]** - half. Is only being read.

Returns

short int

- h converted to a signed short integer using round-up mode.
- __half2short_ru (± 0) returns 0.
- __half2short_ru $(x), x > 32767$ returns SHRT_MAX = `0x7FFF`.
- __half2short_ru $(x), x < -32768$ returns SHRT_MIN = `0x8000`.
- __half2short_ru(NaN) returns 0.

__host__ __device__ short int **__half2short_rz**(const *__half* h)

Convert a half to a signed short integer in round-towards-zero mode.

Convert the half-precision floating-point value h to a signed short integer in round-towards-zero mode. NaN inputs are converted to 0.

Parameters

h – **[in]** - half. Is only being read.

Returns
short int

- h converted to a signed short integer using round-towards-zero mode.
- __half2short_rz (± 0) returns 0.
- __half2short_rz $(x), x > 32767$ returns SHRT_MAX = `0x7FFF`.
- __half2short_rz $(x), x < -32768$ returns SHRT_MIN = `0x8000`.
- __half2short_rz(NaN) returns 0.

__host__ __device__ unsigned char **__half2uchar_rz**(const *__half* h)

Convert a half to an unsigned char in round-towards-zero mode.

Convert the half-precision floating-point value h to an unsigned char in round-towards-zero mode. NaN inputs are converted to 0.

Parameters
h – **[in]** - half. Is only being read.

Returns
unsigned char

- h converted to an unsigned char using round-towards-zero mode.
- __half2uchar_rz (± 0) returns 0.
- __half2uchar_rz $(x), x > 255$ returns UCHAR_MAX = `0xFF`.
- __half2uchar_rz $(x), x < 0.0$ returns 0.
- __half2uchar_rz(NaN) returns 0.

__device__ unsigned int **__half2uint_rd**(const *__half* h)

Convert a half to an unsigned integer in round-down mode.

Convert the half-precision floating-point value h to an unsigned integer in round-down mode. NaN inputs are converted to 0.

Parameters
h – **[in]** - half. Is only being read.

Returns
unsigned int

- h converted to an unsigned integer using round-down mode.
- __half2uint_rd (± 0) returns 0.
- __half2uint_rd $(+\infty)$ returns UINT_MAX = `0xFFFFFFFF`.
- __half2uint_rd $(x), x < 0.0$ returns 0.
- __half2uint_rd(NaN) returns 0.

__device__ unsigned int **__half2uint_rn**(const *__half* h)

Convert a half to an unsigned integer in round-to-nearest-even mode.

Convert the half-precision floating-point value h to an unsigned integer in round-to-nearest-even mode. NaN inputs are converted to 0.

Parameters
h – **[in]** - half. Is only being read.

Returns

unsigned int

- h converted to an unsigned integer using round-to-nearest-even mode.
- __half2uint_rn (± 0) returns 0.
- __half2uint_rn $(+\infty)$ returns UINT_MAX = `0xFFFFFFFF`.
- __half2uint_rn $(x), x < 0.0$ returns 0.
- __half2uint_rn(NaN) returns 0.

__device__ unsigned int **__half2uint_ru**(const *__half* h)

Convert a half to an unsigned integer in round-up mode.

Convert the half-precision floating-point value h to an unsigned integer in round-up mode. NaN inputs are converted to 0.

Parameters

h – **[in]** - half. Is only being read.

Returns

unsigned int

- h converted to an unsigned integer using round-up mode.
- __half2uint_ru (± 0) returns 0.
- __half2uint_ru $(+\infty)$ returns UINT_MAX = `0xFFFFFFFF`.
- __half2uint_ru $(x), x < 0.0$ returns 0.
- __half2uint_ru(NaN) returns 0.

__host__ __device__ unsigned int **__half2uint_rz**(const *__half* h)

Convert a half to an unsigned integer in round-towards-zero mode.

Convert the half-precision floating-point value h to an unsigned integer in round-towards-zero mode. NaN inputs are converted to 0.

Parameters

h – **[in]** - half. Is only being read.

Returns

unsigned int

- h converted to an unsigned integer using round-towards-zero mode.
- __half2uint_rz (± 0) returns 0.
- __half2uint_rz $(+\infty)$ returns UINT_MAX = `0xFFFFFFFF`.
- __half2uint_rz $(x), x < 0.0$ returns 0.
- __half2uint_rz(NaN) returns 0.

__device__ unsigned long long int **__half2ull_rd**(const *__half* h)

Convert a half to an unsigned 64-bit integer in round-down mode.

Convert the half-precision floating-point value h to an unsigned 64-bit integer in round-down mode. NaN inputs return `0x8000000000000000`.

Parameters

h – **[in]** - half. Is only being read.

Returns

unsigned long long int

- h converted to an unsigned 64-bit integer using round-down mode.
- __half2ull_rd (± 0) returns 0.
- __half2ull_rd $(+\infty)$ returns ULLONG_MAX = `0xFFFFFFFFFFFFFFFF`.
- __half2ull_rd $(x), x < 0.0$ returns 0.
- __half2ull_rd(NaN) returns `0x8000000000000000`.

__device__ unsigned long long int **__half2ull_rn**(const *__half* h)

Convert a half to an unsigned 64-bit integer in round-to-nearest-even mode.

Convert the half-precision floating-point value h to an unsigned 64-bit integer in round-to-nearest-even mode. NaN inputs return `0x8000000000000000`.

Parameters

h – **[in]** - half. Is only being read.

Returns

unsigned long long int

- h converted to an unsigned 64-bit integer using round-to-nearest-even mode.
- __half2ull_rn (± 0) returns 0.
- __half2ull_rn $(+\infty)$ returns ULLONG_MAX = `0xFFFFFFFFFFFFFFFF`.
- __half2ull_rn $(x), x < 0.0$ returns 0.
- __half2ull_rn(NaN) returns `0x8000000000000000`.

__device__ unsigned long long int **__half2ull_ru**(const *__half* h)

Convert a half to an unsigned 64-bit integer in round-up mode.

Convert the half-precision floating-point value h to an unsigned 64-bit integer in round-up mode. NaN inputs return `0x8000000000000000`.

Parameters

h – **[in]** - half. Is only being read.

Returns

unsigned long long int

- h converted to an unsigned 64-bit integer using round-up mode.
- __half2ull_ru (± 0) returns 0.
- __half2ull_ru $(+\infty)$ returns ULLONG_MAX = `0xFFFFFFFFFFFFFFFF`.
- __half2ull_ru $(x), x < 0.0$ returns 0.
- __half2ull_ru(NaN) returns `0x8000000000000000`.

__host__ __device__ unsigned long long int **__half2ull_rz**(const *__half* h)

Convert a half to an unsigned 64-bit integer in round-towards-zero mode.

Convert the half-precision floating-point value h to an unsigned 64-bit integer in round-towards-zero mode. NaN inputs return `0x8000000000000000`.

Parameters

h – **[in]** - half. Is only being read.

Returns

unsigned long long int

- h converted to an unsigned 64-bit integer using round-towards-zero mode.
- __half2ull_rz (± 0) returns 0.
- __half2ull_rz $(+\infty)$ returns ULLONG_MAX = `0xFFFFFFFFFFFFFFFF`.
- __half2ull_rz $(x), x < 0.0$ returns 0.
- __half2ull_rz(NaN) returns `0x8000000000000000`.

__device__ unsigned short int **`__half2ushort_rd`**(const *__half* h)

Convert a half to an unsigned short integer in round-down mode.

Convert the half-precision floating-point value h to an unsigned short integer in round-down mode. NaN inputs are converted to 0.

Parameters

h – **[in]** - half. Is only being read.

Returns

unsigned short int

- h converted to an unsigned short integer using round-down mode.
- __half2ushort_rd (± 0) returns 0.
- __half2ushort_rd $(+\infty)$ returns USHRT_MAX = `0xFFFF`.
- __half2ushort_rd $(x), x < 0.0$ returns 0.
- __half2ushort_rd(NaN) returns 0.

__device__ unsigned short int **`__half2ushort_rn`**(const *__half* h)

Convert a half to an unsigned short integer in round-to-nearest-even mode.

Convert the half-precision floating-point value h to an unsigned short integer in round-to-nearest-even mode. NaN inputs are converted to 0.

Parameters

h – **[in]** - half. Is only being read.

Returns

unsigned short int

- h converted to an unsigned short integer using round-to-nearest-even mode.
- __half2ushort_rn (± 0) returns 0.
- __half2ushort_rn $(+\infty)$ returns USHRT_MAX = `0xFFFF`.
- __half2ushort_rn $(x), x < 0.0$ returns 0.
- __half2ushort_rn(NaN) returns 0.

__device__ unsigned short int **`__half2ushort_ru`**(const *__half* h)

Convert a half to an unsigned short integer in round-up mode.

Convert the half-precision floating-point value h to an unsigned short integer in round-up mode. NaN inputs are converted to 0.

Parameters

h – **[in]** - half. Is only being read.

Returns

unsigned short int

- h converted to an unsigned short integer using round-up mode.
- __half2ushort_ru (± 0) returns 0.
- __half2ushort_ru $(+\infty)$ returns USHRT_MAX = `0xFFFF`.
- __half2ushort_ru $(x), x < 0.0$ returns 0.
- __half2ushort_ru(NaN) returns 0.

__host__ __device__ unsigned short int **__half2ushort_rz**(const *__half* h)

Convert a half to an unsigned short integer in round-towards-zero mode.

Convert the half-precision floating-point value h to an unsigned short integer in round-towards-zero mode. NaN inputs are converted to 0.

Parameters

h – **[in]** - half. Is only being read.

Returns

unsigned short int

- h converted to an unsigned short integer using round-towards-zero mode.
- __half2ushort_rz (± 0) returns 0.
- __half2ushort_rz $(+\infty)$ returns USHRT_MAX = `0xFFFF`.
- __half2ushort_rz $(x), x < 0.0$ returns 0.
- __half2ushort_rz(NaN) returns 0.

__host__ __device__ short int **__half_as_short**(const *__half* h)

Reinterprets bits in a `half` as a signed short integer.

Reinterprets the bits in the half-precision floating-point number h as a signed short integer.

Parameters

h – **[in]** - half. Is only being read.

Returns

short int

- The reinterpreted value.

__host__ __device__ unsigned short int **__half_as_ushort**(const *__half* h)

Reinterprets bits in a `half` as an unsigned short integer.

Reinterprets the bits in the half-precision floating-point h as an unsigned short number.

Parameters

h – **[in]** - half. Is only being read.

Returns

unsigned short int

- The reinterpreted value.

__host__ __device__ *__half2* **__halves2half2**(const *__half* a, const *__half* b)

Combines two `half` numbers into one `half2` number.

Combines two input `half` number a and b into one `half2` number. Input a is stored in low 16 bits of the return value, input b is stored in high 16 bits of the return value.

Parameters

- **a** – **[in]** - half. Is only being read.
- **b** – **[in]** - half. Is only being read.

Returns
half2

- The half2 with one half equal to a and the other to b.

__host__ __device__ float **__high2float**(const *__half2* a)

Converts high 16 bits of `half2` to float and returns the result.

Converts high 16 bits of `half2` input a to 32-bit floating-point number and returns the result.

See also:

__half2float(__half) for further details.

Parameters
a – **[in]** - half2. Is only being read.

Returns
float

- The high 16 bits of a converted to float.

__host__ __device__ *__half* **__high2half**(const *__half2* a)

Returns high 16 bits of `half2` input.

Returns high 16 bits of `half2` input a.

Parameters
a – **[in]** - half2. Is only being read.

Returns
half

- The high 16 bits of the input.

__host__ __device__ *__half2* **__high2half2**(const *__half2* a)

Extracts high 16 bits from `half2` input.

Extracts high 16 bits from `half2` input a and returns a new `half2` number which has both halves equal to the extracted bits.

Parameters
a – **[in]** - half2. Is only being read.

Returns
half2

- The half2 with both halves equal to the high 16 bits of the input.

__host__ __device__ *__half2* **__highs2half2**(const *__half2* a, const *__half2* b)

Extracts high 16 bits from each of the two `half2` inputs and combines into one `half2` number.

Extracts high 16 bits from each of the two `half2` inputs and combines into one `half2` number. High 16 bits from input a is stored in low 16 bits of the return value, high 16 bits from input b is stored in high 16 bits of the return value.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns
half2

- The high 16 bits of a and of b.

__host__ __device__ *__half* **__int2half_rd**(const int i)
Convert a signed integer to a half in round-down mode.

Convert the signed integer value `i` to a half-precision floating-point value in round-down mode.

Parameters
i – **[in]** - int. Is only being read.

Returns
half

- `i` converted to half.

__host__ __device__ *__half* **__int2half_rn**(const int i)
Convert a signed integer to a half in round-to-nearest-even mode.

Convert the signed integer value `i` to a half-precision floating-point value in round-to-nearest-even mode.

Parameters
i – **[in]** - int. Is only being read.

Returns
half

- `i` converted to half.

__host__ __device__ *__half* **__int2half_ru**(const int i)
Convert a signed integer to a half in round-up mode.

Convert the signed integer value `i` to a half-precision floating-point value in round-up mode.

Parameters
i – **[in]** - int. Is only being read.

Returns
half

- `i` converted to half.

__host__ __device__ *__half* **__int2half_rz**(const int i)
Convert a signed integer to a half in round-towards-zero mode.

Convert the signed integer value `i` to a half-precision floating-point value in round-towards-zero mode.

Parameters
i – **[in]** - int. Is only being read.

Returns
half

- `i` converted to half.

__device__ *__half2* **__ldca**(const *__half2* *const ptr)

Generates a `ld.global.ca` load instruction.

Parameters

ptr – **[in]** - memory location

Returns

The value pointed by `ptr`

__device__ *__half* **__ldca**(const *__half* *const ptr)

Generates a `ld.global.ca` load instruction.

Parameters

ptr – **[in]** - memory location

Returns

The value pointed by `ptr`

__device__ *__half* **__ldcg**(const *__half* *const ptr)

Generates a `ld.global.cg` load instruction.

Parameters

ptr – **[in]** - memory location

Returns

The value pointed by `ptr`

__device__ *__half2* **__ldcg**(const *__half2* *const ptr)

Generates a `ld.global.cg` load instruction.

Parameters

ptr – **[in]** - memory location

Returns

The value pointed by `ptr`

__device__ *__half* **__ldcs**(const *__half* *const ptr)

Generates a `ld.global.cs` load instruction.

Parameters

ptr – **[in]** - memory location

Returns

The value pointed by `ptr`

__device__ *__half2* **__ldcs**(const *__half2* *const ptr)

Generates a `ld.global.cs` load instruction.

Parameters

ptr – **[in]** - memory location

Returns

The value pointed by `ptr`

__device__ *__half2* **__ldcv**(const *__half2* *const ptr)

Generates a `ld.global.cv` load instruction.

Parameters

ptr – **[in]** - memory location

Returns

The value pointed by `ptr`

__device__ *__half* **__ldcv**(const *__half* *const ptr)

Generates a `ld.global.cv` load instruction.

Parameters

ptr – **[in]** - memory location

Returns

The value pointed by `ptr`

__device__ *__half2* **__ldg**(const *__half2* *const ptr)

Generates a `ld.global.nc` load instruction.

defined(**CUDA_ARCH**) || (**CUDA_ARCH** >= 300)

Parameters

ptr – **[in]** - memory location

Returns

The value pointed by `ptr`

__device__ *__half* **__ldg**(const *__half* *const ptr)

Generates a `ld.global.nc` load instruction.

Parameters

ptr – **[in]** - memory location

Returns

The value pointed by `ptr`

__device__ *__half* **__ldlu**(const *__half* *const ptr)

Generates a `ld.global.lu` load instruction.

Parameters

ptr – **[in]** - memory location

Returns

The value pointed by `ptr`

__device__ *__half2* **__ldlu**(const *__half2* *const ptr)

Generates a `ld.global.lu` load instruction.

Parameters

ptr – **[in]** - memory location

Returns

The value pointed by `ptr`

__host__ __device__ *__half* **__ll2half_rd**(const long long int i)

Convert a signed 64-bit integer to a half in round-down mode.

Convert the signed 64-bit integer value `i` to a half-precision floating-point value in round-down mode.

Parameters

i – **[in]** - long long int. Is only being read.

Returns

half

- `i` converted to half.

__host__ __device__ *__half* **__ll2half_rn**(const long long int i)

Convert a signed 64-bit integer to a half in round-to-nearest-even mode.

Convert the signed 64-bit integer value `i` to a half-precision floating-point value in round-to-nearest-even mode.

Parameters

i – **[in]** - long long int. Is only being read.

Returns

half

- `i` converted to half.

__host__ __device__ *__half* **__ll2half_ru**(const long long int i)

Convert a signed 64-bit integer to a half in round-up mode.

Convert the signed 64-bit integer value `i` to a half-precision floating-point value in round-up mode.

Parameters

i – **[in]** - long long int. Is only being read.

Returns

half

- `i` converted to half.

__host__ __device__ *__half* **__ll2half_rz**(const long long int i)

Convert a signed 64-bit integer to a half in round-towards-zero mode.

Convert the signed 64-bit integer value `i` to a half-precision floating-point value in round-towards-zero mode.

Parameters

i – **[in]** - long long int. Is only being read.

Returns

half

- `i` converted to half.

__host__ __device__ float **__low2float**(const *__half2* a)

Converts low 16 bits of `half2` to float and returns the result.

Converts low 16 bits of `half2` input a to 32-bit floating-point number and returns the result.

See also:

__half2float(__half) for further details.

Parameters

a – **[in]** - half2. Is only being read.

Returns

float

- The low 16 bits of a converted to float.

__host__ __device__ *__half* **__low2half**(const *__half2* a)

Returns low 16 bits of `half2` input.

Returns low 16 bits of `half2` input a.

Parameters

a – **[in]** - half2. Is only being read.

Returns

half

- Returns `half` which contains low 16 bits of the input a.

__host__ __device__ *__half2* **__low2half2**(const *__half2* a)

Extracts low 16 bits from `half2` input.

Extracts low 16 bits from `half2` input a and returns a new `half2` number which has both halves equal to the extracted bits.

Parameters

a – **[in]** - half2. Is only being read.

Returns

half2

- The half2 with both halves equal to the low 16 bits of the input.

__host__ __device__ *__half2* **__lowhigh2highlow**(const *__half2* a)

Swaps both halves of the `half2` input.

Swaps both halves of the `half2` input and returns a new `half2` number with swapped halves.

Parameters

a – **[in]** - half2. Is only being read.

Returns

half2

- a with its halves being swapped.

__host__ __device__ *__half2* **__lows2half2**(const *__half2* a, const *__half2* b)

Extracts low 16 bits from each of the two `half2` inputs and combines into one `half2` number.

Extracts low 16 bits from each of the two `half2` inputs and combines into one `half2` number. Low 16 bits from input a is stored in low 16 bits of the return value, low 16 bits from input b is stored in high 16 bits of the return value.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns

half2

- The low 16 bits of a and of b.

__device__ *__half* **__shfl_down_sync**(const unsigned int mask, const *__half* var, const unsigned int delta, const int width = warpSize)

Exchange a variable between threads within a warp.

Copy from a thread with higher ID relative to the caller.

Calculates a source thread ID by adding `delta` to the caller's thread ID. The value of `var` held by the resulting thread ID is returned: this has the effect of shifting `var` down the warp by `delta` threads. If the `width` is less than `warpSize`, then each subsection of the warp behaves as a separate entity with a starting logical thread ID of 0. Similarly to the *__shfl_up_sync()*, the ID number of the source thread will not wrap around the value of `width` and the upper `delta` threads will remain unchanged. Threads may only read data from another thread which is actively participating in the `__shfl_*sync()` command. If the target thread is inactive, the retrieved value is undefined.

Note: For more details about this function, see the Warp Shuffle Functions section in the CUDA C++ Programming Guide.

Parameters

- **`mask`** – **[in]** - unsigned int. Is only being read.
  - Indicates the threads participating in the call.
  - A bit, representing the thread's lane ID, must be set for each participating thread to ensure they are properly converged before the intrinsic is executed by the hardware.
  - Each calling thread must have its own bit set in the `mask` and all non-exited threads named in `mask` must execute the same intrinsic with the same `mask`, or the result is undefined.
- **`var`** – **[in]** - half. Is only being read.
- **`delta`** – **[in]** - unsigned int. Is only being read.
- **`width`** – **[in]** - int. Is only being read.

Returns

Returns the 2-byte word referenced by `var` from the source thread ID as `half`.

__device__ *__half2* **`__shfl_down_sync`**(const unsigned int mask, const *__half2* var, const unsigned int delta, const int width = warpSize)

Exchange a variable between threads within a warp.

Copy from a thread with higher ID relative to the caller.

Calculates a source thread ID by adding `delta` to the caller's thread ID. The value of `var` held by the resulting thread ID is returned: this has the effect of shifting `var` down the warp by `delta` threads. If the `width` is less than `warpSize`, then each subsection of the warp behaves as a separate entity with a starting logical thread ID of 0. Similarly to the *__shfl_up_sync()*, the ID number of the source thread will not wrap around the value of `width` and the upper `delta` threads will remain unchanged. Threads may only read data from another thread which is actively participating in the `__shfl_*sync()` command. If the target thread is inactive, the retrieved value is undefined.

Note: For more details about this function, see the Warp Shuffle Functions section in the CUDA C++ Programming Guide.

Parameters

- **`mask`** – **[in]** - unsigned int. Is only being read.
  - Indicates the threads participating in the call.

  - A bit, representing the thread's lane ID, must be set for each participating thread to ensure they are properly converged before the intrinsic is executed by the hardware.
  - Each calling thread must have its own bit set in the `mask` and all non-exited threads named in `mask` must execute the same intrinsic with the same `mask`, or the result is undefined.
- **var** – **[in]** - half2. Is only being read.
- **delta** – **[in]** - unsigned int. Is only being read.
- **width** – **[in]** - int. Is only being read.

Returns

Returns the 4-byte word referenced by `var` from the source thread ID as `half2`.

__device__ *__half2* **__shfl_sync**(const unsigned int mask, const *__half2* var, const int srcLane, const int width = warpSize)

Exchange a variable between threads within a warp.

Direct copy from indexed thread.

Returns the value of `var` held by the thread whose ID is given by `srcLane`. If the `width` is less than `warpSize`, then each subsection of the warp behaves as a separate entity with a starting logical thread ID of 0. If `srcLane` is outside the range [0:width-1], the value returned corresponds to the value of `var` held by the `srcLane` modulo `width` (i.e. within the same subsection). `width` must have a value which is a power of 2; results are undefined if `width` is not a power of 2, or is a number greater than `warpSize`. Threads may only read data from another thread which is actively participating in the `__shfl_*sync`() command. If the target thread is inactive, the retrieved value is undefined.

Note: For more details about this function, see the Warp Shuffle Functions section in the CUDA C++ Programming Guide.

Parameters

- **mask** – **[in]** - unsigned int. Is only being read.
  - Indicates the threads participating in the call.
  - A bit, representing the thread's lane ID, must be set for each participating thread to ensure they are properly converged before the intrinsic is executed by the hardware.
  - Each calling thread must have its own bit set in the `mask` and all non-exited threads named in `mask` must execute the same intrinsic with the same `mask`, or the result is undefined.
- **var** – **[in]** - half2. Is only being read.
- **srcLane** – **[in]** - int. Is only being read.
- **width** – **[in]** - int. Is only being read.

Returns

Returns the 4-byte word referenced by `var` from the source thread ID as `half2`.

__device__ *__half* **__shfl_sync**(const unsigned int mask, const *__half* var, const int srcLane, const int width = warpSize)

Exchange a variable between threads within a warp.

Direct copy from indexed thread.

Returns the value of `var` held by the thread whose ID is given by `srcLane`. If the `width` is less than `warpSize`, then each subsection of the warp behaves as a separate entity with a starting logical thread ID of 0. If `srcLane` is outside the range [0:width-1], the value returned corresponds to the value of `var` held by the `srcLane` modulo `width` (i.e. within the same subsection). `width` must have a value which is a power of 2; results are undefined if `width` is not a power of 2, or is a number greater than `warpSize`. Threads may only read data from another thread which is actively participating in the `__shfl_*sync`() command. If the target thread is inactive, the retrieved value is undefined.

Note: For more details about this function, see the Warp Shuffle Functions section in the CUDA C++ Programming Guide.

Parameters

- **mask** – **[in]** - unsigned int. Is only being read.
  - Indicates the threads participating in the call.
  - A bit, representing the thread's lane ID, must be set for each participating thread to ensure they are properly converged before the intrinsic is executed by the hardware.
  - Each calling thread must have its own bit set in the `mask` and all non-exited threads named in `mask` must execute the same intrinsic with the same `mask`, or the result is undefined.
- **var** – **[in]** - half. Is only being read.
- **srcLane** – **[in]** - int. Is only being read.
- **width** – **[in]** - int. Is only being read.

Returns

Returns the 2-byte word referenced by `var` from the source thread ID as `half`.

__device__ *__half2* **__shfl_up_sync**(const unsigned int mask, const *__half2* var, const unsigned int delta, const int width = warpSize)

Exchange a variable between threads within a warp.

Copy from a thread with lower ID relative to the caller.

Calculates a source thread ID by subtracting `delta` from the caller's lane ID. The value of `var` held by the resulting lane ID is returned: in effect, `var` is shifted up the warp by `delta` threads. If the `width` is less than `warpSize`, then each subsection of the warp behaves as a separate entity with a starting logical thread ID of 0. The source thread index will not wrap around the value of `width`, so effectively the lower `delta` threads will be unchanged. `width` must have a value which is a power of 2; results are undefined if `width` is not a power of 2, or is a number greater than `warpSize`. Threads may only read data from another thread which is actively participating in the `__shfl_*sync`() command. If the target thread is inactive, the retrieved value is undefined.

Note: For more details about this function, see the Warp Shuffle Functions section in the CUDA C++ Programming Guide.

Parameters

- **mask** – **[in]** - unsigned int. Is only being read.
  - Indicates the threads participating in the call.
  - A bit, representing the thread's lane ID, must be set for each participating thread to ensure they are properly converged before the intrinsic is executed by the hardware.
  - Each calling thread must have its own bit set in the `mask` and all non-exited threads named in `mask` must execute the same intrinsic with the same `mask`, or the result is undefined.
- **var** – **[in]** - half2. Is only being read.
- **delta** – **[in]** - unsigned int. Is only being read.
- **width** – **[in]** - int. Is only being read.

Returns

Returns the 4-byte word referenced by `var` from the source thread ID as `half2`.

__device__ *__half* **__shfl_up_sync**(const unsigned int mask, const *__half* var, const unsigned int delta, const int width = warpSize)

Exchange a variable between threads within a warp.

Copy from a thread with lower ID relative to the caller.

Calculates a source thread ID by subtracting `delta` from the caller's lane ID. The value of `var` held by the resulting lane ID is returned: in effect, `var` is shifted up the warp by `delta` threads. If the `width` is less than `warpSize`, then each subsection of the warp behaves as a separate entity with a starting logical thread ID of 0. The source thread index will not wrap around the value of `width`, so effectively the lower `delta` threads will be unchanged. `width` must have a value which is a power of 2; results are undefined if `width` is not a power of 2, or is a number greater than `warpSize`. Threads may only read data from another thread which is actively participating in the `__shfl_*sync`() command. If the target thread is inactive, the retrieved value is undefined.

Note: For more details about this function, see the Warp Shuffle Functions section in the CUDA C++ Programming Guide.

Parameters

- **mask** – **[in]** - unsigned int. Is only being read.
  - Indicates the threads participating in the call.
  - A bit, representing the thread's lane ID, must be set for each participating thread to ensure they are properly converged before the intrinsic is executed by the hardware.

    - Each calling thread must have its own bit set in the `mask` and all non-exited threads named in `mask` must execute the same intrinsic with the same `mask`, or the result is undefined.
- **var** – **[in]** - half. Is only being read.
- **delta** – **[in]** - unsigned int. Is only being read.
- **width** – **[in]** - int. Is only being read.

Returns

Returns the 2-byte word referenced by `var` from the source thread ID as `half`.

__device__ *__half2* **__shfl_xor_sync**(const unsigned int mask, const *__half2* var, const int laneMask, const int width = warpSize)

Exchange a variable between threads within a warp.

Copy from a thread based on bitwise XOR of own thread ID.

Calculates a source thread ID by performing a bitwise XOR of the caller's thread ID with `laneMask`: the value of `var` held by the resulting thread ID is returned. If the `width` is less than `warpSize`, then each group of `width` consecutive threads are able to access elements from earlier groups of threads, however if they attempt to access elements from later groups of threads their own value of `var` will be returned. This mode implements a butterfly addressing pattern such as is used in tree reduction and broadcast. Threads may only read data from another thread which is actively participating in the `__shfl_*sync`() command. If the target thread is inactive, the retrieved value is undefined.

Note: For more details about this function, see the Warp Shuffle Functions section in the CUDA C++ Programming Guide.

Parameters

- **mask** – **[in]** - unsigned int. Is only being read.
    - Indicates the threads participating in the call.
    - A bit, representing the thread's lane ID, must be set for each participating thread to ensure they are properly converged before the intrinsic is executed by the hardware.
    - Each calling thread must have its own bit set in the `mask` and all non-exited threads named in `mask` must execute the same intrinsic with the same `mask`, or the result is undefined.
- **var** – **[in]** - half2. Is only being read.
- **laneMask** – **[in]** - int. Is only being read.
- **width** – **[in]** - int. Is only being read.

Returns

Returns the 4-byte word referenced by `var` from the source thread ID as `half2`.

__device__ *__half* **__shfl_xor_sync**(const unsigned int mask, const *__half* var, const int laneMask, const int width = warpSize)

Exchange a variable between threads within a warp.

Copy from a thread based on bitwise XOR of own thread ID.

Calculates a source thread ID by performing a bitwise XOR of the caller's thread ID with `laneMask`: the value of `var` held by the resulting thread ID is returned. If the `width` is less than `warpSize`, then each group of `width` consecutive threads are able to access elements from earlier groups of threads, however if they attempt to access elements from later groups of threads their own value of `var` will be returned. This mode implements a butterfly addressing pattern such as is used in tree reduction and broadcast. Threads may only read data from another thread which is actively participating in the `__shfl_*sync()` command. If the target thread is inactive, the retrieved value is undefined.

Note: For more details about this function, see the Warp Shuffle Functions section in the CUDA C++ Programming Guide.

Parameters

- **`mask`** – **[in]** - unsigned int. Is only being read.
  - Indicates the threads participating in the call.
  - A bit, representing the thread's lane ID, must be set for each participating thread to ensure they are properly converged before the intrinsic is executed by the hardware.
  - Each calling thread must have its own bit set in the `mask` and all non-exited threads named in `mask` must execute the same intrinsic with the same `mask`, or the result is undefined.
- **`var`** – **[in]** - half. Is only being read.
- **`laneMask`** – **[in]** - int. Is only being read.
- **`width`** – **[in]** - int. Is only being read.

Returns

Returns the 2-byte word referenced by `var` from the source thread ID as `half`.

__host__ __device__ *__half* **`__short2half_rd`**(const short int i)

Convert a signed short integer to a half in round-down mode.

Convert the signed short integer value `i` to a half-precision floating-point value in round-down mode.

Parameters

`i` – **[in]** - short int. Is only being read.

Returns

half

- `i` converted to half.

__host__ __device__ *__half* **`__short2half_rn`**(const short int i)

Convert a signed short integer to a half in round-to-nearest-even mode.

Convert the signed short integer value `i` to a half-precision floating-point value in round-to-nearest-even mode.

Parameters

`i` – **[in]** - short int. Is only being read.

Returns
half

- `i` converted to half.

__host__ __device__ *__half* **`__short2half_ru`**(const short int i)
Convert a signed short integer to a half in round-up mode.

Convert the signed short integer value `i` to a half-precision floating-point value in round-up mode.

Parameters
i – **[in]** - short int. Is only being read.

Returns
half

- `i` converted to half.

__host__ __device__ *__half* **`__short2half_rz`**(const short int i)
Convert a signed short integer to a half in round-towards-zero mode.

Convert the signed short integer value `i` to a half-precision floating-point value in round-towards-zero mode.

Parameters
i – **[in]** - short int. Is only being read.

Returns
half

- `i` converted to half.

__host__ __device__ *__half* **`__short_as_half`**(const short int i)
Reinterprets bits in a signed short integer as a `half`.

Reinterprets the bits in the signed short integer `i` as a half-precision floating-point number.

Parameters
i – **[in]** - short int. Is only being read.

Returns
half

- The reinterpreted value.

__device__ void **`__stcg`**(*__half2* *const ptr, const *__half2* value)
Generates a `st.global.cg` store instruction.

Parameters

- **`ptr`** – **[out]** - memory location
- **`value`** – **[in]** - the value to be stored

__device__ void **`__stcg`**(*__half* *const ptr, const *__half* value)
Generates a `st.global.cg` store instruction.

Parameters

- **`ptr`** – **[out]** - memory location
- **`value`** – **[in]** - the value to be stored

__device__ void **__stcs**(*__half2* *const ptr, const *__half2* value)

Generates a `st.global.cs` store instruction.

Parameters

- **ptr** – **[out]** - memory location
- **value** – **[in]** - the value to be stored

__device__ void **__stcs**(*__half* *const ptr, const *__half* value)

Generates a `st.global.cs` store instruction.

Parameters

- **ptr** – **[out]** - memory location
- **value** – **[in]** - the value to be stored

__device__ void **__stwb**(*__half2* *const ptr, const *__half2* value)

Generates a `st.global.wb` store instruction.

Parameters

- **ptr** – **[out]** - memory location
- **value** – **[in]** - the value to be stored

__device__ void **__stwb**(*__half* *const ptr, const *__half* value)

Generates a `st.global.wb` store instruction.

Parameters

- **ptr** – **[out]** - memory location
- **value** – **[in]** - the value to be stored

__device__ void **__stwt**(*__half* *const ptr, const *__half* value)

Generates a `st.global.wt` store instruction.

Parameters

- **ptr** – **[out]** - memory location
- **value** – **[in]** - the value to be stored

__device__ void **__stwt**(*__half2* *const ptr, const *__half2* value)

Generates a `st.global.wt` store instruction.

Parameters

- **ptr** – **[out]** - memory location
- **value** – **[in]** - the value to be stored

__host__ __device__ *__half* **__uint2half_rd**(const unsigned int i)

Convert an unsigned integer to a half in round-down mode.

Convert the unsigned integer value `i` to a half-precision floating-point value in round-down mode.

Parameters
: **i** – **[in]** - unsigned int. Is only being read.

Returns
: half

- `i` converted to half.

__host__ __device__ *__half* **`__uint2half_rn`**(const unsigned int i)

Convert an unsigned integer to a half in round-to-nearest-even mode.

Convert the unsigned integer value `i` to a half-precision floating-point value in round-to-nearest-even mode.

Parameters
: **i** – **[in]** - unsigned int. Is only being read.

Returns
: half

- `i` converted to half.

__host__ __device__ *__half* **`__uint2half_ru`**(const unsigned int i)

Convert an unsigned integer to a half in round-up mode.

Convert the unsigned integer value `i` to a half-precision floating-point value in round-up mode.

Parameters
: **i** – **[in]** - unsigned int. Is only being read.

Returns
: half

- `i` converted to half.

__host__ __device__ *__half* **`__uint2half_rz`**(const unsigned int i)

Convert an unsigned integer to a half in round-towards-zero mode.

Convert the unsigned integer value `i` to a half-precision floating-point value in round-towards-zero mode.

Parameters
: **i** – **[in]** - unsigned int. Is only being read.

Returns
: half

- `i` converted to half.

__host__ __device__ *__half* **`__ull2half_rd`**(const unsigned long long int i)

Convert an unsigned 64-bit integer to a half in round-down mode.

Convert the unsigned 64-bit integer value `i` to a half-precision floating-point value in round-down mode.

Parameters
: **i** – **[in]** - unsigned long long int. Is only being read.

Returns
: half

- `i` converted to half.

__host__ __device__ *__half* **__ull2half_rn**(const unsigned long long int i)

Convert an unsigned 64-bit integer to a half in round-to-nearest-even mode.

Convert the unsigned 64-bit integer value `i` to a half-precision floating-point value in round-to-nearest-even mode.

Parameters

i – **[in]** - unsigned long long int. Is only being read.

Returns

half

- `i` converted to half.

__host__ __device__ *__half* **__ull2half_ru**(const unsigned long long int i)

Convert an unsigned 64-bit integer to a half in round-up mode.

Convert the unsigned 64-bit integer value `i` to a half-precision floating-point value in round-up mode.

Parameters

i – **[in]** - unsigned long long int. Is only being read.

Returns

half

- `i` converted to half.

__host__ __device__ *__half* **__ull2half_rz**(const unsigned long long int i)

Convert an unsigned 64-bit integer to a half in round-towards-zero mode.

Convert the unsigned 64-bit integer value `i` to a half-precision floating-point value in round-towards-zero mode.

Parameters

i – **[in]** - unsigned long long int. Is only being read.

Returns

half

- `i` converted to half.

__host__ __device__ *__half* **__ushort2half_rd**(const unsigned short int i)

Convert an unsigned short integer to a half in round-down mode.

Convert the unsigned short integer value `i` to a half-precision floating-point value in round-down mode.

Parameters

i – **[in]** - unsigned short int. Is only being read.

Returns

half

- `i` converted to half.

__host__ __device__ *__half* **__ushort2half_rn**(const unsigned short int i)

Convert an unsigned short integer to a half in round-to-nearest-even mode.

Convert the unsigned short integer value `i` to a half-precision floating-point value in round-to-nearest-even mode.

Parameters

i – **[in]** - unsigned short int. Is only being read.

Returns

half

- i converted to half.

__host__ __device__ *__half* **__ushort2half_ru**(const unsigned short int i)

Convert an unsigned short integer to a half in round-up mode.

Convert the unsigned short integer value i to a half-precision floating-point value in round-up mode.

Parameters

i – **[in]** - unsigned short int. Is only being read.

Returns

half

- i converted to half.

__host__ __device__ *__half* **__ushort2half_rz**(const unsigned short int i)

Convert an unsigned short integer to a half in round-towards-zero mode.

Convert the unsigned short integer value i to a half-precision floating-point value in round-towards-zero mode.

Parameters

i – **[in]** - unsigned short int. Is only being read.

Returns

half

- i converted to half.

__host__ __device__ *__half* **__ushort_as_half**(const unsigned short int i)

Reinterprets bits in an unsigned short integer as a half.

Reinterprets the bits in the unsigned short integer i as a half-precision floating-point number.

Parameters

i – **[in]** - unsigned short int. Is only being read.

Returns

half

- The reinterpreted value.

__host__ __device__ *__half2* **make_half2**(const *__half* x, const *__half* y)

Vector function, combines two *__half* numbers into one *__half2* number.

Combines two input *__half* number x and y into one *__half2* number. Input x is stored in low 16 bits of the return value, input y is stored in high 16 bits of the return value.

Parameters

- **x** – **[in]** - half. Is only being read.
- **y** – **[in]** - half. Is only being read.

Returns

__half2

- The *__half2* vector with one half equal to x and the other to y.

4.6. Half2 Arithmetic Functions

To use these functions, include the header file `cuda_fp16.h` in your program.

Functions

__host__ __device__ __half2 *__h2div*(const __half2 a, const __half2 b)
Performs `half2` vector division in round-to-nearest-even mode.

__host__ __device__ __half2 *__habs2*(const __half2 a)
Calculates the absolute value of both halves of the input `half2` number and returns the result.

__host__ __device__ __half2 *__hadd2*(const __half2 a, const __half2 b)
Performs `half2` vector addition in round-to-nearest-even mode.

__host__ __device__ __half2 *__hadd2_rn*(const __half2 a, const __half2 b)
Performs `half2` vector addition in round-to-nearest-even mode.

__host__ __device__ __half2 *__hadd2_sat*(const __half2 a, const __half2 b)
Performs `half2` vector addition in round-to-nearest-even mode, with saturation to [0.0, 1.0].

__device__ __half2 *__hcmadd*(const __half2 a, const __half2 b, const __half2 c)
Performs fast complex multiply-accumulate.

__device__ __half2 *__hfma2*(const __half2 a, const __half2 b, const __half2 c)
Performs `half2` vector fused multiply-add in round-to-nearest-even mode.

__device__ __half2 *__hfma2_relu*(const __half2 a, const __half2 b, const __half2 c)
Performs `half2` vector fused multiply-add in round-to-nearest-even mode with relu saturation.

__device__ __half2 *__hfma2_sat*(const __half2 a, const __half2 b, const __half2 c)
Performs `half2` vector fused multiply-add in round-to-nearest-even mode, with saturation to [0.0, 1.0].

__host__ __device__ __half2 *__hmul2*(const __half2 a, const __half2 b)
Performs `half2` vector multiplication in round-to-nearest-even mode.

__host__ __device__ __half2 *__hmul2_rn*(const __half2 a, const __half2 b)
Performs `half2` vector multiplication in round-to-nearest-even mode.

__host__ __device__ __half2 *__hmul2_sat*(const __half2 a, const __half2 b)
Performs `half2` vector multiplication in round-to-nearest-even mode, with saturation to [0.0, 1.0].

__host__ __device__ __half2 *__hneg2*(const __half2 a)
Negates both halves of the input `half2` number and returns the result.

__host__ __device__ __half2 *__hsub2*(const __half2 a, const __half2 b)
Performs `half2` vector subtraction in round-to-nearest-even mode.

__host__ __device__ __half2 *__hsub2_rn*(const __half2 a, const __half2 b)
Performs `half2` vector subtraction in round-to-nearest-even mode.

__host__ __device__ __half2 *__hsub2_sat*(const __half2 a, const __half2 b)
Performs `half2` vector subtraction in round-to-nearest-even mode, with saturation to [0.0, 1.0].

__device__ __half2 *atomicAdd*(__half2 *const address, const __half2 val)
Vector add `val` to the value stored at `address` in global or shared memory, and writes this value back to `address` .

__host__ __device__ __half2 *operator(const __half2 &lh, const __half2 &rh)**
Performs packed `half` multiplication operation.

__host__ __device__ __half2 & *operator*=*(__half2 &lh, const __half2 &rh)
Performs packed `half` compound assignment with multiplication operation.

__host__ __device__ __half2 *operator+*(const __half2 &h)
Implements packed `half` unary plus operator, returns input value.

__host__ __device__ __half2 *operator+*(const __half2 &lh, const __half2 &rh)
Performs packed `half` addition operation.

__host__ __device__ __half2 *operator++*(__half2 &h, const int ignored)
Performs packed `half` postfix increment operation.

__host__ __device__ __half2 & *operator++*(__half2 &h)
Performs packed `half` prefix increment operation.

__host__ __device__ __half2 & *operator+=*(__half2 &lh, const __half2 &rh)
Performs packed `half` compound assignment with addition operation.

__host__ __device__ __half2 *operator-*(const __half2 &h)
Implements packed `half` unary minus operator.

__host__ __device__ __half2 *operator-*(const __half2 &lh, const __half2 &rh)
Performs packed `half` subtraction operation.

__host__ __device__ __half2 & *operator--*(__half2 &h)
Performs packed `half` prefix decrement operation.

__host__ __device__ __half2 *operator--*(__half2 &h, const int ignored)
Performs packed `half` postfix decrement operation.

__host__ __device__ __half2 & *operator-=*(__half2 &lh, const __half2 &rh)
Performs packed `half` compound assignment with subtraction operation.

__host__ __device__ __half2 *operator/*(const __half2 &lh, const __half2 &rh)
Performs packed `half` division operation.

__host__ __device__ __half2 & *operator/=*(__half2 &lh, const __half2 &rh)
Performs packed `half` compound assignment with division operation.

4.6.1. Functions

__host__ __device__ *__half2* **`__h2div`**(const *__half2* a, const *__half2* b)

Performs `half2` vector division in round-to-nearest-even mode.

Divides `half2` input vector a by input vector b in round-to-nearest-even mode.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns

half2

- The elementwise division of a with b.

__host__ __device__ *__half2* **__habs2**(const *__half2* a)

Calculates the absolute value of both halves of the input `half2` number and returns the result.

Calculates the absolute value of both halves of the input `half2` number and returns the result.

See also:

__habs(__half) for further details.

Parameters

a – **[in]** - half2. Is only being read.

Returns

half2

- Returns a with the absolute value of both halves.

__host__ __device__ *__half2* **__hadd2**(const *__half2* a, const *__half2* b)

Performs `half2` vector addition in round-to-nearest-even mode.

Performs `half2` vector add of inputs a and b, in round-to-nearest-even mode.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns

half2

- The sum of vectors a and b.

__host__ __device__ *__half2* **__hadd2_rn**(const *__half2* a, const *__half2* b)

Performs `half2` vector addition in round-to-nearest-even mode.

Performs `half2` vector add of inputs a and b, in round-to-nearest-even mode. Prevents floating-point contractions of mul+add into fma.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns

half2

- The sum of vectors a and b.

__host__ __device__ *__half2* **__hadd2_sat**(const *__half2* a, const *__half2* b)

Performs `half2` vector addition in round-to-nearest-even mode, with saturation to [0.0, 1.0].

Performs `half2` vector add of inputs a and b, in round-to-nearest-even mode, and clamps the results to range [0.0, 1.0]. NaN results are flushed to +0.0.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns

half2

- The sum of a and b, with respect to saturation.

__device__ *__half2* **__hcmadd**(const *__half2* a, const *__half2* b, const *__half2* c)

Performs fast complex multiply-accumulate.

Interprets vector `half2` input pairs a, b, and c as complex numbers in `half` precision: (a.x + I*a.y), (b.x + I*b.y), (c.x + I*c.y) and performs complex multiply-accumulate operation: a*b + c in a simple way: ((a.x*b.x + c.x) - a.y*b.y) + I*((a.x*b.y + c.y) + a.y*b.x)

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.
- **c** – **[in]** - half2. Is only being read.

Returns

half2

- The result of complex multiply-accumulate operation on complex numbers a, b, and c
- *__half2* result = __hcmadd(a, b, c) is numerically in agreement with:
- result.x = __hfma(-a.y, b.y, __hfma(a.x, b.x, c.x))
- result.y = __hfma(a.y, b.x, __hfma(a.x, b.y, c.y))

__device__ *__half2* **__hfma2**(const *__half2* a, const *__half2* b, const *__half2* c)

Performs `half2` vector fused multiply-add in round-to-nearest-even mode.

Performs `half2` vector multiply on inputs a and b, then performs a `half2` vector add of the result with c, rounding the result once in round-to-nearest-even mode.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.
- **c** – **[in]** - half2. Is only being read.

Returns

half2

- The result of elementwise fused multiply-add operation on vectors a, b, and c.

__device__ *__half2* **__hfma2_relu**(const *__half2* a, const *__half2* b, const *__half2* c)

Performs `half2` vector fused multiply-add in round-to-nearest-even mode with relu saturation.

Performs `half2` vector multiply on inputs a and b, then performs a `half2` vector add of the result with c, rounding the result once in round-to-nearest-even mode. Then negative result is clamped to 0. NaN result is converted to canonical NaN.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

- **c** – **[in]** - half2. Is only being read.

Returns
half2

- The result of elementwise fused multiply-add operation on vectors a, b, and c with relu saturation.

__device__ *__half2* **__hfma2_sat**(const *__half2* a, const *__half2* b, const *__half2* c)

Performs `half2` vector fused multiply-add in round-to-nearest-even mode, with saturation to [0.0, 1.0].

Performs `half2` vector multiply on inputs a and b, then performs a `half2` vector add of the result with c, rounding the result once in round-to-nearest-even mode, and clamps the results to range [0.0, 1.0]. NaN results are flushed to +0.0.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.
- **c** – **[in]** - half2. Is only being read.

Returns
half2

- The result of elementwise fused multiply-add operation on vectors a, b, and c, with respect to saturation.

__host__ __device__ *__half2* **__hmul2**(const *__half2* a, const *__half2* b)

Performs `half2` vector multiplication in round-to-nearest-even mode.

Performs `half2` vector multiplication of inputs a and b, in round-to-nearest-even mode.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns
half2

- The result of elementwise multiplying the vectors a and b.

__host__ __device__ *__half2* **__hmul2_rn**(const *__half2* a, const *__half2* b)

Performs `half2` vector multiplication in round-to-nearest-even mode.

Performs `half2` vector multiplication of inputs a and b, in round-to-nearest-even mode. Prevents floating-point contractions of mul+add or sub into fma.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns
half2

- The result of elementwise multiplying the vectors a and b.

__host__ __device__ *__half2* **__hmul2_sat**(const *__half2* a, const *__half2* b)

Performs `half2` vector multiplication in round-to-nearest-even mode, with saturation to [0.0, 1.0].

Performs `half2` vector multiplication of inputs `a` and `b`, in round-to-nearest-even mode, and clamps the results to range [0.0, 1.0]. NaN results are flushed to +0.0.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns

half2

- The result of elementwise multiplication of vectors `a` and `b`, with respect to saturation.

__host__ __device__ *__half2* **__hneg2**(const *__half2* a)

Negates both halves of the input `half2` number and returns the result.

Negates both halves of the input `half2` number `a` and returns the result.

See also:

__hneg(__half) for further details.

Parameters

a – **[in]** - half2. Is only being read.

Returns

half2

- Returns `a` with both halves negated.

__host__ __device__ *__half2* **__hsub2**(const *__half2* a, const *__half2* b)

Performs `half2` vector subtraction in round-to-nearest-even mode.

Subtracts `half2` input vector `b` from input vector `a` in round-to-nearest-even mode.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns

half2

- The subtraction of vector `b` from `a`.

__host__ __device__ *__half2* **__hsub2_rn**(const *__half2* a, const *__half2* b)

Performs `half2` vector subtraction in round-to-nearest-even mode.

Subtracts `half2` input vector `b` from input vector `a` in round-to-nearest-even mode. Prevents floating-point contractions of mul+sub into fma.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns
half2

- The subtraction of vector b from a.

__host__ __device__ *__half2* **__hsub2_sat**(const *__half2* a, const *__half2* b)

Performs `half2` vector subtraction in round-to-nearest-even mode, with saturation to [0.0, 1.0].

Subtracts `half2` input vector b from input vector a in round-to-nearest-even mode, and clamps the results to range [0.0, 1.0]. NaN results are flushed to +0.0.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns
half2

- The subtraction of vector b from a, with respect to saturation.

__device__ *__half2* **atomicAdd**(*__half2* *const address, const *__half2* val)

Vector add `val` to the value stored at `address` in global or shared memory, and writes this value back to `address`.

The atomicity of the add operation is guaranteed separately for each of the two *`__half`* elements; the entire *`__half2`* is not guaranteed to be atomic as a single 32-bit access.

The location of `address` must be in global or shared memory. This operation has undefined behavior otherwise. This operation is natively supported by devices of compute capability 6.x and higher, older devices use emulation path.

Note: For more details about this function, see the Atomic Functions section in the CUDA C++ Programming Guide.

Parameters

- **`address`** – **[in]** - half2*. An address in global or shared memory.
- **`val`** – **[in]** - half2. The value to be added.

Returns
half2

- The old value read from `address`.

__host__ __device__ *__half2* **operator***(const *__half2* &lh, const *__half2* &rh)

Performs packed `half` multiplication operation.

See also:

__hmul2(__half2, __half2)

__host__ __device__ *__half2* &**operator*=**(*__half2* &lh, const *__half2* &rh)

Performs packed `half` compound assignment with multiplication operation.

See also:

__hmul2(__half2, __half2)

__host__ __device__ *__half2* **operator+**(const *__half2* &h)

Implements packed `half` unary plus operator, returns input value.

__host__ __device__ *__half2* **operator+**(const *__half2* &lh, const *__half2* &rh)

Performs packed `half` addition operation.

See also:

__hadd2(__half2, __half2)

__host__ __device__ *__half2* **operator++**(*__half2* &h, const int ignored)

Performs packed `half` postfix increment operation.

See also:

__hadd2(__half2, __half2)

__host__ __device__ *__half2* &**operator++**(*__half2* &h)

Performs packed `half` prefix increment operation.

See also:

__hadd2(__half2, __half2)

__host__ __device__ *__half2* &**operator+=**(*__half2* &lh, const *__half2* &rh)

Performs packed `half` compound assignment with addition operation.

See also:

__hadd2(__half2, __half2)

__host__ __device__ *__half2* **operator-**(const *__half2* &h)

Implements packed `half` unary minus operator.

See also:

__hneg2(__half2)

__host__ __device__ *__half2* **operator-**(const *__half2* &lh, const *__half2* &rh)

Performs packed `half` subtraction operation.

See also:

__hsub2(__half2, __half2)

__host__ __device__ *__half2* &**operator--**(*__half2* &h)

Performs packed `half` prefix decrement operation.

See also:

__hsub2(__half2, __half2)

__host__ __device__ *__half2* **operator--**(*__half2* &h, const int ignored)

Performs packed `half` postfix decrement operation.

See also:

__hsub2(__half2, __half2)

__host__ __device__ *__half2* &**operator-=**(*__half2* &lh, const *__half2* &rh)

Performs packed `half` compound assignment with subtraction operation.

See also:

__hsub2(__half2, __half2)

__host__ __device__ *__half2* **operator/**(const *__half2* &lh, const *__half2* &rh)

Performs packed `half` division operation.

See also:

__h2div(__half2, __half2)

__host__ __device__ *__half2* &**operator/=**(*__half2* &lh, const *__half2* &rh)

Performs packed `half` compound assignment with division operation.

See also:

__h2div(__half2, __half2)

4.7. Half2 Comparison Functions

To use these functions, include the header file `cuda_fp16.h` in your program.

Functions

__host__ __device__ bool *__hbeq2*(const __half2 a, const __half2 b)
Performs `half2` vector if-equal comparison and returns boolean true if both `half` results are true, boolean false otherwise.

__host__ __device__ bool *__hbequ2*(const __half2 a, const __half2 b)
Performs `half2` vector unordered if-equal comparison and returns boolean true if both `half` results are true, boolean false otherwise.

__host__ __device__ bool *__hbge2*(const __half2 a, const __half2 b)
Performs `half2` vector greater-equal comparison and returns boolean true if both `half` results are true, boolean false otherwise.

__host__ __device__ bool *__hbgeu2*(const __half2 a, const __half2 b)
Performs `half2` vector unordered greater-equal comparison and returns boolean true if both `half` results are true, boolean false otherwise.

__host__ __device__ bool *__hbgt2*(const __half2 a, const __half2 b)
Performs `half2` vector greater-than comparison and returns boolean true if both `half` results are true, boolean false otherwise.

__host__ __device__ bool *__hbgtu2*(const __half2 a, const __half2 b)
Performs `half2` vector unordered greater-than comparison and returns boolean true if both `half` results are true, boolean false otherwise.

__host__ __device__ bool *__hble2*(const __half2 a, const __half2 b)
Performs `half2` vector less-equal comparison and returns boolean true if both `half` results are true, boolean false otherwise.

__host__ __device__ bool *__hbleu2*(const __half2 a, const __half2 b)
Performs `half2` vector unordered less-equal comparison and returns boolean true if both `half` results are true, boolean false otherwise.

__host__ __device__ bool *__hblt2*(const __half2 a, const __half2 b)
Performs `half2` vector less-than comparison and returns boolean true if both `half` results are true, boolean false otherwise.

__host__ __device__ bool *__hbltu2*(const __half2 a, const __half2 b)
Performs `half2` vector unordered less-than comparison and returns boolean true if both `half` results are true, boolean false otherwise.

__host__ __device__ bool *__hbne2*(const __half2 a, const __half2 b)
Performs `half2` vector not-equal comparison and returns boolean true if both `half` results are true, boolean false otherwise.

__host__ __device__ bool *__hbneu2*(const __half2 a, const __half2 b)
Performs `half2` vector unordered not-equal comparison and returns boolean true if both `half` results are true, boolean false otherwise.

__host__ __device__ __half2 *__heq2*(const __half2 a, const __half2 b)
Performs half2 vector if-equal comparison.

__host__ __device__ unsigned int *__heq2_mask*(const __half2 a, const __half2 b)
Performs half2 vector if-equal comparison.

__host__ __device__ __half2 *__hequ2*(const __half2 a, const __half2 b)
Performs `half2` vector unordered if-equal comparison.

__host__ __device__ unsigned int *__hequ2_mask*(const __half2 a, const __half2 b)
Performs `half2` vector unordered if-equal comparison.

__host__ __device__ __half2 *__hge2*(const __half2 a, const __half2 b)
Performs `half2` vector greater-equal comparison.

__host__ __device__ unsigned int *__hge2_mask*(const __half2 a, const __half2 b)
Performs `half2` vector greater-equal comparison.

__host__ __device__ __half2 *__hgeu2*(const __half2 a, const __half2 b)
Performs `half2` vector unordered greater-equal comparison.

__host__ __device__ unsigned int *__hgeu2_mask*(const __half2 a, const __half2 b)
Performs `half2` vector unordered greater-equal comparison.

__host__ __device__ __half2 *__hgt2*(const __half2 a, const __half2 b)
Performs `half2` vector greater-than comparison.

__host__ __device__ unsigned int *__hgt2_mask*(const __half2 a, const __half2 b)
Performs `half2` vector greater-than comparison.

__host__ __device__ __half2 *__hgtu2*(const __half2 a, const __half2 b)
Performs `half2` vector unordered greater-than comparison.

__host__ __device__ unsigned int *__hgtu2_mask*(const __half2 a, const __half2 b)
Performs `half2` vector unordered greater-than comparison.

__host__ __device__ __half2 *__hisnan2*(const __half2 a)
Determine whether `half2` argument is a NaN.

__host__ __device__ __half2 *__hle2*(const __half2 a, const __half2 b)
Performs `half2` vector less-equal comparison.

__host__ __device__ unsigned int *__hle2_mask*(const __half2 a, const __half2 b)
Performs `half2` vector less-equal comparison.

__host__ __device__ __half2 *__hleu2*(const __half2 a, const __half2 b)
Performs `half2` vector unordered less-equal comparison.

__host__ __device__ unsigned int *__hleu2_mask*(const __half2 a, const __half2 b)
Performs `half2` vector unordered less-equal comparison.

__host__ __device__ __half2 *__hlt2*(const __half2 a, const __half2 b)
Performs `half2` vector less-than comparison.

__host__ __device__ unsigned int *__hlt2_mask*(const __half2 a, const __half2 b)
Performs `half2` vector less-than comparison.

__host__ __device__ __half2 *__hltu2*(const __half2 a, const __half2 b)
Performs `half2` vector unordered less-than comparison.

__host__ __device__ unsigned int *__hltu2_mask*(const __half2 a, const __half2 b)
Performs `half2` vector unordered less-than comparison.

__host__ __device__ __half2 *__hmax2*(const __half2 a, const __half2 b)
Calculates `half2` vector maximum of two inputs.

__host__ __device__ __half2 *__hmax2_nan*(const __half2 a, const __half2 b)
Calculates `half2` vector maximum of two inputs, NaNs pass through.

__host__ __device__ __half2 *__hmin2*(const __half2 a, const __half2 b)
Calculates `half2` vector minimum of two inputs.

__host__ __device__ __half2 *__hmin2_nan*(const __half2 a, const __half2 b)
Calculates `half2` vector minimum of two inputs, NaNs pass through.

__host__ __device__ __half2 *__hne2*(const __half2 a, const __half2 b)
Performs `half2` vector not-equal comparison.

__host__ __device__ unsigned int *__hne2_mask*(const __half2 a, const __half2 b)
Performs `half2` vector not-equal comparison.

__host__ __device__ __half2 *__hneu2*(const __half2 a, const __half2 b)
Performs `half2` vector unordered not-equal comparison.

__host__ __device__ unsigned int *__hneu2_mask*(const __half2 a, const __half2 b)
Performs `half2` vector unordered not-equal comparison.

__host__ __device__ bool *operator!=*(const __half2 &lh, const __half2 &rh)
Performs packed `half` unordered compare not-equal operation.

__host__ __device__ bool *operator<*(const __half2 &lh, const __half2 &rh)
Performs packed `half` ordered less-than compare operation.

__host__ __device__ bool *operator<=*(const __half2 &lh, const __half2 &rh)
Performs packed `half` ordered less-or-equal compare operation.

__host__ __device__ bool *operator==*(const __half2 &lh, const __half2 &rh)
Performs packed `half` ordered compare equal operation.

__host__ __device__ bool *operator>*(const __half2 &lh, const __half2 &rh)
Performs packed `half` ordered greater-than compare operation.

__host__ __device__ bool *operator>=*(const __half2 &lh, const __half2 &rh)
Performs packed `half` ordered greater-or-equal compare operation.

4.7.1. Functions

__host__ __device__ bool **__hbeq2**(const *__half2* a, const *__half2* b)

Performs `half2` vector if-equal comparison and returns boolean true if both `half` results are true, boolean false otherwise.

Performs `half2` vector if-equal comparison of inputs a and b. The bool result is set to true only if both `half` if-equal comparisons evaluate to true, or false otherwise. NaN inputs generate false results.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns

bool

- true if both `half` results of if-equal comparison of vectors a and b are true;
- false otherwise.

__host__ __device__ bool **__hbequ2**(const *__half2* a, const *__half2* b)

Performs `half2` vector unordered if-equal comparison and returns boolean true if both `half` results are true, boolean false otherwise.

Performs `half2` vector if-equal comparison of inputs a and b. The bool result is set to true only if both `half` if-equal comparisons evaluate to true, or false otherwise. NaN inputs generate true results.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns

bool

- true if both `half` results of unordered if-equal comparison of vectors a and b are true;
- false otherwise.

__host__ __device__ bool **__hbge2**(const *__half2* a, const *__half2* b)

Performs `half2` vector greater-equal comparison and returns boolean true if both `half` results are true, boolean false otherwise.

Performs `half2` vector greater-equal comparison of inputs a and b. The bool result is set to true only if both `half` greater-equal comparisons evaluate to true, or false otherwise. NaN inputs generate false results.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns

bool

- true if both `half` results of greater-equal comparison of vectors a and b are true;
- false otherwise.

__host__ __device__ bool **__hbgeu2**(const *__half2* a, const *__half2* b)

Performs `half2` vector unordered greater-equal comparison and returns boolean true if both `half` results are true, boolean false otherwise.

Performs `half2` vector greater-equal comparison of inputs a and b. The bool result is set to true only if both `half` greater-equal comparisons evaluate to true, or false otherwise. NaN inputs generate true results.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns

bool

- true if both `half` results of unordered greater-equal comparison of vectors a and b are true;
- false otherwise.

__host__ __device__ bool **__hbgt2**(const *__half2* a, const *__half2* b)

Performs `half2` vector greater-than comparison and returns boolean true if both `half` results are true, boolean false otherwise.

Performs `half2` vector greater-than comparison of inputs a and b. The bool result is set to true only if both `half` greater-than comparisons evaluate to true, or false otherwise. NaN inputs generate false results.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns

bool

- true if both `half` results of greater-than comparison of vectors a and b are true;
- false otherwise.

__host__ __device__ bool **__hbgtu2**(const *__half2* a, const *__half2* b)

Performs `half2` vector unordered greater-than comparison and returns boolean true if both `half` results are true, boolean false otherwise.

Performs `half2` vector greater-than comparison of inputs a and b. The bool result is set to true only if both `half` greater-than comparisons evaluate to true, or false otherwise. NaN inputs generate true results.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns

bool

- true if both `half` results of unordered greater-than comparison of vectors a and b are true;
- false otherwise.

__host__ __device__ bool **__hble2**(const *__half2* a, const *__half2* b)

Performs `half2` vector less-equal comparison and returns boolean true if both `half` results are true, boolean false otherwise.

Performs `half2` vector less-equal comparison of inputs a and b. The bool result is set to true only if both `half` less-equal comparisons evaluate to true, or false otherwise. NaN inputs generate false results.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns

bool

- true if both `half` results of less-equal comparison of vectors a and b are true;

- false otherwise.

__host__ __device__ bool **__hbleu2**(const *__half2* a, const *__half2* b)

Performs `half2` vector unordered less-equal comparison and returns boolean true if both `half` results are true, boolean false otherwise.

Performs `half2` vector less-equal comparison of inputs a and b. The bool result is set to true only if both `half` less-equal comparisons evaluate to true, or false otherwise. NaN inputs generate true results.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns

bool

- true if both `half` results of unordered less-equal comparison of vectors a and b are true;
- false otherwise.

__host__ __device__ bool **__hblt2**(const *__half2* a, const *__half2* b)

Performs `half2` vector less-than comparison and returns boolean true if both `half` results are true, boolean false otherwise.

Performs `half2` vector less-than comparison of inputs a and b. The bool result is set to true only if both `half` less-than comparisons evaluate to true, or false otherwise. NaN inputs generate false results.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns

bool

- true if both `half` results of less-than comparison of vectors a and b are true;
- false otherwise.

__host__ __device__ bool **__hbltu2**(const *__half2* a, const *__half2* b)

Performs `half2` vector unordered less-than comparison and returns boolean true if both `half` results are true, boolean false otherwise.

Performs `half2` vector less-than comparison of inputs a and b. The bool result is set to true only if both `half` less-than comparisons evaluate to true, or false otherwise. NaN inputs generate true results.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns

bool

- true if both `half` results of unordered less-than comparison of vectors a and b are true;
- false otherwise.

__host__ __device__ bool **__hbne2** (const *__half2* a, const *__half2* b)

Performs `half2` vector not-equal comparison and returns boolean true if both `half` results are true, boolean false otherwise.

Performs `half2` vector not-equal comparison of inputs a and b. The bool result is set to true only if both `half` not-equal comparisons evaluate to true, or false otherwise. NaN inputs generate false results.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns

bool

- true if both `half` results of not-equal comparison of vectors a and b are true,
- false otherwise.

__host__ __device__ bool **__hbneu2** (const *__half2* a, const *__half2* b)

Performs `half2` vector unordered not-equal comparison and returns boolean true if both `half` results are true, boolean false otherwise.

Performs `half2` vector not-equal comparison of inputs a and b. The bool result is set to true only if both `half` not-equal comparisons evaluate to true, or false otherwise. NaN inputs generate true results.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns

bool

- true if both `half` results of unordered not-equal comparison of vectors a and b are true;
- false otherwise.

__host__ __device__ *__half2* **__heq2** (const *__half2* a, const *__half2* b)

Performs half2 vector if-equal comparison.

Performs `half2` vector if-equal comparison of inputs a and b. The corresponding `half` results are set to 1.0 for true, or 0.0 for false. NaN inputs generate false results.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns

half2

- The vector result of if-equal comparison of vectors a and b.

__host__ __device__ unsigned int **__heq2_mask**(const *__half2* a, const *__half2* b)

Performs half2 vector if-equal comparison.

Performs `half2` vector if-equal comparison of inputs a and b. The corresponding `unsigned` bits are set to `0xFFFF` for true, or `0x0` for false. NaN inputs generate false results.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns

unsigned int

- The vector mask result of if-equal comparison of vectors a and b.

__host__ __device__ *__half2* **__hequ2**(const *__half2* a, const *__half2* b)

Performs `half2` vector unordered if-equal comparison.

Performs `half2` vector if-equal comparison of inputs a and b. The corresponding `half` results are set to 1.0 for true, or 0.0 for false. NaN inputs generate true results.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns

half2

- The vector result of unordered if-equal comparison of vectors a and b.

__host__ __device__ unsigned int **__hequ2_mask**(const *__half2* a, const *__half2* b)

Performs `half2` vector unordered if-equal comparison.

Performs `half2` vector if-equal comparison of inputs a and b. The corresponding `unsigned` bits are set to `0xFFFF` for true, or `0x0` for false. NaN inputs generate true results.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns

unsigned int

- The vector mask result of unordered if-equal comparison of vectors a and b.

__host__ __device__ *__half2* **__hge2**(const *__half2* a, const *__half2* b)

Performs `half2` vector greater-equal comparison.

Performs `half2` vector greater-equal comparison of inputs a and b. The corresponding `half` results are set to 1.0 for true, or 0.0 for false. NaN inputs generate false results.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns

half2

- The vector result of greater-equal comparison of vectors a and b.

__host__ __device__ unsigned int **__hge2_mask**(const *__half2* a, const *__half2* b)

Performs `half2` vector greater-equal comparison.

Performs `half2` vector greater-equal comparison of inputs a and b. The corresponding `unsigned` bits are set to `0xFFFF` for true, or `0x0` for false. NaN inputs generate false results.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns

unsigned int

- The vector mask result of greater-equal comparison of vectors a and b.

__host__ __device__ *__half2* **__hgeu2**(const *__half2* a, const *__half2* b)

Performs `half2` vector unordered greater-equal comparison.

Performs `half2` vector greater-equal comparison of inputs a and b. The corresponding `half` results are set to 1.0 for true, or 0.0 for false. NaN inputs generate true results.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns

half2

- The `half2` vector result of unordered greater-equal comparison of vectors a and b.

__host__ __device__ unsigned int **__hgeu2_mask**(const *__half2* a, const *__half2* b)

Performs `half2` vector unordered greater-equal comparison.

Performs `half2` vector greater-equal comparison of inputs a and b. The corresponding `unsigned` bits are set to `0xFFFF` for true, or `0x0` for false. NaN inputs generate true results.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns

unsigned int

- The vector mask result of unordered greater-equal comparison of vectors a and b.

__host__ __device__ *__half2* **__hgt2**(const *__half2* a, const *__half2* b)

Performs `half2` vector greater-than comparison.

Performs `half2` vector greater-than comparison of inputs a and b. The corresponding `half` results are set to 1.0 for true, or 0.0 for false. NaN inputs generate false results.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns

half2

- The vector result of greater-than comparison of vectors a and b.

__host__ __device__ unsigned int **__hgt2_mask**(const *__half2* a, const *__half2* b)

Performs `half2` vector greater-than comparison.

Performs `half2` vector greater-than comparison of inputs a and b. The corresponding `unsigned` bits are set to `0xFFFF` for true, or `0x0` for false. NaN inputs generate false results.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns

unsigned int

- The vector mask result of greater-than comparison of vectors a and b.

__host__ __device__ *__half2* **__hgtu2**(const *__half2* a, const *__half2* b)

Performs `half2` vector unordered greater-than comparison.

Performs `half2` vector greater-than comparison of inputs a and b. The corresponding `half` results are set to 1.0 for true, or 0.0 for false. NaN inputs generate true results.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns

half2

- The `half2` vector result of unordered greater-than comparison of vectors a and b.

__host__ __device__ unsigned int **__hgtu2_mask**(const *__half2* a, const *__half2* b)

Performs `half2` vector unordered greater-than comparison.

Performs `half2` vector greater-than comparison of inputs a and b. The corresponding `unsigned` bits are set to `0xFFFF` for true, or `0x0` for false. NaN inputs generate true results.

Parameters

- **a** – **[in]** - half2. Is only being read.

- **b** – **[in]** - half2. Is only being read.

Returns

unsigned int

- The vector mask result of unordered greater-than comparison of vectors a and b.

__host__ __device__ *__half2* **__hisnan2**(const *__half2* a)

Determine whether `half2` argument is a NaN.

Determine whether each half of input `half2` number a is a NaN.

Parameters

a – **[in]** - half2. Is only being read.

Returns

half2

- The half2 with the corresponding `half` results set to 1.0 for NaN, 0.0 otherwise.

__host__ __device__ *__half2* **__hle2**(const *__half2* a, const *__half2* b)

Performs `half2` vector less-equal comparison.

Performs `half2` vector less-equal comparison of inputs a and b. The corresponding `half` results are set to 1.0 for true, or 0.0 for false. NaN inputs generate false results.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns

half2

- The `half2` result of less-equal comparison of vectors a and b.

__host__ __device__ unsigned int **__hle2_mask**(const *__half2* a, const *__half2* b)

Performs `half2` vector less-equal comparison.

Performs `half2` vector less-equal comparison of inputs a and b. The corresponding `unsigned` bits are set to `0xFFFF` for true, or `0x0` for false. NaN inputs generate false results.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns

unsigned int

- The vector mask result of less-equal comparison of vectors a and b.

__host__ __device__ *__half2* **__hleu2**(const *__half2* a, const *__half2* b)

Performs `half2` vector unordered less-equal comparison.

Performs `half2` vector less-equal comparison of inputs a and b. The corresponding `half` results are set to 1.0 for true, or 0.0 for false. NaN inputs generate true results.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns

half2

- The vector result of unordered less-equal comparison of vectors a and b.

__host__ __device__ unsigned int **`__hleu2_mask`**(const *__half2* a, const *__half2* b)

Performs `half2` vector unordered less-equal comparison.

Performs `half2` vector less-equal comparison of inputs a and b. The corresponding `unsigned` bits are set to `0xFFFF` for true, or `0x0` for false. NaN inputs generate true results.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns

unsigned int

- The vector mask result of unordered less-equal comparison of vectors a and b.

__host__ __device__ *__half2* **`__hlt2`**(const *__half2* a, const *__half2* b)

Performs `half2` vector less-than comparison.

Performs `half2` vector less-than comparison of inputs a and b. The corresponding `half` results are set to 1.0 for true, or 0.0 for false. NaN inputs generate false results.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns

half2

- The half2 vector result of less-than comparison of vectors a and b.

__host__ __device__ unsigned int **`__hlt2_mask`**(const *__half2* a, const *__half2* b)

Performs `half2` vector less-than comparison.

Performs `half2` vector less-than comparison of inputs a and b. The corresponding `unsigned` bits are set to `0xFFFF` for true, or `0x0` for false. NaN inputs generate false results.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns

unsigned int

- The vector mask result of less-than comparison of vectors a and b.

__host__ __device__ *__half2* **__hltu2**(const *__half2* a, const *__half2* b)

Performs `half2` vector unordered less-than comparison.

Performs `half2` vector less-than comparison of inputs a and b. The corresponding `half` results are set to 1.0 for true, or 0.0 for false. NaN inputs generate true results.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns

half2

- The vector result of unordered less-than comparison of vectors a and b.

__host__ __device__ unsigned int **__hltu2_mask**(const *__half2* a, const *__half2* b)

Performs `half2` vector unordered less-than comparison.

Performs `half2` vector less-than comparison of inputs a and b. The corresponding `unsigned` bits are set to `0xFFFF` for true, or `0x0` for false. NaN inputs generate true results.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns

unsigned int

- The vector mask result of unordered less-than comparison of vectors a and b.

__host__ __device__ *__half2* **__hmax2**(const *__half2* a, const *__half2* b)

Calculates `half2` vector maximum of two inputs.

Calculates `half2` vector max(a, b). Elementwise `half` operation is defined as (a > b) ? a : b.

- If either of inputs is NaN, the other input is returned.
- If both inputs are NaNs, then canonical NaN is returned.
- If values of both inputs are 0.0, then +0.0 > -0.0
- The result of elementwise maximum of vectors a and b

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns

half2

__host__ __device__ *__half2* **__hmax2_nan**(const *__half2* a, const *__half2* b)

Calculates `half2` vector maximum of two inputs, NaNs pass through.

Calculates `half2` vector max(a, b). Elementwise `half` operation is defined as (a > b) ? a : b.

- If either of inputs is NaN, then canonical NaN is returned.

- If values of both inputs are 0.0, then +0.0 > -0.0
- The result of elementwise maximum of vectors a and b, with NaNs pass through

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns
half2

__host__ __device__ *__half2* **__hmin2**(const *__half2* a, const *__half2* b)

Calculates `half2` vector minimum of two inputs.

Calculates `half2` vector min(a, b). Elementwise `half` operation is defined as (a < b) ? a : b.

- If either of inputs is NaN, the other input is returned.
- If both inputs are NaNs, then canonical NaN is returned.
- If values of both inputs are 0.0, then +0.0 > -0.0
- The result of elementwise minimum of vectors a and b

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns
half2

__host__ __device__ *__half2* **__hmin2_nan**(const *__half2* a, const *__half2* b)

Calculates `half2` vector minimum of two inputs, NaNs pass through.

Calculates `half2` vector min(a, b). Elementwise `half` operation is defined as (a < b) ? a : b.

- If either of inputs is NaN, then canonical NaN is returned.
- If values of both inputs are 0.0, then +0.0 > -0.0
- The result of elementwise minimum of vectors a and b, with NaNs pass through

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns
half2

__host__ __device__ *__half2* **__hne2**(const *__half2* a, const *__half2* b)

Performs `half2` vector not-equal comparison.

Performs `half2` vector not-equal comparison of inputs a and b. The corresponding `half` results are set to 1.0 for true, or 0.0 for false. NaN inputs generate false results.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns
half2

- The vector result of not-equal comparison of vectors a and b.

__host__ __device__ unsigned int **__hne2_mask**(const *__half2* a, const *__half2* b)

Performs `half2` vector not-equal comparison.

Performs `half2` vector not-equal comparison of inputs a and b. The corresponding `unsigned` bits are set to `0xFFFF` for true, or `0x0` for false. NaN inputs generate false results.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns
unsigned int

- The vector mask result of not-equal comparison of vectors a and b.

__host__ __device__ *__half2* **__hneu2**(const *__half2* a, const *__half2* b)

Performs `half2` vector unordered not-equal comparison.

Performs `half2` vector not-equal comparison of inputs a and b. The corresponding `half` results are set to 1.0 for true, or 0.0 for false. NaN inputs generate true results.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns
half2

- The vector result of unordered not-equal comparison of vectors a and b.

__host__ __device__ unsigned int **__hneu2_mask**(const *__half2* a, const *__half2* b)

Performs `half2` vector unordered not-equal comparison.

Performs `half2` vector not-equal comparison of inputs a and b. The corresponding `unsigned` bits are set to `0xFFFF` for true, or `0x0` for false. NaN inputs generate true results.

Parameters

- **a** – **[in]** - half2. Is only being read.
- **b** – **[in]** - half2. Is only being read.

Returns
unsigned int

- The vector mask result of unordered not-equal comparison of vectors a and b.

__host__ __device__ bool **operator!=**(const *__half2* &lh, const *__half2* &rh)

Performs packed `half` unordered compare not-equal operation.

See also:

__hbneu2(__half2, __half2)

__host__ __device__ bool **operator<**(const *__half2* &lh, const *__half2* &rh)

Performs packed `half` ordered less-than compare operation.

See also:

__hblt2(__half2, __half2)

__host__ __device__ bool **operator<=**(const *__half2* &lh, const *__half2* &rh)

Performs packed `half` ordered less-or-equal compare operation.

See also:

__hble2(__half2, __half2)

__host__ __device__ bool **operator==**(const *__half2* &lh, const *__half2* &rh)

Performs packed `half` ordered compare equal operation.

See also:

__hbeq2(__half2, __half2)

__host__ __device__ bool **operator>**(const *__half2* &lh, const *__half2* &rh)

Performs packed `half` ordered greater-than compare operation.

See also:

__hbgt2(__half2, __half2)

__host__ __device__ bool **operator>=**(const *__half2* &lh, const *__half2* &rh)

Performs packed `half` ordered greater-or-equal compare operation.

See also:

__hbge2(__half2, __half2)

4.8. Half2 Math Functions

To use these functions, include the header file `cuda_fp16.h` in your program.

Functions

__device__ __half2 *h2ceil*(const __half2 h)
Calculate `half2` vector ceiling of the input argument.

__device__ __half2 *h2cos*(const __half2 a)
Calculates `half2` vector cosine in round-to-nearest-even mode.

__device__ __half2 *h2exp*(const __half2 a)
Calculates `half2` vector exponential function in round-to-nearest-even mode.

__device__ __half2 *h2exp10*(const __half2 a)
Calculates `half2` vector decimal exponential function in round-to-nearest-even mode.

__device__ __half2 *h2exp2*(const __half2 a)
Calculates `half2` vector binary exponential function in round-to-nearest-even mode.

__device__ __half2 *h2floor*(const __half2 h)
Calculate the largest integer less than or equal to h .

__device__ __half2 *h2log*(const __half2 a)
Calculates `half2` vector natural logarithm in round-to-nearest-even mode.

__device__ __half2 *h2log10*(const __half2 a)
Calculates `half2` vector decimal logarithm in round-to-nearest-even mode.

__device__ __half2 *h2log2*(const __half2 a)
Calculates `half2` vector binary logarithm in round-to-nearest-even mode.

__device__ __half2 *h2rcp*(const __half2 a)
Calculates `half2` vector reciprocal in round-to-nearest-even mode.

__device__ __half2 *h2rint*(const __half2 h)
Round input to nearest integer value in half-precision floating-point number.

__device__ __half2 *h2rsqrt*(const __half2 a)
Calculates `half2` vector reciprocal square root in round-to-nearest-even mode.

__device__ __half2 *h2sin*(const __half2 a)
Calculates `half2` vector sine in round-to-nearest-even mode.

__device__ __half2 *h2sqrt*(const __half2 a)
Calculates `half2` vector square root in round-to-nearest-even mode.

__device__ __half2 *h2tanh*(const __half2 a)
Calculates `half2` vector hyperbolic tangent function in round-to-nearest-even mode.

__device__ __half2 *h2tanh_approx*(const __half2 a)
Calculates `half2` vector approximate hyperbolic tangent function.

__device__ __half2 *h2trunc*(const __half2 h)
Truncate `half2` vector input argument to the integral part.

4.8.1. Functions

__device__ *__half2* **h2ceil**(const *__half2* h)

Calculate `half2` vector ceiling of the input argument.

For each component of vector h compute the smallest integer value not less than h.

See also:

hceil(__half) for further details.

Parameters

h – **[in]** - half2. Is only being read.

Returns

half2

- The vector of smallest integers not less than h.

__device__ *__half2* **h2cos**(const *__half2* a)

Calculates `half2` vector cosine in round-to-nearest-even mode.

Calculates `half2` cosine of input vector a in round-to-nearest-even mode.

See also:

hcos(__half) for further details.

Parameters

a – **[in]** - half2. Is only being read.

Returns

half2

- The elementwise cosine on vector a.

__device__ *__half2* **h2exp**(const *__half2* a)

Calculates `half2` vector exponential function in round-to-nearest-even mode.

Calculates `half2` exponential function of input vector a in round-to-nearest-even mode.

See also:

hexp(__half) for further details.

Parameters

a – **[in]** - half2. Is only being read.

Returns

half2

- The elementwise exponential function on vector a.

__device__ *__half2* **h2exp10**(const *__half2* a)

Calculates `half2` vector decimal exponential function in round-to-nearest-even mode.

Calculates `half2` decimal exponential function of input vector a in round-to-nearest-even mode.

See also:

hexp10(__half) for further details.

Parameters
a – **[in]** - half2. Is only being read.

Returns
half2

- The elementwise decimal exponential function on vector a.

__device__ *__half2* **h2exp2**(const *__half2* a)

Calculates `half2` vector binary exponential function in round-to-nearest-even mode.

Calculates `half2` binary exponential function of input vector a in round-to-nearest-even mode.

See also:

hexp2(__half) for further details.

Parameters
a – **[in]** - half2. Is only being read.

Returns
half2

- The elementwise binary exponential function on vector a.

__device__ *__half2* **h2floor**(const *__half2* h)

Calculate the largest integer less than or equal to h.

For each component of vector h calculate the largest integer value which is less than or equal to h.

See also:

hfloor(__half) for further details.

Parameters
h – **[in]** - half2. Is only being read.

Returns
half2

- The vector of largest integers which is less than or equal to h.

__device__ *__half2* **h2log**(const *__half2* a)

Calculates `half2` vector natural logarithm in round-to-nearest-even mode.

Calculates `half2` natural logarithm of input vector a in round-to-nearest-even mode.

See also:

hlog(__half) for further details.

Parameters
a – **[in]** - half2. Is only being read.

Returns
half2

- The elementwise natural logarithm on vector a.

__device__ *__half2* **h2log10**(const *__half2* a)

Calculates `half2` vector decimal logarithm in round-to-nearest-even mode.

Calculates `half2` decimal logarithm of input vector a in round-to-nearest-even mode.

See also:

hlog10(__half) for further details.

Parameters

a – **[in]** - half2. Is only being read.

Returns

half2

- The elementwise decimal logarithm on vector a.

__device__ *__half2* **h2log2**(const *__half2* a)

Calculates `half2` vector binary logarithm in round-to-nearest-even mode.

Calculates `half2` binary logarithm of input vector a in round-to-nearest-even mode.

See also:

hlog2(__half) for further details.

Parameters

a – **[in]** - half2. Is only being read.

Returns

half2

- The elementwise binary logarithm on vector a.

__device__ *__half2* **h2rcp**(const *__half2* a)

Calculates `half2` vector reciprocal in round-to-nearest-even mode.

Calculates `half2` reciprocal of input vector a in round-to-nearest-even mode.

See also:

hrcp(__half) for further details.

Parameters

a – **[in]** - half2. Is only being read.

Returns

half2

- The elementwise reciprocal on vector a.

__device__ *__half2* **h2rint**(const *__half2* h)

Round input to nearest integer value in half-precision floating-point number.

Round each component of `half2` vector h to the nearest integer value in half-precision floating-point format, with halfway cases rounded to the nearest even integer value.

See also:

hrint(__half) for further details.

Parameters
h – **[in]** - half2. Is only being read.

Returns
half2

- The vector of rounded integer values.

__device__ *__half2* **h2rsqrt**(const *__half2* a)

Calculates `half2` vector reciprocal square root in round-to-nearest-even mode.

Calculates `half2` reciprocal square root of input vector a in round-to-nearest-even mode.

See also:

hrsqrt(__half) for further details.

Parameters
a – **[in]** - half2. Is only being read.

Returns
half2

- The elementwise reciprocal square root on vector a.

__device__ *__half2* **h2sin**(const *__half2* a)

Calculates `half2` vector sine in round-to-nearest-even mode.

Calculates `half2` sine of input vector a in round-to-nearest-even mode.

See also:

hsin(__half) for further details.

Parameters
a – **[in]** - half2. Is only being read.

Returns
half2

- The elementwise sine on vector a.

__device__ *__half2* **h2sqrt**(const *__half2* a)

Calculates `half2` vector square root in round-to-nearest-even mode.

Calculates `half2` square root of input vector a in round-to-nearest-even mode.

See also:

hsqrt(__half) for further details.

Parameters
a – **[in]** - half2. Is only being read.

Returns
half2

- The elementwise square root on vector a.

__device__ *__half2* **h2tanh**(const *__half2* a)

Calculates `half2` vector hyperbolic tangent function in round-to-nearest-even mode.

Calculates `half2` hyperbolic tangent function of input vector a in round-to-nearest-even mode.

See also:

htanh(__half) for further details.

Parameters

a – **[in]** - half2. Is only being read.

Returns

half2

- The elementwise hyperbolic tangent function on vector a.

__device__ *__half2* **h2tanh_approx**(const *__half2* a)

Calculates `half2` vector approximate hyperbolic tangent function.

Calculates `half2` approximate hyperbolic tangent function of input vector a. This operation uses HW acceleration on devices of compute capability 7.5 and higher.

See also:

htanh_approx(__half) for further details.

Parameters

a – **[in]** - half2. Is only being read.

Returns

half2

- The elementwise approximate hyperbolic tangent function on vector a.

__device__ *__half2* **h2trunc**(const *__half2* h)

Truncate `half2` vector input argument to the integral part.

Round each component of vector h to the largest integer value that does not exceed h in magnitude.

See also:

htrunc(__half) for further details.

Parameters

h – **[in]** - half2. Is only being read.

Returns

half2

- The truncated h.

Groups

Half Arithmetic Constants
To use these constants, include the header file `cuda_fp16.h` in your program.

Half Arithmetic Functions
To use these functions, include the header file `cuda_fp16.h` in your program.

Half Comparison Functions
To use these functions, include the header file `cuda_fp16.h` in your program.

Half Math Functions
To use these functions, include the header file `cuda_fp16.h` in your program.

Half Precision Conversion and Data Movement
To use these functions, include the header file `cuda_fp16.h` in your program.

Half2 Arithmetic Functions
To use these functions, include the header file `cuda_fp16.h` in your program.

Half2 Comparison Functions
To use these functions, include the header file `cuda_fp16.h` in your program.

Half2 Math Functions
To use these functions, include the header file `cuda_fp16.h` in your program.

Structs

__half
__half data type

__half2
__half2 data type

__half2_raw
__half2_raw data type

__half_raw
__half_raw data type

Typedefs

__nv_half
This datatype is an `__nv_` prefixed alias.

__nv_half2
This datatype is an `__nv_` prefixed alias.

__nv_half2_raw
This datatype is an `__nv_` prefixed alias.

__nv_half_raw
This datatype is an `__nv_` prefixed alias.

half This datatype is meant to be the first-class or fundamental implementation of the half-precision numbers format.

half2

This datatype is meant to be the first-class or fundamental implementation of type for pairs of half-precision numbers.

nv_half

This datatype is an nv_ prefixed alias.

nv_half2

This datatype is an nv_ prefixed alias.

4.9. Typedefs

typedef *__half* **`__nv_half`**

This datatype is an __nv_ prefixed alias.

typedef *__half2* **`__nv_half2`**

This datatype is an __nv_ prefixed alias.

typedef *__half2_raw* **`__nv_half2_raw`**

This datatype is an __nv_ prefixed alias.

typedef *__half_raw* **`__nv_half_raw`**

This datatype is an __nv_ prefixed alias.

typedef *__half* **`half`**

This datatype is meant to be the first-class or fundamental implementation of the half-precision numbers format.

Should be implemented in the compiler in the future. Current implementation is a simple typedef to a respective user-level type with underscores.

typedef *__half2* **`half2`**

This datatype is meant to be the first-class or fundamental implementation of type for pairs of half-precision numbers.

Should be implemented in the compiler in the future. Current implementation is a simple typedef to a respective user-level type with underscores.

typedef *__half* **`nv_half`**

This datatype is an nv_ prefixed alias.

typedef *__half2* **`nv_half2`**

This datatype is an nv_ prefixed alias.

Chapter 5. Bfloat16 Precision Intrinsics

This section describes nv_bfloat16 precision intrinsic functions.

To use these functions, include the header file `cuda_bf16.h` in your program. All of the functions defined here are available in device code. Some of the functions are also available to host compilers, please refer to respective functions' documentation for details.

NOTE: Aggressive floating-point optimizations performed by host or device compilers may affect numeric behavior of the functions implemented in this header. Specific examples are:

- *hsin(__nv_bfloat16)*;
- *hcos(__nv_bfloat16)*;
- *h2sin(__nv_bfloat162)*;
- *h2cos(__nv_bfloat162)*;

The following macros are available to help users selectively enable/disable various definitions present in the header file:

- `CUDA_NO_BFLOAT16` - If defined, this macro will prevent the definition of additional type aliases in the global namespace, helping to avoid potential conflicts with symbols defined in the user program.
- `__CUDA_NO_BFLOAT16_CONVERSIONS__` - If defined, this macro will prevent the use of the C++ type conversions (converting constructors and conversion operators) that are common for built-in floating-point types, but may be undesirable for *`__nv_bfloat16`* which is essentially a user-defined type.
- `__CUDA_NO_BFLOAT16_OPERATORS__` and `__CUDA_NO_BFLOAT162_OPERATORS__` - If defined, these macros will prevent the inadvertent use of usual arithmetic and comparison operators. This enforces the storage-only type semantics and prevents C++ style computations on *`__nv_bfloat16`* and *`__nv_bfloat162`* types.

5.1. Bfloat16 Arithmetic Constants

To use these constants, include the header file `cuda_bf16.h` in your program.

Macros

CUDART_INF_BF16
Defines floating-point positive infinity value for the `nv_bfloat16` data type.

CUDART_MAX_NORMAL_BF16
Defines a maximum representable value for the `nv_bfloat16` data type.

CUDART_MIN_DENORM_BF16
Defines a minimum representable (denormalized) value for the `nv_bfloat16` data type.

CUDART_NAN_BF16
Defines canonical NaN value for the `nv_bfloat16` data type.

CUDART_NEG_ZERO_BF16
Defines a negative zero value for the `nv_bfloat16` data type.

CUDART_ONE_BF16
Defines a value of 1.0 for the `nv_bfloat16` data type.

CUDART_ZERO_BF16
Defines a positive zero value for the `nv_bfloat16` data type.

5.1.1. Macros

CUDART_INF_BF16 *__ushort_as_bfloat16*((unsigned short)0x7F80U)

Defines floating-point positive infinity value for the `nv_bfloat16` data type.

CUDART_MAX_NORMAL_BF16 *__ushort_as_bfloat16*((unsigned short)0x7F7FU)

Defines a maximum representable value for the `nv_bfloat16` data type.

CUDART_MIN_DENORM_BF16 *__ushort_as_bfloat16*((unsigned short)0x0001U)

Defines a minimum representable (denormalized) value for the `nv_bfloat16` data type.

CUDART_NAN_BF16 *__ushort_as_bfloat16*((unsigned short)0x7FFFU)

Defines canonical NaN value for the `nv_bfloat16` data type.

CUDART_NEG_ZERO_BF16 *__ushort_as_bfloat16*((unsigned short)0x8000U)

Defines a negative zero value for the `nv_bfloat16` data type.

CUDART_ONE_BF16 *__ushort_as_bfloat16*((unsigned short)0x3F80U)

Defines a value of 1.0 for the `nv_bfloat16` data type.

CUDART_ZERO_BF16 *__ushort_as_bfloat16*((unsigned short)0x0000U)

Defines a positive zero value for the `nv_bfloat16` data type.

5.2. Bfloat16 Arithmetic Functions

To use these functions, include the header file `cuda_bf16.h` in your program.

Functions

__host__ __device__ __nv_bfloat16 *__habs*(const __nv_bfloat16 a)
Calculates the absolute value of input `nv_bfloat16` number and returns the result.

__host__ __device__ __nv_bfloat16 *__hadd*(const __nv_bfloat16 a, const __nv_bfloat16 b)
Performs `nv_bfloat16` addition in round-to-nearest-even mode.

__host__ __device__ __nv_bfloat16 *__hadd_rn*(const __nv_bfloat16 a, const __nv_bfloat16 b)
Performs `nv_bfloat16` addition in round-to-nearest-even mode.

__host__ __device__ __nv_bfloat16 *__hadd_sat*(const __nv_bfloat16 a, const __nv_bfloat16 b)
Performs `nv_bfloat16` addition in round-to-nearest-even mode, with saturation to [0.0, 1.0].

__host__ __device__ __nv_bfloat16 *__hdiv*(const __nv_bfloat16 a, const __nv_bfloat16 b)
Performs `nv_bfloat16` division in round-to-nearest-even mode.

__device__ __nv_bfloat16 *__hfma*(const __nv_bfloat16 a, const __nv_bfloat16 b, const __nv_bfloat16 c)
Performs `nv_bfloat16` fused multiply-add in round-to-nearest-even mode.

__device__ __nv_bfloat16 *__hfma_relu*(const __nv_bfloat16 a, const __nv_bfloat16 b, const __nv_bfloat16 c)
Performs `nv_bfloat16` fused multiply-add in round-to-nearest-even mode with relu saturation.

__device__ __nv_bfloat16 *__hfma_sat*(const __nv_bfloat16 a, const __nv_bfloat16 b, const __nv_bfloat16 c)
Performs `nv_bfloat16` fused multiply-add in round-to-nearest-even mode, with saturation to [0.0, 1.0].

__host__ __device__ __nv_bfloat16 *__hmul*(const __nv_bfloat16 a, const __nv_bfloat16 b)
Performs `nv_bfloat16` multiplication in round-to-nearest-even mode.

__host__ __device__ __nv_bfloat16 *__hmul_rn*(const __nv_bfloat16 a, const __nv_bfloat16 b)
Performs `nv_bfloat16` multiplication in round-to-nearest-even mode.

__host__ __device__ __nv_bfloat16 *__hmul_sat*(const __nv_bfloat16 a, const __nv_bfloat16 b)
Performs `nv_bfloat16` multiplication in round-to-nearest-even mode, with saturation to [0.0, 1.0].

__host__ __device__ __nv_bfloat16 *__hneg*(const __nv_bfloat16 a)
Negates input `nv_bfloat16` number and returns the result.

__host__ __device__ __nv_bfloat16 *__hsub*(const __nv_bfloat16 a, const __nv_bfloat16 b)
Performs `nv_bfloat16` subtraction in round-to-nearest-even mode.

__host__ __device__ __nv_bfloat16 *__hsub_rn*(const __nv_bfloat16 a, const __nv_bfloat16 b)
Performs `nv_bfloat16` subtraction in round-to-nearest-even mode.

__host__ __device__ __nv_bfloat16 *__hsub_sat*(const __nv_bfloat16 a, const __nv_bfloat16 b)
Performs `nv_bfloat16` subtraction in round-to-nearest-even mode, with saturation to [0.0, 1.0].

__device__ __nv_bfloat16 *atomicAdd*(__nv_bfloat16 *const address, const __nv_bfloat16 val)
Adds `val` to the value stored at `address` in global or shared memory, and writes this value back to `address`.

__host__ __device__ __nv_bfloat16 *operator(const __nv_bfloat16 &lh, const __nv_bfloat16 &rh)**
Performs nv_bfloat16 multiplication operation.

__host__ __device__ __nv_bfloat16 & *operator*=*(__nv_bfloat16 &lh, const __nv_bfloat16 &rh)
Performs nv_bfloat16 compound assignment with multiplication operation.

__host__ __device__ __nv_bfloat16 *operator+*(const __nv_bfloat16 &h)
Implements nv_bfloat16 unary plus operator, returns input value.

__host__ __device__ __nv_bfloat16 *operator+*(const __nv_bfloat16 &lh, const __nv_bfloat16 &rh)
Performs nv_bfloat16 addition operation.

__host__ __device__ __nv_bfloat16 *operator++*(__nv_bfloat16 &h, const int ignored)
Performs nv_bfloat16 postfix increment operation.

__host__ __device__ __nv_bfloat16 & *operator++*(__nv_bfloat16 &h)
Performs nv_bfloat16 prefix increment operation.

__host__ __device__ __nv_bfloat16 & *operator+=*(__nv_bfloat16 &lh, const __nv_bfloat16 &rh)
Performs nv_bfloat16 compound assignment with addition operation.

__host__ __device__ __nv_bfloat16 *operator-*(const __nv_bfloat16 &h)
Implements nv_bfloat16 unary minus operator.

__host__ __device__ __nv_bfloat16 *operator-*(const __nv_bfloat16 &lh, const __nv_bfloat16 &rh)
Performs nv_bfloat16 subtraction operation.

__host__ __device__ __nv_bfloat16 & *operator--*(__nv_bfloat16 &h)
Performs nv_bfloat16 prefix decrement operation.

__host__ __device__ __nv_bfloat16 *operator--*(__nv_bfloat16 &h, const int ignored)
Performs nv_bfloat16 postfix decrement operation.

__host__ __device__ __nv_bfloat16 & *operator-=*(__nv_bfloat16 &lh, const __nv_bfloat16 &rh)
Performs nv_bfloat16 compound assignment with subtraction operation.

__host__ __device__ __nv_bfloat16 *operator/*(const __nv_bfloat16 &lh, const __nv_bfloat16 &rh)
Performs nv_bfloat16 division operation.

__host__ __device__ __nv_bfloat16 & *operator/=*(__nv_bfloat16 &lh, const __nv_bfloat16 &rh)
Performs nv_bfloat16 compound assignment with division operation.

5.2.1. Functions

__host__ __device__ *__nv_bfloat16* **__habs**(const *__nv_bfloat16* a)

Calculates the absolute value of input nv_bfloat16 number and returns the result.

Calculates the absolute value of input nv_bfloat16 number and returns the result.

Parameters

a – **[in]** - nv_bfloat16. Is only being read.

Returns

nv_bfloat16

- The absolute value of a.

__host__ __device__ *__nv_bfloat16* **__hadd**(const *__nv_bfloat16* a, const *__nv_bfloat16* b)

Performs nv_bfloat16 addition in round-to-nearest-even mode.

Performs nv_bfloat16 addition of inputs a and b, in round-to-nearest-even mode.

Parameters

- **a** – **[in]** - nv_bfloat16. Is only being read.
- **b** – **[in]** - nv_bfloat16. Is only being read.

Returns

nv_bfloat16

- The sum of a and b.

__host__ __device__ *__nv_bfloat16* **__hadd_rn**(const *__nv_bfloat16* a, const *__nv_bfloat16* b)

Performs `nv_bfloat16` addition in round-to-nearest-even mode.

Performs `nv_bfloat16` addition of inputs a and b, in round-to-nearest-even mode. Prevents floating-point contractions of mul+add into fma.

Parameters

- **a** – **[in]** - nv_bfloat16. Is only being read.
- **b** – **[in]** - nv_bfloat16. Is only being read.

Returns

nv_bfloat16

- The sum of a and b.

__host__ __device__ *__nv_bfloat16* **__hadd_sat**(const *__nv_bfloat16* a, const *__nv_bfloat16* b)

Performs `nv_bfloat16` addition in round-to-nearest-even mode, with saturation to [0.0, 1.0].

Performs `nv_bfloat16` add of inputs a and b, in round-to-nearest-even mode, and clamps the result to range [0.0, 1.0]. NaN results are flushed to +0.0.

Parameters

- **a** – **[in]** - nv_bfloat16. Is only being read.
- **b** – **[in]** - nv_bfloat16. Is only being read.

Returns

nv_bfloat16

- The sum of a and b, with respect to saturation.

__host__ __device__ *__nv_bfloat16* **__hdiv**(const *__nv_bfloat16* a, const *__nv_bfloat16* b)

Performs `nv_bfloat16` division in round-to-nearest-even mode.

Divides `nv_bfloat16` input a by input b in round-to-nearest-even mode.

Parameters

- **a** – **[in]** - nv_bfloat16. Is only being read.
- **b** – **[in]** - nv_bfloat16. Is only being read.

Returns

nv_bfloat16

- The result of dividing a by b.

__device__ *__nv_bfloat16* **__hfma**(const *__nv_bfloat16* a, const *__nv_bfloat16* b, const *__nv_bfloat16* c)

Performs `nv_bfloat16` fused multiply-add in round-to-nearest-even mode.

Performs `nv_bfloat16` multiply on inputs `a` and `b`, then performs a `nv_bfloat16` add of the result with `c`, rounding the result once in round-to-nearest-even mode.

Parameters

- **a** – **[in]** - nv_bfloat16. Is only being read.
- **b** – **[in]** - nv_bfloat16. Is only being read.
- **c** – **[in]** - nv_bfloat16. Is only being read.

Returns

nv_bfloat16

- The result of fused multiply-add operation on `a`, `b`, and `c`.

__device__ *__nv_bfloat16* **__hfma_relu**(const *__nv_bfloat16* a, const *__nv_bfloat16* b, const *__nv_bfloat16* c)

Performs `nv_bfloat16` fused multiply-add in round-to-nearest-even mode with relu saturation.

Performs `nv_bfloat16` multiply on inputs `a` and `b`, then performs a `nv_bfloat16` add of the result with `c`, rounding the result once in round-to-nearest-even mode. Then negative result is clamped to 0. NaN result is converted to canonical NaN.

Parameters

- **a** – **[in]** - nv_bfloat16. Is only being read.
- **b** – **[in]** - nv_bfloat16. Is only being read.
- **c** – **[in]** - nv_bfloat16. Is only being read.

Returns

nv_bfloat16

- The result of fused multiply-add operation on `a`, `b`, and `c` with relu saturation.

__device__ *__nv_bfloat16* **__hfma_sat**(const *__nv_bfloat16* a, const *__nv_bfloat16* b, const *__nv_bfloat16* c)

Performs `nv_bfloat16` fused multiply-add in round-to-nearest-even mode, with saturation to [0.0, 1.0].

Performs `nv_bfloat16` multiply on inputs `a` and `b`, then performs a `nv_bfloat16` add of the result with `c`, rounding the result once in round-to-nearest-even mode, and clamps the result to range [0.0, 1.0]. NaN results are flushed to +0.0.

Parameters

- **a** – **[in]** - nv_bfloat16. Is only being read.
- **b** – **[in]** - nv_bfloat16. Is only being read.
- **c** – **[in]** - nv_bfloat16. Is only being read.

Returns

nv_bfloat16

- The result of fused multiply-add operation on a, b, and c, with respect to saturation.

__host__ __device__ *__nv_bfloat16* **__hmul**(const *__nv_bfloat16* a, const *__nv_bfloat16* b)

Performs `nv_bfloat16` multiplication in round-to-nearest-even mode.

Performs `nv_bfloat16` multiplication of inputs a and b, in round-to-nearest-even mode.

Parameters

- **a** – **[in]** - nv_bfloat16. Is only being read.
- **b** – **[in]** - nv_bfloat16. Is only being read.

Returns

nv_bfloat16

- The result of multiplying a and b.

__host__ __device__ *__nv_bfloat16* **__hmul_rn**(const *__nv_bfloat16* a, const *__nv_bfloat16* b)

Performs `nv_bfloat16` multiplication in round-to-nearest-even mode.

Performs `nv_bfloat16` multiplication of inputs a and b, in round-to-nearest-even mode. Prevents floating-point contractions of mul+add or sub into fma.

Parameters

- **a** – **[in]** - nv_bfloat16. Is only being read.
- **b** – **[in]** - nv_bfloat16. Is only being read.

Returns

nv_bfloat16

- The result of multiplying a and b.

__host__ __device__ *__nv_bfloat16* **__hmul_sat**(const *__nv_bfloat16* a, const *__nv_bfloat16* b)

Performs `nv_bfloat16` multiplication in round-to-nearest-even mode, with saturation to [0.0, 1.0].

Performs `nv_bfloat16` multiplication of inputs a and b, in round-to-nearest-even mode, and clamps the result to range [0.0, 1.0]. NaN results are flushed to +0.0.

Parameters

- **a** – **[in]** - nv_bfloat16. Is only being read.
- **b** – **[in]** - nv_bfloat16. Is only being read.

Returns

nv_bfloat16

- The result of multiplying a and b, with respect to saturation.

__host__ __device__ *__nv_bfloat16* **__hneg**(const *__nv_bfloat16* a)

Negates input `nv_bfloat16` number and returns the result.

Negates input `nv_bfloat16` number and returns the result.

Parameters

a – **[in]** - nv_bfloat16. Is only being read.

Returns

nv_bfloat16

- minus a

__host__ __device__ *__nv_bfloat16* **__hsub**(const *__nv_bfloat16* a, const *__nv_bfloat16* b)

Performs `nv_bfloat16` subtraction in round-to-nearest-even mode.

Subtracts `nv_bfloat16` input b from input a in round-to-nearest-even mode.

Parameters

- **a** – **[in]** - nv_bfloat16. Is only being read.
- **b** – **[in]** - nv_bfloat16. Is only being read.

Returns

nv_bfloat16

- The result of subtracting b from a.

__host__ __device__ *__nv_bfloat16* **__hsub_rn**(const *__nv_bfloat16* a, const *__nv_bfloat16* b)

Performs `nv_bfloat16` subtraction in round-to-nearest-even mode.

Subtracts `nv_bfloat16` input b from input a in round-to-nearest-even mode. Prevents floating-point contractions of mul+sub into fma.

Parameters

- **a** – **[in]** - nv_bfloat16. Is only being read.
- **b** – **[in]** - nv_bfloat16. Is only being read.

Returns

nv_bfloat16

- The result of subtracting b from a.

__host__ __device__ *__nv_bfloat16* **__hsub_sat**(const *__nv_bfloat16* a, const *__nv_bfloat16* b)

Performs `nv_bfloat16` subtraction in round-to-nearest-even mode, with saturation to [0.0, 1.0].

Subtracts `nv_bfloat16` input b from input a in round-to-nearest-even mode, and clamps the result to range [0.0, 1.0]. NaN results are flushed to +0.0.

Parameters

- **a** – **[in]** - nv_bfloat16. Is only being read.
- **b** – **[in]** - nv_bfloat16. Is only being read.

Returns

nv_bfloat16

- The result of subtraction of b from a, with respect to saturation.

__device__ *__nv_bfloat16* **atomicAdd**(*__nv_bfloat16* *const address, const *__nv_bfloat16* val)

Adds `val` to the value stored at `address` in global or shared memory, and writes this value back to `address`.

This operation is performed in one atomic operation.

The location of `address` must be in global or shared memory. This operation has undefined behavior otherwise. This operation is natively supported by devices of compute capability 9.x and higher, older devices of compute capability 7.x and 8.x use emulation path.

Note: For more details about this function, see the Atomic Functions section in the CUDA C++ Programming Guide.

Parameters

- **address** – **[in]** - __nv_bfloat16*. An address in global or shared memory.
- **val** – **[in]** - *__nv_bfloat16*. The value to be added.

Returns

__nv_bfloat16

- The old value read from `address`.

__host__ __device__ *__nv_bfloat16* **operator***(const *__nv_bfloat16* &lh, const *__nv_bfloat16* &rh)

Performs `nv_bfloat16` multiplication operation.

See also *__hmul(__nv_bfloat16, __nv_bfloat16)*

__host__ __device__ *__nv_bfloat16* &**operator*=**(*__nv_bfloat16* &lh, const *__nv_bfloat16* &rh)

Performs `nv_bfloat16` compound assignment with multiplication operation.

__host__ __device__ *__nv_bfloat16* **operator+**(const *__nv_bfloat16* &h)

Implements `nv_bfloat16` unary plus operator, returns input value.

__host__ __device__ *__nv_bfloat16* **operator+**(const *__nv_bfloat16* &lh, const *__nv_bfloat16* &rh)

Performs `nv_bfloat16` addition operation.

See also *__hadd(__nv_bfloat16, __nv_bfloat16)*

__host__ __device__ *__nv_bfloat16* **operator++**(*__nv_bfloat16* &h, const int ignored)

Performs `nv_bfloat16` postfix increment operation.

__host__ __device__ *__nv_bfloat16* &**operator++**(*__nv_bfloat16* &h)

Performs `nv_bfloat16` prefix increment operation.

__host__ __device__ *__nv_bfloat16* &**operator+=**(*__nv_bfloat16* &lh, const *__nv_bfloat16* &rh)

Performs `nv_bfloat16` compound assignment with addition operation.

__host__ __device__ *__nv_bfloat16* **operator-**(const *__nv_bfloat16* &h)

Implements `nv_bfloat16` unary minus operator.

See also *__hneg(__nv_bfloat16)*

__host__ __device__ *__nv_bfloat16* **operator-**(const *__nv_bfloat16* &lh, const *__nv_bfloat16* &rh)

Performs `nv_bfloat16` subtraction operation.

See also *__hsub(__nv_bfloat16, __nv_bfloat16)*

__host__ __device__ *__nv_bfloat16* &**operator--**(*__nv_bfloat16* &h)

Performs `nv_bfloat16` prefix decrement operation.

__host__ __device__ *__nv_bfloat16* **operator--**(*__nv_bfloat16* &h, const int ignored)
Performs nv_bfloat16 postfix decrement operation.

__host__ __device__ *__nv_bfloat16* &**operator-=**(*__nv_bfloat16* &lh, const *__nv_bfloat16* &rh)
Performs nv_bfloat16 compound assignment with subtraction operation.

__host__ __device__ *__nv_bfloat16* **operator/**(const *__nv_bfloat16* &lh, const *__nv_bfloat16* &rh)
Performs nv_bfloat16 division operation.

See also *__hdiv(__nv_bfloat16, __nv_bfloat16)*

__host__ __device__ *__nv_bfloat16* &**operator/=**(*__nv_bfloat16* &lh, const *__nv_bfloat16* &rh)
Performs nv_bfloat16 compound assignment with division operation.

5.3. Bfloat16 Comparison Functions

To use these functions, include the header file cuda_bf16.h in your program.

Functions

__host__ __device__ bool *__heq*(const __nv_bfloat16 a, const __nv_bfloat16 b)
Performs nv_bfloat16 if-equal comparison.

__host__ __device__ bool *__hequ*(const __nv_bfloat16 a, const __nv_bfloat16 b)
Performs nv_bfloat16 unordered if-equal comparison.

__host__ __device__ bool *__hge*(const __nv_bfloat16 a, const __nv_bfloat16 b)
Performs nv_bfloat16 greater-equal comparison.

__host__ __device__ bool *__hgeu*(const __nv_bfloat16 a, const __nv_bfloat16 b)
Performs nv_bfloat16 unordered greater-equal comparison.

__host__ __device__ bool *__hgt*(const __nv_bfloat16 a, const __nv_bfloat16 b)
Performs nv_bfloat16 greater-than comparison.

__host__ __device__ bool *__hgtu*(const __nv_bfloat16 a, const __nv_bfloat16 b)
Performs nv_bfloat16 unordered greater-than comparison.

__host__ __device__ int *__hisinf*(const __nv_bfloat16 a)
Checks if the input nv_bfloat16 number is infinite.

__host__ __device__ bool *__hisnan*(const __nv_bfloat16 a)
Determine whether nv_bfloat16 argument is a NaN.

__host__ __device__ bool *__hle*(const __nv_bfloat16 a, const __nv_bfloat16 b)
Performs nv_bfloat16 less-equal comparison.

__host__ __device__ bool *__hleu*(const __nv_bfloat16 a, const __nv_bfloat16 b)
Performs nv_bfloat16 unordered less-equal comparison.

__host__ __device__ bool *__hlt*(const __nv_bfloat16 a, const __nv_bfloat16 b)
Performs nv_bfloat16 less-than comparison.

__host__ __device__ bool *__hltu*(const __nv_bfloat16 a, const __nv_bfloat16 b)
Performs nv_bfloat16 unordered less-than comparison.

__host__ __device__ __nv_bfloat16 *__hmax*(const __nv_bfloat16 a, const __nv_bfloat16 b)
Calculates nv_bfloat16 maximum of two input values.

__host__ __device__ __nv_bfloat16 *__hmax_nan*(const __nv_bfloat16 a, const __nv_bfloat16 b)
Calculates nv_bfloat16 maximum of two input values, NaNs pass through.

__host__ __device__ __nv_bfloat16 *__hmin*(const __nv_bfloat16 a, const __nv_bfloat16 b)
Calculates nv_bfloat16 minimum of two input values.

__host__ __device__ __nv_bfloat16 *__hmin_nan*(const __nv_bfloat16 a, const __nv_bfloat16 b)
Calculates nv_bfloat16 minimum of two input values, NaNs pass through.

__host__ __device__ bool *__hne*(const __nv_bfloat16 a, const __nv_bfloat16 b)
Performs nv_bfloat16 not-equal comparison.

__host__ __device__ bool *__hneu*(const __nv_bfloat16 a, const __nv_bfloat16 b)
Performs nv_bfloat16 unordered not-equal comparison.

__host__ __device__ bool *operator!=*(const __nv_bfloat16 &lh, const __nv_bfloat16 &rh)
Performs nv_bfloat16 unordered compare not-equal operation.

__host__ __device__ bool *operator<*(const __nv_bfloat16 &lh, const __nv_bfloat16 &rh)
Performs nv_bfloat16 ordered less-than compare operation.

__host__ __device__ bool *operator<=*(const __nv_bfloat16 &lh, const __nv_bfloat16 &rh)
Performs nv_bfloat16 ordered less-or-equal compare operation.

__host__ __device__ bool *operator==*(const __nv_bfloat16 &lh, const __nv_bfloat16 &rh)
Performs nv_bfloat16 ordered compare equal operation.

__host__ __device__ bool *operator>*(const __nv_bfloat16 &lh, const __nv_bfloat16 &rh)
Performs nv_bfloat16 ordered greater-than compare operation.

__host__ __device__ bool *operator>=*(const __nv_bfloat16 &lh, const __nv_bfloat16 &rh)
Performs nv_bfloat16 ordered greater-or-equal compare operation.

5.3.1. Functions

__host__ __device__ bool **__heq**(const *__nv_bfloat16* a, const *__nv_bfloat16* b)
Performs nv_bfloat16 if-equal comparison.

Performs nv_bfloat16 if-equal comparison of inputs a and b. NaN inputs generate false results.

Parameters

- **a** – **[in]** - nv_bfloat16. Is only being read.
- **b** – **[in]** - nv_bfloat16. Is only being read.

Returns
bool

- The boolean result of if-equal comparison of a and b.

__host__ __device__ bool **__hequ**(const *__nv_bfloat16* a, const *__nv_bfloat16* b)
Performs nv_bfloat16 unordered if-equal comparison.

Performs nv_bfloat16 if-equal comparison of inputs a and b. NaN inputs generate true results.

Parameters

- **a** – **[in]** - nv_bfloat16. Is only being read.
- **b** – **[in]** - nv_bfloat16. Is only being read.

Returns
bool

- The boolean result of unordered if-equal comparison of a and b.

__host__ __device__ bool **__hge**(const *__nv_bfloat16* a, const *__nv_bfloat16* b)

Performs `nv_bfloat16` greater-equal comparison.

Performs `nv_bfloat16` greater-equal comparison of inputs a and b. NaN inputs generate false results.

Parameters

- **a** – **[in]** - nv_bfloat16. Is only being read.
- **b** – **[in]** - nv_bfloat16. Is only being read.

Returns
bool

- The boolean result of greater-equal comparison of a and b.

__host__ __device__ bool **__hgeu**(const *__nv_bfloat16* a, const *__nv_bfloat16* b)

Performs `nv_bfloat16` unordered greater-equal comparison.

Performs `nv_bfloat16` greater-equal comparison of inputs a and b. NaN inputs generate true results.

Parameters

- **a** – **[in]** - nv_bfloat16. Is only being read.
- **b** – **[in]** - nv_bfloat16. Is only being read.

Returns
bool

- The boolean result of unordered greater-equal comparison of a and b.

__host__ __device__ bool **__hgt**(const *__nv_bfloat16* a, const *__nv_bfloat16* b)

Performs `nv_bfloat16` greater-than comparison.

Performs `nv_bfloat16` greater-than comparison of inputs a and b. NaN inputs generate false results.

Parameters

- **a** – **[in]** - nv_bfloat16. Is only being read.
- **b** – **[in]** - nv_bfloat16. Is only being read.

Returns
bool

- The boolean result of greater-than comparison of a and b.

__host__ __device__ bool **__hgtu**(const *__nv_bfloat16* a, const *__nv_bfloat16* b)

Performs `nv_bfloat16` unordered greater-than comparison.

Performs `nv_bfloat16` greater-than comparison of inputs a and b. NaN inputs generate true results.

Parameters

- **a** – **[in]** - nv_bfloat16. Is only being read.
- **b** – **[in]** - nv_bfloat16. Is only being read.

Returns

bool

- The boolean result of unordered greater-than comparison of a and b.

__host__ __device__ int **__hisinf**(const *__nv_bfloat16* a)

Checks if the input `nv_bfloat16` number is infinite.

Checks if the input `nv_bfloat16` number a is infinite.

Parameters

a – **[in]** - nv_bfloat16. Is only being read.

Returns

int

- -1 if a is equal to negative infinity,
- 1 if a is equal to positive infinity,
- 0 otherwise.

__host__ __device__ bool **__hisnan**(const *__nv_bfloat16* a)

Determine whether `nv_bfloat16` argument is a NaN.

Determine whether `nv_bfloat16` value a is a NaN.

Parameters

a – **[in]** - nv_bfloat16. Is only being read.

Returns

bool

- true if argument is NaN.

__host__ __device__ bool **__hle**(const *__nv_bfloat16* a, const *__nv_bfloat16* b)

Performs `nv_bfloat16` less-equal comparison.

Performs `nv_bfloat16` less-equal comparison of inputs a and b. NaN inputs generate false results.

Parameters

- **a** – **[in]** - nv_bfloat16. Is only being read.
- **b** – **[in]** - nv_bfloat16. Is only being read.

Returns

bool

- The boolean result of less-equal comparison of a and b.

__host__ __device__ bool **__hleu**(const *__nv_bfloat16* a, const *__nv_bfloat16* b)

Performs `nv_bfloat16` unordered less-equal comparison.

Performs `nv_bfloat16` less-equal comparison of inputs a and b. NaN inputs generate true results.

Parameters

- **a** – **[in]** - nv_bfloat16. Is only being read.
- **b** – **[in]** - nv_bfloat16. Is only being read.

Returns

bool

- The boolean result of unordered less-equal comparison of a and b.

__host__ __device__ bool **__hlt**(const *__nv_bfloat16* a, const *__nv_bfloat16* b)

Performs `nv_bfloat16` less-than comparison.

Performs `nv_bfloat16` less-than comparison of inputs a and b. NaN inputs generate false results.

Parameters

- **a** – **[in]** - nv_bfloat16. Is only being read.
- **b** – **[in]** - nv_bfloat16. Is only being read.

Returns

bool

- The boolean result of less-than comparison of a and b.

__host__ __device__ bool **__hltu**(const *__nv_bfloat16* a, const *__nv_bfloat16* b)

Performs `nv_bfloat16` unordered less-than comparison.

Performs `nv_bfloat16` less-than comparison of inputs a and b. NaN inputs generate true results.

Parameters

- **a** – **[in]** - nv_bfloat16. Is only being read.
- **b** – **[in]** - nv_bfloat16. Is only being read.

Returns

bool

- The boolean result of unordered less-than comparison of a and b.

__host__ __device__ *__nv_bfloat16* **__hmax**(const *__nv_bfloat16* a, const *__nv_bfloat16* b)

Calculates `nv_bfloat16` maximum of two input values.

Calculates `nv_bfloat16` max(a, b) defined as (a > b) ? a : b.

- If either of inputs is NaN, the other input is returned.
- If both inputs are NaNs, then canonical NaN is returned.
- If values of both inputs are 0.0, then +0.0 > -0.0

Parameters

- **a** – **[in]** - nv_bfloat16. Is only being read.
- **b** – **[in]** - nv_bfloat16. Is only being read.

Returns
nv_bfloat16

__host__ __device__ *__nv_bfloat16* **__hmax_nan**(const *__nv_bfloat16* a, const *__nv_bfloat16* b)

Calculates `nv_bfloat16` maximum of two input values, NaNs pass through.

Calculates `nv_bfloat16` max(a, b) defined as (a > b) ? a : b.

- If either of inputs is NaN, then canonical NaN is returned.
- If values of both inputs are 0.0, then +0.0 > -0.0

Parameters

- **a** – **[in]** - nv_bfloat16. Is only being read.
- **b** – **[in]** - nv_bfloat16. Is only being read.

Returns
nv_bfloat16

__host__ __device__ *__nv_bfloat16* **__hmin**(const *__nv_bfloat16* a, const *__nv_bfloat16* b)

Calculates `nv_bfloat16` minimum of two input values.

Calculates `nv_bfloat16` min(a, b) defined as (a < b) ? a : b.

- If either of inputs is NaN, the other input is returned.
- If both inputs are NaNs, then canonical NaN is returned.
- If values of both inputs are 0.0, then +0.0 > -0.0

Parameters

- **a** – **[in]** - nv_bfloat16. Is only being read.
- **b** – **[in]** - nv_bfloat16. Is only being read.

Returns
nv_bfloat16

__host__ __device__ *__nv_bfloat16* **__hmin_nan**(const *__nv_bfloat16* a, const *__nv_bfloat16* b)

Calculates `nv_bfloat16` minimum of two input values, NaNs pass through.

Calculates `nv_bfloat16` min(a, b) defined as (a < b) ? a : b.

- If either of inputs is NaN, then canonical NaN is returned.
- If values of both inputs are 0.0, then +0.0 > -0.0

Parameters

- **a** – **[in]** - nv_bfloat16. Is only being read.

- **b** – **[in]** - nv_bfloat16. Is only being read.

Returns
nv_bfloat16

__host__ __device__ bool **__hne**(const *__nv_bfloat16* a, const *__nv_bfloat16* b)

Performs `nv_bfloat16` not-equal comparison.

Performs `nv_bfloat16` not-equal comparison of inputs a and b. NaN inputs generate false results.

Parameters

- **a** – **[in]** - nv_bfloat16. Is only being read.
- **b** – **[in]** - nv_bfloat16. Is only being read.

Returns
bool

- The boolean result of not-equal comparison of a and b.

__host__ __device__ bool **__hneu**(const *__nv_bfloat16* a, const *__nv_bfloat16* b)

Performs `nv_bfloat16` unordered not-equal comparison.

Performs `nv_bfloat16` not-equal comparison of inputs a and b. NaN inputs generate true results.

Parameters

- **a** – **[in]** - nv_bfloat16. Is only being read.
- **b** – **[in]** - nv_bfloat16. Is only being read.

Returns
bool

- The boolean result of unordered not-equal comparison of a and b.

__host__ __device__ bool **operator!=**(const *__nv_bfloat16* &lh, const *__nv_bfloat16* &rh)

Performs `nv_bfloat16` unordered compare not-equal operation.

See also *__hneu(__nv_bfloat16, __nv_bfloat16)*

__host__ __device__ bool **operator<**(const *__nv_bfloat16* &lh, const *__nv_bfloat16* &rh)

Performs `nv_bfloat16` ordered less-than compare operation.

See also *__hlt(__nv_bfloat16, __nv_bfloat16)*

__host__ __device__ bool **operator<=**(const *__nv_bfloat16* &lh, const *__nv_bfloat16* &rh)

Performs `nv_bfloat16` ordered less-or-equal compare operation.

See also *__hle(__nv_bfloat16, __nv_bfloat16)*

__host__ __device__ bool **operator==**(const *__nv_bfloat16* &lh, const *__nv_bfloat16* &rh)

Performs `nv_bfloat16` ordered compare equal operation.

See also *__heq(__nv_bfloat16, __nv_bfloat16)* __host__ __device__ bool **operator>**(const *__nv_bfloat16* &lh, const *__nv_bfloat16* &rh)

Performs nv_bfloat16 ordered greater-than compare operation.

See also *__hgt(__nv_bfloat16, __nv_bfloat16)*

__host__ __device__ bool **operator>=**(const *__nv_bfloat16* &lh, const *__nv_bfloat16* &rh)

Performs nv_bfloat16 ordered greater-or-equal compare operation.

See also *__hge(__nv_bfloat16, __nv_bfloat16)*

5.4. Bfloat16 Math Functions

To use these functions, include the header file cuda_bf16.h in your program.

Functions

__device__ __nv_bfloat16 *hceil*(const __nv_bfloat16 h)

Calculate ceiling of the input argument.

__device__ __nv_bfloat16 *hcos*(const __nv_bfloat16 a)

Calculates nv_bfloat16 cosine in round-to-nearest-even mode.

__device__ __nv_bfloat16 *hexp*(const __nv_bfloat16 a)

Calculates nv_bfloat16 natural exponential function in round-to-nearest-even mode.

__device__ __nv_bfloat16 *hexp10*(const __nv_bfloat16 a)

Calculates nv_bfloat16 decimal exponential function in round-to-nearest-even mode.

__device__ __nv_bfloat16 *hexp2*(const __nv_bfloat16 a)

Calculates nv_bfloat16 binary exponential function in round-to-nearest-even mode.

__device__ __nv_bfloat16 *hfloor*(const __nv_bfloat16 h)

Calculate the largest integer less than or equal to h .

__device__ __nv_bfloat16 *hlog*(const __nv_bfloat16 a)

Calculates nv_bfloat16 natural logarithm in round-to-nearest-even mode.

__device__ __nv_bfloat16 *hlog10*(const __nv_bfloat16 a)

Calculates nv_bfloat16 decimal logarithm in round-to-nearest-even mode.

__device__ __nv_bfloat16 *hlog2*(const __nv_bfloat16 a)

Calculates nv_bfloat16 binary logarithm in round-to-nearest-even mode.

__device__ __nv_bfloat16 *hrcp*(const __nv_bfloat16 a)

Calculates nv_bfloat16 reciprocal in round-to-nearest-even mode.

__device__ __nv_bfloat16 *hrint*(const __nv_bfloat16 h)

Round input to nearest integer value in nv_bfloat16 floating-point number.

__device__ __nv_bfloat16 *hrsqrt*(const __nv_bfloat16 a)

Calculates nv_bfloat16 reciprocal square root in round-to-nearest-even mode.

__device__ __nv_bfloat16 *hsin*(const __nv_bfloat16 a)

Calculates nv_bfloat16 sine in round-to-nearest-even mode.

__device__ __nv_bfloat16 *hsqrt*(const __nv_bfloat16 a)

Calculates nv_bfloat16 square root in round-to-nearest-even mode.

__device__ __nv_bfloat16 *htanh*(const __nv_bfloat16 a)
Calculates `nv_bfloat16` hyperbolic tangent function in round-to-nearest-even mode.

__device__ __nv_bfloat16 *htanh_approx*(const __nv_bfloat16 a)
Calculates approximate `nv_bfloat16` hyperbolic tangent function.

__device__ __nv_bfloat16 *htrunc*(const __nv_bfloat16 h)
Truncate input argument to the integral part.

5.4.1. Functions

__device__ *__nv_bfloat16* **hceil**(const *__nv_bfloat16* h)
Calculate ceiling of the input argument.

Compute the smallest integer value not less than h.

Parameters
h – **[in]** - nv_bfloat16. Is only being read.

Returns
nv_bfloat16

- The smallest integer value not less than h.

__device__ *__nv_bfloat16* **hcos**(const *__nv_bfloat16* a)
Calculates `nv_bfloat16` cosine in round-to-nearest-even mode.

Calculates `nv_bfloat16` cosine of input a in round-to-nearest-even mode.

NOTE: this function's implementation calls *cosf(float)* function and is exposed to compiler optimizations. Specifically, `--use_fast_math` flag changes *cosf(float)* into an intrinsic *__cosf(float)*, which has less accurate numeric behavior.

Parameters
a – **[in]** - nv_bfloat16. Is only being read.

Returns
nv_bfloat16

- The cosine of a.

__device__ *__nv_bfloat16* **hexp**(const *__nv_bfloat16* a)
Calculates `nv_bfloat16` natural exponential function in round-to-nearest-even mode.

Calculates `nv_bfloat16` natural exponential function of input a in round-to-nearest-even mode.

Parameters
a – **[in]** - nv_bfloat16. Is only being read.

Returns
nv_bfloat16

- The natural exponential function on a.

__device__ *__nv_bfloat16* **hexp10**(const *__nv_bfloat16* a)
Calculates `nv_bfloat16` decimal exponential function in round-to-nearest-even mode.

Calculates `nv_bfloat16` decimal exponential function of input a in round-to-nearest-even mode.

Parameters
a – **[in]** - nv_bfloat16. Is only being read.

Returns
nv_bfloat16

- The decimal exponential function on a.

__device__ *__nv_bfloat16* **hexp2**(const *__nv_bfloat16* a)

Calculates nv_bfloat16 binary exponential function in round-to-nearest-even mode.

Calculates nv_bfloat16 binary exponential function of input a in round-to-nearest-even mode.

Parameters
a – **[in]** - nv_bfloat16. Is only being read.

Returns
nv_bfloat16

- The binary exponential function on a.

__device__ *__nv_bfloat16* **hfloor**(const *__nv_bfloat16* h)

Calculate the largest integer less than or equal to h.

Calculate the largest integer value which is less than or equal to h.

Parameters
h – **[in]** - nv_bfloat16. Is only being read.

Returns
nv_bfloat16

- The largest integer value which is less than or equal to h.

__device__ *__nv_bfloat16* **hlog**(const *__nv_bfloat16* a)

Calculates nv_bfloat16 natural logarithm in round-to-nearest-even mode.

Calculates nv_bfloat16 natural logarithm of input a in round-to-nearest-even mode.

Parameters
a – **[in]** - nv_bfloat16. Is only being read.

Returns
nv_bfloat16

- The natural logarithm of a.

__device__ *__nv_bfloat16* **hlog10**(const *__nv_bfloat16* a)

Calculates nv_bfloat16 decimal logarithm in round-to-nearest-even mode.

Calculates nv_bfloat16 decimal logarithm of input a in round-to-nearest-even mode.

Parameters
a – **[in]** - nv_bfloat16. Is only being read.

Returns
nv_bfloat16

- The decimal logarithm of a.

__device__ *__nv_bfloat16* **hlog2**(const *__nv_bfloat16* a)

Calculates nv_bfloat16 binary logarithm in round-to-nearest-even mode.

Calculates nv_bfloat16 binary logarithm of input a in round-to-nearest-even mode.

Parameters
a – **[in]** - nv_bfloat16. Is only being read.

Returns
nv_bfloat16

- The binary logarithm of a.

__device__ *__nv_bfloat16* **hrcp**(const *__nv_bfloat16* a)

Calculates `nv_bfloat16` reciprocal in round-to-nearest-even mode.

Calculates `nv_bfloat16` reciprocal of input a in round-to-nearest-even mode.

Parameters
a – **[in]** - nv_bfloat16. Is only being read.

Returns
nv_bfloat16

- The reciprocal of a.

__device__ *__nv_bfloat16* **hrint**(const *__nv_bfloat16* h)

Round input to nearest integer value in nv_bfloat16 floating-point number.

Round h to the nearest integer value in nv_bfloat16 floating-point format, with bfloat16way cases rounded to the nearest even integer value.

Parameters
h – **[in]** - nv_bfloat16. Is only being read.

Returns
nv_bfloat16

- The nearest integer to h.

__device__ *__nv_bfloat16* **hrsqrt**(const *__nv_bfloat16* a)

Calculates `nv_bfloat16` reciprocal square root in round-to-nearest-even mode.

Calculates `nv_bfloat16` reciprocal square root of input a in round-to-nearest-even mode.

Parameters
a – **[in]** - nv_bfloat16. Is only being read.

Returns
nv_bfloat16

- The reciprocal square root of a.

__device__ *__nv_bfloat16* **hsin**(const *__nv_bfloat16* a)

Calculates `nv_bfloat16` sine in round-to-nearest-even mode.

Calculates `nv_bfloat16` sine of input a in round-to-nearest-even mode.

NOTE: this function's implementation calls *sinf(float)* function and is exposed to compiler optimizations. Specifically, `--use_fast_math` flag changes *sinf(float)* into an intrinsic *__sinf(float)*, which has less accurate numeric behavior.

Parameters
a – **[in]** - nv_bfloat16. Is only being read.

Returns
nv_bfloat16

- The sine of a.

__device__ *__nv_bfloat16* **hsqrt**(const *__nv_bfloat16* a)

Calculates nv_bfloat16 square root in round-to-nearest-even mode.

Calculates nv_bfloat16 square root of input a in round-to-nearest-even mode.

Parameters
a – **[in]** - nv_bfloat16. Is only being read.

Returns
nv_bfloat16

- The square root of a.

__device__ *__nv_bfloat16* **htanh**(const *__nv_bfloat16* a)

Calculates nv_bfloat16 hyperbolic tangent function in round-to-nearest-even mode.

Calculates nv_bfloat16 hyperbolic tangent function: tanh(a) in round-to-nearest-even mode.

Parameters
a – **[in]** - nv_bfloat16. Is only being read.

Returns
nv_bfloat16

- The hyperbolic tangent function of a.
- htanh (± 0) returns (± 0).
- htanh $(\pm\infty)$ returns (± 1).
- htanh(NaN) returns NaN.

__device__ *__nv_bfloat16* **htanh_approx**(const *__nv_bfloat16* a)

Calculates approximate nv_bfloat16 hyperbolic tangent function.

Calculates approximate nv_bfloat16 hyperbolic tangent function: tanh(a). This operation uses HW acceleration on devices of compute capability 9.x and higher.

Parameters
a – **[in]** - nv_bfloat16. Is only being read.

Returns
nv_bfloat16

- The approximate hyperbolic tangent function of a.
- htanh_approx (± 0) returns (± 0).
- htanh_approx $(\pm\infty)$ returns (± 1).
- htanh_approx(NaN) returns NaN.

__device__ *__nv_bfloat16* **htrunc**(const *__nv_bfloat16* h)

Truncate input argument to the integral part.

Round h to the nearest integer value that does not exceed h in magnitude.

Parameters
h – **[in]** - nv_bfloat16. Is only being read.

Returns
nv_bfloat16

- The truncated integer value.

5.5. Bfloat16 Precision Conversion and Data Movement

To use these functions, include the header file `cuda_bf16.h` in your program.

Functions

__host__ __device__ float2 *__bfloat1622float2*(const __nv_bfloat162 a)
Converts both halves of `nv_bfloat162` to float2 and returns the result.

__host__ __device__ __nv_bfloat162 *__bfloat162bfloat162*(const __nv_bfloat16 a)
Returns `nv_bfloat162` with both halves equal to the input value.

__host__ __device__ signed char *__bfloat162char_rz*(const __nv_bfloat16 h)
Convert a nv_bfloat16 to a signed char in round-towards-zero mode.

__host__ __device__ float *__bfloat162float*(const __nv_bfloat16 a)
Converts `nv_bfloat16` number to float.

__device__ int *__bfloat162int_rd*(const __nv_bfloat16 h)
Convert a nv_bfloat16 to a signed integer in round-down mode.

__device__ int *__bfloat162int_rn*(const __nv_bfloat16 h)
Convert a nv_bfloat16 to a signed integer in round-to-nearest-even mode.

__device__ int *__bfloat162int_ru*(const __nv_bfloat16 h)
Convert a nv_bfloat16 to a signed integer in round-up mode.

__host__ __device__ int *__bfloat162int_rz*(const __nv_bfloat16 h)
Convert a nv_bfloat16 to a signed integer in round-towards-zero mode.

__device__ long long int *__bfloat162ll_rd*(const __nv_bfloat16 h)
Convert a nv_bfloat16 to a signed 64-bit integer in round-down mode.

__device__ long long int *__bfloat162ll_rn*(const __nv_bfloat16 h)
Convert a nv_bfloat16 to a signed 64-bit integer in round-to-nearest-even mode.

__device__ long long int *__bfloat162ll_ru*(const __nv_bfloat16 h)
Convert a nv_bfloat16 to a signed 64-bit integer in round-up mode.

__host__ __device__ long long int *__bfloat162ll_rz*(const __nv_bfloat16 h)
Convert a nv_bfloat16 to a signed 64-bit integer in round-towards-zero mode.

__device__ short int *__bfloat162short_rd*(const __nv_bfloat16 h)
Convert a nv_bfloat16 to a signed short integer in round-down mode.

__device__ short int *__bfloat162short_rn*(const __nv_bfloat16 h)
Convert a nv_bfloat16 to a signed short integer in round-to-nearest-even mode.

__device__ short int *__bfloat162short_ru*(const __nv_bfloat16 h)
Convert a nv_bfloat16 to a signed short integer in round-up mode.

__host__ __device__ short int *__bfloat162short_rz*(const __nv_bfloat16 h)
Convert a nv_bfloat16 to a signed short integer in round-towards-zero mode.

__host__ __device__ unsigned char *__bfloat162uchar_rz*(const __nv_bfloat16 h)
Convert a nv_bfloat16 to an unsigned char in round-towards-zero mode.

__device__ unsigned int *__bfloat162uint_rd*(const __nv_bfloat16 h)
Convert a nv_bfloat16 to an unsigned integer in round-down mode.

__device__ unsigned int *__bfloat162uint_rn*(const __nv_bfloat16 h)
Convert a nv_bfloat16 to an unsigned integer in round-to-nearest-even mode.

__device__ unsigned int *__bfloat162uint_ru*(const __nv_bfloat16 h)
Convert a nv_bfloat16 to an unsigned integer in round-up mode.

__host__ __device__ unsigned int *__bfloat162uint_rz*(const __nv_bfloat16 h)
Convert a nv_bfloat16 to an unsigned integer in round-towards-zero mode.

__device__ unsigned long long int *__bfloat162ull_rd*(const __nv_bfloat16 h)
Convert a nv_bfloat16 to an unsigned 64-bit integer in round-down mode.

__device__ unsigned long long int *__bfloat162ull_rn*(const __nv_bfloat16 h)
Convert a nv_bfloat16 to an unsigned 64-bit integer in round-to-nearest-even mode.

__device__ unsigned long long int *__bfloat162ull_ru*(const __nv_bfloat16 h)
Convert a nv_bfloat16 to an unsigned 64-bit integer in round-up mode.

__host__ __device__ unsigned long long int *__bfloat162ull_rz*(const __nv_bfloat16 h)
Convert a nv_bfloat16 to an unsigned 64-bit integer in round-towards-zero mode.

__device__ unsigned short int *__bfloat162ushort_rd*(const __nv_bfloat16 h)
Convert a nv_bfloat16 to an unsigned short integer in round-down mode.

__device__ unsigned short int *__bfloat162ushort_rn*(const __nv_bfloat16 h)
Convert a nv_bfloat16 to an unsigned short integer in round-to-nearest-even mode.

__device__ unsigned short int *__bfloat162ushort_ru*(const __nv_bfloat16 h)
Convert a nv_bfloat16 to an unsigned short integer in round-up mode.

__host__ __device__ unsigned short int *__bfloat162ushort_rz*(const __nv_bfloat16 h)
Convert a nv_bfloat16 to an unsigned short integer in round-towards-zero mode.

__host__ __device__ short int *__bfloat16_as_short*(const __nv_bfloat16 h)
Reinterprets bits in a `nv_bfloat16` as a signed short integer.

__host__ __device__ unsigned short int *__bfloat16_as_ushort*(const __nv_bfloat16 h)
Reinterprets bits in a `nv_bfloat16` as an unsigned short integer.

__host__ __device__ __nv_bfloat16 *__double2bfloat16*(const double a)
Converts double number to nv_bfloat16 precision in round-to-nearest-even mode and returns `nv_bfloat16` with converted value.

__host__ __device__ __nv_bfloat162 *__float22bfloat162_rn*(const float2 a)
Converts both components of float2 number to nv_bfloat16 precision in round-to-nearest-even mode and returns `nv_bfloat162` with converted values.

__host__ __device__ __nv_bfloat16 *__float2bfloat16*(const float a)
Converts float number to nv_bfloat16 precision in round-to-nearest-even mode and returns `nv_bfloat16` with converted value.

__host__ __device__ __nv_bfloat162 *__float2bfloat162_rn*(const float a)
Converts input to nv_bfloat16 precision in round-to-nearest-even mode and populates both halves of `nv_bfloat162` with converted value.

__host__ __device__ __nv_bfloat16 *__float2bfloat16_rd*(const float a)
Converts float number to nv_bfloat16 precision in round-down mode and returns `nv_bfloat16` with converted value.

__host__ __device__ __nv_bfloat16 *__float2bfloat16_rn*(const float a)
Converts float number to nv_bfloat16 precision in round-to-nearest-even mode and returns `nv_bfloat16` with converted value.

__host__ __device__ __nv_bfloat16 *__float2bfloat16_ru*(const float a)
Converts float number to nv_bfloat16 precision in round-up mode and returns `nv_bfloat16` with converted value.

__host__ __device__ __nv_bfloat16 *__float2bfloat16_rz*(const float a)
Converts float number to nv_bfloat16 precision in round-towards-zero mode and returns `nv_bfloat16` with converted value.

__host__ __device__ __nv_bfloat162 *__floats2bfloat162_rn*(const float a, const float b)
Converts both input floats to nv_bfloat16 precision in round-to-nearest-even mode and returns `nv_bfloat162` with converted values.

__host__ __device__ __nv_bfloat162 *__halves2bfloat162*(const __nv_bfloat16 a, const __nv_bfloat16 b)
Combines two `nv_bfloat16` numbers into one `nv_bfloat162` number.

__host__ __device__ __nv_bfloat16 *__high2bfloat16*(const __nv_bfloat162 a)
Returns high 16 bits of `nv_bfloat162` input.

__host__ __device__ __nv_bfloat162 *__high2bfloat162*(const __nv_bfloat162 a)
Extracts high 16 bits from `nv_bfloat162` input.

__host__ __device__ float *__high2float*(const __nv_bfloat162 a)
Converts high 16 bits of `nv_bfloat162` to float and returns the result.

__host__ __device__ __nv_bfloat162 *__highs2bfloat162*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Extracts high 16 bits from each of the two `nv_bfloat162` inputs and combines into one `nv_bfloat162` number.

__device__ __nv_bfloat16 *__int2bfloat16_rd*(const int i)
Convert a signed integer to a nv_bfloat16 in round-down mode.

__host__ __device__ __nv_bfloat16 *__int2bfloat16_rn*(const int i)
Convert a signed integer to a nv_bfloat16 in round-to-nearest-even mode.

__device__ __nv_bfloat16 *__int2bfloat16_ru*(const int i)
Convert a signed integer to a nv_bfloat16 in round-up mode.

__device__ __nv_bfloat16 *__int2bfloat16_rz*(const int i)
Convert a signed integer to a nv_bfloat16 in round-towards-zero mode.

__device__ __nv_bfloat162 *__ldca*(const __nv_bfloat162 *const ptr)
Generates a `ld.global.ca` load instruction.

__device__ __nv_bfloat16 *__ldca*(const __nv_bfloat16 *const ptr)
Generates a `ld.global.ca` load instruction.

__device__ __nv_bfloat16 *__ldcg*(const __nv_bfloat16 *const ptr)
Generates a `ld.global.cg` load instruction.

__device__ __nv_bfloat162 *__ldcg*(const __nv_bfloat162 *const ptr)
Generates a `ld.global.cg` load instruction.

__device__ __nv_bfloat162 *__ldcs*(const __nv_bfloat162 *const ptr)
Generates a `ld.global.cs` load instruction.

__device__ __nv_bfloat16 *__ldcs*(const __nv_bfloat16 *const ptr)
Generates a `ld.global.cs` load instruction.

__device__ __nv_bfloat16 *__ldcv*(const __nv_bfloat16 *const ptr)
Generates a `ld.global.cv` load instruction.

__device__ __nv_bfloat162 *__ldcv*(const __nv_bfloat162 *const ptr)
Generates a `ld.global.cv` load instruction.

__device__ __nv_bfloat162 *__ldg*(const __nv_bfloat162 *const ptr)
Generates a `ld.global.nc` load instruction.

__device__ __nv_bfloat16 *__ldg*(const __nv_bfloat16 *const ptr)
Generates a `ld.global.nc` load instruction.

__device__ __nv_bfloat162 *__ldlu*(const __nv_bfloat162 *const ptr)
Generates a `ld.global.lu` load instruction.

__device__ __nv_bfloat16 *__ldlu*(const __nv_bfloat16 *const ptr)
Generates a `ld.global.lu` load instruction.

__device__ __nv_bfloat16 *__ll2bfloat16_rd*(const long long int i)
Convert a signed 64-bit integer to a nv_bfloat16 in round-down mode.

__host__ __device__ __nv_bfloat16 *__ll2bfloat16_rn*(const long long int i)
Convert a signed 64-bit integer to a nv_bfloat16 in round-to-nearest-even mode.

__device__ __nv_bfloat16 *__ll2bfloat16_ru*(const long long int i)
Convert a signed 64-bit integer to a nv_bfloat16 in round-up mode.

__device__ __nv_bfloat16 *__ll2bfloat16_rz*(const long long int i)
Convert a signed 64-bit integer to a nv_bfloat16 in round-towards-zero mode.

__host__ __device__ __nv_bfloat16 *__low2bfloat16*(const __nv_bfloat162 a)
Returns low 16 bits of `nv_bfloat162` input.

__host__ __device__ __nv_bfloat162 *__low2bfloat162*(const __nv_bfloat162 a)
Extracts low 16 bits from `nv_bfloat162` input.

__host__ __device__ float *__low2float*(const __nv_bfloat162 a)
Converts low 16 bits of `nv_bfloat162` to float and returns the result.

__host__ __device__ __nv_bfloat162 *__lowhigh2highlow*(const __nv_bfloat162 a)
Swaps both halves of the `nv_bfloat162` input.

__host__ __device__ __nv_bfloat162 *__lows2bfloat162*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Extracts low 16 bits from each of the two `nv_bfloat162` inputs and combines into one `nv_bfloat162` number.

***__nv_bfloat162::__nv_bfloat162*()=default**
Constructor by default.

__host__ __device__ constexpr *__nv_bfloat162::__nv_bfloat162*(const __nv_bfloat16 &a, const __nv_bfloat16 &b)
Constructor from two `__nv_bfloat16` variables.

__host__ __device__ *__nv_bfloat162::__nv_bfloat162*(__nv_bfloat162 &&src)
Move constructor, available for C++11 and later dialects.

__host__ __device__ *__nv_bfloat162::__nv_bfloat162*(const __nv_bfloat162_raw &h2r)
Constructor from `__nv_bfloat162_raw`.

__host__ __device__ *__nv_bfloat162::__nv_bfloat162*(const __nv_bfloat162 &src)
Copy constructor.

__host__ __device__ *__nv_bfloat162::operator __nv_bfloat162_raw*() const
Conversion operator to `__nv_bfloat162_raw`.

__host__ __device__ __nv_bfloat162 & *__nv_bfloat162::operator=*(const __nv_bfloat162 &src)
Copy assignment operator.

__host__ __device__ __nv_bfloat162 & *__nv_bfloat162::operator=*(const __nv_bfloat162_raw &h2r)
Assignment operator from `__nv_bfloat162_raw`.

__host__ __device__ __nv_bfloat162 & *__nv_bfloat162::operator=*(__nv_bfloat162 &&src)
Move assignment operator, available for C++11 and later dialects.

__host__ __device__ *__nv_bfloat16::__nv_bfloat16*(const double f)
Construct `__nv_bfloat16` from `double` input using default round-to-nearest-even rounding mode.

__host__ __device__ *__nv_bfloat16::__nv_bfloat16*(const float f)
Construct `__nv_bfloat16` from `float` input using default round-to-nearest-even rounding mode.

__host__ __device__ *__nv_bfloat16::__nv_bfloat16*(long long val)
Construct `__nv_bfloat16` from `long long` input using default round-to-nearest-even rounding mode.

__host__ __device__ *__nv_bfloat16::__nv_bfloat16*(unsigned short val)
Construct `__nv_bfloat16` from `unsigned short` integer input using default round-to-nearest-even rounding mode.

__host__ __device__ *__nv_bfloat16::__nv_bfloat16*(unsigned int val)
Construct `__nv_bfloat16` from `unsigned int` input using default round-to-nearest-even rounding mode.

***__nv_bfloat16::__nv_bfloat16*()=default**
Constructor by default.

__host__ __device__ *__nv_bfloat16::__nv_bfloat16*(const __half f)
Construct `__nv_bfloat16` from `__half` input using default round-to-nearest-even rounding mode.

__host__ __device__ *__nv_bfloat16::__nv_bfloat16*(short val)
Construct `__nv_bfloat16` from `short` integer input using default round-to-nearest-even rounding mode.

__host__ __device__ constexpr *__nv_bfloat16::__nv_bfloat16*(const __nv_bfloat16_raw &hr)
Constructor from `__nv_bfloat16_raw`.

__host__ __device__ *__nv_bfloat16::__nv_bfloat16*(unsigned long long val)
Construct `__nv_bfloat16` from `unsigned long long` input using default round-to-nearest-even rounding mode.

__host__ __device__ *__nv_bfloat16::__nv_bfloat16*(int val)
Construct `__nv_bfloat16` from `int` input using default round-to-nearest-even rounding mode.

__host__ __device__ *__nv_bfloat16::__nv_bfloat16*(const unsigned long val)
Construct `__nv_bfloat16` from `unsigned long` input using default round-to-nearest-even rounding mode.

__host__ __device__ *__nv_bfloat16::__nv_bfloat16*(const long val)
Construct `__nv_bfloat16` from `long` input using default round-to-nearest-even rounding mode.

__host__ __device__ ***__nv_bfloat16::operator __nv_bfloat16_raw*****() const**
Type cast to `__nv_bfloat16_raw` operator.

__host__ __device__ ***__nv_bfloat16::operator __nv_bfloat16_raw*****() const volatile**
Type cast to `__nv_bfloat16_raw` operator with `volatile` input.

__host__ __device__ constexpr ***__nv_bfloat16::operator bool*****() const**
Conversion operator to `bool` data type.

__host__ __device__ ***__nv_bfloat16::operator char*****() const**
Conversion operator to an implementation defined `char` data type.

__host__ __device__ ***__nv_bfloat16::operator float*****() const**
Type cast to `float` operator.

__host__ __device__ ***__nv_bfloat16::operator int*****() const**
Conversion operator to `int` data type.

__host__ __device__ ***__nv_bfloat16::operator long*****() const**
Conversion operator to `long` data type.

__host__ __device__ ***__nv_bfloat16::operator long long*****() const**
Conversion operator to `long long` data type.

__host__ __device__ ***__nv_bfloat16::operator short*****() const**
Conversion operator to `short` data type.

__host__ __device__ ***__nv_bfloat16::operator signed char*****() const**
Conversion operator to `signed char` data type.

__host__ __device__ ***__nv_bfloat16::operator unsigned char*****() const**
Conversion operator to `unsigned char` data type.

__host__ __device__ ***__nv_bfloat16::operator unsigned int*****() const**
Conversion operator to `unsigned int` data type.

__host__ __device__ ***__nv_bfloat16::operator unsigned long*****() const**
Conversion operator to `unsigned long` data type.

__host__ __device__ ***__nv_bfloat16::operator unsigned long long*****() const**
Conversion operator to `unsigned long long` data type.

__host__ __device__ ***__nv_bfloat16::operator unsigned short*****() const**
Conversion operator to `unsigned short` data type.

__host__ __device__ __nv_bfloat16 & ***__nv_bfloat16::operator=*****(const __nv_bfloat16_raw &hr)**
Assignment operator from `__nv_bfloat16_raw` .

__host__ __device__ __nv_bfloat16 & ***__nv_bfloat16::operator=*****(unsigned int val)**
Type cast from `unsigned int` assignment operator, using default round-to-nearest-even rounding mode.

__host__ __device__ __nv_bfloat16 & ***__nv_bfloat16::operator=*****(int val)**
Type cast from `int` assignment operator, using default round-to-nearest-even rounding mode.

__host__ __device__ __nv_bfloat16 & ***__nv_bfloat16::operator=*****(long long val)**
Type cast from `long long` assignment operator, using default round-to-nearest-even rounding mode.

__host__ __device__ __nv_bfloat16 & ***__nv_bfloat16::operator=*****(unsigned long long val)**
Type cast from `unsigned long long` assignment operator, using default round-to-nearest-even rounding mode.

__host__ __device__ __nv_bfloat16 & *__nv_bfloat16::operator=*(unsigned short val)
Type cast from `unsigned short` assignment operator, using default round-to-nearest-even rounding mode.

__host__ __device__ volatile __nv_bfloat16 & *__nv_bfloat16::operator=*(const __nv_bfloat16_raw &hr) volatile
Assignment operator from `__nv_bfloat16_raw` to `volatile __nv_bfloat16` .

__host__ __device__ __nv_bfloat16 & *__nv_bfloat16::operator=*(const double f)
Type cast to `__nv_bfloat16` assignment operator from `double` input using default round-to-nearest-even rounding mode.

__host__ __device__ __nv_bfloat16 & *__nv_bfloat16::operator=*(const float f)
Type cast to `__nv_bfloat16` assignment operator from `float` input using default round-to-nearest-even rounding mode.

__host__ __device__ volatile __nv_bfloat16 & *__nv_bfloat16::operator=*(const volatile __nv_bfloat16_raw &hr) volatile
Assignment operator from `volatile __nv_bfloat16_raw` to `volatile __nv_bfloat16` .

__host__ __device__ __nv_bfloat16 & *__nv_bfloat16::operator=*(short val)
Type cast from `short` assignment operator, using default round-to-nearest-even rounding mode.

__device__ __nv_bfloat162 *__shfl_down_sync*(const unsigned int mask, const __nv_bfloat162 var, const unsigned int delta, const int width=warpSize)
Exchange a variable between threads within a warp.

__device__ __nv_bfloat16 *__shfl_down_sync*(const unsigned int mask, const __nv_bfloat16 var, const unsigned int delta, const int width=warpSize)
Exchange a variable between threads within a warp.

__device__ __nv_bfloat162 *__shfl_sync*(const unsigned int mask, const __nv_bfloat162 var, const int srcLane, const int width=warpSize)
Exchange a variable between threads within a warp.

__device__ __nv_bfloat16 *__shfl_sync*(const unsigned int mask, const __nv_bfloat16 var, const int srcLane, const int width=warpSize)
Exchange a variable between threads within a warp.

__device__ __nv_bfloat16 *__shfl_up_sync*(const unsigned int mask, const __nv_bfloat16 var, const unsigned int delta, const int width=warpSize)
Exchange a variable between threads within a warp.

__device__ __nv_bfloat162 *__shfl_up_sync*(const unsigned int mask, const __nv_bfloat162 var, const unsigned int delta, const int width=warpSize)
Exchange a variable between threads within a warp.

__device__ __nv_bfloat16 *__shfl_xor_sync*(const unsigned int mask, const __nv_bfloat16 var, const int laneMask, const int width=warpSize)
Exchange a variable between threads within a warp.

__device__ __nv_bfloat162 *__shfl_xor_sync*(const unsigned int mask, const __nv_bfloat162 var, const int laneMask, const int width=warpSize)
Exchange a variable between threads within a warp.

__device__ __nv_bfloat16 *__short2bfloat16_rd*(const short int i)
Convert a signed short integer to a nv_bfloat16 in round-down mode.

__host__ __device__ __nv_bfloat16 *__short2bfloat16_rn*(const short int i)
Convert a signed short integer to a nv_bfloat16 in round-to-nearest-even mode.

__device__ __nv_bfloat16 *__short2bfloat16_ru*(const short int i)
Convert a signed short integer to a nv_bfloat16 in round-up mode.

__device__ __nv_bfloat16 *__short2bfloat16_rz*(const short int i)
Convert a signed short integer to a nv_bfloat16 in round-towards-zero mode.

__host__ __device__ __nv_bfloat16 *__short_as_bfloat16*(const short int i)
Reinterprets bits in a signed short integer as a `nv_bfloat16`.

__device__ void *__stcg*(__nv_bfloat16 *const ptr, const __nv_bfloat16 value)
Generates a `st.global.cg` store instruction.

__device__ void *__stcg*(__nv_bfloat162 *const ptr, const __nv_bfloat162 value)
Generates a `st.global.cg` store instruction.

__device__ void *__stcs*(__nv_bfloat16 *const ptr, const __nv_bfloat16 value)
Generates a `st.global.cs` store instruction.

__device__ void *__stcs*(__nv_bfloat162 *const ptr, const __nv_bfloat162 value)
Generates a `st.global.cs` store instruction.

__device__ void *__stwb*(__nv_bfloat16 *const ptr, const __nv_bfloat16 value)
Generates a `st.global.wb` store instruction.

__device__ void *__stwb*(__nv_bfloat162 *const ptr, const __nv_bfloat162 value)
Generates a `st.global.wb` store instruction.

__device__ void *__stwt*(__nv_bfloat162 *const ptr, const __nv_bfloat162 value)
Generates a `st.global.wt` store instruction.

__device__ void *__stwt*(__nv_bfloat16 *const ptr, const __nv_bfloat16 value)
Generates a `st.global.wt` store instruction.

__device__ __nv_bfloat16 *__uint2bfloat16_rd*(const unsigned int i)
Convert an unsigned integer to a nv_bfloat16 in round-down mode.

__host__ __device__ __nv_bfloat16 *__uint2bfloat16_rn*(const unsigned int i)
Convert an unsigned integer to a nv_bfloat16 in round-to-nearest-even mode.

__device__ __nv_bfloat16 *__uint2bfloat16_ru*(const unsigned int i)
Convert an unsigned integer to a nv_bfloat16 in round-up mode.

__device__ __nv_bfloat16 *__uint2bfloat16_rz*(const unsigned int i)
Convert an unsigned integer to a nv_bfloat16 in round-towards-zero mode.

__device__ __nv_bfloat16 *__ull2bfloat16_rd*(const unsigned long long int i)
Convert an unsigned 64-bit integer to a nv_bfloat16 in round-down mode.

__host__ __device__ __nv_bfloat16 *__ull2bfloat16_rn*(const unsigned long long int i)
Convert an unsigned 64-bit integer to a nv_bfloat16 in round-to-nearest-even mode.

__device__ __nv_bfloat16 *__ull2bfloat16_ru*(const unsigned long long int i)
Convert an unsigned 64-bit integer to a nv_bfloat16 in round-up mode.

__device__ __nv_bfloat16 *__ull2bfloat16_rz*(const unsigned long long int i)
Convert an unsigned 64-bit integer to a nv_bfloat16 in round-towards-zero mode.

__device__ __nv_bfloat16 *__ushort2bfloat16_rd*(const unsigned short int i)
Convert an unsigned short integer to a nv_bfloat16 in round-down mode.

__host__ __device__ __nv_bfloat16 *__ushort2bfloat16_rn*(const unsigned short int i)
Convert an unsigned short integer to a nv_bfloat16 in round-to-nearest-even mode.

__device__ __nv_bfloat16 *__ushort2bfloat16_ru*(const unsigned short int i)
Convert an unsigned short integer to a nv_bfloat16 in round-up mode.

__device__ __nv_bfloat16 *__ushort2bfloat16_rz*(const unsigned short int i)
Convert an unsigned short integer to a nv_bfloat16 in round-towards-zero mode.

__host__ __device__ __nv_bfloat16 *__ushort_as_bfloat16*(const unsigned short int i)
Reinterprets bits in an unsigned short integer as a `nv_bfloat16` .

__host__ __device__ __nv_bfloat162 *make_bfloat162*(const __nv_bfloat16 x, const __nv_bfloat16 y)
Vector function, combines two `nv_bfloat16` numbers into one `nv_bfloat162` number.

5.5.1. Functions

__host__ __device__ float2 **`__bfloat1622float2`**(const *__nv_bfloat162* a)

Converts both halves of `nv_bfloat162` to float2 and returns the result.

Converts both halves of `nv_bfloat162` input a to float and returns the result as a `float2` packed value.

See also:

__bfloat162float(__nv_bfloat16) for further details.

Parameters
a – **[in]** - nv_bfloat162. Is only being read.

Returns
float2

- a converted to float2.

__host__ __device__ *__nv_bfloat162* **`__bfloat162bfloat162`**(const *__nv_bfloat16* a)

Returns `nv_bfloat162` with both halves equal to the input value.

Returns `nv_bfloat162` number with both halves equal to the input a `nv_bfloat16` number.

Parameters
a – **[in]** - nv_bfloat16. Is only being read.

Returns
nv_bfloat162

- The vector which has both its halves equal to the input a.

__host__ __device__ signed char **`__bfloat162char_rz`**(const *__nv_bfloat16* h)

Convert a nv_bfloat16 to a signed char in round-towards-zero mode.

Convert the nv_bfloat16 floating-point value h to a signed char in round-towards-zero mode. NaN inputs are converted to 0.

Parameters
h – **[in]** - nv_bfloat16. Is only being read.

Returns
signed char

- h converted to a signed char using round-towards-zero mode.
- __bfloat162char_rz (± 0) returns 0.

- __bfloat162char_rz $(x), x > 127$ returns SCHAR_MAX = `0x7F`.
- __bfloat162char_rz $(x), x < -128$ returns SCHAR_MIN = `0x80`.
- __bfloat162char_rz(NaN) returns 0.

__host__ __device__ float **__bfloat162float**(const *__nv_bfloat16* a)

Converts `nv_bfloat16` number to float.

Converts nv_bfloat16 number a to float.

Parameters

a – **[in]** - float. Is only being read.

Returns

float

- a converted to float.
- __bfloat162float (± 0) returns ± 0.
- __bfloat162float $(\pm\infty)$ returns $\pm\infty$.
- __bfloat162float(NaN) returns NaN.

__device__ int **__bfloat162int_rd**(const *__nv_bfloat16* h)

Convert a nv_bfloat16 to a signed integer in round-down mode.

Convert the nv_bfloat16 floating-point value h to a signed integer in round-down mode. NaN inputs are converted to 0.

Parameters

h – **[in]** - nv_bfloat16. Is only being read.

Returns

int

- h converted to a signed integer using round-down mode.
- __bfloat162int_rd (± 0) returns 0.
- __bfloat162int_rd $(x), x > INT_MAX$ returns INT_MAX = `0x7FFFFFFF`.
- __bfloat162int_rd $(x), x < INT_MIN$ returns INT_MIN = `0x80000000`.
- __bfloat162int_rd(NaN) returns 0.*

__device__ int **__bfloat162int_rn**(const *__nv_bfloat16* h)

Convert a nv_bfloat16 to a signed integer in round-to-nearest-even mode.

Convert the nv_bfloat16 floating-point value h to a signed integer in round-to-nearest-even mode. NaN inputs are converted to 0.

Parameters

h – **[in]** - nv_bfloat16. Is only being read.

Returns

int

- h converted to a signed integer using round-to-nearest-even mode.
- __bfloat162int_rn (± 0) returns 0.
- __bfloat162int_rn $(x), x > INT_MAX$ returns INT_MAX = `0x7FFFFFFF`.
- __bfloat162int_rn $(x), x < INT_MIN$ returns INT_MIN = `0x80000000`.

- __bfloat162int_rn(NaN) returns 0.

__device__ int **__bfloat162int_ru**(const *__nv_bfloat16* h)

Convert a nv_bfloat16 to a signed integer in round-up mode.

Convert the nv_bfloat16 floating-point value h to a signed integer in round-up mode. NaN inputs are converted to 0.

Parameters

h – **[in]** - nv_bfloat16. Is only being read.

Returns

int

- h converted to a signed integer using round-up mode.
- __bfloat162int_ru (± 0) returns 0.
- __bfloat162int_ru $(x), x > INT_MAX$ returns INT_MAX = `0x7FFFFFFF`.
- __bfloat162int_ru $(x), x < INT_MIN$ returns INT_MIN = `0x80000000`.
- __bfloat162int_ru(NaN) returns 0.

__host__ __device__ int **__bfloat162int_rz**(const *__nv_bfloat16* h)

Convert a nv_bfloat16 to a signed integer in round-towards-zero mode.

Convert the nv_bfloat16 floating-point value h to a signed integer in round-towards-zero mode. NaN inputs are converted to 0.

Parameters

h – **[in]** - nv_bfloat16. Is only being read.

Returns

int

- h converted to a signed integer using round-towards-zero mode.
- __bfloat162int_rz (± 0) returns 0.
- __bfloat162int_rz $(x), x > INT_MAX$ returns INT_MAX = `0x7FFFFFFF`.
- __bfloat162int_rz $(x), x < INT_MIN$ returns INT_MIN = `0x80000000`.
- __bfloat162int_rz(NaN) returns 0.

__device__ long long int **__bfloat162ll_rd**(const *__nv_bfloat16* h)

Convert a nv_bfloat16 to a signed 64-bit integer in round-down mode.

Convert the nv_bfloat16 floating-point value h to a signed 64-bit integer in round-down mode. NaN inputs return a long long int with hex value of 0x8000000000000000.

Parameters

h – **[in]** - nv_bfloat16. Is only being read.

Returns

long long int

- h converted to a signed 64-bit integer.

__device__ long long int **__bfloat162ll_rn**(const *__nv_bfloat16* h)

Convert a nv_bfloat16 to a signed 64-bit integer in round-to-nearest-even mode.

Convert the nv_bfloat16 floating-point value h to a signed 64-bit integer in round-to-nearest-even mode. NaN inputs return a long long int with hex value of 0x8000000000000000.

Parameters
h – **[in]** - nv_bfloat16. Is only being read.

Returns
long long int

- h converted to a signed 64-bit integer.

__device__ long long int **__bfloat162ll_ru**(const *__nv_bfloat16* h)

Convert a nv_bfloat16 to a signed 64-bit integer in round-up mode.

Convert the nv_bfloat16 floating-point value h to a signed 64-bit integer in round-up mode. NaN inputs return a long long int with hex value of 0x8000000000000000.

Parameters
h – **[in]** - nv_bfloat16. Is only being read.

Returns
long long int

- h converted to a signed 64-bit integer.

__host__ __device__ long long int **__bfloat162ll_rz**(const *__nv_bfloat16* h)

Convert a nv_bfloat16 to a signed 64-bit integer in round-towards-zero mode.

Convert the nv_bfloat16 floating-point value h to a signed 64-bit integer in round-towards-zero mode. NaN inputs return a long long int with hex value of 0x8000000000000000.

Parameters
h – **[in]** - nv_bfloat16. Is only being read.

Returns
long long int

- h converted to a signed 64-bit integer.

__device__ short int **__bfloat162short_rd**(const *__nv_bfloat16* h)

Convert a nv_bfloat16 to a signed short integer in round-down mode.

Convert the nv_bfloat16 floating-point value h to a signed short integer in round-down mode. NaN inputs are converted to 0.

Parameters
h – **[in]** - nv_bfloat16. Is only being read.

Returns
short int

- h converted to a signed short integer using round-down mode.
- __bfloat162short_rd (± 0) returns 0.
- __bfloat162short_rd $(x), x > 32767$ returns SHRT_MAX = `0x7FFF`.
- __bfloat162short_rd $(x), x < -32768$ returns SHRT_MIN = `0x8000`.
- __bfloat162short_rd(NaN) returns 0.

__device__ short int **__bfloat162short_rn**(const *__nv_bfloat16* h)

Convert a nv_bfloat16 to a signed short integer in round-to-nearest-even mode.

Convert the nv_bfloat16 floating-point value h to a signed short integer in round-to-nearest-even mode. NaN inputs are converted to 0.

Parameters
h – **[in]** - nv_bfloat16. Is only being read.

Returns
short int

- h converted to a signed short integer using round-to-nearest-even mode.
- __bfloat162short_rn (± 0) returns 0.
- __bfloat162short_rn $(x), x > 32767$ returns SHRT_MAX = `0x7FFF`.
- __bfloat162short_rn $(x), x < -32768$ returns SHRT_MIN = `0x8000`.
- __bfloat162short_rn(NaN) returns 0.

__device__ short int **__bfloat162short_ru**(const *__nv_bfloat16* h)

Convert a nv_bfloat16 to a signed short integer in round-up mode.

Convert the nv_bfloat16 floating-point value h to a signed short integer in round-up mode. NaN inputs are converted to 0.

Parameters
h – **[in]** - nv_bfloat16. Is only being read.

Returns
short int

- h converted to a signed short integer using round-up mode.
- __bfloat162short_ru (± 0) returns 0.
- __bfloat162short_ru $(x), x > 32767$ returns SHRT_MAX = `0x7FFF`.
- __bfloat162short_ru $(x), x < -32768$ returns SHRT_MIN = `0x8000`.
- __bfloat162short_ru(NaN) returns 0.

__host__ __device__ short int **__bfloat162short_rz**(const *__nv_bfloat16* h)

Convert a nv_bfloat16 to a signed short integer in round-towards-zero mode.

Convert the nv_bfloat16 floating-point value h to a signed short integer in round-towards-zero mode. NaN inputs are converted to 0.

Parameters
h – **[in]** - nv_bfloat16. Is only being read.

Returns
short int

- h converted to a signed short integer using round-towards-zero mode.
- __bfloat162short_rz (± 0) returns 0.
- __bfloat162short_rz $(x), x > 32767$ returns SHRT_MAX = `0x7FFF`.
- __bfloat162short_rz $(x), x < -32768$ returns SHRT_MIN = `0x8000`.
- __bfloat162short_rz(NaN) returns 0.

__host__ __device__ unsigned char **__bfloat162uchar_rz**(const *__nv_bfloat16* h)

Convert a nv_bfloat16 to an unsigned char in round-towards-zero mode.

Convert the nv_bfloat16 floating-point value h to an unsigned char in round-towards-zero mode. NaN inputs are converted to 0.

Parameters
h – **[in]** - nv_bfloat16. Is only being read.

Returns
unsigned char

- h converted to an unsigned char using round-towards-zero mode.
- __bfloat162uchar_rz (± 0) returns 0.
- __bfloat162uchar_rz $(x), x > 255$ returns UCHAR_MAX = `0x`FF.
- __bfloat162uchar_rz $(x), x < 0.0$ returns 0.
- __bfloat162uchar_rz(NaN) returns 0.

__device__ unsigned int **`__bfloat162uint_rd`**(const *__nv_bfloat16* h)

Convert a nv_bfloat16 to an unsigned integer in round-down mode.

Convert the nv_bfloat16 floating-point value h to an unsigned integer in round-down mode. NaN inputs are converted to 0.

Parameters
h – **[in]** - nv_bfloat16. Is only being read.

Returns
unsigned int

- h converted to an unsigned integer.

__device__ unsigned int **`__bfloat162uint_rn`**(const *__nv_bfloat16* h)

Convert a nv_bfloat16 to an unsigned integer in round-to-nearest-even mode.

Convert the nv_bfloat16 floating-point value h to an unsigned integer in round-to-nearest-even mode. NaN inputs are converted to 0.

Parameters
h – **[in]** - nv_bfloat16. Is only being read.

Returns
unsigned int

- h converted to an unsigned integer.

__device__ unsigned int **`__bfloat162uint_ru`**(const *__nv_bfloat16* h)

Convert a nv_bfloat16 to an unsigned integer in round-up mode.

Convert the nv_bfloat16 floating-point value h to an unsigned integer in round-up mode. NaN inputs are converted to 0.

Parameters
h – **[in]** - nv_bfloat16. Is only being read.

Returns
unsigned int

- h converted to an unsigned integer.

__host__ __device__ unsigned int **`__bfloat162uint_rz`**(const *__nv_bfloat16* h)

Convert a nv_bfloat16 to an unsigned integer in round-towards-zero mode.

Convert the nv_bfloat16 floating-point value h to an unsigned integer in round-towards-zero mode. NaN inputs are converted to 0.

Parameters
h – **[in]** - nv_bfloat16. Is only being read.

Returns
unsigned int

- h converted to an unsigned integer.

__device__ unsigned long long int **__bfloat162ull_rd**(const *__nv_bfloat16* h)

Convert a nv_bfloat16 to an unsigned 64-bit integer in round-down mode.

Convert the nv_bfloat16 floating-point value h to an unsigned 64-bit integer in round-down mode. NaN inputs return 0x8000000000000000.

Parameters
h – **[in]** - nv_bfloat16. Is only being read.

Returns
unsigned long long int

- h converted to an unsigned 64-bit integer.

__device__ unsigned long long int **__bfloat162ull_rn**(const *__nv_bfloat16* h)

Convert a nv_bfloat16 to an unsigned 64-bit integer in round-to-nearest-even mode.

Convert the nv_bfloat16 floating-point value h to an unsigned 64-bit integer in round-to-nearest-even mode. NaN inputs return 0x8000000000000000.

Parameters
h – **[in]** - nv_bfloat16. Is only being read.

Returns
unsigned long long int

- h converted to an unsigned 64-bit integer.

__device__ unsigned long long int **__bfloat162ull_ru**(const *__nv_bfloat16* h)

Convert a nv_bfloat16 to an unsigned 64-bit integer in round-up mode.

Convert the nv_bfloat16 floating-point value h to an unsigned 64-bit integer in round-up mode. NaN inputs return 0x8000000000000000.

Parameters
h – **[in]** - nv_bfloat16. Is only being read.

Returns
unsigned long long int

- h converted to an unsigned 64-bit integer.

__host__ __device__ unsigned long long int **__bfloat162ull_rz**(const *__nv_bfloat16* h)

Convert a nv_bfloat16 to an unsigned 64-bit integer in round-towards-zero mode.

Convert the nv_bfloat16 floating-point value h to an unsigned 64-bit integer in round-towards-zero mode. NaN inputs return 0x8000000000000000.

Parameters
h – **[in]** - nv_bfloat16. Is only being read.

Returns
unsigned long long int

- h converted to an unsigned 64-bit integer.

__device__ unsigned short int **__bfloat162ushort_rd**(const *__nv_bfloat16* h)

Convert a nv_bfloat16 to an unsigned short integer in round-down mode.

Convert the nv_bfloat16 floating-point value h to an unsigned short integer in round-down mode. NaN inputs are converted to 0.

Parameters

h – **[in]** - nv_bfloat16. Is only being read.

Returns

unsigned short int

- h converted to an unsigned short integer.

__device__ unsigned short int **__bfloat162ushort_rn**(const *__nv_bfloat16* h)

Convert a nv_bfloat16 to an unsigned short integer in round-to-nearest-even mode.

Convert the nv_bfloat16 floating-point value h to an unsigned short integer in round-to-nearest-even mode. NaN inputs are converted to 0.

Parameters

h – **[in]** - nv_bfloat16. Is only being read.

Returns

unsigned short int

- h converted to an unsigned short integer.

__device__ unsigned short int **__bfloat162ushort_ru**(const *__nv_bfloat16* h)

Convert a nv_bfloat16 to an unsigned short integer in round-up mode.

Convert the nv_bfloat16 floating-point value h to an unsigned short integer in round-up mode. NaN inputs are converted to 0.

Parameters

h – **[in]** - nv_bfloat16. Is only being read.

Returns

unsigned short int

- h converted to an unsigned short integer.

__host__ __device__ unsigned short int **__bfloat162ushort_rz**(const *__nv_bfloat16* h)

Convert a nv_bfloat16 to an unsigned short integer in round-towards-zero mode.

Convert the nv_bfloat16 floating-point value h to an unsigned short integer in round-towards-zero mode. NaN inputs are converted to 0.

Parameters

h – **[in]** - nv_bfloat16. Is only being read.

Returns

unsigned short int

- h converted to an unsigned short integer.

__host__ __device__ short int **__bfloat16_as_short**(const *__nv_bfloat16* h)

Reinterprets bits in a `nv_bfloat16` as a signed short integer.

Reinterprets the bits in the nv_bfloat16 floating-point number h as a signed short integer.

Parameters

h – **[in]** - nv_bfloat16. Is only being read.

Returns

short int

- The reinterpreted value.

__host__ __device__ unsigned short int **__bfloat16_as_ushort**(const *__nv_bfloat16* h)

Reinterprets bits in a `nv_bfloat16` as an unsigned short integer.

Reinterprets the bits in the nv_bfloat16 floating-point h as an unsigned short number.

Parameters

h – **[in]** - nv_bfloat16. Is only being read.

Returns

unsigned short int

- The reinterpreted value.

__host__ __device__ *__nv_bfloat16* **__double2bfloat16**(const double a)

Converts double number to nv_bfloat16 precision in round-to-nearest-even mode and returns `nv_bfloat16` with converted value.

Converts double number a to nv_bfloat16 precision in round-to-nearest-even mode.

Parameters

a – **[in]** - double. Is only being read.

Returns

nv_bfloat16

- a converted to `nv_bfloat16` using round-to-nearest-even mode.
- __double2bfloat16 (± 0) returns ± 0.
- __double2bfloat16 $(\pm\infty)$ returns $\pm\infty$.
- __double2bfloat16(NaN) returns NaN.

__host__ __device__ *__nv_bfloat162* **__float22bfloat162_rn**(const float2 a)

Converts both components of float2 number to nv_bfloat16 precision in round-to-nearest-even mode and returns `nv_bfloat162` with converted values.

Converts both components of float2 to nv_bfloat16 precision in round-to-nearest-even mode and combines the results into one `nv_bfloat162` number. Low 16 bits of the return value correspond to `a.x` and high 16 bits of the return value correspond to `a.y`.

See also:

__float2bfloat16_rn(float) for further details.

Parameters

a – **[in]** - float2. Is only being read.

Returns

nv_bfloat162

- The `nv_bfloat162` which has corresponding halves equal to the converted float2 components.

__host__ __device__ *__nv_bfloat16* **__float2bfloat16**(const float a)

Converts float number to nv_bfloat16 precision in round-to-nearest-even mode and returns `nv_bfloat16` with converted value.

Converts float number a to nv_bfloat16 precision in round-to-nearest-even mode.

See also:

__float2bfloat16_rn(float) for further details.

Parameters

a – **[in]** - float. Is only being read.

Returns

nv_bfloat16

- a converted to nv_bfloat16 using round-to-nearest-even mode.

__host__ __device__ *__nv_bfloat162* **`__float2bfloat162_rn`**(const float a)

Converts input to nv_bfloat16 precision in round-to-nearest-even mode and populates both halves of `nv_bfloat162` with converted value.

Converts input a to nv_bfloat16 precision in round-to-nearest-even mode and populates both halves of `nv_bfloat162` with converted value.

See also:

__float2bfloat16_rn(float) for further details.

Parameters

a – **[in]** - float. Is only being read.

Returns

nv_bfloat162

- The `nv_bfloat162` value with both halves equal to the converted nv_bfloat16 precision number.

__host__ __device__ *__nv_bfloat16* **`__float2bfloat16_rd`**(const float a)

Converts float number to nv_bfloat16 precision in round-down mode and returns `nv_bfloat16` with converted value.

Converts float number a to nv_bfloat16 precision in round-down mode.

Parameters

a – **[in]** - float. Is only being read.

Returns

nv_bfloat16

- a converted to nv_bfloat16 using round-down mode.
- __float2bfloat16_rd (± 0) returns ± 0.
- __float2bfloat16_rd $(\pm\infty)$ returns $\pm\infty$.
- __float2bfloat16_rd(NaN) returns NaN.

__host__ __device__ *__nv_bfloat16* **`__float2bfloat16_rn`**(const float a)

Converts float number to nv_bfloat16 precision in round-to-nearest-even mode and returns `nv_bfloat16` with converted value.

Converts float number a to nv_bfloat16 precision in round-to-nearest-even mode.

Parameters

a – **[in]** - float. Is only being read.

Returns

nv_bfloat16

- a converted to nv_bfloat16 using round-to-nearest-even mode.
- __float2bfloat16_rn (± 0) returns ± 0.
- __float2bfloat16_rn $(\pm\infty)$ returns $\pm\infty$.
- __float2bfloat16_rn(NaN) returns NaN.

__host__ __device__ *__nv_bfloat16* **__float2bfloat16_ru**(const float a)

Converts float number to nv_bfloat16 precision in round-up mode and returns `nv_bfloat16` with converted value.

Converts float number a to nv_bfloat16 precision in round-up mode.

Parameters

a – **[in]** - float. Is only being read.

Returns

nv_bfloat16

- a converted to nv_bfloat16 using round-up mode.
- __float2bfloat16_ru (± 0) returns ± 0.
- __float2bfloat16_ru $(\pm\infty)$ returns $\pm\infty$.
- __float2bfloat16_ru(NaN) returns NaN.

__host__ __device__ *__nv_bfloat16* **__float2bfloat16_rz**(const float a)

Converts float number to nv_bfloat16 precision in round-towards-zero mode and returns `nv_bfloat16` with converted value.

Converts float number a to nv_bfloat16 precision in round-towards-zero mode.

Parameters

a – **[in]** - float. Is only being read.

Returns

nv_bfloat16

- a converted to nv_bfloat16 using round-towards-zero mode.
- __float2bfloat16_rz (± 0) returns ± 0.
- __float2bfloat16_rz $(\pm\infty)$ returns $\pm\infty$.
- __float2bfloat16_rz(NaN) returns NaN.

__host__ __device__ *__nv_bfloat162* **__floats2bfloat162_rn**(const float a, const float b)

Converts both input floats to nv_bfloat16 precision in round-to-nearest-even mode and returns `nv_bfloat162` with converted values.

Converts both input floats to nv_bfloat16 precision in round-to-nearest-even mode and combines the results into one `nv_bfloat162` number. Low 16 bits of the return value correspond to the input a, high 16 bits correspond to the input b.

See also:

__float2bfloat16_rn(float) for further details.

Parameters

- **a** – **[in]** - float. Is only being read.
- **b** – **[in]** - float. Is only being read.

Returns

nv_bfloat162

- The `nv_bfloat162` value with corresponding halves equal to the converted input floats.

__host__ __device__ *__nv_bfloat162* **`__halves2bfloat162`**(const *__nv_bfloat16* a, const *__nv_bfloat16* b)

Combines two `nv_bfloat16` numbers into one `nv_bfloat162` number.

Combines two input `nv_bfloat16` number a and b into one `nv_bfloat162` number. Input a is stored in low 16 bits of the return value, input b is stored in high 16 bits of the return value.

Parameters

- **a** – **[in]** - nv_bfloat16. Is only being read.
- **b** – **[in]** - nv_bfloat16. Is only being read.

Returns

nv_bfloat162

- The nv_bfloat162 with one nv_bfloat16 equal to a and the other to b.

__host__ __device__ *__nv_bfloat16* **`__high2bfloat16`**(const *__nv_bfloat162* a)

Returns high 16 bits of `nv_bfloat162` input.

Returns high 16 bits of `nv_bfloat162` input a.

Parameters

a – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat16

- The high 16 bits of the input.

__host__ __device__ *__nv_bfloat162* **`__high2bfloat162`**(const *__nv_bfloat162* a)

Extracts high 16 bits from `nv_bfloat162` input.

Extracts high 16 bits from `nv_bfloat162` input a and returns a new `nv_bfloat162` number which has both halves equal to the extracted bits.

Parameters

a – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

- The nv_bfloat162 with both halves equal to the high 16 bits of the input.

__host__ __device__ float **`__high2float`**(const *__nv_bfloat162* a)

Converts high 16 bits of `nv_bfloat162` to float and returns the result.

Converts high 16 bits of `nv_bfloat162` input a to 32-bit floating-point number and returns the result.

See also:

__bfloat162float(__nv_bfloat16) for further details.

Parameters

a – **[in]** - nv_bfloat162. Is only being read.

Returns

float

- The high 16 bits of a converted to float.

__host__ __device__ *__nv_bfloat162* **__highs2bfloat162**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Extracts high 16 bits from each of the two `nv_bfloat162` inputs and combines into one `nv_bfloat162` number.

Extracts high 16 bits from each of the two `nv_bfloat162` inputs and combines into one `nv_bfloat162` number. High 16 bits from input a is stored in low 16 bits of the return value, high 16 bits from input b is stored in high 16 bits of the return value.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

- The high 16 bits of a and of b.

__device__ *__nv_bfloat16* **__int2bfloat16_rd**(const int i)

Convert a signed integer to a nv_bfloat16 in round-down mode.

Convert the signed integer value `i` to a nv_bfloat16 floating-point value in round-down mode.

Parameters

i – **[in]** - int. Is only being read.

Returns

nv_bfloat16

- `i` converted to nv_bfloat16.

__host__ __device__ *__nv_bfloat16* **__int2bfloat16_rn**(const int i)

Convert a signed integer to a nv_bfloat16 in round-to-nearest-even mode.

Convert the signed integer value `i` to a nv_bfloat16 floating-point value in round-to-nearest-even mode.

Parameters

i – **[in]** - int. Is only being read.

Returns

nv_bfloat16

- `i` converted to nv_bfloat16.

__device__ *__nv_bfloat16* **__int2bfloat16_ru**(const int i)

Convert a signed integer to a nv_bfloat16 in round-up mode.

Convert the signed integer value `i` to a nv_bfloat16 floating-point value in round-up mode.

Parameters

i – **[in]** - int. Is only being read.

Returns

nv_bfloat16

- `i` converted to nv_bfloat16.

__device__ *__nv_bfloat16* **`__int2bfloat16_rz`**(const int i)

Convert a signed integer to a nv_bfloat16 in round-towards-zero mode.

Convert the signed integer value `i` to a nv_bfloat16 floating-point value in round-towards-zero mode.

Parameters

`i` – **[in]** - int. Is only being read.

Returns

nv_bfloat16

- `i` converted to nv_bfloat16.

__device__ *__nv_bfloat162* **`__ldca`**(const *__nv_bfloat162* *const ptr)

Generates a `ld.global.ca` load instruction.

Parameters

`ptr` – **[in]** - memory location

Returns

The value pointed by `ptr`

__device__ *__nv_bfloat16* **`__ldca`**(const *__nv_bfloat16* *const ptr)

Generates a `ld.global.ca` load instruction.

Parameters

`ptr` – **[in]** - memory location

Returns

The value pointed by `ptr`

__device__ *__nv_bfloat16* **`__ldcg`**(const *__nv_bfloat16* *const ptr)

Generates a `ld.global.cg` load instruction.

Parameters

`ptr` – **[in]** - memory location

Returns

The value pointed by `ptr`

__device__ *__nv_bfloat162* **`__ldcg`**(const *__nv_bfloat162* *const ptr)

Generates a `ld.global.cg` load instruction.

Parameters

`ptr` – **[in]** - memory location

Returns

The value pointed by `ptr`

__device__ *__nv_bfloat162* **`__ldcs`**(const *__nv_bfloat162* *const ptr)

Generates a `ld.global.cs` load instruction.

Parameters

`ptr` – **[in]** - memory location

Returns

The value pointed by `ptr`

__device__ *__nv_bfloat16* **__ldcs**(const *__nv_bfloat16* *const ptr)

Generates a `ld.global.cs` load instruction.

Parameters

ptr – **[in]** - memory location

Returns

The value pointed by `ptr`

__device__ *__nv_bfloat16* **__ldcv**(const *__nv_bfloat16* *const ptr)

Generates a `ld.global.cv` load instruction.

Parameters

ptr – **[in]** - memory location

Returns

The value pointed by `ptr`

__device__ *__nv_bfloat162* **__ldcv**(const *__nv_bfloat162* *const ptr)

Generates a `ld.global.cv` load instruction.

Parameters

ptr – **[in]** - memory location

Returns

The value pointed by `ptr`

__device__ *__nv_bfloat162* **__ldg**(const *__nv_bfloat162* *const ptr)

Generates a `ld.global.nc` load instruction.

Parameters

ptr – **[in]** - memory location

Returns

The value pointed by `ptr`

__device__ *__nv_bfloat16* **__ldg**(const *__nv_bfloat16* *const ptr)

Generates a `ld.global.nc` load instruction.

Parameters

ptr – **[in]** - memory location

Returns

The value pointed by `ptr`

__device__ *__nv_bfloat162* **__ldlu**(const *__nv_bfloat162* *const ptr)

Generates a `ld.global.lu` load instruction.

Parameters

ptr – **[in]** - memory location

Returns

The value pointed by `ptr`

__device__ *__nv_bfloat16* **__ldlu**(const *__nv_bfloat16* *const ptr)

Generates a `ld.global.lu` load instruction.

Parameters

ptr – **[in]** - memory location

Returns

The value pointed by `ptr`

__device__ *__nv_bfloat16* **__ll2bfloat16_rd**(const long long int i)

Convert a signed 64-bit integer to a nv_bfloat16 in round-down mode.

Convert the signed 64-bit integer value `i` to a nv_bfloat16 floating-point value in round-down mode.

Parameters

i – **[in]** - long long int. Is only being read.

Returns

nv_bfloat16

- `i` converted to nv_bfloat16.

__host__ __device__ *__nv_bfloat16* **__ll2bfloat16_rn**(const long long int i)

Convert a signed 64-bit integer to a nv_bfloat16 in round-to-nearest-even mode.

Convert the signed 64-bit integer value `i` to a nv_bfloat16 floating-point value in round-to-nearest-even mode.

Parameters

i – **[in]** - long long int. Is only being read.

Returns

nv_bfloat16

- `i` converted to nv_bfloat16.

__device__ *__nv_bfloat16* **__ll2bfloat16_ru**(const long long int i)

Convert a signed 64-bit integer to a nv_bfloat16 in round-up mode.

Convert the signed 64-bit integer value `i` to a nv_bfloat16 floating-point value in round-up mode.

Parameters

i – **[in]** - long long int. Is only being read.

Returns

nv_bfloat16

- `i` converted to nv_bfloat16.

__device__ *__nv_bfloat16* **__ll2bfloat16_rz**(const long long int i)

Convert a signed 64-bit integer to a nv_bfloat16 in round-towards-zero mode.

Convert the signed 64-bit integer value `i` to a nv_bfloat16 floating-point value in round-towards-zero mode.

Parameters

i – **[in]** - long long int. Is only being read.

Returns

nv_bfloat16

- `i` converted to nv_bfloat16.

__host__ __device__ *__nv_bfloat16* **__low2bfloat16**(const *__nv_bfloat162* a)

Returns low 16 bits of `nv_bfloat162` input.

Returns low 16 bits of `nv_bfloat162` input a.

Parameters

a – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat16

- Returns `nv_bfloat16` which contains low 16 bits of the input a.

__host__ __device__ *__nv_bfloat162* **`__low2bfloat162`**(const *__nv_bfloat162* a)

Extracts low 16 bits from `nv_bfloat162` input.

Extracts low 16 bits from `nv_bfloat162` input a and returns a new `nv_bfloat162` number which has both halves equal to the extracted bits.

Parameters

a – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

- The nv_bfloat162 with both halves equal to the low 16 bits of the input.

__host__ __device__ float **`__low2float`**(const *__nv_bfloat162* a)

Converts low 16 bits of `nv_bfloat162` to float and returns the result.

Converts low 16 bits of `nv_bfloat162` input a to 32-bit floating-point number and returns the result.

See also:

__bfloat162float(__nv_bfloat16) for further details.

Parameters

a – **[in]** - nv_bfloat162. Is only being read.

Returns

float

- The low 16 bits of a converted to float.

__host__ __device__ *__nv_bfloat162* **`__lowhigh2highlow`**(const *__nv_bfloat162* a)

Swaps both halves of the `nv_bfloat162` input.

Swaps both halves of the `nv_bfloat162` input and returns a new `nv_bfloat162` number with swapped halves.

Parameters

a – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

- a with its halves being swapped.

__host__ __device__ *__nv_bfloat162* **`__lows2bfloat162`**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Extracts low 16 bits from each of the two `nv_bfloat162` inputs and combines into one `nv_bfloat162` number.

Extracts low 16 bits from each of the two `nv_bfloat162` inputs and combines into one `nv_bfloat162` number. Low 16 bits from input a is stored in low 16 bits of the return value, low 16 bits from input b is stored in high 16 bits of the return value.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

- The low 16 bits of a and of b.

__device__ *__nv_bfloat162* **__shfl_down_sync**(const unsigned int mask, const *__nv_bfloat162* var, const unsigned int delta, const int width = warpSize)

Exchange a variable between threads within a warp.

Copy from a thread with higher ID relative to the caller.

Calculates a source thread ID by adding `delta` to the caller's thread ID. The value of `var` held by the resulting thread ID is returned: this has the effect of shifting `var` down the warp by `delta` threads. If the `width` is less than `warpSize`, then each subsection of the warp behaves as a separate entity with a starting logical thread ID of 0. Similarly to the *__shfl_up_sync()*, the ID number of the source thread will not wrap around the value of `width` and the upper `delta` threads will remain unchanged. Threads may only read data from another thread which is actively participating in the `__shfl_*sync()` command. If the target thread is inactive, the retrieved value is undefined.

Note: For more details about this function, see the Warp Shuffle Functions section in the CUDA C++ Programming Guide.

Parameters

- **`mask`** – **[in]** - unsigned int. Is only being read.
  - Indicates the threads participating in the call.
  - A bit, representing the thread's lane ID, must be set for each participating thread to ensure they are properly converged before the intrinsic is executed by the hardware.
  - Each calling thread must have its own bit set in the `mask` and all non-exited threads named in `mask` must execute the same intrinsic with the same `mask`, or the result is undefined.
- **`var`** – **[in]** - nv_bfloat162. Is only being read.
- **`delta`** – **[in]** - unsigned int. Is only being read.
- **`width`** – **[in]** - int. Is only being read.

Returns

Returns the 4-byte word referenced by `var` from the source thread ID as `nv_bfloat162`.

__device__ *__nv_bfloat16* **__shfl_down_sync**(const unsigned int mask, const *__nv_bfloat16* var, const unsigned int delta, const int width = warpSize)

Exchange a variable between threads within a warp.

Copy from a thread with higher ID relative to the caller.

Calculates a source thread ID by adding `delta` to the caller's thread ID. The value of `var` held by the resulting thread ID is returned: this has the effect of shifting `var` down the warp by `delta` threads. If the `width` is less than `warpSize`, then each subsection of the warp behaves as a separate entity with a starting logical thread ID of 0. Similarly to the *__shfl_up_sync()*, the ID number of the source thread will not wrap around the value of `width` and the upper `delta` threads will remain unchanged. Threads may only read data from another thread which is actively participating in the `__shfl_*sync()` command. If the target thread is inactive, the retrieved value is undefined.

Note: For more details about this function, see the Warp Shuffle Functions section in the CUDA C++ Programming Guide.

Parameters

- **`mask`** – **[in]** - unsigned int. Is only being read.
  - Indicates the threads participating in the call.
  - A bit, representing the thread's lane ID, must be set for each participating thread to ensure they are properly converged before the intrinsic is executed by the hardware.
  - Each calling thread must have its own bit set in the `mask` and all non-exited threads named in `mask` must execute the same intrinsic with the same `mask`, or the result is undefined.
- **`var`** – **[in]** - nv_bfloat16. Is only being read.
- **`delta`** – **[in]** - unsigned int. Is only being read.
- **`width`** – **[in]** - int. Is only being read.

Returns

Returns the 2-byte word referenced by `var` from the source thread ID as `nv_bfloat16`.

__device__ *__nv_bfloat162* **`__shfl_sync`**(const unsigned int mask, const *__nv_bfloat162* var, const int srcLane, const int width = warpSize)

Exchange a variable between threads within a warp.

Direct copy from indexed thread.

Returns the value of `var` held by the thread whose ID is given by `srcLane`. If the `width` is less than `warpSize`, then each subsection of the warp behaves as a separate entity with a starting logical thread ID of 0. If `srcLane` is outside the range [0:width-1], the value returned corresponds to the value of `var` held by the `srcLane` modulo `width` (i.e. within the same subsection). `width` must have a value which is a power of 2; results are undefined if `width` is not a power of 2, or is a number greater than `warpSize`. Threads may only read data from another thread which is actively participating in the `__shfl_*sync()` command. If the target thread is inactive, the retrieved value is undefined.

Note: For more details about this function, see the Warp Shuffle Functions section in the CUDA C++ Programming Guide.

Parameters

- **mask** – **[in]** - unsigned int. Is only being read.
  - Indicates the threads participating in the call.
  - A bit, representing the thread's lane ID, must be set for each participating thread to ensure they are properly converged before the intrinsic is executed by the hardware.
  - Each calling thread must have its own bit set in the mask and all non-exited threads named in mask must execute the same intrinsic with the same mask, or the result is undefined.
- **var** – **[in]** - nv_bfloat162. Is only being read.
- **srcLane** – **[in]** - int. Is only being read.
- **width** – **[in]** - int. Is only being read.

Returns

Returns the 4-byte word referenced by var from the source thread ID as nv_bfloat162.

__device__ *__nv_bfloat16* **__shfl_sync** (const unsigned int mask, const *__nv_bfloat16* var, const int srcLane, const int width = warpSize)

Exchange a variable between threads within a warp.

Direct copy from indexed thread.

Returns the value of var held by the thread whose ID is given by srcLane. If the width is less than warpSize, then each subsection of the warp behaves as a separate entity with a starting logical thread ID of 0. If srcLane is outside the range [0:width-1], the value returned corresponds to the value of var held by the srcLane modulo width (i.e. within the same subsection). width must have a value which is a power of 2; results are undefined if width is not a power of 2, or is a number greater than warpSize. Threads may only read data from another thread which is actively participating in the __shfl_*sync() command. If the target thread is inactive, the retrieved value is undefined.

Note: For more details about this function, see the Warp Shuffle Functions section in the CUDA C++ Programming Guide.

Parameters

- **mask** – **[in]** - unsigned int. Is only being read.
  - Indicates the threads participating in the call.
  - A bit, representing the thread's lane ID, must be set for each participating thread to ensure they are properly converged before the intrinsic is executed by the hardware.
  - Each calling thread must have its own bit set in the mask and all non-exited threads named in mask must execute the same intrinsic with the same mask, or the result is undefined.
- **var** – **[in]** - nv_bfloat16. Is only being read.

- **srcLane** – **[in]** - int. Is only being read.
- **width** – **[in]** - int. Is only being read.

Returns

Returns the 2-byte word referenced by `var` from the source thread ID as `nv_bfloat16`.

__device__ *__nv_bfloat16* **__shfl_up_sync**(const unsigned int mask, const *__nv_bfloat16* var, const unsigned int delta, const int width = warpSize)

Exchange a variable between threads within a warp.

Copy from a thread with lower ID relative to the caller.

Calculates a source thread ID by subtracting `delta` from the caller's lane ID. The value of `var` held by the resulting lane ID is returned: in effect, `var` is shifted up the warp by `delta` threads. If the `width` is less than `warpSize`, then each subsection of the warp behaves as a separate entity with a starting logical thread ID of 0. The source thread index will not wrap around the value of `width`, so effectively the lower `delta` threads will be unchanged. `width` must have a value which is a power of 2; results are undefined if `width` is not a power of 2, or is a number greater than `warpSize`. Threads may only read data from another thread which is actively participating in the `__shfl_*sync`() command. If the target thread is inactive, the retrieved value is undefined.

Note: For more details about this function, see the Warp Shuffle Functions section in the CUDA C++ Programming Guide.

Parameters

- **mask** – **[in]** - unsigned int. Is only being read.
  - Indicates the threads participating in the call.
  - A bit, representing the thread's lane ID, must be set for each participating thread to ensure they are properly converged before the intrinsic is executed by the hardware.
  - Each calling thread must have its own bit set in the `mask` and all non-exited threads named in `mask` must execute the same intrinsic with the same `mask`, or the result is undefined.
- **var** – **[in]** - nv_bfloat16. Is only being read.
- **delta** – **[in]** - unsigned int. Is only being read.
- **width** – **[in]** - int. Is only being read.

Returns

Returns the 2-byte word referenced by `var` from the source thread ID as `nv_bfloat16`.

__device__ *__nv_bfloat162* **__shfl_up_sync**(const unsigned int mask, const *__nv_bfloat162* var, const unsigned int delta, const int width = warpSize)

Exchange a variable between threads within a warp.

Copy from a thread with lower ID relative to the caller.

Calculates a source thread ID by subtracting `delta` from the caller's lane ID. The value of `var` held by the resulting lane ID is returned: in effect, `var` is shifted up the warp by `delta` threads. If the `width` is less than `warpSize`, then each subsection of the warp behaves as a separate entity with a starting logical thread ID of 0. The source thread index will not wrap around the value of `width`, so effectively the lower `delta` threads will be unchanged. `width` must have a value which is a power of 2; results are undefined if `width` is not a power of 2, or is a number greater than `warpSize`. Threads may only read data from another thread which is actively participating in the `__shfl_*sync()` command. If the target thread is inactive, the retrieved value is undefined.

Note: For more details about this function, see the Warp Shuffle Functions section in the CUDA C++ Programming Guide.

Parameters

- **`mask`** – **[in]** - unsigned int. Is only being read.
  - Indicates the threads participating in the call.
  - A bit, representing the thread's lane ID, must be set for each participating thread to ensure they are properly converged before the intrinsic is executed by the hardware.
  - Each calling thread must have its own bit set in the `mask` and all non-exited threads named in `mask` must execute the same intrinsic with the same `mask`, or the result is undefined.
- **`var`** – **[in]** - nv_bfloat162. Is only being read.
- **`delta`** – **[in]** - unsigned int. Is only being read.
- **`width`** – **[in]** - int. Is only being read.

Returns

Returns the 4-byte word referenced by `var` from the source thread ID as `nv_bfloat162`.

__device__ *__nv_bfloat16* **`__shfl_xor_sync`**(const unsigned int mask, const *__nv_bfloat16* var, const int laneMask, const int width = warpSize)

Exchange a variable between threads within a warp.

Copy from a thread based on bitwise XOR of own thread ID.

Calculates a source thread ID by performing a bitwise XOR of the caller's thread ID with `laneMask:` the value of `var` held by the resulting thread ID is returned. If the `width` is less than `warpSize`, then each group of `width` consecutive threads are able to access elements from earlier groups of threads, however if they attempt to access elements from later groups of threads their own value of `var` will be returned. This mode implements a butterfly addressing pattern such as is used in tree reduction and broadcast. Threads may only read data from another thread which is actively participating in the `__shfl_*sync()` command. If the target thread is inactive, the retrieved value is undefined.

Note: For more details about this function, see the Warp Shuffle Functions section in the CUDA C++ Programming Guide.

Parameters

- **mask** – **[in]** - unsigned int. Is only being read.
  - Indicates the threads participating in the call.
  - A bit, representing the thread's lane ID, must be set for each participating thread to ensure they are properly converged before the intrinsic is executed by the hardware.
  - Each calling thread must have its own bit set in the `mask` and all non-exited threads named in `mask` must execute the same intrinsic with the same `mask`, or the result is undefined.
- **var** – **[in]** - nv_bfloat16. Is only being read.
- **laneMask** – **[in]** - int. Is only being read.
- **width** – **[in]** - int. Is only being read.

Returns

Returns the 2-byte word referenced by `var` from the source thread ID as `nv_bfloat16`.

__device__ *__nv_bfloat162* **`__shfl_xor_sync`**(const unsigned int mask, const *__nv_bfloat162* var, const int laneMask, const int width = warpSize)

Exchange a variable between threads within a warp.

Copy from a thread based on bitwise XOR of own thread ID.

Calculates a source thread ID by performing a bitwise XOR of the caller's thread ID with `laneMask`: the value of `var` held by the resulting thread ID is returned. If the `width` is less than `warpSize`, then each group of `width` consecutive threads are able to access elements from earlier groups of threads, however if they attempt to access elements from later groups of threads their own value of `var` will be returned. This mode implements a butterfly addressing pattern such as is used in tree reduction and broadcast. Threads may only read data from another thread which is actively participating in the `__shfl_*sync`() command. If the target thread is inactive, the retrieved value is undefined.

Note: For more details about this function, see the Warp Shuffle Functions section in the CUDA C++ Programming Guide.

Parameters

- **mask** – **[in]** - unsigned int. Is only being read.
  - Indicates the threads participating in the call.
  - A bit, representing the thread's lane ID, must be set for each participating thread to ensure they are properly converged before the intrinsic is executed by the hardware.
  - Each calling thread must have its own bit set in the `mask` and all non-exited threads named in `mask` must execute the same intrinsic with the same `mask`, or the result is undefined.
- **var** – **[in]** - nv_bfloat162. Is only being read.
- **laneMask** – **[in]** - int. Is only being read.
- **width** – **[in]** - int. Is only being read.

Returns

Returns the 4-byte word referenced by `var` from the source thread ID as `nv_bfloat162`.

__device__ *__nv_bfloat16* **`__short2bfloat16_rd`**(const short int i)

Convert a signed short integer to a nv_bfloat16 in round-down mode.

Convert the signed short integer value `i` to a nv_bfloat16 floating-point value in round-down mode.

Parameters

i – **[in]** - short int. Is only being read.

Returns

nv_bfloat16

- `i` converted to nv_bfloat16.

__host__ __device__ *__nv_bfloat16* **`__short2bfloat16_rn`**(const short int i)

Convert a signed short integer to a nv_bfloat16 in round-to-nearest-even mode.

Convert the signed short integer value `i` to a nv_bfloat16 floating-point value in round-to-nearest-even mode.

Parameters

i – **[in]** - short int. Is only being read.

Returns

nv_bfloat16

- `i` converted to nv_bfloat16.

__device__ *__nv_bfloat16* **`__short2bfloat16_ru`**(const short int i)

Convert a signed short integer to a nv_bfloat16 in round-up mode.

Convert the signed short integer value `i` to a nv_bfloat16 floating-point value in round-up mode.

Parameters

i – **[in]** - short int. Is only being read.

Returns

nv_bfloat16

- `i` converted to nv_bfloat16.

__device__ *__nv_bfloat16* **`__short2bfloat16_rz`**(const short int i)

Convert a signed short integer to a nv_bfloat16 in round-towards-zero mode.

Convert the signed short integer value `i` to a nv_bfloat16 floating-point value in round-towards-zero mode.

Parameters

i – **[in]** - short int. Is only being read.

Returns

nv_bfloat16

- `i` converted to nv_bfloat16.

__host__ __device__ *__nv_bfloat16* **`__short_as_bfloat16`**(const short int i)

Reinterprets bits in a signed short integer as a `nv_bfloat16`.

Reinterprets the bits in the signed short integer `i` as a nv_bfloat16 floating-point number.

Parameters

i – **[in]** - short int. Is only being read.

Returns

nv_bfloat16

- The reinterpreted value.

__device__ void **__stcg**(*__nv_bfloat16* *const ptr, const *__nv_bfloat16* value)

Generates a `st.global.cg` store instruction.

Parameters

- **ptr** – **[out]** - memory location
- **value** – **[in]** - the value to be stored

__device__ void **__stcg**(*__nv_bfloat162* *const ptr, const *__nv_bfloat162* value)

Generates a `st.global.cg` store instruction.

Parameters

- **ptr** – **[out]** - memory location
- **value** – **[in]** - the value to be stored

__device__ void **__stcs**(*__nv_bfloat16* *const ptr, const *__nv_bfloat16* value)

Generates a `st.global.cs` store instruction.

Parameters

- **ptr** – **[out]** - memory location
- **value** – **[in]** - the value to be stored

__device__ void **__stcs**(*__nv_bfloat162* *const ptr, const *__nv_bfloat162* value)

Generates a `st.global.cs` store instruction.

Parameters

- **ptr** – **[out]** - memory location
- **value** – **[in]** - the value to be stored

__device__ void **__stwb**(*__nv_bfloat16* *const ptr, const *__nv_bfloat16* value)

Generates a `st.global.wb` store instruction.

Parameters

- **ptr** – **[out]** - memory location
- **value** – **[in]** - the value to be stored

__device__ void **__stwb**(*__nv_bfloat162* *const ptr, const *__nv_bfloat162* value)

Generates a `st.global.wb` store instruction.

Parameters

- **ptr** – **[out]** - memory location

- **value** – **[in]** - the value to be stored

__device__ void **__stwt**(*__nv_bfloat162* *const ptr, const *__nv_bfloat162* value)

Generates a `st.global.wt` store instruction.

Parameters

- **ptr** – **[out]** - memory location
- **value** – **[in]** - the value to be stored

__device__ void **__stwt**(*__nv_bfloat16* *const ptr, const *__nv_bfloat16* value)

Generates a `st.global.wt` store instruction.

Parameters

- **ptr** – **[out]** - memory location
- **value** – **[in]** - the value to be stored

__device__ *__nv_bfloat16* **__uint2bfloat16_rd**(const unsigned int i)

Convert an unsigned integer to a nv_bfloat16 in round-down mode.

Convert the unsigned integer value `i` to a nv_bfloat16 floating-point value in round-down mode.

Parameters

i – **[in]** - unsigned int. Is only being read.

Returns

nv_bfloat16

- `i` converted to nv_bfloat16.

__host__ __device__ *__nv_bfloat16* **__uint2bfloat16_rn**(const unsigned int i)

Convert an unsigned integer to a nv_bfloat16 in round-to-nearest-even mode.

Convert the unsigned integer value `i` to a nv_bfloat16 floating-point value in round-to-nearest-even mode.

Parameters

i – **[in]** - unsigned int. Is only being read.

Returns

nv_bfloat16

- `i` converted to nv_bfloat16.

__device__ *__nv_bfloat16* **__uint2bfloat16_ru**(const unsigned int i)

Convert an unsigned integer to a nv_bfloat16 in round-up mode.

Convert the unsigned integer value `i` to a nv_bfloat16 floating-point value in round-up mode.

Parameters

i – **[in]** - unsigned int. Is only being read.

Returns

nv_bfloat16

- `i` converted to nv_bfloat16.

__device__ *__nv_bfloat16* **__uint2bfloat16_rz**(const unsigned int i)

Convert an unsigned integer to a nv_bfloat16 in round-towards-zero mode.

Convert the unsigned integer value i to a nv_bfloat16 floating-point value in round-towards-zero mode.

Parameters

i – **[in]** - unsigned int. Is only being read.

Returns

nv_bfloat16

- i converted to nv_bfloat16.

__device__ *__nv_bfloat16* **__ull2bfloat16_rd**(const unsigned long long int i)

Convert an unsigned 64-bit integer to a nv_bfloat16 in round-down mode.

Convert the unsigned 64-bit integer value i to a nv_bfloat16 floating-point value in round-down mode.

Parameters

i – **[in]** - unsigned long long int. Is only being read.

Returns

nv_bfloat16

- i converted to nv_bfloat16.

__host__ __device__ *__nv_bfloat16* **__ull2bfloat16_rn**(const unsigned long long int i)

Convert an unsigned 64-bit integer to a nv_bfloat16 in round-to-nearest-even mode.

Convert the unsigned 64-bit integer value i to a nv_bfloat16 floating-point value in round-to-nearest-even mode.

Parameters

i – **[in]** - unsigned long long int. Is only being read.

Returns

nv_bfloat16

- i converted to nv_bfloat16.

__device__ *__nv_bfloat16* **__ull2bfloat16_ru**(const unsigned long long int i)

Convert an unsigned 64-bit integer to a nv_bfloat16 in round-up mode.

Convert the unsigned 64-bit integer value i to a nv_bfloat16 floating-point value in round-up mode.

Parameters

i – **[in]** - unsigned long long int. Is only being read.

Returns

nv_bfloat16

- i converted to nv_bfloat16.

__device__ *__nv_bfloat16* **__ull2bfloat16_rz**(const unsigned long long int i)

Convert an unsigned 64-bit integer to a nv_bfloat16 in round-towards-zero mode.

Convert the unsigned 64-bit integer value i to a nv_bfloat16 floating-point value in round-towards-zero mode.

Parameters

i – **[in]** - unsigned long long int. Is only being read.

Returns

nv_bfloat16

- `i` converted to nv_bfloat16.

__device__ *__nv_bfloat16* **`__ushort2bfloat16_rd`**(const unsigned short int i)

Convert an unsigned short integer to a nv_bfloat16 in round-down mode.

Convert the unsigned short integer value `i` to a nv_bfloat16 floating-point value in round-down mode.

Parameters

i – **[in]** - unsigned short int. Is only being read.

Returns

nv_bfloat16

- `i` converted to nv_bfloat16.

__host__ __device__ *__nv_bfloat16* **`__ushort2bfloat16_rn`**(const unsigned short int i)

Convert an unsigned short integer to a nv_bfloat16 in round-to-nearest-even mode.

Convert the unsigned short integer value `i` to a nv_bfloat16 floating-point value in round-to-nearest-even mode.

Parameters

i – **[in]** - unsigned short int. Is only being read.

Returns

nv_bfloat16

- `i` converted to nv_bfloat16.

__device__ *__nv_bfloat16* **`__ushort2bfloat16_ru`**(const unsigned short int i)

Convert an unsigned short integer to a nv_bfloat16 in round-up mode.

Convert the unsigned short integer value `i` to a nv_bfloat16 floating-point value in round-up mode.

Parameters

i – **[in]** - unsigned short int. Is only being read.

Returns

nv_bfloat16

- `i` converted to nv_bfloat16.

__device__ *__nv_bfloat16* **`__ushort2bfloat16_rz`**(const unsigned short int i)

Convert an unsigned short integer to a nv_bfloat16 in round-towards-zero mode.

Convert the unsigned short integer value `i` to a nv_bfloat16 floating-point value in round-towards-zero mode.

Parameters

i – **[in]** - unsigned short int. Is only being read.

Returns

nv_bfloat16

- `i` converted to nv_bfloat16.

__host__ __device__ *__nv_bfloat16* **__ushort_as_bfloat16**(const unsigned short int i)

Reinterprets bits in an unsigned short integer as a `nv_bfloat16`.

Reinterprets the bits in the unsigned short integer `i` as a nv_bfloat16 floating-point number.

Parameters

i – **[in]** - unsigned short int. Is only being read.

Returns

nv_bfloat16

- The reinterpreted value.

__host__ __device__ *__nv_bfloat162* **make_bfloat162**(const *__nv_bfloat16* x, const *__nv_bfloat16* y)

Vector function, combines two `nv_bfloat16` numbers into one `nv_bfloat162` number.

Combines two input `nv_bfloat16` number x and y into one `nv_bfloat162` number. Input x is stored in low 16 bits of the return value, input y is stored in high 16 bits of the return value.

Parameters

- **x** – **[in]** - nv_bfloat16. Is only being read.
- **y** – **[in]** - nv_bfloat16. Is only being read.

Returns

__nv_bfloat162

- The *__nv_bfloat162* vector with one half equal to x and the other to y.

5.6. Bfloat162 Arithmetic Functions

To use these functions, include the header file `cuda_bf16.h` in your program.

Functions

__host__ __device__ __nv_bfloat162 *__h2div*(const __nv_bfloat162 a, const __nv_bfloat162 b)

Performs `nv_bfloat162` vector division in round-to-nearest-even mode.

__host__ __device__ __nv_bfloat162 *__habs2*(const __nv_bfloat162 a)

Calculates the absolute value of both halves of the input `nv_bfloat162` number and returns the result.

__host__ __device__ __nv_bfloat162 *__hadd2*(const __nv_bfloat162 a, const __nv_bfloat162 b)

Performs `nv_bfloat162` vector addition in round-to-nearest-even mode.

__host__ __device__ __nv_bfloat162 *__hadd2_rn*(const __nv_bfloat162 a, const __nv_bfloat162 b)

Performs `nv_bfloat162` vector addition in round-to-nearest-even mode.

__host__ __device__ __nv_bfloat162 *__hadd2_sat*(const __nv_bfloat162 a, const __nv_bfloat162 b)

Performs `nv_bfloat162` vector addition in round-to-nearest-even mode, with saturation to [0.0, 1.0].

__device__ __nv_bfloat162 *__hcmadd*(const __nv_bfloat162 a, const __nv_bfloat162 b, const __nv_bfloat162 c)

Performs fast complex multiply-accumulate.

__device__ __nv_bfloat162 *__hfma2*(const __nv_bfloat162 a, const __nv_bfloat162 b, const __nv_bfloat162 c)
Performs nv_bfloat162 vector fused multiply-add in round-to-nearest-even mode.

__device__ __nv_bfloat162 *__hfma2_relu*(const __nv_bfloat162 a, const __nv_bfloat162 b, const __nv_bfloat162 c)
Performs nv_bfloat162 vector fused multiply-add in round-to-nearest-even mode with relu saturation.

__device__ __nv_bfloat162 *__hfma2_sat*(const __nv_bfloat162 a, const __nv_bfloat162 b, const __nv_bfloat162 c)
Performs nv_bfloat162 vector fused multiply-add in round-to-nearest-even mode, with saturation to [0.0, 1.0].

__host__ __device__ __nv_bfloat162 *__hmul2*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Performs nv_bfloat162 vector multiplication in round-to-nearest-even mode.

__host__ __device__ __nv_bfloat162 *__hmul2_rn*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Performs nv_bfloat162 vector multiplication in round-to-nearest-even mode.

__host__ __device__ __nv_bfloat162 *__hmul2_sat*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Performs nv_bfloat162 vector multiplication in round-to-nearest-even mode, with saturation to [0.0, 1.0].

__host__ __device__ __nv_bfloat162 *__hneg2*(const __nv_bfloat162 a)
Negates both halves of the input nv_bfloat162 number and returns the result.

__host__ __device__ __nv_bfloat162 *__hsub2*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Performs nv_bfloat162 vector subtraction in round-to-nearest-even mode.

__host__ __device__ __nv_bfloat162 *__hsub2_rn*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Performs nv_bfloat162 vector subtraction in round-to-nearest-even mode.

__host__ __device__ __nv_bfloat162 *__hsub2_sat*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Performs nv_bfloat162 vector subtraction in round-to-nearest-even mode, with saturation to [0.0, 1.0].

__device__ __nv_bfloat162 *atomicAdd*(__nv_bfloat162 *const address, const __nv_bfloat162 val)
Vector add `val` to the value stored at `address` in global or shared memory, and writes this value back to `address` .

__host__ __device__ __nv_bfloat162 *operator(const __nv_bfloat162 &lh, const __nv_bfloat162 &rh)**
Performs packed nv_bfloat16 multiplication operation.

__host__ __device__ __nv_bfloat162 & *operator*=*(__nv_bfloat162 &lh, const __nv_bfloat162 &rh)
Performs packed nv_bfloat16 compound assignment with multiplication operation.

__host__ __device__ __nv_bfloat162 *operator+*(const __nv_bfloat162 &lh, const __nv_bfloat162 &rh)
Performs packed nv_bfloat16 addition operation.

__host__ __device__ __nv_bfloat162 *operator+*(const __nv_bfloat162 &h)
Implements packed nv_bfloat16 unary plus operator, returns input value.

__host__ __device__ __nv_bfloat162 *operator++*(__nv_bfloat162 &h, const int ignored)
Performs packed nv_bfloat16 postfix increment operation.

__host__ __device__ __nv_bfloat162 & *operator++*(__nv_bfloat162 &h)
Performs packed nv_bfloat16 prefix increment operation.

__host__ __device__ __nv_bfloat162 & *operator+=*(__nv_bfloat162 &lh, const __nv_bfloat162 &rh)
Performs packed nv_bfloat16 compound assignment with addition operation.

__host__ __device__ __nv_bfloat162 *operator-*(const __nv_bfloat162 &h)
Implements packed `nv_bfloat16` unary minus operator.

__host__ __device__ __nv_bfloat162 *operator-*(const __nv_bfloat162 &lh, const __nv_bfloat162 &rh)
Performs packed `nv_bfloat16` subtraction operation.

__host__ __device__ __nv_bfloat162 *operator--*(__nv_bfloat162 &h, const int ignored)
Performs packed `nv_bfloat16` postfix decrement operation.

__host__ __device__ __nv_bfloat162 & *operator--*(__nv_bfloat162 &h)
Performs packed `nv_bfloat16` prefix decrement operation.

__host__ __device__ __nv_bfloat162 & *operator-=*(__nv_bfloat162 &lh, const __nv_bfloat162 &rh)
Performs packed `nv_bfloat16` compound assignment with subtraction operation.

__host__ __device__ __nv_bfloat162 *operator/*(const __nv_bfloat162 &lh, const __nv_bfloat162 &rh)
Performs packed `nv_bfloat16` division operation.

__host__ __device__ __nv_bfloat162 & *operator/=*(__nv_bfloat162 &lh, const __nv_bfloat162 &rh)
Performs packed `nv_bfloat16` compound assignment with division operation.

5.6.1. Functions

__host__ __device__ *__nv_bfloat162* **`__h2div`**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs `nv_bfloat162` vector division in round-to-nearest-even mode.

Divides `nv_bfloat162` input vector a by input vector b in round-to-nearest-even mode.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

- The elementwise division of a with b.

__host__ __device__ *__nv_bfloat162* **`__habs2`**(const *__nv_bfloat162* a)

Calculates the absolute value of both halves of the input `nv_bfloat162` number and returns the result.

Calculates the absolute value of both halves of the input `nv_bfloat162` number and returns the result.

Parameters

a – **[in]** - nv_bfloat162. Is only being read.

Returns

bfloat2

- Returns a with the absolute value of both halves.

__host__ __device__ *__nv_bfloat162* **`__hadd2`**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs `nv_bfloat162` vector addition in round-to-nearest-even mode.

Performs `nv_bfloat162` vector add of inputs a and b, in round-to-nearest-even mode.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns
nv_bfloat162

- The sum of vectors a and b.

__host__ __device__ *__nv_bfloat162* **__hadd2_rn**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs `nv_bfloat162` vector addition in round-to-nearest-even mode.

Performs `nv_bfloat162` vector add of inputs a and b, in round-to-nearest-even mode. Prevents floating-point contractions of mul+add into fma.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns
nv_bfloat162

- The sum of vectors a and b.

__host__ __device__ *__nv_bfloat162* **__hadd2_sat**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs `nv_bfloat162` vector addition in round-to-nearest-even mode, with saturation to [0.0, 1.0].

Performs `nv_bfloat162` vector add of inputs a and b, in round-to-nearest-even mode, and clamps the results to range [0.0, 1.0]. NaN results are flushed to +0.0.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns
nv_bfloat162

- The sum of a and b, with respect to saturation.

__device__ *__nv_bfloat162* **__hcmadd**(const *__nv_bfloat162* a, const *__nv_bfloat162* b, const *__nv_bfloat162* c)

Performs fast complex multiply-accumulate.

Interprets vector `nv_bfloat162` input pairs a, b, and c as complex numbers in `nv_bfloat16` precision and performs complex multiply-accumulate operation: a*b + c

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.
- **c** – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

- The result of complex multiply-accumulate operation on complex numbers a, b, and c

__device__ *__nv_bfloat162* **`__hfma2`**(const *__nv_bfloat162* a, const *__nv_bfloat162* b, const *__nv_bfloat162* c)

Performs `nv_bfloat162` vector fused multiply-add in round-to-nearest-even mode.

Performs `nv_bfloat162` vector multiply on inputs `a` and `b`, then performs a `nv_bfloat162` vector add of the result with `c`, rounding the result once in round-to-nearest-even mode.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.
- **c** – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

- The result of elementwise fused multiply-add operation on vectors a, b, and c.

__device__ *__nv_bfloat162* **`__hfma2_relu`**(const *__nv_bfloat162* a, const *__nv_bfloat162* b, const *__nv_bfloat162* c)

Performs `nv_bfloat162` vector fused multiply-add in round-to-nearest-even mode with relu saturation.

Performs `nv_bfloat162` vector multiply on inputs `a` and `b`, then performs a `nv_bfloat162` vector add of the result with `c`, rounding the result once in round-to-nearest-even mode. Then negative result is clamped to 0. NaN result is converted to canonical NaN.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.
- **c** – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

- The result of elementwise fused multiply-add operation on vectors a, b, and c with relu saturation.

__device__ *__nv_bfloat162* **`__hfma2_sat`**(const *__nv_bfloat162* a, const *__nv_bfloat162* b, const *__nv_bfloat162* c)

Performs `nv_bfloat162` vector fused multiply-add in round-to-nearest-even mode, with saturation to [0.0, 1.0].

Performs `nv_bfloat162` vector multiply on inputs `a` and `b`, then performs a `nv_bfloat162` vector add of the result with `c`, rounding the result once in round-to-nearest-even mode, and clamps the results to range [0.0, 1.0]. NaN results are flushed to +0.0.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.
- **c** – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

- The result of elementwise fused multiply-add operation on vectors a, b, and c, with respect to saturation.

__host__ __device__ *__nv_bfloat162* **__hmul2**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs `nv_bfloat162` vector multiplication in round-to-nearest-even mode.

Performs `nv_bfloat162` vector multiplication of inputs a and b, in round-to-nearest-even mode.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

- The result of elementwise multiplying the vectors a and b.

__host__ __device__ *__nv_bfloat162* **__hmul2_rn**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs `nv_bfloat162` vector multiplication in round-to-nearest-even mode.

Performs `nv_bfloat162` vector multiplication of inputs a and b, in round-to-nearest-even mode. Prevents floating-point contractions of mul+add or sub into fma.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

- The result of elementwise multiplying the vectors a and b.

__host__ __device__ *__nv_bfloat162* **__hmul2_sat**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs `nv_bfloat162` vector multiplication in round-to-nearest-even mode, with saturation to [0.0, 1.0].

Performs `nv_bfloat162` vector multiplication of inputs a and b, in round-to-nearest-even mode, and clamps the results to range [0.0, 1.0]. NaN results are flushed to +0.0.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

- The result of elementwise multiplication of vectors a and b, with respect to saturation.

__host__ __device__ *__nv_bfloat162* **__hneg2**(const *__nv_bfloat162* a)

Negates both halves of the input `nv_bfloat162` number and returns the result.

Negates both halves of the input `nv_bfloat162` number a and returns the result.

Parameters

a – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

- Returns a with both halves negated.

__host__ __device__ *__nv_bfloat162* **__hsub2**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs `nv_bfloat162` vector subtraction in round-to-nearest-even mode.

Subtracts `nv_bfloat162` input vector b from input vector a in round-to-nearest-even mode.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

- The subtraction of vector b from a.

__host__ __device__ *__nv_bfloat162* **__hsub2_rn**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs `nv_bfloat162` vector subtraction in round-to-nearest-even mode.

Subtracts `nv_bfloat162` input vector b from input vector a in round-to-nearest-even mode. Prevents floating-point contractions of mul+sub into fma.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

- The subtraction of vector b from a.

__host__ __device__ *__nv_bfloat162* **__hsub2_sat**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs `nv_bfloat162` vector subtraction in round-to-nearest-even mode, with saturation to [0.0, 1.0].

Subtracts `nv_bfloat162` input vector b from input vector a in round-to-nearest-even mode, and clamps the results to range [0.0, 1.0]. NaN results are flushed to +0.0.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

- The subtraction of vector b from a, with respect to saturation.

__device__ *__nv_bfloat162* **atomicAdd**(*__nv_bfloat162* *const address, const *__nv_bfloat162* val)

Vector add `val` to the value stored at `address` in global or shared memory, and writes this value back to `address`.

The atomicity of the add operation is guaranteed separately for each of the two nv_bfloat16 elements; the entire *__nv_bfloat162* is not guaranteed to be atomic as a single 32-bit access.

The location of `address` must be in global or shared memory. This operation has undefined behavior otherwise. This operation is natively supported by devices of compute capability 9.x and higher, older devices use emulation path.

Note: For more details about this function, see the Atomic Functions section in the CUDA C++ Programming Guide.

Parameters

- **address** – **[in]** - __nv_bfloat162*. An address in global or shared memory.
- **val** – **[in]** - *__nv_bfloat162*. The value to be added.

Returns

__nv_bfloat162

- The old value read from `address`.

__host__ __device__ *__nv_bfloat162* **operator***(const *__nv_bfloat162* &lh, const *__nv_bfloat162* &rh)

Performs packed `nv_bfloat16` multiplication operation.

See also *__hmul2(__nv_bfloat162, __nv_bfloat162)*

__host__ __device__ *__nv_bfloat162* &**operator*=**(*__nv_bfloat162* &lh, const *__nv_bfloat162* &rh)

Performs packed `nv_bfloat16` compound assignment with multiplication operation.

__host__ __device__ *__nv_bfloat162* **operator+**(const *__nv_bfloat162* &lh, const *__nv_bfloat162* &rh)

Performs packed `nv_bfloat16` addition operation.

See also *__hadd2(__nv_bfloat162, __nv_bfloat162)*

__host__ __device__ *__nv_bfloat162* **operator+**(const *__nv_bfloat162* &h)

Implements packed `nv_bfloat16` unary plus operator, returns input value.

__host__ __device__ *__nv_bfloat162* **operator++**(*__nv_bfloat162* &h, const int ignored)

Performs packed `nv_bfloat16` postfix increment operation.

__host__ __device__ *__nv_bfloat162* &**operator++**(*__nv_bfloat162* &h)

Performs packed `nv_bfloat16` prefix increment operation.

__host__ __device__ *__nv_bfloat162* &**operator+=**(*__nv_bfloat162* &lh, const *__nv_bfloat162* &rh)

Performs packed `nv_bfloat16` compound assignment with addition operation.

__host__ __device__ *__nv_bfloat162* **operator-**(const *__nv_bfloat162* &h)

Implements packed `nv_bfloat16` unary minus operator.

See also *__hneg2(__nv_bfloat162)*

__host__ __device__ *__nv_bfloat162* **operator-**(const *__nv_bfloat162* &lh, const *__nv_bfloat162* &rh)

Performs packed `nv_bfloat16` subtraction operation.

See also *__hsub2(__nv_bfloat162, __nv_bfloat162)*

__host__ __device__ *__nv_bfloat162* **operator--**(*__nv_bfloat162* &h, const int ignored)

Performs packed `nv_bfloat16` postfix decrement operation.

__host__ __device__ *__nv_bfloat162* &**operator--**(*__nv_bfloat162* &h)

Performs packed `nv_bfloat16` prefix decrement operation.

__host__ __device__ *__nv_bfloat162* &**operator-=**(*__nv_bfloat162* &lh, const *__nv_bfloat162* &rh)

Performs packed `nv_bfloat16` compound assignment with subtraction operation.

__host__ __device__ *__nv_bfloat162* **operator/**(const *__nv_bfloat162* &lh, const *__nv_bfloat162* &rh)

Performs packed `nv_bfloat16` division operation.

See also *__h2div(__nv_bfloat162, __nv_bfloat162)*

__host__ __device__ *__nv_bfloat162* &**operator/=**(*__nv_bfloat162* &lh, const *__nv_bfloat162* &rh)

Performs packed `nv_bfloat16` compound assignment with division operation.

5.7. Bfloat162 Comparison Functions

To use these functions, include the header file `cuda_bf16.h` in your program.

Functions

__host__ __device__ bool *__hbeq2*(const __nv_bfloat162 a, const __nv_bfloat162 b)

Performs `nv_bfloat162` vector if-equal comparison and returns boolean true if both `nv_bfloat16` results are true, boolean false otherwise.

__host__ __device__ bool *__hbequ2*(const __nv_bfloat162 a, const __nv_bfloat162 b)

Performs `nv_bfloat162` vector unordered if-equal comparison and returns boolean true if both `nv_bfloat16` results are true, boolean false otherwise.

__host__ __device__ bool *__hbge2*(const __nv_bfloat162 a, const __nv_bfloat162 b)

Performs `nv_bfloat162` vector greater-equal comparison and returns boolean true if both `nv_bfloat16` results are true, boolean false otherwise.

__host__ __device__ bool *__hbgeu2*(const __nv_bfloat162 a, const __nv_bfloat162 b)

Performs `nv_bfloat162` vector unordered greater-equal comparison and returns boolean true if both `nv_bfloat16` results are true, boolean false otherwise.

__host__ __device__ bool *__hbgt2*(const __nv_bfloat162 a, const __nv_bfloat162 b)

Performs `nv_bfloat162` vector greater-than comparison and returns boolean true if both `nv_bfloat16` results are true, boolean false otherwise.

__host__ __device__ bool *__hbgtu2*(const __nv_bfloat162 a, const __nv_bfloat162 b)

Performs `nv_bfloat162` vector unordered greater-than comparison and returns boolean true if both `nv_bfloat16` results are true, boolean false otherwise.

__host__ __device__ bool *__hble2*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Performs nv_bfloat162 vector less-equal comparison and returns boolean true if both nv_bfloat16 results are true, boolean false otherwise.

__host__ __device__ bool *__hbleu2*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Performs nv_bfloat162 vector unordered less-equal comparison and returns boolean true if both nv_bfloat16 results are true, boolean false otherwise.

__host__ __device__ bool *__hblt2*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Performs nv_bfloat162 vector less-than comparison and returns boolean true if both nv_bfloat16 results are true, boolean false otherwise.

__host__ __device__ bool *__hbltu2*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Performs nv_bfloat162 vector unordered less-than comparison and returns boolean true if both nv_bfloat16 results are true, boolean false otherwise.

__host__ __device__ bool *__hbne2*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Performs nv_bfloat162 vector not-equal comparison and returns boolean true if both nv_bfloat16 results are true, boolean false otherwise.

__host__ __device__ bool *__hbneu2*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Performs nv_bfloat162 vector unordered not-equal comparison and returns boolean true if both nv_bfloat16 results are true, boolean false otherwise.

__host__ __device__ __nv_bfloat162 *__heq2*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Performs nv_bfloat162 vector if-equal comparison.

__host__ __device__ unsigned int *__heq2_mask*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Performs nv_bfloat162 vector if-equal comparison.

__host__ __device__ __nv_bfloat162 *__hequ2*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Performs nv_bfloat162 vector unordered if-equal comparison.

__host__ __device__ unsigned int *__hequ2_mask*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Performs nv_bfloat162 vector unordered if-equal comparison.

__host__ __device__ __nv_bfloat162 *__hge2*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Performs nv_bfloat162 vector greater-equal comparison.

__host__ __device__ unsigned int *__hge2_mask*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Performs nv_bfloat162 vector greater-equal comparison.

__host__ __device__ __nv_bfloat162 *__hgeu2*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Performs nv_bfloat162 vector unordered greater-equal comparison.

__host__ __device__ unsigned int *__hgeu2_mask*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Performs nv_bfloat162 vector unordered greater-equal comparison.

__host__ __device__ __nv_bfloat162 *__hgt2*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Performs nv_bfloat162 vector greater-than comparison.

__host__ __device__ unsigned int *__hgt2_mask*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Performs nv_bfloat162 vector greater-than comparison.

__host__ __device__ __nv_bfloat162 *__hgtu2*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Performs nv_bfloat162 vector unordered greater-than comparison.

__host__ __device__ unsigned int *__hgtu2_mask*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Performs nv_bfloat162 vector unordered greater-than comparison.

__host__ __device__ __nv_bfloat162 *__hisnan2*(const __nv_bfloat162 a)
Determine whether nv_bfloat162 argument is a NaN.

__host__ __device__ __nv_bfloat162 *__hle2*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Performs nv_bfloat162 vector less-equal comparison.

__host__ __device__ unsigned int *__hle2_mask*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Performs nv_bfloat162 vector less-equal comparison.

__host__ __device__ __nv_bfloat162 *__hleu2*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Performs nv_bfloat162 vector unordered less-equal comparison.

__host__ __device__ unsigned int *__hleu2_mask*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Performs nv_bfloat162 vector unordered less-equal comparison.

__host__ __device__ __nv_bfloat162 *__hlt2*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Performs nv_bfloat162 vector less-than comparison.

__host__ __device__ unsigned int *__hlt2_mask*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Performs nv_bfloat162 vector less-than comparison.

__host__ __device__ __nv_bfloat162 *__hltu2*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Performs nv_bfloat162 vector unordered less-than comparison.

__host__ __device__ unsigned int *__hltu2_mask*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Performs nv_bfloat162 vector unordered less-than comparison.

__host__ __device__ __nv_bfloat162 *__hmax2*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Calculates nv_bfloat162 vector maximum of two inputs.

__host__ __device__ __nv_bfloat162 *__hmax2_nan*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Calculates nv_bfloat162 vector maximum of two inputs, NaNs pass through.

__host__ __device__ __nv_bfloat162 *__hmin2*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Calculates nv_bfloat162 vector minimum of two inputs.

__host__ __device__ __nv_bfloat162 *__hmin2_nan*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Calculates nv_bfloat162 vector minimum of two inputs, NaNs pass through.

__host__ __device__ __nv_bfloat162 *__hne2*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Performs nv_bfloat162 vector not-equal comparison.

__host__ __device__ unsigned int *__hne2_mask*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Performs nv_bfloat162 vector not-equal comparison.

__host__ __device__ __nv_bfloat162 *__hneu2*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Performs nv_bfloat162 vector unordered not-equal comparison.

__host__ __device__ unsigned int *__hneu2_mask*(const __nv_bfloat162 a, const __nv_bfloat162 b)
Performs nv_bfloat162 vector unordered not-equal comparison.

__host__ __device__ bool *operator!=*(const __nv_bfloat162 &lh, const __nv_bfloat162 &rh)
Performs packed nv_bfloat16 unordered compare not-equal operation.

__host__ __device__ bool *operator<*(const __nv_bfloat162 &lh, const __nv_bfloat162 &rh)
Performs packed nv_bfloat16 ordered less-than compare operation.

__host__ __device__ bool *operator<=*(const __nv_bfloat162 &lh, const __nv_bfloat162 &rh)
Performs packed nv_bfloat16 ordered less-or-equal compare operation.

__host__ __device__ bool *operator==*(const __nv_bfloat162 &lh, const __nv_bfloat162 &rh)
Performs packed nv_bfloat16 ordered compare equal operation.

__host__ __device__ bool *operator>*(const __nv_bfloat162 &lh, const __nv_bfloat162 &rh)
Performs packed nv_bfloat16 ordered greater-than compare operation.

__host__ __device__ bool *operator>=*(const __nv_bfloat162 &lh, const __nv_bfloat162 &rh)
Performs packed `nv_bfloat16` ordered greater-or-equal compare operation.

5.7.1. Functions

__host__ __device__ bool **__hbeq2**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs `nv_bfloat162` vector if-equal comparison and returns boolean true if both `nv_bfloat16` results are true, boolean false otherwise.

Performs `nv_bfloat162` vector if-equal comparison of inputs a and b. The bool result is set to true only if both `nv_bfloat16` if-equal comparisons evaluate to true, or false otherwise. NaN inputs generate false results.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

bool

- true if both `nv_bfloat16` results of if-equal comparison of vectors a and b are true;
- false otherwise.

__host__ __device__ bool **__hbequ2**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs `nv_bfloat162` vector unordered if-equal comparison and returns boolean true if both `nv_bfloat16` results are true, boolean false otherwise.

Performs `nv_bfloat162` vector if-equal comparison of inputs a and b. The bool result is set to true only if both `nv_bfloat16` if-equal comparisons evaluate to true, or false otherwise. NaN inputs generate true results.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

bool

- true if both `nv_bfloat16` results of unordered if-equal comparison of vectors a and b are true;
- false otherwise.

__host__ __device__ bool **__hbge2**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs `nv_bfloat162` vector greater-equal comparison and returns boolean true if both `nv_bfloat16` results are true, boolean false otherwise.

Performs `nv_bfloat162` vector greater-equal comparison of inputs a and b. The bool result is set to true only if both `nv_bfloat16` greater-equal comparisons evaluate to true, or false otherwise. NaN inputs generate false results.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns
bool

- true if both `nv_bfloat16` results of greater-equal comparison of vectors a and b are true;
- false otherwise.

__host__ __device__ bool **__hbgeu2** (const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs `nv_bfloat162` vector unordered greater-equal comparison and returns boolean true if both `nv_bfloat16` results are true, boolean false otherwise.

Performs `nv_bfloat162` vector greater-equal comparison of inputs a and b. The bool result is set to true only if both `nv_bfloat16` greater-equal comparisons evaluate to true, or false otherwise. NaN inputs generate true results.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns
bool

- true if both `nv_bfloat16` results of unordered greater-equal comparison of vectors a and b are true;
- false otherwise.

__host__ __device__ bool **__hbgt2** (const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs `nv_bfloat162` vector greater-than comparison and returns boolean true if both `nv_bfloat16` results are true, boolean false otherwise.

Performs `nv_bfloat162` vector greater-than comparison of inputs a and b. The bool result is set to true only if both `nv_bfloat16` greater-than comparisons evaluate to true, or false otherwise. NaN inputs generate false results.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns
bool

- true if both `nv_bfloat16` results of greater-than comparison of vectors a and b are true;
- false otherwise.

__host__ __device__ bool **__hbgtu2** (const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs `nv_bfloat162` vector unordered greater-than comparison and returns boolean true if both `nv_bfloat16` results are true, boolean false otherwise.

Performs `nv_bfloat162` vector greater-than comparison of inputs a and b. The bool result is set to true only if both `nv_bfloat16` greater-than comparisons evaluate to true, or false otherwise. NaN inputs generate true results.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

bool

- true if both `nv_bfloat16` results of unordered greater-than comparison of vectors a and b are true;
- false otherwise.

__host__ __device__ bool **`__hble2`**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs `nv_bfloat162` vector less-equal comparison and returns boolean true if both `nv_bfloat16` results are true, boolean false otherwise.

Performs `nv_bfloat162` vector less-equal comparison of inputs a and b. The bool result is set to true only if both `nv_bfloat16` less-equal comparisons evaluate to true, or false otherwise. NaN inputs generate false results.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

bool

- true if both `nv_bfloat16` results of less-equal comparison of vectors a and b are true;
- false otherwise.

__host__ __device__ bool **`__hbleu2`**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs `nv_bfloat162` vector unordered less-equal comparison and returns boolean true if both `nv_bfloat16` results are true, boolean false otherwise.

Performs `nv_bfloat162` vector less-equal comparison of inputs a and b. The bool result is set to true only if both `nv_bfloat16` less-equal comparisons evaluate to true, or false otherwise. NaN inputs generate true results.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

bool

- true if both `nv_bfloat16` results of unordered less-equal comparison of vectors a and b are true;
- false otherwise.

__host__ __device__ bool **__hblt2**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs nv_bfloat162 vector less-than comparison and returns boolean true if both nv_bfloat16 results are true, boolean false otherwise.

Performs nv_bfloat162 vector less-than comparison of inputs a and b. The bool result is set to true only if both nv_bfloat16 less-than comparisons evaluate to true, or false otherwise. NaN inputs generate false results.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

bool

- true if both nv_bfloat16 results of less-than comparison of vectors a and b are true;
- false otherwise.

__host__ __device__ bool **__hbltu2**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs nv_bfloat162 vector unordered less-than comparison and returns boolean true if both nv_bfloat16 results are true, boolean false otherwise.

Performs nv_bfloat162 vector less-than comparison of inputs a and b. The bool result is set to true only if both nv_bfloat16 less-than comparisons evaluate to true, or false otherwise. NaN inputs generate true results.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

bool

- true if both nv_bfloat16 results of unordered less-than comparison of vectors a and b are true;
- false otherwise.

__host__ __device__ bool **__hbne2**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs nv_bfloat162 vector not-equal comparison and returns boolean true if both nv_bfloat16 results are true, boolean false otherwise.

Performs nv_bfloat162 vector not-equal comparison of inputs a and b. The bool result is set to true only if both nv_bfloat16 not-equal comparisons evaluate to true, or false otherwise. NaN inputs generate false results.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

bool

- true if both nv_bfloat16 results of not-equal comparison of vectors a and b are true,
- false otherwise.

__host__ __device__ bool **__hbneu2**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs nv_bfloat162 vector unordered not-equal comparison and returns boolean true if both nv_bfloat16 results are true, boolean false otherwise.

Performs nv_bfloat162 vector not-equal comparison of inputs a and b. The bool result is set to true only if both nv_bfloat16 not-equal comparisons evaluate to true, or false otherwise. NaN inputs generate true results.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

bool

- true if both nv_bfloat16 results of unordered not-equal comparison of vectors a and b are true;
- false otherwise.

__host__ __device__ *__nv_bfloat162* **__heq2**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs nv_bfloat162 vector if-equal comparison.

Performs nv_bfloat162 vector if-equal comparison of inputs a and b. The corresponding nv_bfloat16 results are set to 1.0 for true, or 0.0 for false. NaN inputs generate false results.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

- The vector result of if-equal comparison of vectors a and b.

__host__ __device__ unsigned int **__heq2_mask**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs nv_bfloat162 vector if-equal comparison.

Performs nv_bfloat162 vector if-equal comparison of inputs a and b. The corresponding unsigned bits are set to 0xFFFF for true, or 0x0 for false. NaN inputs generate false results.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

unsigned int

- The vector mask result of if-equal comparison of vectors a and b.

__host__ __device__ *__nv_bfloat162* **__hequ2**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs `nv_bfloat162` vector unordered if-equal comparison.

Performs `nv_bfloat162` vector if-equal comparison of inputs a and b. The corresponding `nv_bfloat16` results are set to 1.0 for true, or 0.0 for false. NaN inputs generate true results.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

- The vector result of unordered if-equal comparison of vectors a and b.

__host__ __device__ unsigned int **__hequ2_mask**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs `nv_bfloat162` vector unordered if-equal comparison.

Performs `nv_bfloat162` vector if-equal comparison of inputs a and b. The corresponding `unsigned` bits are set to 0xFFFF for true, or 0x0 for false. NaN inputs generate true results.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

unsigned int

- The vector mask result of unordered if-equal comparison of vectors a and b.

__host__ __device__ *__nv_bfloat162* **__hge2**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs `nv_bfloat162` vector greater-equal comparison.

Performs `nv_bfloat162` vector greater-equal comparison of inputs a and b. The corresponding `nv_bfloat16` results are set to 1.0 for true, or 0.0 for false. NaN inputs generate false results.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

- The vector result of greater-equal comparison of vectors a and b.

__host__ __device__ unsigned int **__hge2_mask**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs `nv_bfloat162` vector greater-equal comparison.

Performs `nv_bfloat162` vector greater-equal comparison of inputs a and b. The corresponding `unsigned` bits are set to 0xFFFF for true, or 0x0 for false. NaN inputs generate false results.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.

- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

unsigned int

- The vector mask result of greater-equal comparison of vectors a and b.

__host__ __device__ *__nv_bfloat162* **__hgeu2**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs `nv_bfloat162` vector unordered greater-equal comparison.

Performs `nv_bfloat162` vector greater-equal comparison of inputs a and b. The corresponding `nv_bfloat16` results are set to 1.0 for true, or 0.0 for false. NaN inputs generate true results.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

- The `nv_bfloat162` vector result of unordered greater-equal comparison of vectors a and b.

__host__ __device__ unsigned int **__hgeu2_mask**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs `nv_bfloat162` vector unordered greater-equal comparison.

Performs `nv_bfloat162` vector greater-equal comparison of inputs a and b. The corresponding `unsigned` bits are set to 0xFFFF for true, or 0x0 for false. NaN inputs generate true results.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

unsigned int

- The vector mask result of unordered greater-equal comparison of vectors a and b.

__host__ __device__ *__nv_bfloat162* **__hgt2**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs `nv_bfloat162` vector greater-than comparison.

Performs `nv_bfloat162` vector greater-than comparison of inputs a and b. The corresponding `nv_bfloat16` results are set to 1.0 for true, or 0.0 for false. NaN inputs generate false results.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

- The vector result of greater-than comparison of vectors a and b.

__host__ __device__ unsigned int **__hgt2_mask**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs `nv_bfloat162` vector greater-than comparison.

Performs `nv_bfloat162` vector greater-than comparison of inputs a and b. The corresponding `unsigned` bits are set to 0xFFFF for true, or 0x0 for false. NaN inputs generate false results.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

unsigned int

- The vector mask result of greater-than comparison of vectors a and b.

__host__ __device__ *__nv_bfloat162* **__hgtu2**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs `nv_bfloat162` vector unordered greater-than comparison.

Performs `nv_bfloat162` vector greater-than comparison of inputs a and b. The corresponding `nv_bfloat16` results are set to 1.0 for true, or 0.0 for false. NaN inputs generate true results.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

- The `nv_bfloat162` vector result of unordered greater-than comparison of vectors a and b.

__host__ __device__ unsigned int **__hgtu2_mask**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs `nv_bfloat162` vector unordered greater-than comparison.

Performs `nv_bfloat162` vector greater-than comparison of inputs a and b. The corresponding `unsigned` bits are set to 0xFFFF for true, or 0x0 for false. NaN inputs generate true results.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

unsigned int

- The vector mask result of unordered greater-than comparison of vectors a and b.

__host__ __device__ *__nv_bfloat162* **__hisnan2**(const *__nv_bfloat162* a)

Determine whether `nv_bfloat162` argument is a NaN.

Determine whether each nv_bfloat16 of input `nv_bfloat162` number a is a NaN.

Parameters

a – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

- The nv_bfloat162 with the corresponding `nv_bfloat16` results set to 1.0 for NaN, 0.0 otherwise.

__host__ __device__ *__nv_bfloat162* **__hle2**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs `nv_bfloat162` vector less-equal comparison.

Performs `nv_bfloat162` vector less-equal comparison of inputs a and b. The corresponding `nv_bfloat16` results are set to 1.0 for true, or 0.0 for false. NaN inputs generate false results.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

- The `nv_bfloat162` result of less-equal comparison of vectors a and b.

__host__ __device__ unsigned int **__hle2_mask**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs `nv_bfloat162` vector less-equal comparison.

Performs `nv_bfloat162` vector less-equal comparison of inputs a and b. The corresponding `unsigned` bits are set to 0xFFFF for true, or 0x0 for false. NaN inputs generate false results.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

unsigned int

- The vector mask result of less-equal comparison of vectors a and b.

__host__ __device__ *__nv_bfloat162* **__hleu2**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs `nv_bfloat162` vector unordered less-equal comparison.

Performs `nv_bfloat162` vector less-equal comparison of inputs a and b. The corresponding `nv_bfloat16` results are set to 1.0 for true, or 0.0 for false. NaN inputs generate true results.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

- The vector result of unordered less-equal comparison of vectors a and b.

__host__ __device__ unsigned int **__hleu2_mask**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs `nv_bfloat162` vector unordered less-equal comparison.

Performs `nv_bfloat162` vector less-equal comparison of inputs a and b. The corresponding `unsigned` bits are set to 0xFFFF for true, or 0x0 for false. NaN inputs generate true results.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns
unsigned int

- The vector mask result of unordered less-equal comparison of vectors a and b.

__host__ __device__ *__nv_bfloat162* **__hlt2**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs `nv_bfloat162` vector less-than comparison.

Performs `nv_bfloat162` vector less-than comparison of inputs a and b. The corresponding `nv_bfloat16` results are set to 1.0 for true, or 0.0 for false. NaN inputs generate false results.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns
nv_bfloat162

- The nv_bfloat162 vector result of less-than comparison of vectors a and b.

__host__ __device__ unsigned int **__hlt2_mask**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs `nv_bfloat162` vector less-than comparison.

Performs `nv_bfloat162` vector less-than comparison of inputs a and b. The corresponding `unsigned` bits are set to 0xFFFF for true, or 0x0 for false. NaN inputs generate false results.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns
unsigned int

- The vector mask result of less-than comparison of vectors a and b.

__host__ __device__ *__nv_bfloat162* **__hltu2**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs `nv_bfloat162` vector unordered less-than comparison.

Performs `nv_bfloat162` vector less-than comparison of inputs a and b. The corresponding `nv_bfloat16` results are set to 1.0 for true, or 0.0 for false. NaN inputs generate true results.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns
nv_bfloat162

- The vector result of unordered less-than comparison of vectors a and b.

__host__ __device__ unsigned int **__hltu2_mask**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs `nv_bfloat162` vector unordered less-than comparison.

Performs `nv_bfloat162` vector less-than comparison of inputs `a` and `b`. The corresponding `unsigned` bits are set to 0xFFFF for true, or 0x0 for false. NaN inputs generate true results.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

unsigned int

- The vector mask result of unordered less-than comparison of vectors `a` and `b`.

__host__ __device__ *__nv_bfloat162* **__hmax2**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Calculates `nv_bfloat162` vector maximum of two inputs.

Calculates `nv_bfloat162` vector max(a, b). Elementwise `nv_bfloat16` operation is defined as (a > b) ? a : b.

- If either of inputs is NaN, the other input is returned.
- If both inputs are NaNs, then canonical NaN is returned.
- If values of both inputs are 0.0, then +0.0 > -0.0
- The result of elementwise maximum of vectors a and b

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

__host__ __device__ *__nv_bfloat162* **__hmax2_nan**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Calculates `nv_bfloat162` vector maximum of two inputs, NaNs pass through.

Calculates `nv_bfloat162` vector max(a, b). Elementwise `nv_bfloat16` operation is defined as (a > b) ? a : b.

- If either of inputs is NaN, then canonical NaN is returned.
- If values of both inputs are 0.0, then +0.0 > -0.0
- The result of elementwise maximum of vectors a and b, with NaNs pass through

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

__host__ __device__ *__nv_bfloat162* **__hmin2**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Calculates `nv_bfloat162` vector minimum of two inputs.

Calculates `nv_bfloat162` vector min(a, b). Elementwise `nv_bfloat16` operation is defined as (a < b) ? a : b.

- If either of inputs is NaN, the other input is returned.
- If both inputs are NaNs, then canonical NaN is returned.
- If values of both inputs are 0.0, then +0.0 > -0.0
- The result of elementwise minimum of vectors a and b

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

__host__ __device__ *__nv_bfloat162* **__hmin2_nan**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Calculates `nv_bfloat162` vector minimum of two inputs, NaNs pass through.

Calculates `nv_bfloat162` vector min(a, b). Elementwise `nv_bfloat16` operation is defined as (a < b) ? a : b.

- If either of inputs is NaN, then canonical NaN is returned.
- If values of both inputs are 0.0, then +0.0 > -0.0
- The result of elementwise minimum of vectors a and b, with NaNs pass through

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

__host__ __device__ *__nv_bfloat162* **__hne2**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs `nv_bfloat162` vector not-equal comparison.

Performs `nv_bfloat162` vector not-equal comparison of inputs a and b. The corresponding `nv_bfloat16` results are set to 1.0 for true, or 0.0 for false. NaN inputs generate false results.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

- The vector result of not-equal comparison of vectors a and b.

__host__ __device__ unsigned int **__hne2_mask**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs `nv_bfloat162` vector not-equal comparison.

Performs `nv_bfloat162` vector not-equal comparison of inputs `a` and `b`. The corresponding `unsigned` bits are set to 0xFFFF for true, or 0x0 for false. NaN inputs generate false results.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

unsigned int

- The vector mask result of not-equal comparison of vectors `a` and `b`.

__host__ __device__ *__nv_bfloat162* **__hneu2**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs `nv_bfloat162` vector unordered not-equal comparison.

Performs `nv_bfloat162` vector not-equal comparison of inputs `a` and `b`. The corresponding `nv_bfloat16` results are set to 1.0 for true, or 0.0 for false. NaN inputs generate true results.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

- The vector result of unordered not-equal comparison of vectors `a` and `b`.

__host__ __device__ unsigned int **__hneu2_mask**(const *__nv_bfloat162* a, const *__nv_bfloat162* b)

Performs `nv_bfloat162` vector unordered not-equal comparison.

Performs `nv_bfloat162` vector not-equal comparison of inputs `a` and `b`. The corresponding `unsigned` bits are set to 0xFFFF for true, or 0x0 for false. NaN inputs generate true results.

Parameters

- **a** – **[in]** - nv_bfloat162. Is only being read.
- **b** – **[in]** - nv_bfloat162. Is only being read.

Returns

unsigned int

- The vector mask result of unordered not-equal comparison of vectors `a` and `b`.

__host__ __device__ bool **operator!=**(const *__nv_bfloat162* &lh, const *__nv_bfloat162* &rh)

Performs packed `nv_bfloat16` unordered compare not-equal operation.

See also *__hbneu2(__nv_bfloat162, __nv_bfloat162)*

__host__ __device__ bool **operator<**(const *__nv_bfloat162* &lh, const *__nv_bfloat162* &rh)

Performs packed `nv_bfloat16` ordered less-than compare operation.

See also *__hblt2(__nv_bfloat162, __nv_bfloat162)* __host__ __device__ bool **operator<=**(const *__nv_bfloat162* &lh, const *__nv_bfloat162* &rh)

Performs packed nv_bfloat16 ordered less-or-equal compare operation.

See also *__hble2(__nv_bfloat162, __nv_bfloat162)*

__host__ __device__ bool **operator==**(const *__nv_bfloat162* &lh, const *__nv_bfloat162* &rh)

Performs packed nv_bfloat16 ordered compare equal operation.

See also *__hbeq2(__nv_bfloat162, __nv_bfloat162)*

__host__ __device__ bool **operator>**(const *__nv_bfloat162* &lh, const *__nv_bfloat162* &rh)

Performs packed nv_bfloat16 ordered greater-than compare operation.

See also *__hbgt2(__nv_bfloat162, __nv_bfloat162)*

__host__ __device__ bool **operator>=**(const *__nv_bfloat162* &lh, const *__nv_bfloat162* &rh)

Performs packed nv_bfloat16 ordered greater-or-equal compare operation.

See also *__hbge2(__nv_bfloat162, __nv_bfloat162)*

5.8. Bfloat162 Math Functions

To use these functions, include the header file cuda_bf16.h in your program.

Functions

__device__ __nv_bfloat162 *h2ceil*(const __nv_bfloat162 h)

Calculate nv_bfloat162 vector ceiling of the input argument.

__device__ __nv_bfloat162 *h2cos*(const __nv_bfloat162 a)

Calculates nv_bfloat162 vector cosine in round-to-nearest-even mode.

__device__ __nv_bfloat162 *h2exp*(const __nv_bfloat162 a)

Calculates nv_bfloat162 vector exponential function in round-to-nearest-even mode.

__device__ __nv_bfloat162 *h2exp10*(const __nv_bfloat162 a)

Calculates nv_bfloat162 vector decimal exponential function in round-to-nearest-even mode.

__device__ __nv_bfloat162 *h2exp2*(const __nv_bfloat162 a)

Calculates nv_bfloat162 vector binary exponential function in round-to-nearest-even mode.

__device__ __nv_bfloat162 *h2floor*(const __nv_bfloat162 h)

Calculate the largest integer less than or equal to h .

__device__ __nv_bfloat162 *h2log*(const __nv_bfloat162 a)

Calculates nv_bfloat162 vector natural logarithm in round-to-nearest-even mode.

__device__ __nv_bfloat162 *h2log10*(const __nv_bfloat162 a)

Calculates nv_bfloat162 vector decimal logarithm in round-to-nearest-even mode.

__device__ __nv_bfloat162 *h2log2*(const __nv_bfloat162 a)

Calculates nv_bfloat162 vector binary logarithm in round-to-nearest-even mode.

__device__ __nv_bfloat162 *h2rcp*(const __nv_bfloat162 a)

Calculates nv_bfloat162 vector reciprocal in round-to-nearest-even mode.

__device__ __nv_bfloat162 *h2rint*(const __nv_bfloat162 h)

Round input to nearest integer value in nv_bfloat16 floating-point number.

__device__ __nv_bfloat162 *h2rsqrt*(const __nv_bfloat162 a)
Calculates nv_bfloat162 vector reciprocal square root in round-to-nearest-even mode.

__device__ __nv_bfloat162 *h2sin*(const __nv_bfloat162 a)
Calculates nv_bfloat162 vector sine in round-to-nearest-even mode.

__device__ __nv_bfloat162 *h2sqrt*(const __nv_bfloat162 a)
Calculates nv_bfloat162 vector square root in round-to-nearest-even mode.

__device__ __nv_bfloat162 *h2tanh*(const __nv_bfloat162 a)
Calculates nv_bfloat162 vector hyperbolic tangent function in round-to-nearest-even mode.

__device__ __nv_bfloat162 *h2tanh_approx*(const __nv_bfloat162 a)
Calculates nv_bfloat162 vector approximate hyperbolic tangent function.

__device__ __nv_bfloat162 *h2trunc*(const __nv_bfloat162 h)
Truncate nv_bfloat162 vector input argument to the integral part.

5.8.1. Functions

__device__ *__nv_bfloat162* **h2ceil**(const *__nv_bfloat162* h)

Calculate nv_bfloat162 vector ceiling of the input argument.

For each component of vector h compute the smallest integer value not less than h.

Parameters
h – **[in]** - nv_bfloat162. Is only being read.

Returns
nv_bfloat162

- The vector of smallest integers not less than h.

__device__ *__nv_bfloat162* **h2cos**(const *__nv_bfloat162* a)

Calculates nv_bfloat162 vector cosine in round-to-nearest-even mode.

Calculates nv_bfloat162 cosine of input vector a in round-to-nearest-even mode.

NOTE: this function's implementation calls *cosf(float)* function and is exposed to compiler optimizations. Specifically, --use_fast_math flag changes *cosf(float)* into an intrinsic *__cosf(float)*, which has less accurate numeric behavior.

Parameters
a – **[in]** - nv_bfloat162. Is only being read.

Returns
nv_bfloat162

- The elementwise cosine on vector a.

__device__ *__nv_bfloat162* **h2exp**(const *__nv_bfloat162* a)

Calculates nv_bfloat162 vector exponential function in round-to-nearest-even mode.

Calculates nv_bfloat162 exponential function of input vector a in round-to-nearest-even mode.

Parameters
a – **[in]** - nv_bfloat162. Is only being read.

Returns
nv_bfloat162

- The elementwise exponential function on vector a.

__device__ *__nv_bfloat162* **h2exp10**(const *__nv_bfloat162* a)

Calculates nv_bfloat162 vector decimal exponential function in round-to-nearest-even mode.

Calculates nv_bfloat162 decimal exponential function of input vector a in round-to-nearest-even mode.

Parameters

a – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

- The elementwise decimal exponential function on vector a.

__device__ *__nv_bfloat162* **h2exp2**(const *__nv_bfloat162* a)

Calculates nv_bfloat162 vector binary exponential function in round-to-nearest-even mode.

Calculates nv_bfloat162 binary exponential function of input vector a in round-to-nearest-even mode.

Parameters

a – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

- The elementwise binary exponential function on vector a.

__device__ *__nv_bfloat162* **h2floor**(const *__nv_bfloat162* h)

Calculate the largest integer less than or equal to h.

For each component of vector h calculate the largest integer value which is less than or equal to h.

Parameters

h – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

- The vector of largest integers which is less than or equal to h.

__device__ *__nv_bfloat162* **h2log**(const *__nv_bfloat162* a)

Calculates nv_bfloat162 vector natural logarithm in round-to-nearest-even mode.

Calculates nv_bfloat162 natural logarithm of input vector a in round-to-nearest-even mode.

Parameters

a – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

- The elementwise natural logarithm on vector a.

__device__ *__nv_bfloat162* **h2log10**(const *__nv_bfloat162* a)

Calculates nv_bfloat162 vector decimal logarithm in round-to-nearest-even mode.

Calculates nv_bfloat162 decimal logarithm of input vector a in round-to-nearest-even mode.

Parameters

a – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

- The elementwise decimal logarithm on vector a.

__device__ *__nv_bfloat162* **h2log2**(const *__nv_bfloat162* a)

Calculates nv_bfloat162 vector binary logarithm in round-to-nearest-even mode.

Calculates `nv_bfloat162` binary logarithm of input vector a in round-to-nearest-even mode.

Parameters

a – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

- The elementwise binary logarithm on vector a.

__device__ *__nv_bfloat162* **h2rcp**(const *__nv_bfloat162* a)

Calculates `nv_bfloat162` vector reciprocal in round-to-nearest-even mode.

Calculates `nv_bfloat162` reciprocal of input vector a in round-to-nearest-even mode.

Parameters

a – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

- The elementwise reciprocal on vector a.

__device__ *__nv_bfloat162* **h2rint**(const *__nv_bfloat162* h)

Round input to nearest integer value in nv_bfloat16 floating-point number.

Round each component of `nv_bfloat162` vector h to the nearest integer value in nv_bfloat16 floating-point format, with bfloat16way cases rounded to the nearest even integer value.

Parameters

h – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

- The vector of rounded integer values.

__device__ *__nv_bfloat162* **h2rsqrt**(const *__nv_bfloat162* a)

Calculates `nv_bfloat162` vector reciprocal square root in round-to-nearest-even mode.

Calculates `nv_bfloat162` reciprocal square root of input vector a in round-to-nearest-even mode.

Parameters

a – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

- The elementwise reciprocal square root on vector a.

__device__ *__nv_bfloat162* **h2sin**(const *__nv_bfloat162* a)

Calculates `nv_bfloat162` vector sine in round-to-nearest-even mode.

Calculates `nv_bfloat162` sine of input vector a in round-to-nearest-even mode.

NOTE: this function's implementation calls *sinf(float)* function and is exposed to compiler optimizations. Specifically, `--use_fast_math` flag changes *sinf(float)* into an intrinsic *__sinf(float)*, which has less accurate numeric behavior.

Parameters
a – **[in]** - nv_bfloat162. Is only being read.

Returns
nv_bfloat162

- The elementwise sine on vector a.

__device__ *__nv_bfloat162* **h2sqrt**(const *__nv_bfloat162* a)

Calculates `nv_bfloat162` vector square root in round-to-nearest-even mode.

Calculates `nv_bfloat162` square root of input vector a in round-to-nearest-even mode.

Parameters
a – **[in]** - nv_bfloat162. Is only being read.

Returns
nv_bfloat162

- The elementwise square root on vector a.

__device__ *__nv_bfloat162* **h2tanh**(const *__nv_bfloat162* a)

Calculates `nv_bfloat162` vector hyperbolic tangent function in round-to-nearest-even mode.

Calculates `nv_bfloat162` hyperbolic tangent function of input vector a in round-to-nearest-even mode.

See also:

htanh(__nv_bfloat16) for further details.

Parameters
a – **[in]** - nv_bfloat162. Is only being read.

Returns
nv_bfloat162

- The elementwise hyperbolic tangent function on vector a.

__device__ *__nv_bfloat162* **h2tanh_approx**(const *__nv_bfloat162* a)

Calculates `nv_bfloat162` vector approximate hyperbolic tangent function.

Calculates `nv_bfloat162` approximate hyperbolic tangent function of input vector a. This operation uses HW acceleration on devices of compute capability 9.x and higher.

See also:

htanh_approx(__nv_bfloat16) for further details.

Parameters
a – **[in]** - nv_bfloat162. Is only being read.

Returns
nv_bfloat162

- The elementwise approximate hyperbolic tangent function on vector a.

__device__ *__nv_bfloat162* **h2trunc**(const *__nv_bfloat162* h)

Truncate `nv_bfloat162` vector input argument to the integral part.

Round each component of vector h to the nearest integer value that does not exceed h in magnitude.

Parameters

h – **[in]** - nv_bfloat162. Is only being read.

Returns

nv_bfloat162

- The truncated h.

Groups

Bfloat16 Arithmetic Constants

To use these constants, include the header file `cuda_bf16.h` in your program.

Bfloat16 Arithmetic Functions

To use these functions, include the header file `cuda_bf16.h` in your program.

Bfloat16 Comparison Functions

To use these functions, include the header file `cuda_bf16.h` in your program.

Bfloat16 Math Functions

To use these functions, include the header file `cuda_bf16.h` in your program.

Bfloat16 Precision Conversion and Data Movement

To use these functions, include the header file `cuda_bf16.h` in your program.

Bfloat162 Arithmetic Functions

To use these functions, include the header file `cuda_bf16.h` in your program.

Bfloat162 Comparison Functions

To use these functions, include the header file `cuda_bf16.h` in your program.

Bfloat162 Math Functions

To use these functions, include the header file `cuda_bf16.h` in your program.

Structs

__nv_bfloat16

nv_bfloat16 datatype

__nv_bfloat162

nv_bfloat162 datatype

__nv_bfloat162_raw

__nv_bfloat162_raw data type

__nv_bfloat16_raw

__nv_bfloat16_raw data type

Typedefs

nv_bfloat16
This datatype is meant to be the first-class or fundamental implementation of the bfloat16 numbers format.

nv_bfloat162
This datatype is meant to be the first-class or fundamental implementation of type for pairs of bfloat16 numbers.

5.9. Typedefs

typedef *__nv_bfloat16* **nv_bfloat16**

This datatype is meant to be the first-class or fundamental implementation of the bfloat16 numbers format.

Should be implemented in the compiler in the future. Current implementation is a simple typedef to a respective user-level type with underscores.

typedef *__nv_bfloat162* **nv_bfloat162**

This datatype is meant to be the first-class or fundamental implementation of type for pairs of bfloat16 numbers.

Should be implemented in the compiler in the future. Current implementation is a simple typedef to a respective user-level type with underscores.

Chapter 6. Single Precision Mathematical Functions

This section describes single precision mathematical functions.

To use these functions, you do not need to include any additional header file in your program.

Functions

__device__ float *acosf*(float x)
Calculate the arc cosine of the input argument.

__device__ float *acoshf*(float x)
Calculate the nonnegative inverse hyperbolic cosine of the input argument.

__device__ float *asinf*(float x)
Calculate the arc sine of the input argument.

__device__ float *asinhf*(float x)
Calculate the inverse hyperbolic sine of the input argument.

__device__ float *atan2f*(float y, float x)
Calculate the arc tangent of the ratio of first and second input arguments.

__device__ float *atanf*(float x)
Calculate the arc tangent of the input argument.

__device__ float *atanhf*(float x)
Calculate the inverse hyperbolic tangent of the input argument.

__device__ float *cbrtf*(float x)
Calculate the cube root of the input argument.

__device__ float *ceilf*(float x)
Calculate ceiling of the input argument.

__device__ float *copysignf*(float x, float y)
Create value with given magnitude, copying sign of second value.

__device__ float *cosf*(float x)
Calculate the cosine of the input argument.

__device__ float *coshf*(float x)
Calculate the hyperbolic cosine of the input argument.

__device__ float *cospif*(float x)
Calculate the cosine of the input argument $\times \pi$.

__device__ float *cyl_bessel_i0f*(float x)
Calculate the value of the regular modified cylindrical Bessel function of order 0 for the input argument.

__device__ float *cyl_bessel_i1f*(float x)
Calculate the value of the regular modified cylindrical Bessel function of order 1 for the input argument.

__device__ float *erfcf*(float x)
Calculate the complementary error function of the input argument.

__device__ float *erfcinvf*(float x)
Calculate the inverse complementary error function of the input argument.

__device__ float *erfcxf*(float x)
Calculate the scaled complementary error function of the input argument.

__device__ float *erff*(float x)
Calculate the error function of the input argument.

__device__ float *erfinvf*(float x)
Calculate the inverse error function of the input argument.

__device__ float *exp10f*(float x)
Calculate the base 10 exponential of the input argument.

__device__ float *exp2f*(float x)
Calculate the base 2 exponential of the input argument.

__device__ float *expf*(float x)
Calculate the base e exponential of the input argument.

__device__ float *expm1f*(float x)
Calculate the base e exponential of the input argument, minus 1.

__device__ float *fabsf*(float x)
Calculate the absolute value of its argument.

__device__ float *fdimf*(float x, float y)
Compute the positive difference between x and y .

__device__ float *fdividef*(float x, float y)
Divide two floating-point values.

__device__ float *floorf*(float x)
Calculate the largest integer less than or equal to x .

__device__ float *fmaf*(float x, float y, float z)
Compute $x \times y + z$ as a single operation.

__device__ float *fmaxf*(float x, float y)
Determine the maximum numeric value of the arguments.

__device__ float *fminf*(float x, float y)
Determine the minimum numeric value of the arguments.

__device__ float *fmodf*(float x, float y)
Calculate the floating-point remainder of x / y .

__device__ float *frexpf*(float x, int *nptr)
Extract mantissa and exponent of a floating-point value.

__device__ float *hypotf*(float x, float y)
Calculate the square root of the sum of squares of two arguments.

__device__ int *ilogbf*(float x)
Compute the unbiased integer exponent of the argument.

__device__ __RETURN_TYPE *isfinite*(float a)
Determine whether argument is finite.

__device__ __RETURN_TYPE *isinf*(float a)
Determine whether argument is infinite.

__device__ __RETURN_TYPE *isnan*(float a)
Determine whether argument is a NaN.

__device__ float *j0f*(float x)
Calculate the value of the Bessel function of the first kind of order 0 for the input argument.

__device__ float *j1f*(float x)
Calculate the value of the Bessel function of the first kind of order 1 for the input argument.

__device__ float *jnf*(int n, float x)
Calculate the value of the Bessel function of the first kind of order n for the input argument.

__device__ float *ldexpf*(float x, int exp)
Calculate the value of $x \cdot 2^{exp}$.

__device__ float *lgammaf*(float x)
Calculate the natural logarithm of the absolute value of the gamma function of the input argument.

__device__ long long int *llrintf*(float x)
Round input to nearest integer value.

__device__ long long int *llroundf*(float x)
Round to nearest integer value.

__device__ float *log10f*(float x)
Calculate the base 10 logarithm of the input argument.

__device__ float *log1pf*(float x)
Calculate the value of $\log_e(1 + x)$.

__device__ float *log2f*(float x)
Calculate the base 2 logarithm of the input argument.

__device__ float *logbf*(float x)
Calculate the floating-point representation of the exponent of the input argument.

__device__ float *logf*(float x)
Calculate the natural logarithm of the input argument.

__device__ long int *lrintf*(float x)
Round input to nearest integer value.

__device__ long int *lroundf*(float x)
Round to nearest integer value.

__device__ float *max*(const float a, const float b)
Calculate the maximum value of the input `float` arguments.

__device__ float *min*(const float a, const float b)
Calculate the minimum value of the input `float` arguments.

__device__ float *modff*(float x, float *iptr)
Break down the input argument into fractional and integral parts.

__device__ float *nanf*(const char *tagp)
Returns "Not a Number" value.

__device__ float *nearbyintf*(float x)
Round the input argument to the nearest integer.

__device__ float *nextafterf*(float x, float y)
Return next representable single-precision floating-point value after argument x in the direction of y .

__device__ float *norm3df*(float a, float b, float c)
Calculate the square root of the sum of squares of three coordinates of the argument.

__device__ float *norm4df*(float a, float b, float c, float d)
Calculate the square root of the sum of squares of four coordinates of the argument.

__device__ float *normcdff*(float x)
Calculate the standard normal cumulative distribution function.

__device__ float *normcdfinvf*(float x)
Calculate the inverse of the standard normal cumulative distribution function.

__device__ float *normf*(int dim, float const *p)
Calculate the square root of the sum of squares of any number of coordinates.

__device__ float *powf*(float x, float y)
Calculate the value of first argument to the power of second argument.

__device__ float *rcbrtf*(float x)
Calculate reciprocal cube root function.

__device__ float *remainderf*(float x, float y)
Compute single-precision floating-point remainder.

__device__ float *remquof*(float x, float y, int *quo)
Compute single-precision floating-point remainder and part of quotient.

__device__ float *rhypotf*(float x, float y)
Calculate one over the square root of the sum of squares of two arguments.

__device__ float *rintf*(float x)
Round input to nearest integer value in floating-point.

__device__ float *rnorm3df*(float a, float b, float c)
Calculate one over the square root of the sum of squares of three coordinates.

__device__ float *rnorm4df*(float a, float b, float c, float d)
Calculate one over the square root of the sum of squares of four coordinates.

__device__ float *rnormf*(int dim, float const *p)
Calculate the reciprocal of square root of the sum of squares of any number of coordinates.

__device__ float *roundf*(float x)
Round to nearest integer value in floating-point.

__device__ float *rsqrtf*(float x)
Calculate the reciprocal of the square root of the input argument.

__device__ float *scalblnf*(float x, long int n)
Scale floating-point input by integer power of two.

__device__ float *scalbnf*(float x, int n)
Scale floating-point input by integer power of two.

__device__ __RETURN_TYPE *signbit*(float a)
Return the sign bit of the input.

__device__ void *sincosf*(float x, float *sptr, float *cptr)
Calculate the sine and cosine of the first input argument.

__device__ void *sincospif*(float x, float *sptr, float *cptr)
Calculate the sine and cosine of the first input argument $\times \pi$.

__device__ float *sinf*(float x)
Calculate the sine of the input argument.

__device__ float *sinhf*(float x)
Calculate the hyperbolic sine of the input argument.

__device__ float *sinpif*(float x)
Calculate the sine of the input argument $\times \pi$.

__device__ float *sqrtf*(float x)
Calculate the square root of the input argument.

__device__ float *tanf*(float x)
Calculate the tangent of the input argument.

__device__ float *tanhf*(float x)
Calculate the hyperbolic tangent of the input argument.

__device__ float *tgammaf*(float x)
Calculate the gamma function of the input argument.

__device__ float *truncf*(float x)
Truncate input argument to the integral part.

__device__ float *y0f*(float x)
Calculate the value of the Bessel function of the second kind of order 0 for the input argument.

__device__ float *y1f*(float x)
Calculate the value of the Bessel function of the second kind of order 1 for the input argument.

__device__ float *ynf*(int n, float x)
Calculate the value of the Bessel function of the second kind of order n for the input argument.

6.1. Functions

__device__ float **acosf** (float x)

Calculate the arc cosine of the input argument.

Calculate the principal value of the arc cosine of the input argument x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

Result will be in radians, in the interval [0, π] for x inside [-1, +1].

- acosf(1) returns +0.

- acosf(x) returns NaN for x outside [-1, +1].
- acosf(NaN) returns NaN.

__device__ float **acoshf** (float x)

Calculate the nonnegative inverse hyperbolic cosine of the input argument.

Calculate the nonnegative inverse hyperbolic cosine of the input argument x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

Result will be in the interval [0, $+\infty$].

- acoshf(1) returns 0.
- acoshf(x) returns NaN for x in the interval [$-\infty$, 1).
- acoshf($+\infty$) returns $+\infty$.
- acoshf(NaN) returns NaN.

__device__ float **asinf** (float x)

Calculate the arc sine of the input argument.

Calculate the principal value of the arc sine of the input argument x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

Result will be in radians, in the interval [- $\pi/2$, + $\pi/2$] for x inside [-1, +1].

- asinf(± 0) returns ± 0.
- asinf(x) returns NaN for x outside [-1, +1].
- asinf(NaN) returns NaN.

__device__ float **asinhf** (float x)

Calculate the inverse hyperbolic sine of the input argument.

Calculate the inverse hyperbolic sine of the input argument x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

- asinhf(± 0) returns ± 0.
- asinhf($\pm\infty$) returns $\pm\infty$.
- asinhf(NaN) returns NaN.

__device__ float **atan2f** (float y, float x)

Calculate the arc tangent of the ratio of first and second input arguments.

Calculate the principal value of the arc tangent of the ratio of first and second input arguments y / x. The quadrant of the result is determined by the signs of inputs y and x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

Result will be in radians, in the interval [- π , + π].

- atan2f(± 0 , -0) returns $\pm\pi$.
- atan2f(± 0 , +0) returns ± 0.
- atan2f(± 0 , x) returns $\pm\pi$ for x < 0.
- atan2f(± 0 , x) returns ± 0 for x > 0.
- atan2f(y, ± 0) returns $-\pi$ /2 for y < 0.
- atan2f(y, ± 0) returns π /2 for y > 0.
- atan2f($\pm y$, $-\infty$) returns $\pm\pi$ for finite y > 0.
- atan2f($\pm y$, $+\infty$) returns ± 0 for finite y > 0.
- atan2f($\pm\infty$, x) returns $\pm\pi$ /2 for finite x.
- atan2f($\pm\infty$, $-\infty$) returns $\pm 3\pi$ /4.
- atan2f($\pm\infty$, $+\infty$) returns $\pm\pi$ /4.
- If either argument is NaN, NaN is returned.

__device__ float **atanf** (float x)

Calculate the arc tangent of the input argument.

Calculate the principal value of the arc tangent of the input argument x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

Result will be in radians, in the interval [- $\pi/2$, + $\pi/2$].

- atanf(± 0) returns ± 0.
- atanf($\pm\infty$) returns $\pm\pi$ /2.
- atanf(NaN) returns NaN.

__device__ float **atanhf** (float x)

Calculate the inverse hyperbolic tangent of the input argument.

Calculate the inverse hyperbolic tangent of the input argument x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

- atanhf(± 0) returns ± 0.
- atanhf(± 1) returns $\pm \infty$.
- atanhf(x) returns NaN for x outside interval [-1, 1].
- atanhf(NaN) returns NaN.

__device__ float **cbrtf**(float x)

Calculate the cube root of the input argument.

Calculate the cube root of x, $x^{1/3}$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

Returns $x^{1/3}$.

- cbrtf(± 0) returns ± 0.
- cbrtf($\pm \infty$) returns $\pm \infty$.
- cbrtf(NaN) returns NaN.

__device__ float **ceilf**(float x)

Calculate ceiling of the input argument.

Compute the smallest integer value not less than x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

Returns $\lceil x \rceil$ expressed as a floating-point number.

- ceilf(± 0) returns ± 0.
- ceilf($\pm \infty$) returns $\pm \infty$.
- ceilf(NaN) returns NaN.

__device__ float **copysignf**(float x, float y)

Create value with given magnitude, copying sign of second value.

Create a floating-point value with the magnitude x and the sign of y.

Returns

- a value with the magnitude of x and the sign of y.
- copysignf(NaN, y) returns a NaN with the sign of y.

__device__ float **cosf** (float x)

Calculate the cosine of the input argument.

Calculate the cosine of the input argument x (measured in radians).

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Note: This function is affected by the `use_fast_math` compiler flag. See the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section for a complete list of functions affected.

Returns

- cosf(± 0) returns 1.
- cosf($\pm\infty$) returns NaN.
- cosf(NaN) returns NaN.

__device__ float **coshf** (float x)

Calculate the hyperbolic cosine of the input argument.

Calculate the hyperbolic cosine of the input argument x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

- coshf(± 0) returns 1.
- coshf($\pm\infty$) returns $+\infty$.
- coshf(NaN) returns NaN.

__device__ float **cospif** (float x)

Calculate the cosine of the input argument $\times\pi$.

Calculate the cosine of x $\times\pi$ (measured in radians), where x is the input argument.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

- cospif(± 0) returns 1.
- cospif($\pm\infty$) returns NaN.
- cospif(NaN) returns NaN.

__device__ float **cyl_bessel_i0f**(float x)

Calculate the value of the regular modified cylindrical Bessel function of order 0 for the input argument.

Calculate the value of the regular modified cylindrical Bessel function of order 0 for the input argument x, $I_0(x)$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

Returns the value of the regular modified cylindrical Bessel function of order 0.

- cyl_bessel_i0f(± 0) returns +1.
- cyl_bessel_i0f($\pm\infty$) returns $+\infty$.
- cyl_bessel_i0f(NaN) returns NaN.

__device__ float **cyl_bessel_i1f**(float x)

Calculate the value of the regular modified cylindrical Bessel function of order 1 for the input argument.

Calculate the value of the regular modified cylindrical Bessel function of order 1 for the input argument x, $I_1(x)$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

Returns the value of the regular modified cylindrical Bessel function of order 1.

- cyl_bessel_i1f(± 0) returns ± 0.
- cyl_bessel_i1f($\pm\infty$) returns $\pm\infty$.
- cyl_bessel_i1f(NaN) returns NaN.

__device__ float **erfcf**(float x)

Calculate the complementary error function of the input argument.

Calculate the complementary error function of the input argument x, 1 - erf(x).

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

- erfcf($-\infty$) returns 2.
- erfcf($+\infty$) returns +0.
- erfcf(NaN) returns NaN.

__device__ float **erfcinvf** (float x)

Calculate the inverse complementary error function of the input argument.

Calculate the inverse complementary error function erfc^{-1} (x), of the input argument x in the interval [0, 2].

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

- erfcinvf(± 0) returns $+\infty$.
- erfcinvf(2) returns $-\infty$.
- erfcinvf(x) returns NaN for x outside [0, 2].
- erfcinvf(NaN) returns NaN.

__device__ float **erfcxf** (float x)

Calculate the scaled complementary error function of the input argument.

Calculate the scaled complementary error function of the input argument x, $e^{x^2} \cdot \mathrm{erfc}(x)$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

- erfcxf($-\infty$) returns $+\infty$.
- erfcxf($+\infty$) returns +0.
- erfcxf(NaN) returns NaN.

__device__ float **erff** (float x)

Calculate the error function of the input argument.

Calculate the value of the error function for the input argument x, $\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

- erff(± 0) returns ± 0.

- erff($\pm\infty$) returns ± 1.
- erff(NaN) returns NaN.

__device__ float **erfinvf**(float x)

Calculate the inverse error function of the input argument.

Calculate the inverse error function erf^{-1} (x), of the input argument x in the interval [-1, 1].

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

- erfinvf(± 0) returns ± 0.
- erfinvf(1) returns $+\infty$.
- erfinvf(-1) returns $-\infty$.
- erfinvf(x) returns NaN for x outside [-1, +1].
- erfinvf(NaN) returns NaN.

__device__ float **exp10f**(float x)

Calculate the base 10 exponential of the input argument.

Calculate 10^x , the base 10 exponential of the input argument x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Note: This function is affected by the `use_fast_math` compiler flag. See the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section for a complete list of functions affected.

Returns

- exp10f(± 0) returns 1.
- exp10f($-\infty$) returns +0.
- exp10f($+\infty$) returns $+\infty$.
- exp10f(NaN) returns NaN.

__device__ float **exp2f**(float x)

Calculate the base 2 exponential of the input argument.

Calculate 2^x , the base 2 exponential of the input argument x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

- exp2f(± 0) returns 1.
- exp2f($-\infty$) returns +0.
- exp2f($+\infty$) returns $+\infty$.
- exp2f(NaN) returns NaN.

__device__ float **expf** (float x)

Calculate the base e exponential of the input argument.

Calculate e^x , the base e exponential of the input argument x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Note: This function is affected by the `use_fast_math` compiler flag. See the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section for a complete list of functions affected.

Returns

- expf(± 0) returns 1.
- expf($-\infty$) returns +0.
- expf($+\infty$) returns $+\infty$.
- expf(NaN) returns NaN.

__device__ float **expm1f** (float x)

Calculate the base e exponential of the input argument, minus 1.

Calculate e^x -1, the base e exponential of the input argument x, minus 1.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

- expm1f(± 0) returns ± 0.
- expm1f($-\infty$) returns -1.
- expm1f($+\infty$) returns $+\infty$.

- expm1f(NaN) returns NaN.

__device__ float **fabsf**(float x)

Calculate the absolute value of its argument.

Calculate the absolute value of the input argument x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

Returns the absolute value of its argument.

- fabsf($\pm\infty$) returns $+\infty$.
- fabsf(± 0) returns +0.
- fabsf(NaN) returns an unspecified NaN.

__device__ float **fdimf**(float x, float y)

Compute the positive difference between x and y.

Compute the positive difference between x and y. The positive difference is x - y when x > y and +0 otherwise.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

Returns the positive difference between x and y.

- fdimf(x, y) returns x - y if x > y.
- fdimf(x, y) returns +0 if x $\leq$ y.
- If either argument is NaN, NaN is returned.

__device__ float **fdividef**(float x, float y)

Divide two floating-point values.

Compute x divided by y.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Note: This function is affected by the `use_fast_math` compiler flag. See the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section for a complete list of functions affected.

Returns

Returns x / y.

- Follows the regular division operation behavior by default.
- If `-use_fast_math` is specified and is not amended by an explicit `-prec_div=true`, uses *__fdividef()* for higher performance

__device__ float **floorf**(float x)

Calculate the largest integer less than or equal to x.

Calculate the largest integer value which is less than or equal to x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

Returns $\lfloor x \rfloor$ expressed as a floating-point number.

- floorf($\pm\infty$) returns $\pm\infty$.
- floorf(± 0) returns ± 0.
- floorf(NaN) returns NaN.

__device__ float **fmaf**(float x, float y, float z)

Compute $x \times y + z$ as a single operation.

Compute the value of $x \times y + z$ as a single ternary operation. After computing the value to infinite precision, the value is rounded once using round-to-nearest, ties-to-even rounding mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

Returns the rounded value of $x \times y + z$ as a single operation.

- fmaf($\pm\infty$, ± 0 , z) returns NaN.
- fmaf(± 0 , $\pm\infty$, z) returns NaN.
- fmaf(x, y, $-\infty$) returns NaN if $x \times y$ is an exact $+\infty$.
- fmaf(x, y, $+\infty$) returns NaN if $x \times y$ is an exact $-\infty$.
- fmaf(x, y, ± 0) returns ± 0 if $x \times y$ is exact ± 0.
- fmaf(x, y, ∓ 0) returns $+0$ if $x \times y$ is exact ± 0.
- fmaf(x, y, z) returns $+0$ if $x \times y + z$ is exactly zero and $z \neq 0$.
- If either argument is NaN, NaN is returned.

__device__ float **fmaxf**(float x, float y)

Determine the maximum numeric value of the arguments.

Determines the maximum numeric value of the arguments x and y. Treats NaN arguments as missing data. If one argument is a NaN and the other is legitimate numeric value, the numeric value is chosen.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

Returns the maximum numeric values of the arguments x and y.

- If both arguments are NaN, returns NaN.
- If one argument is NaN, returns the numeric argument.

__device__ float **fminf**(float x, float y)

Determine the minimum numeric value of the arguments.

Determines the minimum numeric value of the arguments x and y. Treats NaN arguments as missing data. If one argument is a NaN and the other is legitimate numeric value, the numeric value is chosen.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

Returns the minimum numeric value of the arguments x and y.

- If both arguments are NaN, returns NaN.
- If one argument is NaN, returns the numeric argument.

__device__ float **fmodf**(float x, float y)

Calculate the floating-point remainder of x / y.

Calculate the floating-point remainder of x / y. The floating-point remainder of the division operation x / y calculated by this function is exactly the value `x - n*y`, where n is x / y with its fractional part truncated. The computed value will have the same sign as x, and its magnitude will be less than the magnitude of y.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

- Returns the floating-point remainder of x / y.
- fmodf(± 0 , y) returns ± 0 if y is not zero.
- fmodf(x, $\pm\infty$) returns x if x is finite.
- fmodf(x, y) returns NaN if x is $\pm\infty$ or y is zero.
- If either argument is NaN, NaN is returned.

__device__ float **frexpf**(float x, int *nptr)

Extract mantissa and exponent of a floating-point value.

Decomposes the floating-point value `x` into a component `m` for the normalized fraction element and another term `n` for the exponent. The absolute value of `m` will be greater than or equal to 0.5 and less than 1.0 or it will be equal to 0; $x = m \cdot 2^n$. The integer exponent `n` will be stored in the location to which `nptr` points.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

Returns the fractional component `m`.

- frexpf(± 0 , `nptr`) returns ± 0 and stores zero in the location pointed to by `nptr`.
- frexpf($\pm\infty$, `nptr`) returns $\pm\infty$ and stores an unspecified value in the location to which `nptr` points.
- frexpf(NaN, y) returns a NaN and stores an unspecified value in the location to which `nptr` points.

__device__ float **hypotf**(float x, float y)

Calculate the square root of the sum of squares of two arguments.

Calculates the length of the hypotenuse of a right triangle whose two sides have lengths `x` and `y` without undue overflow or underflow.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

Returns the length of the hypotenuse $\sqrt{x^2 + y^2}$.

- hypotf(x,y), hypotf(y,x), and hypotf(x, -y) are equivalent.
- hypotf(x, ± 0) is equivalent to fabsf(x).
- hypotf($\pm\infty$,y) returns $+\infty$, even if y is a NaN.
- hypotf(NaN, y) returns NaN, when y is not $\pm\infty$.

__device__ int **ilogbf**(float x)

Compute the unbiased integer exponent of the argument.

Calculates the unbiased integer exponent of the input argument `x`.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

- If successful, returns the unbiased exponent of the argument.

- ilogbf(± 0) returns INT_MIN.
- ilogbf(NaN) returns INT_MIN.
- ilogbf($\pm\infty$) returns INT_MAX.
- Note: above behavior does not take into account FP_ILOGB0 nor FP_ILOGBNAN.

__device__ __RETURN_TYPE **isfinite**(float a)

Determine whether argument is finite.

Determine whether the floating-point value a is a finite value (zero, subnormal, or normal and not infinity or NaN).

Returns

- With Visual Studio 2013 host compiler: __RETURN_TYPE is 'bool'. Returns true if and only if a is a finite value.
- With other host compilers: __RETURN_TYPE is 'int'. Returns a nonzero value if and only if a is a finite value.

__device__ __RETURN_TYPE **isinf**(float a)

Determine whether argument is infinite.

Determine whether the floating-point value a is an infinite value (positive or negative).

Returns

- With Visual Studio 2013 host compiler: __RETURN_TYPE is 'bool'. Returns true if and only if a is an infinite value.
- With other host compilers: __RETURN_TYPE is 'int'. Returns a nonzero value if and only if a is an infinite value.

__device__ __RETURN_TYPE **isnan**(float a)

Determine whether argument is a NaN.

Determine whether the floating-point value a is a NaN.

Returns

- With Visual Studio 2013 host compiler: __RETURN_TYPE is 'bool'. Returns true if and only if a is a NaN value.
- With other host compilers: __RETURN_TYPE is 'int'. Returns a nonzero value if and only if a is a NaN value.

__device__ float **j0f**(float x)

Calculate the value of the Bessel function of the first kind of order 0 for the input argument.

Calculate the value of the Bessel function of the first kind of order 0 for the input argument x, $J_0(x)$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

Returns the value of the Bessel function of the first kind of order 0.

- j0f($\pm\infty$) returns +0.
- j0f(NaN) returns NaN.

__device__ float **j1f** (float x)

Calculate the value of the Bessel function of the first kind of order 1 for the input argument.

Calculate the value of the Bessel function of the first kind of order 1 for the input argument x, $J_1(x)$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

Returns the value of the Bessel function of the first kind of order 1.

- j1f(± 0) returns ± 0.
- j1f($\pm\infty$) returns ± 0.
- j1f(NaN) returns NaN.

__device__ float **jnf** (int n, float x)

Calculate the value of the Bessel function of the first kind of order n for the input argument.

Calculate the value of the Bessel function of the first kind of order n for the input argument x, $J_n(x)$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

Returns the value of the Bessel function of the first kind of order n.

- jnf(n, NaN) returns NaN.
- jnf(n, x) returns NaN for n < 0.
- jnf(n, $+\infty$) returns +0.

__device__ float **ldexpf** (float x, int exp)

Calculate the value of $x \cdot 2^{exp}$.

Calculate the value of $x \cdot 2^{exp}$ of the input arguments x and exp.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

- ldexpf(x, exp) is equivalent to scalbnf(x, exp).

__device__ float **lgammaf** (float x)

Calculate the natural logarithm of the absolute value of the gamma function of the input argument.

Calculate the natural logarithm of the absolute value of the gamma function of the input argument x, namely the value of $\log_e |\Gamma(x)|$

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

- lgammaf(1) returns +0.
- lgammaf(2) returns +0.
- lgammaf(x) returns $+\infty$ if x $\leq$ 0 and x is an integer.
- lgammaf($-\infty$) returns $+\infty$.
- lgammaf($+\infty$) returns $+\infty$.
- lgammaf(NaN) returns NaN.

__device__ long long int **llrintf** (float x)

Round input to nearest integer value.

Round x to the nearest integer value, with halfway cases rounded to the nearest even integer value. If the result is outside the range of the return type, the behavior is undefined.

Returns

Returns rounded integer value.

__device__ long long int **llroundf** (float x)

Round to nearest integer value.

Round x to the nearest integer value, with halfway cases rounded away from zero. If the result is outside the range of the return type, the behavior is undefined.

Note: This function may be slower than alternate rounding methods. See *llrintf()*.

Returns

Returns rounded integer value.

__device__ float **log10f** (float x)

Calculate the base 10 logarithm of the input argument.

Calculate the base 10 logarithm of the input argument x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Note: This function is affected by the `use_fast_math` compiler flag. See the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section for a complete list of functions affected.

Returns

- log10f(± 0) returns $-\infty$.
- log10f(1) returns +0.
- log10f(x) returns NaN for x < 0.
- log10f($+\infty$) returns $+\infty$.
- log10f(NaN) returns NaN.

__device__ float **log1pf** (float x)

Calculate the value of $\log_e(1 + x)$.

Calculate the value of $\log_e(1 + x)$ of the input argument x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

- log1pf(± 0) returns ± 0.
- log1pf(-1) returns $-\infty$.
- log1pf(x) returns NaN for x < -1.
- log1pf($+\infty$) returns $+\infty$.
- log1pf(NaN) returns NaN.

__device__ float **log2f** (float x)

Calculate the base 2 logarithm of the input argument.

Calculate the base 2 logarithm of the input argument x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Note: This function is affected by the `use_fast_math` compiler flag. See the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section for a complete list of functions affected.

Returns

- log2f(± 0) returns $-\infty$.
- log2f(1) returns +0.
- log2f(x) returns NaN for x < 0.
- log2f($+\infty$) returns $+\infty$.
- log2f(NaN) returns NaN.

__device__ float **logbf**(float x)

Calculate the floating-point representation of the exponent of the input argument.

Calculate the floating-point representation of the exponent of the input argument x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

- logbf(± 0) returns $-\infty$.
- logbf($\pm\infty$) returns $+\infty$.
- logbf(NaN) returns NaN.

__device__ float **logf**(float x)

Calculate the natural logarithm of the input argument.

Calculate the natural logarithm of the input argument x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Note: This function is affected by the `use_fast_math` compiler flag. See the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section for a complete list of functions affected.

Returns

- logf(± 0) returns $-\infty$.
- logf(1) returns +0.
- logf(x) returns NaN for x < 0.
- logf($+\infty$) returns $+\infty$.
- logf(NaN) returns NaN.

__device__ long int **lrintf**(float x)

Round input to nearest integer value.

Round x to the nearest integer value, with halfway cases rounded to the nearest even integer value. If the result is outside the range of the return type, the behavior is undefined.

Returns

Returns rounded integer value.

__device__ long int **lroundf**(float x)

Round to nearest integer value.

Round x to the nearest integer value, with halfway cases rounded away from zero. If the result is outside the range of the return type, the behavior is undefined.

Note: This function may be slower than alternate rounding methods. See *lrintf()*.

Returns

Returns rounded integer value.

__device__ float **max**(const float a, const float b)

Calculate the maximum value of the input `float` arguments.

Calculate the maximum value of the arguments a and b. Behavior is equivalent to *fmaxf()* function.

Note, this is different from `std::` specification

__device__ float **min**(const float a, const float b)

Calculate the minimum value of the input `float` arguments.

Calculate the minimum value of the arguments a and b. Behavior is equivalent to *fminf()* function.

Note, this is different from `std::` specification

__device__ float **modff**(float x, float *iptr)

Break down the input argument into fractional and integral parts.

Break down the argument x into fractional and integral parts. The integral part is stored in the argument `iptr`. Fractional and integral parts are given the same sign as the argument x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

- modff($\pm x$, `iptr`) returns a result with the same sign as x.
- modff($\pm\infty$, `iptr`) returns ± 0 and stores $\pm\infty$ in the object pointed to by `iptr`.
- modff(NaN, `iptr`) stores a NaN in the object pointed to by `iptr` and returns a NaN.

__device__ float **nanf**(const char *tagp)

Returns "Not a Number" value.

Return a representation of a quiet NaN. Argument `tagp` selects one of the possible representations.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

- nanf(`tagp`) returns NaN.

__device__ float **nearbyintf**(float x)

Round the input argument to the nearest integer.

Round argument `x` to an integer value in single precision floating-point format. Uses round to nearest rounding, with ties rounding to even.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

- nearbyintf(± 0) returns ± 0.
- nearbyintf($\pm\infty$) returns $\pm\infty$.
- nearbyintf(NaN) returns NaN.

__device__ float **nextafterf**(float x, float y)

Return next representable single-precision floating-point value after argument x in the direction of y.

Calculate the next representable single-precision floating-point value following x in the direction of y. For example, if y is greater than x, *nextafterf()* returns the smallest representable number greater than x

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

- nextafterf(x, y) = y if x equals y.
- nextafterf(x, y) = NaN if either x or y are NaN.

__device__ float **norm3df**(float a, float b, float c)

Calculate the square root of the sum of squares of three coordinates of the argument.

Calculates the length of three dimensional vector in Euclidean space without undue overflow or underflow.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

Returns the length of the 3D vector $\sqrt{a^2 + b^2 + c^2}$.

- In the presence of an exactly infinite coordinate $+\infty$ is returned, even if there are NaNs.
- returns +0, when all coordinates are ± 0.
- returns NaN, when at least one of the coordinates is NaN and none are infinite.

__device__ float **norm4df**(float a, float b, float c, float d)

Calculate the square root of the sum of squares of four coordinates of the argument.

Calculates the length of four dimensional vector in Euclidean space without undue overflow or underflow.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

Returns the length of the 4D vector $\sqrt{a^2 + b^2 + c^2 + d^2}$.

- In the presence of an exactly infinite coordinate $+\infty$ is returned, even if there are NaNs.
- returns +0, when all coordinates are ± 0.
- returns NaN, when at least one of the coordinates is NaN and none are infinite.

__device__ float **normcdff**(float x)

Calculate the standard normal cumulative distribution function.

Calculate the cumulative distribution function of the standard normal distribution for input argument x, $\Phi(x)$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

- normcdff($+\infty$) returns 1.
- normcdff($-\infty$) returns +0
- normcdff(NaN) returns NaN.

__device__ float **normcdfinvf**(float x)

Calculate the inverse of the standard normal cumulative distribution function.

Calculate the inverse of the standard normal cumulative distribution function for input argument x, $\Phi^{-1}(x)$. The function is defined for input values in the interval $(0, 1)$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

- normcdfinvf(± 0) returns $-\infty$.
- normcdfinvf(1) returns $+\infty$.
- normcdfinvf(x) returns NaN if x is not in the interval [0,1].
- normcdfinvf(NaN) returns NaN.

__device__ float **normf**(int dim, float const *p)

Calculate the square root of the sum of squares of any number of coordinates.

Calculates the length of a vector p, dimension of which is passed as an argument without undue overflow or underflow.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

Returns the length of the dim-D vector $\sqrt{\sum_{i=0}^{dim-1} p_i^2}$.

- In the presence of an exactly infinite coordinate $+\infty$ is returned, even if there are NaNs.
- returns +0, when all coordinates are ± 0.
- returns NaN, when at least one of the coordinates is NaN and none are infinite.

__device__ float **powf**(float x, float y)

Calculate the value of first argument to the power of second argument.

Calculate the value of x to the power of y.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Note: This function is affected by the `use_fast_math` compiler flag. See the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section for a complete list of functions affected.

Returns

- powf(± 0 , y) returns $\pm\infty$ for y an odd integer less than 0.
- powf(± 0 , y) returns $+\infty$ for y less than 0 and not an odd integer.
- powf(± 0 , y) returns ± 0 for y an odd integer greater than 0.
- powf(± 0 , y) returns +0 for y > 0 and not an odd integer.
- powf(-1, $\pm\infty$) returns 1.
- powf(+1, y) returns 1 for any y, even a NaN.
- powf(x, ± 0) returns 1 for any x, even a NaN.
- powf(x, y) returns a NaN for finite x < 0 and finite non-integer y.
- powf(x, $-\infty$) returns $+\infty$ for $|x| < 1$.
- powf(x, $-\infty$) returns +0 for $|x| > 1$.
- powf(x, $+\infty$) returns +0 for $|x| < 1$.
- powf(x, $+\infty$) returns $+\infty$ for $|x| > 1$.
- powf($-\infty$, y) returns -0 for y an odd integer less than 0.
- powf($-\infty$, y) returns +0 for y < 0 and not an odd integer.
- powf($-\infty$, y) returns $-\infty$ for y an odd integer greater than 0.
- powf($-\infty$, y) returns $+\infty$ for y > 0 and not an odd integer.
- powf($+\infty$, y) returns +0 for y < 0.
- powf($+\infty$, y) returns $+\infty$ for y > 0.
- powf(x, y) returns NaN if either x or y or both are NaN and x $\neq$ +1 and y $\neq \pm 0$.

__device__ float **rcbrtf**(float x)

Calculate reciprocal cube root function.

Calculate reciprocal cube root function of x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

- rcbrtf(± 0) returns $\pm\infty$.
- rcbrtf($\pm\infty$) returns ± 0.
- rcbrtf(NaN) returns NaN.

__device__ float **remainderf**(float x, float y)

Compute single-precision floating-point remainder.

Compute single-precision floating-point remainder r of dividing x by y for nonzero y. Thus $r = x - ny$. The value n is the integer value nearest $\frac{x}{y}$. In the case when $|n - \frac{x}{y}| = \frac{1}{2}$, the even n value is chosen.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

- remainderf(x, ± 0) returns NaN.
- remainderf($\pm\infty$, y) returns NaN.
- remainderf(x, $\pm\infty$) returns x for finite x.
- If either argument is NaN, NaN is returned.

__device__ float **remquof** (float x, float y, int *quo)

Compute single-precision floating-point remainder and part of quotient.

Compute a single-precision floating-point remainder in the same way as the *remainderf()* function. Argument quo returns part of quotient upon division of x by y. Value quo has the same sign as $\frac{x}{y}$ and may not be the exact quotient but agrees with the exact quotient in the low order 3 bits.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

Returns the remainder.

- remquof(x, ± 0 , quo) returns NaN and stores an unspecified value in the location to which quo points.
- remquof($\pm\infty$, y, quo) returns NaN and stores an unspecified value in the location to which quo points.
- remquof(x, y, quo) returns NaN and stores an unspecified value in the location to which quo points if either of x or y is NaN.
- remquof(x, $\pm\infty$, quo) returns x and stores zero in the location to which quo points for finite x.

__device__ float **rhypotf** (float x, float y)

Calculate one over the square root of the sum of squares of two arguments.

Calculates one over the length of the hypotenuse of a right triangle whose two sides have lengths x and y without undue overflow or underflow.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

Returns one over the length of the hypotenuse $\frac{1}{\sqrt{x^2+y^2}}$.

- rhypotf(x,y), rhypotf(y,x), and rhypotf(x, -y) are equivalent.

- rhypotf($\pm\infty$,y) returns +0, even if y is a NaN.
- rhypotf($\pm 0, \pm 0$) returns $+\infty$.
- rhypotf(NaN, y) returns NaN, when y is not $\pm\infty$.

__device__ float **rintf**(float x)

Round input to nearest integer value in floating-point.

Round x to the nearest integer value in floating-point format, with halfway cases rounded to the nearest even integer value.

Returns

Returns rounded integer value.

- rintf(± 0) returns ± 0.
- rintf($\pm\infty$) returns $\pm\infty$.
- rintf(NaN) returns NaN.

__device__ float **rnorm3df**(float a, float b, float c)

Calculate one over the square root of the sum of squares of three coordinates.

Calculates one over the length of three dimension vector in Euclidean space without undue overflow or underflow.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

Returns one over the length of the 3D vector $\frac{1}{\sqrt{a^2+b^2+c^2}}$.

- In the presence of an exactly infinite coordinate $+0$ is returned, even if there are NaNs.
- returns $+\infty$, when all coordinates are ± 0.
- returns NaN, when at least one of the coordinates is NaN and none are infinite.

__device__ float **rnorm4df**(float a, float b, float c, float d)

Calculate one over the square root of the sum of squares of four coordinates.

Calculates one over the length of four dimension vector in Euclidean space without undue overflow or underflow.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

Returns one over the length of the 3D vector $\frac{1}{\sqrt{a^2+b^2+c^2+d^2}}$.

- In the presence of an exactly infinite coordinate $+0$ is returned, even if there are NaNs.
- returns $+\infty$, when all coordinates are ± 0.
- returns NaN, when at least one of the coordinates is NaN and none are infinite.

__device__ float **rnormf**(int dim, float const *p)

Calculate the reciprocal of square root of the sum of squares of any number of coordinates.

Calculates one over the length of vector p, dimension of which is passed as an argument, in Euclidean space without undue overflow or underflow.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

Returns one over the length of the vector $\frac{1}{\sqrt{\sum_{i=0}^{dim-1} p_i^2}}$.

- In the presence of an exactly infinite coordinate $+0$ is returned, even if there are NaNs.
- returns $+\infty$, when all coordinates are ± 0.
- returns NaN, when at least one of the coordinates is NaN and none are infinite.

__device__ float **roundf**(float x)

Round to nearest integer value in floating-point.

Round x to the nearest integer value in floating-point format, with halfway cases rounded away from zero.

Note: This function may be slower than alternate rounding methods. See *rintf()*.

Returns

Returns rounded integer value.

- roundf(± 0) returns ± 0.
- roundf($\pm\infty$) returns $\pm\infty$.
- roundf(NaN) returns NaN.

__device__ float **rsqrtf**(float x)

Calculate the reciprocal of the square root of the input argument.

Calculate the reciprocal of the nonnegative square root of x, $1/\sqrt{x}$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

Returns $1/\sqrt{x}$.

- rsqrtf($+\infty$) returns +0.
- rsqrtf(± 0) returns $\pm\infty$.
- rsqrtf(x) returns NaN if x is less than 0.
- rsqrtf(NaN) returns NaN.

__device__ float **scalblnf** (float x, long int n)

Scale floating-point input by integer power of two.

Scale x by 2^n by efficient manipulation of the floating-point exponent.

Returns

Returns x * 2^n.

- scalblnf(± 0 , n) returns ± 0.
- scalblnf(x, 0) returns x.
- scalblnf($\pm\infty$, n) returns $\pm\infty$.
- scalblnf(NaN, n) returns NaN.

__device__ float **scalbnf** (float x, int n)

Scale floating-point input by integer power of two.

Scale x by 2^n by efficient manipulation of the floating-point exponent.

Returns

Returns x * 2^n.

- scalbnf(± 0 , n) returns ± 0.
- scalbnf(x, 0) returns x.
- scalbnf($\pm\infty$, n) returns $\pm\infty$.
- scalbnf(NaN, n) returns NaN.

__device__ __RETURN_TYPE **signbit** (float a)

Return the sign bit of the input.

Determine whether the floating-point value a is negative.

Returns

Reports the sign bit of all values including infinities, zeros, and NaNs.

- With Visual Studio 2013 host compiler: __RETURN_TYPE is 'bool'. Returns true if and only if a is negative.
- With other host compilers: __RETURN_TYPE is 'int'. Returns a nonzero value if and only if a is negative.

__device__ void **sincosf** (float x, float *sptr, float *cptr)

Calculate the sine and cosine of the first input argument.

Calculate the sine and cosine of the first input argument x (measured in radians). The results for sine and cosine are written into the second argument, `sptr`, and, respectively, third argument, `cptr`.

See also:

sinf() and *cosf()*.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Note: This function is affected by the `use_fast_math` compiler flag. See the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section for a complete list of functions affected.

__device__ void **sincospif** (float x, float *sptr, float *cptr)

Calculate the sine and cosine of the first input argument $\times\pi$.

Calculate the sine and cosine of the first input argument, x (measured in radians), $\times\pi$. The results for sine and cosine are written into the second argument, `sptr`, and, respectively, third argument, `cptr`.

See also:

sinpif() and *cospif()*.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

__device__ float **sinf** (float x)

Calculate the sine of the input argument.

Calculate the sine of the input argument x (measured in radians).

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Note: This function is affected by the `use_fast_math` compiler flag. See the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section for a complete list of functions affected.

Returns

- sinf(± 0) returns ± 0.
- sinf($\pm\infty$) returns NaN.
- sinf(NaN) returns NaN.

__device__ float **sinhf** (float x)

Calculate the hyperbolic sine of the input argument.

Calculate the hyperbolic sine of the input argument x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

- sinhf(± 0) returns ± 0.
- sinhf($\pm\infty$) returns $\pm\infty$.
- sinhf(NaN) returns NaN.

__device__ float **sinpif**(float x)

Calculate the sine of the input argument $\times\pi$.

Calculate the sine of x $\times\pi$ (measured in radians), where x is the input argument.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

- sinpif(± 0) returns ± 0.
- sinpif($\pm\infty$) returns NaN.
- sinpif(NaN) returns NaN.

__device__ float **sqrtf**(float x)

Calculate the square root of the input argument.

Calculate the nonnegative square root of x, $\sqrt{x}$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

Returns $\sqrt{x}$.

- sqrtf(± 0) returns ± 0.
- sqrtf($+\infty$) returns $+\infty$.
- sqrtf(x) returns NaN if x is less than 0.
- sqrtf(NaN) returns NaN.

__device__ float **tanf**(float x)

Calculate the tangent of the input argument.

Calculate the tangent of the input argument x (measured in radians).

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Note: This function is affected by the `use_fast_math` compiler flag. See the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section for a complete list of functions affected.

Returns

- tanf(± 0) returns ± 0.
- tanf($\pm\infty$) returns NaN.
- tanf(NaN) returns NaN.

__device__ float **tanhf** (float x)

Calculate the hyperbolic tangent of the input argument.

Calculate the hyperbolic tangent of the input argument x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

- tanhf(± 0) returns ± 0.
- tanhf($\pm\infty$) returns ± 1.
- tanhf(NaN) returns NaN.

__device__ float **tgammaf** (float x)

Calculate the gamma function of the input argument.

Calculate the gamma function of the input argument x, namely the value of $\Gamma(x)$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

- tgammaf(± 0) returns $\pm\infty$.
- tgammaf(x) returns NaN if x < 0 and x is an integer.
- tgammaf($-\infty$) returns NaN.
- tgammaf($+\infty$) returns $+\infty$.
- tgammaf(NaN) returns NaN.

__device__ float **truncf** (float x)

Truncate input argument to the integral part.

Round x to the nearest integer value that does not exceed x in magnitude.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

Returns truncated integer value.

- truncf(± 0) returns ± 0.
- truncf($\pm \infty$) returns $\pm \infty$.
- truncf(NaN) returns NaN.

__device__ float **y0f** (float x)

Calculate the value of the Bessel function of the second kind of order 0 for the input argument.

Calculate the value of the Bessel function of the second kind of order 0 for the input argument x, $Y_0(x)$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

Returns the value of the Bessel function of the second kind of order 0.

- y0f(± 0) returns $-\infty$.
- y0f(x) returns NaN for x < 0.
- y0f($+\infty$) returns +0.
- y0f(NaN) returns NaN.

__device__ float **y1f** (float x)

Calculate the value of the Bessel function of the second kind of order 1 for the input argument.

Calculate the value of the Bessel function of the second kind of order 1 for the input argument x, $Y_1(x)$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

Returns the value of the Bessel function of the second kind of order 1.

- y1f(± 0) returns $-\infty$.
- y1f(x) returns NaN for x < 0.
- y1f($+\infty$) returns +0.
- y1f(NaN) returns NaN.

__device__ float **ynf** (int n, float x)

Calculate the value of the Bessel function of the second kind of order n for the input argument.

Calculate the value of the Bessel function of the second kind of order n for the input argument x, $Y_n(x)$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

Returns the value of the Bessel function of the second kind of order n.

- ynf(n, x) returns NaN for n < 0.
- ynf(n, ± 0) returns $-\infty$.
- ynf(n, x) returns NaN for x < 0.
- ynf(n, $+\infty$) returns +0.
- ynf(n, NaN) returns NaN.

Chapter 7. Single Precision Intrinsics

This section describes single precision intrinsic functions that are only supported in device code.

To use these functions, you do not need to include any additional header file in your program.

Functions

__device__ float *__cosf*(float x)
Calculate the fast approximate cosine of the input argument.

__device__ float *__exp10f*(float x)
Calculate the fast approximate base 10 exponential of the input argument.

__device__ float *__expf*(float x)
Calculate the fast approximate base e exponential of the input argument.

__device__ float2 *__fadd2_rd*(float2 x, float2 y)
Compute vector add operation $x + y$ in round-down mode.

__device__ float2 *__fadd2_rn*(float2 x, float2 y)
Compute vector add operation $x + y$ in round-to-nearest-even mode.

__device__ float2 *__fadd2_ru*(float2 x, float2 y)
Compute vector add operation $x + y$ in round-up mode.

__device__ float2 *__fadd2_rz*(float2 x, float2 y)
Compute vector add operation $x + y$ in round-towards-zero mode.

__device__ float *__fadd_rd*(float x, float y)
Add two floating-point values in round-down mode.

__device__ float *__fadd_rn*(float x, float y)
Add two floating-point values in round-to-nearest-even mode.

__device__ float *__fadd_ru*(float x, float y)
Add two floating-point values in round-up mode.

__device__ float *__fadd_rz*(float x, float y)
Add two floating-point values in round-towards-zero mode.

__device__ float *__fdiv_rd*(float x, float y)
Divide two floating-point values in round-down mode.

__device__ float *__fdiv_rn*(float x, float y)
Divide two floating-point values in round-to-nearest-even mode.

__device__ float *__fdiv_ru*(float x, float y)
Divide two floating-point values in round-up mode.

__device__ float *__fdiv_rz*(float x, float y)
Divide two floating-point values in round-towards-zero mode.

__device__ float *__fdividef*(float x, float y)
Calculate the fast approximate division of the input arguments.

__device__ float2 *__ffma2_rd*(float2 x, float2 y, float2 z)
Compute vector fused multiply-add operation $x \times y + z$ in round-down mode.

__device__ float2 *__ffma2_rn*(float2 x, float2 y, float2 z)
Compute vector fused multiply-add operation $x \times y + z$ in round-to-nearest-even mode.

__device__ float2 *__ffma2_ru*(float2 x, float2 y, float2 z)
Compute vector fused multiply-add operation $x \times y + z$ in round-up mode.

__device__ float2 *__ffma2_rz*(float2 x, float2 y, float2 z)
Compute vector fused multiply-add operation $x \times y + z$ in round-towards-zero mode.

__device__ float *__fmaf_ieee_rd*(float x, float y, float z)
Compute fused multiply-add operation in round-down mode, ignore `-ftz=true` compiler flag.

__device__ float *__fmaf_ieee_rn*(float x, float y, float z)
Compute fused multiply-add operation in round-to-nearest-even mode, ignore `-ftz=true` compiler flag.

__device__ float *__fmaf_ieee_ru*(float x, float y, float z)
Compute fused multiply-add operation in round-up mode, ignore `-ftz=true` compiler flag.

__device__ float *__fmaf_ieee_rz*(float x, float y, float z)
Compute fused multiply-add operation in round-towards-zero mode, ignore `-ftz=true` compiler flag.

__device__ float *__fmaf_rd*(float x, float y, float z)
Compute $x \times y + z$ as a single operation, in round-down mode.

__device__ float *__fmaf_rn*(float x, float y, float z)
Compute $x \times y + z$ as a single operation, in round-to-nearest-even mode.

__device__ float *__fmaf_ru*(float x, float y, float z)
Compute $x \times y + z$ as a single operation, in round-up mode.

__device__ float *__fmaf_rz*(float x, float y, float z)
Compute $x \times y + z$ as a single operation, in round-towards-zero mode.

__device__ float2 *__fmul2_rd*(float2 x, float2 y)
Compute vector multiply operation $x \times y$ in round-down mode.

__device__ float2 *__fmul2_rn*(float2 x, float2 y)
Compute vector multiply operation $x \times y$ in round-to-nearest-even mode.

__device__ float2 *__fmul2_ru*(float2 x, float2 y)
Compute vector multiply operation $x \times y$ in round-up mode.

__device__ float2 *__fmul2_rz*(float2 x, float2 y)
Compute vector multiply operation $x \times y$ in round-towards-zero mode.

__device__ float *__fmul_rd*(float x, float y)
Multiply two floating-point values in round-down mode.

__device__ float *__fmul_rn*(float x, float y)
Multiply two floating-point values in round-to-nearest-even mode.

__device__ float *__fmul_ru*(float x, float y)
Multiply two floating-point values in round-up mode.

__device__ float *__fmul_rz*(float x, float y)
Multiply two floating-point values in round-towards-zero mode.

__device__ float *__frcp_rd*(float x)
Compute $\frac{1}{x}$ in round-down mode.

__device__ float *__frcp_rn*(float x)
Compute $\frac{1}{x}$ in round-to-nearest-even mode.

__device__ float *__frcp_ru*(float x)
Compute $\frac{1}{x}$ in round-up mode.

__device__ float *__frcp_rz*(float x)
Compute $\frac{1}{x}$ in round-towards-zero mode.

__device__ float *__frsqrt_rn*(float x)
Compute $1/\sqrt{x}$ in round-to-nearest-even mode.

__device__ float *__fsqrt_rd*(float x)
Compute $\sqrt{x}$ in round-down mode.

__device__ float *__fsqrt_rn*(float x)
Compute $\sqrt{x}$ in round-to-nearest-even mode.

__device__ float *__fsqrt_ru*(float x)
Compute $\sqrt{x}$ in round-up mode.

__device__ float *__fsqrt_rz*(float x)
Compute $\sqrt{x}$ in round-towards-zero mode.

__device__ float *__fsub_rd*(float x, float y)
Subtract two floating-point values in round-down mode.

__device__ float *__fsub_rn*(float x, float y)
Subtract two floating-point values in round-to-nearest-even mode.

__device__ float *__fsub_ru*(float x, float y)
Subtract two floating-point values in round-up mode.

__device__ float *__fsub_rz*(float x, float y)
Subtract two floating-point values in round-towards-zero mode.

__device__ float *__log10f*(float x)
Calculate the fast approximate base 10 logarithm of the input argument.

__device__ float *__log2f*(float x)
Calculate the fast approximate base 2 logarithm of the input argument.

__device__ float *__logf*(float x)
Calculate the fast approximate base e logarithm of the input argument.

__device__ float *__powf*(float x, float y)
Calculate the fast approximate of x^y .

__device__ float *__saturatef*(float x)
Clamp the input argument to [+0.0, 1.0].

__device__ void *__sincosf*(float x, float *sptr, float *cptr)
Calculate the fast approximate of sine and cosine of the first input argument.

__device__ float *__sinf*(float x)
Calculate the fast approximate sine of the input argument.

__device__ float *__tanf*(float x)
Calculate the fast approximate tangent of the input argument.

__device__ float *__tanhf*(float x)
Calculate the fast approximate hyperbolic tangent of the input argument.

7.1. Functions

__device__ float **__cosf**(float x)

Calculate the fast approximate cosine of the input argument.

Calculate the fast approximate cosine of the input argument x, measured in radians.

See also:

cosf() for further special case behavior specification.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Returns

Returns the approximate cosine of x.

__device__ float **__exp10f**(float x)

Calculate the fast approximate base 10 exponential of the input argument.

Calculate the fast approximate base 10 exponential of the input argument x, 10^x.

See also:

exp10f() for further special case behavior specification.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Returns

Returns an approximation to 10^x.

__device__ float **__expf**(float x)

Calculate the fast approximate base e exponential of the input argument.

Calculate the fast approximate base e exponential of the input argument x, e^x.

See also:

expf() for further special case behavior specification.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Returns

Returns an approximation to e^x.

__device__ float2 **__fadd2_rd**(float2 x, float2 y)

Compute vector add operation $x + y$ in round-down mode.

Numeric behavior per component is the same as *__fadd_rd()*.

Note: This intrinsic requires compute capability >= 10.0.

Note: The vector variants may not always provide better performance.

__device__ float2 **__fadd2_rn**(float2 x, float2 y)

Compute vector add operation $x + y$ in round-to-nearest-even mode.

Numeric behavior per component is the same as *__fadd_rn()*.

Note: This intrinsic requires compute capability >= 10.0.

Note: The vector variants may not always provide better performance.

__device__ float2 **__fadd2_ru**(float2 x, float2 y)

Compute vector add operation $x + y$ in round-up mode.

Numeric behavior per component is the same as *__fadd_ru()*.

Note: This intrinsic requires compute capability >= 10.0.

Note: The vector variants may not always provide better performance.

__device__ float2 **__fadd2_rz**(float2 x, float2 y)

Compute vector add operation $x + y$ in round-towards-zero mode.

Numeric behavior per component is the same as *__fadd_rz()*.

Note: This intrinsic requires compute capability >= 10.0.

Note: The vector variants may not always provide better performance.

__device__ float **__fadd_rd**(float x, float y)

Add two floating-point values in round-down mode.

Compute the sum of x and y in round-down (to negative infinity) mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns

Returns x + y.

- __fadd_rd(x, y) is equivalent to __fadd_rd(y, x).
- __fadd_rd(x, $\pm\infty$) returns $\pm\infty$ for finite x.
- __fadd_rd($\pm\infty$, $\pm\infty$) returns $\pm\infty$.
- __fadd_rd($\pm\infty$, $\mp\infty$) returns NaN.
- __fadd_rd(± 0, ± 0) returns ± 0.
- __fadd_rd(x, -x) returns -0 for finite x, including ± 0.
- If either argument is NaN, NaN is returned.

__device__ float **__fadd_rn**(float x, float y)

Add two floating-point values in round-to-nearest-even mode.

Compute the sum of x and y in round-to-nearest-even rounding mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns

Returns x + y.

- __fadd_rn(x, y) is equivalent to __fadd_rn(y, x).
- __fadd_rn(x, $\pm\infty$) returns $\pm\infty$ for finite x.
- __fadd_rn($\pm\infty$, $\pm\infty$) returns $\pm\infty$.
- __fadd_rn($\pm\infty$, $\mp\infty$) returns NaN.
- __fadd_rn(± 0, ± 0) returns ± 0.
- __fadd_rn(x, -x) returns $+0$ for finite x, including ± 0.
- If either argument is NaN, NaN is returned.

__device__ float **__fadd_ru**(float x, float y)

Add two floating-point values in round-up mode.

Compute the sum of x and y in round-up (to positive infinity) mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns

Returns x + y.

- __fadd_ru(x, y) is equivalent to __fadd_ru(y, x).
- __fadd_ru(x, $\pm\infty$) returns $\pm\infty$ for finite x.
- __fadd_ru($\pm\infty$, $\pm\infty$) returns $\pm\infty$.
- __fadd_ru($\pm\infty$, $\mp\infty$) returns NaN.
- __fadd_ru(± 0, ± 0) returns ± 0.
- __fadd_ru(x, -x) returns $+0$ for finite x, including ± 0.
- If either argument is NaN, NaN is returned.

__device__ float **__fadd_rz**(float x, float y)

Add two floating-point values in round-towards-zero mode.

Compute the sum of x and y in round-towards-zero mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns

Returns x + y.

- __fadd_rz(x, y) is equivalent to __fadd_rz(y, x).
- __fadd_rz(x, $\pm\infty$) returns $\pm\infty$ for finite x.
- __fadd_rz($\pm\infty$, $\pm\infty$) returns $\pm\infty$.
- __fadd_rz($\pm\infty$, $\mp\infty$) returns NaN.
- __fadd_rz(± 0, ± 0) returns ± 0.
- __fadd_rz(x, -x) returns $+0$ for finite x, including ± 0.
- If either argument is NaN, NaN is returned.

__device__ float **__fdiv_rd**(float x, float y)

Divide two floating-point values in round-down mode.

Divide two floating-point values x by y in round-down (to negative infinity) mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Returns

Returns x / y.

- sign of the quotient x / y is XOR of the signs of x and y when neither inputs nor result are NaN.
- __fdiv_rd(± 0, ± 0) returns NaN.
- __fdiv_rd($\pm\infty$, $\pm\infty$) returns NaN.
- __fdiv_rd(x, $\pm\infty$) returns 0 of appropriate sign for finite x.
- __fdiv_rd($\pm\infty$, y) returns ∞ of appropriate sign for finite y.
- __fdiv_rd(x, ± 0) returns ∞ of appropriate sign for x $\neq 0$.
- __fdiv_rd(± 0, y) returns 0 of appropriate sign for y $\neq 0$.
- If either argument is NaN, NaN is returned.

__device__ float **__fdiv_rn**(float x, float y)

Divide two floating-point values in round-to-nearest-even mode.

Divide two floating-point values x by y in round-to-nearest-even mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Returns

Returns x / y.

- sign of the quotient x / y is XOR of the signs of x and y when neither inputs nor result are NaN.
- __fdiv_rn(± 0, ± 0) returns NaN.
- __fdiv_rn($\pm\infty$, $\pm\infty$) returns NaN.
- __fdiv_rn(x, $\pm\infty$) returns 0 of appropriate sign for finite x.
- __fdiv_rn($\pm\infty$, y) returns ∞ of appropriate sign for finite y.
- __fdiv_rn(x, ± 0) returns ∞ of appropriate sign for x $\neq 0$.
- __fdiv_rn(± 0, y) returns 0 of appropriate sign for y $\neq 0$.
- If either argument is NaN, NaN is returned.

__device__ float **__fdiv_ru**(float x, float y)

Divide two floating-point values in round-up mode.

Divide two floating-point values x by y in round-up (to positive infinity) mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Returns

Returns x / y.

- sign of the quotient x / y is XOR of the signs of x and y when neither inputs nor result are NaN.
- __fdiv_ru(± 0, ± 0) returns NaN.
- __fdiv_ru($\pm\infty$, $\pm\infty$) returns NaN.
- __fdiv_ru(x, $\pm\infty$) returns 0 of appropriate sign for finite x.
- __fdiv_ru($\pm\infty$, y) returns ∞ of appropriate sign for finite y.
- __fdiv_ru(x, ± 0) returns ∞ of appropriate sign for x $\neq 0$.
- __fdiv_ru(± 0, y) returns 0 of appropriate sign for y $\neq 0$.
- If either argument is NaN, NaN is returned.

__device__ float **__fdiv_rz**(float x, float y)

Divide two floating-point values in round-towards-zero mode.

Divide two floating-point values x by y in round-towards-zero mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Returns

Returns x / y.

- sign of the quotient x / y is XOR of the signs of x and y when neither inputs nor result are NaN.
- __fdiv_rz(± 0, ± 0) returns NaN.
- __fdiv_rz($\pm\infty$, $\pm\infty$) returns NaN.
- __fdiv_rz(x, $\pm\infty$) returns 0 of appropriate sign for finite x.
- __fdiv_rz($\pm\infty$, y) returns ∞ of appropriate sign for finite y.
- __fdiv_rz(x, ± 0) returns ∞ of appropriate sign for x $\neq 0$.
- __fdiv_rz(± 0, y) returns 0 of appropriate sign for y $\neq 0$.
- If either argument is NaN, NaN is returned.

__device__ float **__fdividef**(float x, float y)

Calculate the fast approximate division of the input arguments.

Calculate the fast approximate division of x by y.

See also:

__fdiv_rn() for further special case behavior specification.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Returns

Returns x / y.

- __fdividef(∞ , y) returns NaN for $2^{126} < |y| < 2^{128}$.
- __fdividef(x, y) returns 0 for $2^{126} < |y| < 2^{128}$ and finite x.

__device__ float2 **__ffma2_rd**(float2 x, float2 y, float2 z)

Compute vector fused multiply-add operation $x \times y + z$ in round-down mode.

Numeric behavior per component is the same as *__fmaf_rd()*.

Note: This intrinsic requires compute capability >= 10.0.

Note: The vector variants may not always provide better performance.

__device__ float2 **__ffma2_rn**(float2 x, float2 y, float2 z)

Compute vector fused multiply-add operation $x \times y + z$ in round-to-nearest-even mode.

Numeric behavior per component is the same as *__fmaf_rn()*.

Note: This intrinsic requires compute capability >= 10.0.

Note: The vector variants may not always provide better performance.

__device__ float2 **__ffma2_ru**(float2 x, float2 y, float2 z)

Compute vector fused multiply-add operation $x \times y + z$ in round-up mode.

Numeric behavior per component is the same as *__fmaf_ru()*.

Note: This intrinsic requires compute capability >= 10.0.

Note: The vector variants may not always provide better performance.

__device__ float2 **__ffma2_rz**(float2 x, float2 y, float2 z)

Compute vector fused multiply-add operation $x \times y + z$ in round-towards-zero mode.

Numeric behavior per component is the same as *__fmaf_rz()*.

Note: This intrinsic requires compute capability >= 10.0.

Note: The vector variants may not always provide better performance.

__device__ float **__fmaf_ieee_rd**(float x, float y, float z)

Compute fused multiply-add operation in round-down mode, ignore `-ftz=true` compiler flag.

Behavior is the same as *__fmaf_rd*(x, y, z), the difference is in handling denormalized inputs and outputs: `-ftz` compiler flag has no effect.

__device__ float **__fmaf_ieee_rn**(float x, float y, float z)

Compute fused multiply-add operation in round-to-nearest-even mode, ignore `-ftz=true` compiler flag.

Behavior is the same as *__fmaf_rn*(x, y, z), the difference is in handling denormalized inputs and outputs: `-ftz` compiler flag has no effect.

__device__ float **__fmaf_ieee_ru**(float x, float y, float z)

Compute fused multiply-add operation in round-up mode, ignore `-ftz=true` compiler flag.

Behavior is the same as *__fmaf_ru*(x, y, z), the difference is in handling denormalized inputs and outputs: `-ftz` compiler flag has no effect.

__device__ float **__fmaf_ieee_rz**(float x, float y, float z)

Compute fused multiply-add operation in round-towards-zero mode, ignore `-ftz=true` compiler flag.

Behavior is the same as *__fmaf_rz*(x, y, z), the difference is in handling denormalized inputs and outputs: `-ftz` compiler flag has no effect.

__device__ float **__fmaf_rd**(float x, float y, float z)

Compute $x \times y + z$ as a single operation, in round-down mode.

Computes the value of $x \times y + z$ as a single ternary operation, rounding the result once in round-down (to negative infinity) mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Returns

Returns the rounded value of $x \times y + z$ as a single operation.

- __fmaf_rd($\pm\infty$, ± 0 , z) returns NaN.
- __fmaf_rd(± 0 , $\pm\infty$, z) returns NaN.
- __fmaf_rd(x, y, $-\infty$) returns NaN if $x \times y$ is an exact $+\infty$.
- __fmaf_rd(x, y, $+\infty$) returns NaN if $x \times y$ is an exact $-\infty$.
- __fmaf_rd(x, y, ± 0) returns ± 0 if $x \times y$ is exact ± 0.

- __fmaf_rd(x, y, ∓ 0) returns -0 if $x \times y$ is exact ± 0.
- __fmaf_rd(x, y, z) returns -0 if $x \times y + z$ is exactly zero and $z \neq 0$.
- If either argument is NaN, NaN is returned.

__device__ float **__fmaf_rn**(float x, float y, float z)

Compute $x \times y + z$ as a single operation, in round-to-nearest-even mode.

Computes the value of $x \times y + z$ as a single ternary operation, rounding the result once in round-to-nearest-even mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Returns

Returns the rounded value of $x \times y + z$ as a single operation.

- __fmaf_rn($\pm\infty$, ± 0 , z) returns NaN.
- __fmaf_rn(± 0 , $\pm\infty$, z) returns NaN.
- __fmaf_rn(x, y, $-\infty$) returns NaN if $x \times y$ is an exact $+\infty$.
- __fmaf_rn(x, y, $+\infty$) returns NaN if $x \times y$ is an exact $-\infty$.
- __fmaf_rn(x, y, ± 0) returns ± 0 if $x \times y$ is exact ± 0.
- __fmaf_rn(x, y, ∓ 0) returns $+0$ if $x \times y$ is exact ± 0.
- __fmaf_rn(x, y, z) returns $+0$ if $x \times y + z$ is exactly zero and $z \neq 0$.
- If either argument is NaN, NaN is returned.

__device__ float **__fmaf_ru**(float x, float y, float z)

Compute $x \times y + z$ as a single operation, in round-up mode.

Computes the value of $x \times y + z$ as a single ternary operation, rounding the result once in round-up (to positive infinity) mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Returns

Returns the rounded value of $x \times y + z$ as a single operation.

- __fmaf_ru($\pm\infty$, ± 0 , z) returns NaN.
- __fmaf_ru(± 0 , $\pm\infty$, z) returns NaN.
- __fmaf_ru(x, y, $-\infty$) returns NaN if $x \times y$ is an exact $+\infty$.
- __fmaf_ru(x, y, $+\infty$) returns NaN if $x \times y$ is an exact $-\infty$.
- __fmaf_ru(x, y, ± 0) returns ± 0 if $x \times y$ is exact ± 0.
- __fmaf_ru(x, y, ∓ 0) returns $+0$ if $x \times y$ is exact ± 0.
- __fmaf_ru(x, y, z) returns $+0$ if $x \times y + z$ is exactly zero and $z \neq 0$.

- If either argument is NaN, NaN is returned.

__device__ float **__fmaf_rz**(float x, float y, float z)

Compute $x \times y + z$ as a single operation, in round-towards-zero mode.

Computes the value of $x \times y + z$ as a single ternary operation, rounding the result once in round-towards-zero mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Returns

Returns the rounded value of $x \times y + z$ as a single operation.

- __fmaf_rz($\pm\infty$, ± 0 , z) returns NaN.
- __fmaf_rz(± 0 , $\pm\infty$, z) returns NaN.
- __fmaf_rz(x, y, $-\infty$) returns NaN if $x \times y$ is an exact $+\infty$.
- __fmaf_rz(x, y, $+\infty$) returns NaN if $x \times y$ is an exact $-\infty$.
- __fmaf_rz(x, y, ± 0) returns ± 0 if $x \times y$ is exact ± 0.
- __fmaf_rz(x, y, ∓ 0) returns $+0$ if $x \times y$ is exact ± 0.
- __fmaf_rz(x, y, z) returns $+0$ if $x \times y + z$ is exactly zero and $z \neq 0$.
- If either argument is NaN, NaN is returned.

__device__ float2 **__fmul2_rd**(float2 x, float2 y)

Compute vector multiply operation $x \times y$ in round-down mode.

Numeric behavior per component is the same as *__fmul_rd()*.

Note: This intrinsic requires compute capability >= 10.0.

Note: The vector variants may not always provide better performance.

__device__ float2 **__fmul2_rn**(float2 x, float2 y)

Compute vector multiply operation $x \times y$ in round-to-nearest-even mode.

Numeric behavior per component is the same as *__fmul_rn()*.

Note: This intrinsic requires compute capability >= 10.0.

Note: The vector variants may not always provide better performance.

__device__ float2 **__fmul2_ru**(float2 x, float2 y)

Compute vector multiply operation $x \times y$ in round-up mode.

Numeric behavior per component is the same as *__fmul_ru()*.

Note: This intrinsic requires compute capability >= 10.0.

Note: The vector variants may not always provide better performance.

__device__ float2 **__fmul2_rz**(float2 x, float2 y)

Compute vector multiply operation $x \times y$ in round-towards-zero mode.

Numeric behavior per component is the same as *__fmul_rz()*.

Note: This intrinsic requires compute capability >= 10.0.

Note: The vector variants may not always provide better performance.

__device__ float **__fmul_rd**(float x, float y)

Multiply two floating-point values in round-down mode.

Compute the product of x and y in round-down (to negative infinity) mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns

Returns x * y.

- sign of the product x * y is XOR of the signs of x and y when neither inputs nor result are NaN.
- __fmul_rd(x, y) is equivalent to __fmul_rd(y, x).
- __fmul_rd(x, $\pm\infty$) returns ∞ of appropriate sign for x $\neq 0$.
- __fmul_rd(± 0, $\pm\infty$) returns NaN.
- __fmul_rd(± 0, y) returns 0 of appropriate sign for finite y.
- If either argument is NaN, NaN is returned.

__device__ float **__fmul_rn**(float x, float y)

Multiply two floating-point values in round-to-nearest-even mode.

Compute the product of x and y in round-to-nearest-even mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns

Returns x * y.

- sign of the product x * y is XOR of the signs of x and y when neither inputs nor result are NaN.
- __fmul_rn(x, y) is equivalent to __fmul_rn(y, x).
- __fmul_rn(x, $\pm\infty$) returns ∞ of appropriate sign for x $\neq 0$.
- __fmul_rn(± 0, $\pm\infty$) returns NaN.
- __fmul_rn(± 0, y) returns 0 of appropriate sign for finite y.
- If either argument is NaN, NaN is returned.

__device__ float **__fmul_ru**(float x, float y)

Multiply two floating-point values in round-up mode.

Compute the product of x and y in round-up (to positive infinity) mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns

Returns x * y.

- sign of the product x * y is XOR of the signs of x and y when neither inputs nor result are NaN.
- __fmul_ru(x, y) is equivalent to __fmul_ru(y, x).
- __fmul_ru(x, $\pm\infty$) returns ∞ of appropriate sign for x $\neq 0$.
- __fmul_ru(± 0, $\pm\infty$) returns NaN.
- __fmul_ru(± 0, y) returns 0 of appropriate sign for finite y.
- If either argument is NaN, NaN is returned.

__device__ float **__fmul_rz**(float x, float y)

Multiply two floating-point values in round-towards-zero mode.

Compute the product of x and y in round-towards-zero mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns

Returns x * y.

- sign of the product x * y is XOR of the signs of x and y when neither inputs nor result are NaN.
- __fmul_rz(x, y) is equivalent to __fmul_rz(y, x).
- __fmul_rz(x, $\pm\infty$) returns ∞ of appropriate sign for x $\neq 0$.
- __fmul_rz(± 0, $\pm\infty$) returns NaN.
- __fmul_rz(± 0, y) returns 0 of appropriate sign for finite y.
- If either argument is NaN, NaN is returned.

__device__ float **__frcp_rd**(float x)

Compute $\frac{1}{x}$ in round-down mode.

Compute the reciprocal of x in round-down (to negative infinity) mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Returns

Returns $\frac{1}{x}$.

- __frcp_rd(± 0) returns $\pm\infty$.
- __frcp_rd($\pm\infty$) returns ± 0.
- __frcp_rd(NaN) returns NaN.

__device__ float **__frcp_rn**(float x)

Compute $\frac{1}{x}$ in round-to-nearest-even mode.

Compute the reciprocal of x in round-to-nearest-even mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Returns

Returns $\frac{1}{x}$.

- __frcp_rn(± 0) returns $\pm\infty$.
- __frcp_rn($\pm\infty$) returns ± 0.
- __frcp_rn(NaN) returns NaN.

__device__ float **__frcp_ru**(float x)

Compute $\frac{1}{x}$ in round-up mode.

Compute the reciprocal of x in round-up (to positive infinity) mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Returns

Returns $\frac{1}{x}$.

- __frcp_ru(± 0) returns $\pm \infty$.
- __frcp_ru($\pm \infty$) returns ± 0.
- __frcp_ru(NaN) returns NaN.

__device__ float **__frcp_rz**(float x)

Compute $\frac{1}{x}$ in round-towards-zero mode.

Compute the reciprocal of x in round-towards-zero mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Returns

Returns $\frac{1}{x}$.

- __frcp_rz(± 0) returns $\pm \infty$.
- __frcp_rz($\pm \infty$) returns ± 0.
- __frcp_rz(NaN) returns NaN.

__device__ float **__frsqrt_rn**(float x)

Compute $1/\sqrt{x}$ in round-to-nearest-even mode.

Compute the reciprocal square root of x in round-to-nearest-even mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Returns

Returns $1/\sqrt{x}$.

- __frsqrt_rn(± 0) returns $\pm \infty$.
- __frsqrt_rn($+\infty$) returns $+0$.
- __frsqrt_rn(x) returns NaN for x < 0.
- __frsqrt_rn(NaN) returns NaN.

__device__ float **__fsqrt_rd**(float x)

Compute $\sqrt{x}$ in round-down mode.

Compute the square root of x in round-down (to negative infinity) mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Returns

Returns $\sqrt{x}$.

- __fsqrt_rd(± 0) returns ± 0.
- __fsqrt_rd($+\infty$) returns $+\infty$.
- __fsqrt_rd(x) returns NaN for x < 0.
- __fsqrt_rd(NaN) returns NaN.

__device__ float **__fsqrt_rn**(float x)

Compute $\sqrt{x}$ in round-to-nearest-even mode.

Compute the square root of x in round-to-nearest-even mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Returns

Returns $\sqrt{x}$.

- __fsqrt_rn(± 0) returns ± 0.
- __fsqrt_rn($+\infty$) returns $+\infty$.
- __fsqrt_rn(x) returns NaN for x < 0.
- __fsqrt_rn(NaN) returns NaN.

__device__ float **__fsqrt_ru**(float x)

Compute $\sqrt{x}$ in round-up mode.

Compute the square root of x in round-up (to positive infinity) mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Returns

Returns $\sqrt{x}$.

- __fsqrt_ru(± 0) returns ± 0.
- __fsqrt_ru($+\infty$) returns $+\infty$.
- __fsqrt_ru(x) returns NaN for x < 0.
- __fsqrt_ru(NaN) returns NaN.

__device__ float **__fsqrt_rz**(float x)

Compute $\sqrt{x}$ in round-towards-zero mode.

Compute the square root of x in round-towards-zero mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Returns

Returns $\sqrt{x}$.

- __fsqrt_rz(± 0) returns ± 0.
- __fsqrt_rz($+\infty$) returns $+\infty$.
- __fsqrt_rz(x) returns NaN for x < 0.
- __fsqrt_rz(NaN) returns NaN.

__device__ float **__fsub_rd**(float x, float y)

Subtract two floating-point values in round-down mode.

Compute the difference of x and y in round-down (to negative infinity) mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns

Returns x - y.

- __fsub_rd($\pm\infty$, y) returns $\pm\infty$ for finite y.
- __fsub_rd(x, $\pm\infty$) returns $\mp\infty$ for finite x.
- __fsub_rd($\pm\infty$, $\pm\infty$) returns NaN.
- __fsub_rd($\pm\infty$, $\mp\infty$) returns $\pm\infty$.
- __fsub_rd(± 0, ∓ 0) returns ± 0.
- __fsub_rd(x, x) returns -0 for finite x, including ± 0.
- If either argument is NaN, NaN is returned.

__device__ float **__fsub_rn**(float x, float y)

Subtract two floating-point values in round-to-nearest-even mode.

Compute the difference of x and y in round-to-nearest-even rounding mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns

Returns x - y.

- __fsub_rn($\pm\infty$, y) returns $\pm\infty$ for finite y.
- __fsub_rn(x, $\pm\infty$) returns $\mp\infty$ for finite x.
- __fsub_rn($\pm\infty$, $\pm\infty$) returns NaN.
- __fsub_rn($\pm\infty$, $\mp\infty$) returns $\pm\infty$.
- __fsub_rn(± 0, ∓ 0) returns ± 0.
- __fsub_rn(x, x) returns $+0$ for finite x, including ± 0.
- If either argument is NaN, NaN is returned.

__device__ float **__fsub_ru**(float x, float y)

Subtract two floating-point values in round-up mode.

Compute the difference of x and y in round-up (to positive infinity) mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns

Returns x - y.

- __fsub_ru($\pm\infty$, y) returns $\pm\infty$ for finite y.
- __fsub_ru(x, $\pm\infty$) returns $\mp\infty$ for finite x.
- __fsub_ru($\pm\infty$, $\pm\infty$) returns NaN.
- __fsub_ru($\pm\infty$, $\mp\infty$) returns $\pm\infty$.
- __fsub_ru(± 0, ∓ 0) returns ± 0.
- __fsub_ru(x, x) returns $+0$ for finite x, including ± 0.
- If either argument is NaN, NaN is returned.

__device__ float **__fsub_rz**(float x, float y)

Subtract two floating-point values in round-towards-zero mode.

Compute the difference of x and y in round-towards-zero mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns

Returns x - y.

- __fsub_rz($\pm\infty$, y) returns $\pm\infty$ for finite y.
- __fsub_rz(x, $\pm\infty$) returns $\mp\infty$ for finite x.
- __fsub_rz($\pm\infty$, $\pm\infty$) returns NaN.
- __fsub_rz($\pm\infty$, $\mp\infty$) returns $\pm\infty$.
- __fsub_rz(± 0, ∓ 0) returns ± 0.
- __fsub_rz(x, x) returns $+0$ for finite x, including ± 0.
- If either argument is NaN, NaN is returned.

__device__ float **__log10f** (float x)

Calculate the fast approximate base 10 logarithm of the input argument.

Calculate the fast approximate base 10 logarithm of the input argument x.

See also:

log10f() for further special case behavior specification.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Returns

Returns an approximation to $\log_{10}(x)$.

__device__ float **__log2f** (float x)

Calculate the fast approximate base 2 logarithm of the input argument.

Calculate the fast approximate base 2 logarithm of the input argument x.

See also:

log2f() for further special case behavior specification.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Returns

Returns an approximation to $\log_2(x)$.

__device__ float **__logf** (float x)

Calculate the fast approximate base e logarithm of the input argument.

Calculate the fast approximate base e logarithm of the input argument x.

See also:

logf() for further special case behavior specification.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Returns

Returns an approximation to $\log_e(x)$.

__device__ float **__powf** (float x, float y)

Calculate the fast approximate of x^y.

Calculate the fast approximate of x, the first input argument, raised to the power of y, the second input argument, x^y.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Returns

Returns an approximation to x^y.

__device__ float **__saturatef** (float x)

Clamp the input argument to [+0.0, 1.0].

Clamp the input argument x to be within the interval [+0.0, 1.0].

Returns

- __saturatef(x) returns +0 if $x \le 0$.
- __saturatef(x) returns 1 if $x \ge 1$.
- __saturatef(x) returns x if $0 < x < 1$.
- __saturatef(NaN) returns +0.

__device__ void **__sincosf** (float x, float *sptr, float *cptr)

Calculate the fast approximate of sine and cosine of the first input argument.

Calculate the fast approximate of sine and cosine of the first input argument x (measured in radians). The results for sine and cosine are written into the second argument, `sptr`, and, respectively, third argument, `cptr`.

See also:

__sinf() and *__cosf()*.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: Denorm input/output is flushed to sign preserving 0.0.

__device__ float **__sinf**(float x)

Calculate the fast approximate sine of the input argument.

Calculate the fast approximate sine of the input argument x, measured in radians.

See also:

sinf() for further special case behavior specification.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: Output in the denormal range is flushed to sign preserving 0.0.

Returns

Returns the approximate sine of x.

__device__ float **__tanf**(float x)

Calculate the fast approximate tangent of the input argument.

Calculate the fast approximate tangent of the input argument x, measured in radians.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: The result is computed as the fast divide of *__sinf()* by *__cosf()*. Denormal output is flushed to sign-preserving 0.0.

Returns

Returns the approximate tangent of x.

__device__ float **__tanhf**(float x)

Calculate the fast approximate hyperbolic tangent of the input argument.

Calculate the fast approximate hyperbolic tangent of the input argument x, measured in radians.

See also:

tanhf() for further special case behavior specification.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Returns

Returns the approximate hyperbolic tangent of x.

Chapter 8. Double Precision Mathematical Functions

This section describes double precision mathematical functions.

To use these functions, you do not need to include any additional header file in your program.

Functions

__device__ double *acos*(double x)
Calculate the arc cosine of the input argument.

__device__ double *acosh*(double x)
Calculate the nonnegative inverse hyperbolic cosine of the input argument.

__device__ double *asin*(double x)
Calculate the arc sine of the input argument.

__device__ double *asinh*(double x)
Calculate the inverse hyperbolic sine of the input argument.

__device__ double *atan*(double x)
Calculate the arc tangent of the input argument.

__device__ double *atan2*(double y, double x)
Calculate the arc tangent of the ratio of first and second input arguments.

__device__ double *atanh*(double x)
Calculate the inverse hyperbolic tangent of the input argument.

__device__ double *cbrt*(double x)
Calculate the cube root of the input argument.

__device__ double *ceil*(double x)
Calculate ceiling of the input argument.

__device__ double *copysign*(double x, double y)
Create value with given magnitude, copying sign of second value.

__device__ double *cos*(double x)
Calculate the cosine of the input argument.

__device__ double *cosh*(double x)
Calculate the hyperbolic cosine of the input argument.

__device__ double *cospi*(double x)
Calculate the cosine of the input argument $\times \pi$.

__device__ double *cyl_bessel_i0*(double x)
Calculate the value of the regular modified cylindrical Bessel function of order 0 for the input argument.

__device__ double *cyl_bessel_i1*(double x)
Calculate the value of the regular modified cylindrical Bessel function of order 1 for the input argument.

__device__ double *erf*(double x)
Calculate the error function of the input argument.

__device__ double *erfc*(double x)
Calculate the complementary error function of the input argument.

__device__ double *erfcinv*(double x)
Calculate the inverse complementary error function of the input argument.

__device__ double *erfcx*(double x)
Calculate the scaled complementary error function of the input argument.

__device__ double *erfinv*(double x)
Calculate the inverse error function of the input argument.

__device__ double *exp*(double x)
Calculate the base e exponential of the input argument.

__device__ double *exp10*(double x)
Calculate the base 10 exponential of the input argument.

__device__ double *exp2*(double x)
Calculate the base 2 exponential of the input argument.

__device__ double *expm1*(double x)
Calculate the base e exponential of the input argument, minus 1.

__device__ double *fabs*(double x)
Calculate the absolute value of the input argument.

__device__ double *fdim*(double x, double y)
Compute the positive difference between x and y .

__device__ double *floor*(double x)
Calculate the largest integer less than or equal to x .

__device__ double *fma*(double x, double y, double z)
Compute $x \times y + z$ as a single operation.

__device__ double *fmax*(double, double)
Determine the maximum numeric value of the arguments.

__device__ double *fmin*(double x, double y)
Determine the minimum numeric value of the arguments.

__device__ double *fmod*(double x, double y)
Calculate the double-precision floating-point remainder of x / y .

__device__ double *frexp*(double x, int *nptr)
Extract mantissa and exponent of a floating-point value.

__device__ double *hypot*(double x, double y)
Calculate the square root of the sum of squares of two arguments.

__device__ int *ilogb*(double x)
Compute the unbiased integer exponent of the argument.

__device__ __RETURN_TYPE *isfinite*(double a)
Determine whether argument is finite.

__device__ __RETURN_TYPE *isinf*(double a)
Determine whether argument is infinite.

__device__ __RETURN_TYPE *isnan*(double a)
Determine whether argument is a NaN.

__device__ double *j0*(double x)
Calculate the value of the Bessel function of the first kind of order 0 for the input argument.

__device__ double *j1*(double x)
Calculate the value of the Bessel function of the first kind of order 1 for the input argument.

__device__ double *jn*(int n, double x)
Calculate the value of the Bessel function of the first kind of order n for the input argument.

__device__ double *ldexp*(double x, int exp)
Calculate the value of $x \cdot 2^{exp}$.

__device__ double *lgamma*(double x)
Calculate the natural logarithm of the absolute value of the gamma function of the input argument.

__device__ long long int *llrint*(double x)
Round input to nearest integer value.

__device__ long long int *llround*(double x)
Round to nearest integer value.

__device__ double *log*(double x)
Calculate the base e logarithm of the input argument.

__device__ double *log10*(double x)
Calculate the base 10 logarithm of the input argument.

__device__ double *log1p*(double x)
Calculate the value of $\log_e(1 + x)$.

__device__ double *log2*(double x)
Calculate the base 2 logarithm of the input argument.

__device__ double *logb*(double x)
Calculate the floating-point representation of the exponent of the input argument.

__device__ long int *lrint*(double x)
Round input to nearest integer value.

__device__ long int *lround*(double x)
Round to nearest integer value.

__device__ double *max*(const float a, const double b)
Calculate the maximum value of the input `float` and `double` arguments.

__device__ double *max*(const double a, const float b)
Calculate the maximum value of the input `double` and `float` arguments.

__device__ double *max*(const double a, const double b)
Calculate the maximum value of the input `float` arguments.

__device__ double *min*(const float a, const double b)
Calculate the minimum value of the input `float` and `double` arguments.

__device__ double *min*(const double a, const double b)
Calculate the minimum value of the input `float` arguments.

__device__ double *min*(const double a, const float b)
Calculate the minimum value of the input `double` and `float` arguments.

__device__ double *modf*(double x, double *iptr)
Break down the input argument into fractional and integral parts.

__device__ double *nan*(const char *tagp)
Returns "Not a Number" value.

__device__ double *nearbyint*(double x)
Round the input argument to the nearest integer.

__device__ double *nextafter*(double x, double y)
Return next representable double-precision floating-point value after argument x in the direction of y .

__device__ double *norm*(int dim, double const *p)
Calculate the square root of the sum of squares of any number of coordinates.

__device__ double *norm3d*(double a, double b, double c)
Calculate the square root of the sum of squares of three coordinates of the argument.

__device__ double *norm4d*(double a, double b, double c, double d)
Calculate the square root of the sum of squares of four coordinates of the argument.

__device__ double *normcdf*(double x)
Calculate the standard normal cumulative distribution function.

__device__ double *normcdfinv*(double x)
Calculate the inverse of the standard normal cumulative distribution function.

__device__ double *pow*(double x, double y)
Calculate the value of first argument to the power of second argument.

__device__ double *rcbrt*(double x)
Calculate reciprocal cube root function.

__device__ double *remainder*(double x, double y)
Compute double-precision floating-point remainder.

__device__ double *remquo*(double x, double y, int *quo)
Compute double-precision floating-point remainder and part of quotient.

__device__ double *rhypot*(double x, double y)
Calculate one over the square root of the sum of squares of two arguments.

__device__ double *rint*(double x)
Round to nearest integer value in floating-point.

__device__ double *rnorm*(int dim, double const *p)
Calculate the reciprocal of square root of the sum of squares of any number of coordinates.

__device__ double *rnorm3d*(double a, double b, double c)
Calculate one over the square root of the sum of squares of three coordinates.

__device__ double *rnorm4d*(double a, double b, double c, double d)
Calculate one over the square root of the sum of squares of four coordinates.

__device__ double *round*(double x)
Round to nearest integer value in floating-point.

__device__ double *rsqrt*(double x)
Calculate the reciprocal of the square root of the input argument.

__device__ double *scalbln*(double x, long int n)
Scale floating-point input by integer power of two.

__device__ double *scalbn*(double x, int n)
Scale floating-point input by integer power of two.

__device__ __RETURN_TYPE *signbit*(double a)
Return the sign bit of the input.

__device__ double *sin*(double x)
Calculate the sine of the input argument.

__device__ void *sincos*(double x, double *sptr, double *cptr)
Calculate the sine and cosine of the first input argument.

__device__ void *sincospi*(double x, double *sptr, double *cptr)
Calculate the sine and cosine of the first input argument $\times\pi$.

__device__ double *sinh*(double x)
Calculate the hyperbolic sine of the input argument.

__device__ double *sinpi*(double x)
Calculate the sine of the input argument $\times\pi$.

__device__ double *sqrt*(double x)
Calculate the square root of the input argument.

__device__ double *tan*(double x)
Calculate the tangent of the input argument.

__device__ double *tanh*(double x)
Calculate the hyperbolic tangent of the input argument.

__device__ double *tgamma*(double x)
Calculate the gamma function of the input argument.

__device__ double *trunc*(double x)
Truncate input argument to the integral part.

__device__ double *y0*(double x)
Calculate the value of the Bessel function of the second kind of order 0 for the input argument.

__device__ double *y1*(double x)
Calculate the value of the Bessel function of the second kind of order 1 for the input argument.

__device__ double *yn*(int n, double x)
Calculate the value of the Bessel function of the second kind of order n for the input argument.

8.1. Functions

__device__ double **acos**(double x)

Calculate the arc cosine of the input argument.

Calculate the principal value of the arc cosine of the input argument x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

Result will be in radians, in the interval [0, π] for x inside [-1, +1].

- acos(1) returns +0.
- acos(x) returns NaN for x outside [-1, +1].
- acos(NaN) returns NaN.

__device__ double **acosh**(double x)

Calculate the nonnegative inverse hyperbolic cosine of the input argument.

Calculate the nonnegative inverse hyperbolic cosine of the input argument x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

Result will be in the interval [0, $+\infty$].

- acosh(1) returns 0.
- acosh(x) returns NaN for x in the interval [$-\infty$, 1).
- acosh($+\infty$) returns $+\infty$.
- acosh(NaN) returns NaN.

__device__ double **asin**(double x)

Calculate the arc sine of the input argument.

Calculate the principal value of the arc sine of the input argument x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

Result will be in radians, in the interval [- π /2, + π /2] for x inside [-1, +1].

- asin(± 0) returns ± 0.
- asin(x) returns NaN for x outside [-1, +1].
- asin(NaN) returns NaN.

__device__ double **asinh**(double x)

Calculate the inverse hyperbolic sine of the input argument.

Calculate the inverse hyperbolic sine of the input argument x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- asinh(± 0) returns ± 0.
- asinh($\pm \infty$) returns $\pm \infty$.
- asinh(NaN) returns NaN.

__device__ double **atan**(double x)

Calculate the arc tangent of the input argument.

Calculate the principal value of the arc tangent of the input argument x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

Result will be in radians, in the interval [- π /2, + π /2].

- atan(± 0) returns ± 0.
- atan($\pm \infty$) returns $\pm \pi$ /2.
- atan(NaN) returns NaN.

__device__ double **atan2**(double y, double x)

Calculate the arc tangent of the ratio of first and second input arguments.

Calculate the principal value of the arc tangent of the ratio of first and second input arguments y / x. The quadrant of the result is determined by the signs of inputs y and x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

Result will be in radians, in the interval [- π , + π].

- atan2(± 0 , -0) returns $\pm \pi$.
- atan2(± 0 , +0) returns ± 0.
- atan2(± 0 , x) returns $\pm \pi$ for x < 0.
- atan2(± 0 , x) returns ± 0 for x > 0.
- atan2(y, ± 0) returns $-\pi$ /2 for y < 0.

- atan2(y, ± 0) returns π /2 for y > 0.
- atan2($\pm y$, $-\infty$) returns $\pm \pi$ for finite y > 0.
- atan2($\pm y$, $+\infty$) returns ± 0 for finite y > 0.
- atan2($\pm \infty$, x) returns $\pm \pi$ /2 for finite x.
- atan2($\pm \infty$, $-\infty$) returns $\pm 3\pi$ /4.
- atan2($\pm \infty$, $+\infty$) returns $\pm \pi$ /4.
- If either argument is NaN, NaN is returned.

__device__ double **atanh**(double x)

Calculate the inverse hyperbolic tangent of the input argument.

Calculate the inverse hyperbolic tangent of the input argument x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- atanh(± 0) returns ± 0.
- atanh(± 1) returns $\pm \infty$.
- atanh(x) returns NaN for x outside interval [-1, 1].
- atanh(NaN) returns NaN.

__device__ double **cbrt**(double x)

Calculate the cube root of the input argument.

Calculate the cube root of x, $x^{1/3}$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

Returns $x^{1/3}$.

- cbrt(± 0) returns ± 0.
- cbrt($\pm \infty$) returns $\pm \infty$.
- cbrt(NaN) returns NaN.

__device__ double **ceil**(double x)

Calculate ceiling of the input argument.

Compute the smallest integer value not less than x.

Returns

Returns $\lceil x \rceil$ expressed as a floating-point number.

- ceil(± 0) returns ± 0.

- ceil($\pm\infty$) returns $\pm\infty$.
- ceil(NaN) returns NaN.

__device__ double **copysign**(double x, double y)

Create value with given magnitude, copying sign of second value.

Create a floating-point value with the magnitude x and the sign of y.

Returns

- a value with the magnitude of x and the sign of y.
- copysign(NaN, y) returns a NaN with the sign of y.

__device__ double **cos**(double x)

Calculate the cosine of the input argument.

Calculate the cosine of the input argument x (measured in radians).

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- cos(± 0) returns 1.
- cos($\pm\infty$) returns NaN.
- cos(NaN) returns NaN.

__device__ double **cosh**(double x)

Calculate the hyperbolic cosine of the input argument.

Calculate the hyperbolic cosine of the input argument x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- cosh(± 0) returns 1.
- cosh($\pm\infty$) returns $+\infty$.
- cosh(NaN) returns NaN.

__device__ double **cospi**(double x)

Calculate the cosine of the input argument $\times\pi$.

Calculate the cosine of x $\times\pi$ (measured in radians), where x is the input argument.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- cospi(± 0) returns 1.
- cospi($\pm\infty$) returns NaN.
- cospi(NaN) returns NaN.

__device__ double **cyl_bessel_i0**(double x)

Calculate the value of the regular modified cylindrical Bessel function of order 0 for the input argument.

Calculate the value of the regular modified cylindrical Bessel function of order 0 for the input argument x, $I_0(x)$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

Returns the value of the regular modified cylindrical Bessel function of order 0.

- cyl_bessel_i0(± 0) returns +1.
- cyl_bessel_i0($\pm\infty$) returns $+\infty$.
- cyl_bessel_i0(NaN) returns NaN.

__device__ double **cyl_bessel_i1**(double x)

Calculate the value of the regular modified cylindrical Bessel function of order 1 for the input argument.

Calculate the value of the regular modified cylindrical Bessel function of order 1 for the input argument x, $I_1(x)$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

Returns the value of the regular modified cylindrical Bessel function of order 1.

- cyl_bessel_i1(± 0) returns ± 0.
- cyl_bessel_i1($\pm\infty$) returns $\pm\infty$.
- cyl_bessel_i1(NaN) returns NaN.

__device__ double **erf**(double x)

Calculate the error function of the input argument.

Calculate the value of the error function for the input argument x, $\frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- erf(± 0) returns ± 0.
- erf($\pm\infty$) returns ± 1.
- erf(NaN) returns NaN.

__device__ double **erfc**(double x)

Calculate the complementary error function of the input argument.

Calculate the complementary error function of the input argument x, 1 - erf(x).

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- erfc($-\infty$) returns 2.
- erfc($+\infty$) returns +0.
- erfc(NaN) returns NaN.

__device__ double **erfcinv**(double x)

Calculate the inverse complementary error function of the input argument.

Calculate the inverse complementary error function erfc^{-1} (x), of the input argument x in the interval [0, 2].

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- erfcinv(± 0) returns $+\infty$.
- erfcinv(2) returns $-\infty$.
- erfcinv(x) returns NaN for x outside [0, 2].
- erfcinv(NaN) returns NaN.

__device__ double **erfcx**(double x)

Calculate the scaled complementary error function of the input argument.

Calculate the scaled complementary error function of the input argument x, $e^{x^2} \cdot \text{erfc}(x)$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- erfcx($-\infty$) returns $+\infty$.
- erfcx($+\infty$) returns +0.
- erfcx(NaN) returns NaN.

__device__ double **erfinv**(double x)

Calculate the inverse error function of the input argument.

Calculate the inverse error function erf^{-1} (x), of the input argument x in the interval [-1, 1].

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- erfinv(± 0) returns ± 0.
- erfinv(1) returns $+\infty$.
- erfinv(-1) returns $-\infty$.
- erfinv(x) returns NaN for x outside [-1, +1].
- erfinv(NaN) returns NaN.

__device__ double **exp**(double x)

Calculate the base e exponential of the input argument.

Calculate e^x , the base e exponential of the input argument x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- exp(± 0) returns 1.
- exp($-\infty$) returns +0.
- exp($+\infty$) returns $+\infty$.
- exp(NaN) returns NaN.

__device__ double **exp10**(double x)

Calculate the base 10 exponential of the input argument.

Calculate 10^x , the base 10 exponential of the input argument x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- exp10(± 0) returns 1.
- exp10($-\infty$) returns +0.
- exp10($+\infty$) returns $+\infty$.
- exp10(NaN) returns NaN.

__device__ double **exp2** (double x)

Calculate the base 2 exponential of the input argument.

Calculate 2^x , the base 2 exponential of the input argument x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- exp2(± 0) returns 1.
- exp2($-\infty$) returns +0.
- exp2($+\infty$) returns $+\infty$.
- exp2(NaN) returns NaN.

__device__ double **expm1** (double x)

Calculate the base e exponential of the input argument, minus 1.

Calculate e^x -1, the base e exponential of the input argument x, minus 1.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- expm1(± 0) returns ± 0.
- expm1($-\infty$) returns -1.
- expm1($+\infty$) returns $+\infty$.
- expm1(NaN) returns NaN.

__device__ double **fabs** (double x)

Calculate the absolute value of the input argument.

Calculate the absolute value of the input argument x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

Returns the absolute value of the input argument.

- fabs($\pm\infty$) returns $+\infty$.
- fabs(± 0) returns +0.
- fabs(NaN) returns an unspecified NaN.

__device__ double **fdim**(double x, double y)

Compute the positive difference between x and y.

Compute the positive difference between x and y. The positive difference is x - y when x > y and +0 otherwise.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

Returns the positive difference between x and y.

- fdim(x, y) returns x - y if x > y.
- fdim(x, y) returns +0 if x $\leq$ y.
- If either argument is NaN, NaN is returned.

__device__ double **floor**(double x)

Calculate the largest integer less than or equal to x.

Calculates the largest integer value which is less than or equal to x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

Returns $\lfloor x \rfloor$ expressed as a floating-point number.

- floor($\pm\infty$) returns $\pm\infty$.
- floor(± 0) returns ± 0.
- floor(NaN) returns NaN.

__device__ double **fma**(double x, double y, double z)

Compute $x \times y + z$ as a single operation.

Compute the value of $x \times y + z$ as a single ternary operation. After computing the value to infinite precision, the value is rounded once using round-to-nearest, ties-to-even rounding mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

Returns the rounded value of $x \times y + z$ as a single operation.

- fma($\pm\infty$, ± 0 , z) returns NaN.
- fma(± 0 , $\pm\infty$, z) returns NaN.
- fma(x, y, $-\infty$) returns NaN if $x \times y$ is an exact $+\infty$.
- fma(x, y, $+\infty$) returns NaN if $x \times y$ is an exact $-\infty$.
- fma(x, y, ± 0) returns ± 0 if $x \times y$ is exact ± 0.
- fma(x, y, ∓ 0) returns $+0$ if $x \times y$ is exact ± 0.
- fma(x, y, z) returns $+0$ if $x \times y + z$ is exactly zero and $z \neq 0$.
- If either argument is NaN, NaN is returned.

__device__ double **fmax**(double, double)

Determine the maximum numeric value of the arguments.

Determines the maximum numeric value of the arguments x and y. Treats NaN arguments as missing data. If one argument is a NaN and the other is legitimate numeric value, the numeric value is chosen.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

Returns the maximum numeric values of the arguments x and y.

- If both arguments are NaN, returns NaN.
- If one argument is NaN, returns the numeric argument.

__device__ double **fmin**(double x, double y)

Determine the minimum numeric value of the arguments.

Determines the minimum numeric value of the arguments x and y. Treats NaN arguments as missing data. If one argument is a NaN and the other is legitimate numeric value, the numeric value is chosen.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

Returns the minimum numeric value of the arguments x and y.

- If both arguments are NaN, returns NaN.
- If one argument is NaN, returns the numeric argument.

__device__ double **fmod**(double x, double y)

Calculate the double-precision floating-point remainder of x / y.

Calculate the double-precision floating-point remainder of x / y. The floating-point remainder of the division operation x / y calculated by this function is exactly the value `x - n*y`, where n is x / y with its fractional part truncated. The computed value will have the same sign as x, and its magnitude will be less than the magnitude of y.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- Returns the floating-point remainder of x / y.
- fmod(± 0 , y) returns ± 0 if y is not zero.
- fmod(x, $\pm\infty$) returns x if x is finite.
- fmod(x, y) returns NaN if x is $\pm\infty$ or y is zero.
- If either argument is NaN, NaN is returned.

__device__ double **frexp**(double x, int *nptr)

Extract mantissa and exponent of a floating-point value.

Decompose the floating-point value `x` into a component `m` for the normalized fraction element and another term `n` for the exponent. The absolute value of `m` will be greater than or equal to 0.5 and less than 1.0 or it will be equal to 0; $x = m \cdot 2^n$. The integer exponent `n` will be stored in the location to which `nptr` points.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

Returns the fractional component `m`.

- frexp(± 0 , `nptr`) returns ± 0 and stores zero in the location pointed to by `nptr`.
- frexp($\pm\infty$, `nptr`) returns $\pm\infty$ and stores an unspecified value in the location to which `nptr` points.
- frexp(NaN, y) returns a NaN and stores an unspecified value in the location to which `nptr` points.

__device__ double **hypot**(double x, double y)

Calculate the square root of the sum of squares of two arguments.

Calculate the length of the hypotenuse of a right triangle whose two sides have lengths x and y without undue overflow or underflow.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

Returns the length of the hypotenuse $\sqrt{x^2 + y^2}$.

- hypot(x,y), hypot(y,x), and hypot(x, -y) are equivalent.
- hypot(x, ± 0) is equivalent to fabs(x).
- hypot($\pm\infty$,y) returns $+\infty$, even if y is a NaN.

- hypot(NaN, y) returns NaN, when y is not $\pm\infty$.

__device__ int **ilogb**(double x)

Compute the unbiased integer exponent of the argument.

Calculates the unbiased integer exponent of the input argument x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- If successful, returns the unbiased exponent of the argument.
- ilogb(± 0) returns `INT_MIN`.
- ilogb(NaN) returns `INT_MIN`.
- ilogb($\pm\infty$) returns `INT_MAX`.
- Note: above behavior does not take into account `FP_ILOGB0` nor `FP_ILOGBNAN`.

__device__ __RETURN_TYPE **isfinite**(double a)

Determine whether argument is finite.

Determine whether the floating-point value a is a finite value (zero, subnormal, or normal and not infinity or NaN).

Returns

- With Visual Studio 2013 host compiler: __RETURN_TYPE is 'bool'. Returns true if and only if a is a finite value.
- With other host compilers: __RETURN_TYPE is 'int'. Returns a nonzero value if and only if a is a finite value.

__device__ __RETURN_TYPE **isinf**(double a)

Determine whether argument is infinite.

Determine whether the floating-point value a is an infinite value (positive or negative).

Returns

- With Visual Studio 2013 host compiler: Returns true if and only if a is an infinite value.
- With other host compilers: Returns a nonzero value if and only if a is an infinite value.

__device__ __RETURN_TYPE **isnan**(double a)

Determine whether argument is a NaN.

Determine whether the floating-point value a is a NaN.

Returns

- With Visual Studio 2013 host compiler: __RETURN_TYPE is 'bool'. Returns true if and only if a is a NaN value.
- With other host compilers: __RETURN_TYPE is 'int'. Returns a nonzero value if and only if a is a NaN value.

__device__ double **j0**(double x)

Calculate the value of the Bessel function of the first kind of order 0 for the input argument.

Calculate the value of the Bessel function of the first kind of order 0 for the input argument x, $J_0(x)$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

Returns the value of the Bessel function of the first kind of order 0.

- j0($\pm\infty$) returns +0.
- j0(NaN) returns NaN.

__device__ double **j1**(double x)

Calculate the value of the Bessel function of the first kind of order 1 for the input argument.

Calculate the value of the Bessel function of the first kind of order 1 for the input argument x, $J_1(x)$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

Returns the value of the Bessel function of the first kind of order 1.

- j1(± 0) returns ± 0.
- j1($\pm\infty$) returns ± 0.
- j1(NaN) returns NaN.

__device__ double **jn**(int n, double x)

Calculate the value of the Bessel function of the first kind of order n for the input argument.

Calculate the value of the Bessel function of the first kind of order n for the input argument x, $J_n(x)$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

Returns the value of the Bessel function of the first kind of order n.

- jn(n, NaN) returns NaN.
- jn(n, x) returns NaN for n < 0.

- jn(n, $+\infty$) returns +0.

__device__ double **ldexp**(double x, int exp)

Calculate the value of $x \cdot 2^{exp}$.

Calculate the value of $x \cdot 2^{exp}$ of the input arguments x and exp.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- ldexp(x, exp) is equivalent to scalbn(x, exp).

__device__ double **lgamma**(double x)

Calculate the natural logarithm of the absolute value of the gamma function of the input argument.

Calculate the natural logarithm of the absolute value of the gamma function of the input argument x, namely the value of $\log_e |\Gamma(x)|$

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- lgamma(1) returns +0.
- lgamma(2) returns +0.
- lgamma(x) returns $+\infty$ if x $\leq$ 0 and x is an integer.
- lgamma($-\infty$) returns $+\infty$.
- lgamma($+\infty$) returns $+\infty$.
- lgamma(NaN) returns NaN.

__device__ long long int **llrint**(double x)

Round input to nearest integer value.

Round x to the nearest integer value, with halfway cases rounded to the nearest even integer value. If the result is outside the range of the return type, the behavior is undefined.

Returns

Returns rounded integer value.

__device__ long long int **llround**(double x)

Round to nearest integer value.

Round x to the nearest integer value, with halfway cases rounded away from zero. If the result is outside the range of the return type, the behavior is undefined.

Note: This function may be slower than alternate rounding methods. See *llrint()*.

Returns

Returns rounded integer value.

__device__ double **log**(double x)

Calculate the base e logarithm of the input argument.

Calculate the base e logarithm of the input argument x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- log(± 0) returns $-\infty$.
- log(1) returns +0.
- log(x) returns NaN for x < 0.
- log($+\infty$) returns $+\infty$.
- log(NaN) returns NaN.

__device__ double **log10**(double x)

Calculate the base 10 logarithm of the input argument.

Calculate the base 10 logarithm of the input argument x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- log10(± 0) returns $-\infty$.
- log10(1) returns +0.
- log10(x) returns NaN for x < 0.
- log10($+\infty$) returns $+\infty$.
- log10(NaN) returns NaN.

__device__ double **log1p**(double x)

Calculate the value of $\log_e(1+x)$.

Calculate the value of $\log_e(1+x)$ of the input argument x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- log1p(± 0) returns ± 0.
- log1p(-1) returns $-\infty$.
- log1p(x) returns NaN for x < -1.
- log1p($+\infty$) returns $+\infty$.
- log1p(NaN) returns NaN.

__device__ double **log2** (double x)

Calculate the base 2 logarithm of the input argument.

Calculate the base 2 logarithm of the input argument x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- log2(± 0) returns $-\infty$.
- log2(1) returns +0.
- log2(x) returns NaN for x < 0.
- log2($+\infty$) returns $+\infty$.
- log2(NaN) returns NaN.

__device__ double **logb** (double x)

Calculate the floating-point representation of the exponent of the input argument.

Calculate the floating-point representation of the exponent of the input argument x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- logb(± 0) returns $-\infty$.
- logb($\pm\infty$) returns $+\infty$.
- logb(NaN) returns NaN.

__device__ long int **lrint**(double x)

Round input to nearest integer value.

Round x to the nearest integer value, with halfway cases rounded to the nearest even integer value. If the result is outside the range of the return type, the behavior is undefined.

Returns

Returns rounded integer value.

__device__ long int **lround**(double x)

Round to nearest integer value.

Round x to the nearest integer value, with halfway cases rounded away from zero. If the result is outside the range of the return type, the behavior is undefined.

Note: This function may be slower than alternate rounding methods. See *lrint()*.

Returns

Returns rounded integer value.

__device__ double **max**(const float a, const double b)

Calculate the maximum value of the input `float` and `double` arguments.

Convert `float` argument a to `double`, followed by *fmax()*.

Note, this is different from `std::` specification

__device__ double **max**(const double a, const float b)

Calculate the maximum value of the input `double` and `float` arguments.

Convert `float` argument b to `double`, followed by *fmax()*.

Note, this is different from `std::` specification

__device__ double **max**(const double a, const double b)

Calculate the maximum value of the input `float` arguments.

Calculate the maximum value of the arguments a and b. Behavior is equivalent to *fmax()* function.

Note, this is different from `std::` specification

__device__ double **min**(const float a, const double b)

Calculate the minimum value of the input `float` and `double` arguments.

Convert `float` argument a to `double`, followed by *fmin()*.

Note, this is different from `std::` specification

__device__ double **min**(const double a, const double b)

Calculate the minimum value of the input `float` arguments.

Calculate the minimum value of the arguments a and b. Behavior is equivalent to *fmin()* function.

Note, this is different from `std::` specification

__device__ double **min**(const double a, const float b)

Calculate the minimum value of the input `double` and `float` arguments.

Convert `float` argument b to `double`, followed by *fmin()*.

Note, this is different from `std::` specification __device__ double **modf**(double x, double *iptr)

Break down the input argument into fractional and integral parts.

Break down the argument x into fractional and integral parts. The integral part is stored in the argument `iptr`. Fractional and integral parts are given the same sign as the argument x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- modf($\pm x$, `iptr`) returns a result with the same sign as x.
- modf($\pm\infty$, `iptr`) returns ± 0 and stores $\pm\infty$ in the object pointed to by `iptr`.
- modf(NaN, `iptr`) stores a NaN in the object pointed to by `iptr` and returns a NaN.

__device__ double **nan**(const char *tagp)

Returns "Not a Number" value.

Return a representation of a quiet NaN. Argument `tagp` selects one of the possible representations.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- nan(`tagp`) returns NaN.

__device__ double **nearbyint**(double x)

Round the input argument to the nearest integer.

Round argument x to an integer value in double precision floating-point format. Uses round to nearest rounding, with ties rounding to even.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- nearbyint(± 0) returns ± 0.
- nearbyint($\pm\infty$) returns $\pm\infty$.
- nearbyint(NaN) returns NaN.

__device__ double **nextafter**(double x, double y)

Return next representable double-precision floating-point value after argument x in the direction of y.

Calculate the next representable double-precision floating-point value following x in the direction of y. For example, if y is greater than x, *nextafter()* returns the smallest representable number greater than x

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- nextafter(x, y) = y if x equals y.
- nextafter(x, y) = NaN if either x or y are NaN.

__device__ double **norm**(int dim, double const *p)

Calculate the square root of the sum of squares of any number of coordinates.

Calculate the length of a vector p, dimension of which is passed as an argument `without` undue overflow or underflow.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

Returns the length of the dim-D vector $\sqrt{\sum_{i=0}^{dim-1} p_i^2}$.

- In the presence of an exactly infinite coordinate $+\infty$ is returned, even if there are NaNs.
- returns +0, when all coordinates are ± 0.
- returns NaN, when at least one of the coordinates is NaN and none are infinite.

__device__ double **norm3d**(double a, double b, double c)

Calculate the square root of the sum of squares of three coordinates of the argument.

Calculate the length of three dimensional vector in Euclidean space without undue overflow or underflow.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

Returns the length of 3D vector $\sqrt{a^2 + b^2 + c^2}$.

- In the presence of an exactly infinite coordinate $+\infty$ is returned, even if there are NaNs.
- returns +0, when all coordinates are ± 0.

- returns NaN, when at least one of the coordinates is NaN and none are infinite.

__device__ double **norm4d**(double a, double b, double c, double d)

Calculate the square root of the sum of squares of four coordinates of the argument.

Calculate the length of four dimensional vector in Euclidean space without undue overflow or underflow.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

Returns the length of 4D vector $\sqrt{a^2 + b^2 + c^2 + d^2}$.

- In the presence of an exactly infinite coordinate $+\infty$ is returned, even if there are NaNs.
- returns +0, when all coordinates are ± 0.
- returns NaN, when at least one of the coordinates is NaN and none are infinite.

__device__ double **normcdf**(double x)

Calculate the standard normal cumulative distribution function.

Calculate the cumulative distribution function of the standard normal distribution for input argument x, $\Phi(x)$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- normcdf($+\infty$) returns 1.
- normcdf($-\infty$) returns +0.
- normcdf(NaN) returns NaN.

__device__ double **normcdfinv**(double x)

Calculate the inverse of the standard normal cumulative distribution function.

Calculate the inverse of the standard normal cumulative distribution function for input argument x, $\Phi^{-1}(x)$. The function is defined for input values in the interval $(0, 1)$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- normcdfinv(± 0) returns $-\infty$.
- normcdfinv(1) returns $+\infty$.

- normcdfinv(x) returns NaN if x is not in the interval [0,1].
- normcdfinv(NaN) returns NaN.

__device__ double **pow**(double x, double y)

Calculate the value of first argument to the power of second argument.

Calculate the value of x to the power of y.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- pow(± 0 , y) returns $\pm\infty$ for y an odd integer less than 0.
- pow(± 0 , y) returns $+\infty$ for y less than 0 and not an odd integer.
- pow(± 0 , y) returns ± 0 for y an odd integer greater than 0.
- pow(± 0 , y) returns +0 for y > 0 and not an odd integer.
- pow(-1, $\pm\infty$) returns 1.
- pow(+1, y) returns 1 for any y, even a NaN.
- pow(x, ± 0) returns 1 for any x, even a NaN.
- pow(x, y) returns a NaN for finite x < 0 and finite non-integer y.
- pow(x, $-\infty$) returns $+\infty$ for $|x| < 1$.
- pow(x, $-\infty$) returns +0 for $|x| > 1$.
- pow(x, $+\infty$) returns +0 for $|x| < 1$.
- pow(x, $+\infty$) returns $+\infty$ for $|x| > 1$.
- pow($-\infty$, y) returns -0 for y an odd integer less than 0.
- pow($-\infty$, y) returns +0 for y < 0 and not an odd integer.
- pow($-\infty$, y) returns $-\infty$ for y an odd integer greater than 0.
- pow($-\infty$, y) returns $+\infty$ for y > 0 and not an odd integer.
- pow($+\infty$, y) returns +0 for y < 0.
- pow($+\infty$, y) returns $+\infty$ for y > 0.
- pow(x, y) returns NaN if either x or y or both are NaN and x $\neq$ +1 and y $\neq \pm 0$.

__device__ double **rcbrt**(double x)

Calculate reciprocal cube root function.

Calculate reciprocal cube root function of x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- rcbrt(± 0) returns $\pm\infty$.
- rcbrt($\pm\infty$) returns ± 0.
- rcbrt(NaN) returns NaN.

__device__ double **remainder**(double x, double y)

Compute double-precision floating-point remainder.

Compute double-precision floating-point remainder r of dividing x by y for nonzero y. Thus $r = x - ny$. The value n is the integer value nearest $\frac{x}{y}$. In the case when $|n - \frac{x}{y}| = \frac{1}{2}$, the even n value is chosen.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- remainder(x, ± 0) returns NaN.
- remainder($\pm\infty$, y) returns NaN.
- remainder(x, $\pm\infty$) returns x for finite x.
- If either argument is NaN, NaN is returned.

__device__ double **remquo**(double x, double y, int *quo)

Compute double-precision floating-point remainder and part of quotient.

Compute a double-precision floating-point remainder in the same way as the *remainder()* function. Argument quo returns part of quotient upon division of x by y. Value quo has the same sign as $\frac{x}{y}$ and may not be the exact quotient but agrees with the exact quotient in the low order 3 bits.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

Returns the remainder.

- remquo(x, ± 0 , quo) returns NaN and stores an unspecified value in the location to which quo points.
- remquo($\pm\infty$, y, quo) returns NaN and stores an unspecified value in the location to which quo points.
- remquo(x, y, quo) returns NaN and stores an unspecified value in the location to which quo points if either of x or y is NaN.
- remquo(x, $\pm\infty$, quo) returns x and stores zero in the location to which quo points for finite x.

__device__ double **rhypot**(double x, double y)

Calculate one over the square root of the sum of squares of two arguments.

Calculate one over the length of the hypotenuse of a right triangle whose two sides have lengths x and y without undue overflow or underflow.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

Returns one over the length of the hypotenuse $\frac{1}{\sqrt{x^2+y^2}}$.

- rhypot(x,y), rhypot(y,x), and rhypot(x, -y) are equivalent.
- rhypot($\pm\infty$,y) returns +0, even if y is a NaN.
- rhypot($\pm 0, \pm 0$) returns $+\infty$.
- rhypot(NaN, y) returns NaN, when y is not $\pm\infty$.

__device__ double **rint**(double x)

Round to nearest integer value in floating-point.

Round x to the nearest integer value in floating-point format, with halfway cases rounded to the nearest even integer value.

Returns

Returns rounded integer value.

- rint(± 0) returns ± 0.
- rint($\pm\infty$) returns $\pm\infty$.
- rint(NaN) returns NaN.

__device__ double **rnorm**(int dim, double const *p)

Calculate the reciprocal of square root of the sum of squares of any number of coordinates.

Calculates one over the length of vector p, dimension of which is passed as an argument, in Euclidean space without undue overflow or underflow.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

Returns one over the length of the vector $\frac{1}{\sqrt{\sum_{i=0}^{dim-1} p_i^2}}$.

- In the presence of an exactly infinite coordinate $+0$ is returned, even if there are NaNs.
- returns $+\infty$, when all coordinates are ± 0.
- returns NaN, when at least one of the coordinates is NaN and none are infinite.

__device__ double **rnorm3d**(double a, double b, double c)

Calculate one over the square root of the sum of squares of three coordinates.

Calculate one over the length of three dimensional vector in Euclidean space without undue overflow or underflow.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

Returns one over the length of the 3D vector $\frac{1}{\sqrt{a^2+b^2+c^2}}$.

- In the presence of an exactly infinite coordinate $+0$ is returned, even if there are NaNs.
- returns $+\infty$, when all coordinates are ± 0.
- returns NaN, when at least one of the coordinates is NaN and none are infinite.

__device__ double **rnorm4d**(double a, double b, double c, double d)

Calculate one over the square root of the sum of squares of four coordinates.

Calculate one over the length of four dimensional vector in Euclidean space without undue overflow or underflow.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

Returns one over the length of the 3D vector $\frac{1}{\sqrt{a^2+b^2+c^2+d^2}}$.

- In the presence of an exactly infinite coordinate $+0$ is returned, even if there are NaNs.
- returns $+\infty$, when all coordinates are ± 0.
- returns NaN, when at least one of the coordinates is NaN and none are infinite.

__device__ double **round**(double x)

Round to nearest integer value in floating-point.

Round x to the nearest integer value in floating-point format, with halfway cases rounded away from zero.

Note: This function may be slower than alternate rounding methods. See *rint()*.

Returns

Returns rounded integer value.

- round(± 0) returns ± 0.
- round($\pm\infty$) returns $\pm\infty$.
- round(NaN) returns NaN.

__device__ double **rsqrt**(double x)

Calculate the reciprocal of the square root of the input argument.

Calculate the reciprocal of the nonnegative square root of x, $1/\sqrt{x}$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

Returns $1/\sqrt{x}$.

- rsqrt($+\infty$) returns +0.
- rsqrt(± 0) returns $\pm\infty$.
- rsqrt(x) returns NaN if x is less than 0.
- rsqrt(NaN) returns NaN.

__device__ double **scalbln**(double x, long int n)

Scale floating-point input by integer power of two.

Scale x by 2^n by efficient manipulation of the floating-point exponent.

Returns

Returns x * 2^n.

- scalbln(± 0 , n) returns ± 0.
- scalbln(x, 0) returns x.
- scalbln($\pm\infty$, n) returns $\pm\infty$.
- scalbln(NaN, n) returns NaN.

__device__ double **scalbn**(double x, int n)

Scale floating-point input by integer power of two.

Scale x by 2^n by efficient manipulation of the floating-point exponent.

Returns

Returns x * 2^n.

- scalbn(± 0 , n) returns ± 0.
- scalbn(x, 0) returns x.
- scalbn($\pm\infty$, n) returns $\pm\infty$.
- scalbn(NaN, n) returns NaN.

__device__ __RETURN_TYPE **signbit**(double a)

Return the sign bit of the input.

Determine whether the floating-point value a is negative.

Returns

Reports the sign bit of all values including infinities, zeros, and NaNs.

- With Visual Studio 2013 host compiler: __RETURN_TYPE is 'bool'. Returns true if and only if a is negative.

- With other host compilers: __RETURN_TYPE is 'int'. Returns a nonzero value if and only if a is negative.

__device__ double **sin**(double x)

Calculate the sine of the input argument.

Calculate the sine of the input argument x (measured in radians).

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- sin(± 0) returns ± 0.
- sin($\pm\infty$) returns NaN.
- sin(NaN) returns NaN.

__device__ void **sincos**(double x, double *sptr, double *cptr)

Calculate the sine and cosine of the first input argument.

Calculate the sine and cosine of the first input argument x (measured in radians). The results for sine and cosine are written into the second argument, `sptr`, and, respectively, third argument, `cptr`.

See also:

sin() and *cos()*.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

__device__ void **sincospi**(double x, double *sptr, double *cptr)

Calculate the sine and cosine of the first input argument $\times \pi$.

Calculate the sine and cosine of the first input argument, x (measured in radians), $\times \pi$. The results for sine and cosine are written into the second argument, `sptr`, and, respectively, third argument, `cptr`.

See also:

sinpi() and *cospi()*.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

__device__ double **sinh**(double x)

Calculate the hyperbolic sine of the input argument.

Calculate the hyperbolic sine of the input argument x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- sinh(± 0) returns ± 0.
- sinh($\pm \infty$) returns $\pm \infty$.
- sinh(NaN) returns NaN.

__device__ double **sinpi**(double x)

Calculate the sine of the input argument $\times \pi$.

Calculate the sine of x $\times \pi$ (measured in radians), where x is the input argument.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- sinpi(± 0) returns ± 0.
- sinpi($\pm \infty$) returns NaN.
- sinpi(NaN) returns NaN.

__device__ double **sqrt**(double x)

Calculate the square root of the input argument.

Calculate the nonnegative square root of x, $\sqrt{x}$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

Returns $\sqrt{x}$.

- sqrt(± 0) returns ± 0.
- sqrt($+\infty$) returns $+\infty$.
- sqrt(x) returns NaN if x is less than 0.
- sqrt(NaN) returns NaN.

__device__ double **tan**(double x)

Calculate the tangent of the input argument.

Calculate the tangent of the input argument x (measured in radians).

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- tan(± 0) returns ± 0.
- tan($\pm\infty$) returns NaN.
- tan(NaN) returns NaN.

__device__ double **tanh**(double x)

Calculate the hyperbolic tangent of the input argument.

Calculate the hyperbolic tangent of the input argument x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- tanh(± 0) returns ± 0.
- tanh($\pm\infty$) returns ± 1.
- tanh(NaN) returns NaN.

__device__ double **tgamma**(double x)

Calculate the gamma function of the input argument.

Calculate the gamma function of the input argument x, namely the value of $\Gamma(x)$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- tgamma(± 0) returns $\pm\infty$.
- tgamma(x) returns NaN if x < 0 and x is an integer.
- tgamma($-\infty$) returns NaN.
- tgamma($+\infty$) returns $+\infty$.
- tgamma(NaN) returns NaN.

__device__ double **trunc**(double x)

Truncate input argument to the integral part.

Round x to the nearest integer value that does not exceed x in magnitude.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

Returns truncated integer value.

- trunc(± 0) returns ± 0.
- trunc($\pm\infty$) returns $\pm\infty$.
- trunc(NaN) returns NaN.

__device__ double **y0** (double x)

Calculate the value of the Bessel function of the second kind of order 0 for the input argument.

Calculate the value of the Bessel function of the second kind of order 0 for the input argument x, $Y_0(x)$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

Returns the value of the Bessel function of the second kind of order 0.

- y0(± 0) returns $-\infty$.
- y0(x) returns NaN for x < 0.
- y0($+\infty$) returns +0.
- y0(NaN) returns NaN.

__device__ double **y1** (double x)

Calculate the value of the Bessel function of the second kind of order 1 for the input argument.

Calculate the value of the Bessel function of the second kind of order 1 for the input argument x, $Y_1(x)$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

Returns the value of the Bessel function of the second kind of order 1.

- y1(± 0) returns $-\infty$.
- y1(x) returns NaN for x < 0.
- y1($+\infty$) returns +0.
- y1(NaN) returns NaN.

__device__ double **yn**(int n, double x)

Calculate the value of the Bessel function of the second kind of order n for the input argument.

Calculate the value of the Bessel function of the second kind of order n for the input argument x, $Y_n(x)$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

Returns the value of the Bessel function of the second kind of order n.

- yn(n, x) returns NaN for n < 0.
- yn(n, ± 0) returns $-\infty$.
- yn(n, x) returns NaN for x < 0.
- yn(n, $+\infty$) returns +0.
- yn(n, NaN) returns NaN.

Chapter 9. Double Precision Intrinsics

This section describes double precision intrinsic functions that are only supported in device code.

To use these functions, you do not need to include any additional header file in your program.

Functions

__device__ double *__dadd_rd*(double x, double y)
Add two floating-point values in round-down mode.

__device__ double *__dadd_rn*(double x, double y)
Add two floating-point values in round-to-nearest-even mode.

__device__ double *__dadd_ru*(double x, double y)
Add two floating-point values in round-up mode.

__device__ double *__dadd_rz*(double x, double y)
Add two floating-point values in round-towards-zero mode.

__device__ double *__ddiv_rd*(double x, double y)
Divide two floating-point values in round-down mode.

__device__ double *__ddiv_rn*(double x, double y)
Divide two floating-point values in round-to-nearest-even mode.

__device__ double *__ddiv_ru*(double x, double y)
Divide two floating-point values in round-up mode.

__device__ double *__ddiv_rz*(double x, double y)
Divide two floating-point values in round-towards-zero mode.

__device__ double *__dmul_rd*(double x, double y)
Multiply two floating-point values in round-down mode.

__device__ double *__dmul_rn*(double x, double y)
Multiply two floating-point values in round-to-nearest-even mode.

__device__ double *__dmul_ru*(double x, double y)
Multiply two floating-point values in round-up mode.

__device__ double *__dmul_rz*(double x, double y)
Multiply two floating-point values in round-towards-zero mode.

__device__ double *__drcp_rd*(double x)
Compute $\frac{1}{x}$ in round-down mode.

__device__ double *__drcp_rn*(double x)
Compute $\frac{1}{x}$ in round-to-nearest-even mode.

__device__ double *__drcp_ru*(double x)
Compute $\frac{1}{x}$ in round-up mode.

__device__ double *__drcp_rz*(double x)
Compute $\frac{1}{x}$ in round-towards-zero mode.

__device__ double *__dsqrt_rd*(double x)
Compute $\sqrt{x}$ in round-down mode.

__device__ double *__dsqrt_rn*(double x)
Compute $\sqrt{x}$ in round-to-nearest-even mode.

__device__ double *__dsqrt_ru*(double x)
Compute $\sqrt{x}$ in round-up mode.

__device__ double *__dsqrt_rz*(double x)
Compute $\sqrt{x}$ in round-towards-zero mode.

__device__ double *__dsub_rd*(double x, double y)
Subtract two floating-point values in round-down mode.

__device__ double *__dsub_rn*(double x, double y)
Subtract two floating-point values in round-to-nearest-even mode.

__device__ double *__dsub_ru*(double x, double y)
Subtract two floating-point values in round-up mode.

__device__ double *__dsub_rz*(double x, double y)
Subtract two floating-point values in round-towards-zero mode.

__device__ double *__fma_rd*(double x, double y, double z)
Compute $x \times y + z$ as a single operation in round-down mode.

__device__ double *__fma_rn*(double x, double y, double z)
Compute $x \times y + z$ as a single operation in round-to-nearest-even mode.

__device__ double *__fma_ru*(double x, double y, double z)
Compute $x \times y + z$ as a single operation in round-up mode.

__device__ double *__fma_rz*(double x, double y, double z)
Compute $x \times y + z$ as a single operation in round-towards-zero mode.

9.1. Functions

__device__ double **__dadd_rd**(double x, double y)

Add two floating-point values in round-down mode.

Adds two floating-point values x and y in round-down (to negative infinity) mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns

Returns x + y.

- __dadd_rd(x, y) is equivalent to __dadd_rd(y, x).
- __dadd_rd(x, $\pm\infty$) returns $\pm\infty$ for finite x.
- __dadd_rd($\pm\infty$, $\pm\infty$) returns $\pm\infty$.
- __dadd_rd($\pm\infty$, $\mp\infty$) returns NaN.
- __dadd_rd(±0, ±0) returns ±0.
- __dadd_rd(x, -x) returns -0 for finite x, including ±0.
- If either argument is NaN, NaN is returned.

__device__ double **__dadd_rn**(double x, double y)

Add two floating-point values in round-to-nearest-even mode.

Adds two floating-point values x and y in round-to-nearest-even mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns

Returns x + y.

- __dadd_rn(x, y) is equivalent to __dadd_rn(y, x).
- __dadd_rn(x, $\pm\infty$) returns $\pm\infty$ for finite x.
- __dadd_rn($\pm\infty$, $\pm\infty$) returns $\pm\infty$.
- __dadd_rn($\pm\infty$, $\mp\infty$) returns NaN.
- __dadd_rn(±0, ±0) returns ±0.
- __dadd_rn(x, -x) returns $+0$ for finite x, including ±0.
- If either argument is NaN, NaN is returned.

__device__ double **__dadd_ru**(double x, double y)

Add two floating-point values in round-up mode.

Adds two floating-point values x and y in round-up (to positive infinity) mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns

Returns x + y.

- __dadd_ru(x, y) is equivalent to __dadd_ru(y, x).
- __dadd_ru(x, $\pm\infty$) returns $\pm\infty$ for finite x.
- __dadd_ru($\pm\infty$, $\pm\infty$) returns $\pm\infty$.
- __dadd_ru($\pm\infty$, $\mp\infty$) returns NaN.
- __dadd_ru(± 0, ± 0) returns ± 0.
- __dadd_ru(x, -x) returns $+0$ for finite x, including ± 0.
- If either argument is NaN, NaN is returned.

__device__ double **__dadd_rz**(double x, double y)

Add two floating-point values in round-towards-zero mode.

Adds two floating-point values x and y in round-towards-zero mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns

Returns x + y.

- __dadd_rz(x, y) is equivalent to __dadd_rz(y, x).
- __dadd_rz(x, $\pm\infty$) returns $\pm\infty$ for finite x.
- __dadd_rz($\pm\infty$, $\pm\infty$) returns $\pm\infty$.
- __dadd_rz($\pm\infty$, $\mp\infty$) returns NaN.
- __dadd_rz(± 0, ± 0) returns ± 0.
- __dadd_rz(x, -x) returns $+0$ for finite x, including ± 0.
- If either argument is NaN, NaN is returned.

__device__ double **__ddiv_rd**(double x, double y)

Divide two floating-point values in round-down mode.

Divides two floating-point values x by y in round-down (to negative infinity) mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: Requires compute capability >= 2.0.

Returns

Returns x / y.

- sign of the quotient x / y is XOR of the signs of x and y when neither inputs nor result are NaN.
- __ddiv_rd(± 0, ± 0) returns NaN.
- __ddiv_rd($\pm\infty$, $\pm\infty$) returns NaN.
- __ddiv_rd(x, $\pm\infty$) returns 0 of appropriate sign for finite x.
- __ddiv_rd($\pm\infty$, y) returns ∞ of appropriate sign for finite y.
- __ddiv_rd(x, ± 0) returns ∞ of appropriate sign for x $\neq 0$.
- __ddiv_rd(± 0, y) returns 0 of appropriate sign for y $\neq 0$.
- If either argument is NaN, NaN is returned.

__device__ double **__ddiv_rn**(double x, double y)

Divide two floating-point values in round-to-nearest-even mode.

Divides two floating-point values x by y in round-to-nearest-even mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: Requires compute capability >= 2.0.

Returns

Returns x / y.

- sign of the quotient x / y is XOR of the signs of x and y when neither inputs nor result are NaN.
- __ddiv_rn(± 0, ± 0) returns NaN.
- __ddiv_rn($\pm\infty$, $\pm\infty$) returns NaN.
- __ddiv_rn(x, $\pm\infty$) returns 0 of appropriate sign for finite x.
- __ddiv_rn($\pm\infty$, y) returns ∞ of appropriate sign for finite y.
- __ddiv_rn(x, ± 0) returns ∞ of appropriate sign for x $\neq 0$.
- __ddiv_rn(± 0, y) returns 0 of appropriate sign for y $\neq 0$.
- If either argument is NaN, NaN is returned.

__device__ double **__ddiv_ru**(double x, double y)

Divide two floating-point values in round-up mode.

Divides two floating-point values x by y in round-up (to positive infinity) mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: Requires compute capability >= 2.0.

Returns

Returns x / y.

- sign of the quotient x / y is XOR of the signs of x and y when neither inputs nor result are NaN.
- __ddiv_ru(± 0, ± 0) returns NaN.
- __ddiv_ru($\pm\infty$, $\pm\infty$) returns NaN.
- __ddiv_ru(x, $\pm\infty$) returns 0 of appropriate sign for finite x.
- __ddiv_ru($\pm\infty$, y) returns ∞ of appropriate sign for finite y.
- __ddiv_ru(x, ± 0) returns ∞ of appropriate sign for x $\neq 0$.
- __ddiv_ru(± 0, y) returns 0 of appropriate sign for y $\neq 0$.
- If either argument is NaN, NaN is returned.

__device__ double **__ddiv_rz**(double x, double y)

Divide two floating-point values in round-towards-zero mode.

Divides two floating-point values x by y in round-towards-zero mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: Requires compute capability >= 2.0.

Returns

Returns x / y.

- sign of the quotient x / y is XOR of the signs of x and y when neither inputs nor result are NaN.
- __ddiv_rz(± 0, ± 0) returns NaN.
- __ddiv_rz($\pm\infty$, $\pm\infty$) returns NaN.
- __ddiv_rz(x, $\pm\infty$) returns 0 of appropriate sign for finite x.
- __ddiv_rz($\pm\infty$, y) returns ∞ of appropriate sign for finite y.
- __ddiv_rz(x, ± 0) returns ∞ of appropriate sign for x $\neq 0$.
- __ddiv_rz(± 0, y) returns 0 of appropriate sign for y $\neq 0$.
- If either argument is NaN, NaN is returned.

__device__ double **__dmul_rd**(double x, double y)

Multiply two floating-point values in round-down mode.

Multiplies two floating-point values x and y in round-down (to negative infinity) mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns

Returns x * y.

- sign of the product x * y is XOR of the signs of x and y when neither inputs nor result are NaN.
- __dmul_rd(x, y) is equivalent to __dmul_rd(y, x).
- __dmul_rd(x, $\pm\infty$) returns ∞ of appropriate sign for x $\neq 0$.
- __dmul_rd(± 0, $\pm\infty$) returns NaN.
- __dmul_rd(± 0, y) returns 0 of appropriate sign for finite y.
- If either argument is NaN, NaN is returned.

__device__ double **__dmul_rn**(double x, double y)

Multiply two floating-point values in round-to-nearest-even mode.

Multiplies two floating-point values x and y in round-to-nearest-even mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns

Returns x * y.

- sign of the product x * y is XOR of the signs of x and y when neither inputs nor result are NaN.
- __dmul_rn(x, y) is equivalent to __dmul_rn(y, x).
- __dmul_rn(x, $\pm\infty$) returns ∞ of appropriate sign for x $\neq 0$.
- __dmul_rn(± 0, $\pm\infty$) returns NaN.
- __dmul_rn(± 0, y) returns 0 of appropriate sign for finite y.
- If either argument is NaN, NaN is returned.

__device__ double **__dmul_ru**(double x, double y)

Multiply two floating-point values in round-up mode.

Multiplies two floating-point values x and y in round-up (to positive infinity) mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns

Returns x * y.

- sign of the product x * y is XOR of the signs of x and y when neither inputs nor result are NaN.
- __dmul_ru(x, y) is equivalent to __dmul_ru(y, x).
- __dmul_ru(x, $\pm\infty$) returns ∞ of appropriate sign for x $\neq 0$.
- __dmul_ru(± 0, $\pm\infty$) returns NaN.
- __dmul_ru(± 0, y) returns 0 of appropriate sign for finite y.
- If either argument is NaN, NaN is returned.

__device__ double **__dmul_rz**(double x, double y)

Multiply two floating-point values in round-towards-zero mode.

Multiplies two floating-point values x and y in round-towards-zero mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns

Returns x * y.

- sign of the product x * y is XOR of the signs of x and y when neither inputs nor result are NaN.
- __dmul_rz(x, y) is equivalent to __dmul_rz(y, x).
- __dmul_rz(x, $\pm\infty$) returns ∞ of appropriate sign for x $\neq 0$.
- __dmul_rz(± 0, $\pm\infty$) returns NaN.
- __dmul_rz(± 0, y) returns 0 of appropriate sign for finite y.
- If either argument is NaN, NaN is returned.

__device__ double **__drcp_rd**(double x)

Compute $\frac{1}{x}$ in round-down mode.

Compute the reciprocal of x in round-down (to negative infinity) mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: Requires compute capability >= 2.0.

Returns

Returns $\frac{1}{x}$.

__device__ double **__drcp_rn**(double x)

Compute $\frac{1}{x}$ in round-to-nearest-even mode.

Compute the reciprocal of x in round-to-nearest-even mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: Requires compute capability >= 2.0.

Returns

Returns $\frac{1}{x}$.

__device__ double **__drcp_ru**(double x)

Compute $\frac{1}{x}$ in round-up mode.

Compute the reciprocal of x in round-up (to positive infinity) mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: Requires compute capability >= 2.0.

Returns

Returns $\frac{1}{x}$.

__device__ double **__drcp_rz**(double x)

Compute $\frac{1}{x}$ in round-towards-zero mode.

Compute the reciprocal of x in round-towards-zero mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: Requires compute capability >= 2.0.

Returns

Returns $\frac{1}{x}$.

__device__ double **__dsqrt_rd**(double x)

Compute $\sqrt{x}$ in round-down mode.

Compute the square root of x in round-down (to negative infinity) mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: Requires compute capability >= 2.0.

Returns

Returns $\sqrt{x}$.

__device__ double **__dsqrt_rn**(double x)

Compute $\sqrt{x}$ in round-to-nearest-even mode.

Compute the square root of x in round-to-nearest-even mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: Requires compute capability >= 2.0.

Returns

Returns $\sqrt{x}$.

__device__ double **__dsqrt_ru**(double x)

Compute $\sqrt{x}$ in round-up mode.

Compute the square root of x in round-up (to positive infinity) mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: Requires compute capability >= 2.0.

Returns

Returns $\sqrt{x}$.

__device__ double **__dsqrt_rz**(double x)

Compute $\sqrt{x}$ in round-towards-zero mode.

Compute the square root of x in round-towards-zero mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: Requires compute capability >= 2.0.

Returns

Returns $\sqrt{x}$.

__device__ double **__dsub_rd**(double x, double y)

Subtract two floating-point values in round-down mode.

Subtracts two floating-point values x and y in round-down (to negative infinity) mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns

Returns x - y.

- __dsub_rd($\pm\infty$, y) returns $\pm\infty$ for finite y.
- __dsub_rd(x, $\pm\infty$) returns $\mp\infty$ for finite x.
- __dsub_rd($\pm\infty$, $\pm\infty$) returns NaN.
- __dsub_rd($\pm\infty$, $\mp\infty$) returns $\pm\infty$.
- __dsub_rd(± 0, ∓ 0) returns ± 0.
- __dsub_rd(x, x) returns -0 for finite x, including ± 0.
- If either argument is NaN, NaN is returned.

__device__ double **__dsub_rn**(double x, double y)

Subtract two floating-point values in round-to-nearest-even mode.

Subtracts two floating-point values x and y in round-to-nearest-even mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns

Returns x - y.

- __dsub_rn($\pm\infty$, y) returns $\pm\infty$ for finite y.
- __dsub_rn(x, $\pm\infty$) returns $\mp\infty$ for finite x.
- __dsub_rn($\pm\infty$, $\pm\infty$) returns NaN.
- __dsub_rn($\pm\infty$, $\mp\infty$) returns $\pm\infty$.
- __dsub_rn(± 0, ∓ 0) returns ± 0.
- __dsub_rn(x, x) returns $+0$ for finite x, including ± 0.
- If either argument is NaN, NaN is returned.

__device__ double **__dsub_ru**(double x, double y)

Subtract two floating-point values in round-up mode.

Subtracts two floating-point values x and y in round-up (to positive infinity) mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns

Returns x - y.

- __dsub_ru($\pm\infty$, y) returns $\pm\infty$ for finite y.
- __dsub_ru(x, $\pm\infty$) returns $\mp\infty$ for finite x.
- __dsub_ru($\pm\infty$, $\pm\infty$) returns NaN.
- __dsub_ru($\pm\infty$, $\mp\infty$) returns $\pm\infty$.
- __dsub_ru(± 0, ∓ 0) returns ± 0.
- __dsub_ru(x, x) returns $+0$ for finite x, including ± 0.
- If either argument is NaN, NaN is returned.

__device__ double **__dsub_rz**(double x, double y)

Subtract two floating-point values in round-towards-zero mode.

Subtracts two floating-point values x and y in round-towards-zero mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns

Returns x - y.

- __dsub_rz($\pm\infty$, y) returns $\pm\infty$ for finite y.
- __dsub_rz(x, $\pm\infty$) returns $\mp\infty$ for finite x.
- __dsub_rz($\pm\infty$, $\pm\infty$) returns NaN.
- __dsub_rz($\pm\infty$, $\mp\infty$) returns $\pm\infty$.
- __dsub_rz(± 0, ∓ 0) returns ± 0.
- __dsub_rz(x, x) returns $+0$ for finite x, including ± 0.
- If either argument is NaN, NaN is returned.

__device__ double **__fma_rd**(double x, double y, double z)

Compute $x \times y + z$ as a single operation in round-down mode.

Computes the value of $x \times y + z$ as a single ternary operation, rounding the result once in round-down (to negative infinity) mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Returns

Returns the rounded value of $x \times y + z$ as a single operation.

- __fma_rd($\pm\infty$, ± 0 , z) returns NaN.
- __fma_rd(± 0 , $\pm\infty$, z) returns NaN.
- __fma_rd(x, y, $-\infty$) returns NaN if $x \times y$ is an exact $+\infty$.
- __fma_rd(x, y, $+\infty$) returns NaN if $x \times y$ is an exact $-\infty$.
- __fma_rd(x, y, ± 0) returns ± 0 if $x \times y$ is exact ± 0.
- __fma_rd(x, y, ∓ 0) returns -0 if $x \times y$ is exact ± 0.
- __fma_rd(x, y, z) returns -0 if $x \times y + z$ is exactly zero and $z \neq 0$.
- If either argument is NaN, NaN is returned.

__device__ double **__fma_rn**(double x, double y, double z)

Compute $x \times y + z$ as a single operation in round-to-nearest-even mode.

Computes the value of $x \times y + z$ as a single ternary operation, rounding the result once in round-to-nearest-even mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Returns

Returns the rounded value of $x \times y + z$ as a single operation.

- __fma_rn($\pm\infty$, ± 0 , z) returns NaN.
- __fma_rn(± 0 , $\pm\infty$, z) returns NaN.

- __fma_rn(x, y, $-\infty$) returns NaN if $x \times y$ is an exact $+\infty$.
- __fma_rn(x, y, $+\infty$) returns NaN if $x \times y$ is an exact $-\infty$.
- __fma_rn(x, y, ± 0) returns ± 0 if $x \times y$ is exact ± 0.
- __fma_rn(x, y, ∓ 0) returns $+0$ if $x \times y$ is exact ± 0.
- __fma_rn(x, y, z) returns $+0$ if $x \times y + z$ is exactly zero and $z \neq 0$.
- If either argument is NaN, NaN is returned.

__device__ double **__fma_ru**(double x, double y, double z)

Compute $x \times y + z$ as a single operation in round-up mode.

Computes the value of $x \times y + z$ as a single ternary operation, rounding the result once in round-up (to positive infinity) mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Returns

Returns the rounded value of $x \times y + z$ as a single operation.

- __fma_ru($\pm\infty$, ± 0 , z) returns NaN.
- __fma_ru(± 0 , $\pm\infty$, z) returns NaN.
- __fma_ru(x, y, $-\infty$) returns NaN if $x \times y$ is an exact $+\infty$.
- __fma_ru(x, y, $+\infty$) returns NaN if $x \times y$ is an exact $-\infty$.
- __fma_ru(x, y, ± 0) returns ± 0 if $x \times y$ is exact ± 0.
- __fma_ru(x, y, ∓ 0) returns $+0$ if $x \times y$ is exact ± 0.
- __fma_ru(x, y, z) returns $+0$ if $x \times y + z$ is exactly zero and $z \neq 0$.
- If either argument is NaN, NaN is returned.

__device__ double **__fma_rz**(double x, double y, double z)

Compute $x \times y + z$ as a single operation in round-towards-zero mode.

Computes the value of $x \times y + z$ as a single ternary operation, rounding the result once in round-towards-zero mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Returns

Returns the rounded value of $x \times y + z$ as a single operation.

- __fma_rz($\pm\infty$, ± 0 , z) returns NaN.
- __fma_rz(± 0 , $\pm\infty$, z) returns NaN.
- __fma_rz(x, y, $-\infty$) returns NaN if $x \times y$ is an exact $+\infty$.
- __fma_rz(x, y, $+\infty$) returns NaN if $x \times y$ is an exact $-\infty$.

- __fma_rz(x, y, ± 0) returns ± 0 if $x \times y$ is exact ± 0.
- __fma_rz(x, y, ∓ 0) returns $+0$ if $x \times y$ is exact ± 0.
- __fma_rz(x, y, z) returns $+0$ if $x \times y + z$ is exactly zero and $z \neq 0$.
- If either argument is NaN, NaN is returned.

Chapter 10. FP128 Quad Precision Mathematical Functions

This section describes quad precision mathematical functions.

To use these functions, include the header file `device_fp128_functions.h` in your program.

Functions declared here have `__nv_fp128_` prefix to distinguish them from other global namespace symbols.

Note that FP128 CUDA Math functions are only available to device programs on platforms where host compiler supports the basic quad precision datatype `__float128` or `_Float128`.

Every FP128 CUDA Math function name is overloaded to support either of these host-compiler-specific types, whenever the types are available. See for example:

```
#ifdef __FLOAT128_CPP_SPELLING_ENABLED__
    __float128 __nv_fp128_sqrt(__float128 x);
#endif
#ifdef __FLOAT128_C_SPELLING_ENABLED__
    _Float128 __nv_fp128_sqrt(_Float128 x);
#endif
```

Note: FP128 device computations require compute capability >= 10.0.

Functions

__device__ __float128 *__nv_fp128_acos*(__float128 x)
Calculate $\cos^{-1} x$, the arc cosine of input argument.

__device__ __float128 *__nv_fp128_acosh*(__float128 x)
Calculate $\cosh^{-1} x$, the nonnegative inverse hyperbolic cosine of the input argument.

__device__ __float128 *__nv_fp128_add*(__float128 x, __float128 y)
Compute $x + y$, the sum of the two floating-point inputs using round-to-nearest-even rounding mode.

__device__ __float128 *__nv_fp128_asin*(__float128 x)
Calculate $\sin^{-1} x$, the arc sine of input argument.

__device__ __float128 *__nv_fp128_asinh*(__float128 x)
Calculate $\sinh^{-1} x$, the inverse hyperbolic sine of the input argument.

__device__ __float128 *__nv_fp128_atan*(__float128 x)
Calculate $\tan^{-1} x$, the arc tangent of input argument.

__device__ __float128 *__nv_fp128_atanh*(__float128 x)
Calculate $\tanh^{-1} x$, the inverse hyperbolic tangent of the input argument.

__device__ __float128 *__nv_fp128_ceil*(__float128 x)
Calculate $\lceil x \rceil$, the smallest integer greater than or equal to x .

__device__ __float128 *__nv_fp128_copysign*(__float128 x, __float128 y)
Create value with the magnitude of the first agument x , and the sign of the second argument y .

__device__ __float128 *__nv_fp128_cos*(__float128 x)
Calculate $\cos x$, the cosine of input argument (measured in radians).

__device__ __float128 *__nv_fp128_cosh*(__float128 x)
Calculate $\cosh x$, the hyperbolic cosine of the input argument.

__device__ __float128 *__nv_fp128_div*(__float128 x, __float128 y)
Compute $\frac{x}{y}$, the quotient of the two floating-point inputs using round-to-nearest-even rounding mode.

__device__ __float128 *__nv_fp128_exp*(__float128 x)
Calculate e^x , the base e exponential of the input argument.

__device__ __float128 *__nv_fp128_exp10*(__float128 x)
Calculate 10^x , the base 10 exponential of the input argument.

__device__ __float128 *__nv_fp128_exp2*(__float128 x)
Calculate 2^x , the base 2 exponential of the input argument.

__device__ __float128 *__nv_fp128_expm1*(__float128 x)
Calculate $e^x - 1$, the base e exponential of the input argument, minus 1.

__device__ __float128 *__nv_fp128_fabs*(__float128 x)
Calculate $|x|$, the absolute value of the input argument.

__device__ __float128 *__nv_fp128_fdim*(__float128 x, __float128 y)
Compute the positive difference between x and y .

__device__ __float128 *__nv_fp128_floor*(__float128 x)
Calculate $\lfloor x \rfloor$, the largest integer less than or equal to x .

__device__ __float128 *__nv_fp128_fma*(__float128 x, __float128 y, __float128 c)
Compute $x \times y + z$ as a single operation using round-to-nearest-even rounding mode.

__device__ __float128 *__nv_fp128_fmax*(__float128 x, __float128 y)
Determine the maximum numeric value of the arguments.

__device__ __float128 *__nv_fp128_fmin*(__float128 x, __float128 y)
Determine the minimum numeric value of the arguments.

__device__ __float128 *__nv_fp128_fmod*(__float128 x, __float128 y)
Calculate the floating-point remainder of x / y .

__device__ __float128 *__nv_fp128_frexp*(__float128 x, int *nptr)
Extract mantissa and exponent of the floating-point input argument.

__device__ __float128 *__nv_fp128_hypot*(__float128 x, __float128 y)
Calculate $\sqrt{x^2 + y^2}$, the square root of the sum of squares of two arguments.

__device__ int *__nv_fp128_ilogb*(__float128 x)
Compute the unbiased integer exponent of the input argument.

__device__ int *__nv_fp128_isnan*(__float128 x)
Determine whether the input argument is a NaN.

__device__ int *__nv_fp128_isunordered*(__float128 x, __float128 y)
Determine whether the pair of inputs is unordered.

__device__ __float128 *__nv_fp128_ldexp*(__float128 x, int exp)
Calculate the value of $x \cdot 2^{exp}$.

__device__ __float128 *__nv_fp128_log*(__float128 x)
Calculate $\log_e x$, the base e logarithm of the input argument.

__device__ __float128 *__nv_fp128_log10*(__float128 x)
Calculate $\log_{10} x$, the base 10 logarithm of the input argument.

__device__ __float128 *__nv_fp128_log1p*(__float128 x)
Calculate the value of $\log_e(1 + x)$.

__device__ __float128 *__nv_fp128_log2*(__float128 x)
Calculate $\log_2 x$, the base 2 logarithm of the input argument.

__device__ __float128 *__nv_fp128_modf*(__float128 x, __float128 *iptr)
Break down the input argument into fractional and integral parts.

__device__ __float128 *__nv_fp128_mul*(__float128 x, __float128 y)
Compute $x \cdot y$, the product of the two floating-point inputs using round-to-nearest-even rounding mode.

__device__ __float128 *__nv_fp128_pow*(__float128 x, __float128 y)
Calculate the value of x^y , first argument to the power of second argument.

__device__ __float128 *__nv_fp128_remainder*(__float128 x, __float128 y)
Compute the floating-point remainder function.

__device__ __float128 *__nv_fp128_rint*(__float128 x)
Round to nearest integer value in floating-point format, with halfway cases rounded to the nearest even integer value.

__device__ __float128 *__nv_fp128_round*(__float128 x)
Round to nearest integer value in floating-point format, with halfway cases rounded away from zero.

__device__ __float128 *__nv_fp128_sin*(__float128 x)
Calculate $\sin x$, the sine of input argument (measured in radians).

__device__ __float128 *__nv_fp128_sinh*(__float128 x)
Calculate $\sinh x$, the hyperbolic sine of the input argument.

__device__ __float128 *__nv_fp128_sqrt*(__float128 x)
Calculate $\sqrt{x}$, the square root of the input argument.

__device__ __float128 *__nv_fp128_sub*(__float128 x, __float128 y)
Compute $x - y$, the difference of the two floating-point inputs using round-to-nearest-even rounding mode.

__device__ __float128 *__nv_fp128_tan*(__float128 x)
Calculate $\tan x$, the tangent of input argument (measured in radians).

__device__ __float128 *__nv_fp128_tanh*(__float128 x)
Calculate $\tanh x$, the hyperbolic tangent of the input argument.

__device__ __float128 *__nv_fp128_trunc*(__float128 x)
Truncate input argument to the integral part.

10.1. Functions

__device__ __float128 **__nv_fp128_acos**(__float128 x)

Calculate $\cos^{-1} x$, the arc cosine of input argument.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

The principal value of the arc cosine of the input argument x. Result will be in radians, in the interval [0, π] for x inside [-1, +1].

- __nv_fp128_acos(1) returns +0.
- __nv_fp128_acos(x) returns NaN for x outside [-1, +1].
- __nv_fp128_acos(NaN) returns NaN.

__device__ __float128 **__nv_fp128_acosh**(__float128 x)

Calculate $\cosh^{-1} x$, the nonnegative inverse hyperbolic cosine of the input argument.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

Result will be in the interval [0, $+\infty$].

- __nv_fp128_acosh(1) returns 0.
- __nv_fp128_acosh(x) returns NaN for x in the interval [$-\infty$, 1).
- __nv_fp128_acosh($+\infty$) returns $+\infty$.
- __nv_fp128_acosh(NaN) returns NaN.

__device__ __float128 **__nv_fp128_add**(__float128 x, __float128 y)

Compute $x + y$, the sum of the two floating-point inputs using round-to-nearest-even rounding mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

Returns x + y.

- __nv_fp128_add(x, y) is equivalent to __nv_fp128_add(y, x).
- __nv_fp128_add(x, $\pm\infty$) returns $\pm\infty$ for finite x.
- __nv_fp128_add($\pm\infty$, $\pm\infty$) returns $\pm\infty$.
- __nv_fp128_add($\pm\infty$, $\mp\infty$) returns NaN.
- __nv_fp128_add(± 0, ± 0) returns ± 0.
- __nv_fp128_add(x, -x) returns $+0$ for finite x, including ± 0.
- If either argument is NaN, NaN is returned.

__device__ __float128 **__nv_fp128_asin**(__float128 x)

Calculate $\sin^{-1} x$, the arc sine of input argument.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

The principal value of the arc sine of the input argument x. Result will be in radians, in the interval [- $\pi/2$, + $\pi/2$] for x inside [-1, +1].

- __nv_fp128_asin(± 0) returns ± 0.
- __nv_fp128_asin(x) returns NaN for x outside [-1, +1].
- __nv_fp128_asin(NaN) returns NaN.

__device__ __float128 **__nv_fp128_asinh**(__float128 x)

Calculate $\sinh^{-1} x$, the inverse hyperbolic sine of the input argument.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

- __nv_fp128_asinh(± 0) returns ± 0.
- __nv_fp128_asinh($\pm\infty$) returns $\pm\infty$.

- __nv_fp128_asinh(NaN) returns NaN.

__device__ __float128 **__nv_fp128_atan**(__float128 x)

Calculate $\tan^{-1} x$, the arc tangent of input argument.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

The principal value of the arc tangent of the input argument x. Result will be in radians, in the interval [- $\pi/2$, + $\pi/2$].

- __nv_fp128_atan(± 0) returns ± 0.
- __nv_fp128_atan($\pm\infty$) returns $\pm\pi$ /2.
- __nv_fp128_atan(NaN) returns NaN.

__device__ __float128 **__nv_fp128_atanh**(__float128 x)

Calculate $\tanh^{-1} x$, the inverse hyperbolic tangent of the input argument.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

- __nv_fp128_atanh(± 0) returns ± 0.
- __nv_fp128_atanh(± 1) returns $\pm\infty$.
- __nv_fp128_atanh(x) returns NaN for x outside interval [-1, 1].
- __nv_fp128_atanh(NaN) returns NaN.

__device__ __float128 **__nv_fp128_ceil**(__float128 x)

Calculate $\lceil x \rceil$, the smallest integer greater than or equal to x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

$\lceil x \rceil$ expressed as a floating-point number.

- __nv_fp128_ceil($\pm\infty$) returns $\pm\infty$.
- __nv_fp128_ceil(± 0) returns ± 0.
- __nv_fp128_ceil(NaN) returns NaN.

__device__ __float128 **__nv_fp128_copysign**(__float128 x, __float128 y)

Create value with the magnitude of the first agument x, and the sign of the second argument y.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

- copysign(NaN, y) returns a NaN with the sign of y.

__device__ __float128 **__nv_fp128_cos**(__float128 x)

Calculate cos x, the cosine of input argument (measured in radians).

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

cos x.

- __nv_fp128_cos(± 0) returns 1.
- __nv_fp128_cos($\pm\infty$) returns NaN.
- __nv_fp128_cos(NaN) returns NaN.

__device__ __float128 **__nv_fp128_cosh**(__float128 x)

Calculate cosh x, the hyperbolic cosine of the input argument.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

- __nv_fp128_cosh(± 0) returns 1.
- __nv_fp128_cosh($\pm\infty$) returns $+\infty$.
- __nv_fp128_cosh(NaN) returns NaN.

__device__ __float128 **__nv_fp128_div**(__float128 x, __float128 y)

Compute $\frac{x}{y}$, the quotient of the two floating-point inputs using round-to-nearest-even rounding mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

- sign of the quotient x / y is XOR of the signs of x and y when neither inputs nor result are NaN.
- __nv_fp128_div(± 0, ± 0) returns NaN.
- __nv_fp128_div($\pm\infty$, $\pm\infty$) returns NaN.
- __nv_fp128_div(x, $\pm\infty$) returns 0 of appropriate sign for finite x.
- __nv_fp128_div($\pm\infty$, y) returns ∞ of appropriate sign for finite y.
- __nv_fp128_div(x, ± 0) returns ∞ of appropriate sign for x $\neq 0$.
- __nv_fp128_div(± 0, y) returns 0 of appropriate sign for y $\neq 0$.
- If either argument is NaN, NaN is returned.

__device__ __float128 **__nv_fp128_exp**(__float128 x)

Calculate e^x, the base e exponential of the input argument.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

- __nv_fp128_exp(± 0) returns 1.
- __nv_fp128_exp($-\infty$) returns +0.
- __nv_fp128_exp($+\infty$) returns $+\infty$.

- __nv_fp128_exp(NaN) returns NaN.

__device__ __float128 **__nv_fp128_exp10**(__float128 x)

Calculate 10^x, the base 10 exponential of the input argument.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

- __nv_fp128_exp10(± 0) returns 1.
- __nv_fp128_exp10($-\infty$) returns +0.
- __nv_fp128_exp10($+\infty$) returns $+\infty$.
- __nv_fp128_exp10(NaN) returns NaN.

__device__ __float128 **__nv_fp128_exp2**(__float128 x)

Calculate 2^x, the base 2 exponential of the input argument.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

- __nv_fp128_exp2(± 0) returns 1.
- ex__nv_fp128_exp2p2f($-\infty$) returns +0.
- __nv_fp128_exp2($+\infty$) returns $+\infty$.
- __nv_fp128_exp2(NaN) returns NaN.

__device__ __float128 **__nv_fp128_expm1**(__float128 x)

Calculate $e^x - 1$, the base e exponential of the input argument, minus 1.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

- __nv_fp128_expm1(± 0) returns ± 0.
- __nv_fp128_expm1($-\infty$) returns -1.
- __nv_fp128_expm1($+\infty$) returns $+\infty$.
- __nv_fp128_expm1(NaN) returns NaN.

__device__ __float128 **__nv_fp128_fabs**(__float128 x)

Calculate $|x|$, the absolute value of the input argument.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

- __nv_fp128_fabs($\pm\infty$) returns $+\infty$.
- __nv_fp128_fabs(± 0) returns +0.
- __nv_fp128_fabs(NaN) returns an unspecified NaN.

__device__ __float128 **__nv_fp128_fdim**(__float128 x, __float128 y)

Compute the positive difference between x and y.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

- __nv_fp128_fdim(x, y) returns x - y if $x > y$.
- __nv_fp128_fdim(x, y) returns +0 if $x \leq y$.
- If either argument is NaN, NaN is returned.

__device__ __float128 **__nv_fp128_floor**(__float128 x)

Calculate $\lfloor x \rfloor$, the largest integer less than or equal to x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

$\lfloor x \rfloor$ expressed as a floating-point number.

- __nv_fp128_floor($\pm\infty$) returns $\pm\infty$.
- __nv_fp128_floor(± 0) returns ± 0.
- __nv_fp128_floor(NaN) returns NaN.

__device__ __float128 **__nv_fp128_fma**(__float128 x, __float128 y, __float128 c)

Compute $x \times y + z$ as a single operation using round-to-nearest-even rounding mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

The value of $x \times y + z$ as a single ternary operation, rounded once using round-to-nearest, ties-to-even rounding mode.

- __nv_fp128_fma($\pm\infty$, ± 0 , z) returns NaN.
- __nv_fp128_fma(± 0 , $\pm\infty$, z) returns NaN.
- __nv_fp128_fma(x, y, $-\infty$) returns NaN if $x \times y$ is an exact $+\infty$.
- __nv_fp128_fma(x, y, $+\infty$) returns NaN if $x \times y$ is an exact $-\infty$.
- __nv_fp128_fma(x, y, ± 0) returns ± 0 if $x \times y$ is exact ± 0.
- __nv_fp128_fma(x, y, ∓ 0) returns $+0$ if $x \times y$ is exact ± 0.
- __nv_fp128_fma(x, y, z) returns $+0$ if $x \times y + z$ is exactly zero and $z \neq 0$.
- If either argument is NaN, NaN is returned.

__device__ __float128 **__nv_fp128_fmax**(__float128 x, __float128 y)

Determine the maximum numeric value of the arguments.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

The maximum numeric value of the arguments x and y. Treats NaN arguments as missing data.

- If both arguments are NaN, returns NaN.
- If one argument is NaN, returns the numeric argument.

__device__ __float128 **__nv_fp128_fmin**(__float128 x, __float128 y)

Determine the minimum numeric value of the arguments.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

The minimum numeric value of the arguments `x` and `y`. Treats NaN arguments as missing data.

- If both arguments are NaN, returns NaN.
- If one argument is NaN, returns the numeric argument.

__device__ __float128 **__nv_fp128_fmod**(__float128 x, __float128 y)

Calculate the floating-point remainder of x / y.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

The floating-point remainder of the division operation x / y calculated by this function is exactly the value `x - n*y`, where n is x / y with its fractional part truncated.

- The computed value will have the same sign as x, and its magnitude will be less than the magnitude of y.
- __nv_fp128_fmod(± 0 , y) returns ± 0 if y is not zero.
- __nv_fp128_fmod(x, $\pm\infty$) returns x if x is finite.
- __nv_fp128_fmod(x, y) returns NaN if x is $\pm\infty$ or y is zero.
- If either argument is NaN, NaN is returned.

__device__ __float128 **__nv_fp128_frexp**(__float128 x, int *nptr)

Extract mantissa and exponent of the floating-point input argument.

Decompose the floating-point value `x` into a component `m` for the normalized fraction element and an integral term `n` for the exponent. The absolute value of `m` will be greater than or equal to 0.5 and less than 1.0 or it will be equal to 0; $x = m \cdot 2^n$. The integer exponent `n` will be stored in the location to which `nptr` points.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

The fractional component `m`.

- __nv_fp128_frexp(± 0 , `nptr`) returns ± 0 and stores zero in the location pointed to by `nptr`.
- __nv_fp128_frexp($\pm\infty$, `nptr`) returns $\pm\infty$ and stores an unspecified value in the location to which `nptr` points.
- __nv_fp128_frexp(NaN, y) returns a NaN and stores an unspecified value in the location to which `nptr` points.

__device__ __float128 **__nv_fp128_hypot**(__float128 x, __float128 y)

Calculate $\sqrt{x^2+y^2}$, the square root of the sum of squares of two arguments.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

The length of the hypotenuse of a right triangle whose two sides have lengths $|x|$ and $|y|$ without undue overflow or underflow.

- __nv_fp128_hypot(x,y), __nv_fp128_hypot(y,x), and __nv_fp128_hypot(x, -y) are equivalent.
- __nv_fp128_hypot(x, ± 0) is equivalent to __nv_fp128_fabs(x).
- __nv_fp128_hypot($\pm\infty$,y) returns $+\infty$, even if y is a NaN.
- __nv_fp128_hypot(NaN, y) returns NaN, when y is not $\pm\infty$.

__device__ int **__nv_fp128_ilogb**(__float128 x)

Compute the unbiased integer exponent of the input argument.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

- If successful, returns the unbiased exponent of the argument.
- __nv_fp128_ilogb(± 0) returns `INT_MIN`.
- __nv_fp128_ilogb(NaN) returns `INT_MIN`.
- __nv_fp128_ilogb($\pm\infty$) returns `INT_MAX`.
- Note: above behavior does not take into account `FP_ILOGB0` nor `FP_ILOGBNAN`.

__device__ int **__nv_fp128_isnan**(__float128 x)

Determine whether the input argument is a NaN.

Note: FP128 device computations require compute capability >= 10.0.

Returns

A nonzero value if and only if x is a NaN value.

__device__ int **__nv_fp128_isunordered**(__float128 x, __float128 y)

Determine whether the pair of inputs is unordered.

Note: FP128 device computations require compute capability >= 10.0.

Returns

- nonzero value if at least one of input values is a NaN.
- zero otherwise

__device__ __float128 **__nv_fp128_ldexp**(__float128 x, int exp)

Calculate the value of $x \cdot 2^{exp}$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

- __nv_fp128_ldexp(± 0 , exp) returns ± 0.
- __nv_fp128_ldexp(x, 0) returns x.
- __nv_fp128_ldexp($\pm\infty$, exp) returns $\pm\infty$.
- __nv_fp128_ldexp(NaN, exp) returns NaN.

__device__ __float128 **__nv_fp128_log**(__float128 x)

Calculate $\log_e x$, the base e logarithm of the input argument.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

- __nv_fp128_log(± 0) returns $-\infty$.
- __nv_fp128_log(1) returns +0.
- __nv_fp128_log(x) returns NaN for x < 0.
- __nv_fp128_log($+\infty$) returns $+\infty$.
- __nv_fp128_log(NaN) returns NaN.

__device__ __float128 **__nv_fp128_log10**(__float128 x)

Calculate $\log_{10} x$, the base 10 logarithm of the input argument.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

- __nv_fp128_log10(± 0) returns $-\infty$.
- __nv_fp128_log10(1) returns +0.
- __nv_fp128_log10(x) returns NaN for x < 0.
- __nv_fp128_log10($+\infty$) returns $+\infty$.
- __nv_fp128_log10(NaN) returns NaN.

__device__ __float128 **__nv_fp128_log1p**(__float128 x)

Calculate the value of $\log_e(1 + x)$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

- __nv_fp128_log1p(± 0) returns ± 0.
- __nv_fp128_log1p(-1) returns $-\infty$.
- __nv_fp128_log1p(x) returns NaN for x < -1.
- __nv_fp128_log1p($+\infty$) returns $+\infty$.
- __nv_fp128_log1p(NaN) returns NaN.

__device__ __float128 **__nv_fp128_log2**(__float128 x)

Calculate $\log_2 x$, the base 2 logarithm of the input argument.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

- __nv_fp128_log2(± 0) returns $-\infty$.
- __nv_fp128_log2(1) returns +0.
- __nv_fp128_log2(x) returns NaN for x < 0.
- __nv_fp128_log2($+\infty$) returns $+\infty$.
- __nv_fp128_log2(NaN) returns NaN.

__device__ __float128 **__nv_fp128_modf**(__float128 x, __float128 *iptr)

Break down the input argument into fractional and integral parts.

Break down the argument x into fractional and integral parts. The integral part is stored in floating-point format in the location to which `iptr` points. Fractional and integral parts are given the same sign as the argument x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

- __nv_fp128_modf($\pm x$, `iptr`) returns a result with the same sign as x.
- __nv_fp128_modf($\pm\infty$, `iptr`) returns ± 0 and stores $\pm\infty$ in the object pointed to by `iptr`.

- __nv_fp128_modf(NaN, `iptr`) stores a NaN in the object pointed to by `iptr` and returns a NaN.

__device__ __float128 **__nv_fp128_mul**(__float128 x, __float128 y)

Compute $x \cdot y$, the product of the two floating-point inputs using round-to-nearest-even rounding mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

Returns x * y.

- sign of the product x * y is XOR of the signs of x and y when neither inputs nor result are NaN.
- __nv_fp128_mul(x, y) is equivalent to __nv_fp128_mul(y, x).
- __nv_fp128_mul(x, $\pm\infty$) returns ∞ of appropriate sign for x $\neq 0$.
- __nv_fp128_mul(± 0, $\pm\infty$) returns NaN.
- __nv_fp128_mul(± 0, y) returns 0 of appropriate sign for finite y.
- If either argument is NaN, NaN is returned.

__device__ __float128 **__nv_fp128_pow**(__float128 x, __float128 y)

Calculate the value of x^y, first argument to the power of second argument.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

- __nv_fp128_pow(± 0 , y) returns $\pm\infty$ for y an odd integer less than 0.
- __nv_fp128_pow(± 0 , y) returns $+\infty$ for y less than 0 and not an odd integer.
- __nv_fp128_pow(± 0 , y) returns ± 0 for y an odd integer greater than 0.
- __nv_fp128_pow(± 0 , y) returns +0 for y > 0 and not an odd integer.
- __nv_fp128_pow(-1, $\pm\infty$) returns 1.
- __nv_fp128_pow(+1, y) returns 1 for any y, even a NaN.
- __nv_fp128_pow(x, ± 0) returns 1 for any x, even a NaN.
- __nv_fp128_pow(x, y) returns a NaN for finite x < 0 and finite non-integer y.

- __nv_fp128_pow(x, $-\infty$) returns $+\infty$ for $|x| < 1$.
- __nv_fp128_pow(x, $-\infty$) returns +0 for $|x| > 1$.
- __nv_fp128_pow(x, $+\infty$) returns +0 for $|x| < 1$.
- __nv_fp128_pow(x, $+\infty$) returns $+\infty$ for $|x| > 1$.
- __nv_fp128_pow($-\infty$, y) returns -0 for y an odd integer less than 0.
- __nv_fp128_pow($-\infty$, y) returns +0 for y < 0 and not an odd integer.
- __nv_fp128_pow($-\infty$, y) returns $-\infty$ for y an odd integer greater than 0.
- __nv_fp128_pow($-\infty$, y) returns $+\infty$ for y > 0 and not an odd integer.
- __nv_fp128_pow($+\infty$, y) returns +0 for y < 0.
- __nv_fp128_pow($+\infty$, y) returns $+\infty$ for y > 0.
- __nv_fp128_pow(x, y) returns NaN if either x or y or both are NaN and x $\neq$ +1 and y $\neq \pm 0$.

__device__ __float128 **__nv_fp128_remainder**(__float128 x, __float128 y)

Compute the floating-point remainder function.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

The floating-point remainder r of dividing x by y for nonzero y is defined as $r = x - ny$. The value n is the integer value nearest $\frac{x}{y}$. In the halfway cases when $|n - \frac{x}{y}| = \frac{1}{2}$, the even n value is chosen.

- __nv_fp128_remainder(x, ± 0) returns NaN.
- __nv_fp128_remainder($\pm\infty$, y) returns NaN.
- __nv_fp128_remainder(x, $\pm\infty$) returns x for finite x.
- If either argument is NaN, NaN is returned.

__device__ __float128 **__nv_fp128_rint**(__float128 x)

Round to nearest integer value in floating-point format, with halfway cases rounded to the nearest even integer value.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

- __nv_fp128_rint(± 0) returns ± 0.
- __nv_fp128_rint($\pm\infty$) returns $\pm\infty$.
- __nv_fp128_rint(NaN) returns NaN.

__device__ __float128 **__nv_fp128_round**(__float128 x)

Round to nearest integer value in floating-point format, with halfway cases rounded away from zero.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

- __nv_fp128_round(± 0) returns ± 0.
- __nv_fp128_round($\pm\infty$) returns $\pm\infty$.
- __nv_fp128_round(NaN) returns NaN.

__device__ __float128 **__nv_fp128_sin**(__float128 x)

Calculate $\sin x$, the sine of input argument (measured in radians).

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

$\sin x$.

- __nv_fp128_sin(± 0) returns ± 0.
- __nv_fp128_sin($\pm\infty$) returns NaN.
- __nv_fp128_sin(NaN) returns NaN.

__device__ __float128 **__nv_fp128_sinh**(__float128 x)

Calculate $\sinh x$, the hyperbolic sine of the input argument.

Calculate $\sinh x$, the hyperbolic sine of the input argument x.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

- __nv_fp128_sinhinh(± 0) returns ± 0.
- __nv_fp128_sinh($\pm\infty$) returns $\pm\infty$.
- __nv_fp128_sinh(NaN) returns NaN.

__device__ __float128 **__nv_fp128_sqrt**(__float128 x)

Calculate $\sqrt{x}$, the square root of the input argument.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

$\sqrt{x}$.

- __nv_fp128_sqrt(± 0) returns ± 0.
- __nv_fp128_sqrt($+\infty$) returns $+\infty$.
- __nv_fp128_sqrt(x) returns NaN if x is less than 0.
- __nv_fp128_sqrt(NaN) returns NaN.

__device__ __float128 **__nv_fp128_sub**(__float128 x, __float128 y)

Compute $x - y$, the difference of the two floating-point inputs using round-to-nearest-even rounding mode.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

Returns x - y.

- __nv_fp128_sub($\pm\infty$, y) returns $\pm\infty$ for finite y.
- __nv_fp128_sub(x, $\pm\infty$) returns $\mp\infty$ for finite x.
- __nv_fp128_sub($\pm\infty$, $\pm\infty$) returns NaN.
- __nv_fp128_sub($\pm\infty$, $\mp\infty$) returns $\pm\infty$.
- __nv_fp128_sub(± 0, ∓ 0) returns ± 0.

- __nv_fp128_sub(x, x) returns $+0$ for finite x, including ± 0.
- If either argument is NaN, NaN is returned.

__device__ __float128 **__nv_fp128_tan**(__float128 x)

Calculate tan x, the tangent of input argument (measured in radians).

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

tan x.

- __nv_fp128_tan(± 0) returns ± 0.
- __nv_fp128_tan($\pm\infty$) returns NaN.
- __nv_fp128_tan(NaN) returns NaN.

__device__ __float128 **__nv_fp128_tanh**(__float128 x)

Calculate tanh x, the hyperbolic tangent of the input argument.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

- __nv_fp128_tanh(± 0) returns ± 0.
- __nv_fp128_tanh($\pm\infty$) returns ± 1.
- __nv_fp128_tanh(NaN) returns NaN.

__device__ __float128 **__nv_fp128_trunc**(__float128 x)

Truncate input argument to the integral part.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Quad-Precision Floating-Point Functions section.

Note: FP128 device computations require compute capability >= 10.0.

Returns

Rounded x to the nearest integer value in floating-point format, that does not exceed x in magnitude.

- __nv_fp128_trunc(± 0) returns ± 0.
- __nv_fp128_trunc($\pm \infty$) returns $\pm \infty$.
- __nv_fp128_trunc(NaN) returns NaN.

Chapter 11. Type Casting Intrinsics

This section describes type casting intrinsic functions that are only supported in device code.

To use these functions, you do not need to include any additional header file in your program.

Functions

__device__ float *__double2float_rd*(double x)
Convert a double to a float in round-down mode.

__device__ float *__double2float_rn*(double x)
Convert a double to a float in round-to-nearest-even mode.

__device__ float *__double2float_ru*(double x)
Convert a double to a float in round-up mode.

__device__ float *__double2float_rz*(double x)
Convert a double to a float in round-towards-zero mode.

__device__ int *__double2hiint*(double x)
Reinterpret high 32 bits in a double as a signed integer.

__device__ int *__double2int_rd*(double x)
Convert a double to a signed int in round-down mode.

__device__ int *__double2int_rn*(double x)
Convert a double to a signed int in round-to-nearest-even mode.

__device__ int *__double2int_ru*(double x)
Convert a double to a signed int in round-up mode.

__device__ int *__double2int_rz*(double x)
Convert a double to a signed int in round-towards-zero mode.

__device__ long long int *__double2ll_rd*(double x)
Convert a double to a signed 64-bit int in round-down mode.

__device__ long long int *__double2ll_rn*(double x)
Convert a double to a signed 64-bit int in round-to-nearest-even mode.

__device__ long long int *__double2ll_ru*(double x)
Convert a double to a signed 64-bit int in round-up mode.

__device__ long long int *__double2ll_rz*(double x)
Convert a double to a signed 64-bit int in round-towards-zero mode.

__device__ int *__double2loint*(double x)
Reinterpret low 32 bits in a double as a signed integer.

__device__ unsigned int *__double2uint_rd*(double x)
Convert a double to an unsigned int in round-down mode.

__device__ unsigned int *__double2uint_rn*(double x)
Convert a double to an unsigned int in round-to-nearest-even mode.

__device__ unsigned int *__double2uint_ru*(double x)
Convert a double to an unsigned int in round-up mode.

__device__ unsigned int *__double2uint_rz*(double x)
Convert a double to an unsigned int in round-towards-zero mode.

__device__ unsigned long long int *__double2ull_rd*(double x)
Convert a double to an unsigned 64-bit int in round-down mode.

__device__ unsigned long long int *__double2ull_rn*(double x)
Convert a double to an unsigned 64-bit int in round-to-nearest-even mode.

__device__ unsigned long long int *__double2ull_ru*(double x)
Convert a double to an unsigned 64-bit int in round-up mode.

__device__ unsigned long long int *__double2ull_rz*(double x)
Convert a double to an unsigned 64-bit int in round-towards-zero mode.

__device__ long long int *__double_as_longlong*(double x)
Reinterpret bits in a double as a 64-bit signed integer.

__device__ int *__float2int_rd*(float x)
Convert a float to a signed integer in round-down mode.

__device__ int *__float2int_rn*(float x)
Convert a float to a signed integer in round-to-nearest-even mode.

__device__ int *__float2int_ru*(float)
Convert a float to a signed integer in round-up mode.

__device__ int *__float2int_rz*(float x)
Convert a float to a signed integer in round-towards-zero mode.

__device__ long long int *__float2ll_rd*(float x)
Convert a float to a signed 64-bit integer in round-down mode.

__device__ long long int *__float2ll_rn*(float x)
Convert a float to a signed 64-bit integer in round-to-nearest-even mode.

__device__ long long int *__float2ll_ru*(float x)
Convert a float to a signed 64-bit integer in round-up mode.

__device__ long long int *__float2ll_rz*(float x)
Convert a float to a signed 64-bit integer in round-towards-zero mode.

__device__ unsigned int *__float2uint_rd*(float x)
Convert a float to an unsigned integer in round-down mode.

__device__ unsigned int *__float2uint_rn*(float x)
Convert a float to an unsigned integer in round-to-nearest-even mode.

__device__ unsigned int *__float2uint_ru*(float x)
Convert a float to an unsigned integer in round-up mode.

__device__ unsigned int *__float2uint_rz*(float x)
Convert a float to an unsigned integer in round-towards-zero mode.

__device__ unsigned long long int *__float2ull_rd*(float x)
Convert a float to an unsigned 64-bit integer in round-down mode.

__device__ unsigned long long int *__float2ull_rn*(float x)
Convert a float to an unsigned 64-bit integer in round-to-nearest-even mode.

__device__ unsigned long long int *__float2ull_ru*(float x)
Convert a float to an unsigned 64-bit integer in round-up mode.

__device__ unsigned long long int *__float2ull_rz*(float x)
Convert a float to an unsigned 64-bit integer in round-towards-zero mode.

__device__ int *__float_as_int*(float x)
Reinterpret bits in a float as a signed integer.

__device__ unsigned int *__float_as_uint*(float x)
Reinterpret bits in a float as a unsigned integer.

__device__ double *__hiloint2double*(int hi, int lo)
Reinterpret high and low 32-bit integer values as a double.

__device__ double *__int2double_rn*(int x)
Convert a signed int to a double.

__device__ float *__int2float_rd*(int x)
Convert a signed integer to a float in round-down mode.

__device__ float *__int2float_rn*(int x)
Convert a signed integer to a float in round-to-nearest-even mode.

__device__ float *__int2float_ru*(int x)
Convert a signed integer to a float in round-up mode.

__device__ float *__int2float_rz*(int x)
Convert a signed integer to a float in round-towards-zero mode.

__device__ float *__int_as_float*(int x)
Reinterpret bits in an integer as a float.

__device__ double *__ll2double_rd*(long long int x)
Convert a signed 64-bit int to a double in round-down mode.

__device__ double *__ll2double_rn*(long long int x)
Convert a signed 64-bit int to a double in round-to-nearest-even mode.

__device__ double *__ll2double_ru*(long long int x)
Convert a signed 64-bit int to a double in round-up mode.

__device__ double *__ll2double_rz*(long long int x)
Convert a signed 64-bit int to a double in round-towards-zero mode.

__device__ float *__ll2float_rd*(long long int x)
Convert a signed integer to a float in round-down mode.

__device__ float *__ll2float_rn*(long long int x)
Convert a signed 64-bit integer to a float in round-to-nearest-even mode.

__device__ float *__ll2float_ru*(long long int x)
Convert a signed integer to a float in round-up mode.

__device__ float *__ll2float_rz*(long long int x)
Convert a signed integer to a float in round-towards-zero mode.

__device__ double *__longlong_as_double*(long long int x)
Reinterpret bits in a 64-bit signed integer as a double.

__device__ double *__uint2double_rn*(unsigned int x)
Convert an unsigned int to a double.

__device__ float *__uint2float_rd*(unsigned int x)
Convert an unsigned integer to a float in round-down mode.

__device__ float *__uint2float_rn*(unsigned int x)
Convert an unsigned integer to a float in round-to-nearest-even mode.

__device__ float *__uint2float_ru*(unsigned int x)
Convert an unsigned integer to a float in round-up mode.

__device__ float *__uint2float_rz*(unsigned int x)
Convert an unsigned integer to a float in round-towards-zero mode.

__device__ float *__uint_as_float*(unsigned int x)
Reinterpret bits in an unsigned integer as a float.

__device__ double *__ull2double_rd*(unsigned long long int x)
Convert an unsigned 64-bit int to a double in round-down mode.

__device__ double *__ull2double_rn*(unsigned long long int x)
Convert an unsigned 64-bit int to a double in round-to-nearest-even mode.

__device__ double *__ull2double_ru*(unsigned long long int x)
Convert an unsigned 64-bit int to a double in round-up mode.

__device__ double *__ull2double_rz*(unsigned long long int x)
Convert an unsigned 64-bit int to a double in round-towards-zero mode.

__device__ float *__ull2float_rd*(unsigned long long int x)
Convert an unsigned integer to a float in round-down mode.

__device__ float *__ull2float_rn*(unsigned long long int x)
Convert an unsigned integer to a float in round-to-nearest-even mode.

__device__ float *__ull2float_ru*(unsigned long long int x)
Convert an unsigned integer to a float in round-up mode.

__device__ float *__ull2float_rz*(unsigned long long int x)
Convert an unsigned integer to a float in round-towards-zero mode.

11.1. Functions

__device__ float **`__double2float_rd`**(double x)

Convert a double to a float in round-down mode.

Convert the double-precision floating-point value x to a single-precision floating-point value in round-down (to negative infinity) mode.

Returns

Returns converted value.

__device__ float **__double2float_rn**(double x)

Convert a double to a float in round-to-nearest-even mode.

Convert the double-precision floating-point value x to a single-precision floating-point value in round-to-nearest-even mode.

Returns

Returns converted value.

__device__ float **__double2float_ru**(double x)

Convert a double to a float in round-up mode.

Convert the double-precision floating-point value x to a single-precision floating-point value in round-up (to positive infinity) mode.

Returns

Returns converted value.

__device__ float **__double2float_rz**(double x)

Convert a double to a float in round-towards-zero mode.

Convert the double-precision floating-point value x to a single-precision floating-point value in round-towards-zero mode.

Returns

Returns converted value.

__device__ int **__double2hiint**(double x)

Reinterpret high 32 bits in a double as a signed integer.

Reinterpret the high 32 bits in the double-precision floating-point value x as a signed integer.

Returns

Returns reinterpreted value.

__device__ int **__double2int_rd**(double x)

Convert a double to a signed int in round-down mode.

Convert the double-precision floating-point value x to a signed integer value in round-down (to negative infinity) mode.

Note: When the floating-point input rounded to integral is outside the range of the return type, the behavior is undefined.

Returns

Returns converted value.

__device__ int **__double2int_rn**(double x)

Convert a double to a signed int in round-to-nearest-even mode.

Convert the double-precision floating-point value x to a signed integer value in round-to-nearest-even mode.

Note: When the floating-point input rounded to integral is outside the range of the return type, the behavior is undefined.

Returns

Returns converted value.

__device__ int **__double2int_ru**(double x)

Convert a double to a signed int in round-up mode.

Convert the double-precision floating-point value x to a signed integer value in round-up (to positive infinity) mode.

Note: When the floating-point input rounded to integral is outside the range of the return type, the behavior is undefined.

Returns

Returns converted value.

__device__ int **__double2int_rz**(double x)

Convert a double to a signed int in round-towards-zero mode.

Convert the double-precision floating-point value x to a signed integer value in round-towards-zero mode.

Note: When the floating-point input rounded to integral is outside the range of the return type, the behavior is undefined.

Returns

Returns converted value.

__device__ long long int **__double2ll_rd**(double x)

Convert a double to a signed 64-bit int in round-down mode.

Convert the double-precision floating-point value x to a signed 64-bit integer value in round-down (to negative infinity) mode.

Note: When the floating-point input rounded to integral is outside the range of the return type, the behavior is undefined.

Returns

Returns converted value.

__device__ long long int **__double2ll_rn**(double x)

Convert a double to a signed 64-bit int in round-to-nearest-even mode.

Convert the double-precision floating-point value x to a signed 64-bit integer value in round-to-nearest-even mode.

Note: When the floating-point input rounded to integral is outside the range of the return type, the behavior is undefined.

Returns

Returns converted value.

__device__ long long int **__double2ll_ru**(double x)

Convert a double to a signed 64-bit int in round-up mode.

Convert the double-precision floating-point value x to a signed 64-bit integer value in round-up (to positive infinity) mode.

Note: When the floating-point input rounded to integral is outside the range of the return type, the behavior is undefined.

Returns

Returns converted value.

__device__ long long int **__double2ll_rz**(double x)

Convert a double to a signed 64-bit int in round-towards-zero mode.

Convert the double-precision floating-point value x to a signed 64-bit integer value in round-towards-zero mode.

Note: When the floating-point input rounded to integral is outside the range of the return type, the behavior is undefined.

Returns

Returns converted value.

__device__ int **__double2loint**(double x)

Reinterpret low 32 bits in a double as a signed integer.

Reinterpret the low 32 bits in the double-precision floating-point value x as a signed integer.

Returns

Returns reinterpreted value.

__device__ unsigned int **__double2uint_rd**(double x)

Convert a double to an unsigned int in round-down mode.

Convert the double-precision floating-point value x to an unsigned integer value in round-down (to negative infinity) mode.

Note: When the floating-point input rounded to integral is outside the range of the return type, the behavior is undefined.

Returns

Returns converted value.

__device__ unsigned int **__double2uint_rn**(double x)

Convert a double to an unsigned int in round-to-nearest-even mode.

Convert the double-precision floating-point value x to an unsigned integer value in round-to-nearest-even mode.

Note: When the floating-point input rounded to integral is outside the range of the return type, the behavior is undefined.

Returns

Returns converted value.

__device__ unsigned int **__double2uint_ru**(double x)

Convert a double to an unsigned int in round-up mode.

Convert the double-precision floating-point value x to an unsigned integer value in round-up (to positive infinity) mode.

Note: When the floating-point input rounded to integral is outside the range of the return type, the behavior is undefined.

Returns

Returns converted value.

__device__ unsigned int **__double2uint_rz**(double x)

Convert a double to an unsigned int in round-towards-zero mode.

Convert the double-precision floating-point value x to an unsigned integer value in round-towards-zero mode.

Note: When the floating-point input rounded to integral is outside the range of the return type, the behavior is undefined.

Returns

Returns converted value.

__device__ unsigned long long int **__double2ull_rd**(double x)

Convert a double to an unsigned 64-bit int in round-down mode.

Convert the double-precision floating-point value x to an unsigned 64-bit integer value in round-down (to negative infinity) mode.

Note: When the floating-point input rounded to integral is outside the range of the return type, the behavior is undefined.

Returns

Returns converted value.

__device__ unsigned long long int **__double2ull_rn**(double x)

Convert a double to an unsigned 64-bit int in round-to-nearest-even mode.

Convert the double-precision floating-point value x to an unsigned 64-bit integer value in round-to-nearest-even mode.

Note: When the floating-point input rounded to integral is outside the range of the return type, the behavior is undefined.

Returns

Returns converted value.

__device__ unsigned long long int **__double2ull_ru**(double x)

Convert a double to an unsigned 64-bit int in round-up mode.

Convert the double-precision floating-point value x to an unsigned 64-bit integer value in round-up (to positive infinity) mode.

Note: When the floating-point input rounded to integral is outside the range of the return type, the behavior is undefined.

Returns

Returns converted value.

__device__ unsigned long long int **__double2ull_rz**(double x)

Convert a double to an unsigned 64-bit int in round-towards-zero mode.

Convert the double-precision floating-point value x to an unsigned 64-bit integer value in round-towards-zero mode.

Note: When the floating-point input rounded to integral is outside the range of the return type, the behavior is undefined.

Returns

Returns converted value.

__device__ long long int **__double_as_longlong**(double x)

Reinterpret bits in a double as a 64-bit signed integer.

Reinterpret the bits in the double-precision floating-point value x as a signed 64-bit integer.

Returns

Returns reinterpreted value.

__device__ int **__float2int_rd**(float x)

Convert a float to a signed integer in round-down mode.

Convert the single-precision floating-point value x to a signed integer in round-down (to negative infinity) mode.

Note: When the floating-point input rounded to integral is outside the range of the return type, the behavior is undefined.

Returns

Returns converted value.

__device__ int **__float2int_rn**(float x)

Convert a float to a signed integer in round-to-nearest-even mode.

Convert the single-precision floating-point value x to a signed integer in round-to-nearest-even mode.

Note: When the floating-point input rounded to integral is outside the range of the return type, the behavior is undefined.

Returns

Returns converted value.

__device__ int **__float2int_ru**(float)

Convert a float to a signed integer in round-up mode.

Convert the single-precision floating-point value x to a signed integer in round-up (to positive infinity) mode.

Note: When the floating-point input rounded to integral is outside the range of the return type, the behavior is undefined.

Returns

Returns converted value.

__device__ int **__float2int_rz**(float x)

Convert a float to a signed integer in round-towards-zero mode.

Convert the single-precision floating-point value x to a signed integer in round-towards-zero mode.

Note: When the floating-point input rounded to integral is outside the range of the return type, the behavior is undefined.

Returns

Returns converted value.

__device__ long long int **__float2ll_rd**(float x)

Convert a float to a signed 64-bit integer in round-down mode.

Convert the single-precision floating-point value x to a signed 64-bit integer in round-down (to negative infinity) mode.

Note: When the floating-point input rounded to integral is outside the range of the return type, the behavior is undefined.

Returns

Returns converted value.

__device__ long long int **__float2ll_rn**(float x)

Convert a float to a signed 64-bit integer in round-to-nearest-even mode.

Convert the single-precision floating-point value x to a signed 64-bit integer in round-to-nearest-even mode.

Note: When the floating-point input rounded to integral is outside the range of the return type, the behavior is undefined.

Returns

Returns converted value.

__device__ long long int **__float2ll_ru**(float x)

Convert a float to a signed 64-bit integer in round-up mode.

Convert the single-precision floating-point value x to a signed 64-bit integer in round-up (to positive infinity) mode.

Note: When the floating-point input rounded to integral is outside the range of the return type, the behavior is undefined.

Returns

Returns converted value.

__device__ long long int **__float2ll_rz**(float x)

Convert a float to a signed 64-bit integer in round-towards-zero mode.

Convert the single-precision floating-point value x to a signed 64-bit integer in round-towards-zero mode.

Note: When the floating-point input rounded to integral is outside the range of the return type, the behavior is undefined.

Returns

Returns converted value.

__device__ unsigned int **__float2uint_rd**(float x)

Convert a float to an unsigned integer in round-down mode.

Convert the single-precision floating-point value x to an unsigned integer in round-down (to negative infinity) mode.

Note: When the floating-point input rounded to integral is outside the range of the return type, the behavior is undefined.

Returns

Returns converted value.

__device__ unsigned int **`__float2uint_rn`**(float x)

Convert a float to an unsigned integer in round-to-nearest-even mode.

Convert the single-precision floating-point value x to an unsigned integer in round-to-nearest-even mode.

Note: When the floating-point input rounded to integral is outside the range of the return type, the behavior is undefined.

Returns

Returns converted value.

__device__ unsigned int **`__float2uint_ru`**(float x)

Convert a float to an unsigned integer in round-up mode.

Convert the single-precision floating-point value x to an unsigned integer in round-up (to positive infinity) mode.

Note: When the floating-point input rounded to integral is outside the range of the return type, the behavior is undefined.

Returns

Returns converted value.

__device__ unsigned int **`__float2uint_rz`**(float x)

Convert a float to an unsigned integer in round-towards-zero mode.

Convert the single-precision floating-point value x to an unsigned integer in round-towards-zero mode.

Note: When the floating-point input rounded to integral is outside the range of the return type, the behavior is undefined.

Returns

Returns converted value.

__device__ unsigned long long int **`__float2ull_rd`**(float x)

Convert a float to an unsigned 64-bit integer in round-down mode.

Convert the single-precision floating-point value x to an unsigned 64-bit integer in round-down (to negative infinity) mode.

Note: When the floating-point input rounded to integral is outside the range of the return type, the behavior is undefined.

Returns

Returns converted value.

__device__ unsigned long long int **__float2ull_rn**(float x)

Convert a float to an unsigned 64-bit integer in round-to-nearest-even mode.

Convert the single-precision floating-point value x to an unsigned 64-bit integer in round-to-nearest-even mode.

Note: When the floating-point input rounded to integral is outside the range of the return type, the behavior is undefined.

Returns
Returns converted value.

__device__ unsigned long long int **__float2ull_ru**(float x)

Convert a float to an unsigned 64-bit integer in round-up mode.

Convert the single-precision floating-point value x to an unsigned 64-bit integer in round-up (to positive infinity) mode.

Note: When the floating-point input rounded to integral is outside the range of the return type, the behavior is undefined.

Returns
Returns converted value.

__device__ unsigned long long int **__float2ull_rz**(float x)

Convert a float to an unsigned 64-bit integer in round-towards-zero mode.

Convert the single-precision floating-point value x to an unsigned 64-bit integer in round-towards-zero mode.

Note: When the floating-point input rounded to integral is outside the range of the return type, the behavior is undefined.

Returns
Returns converted value.

__device__ int **__float_as_int**(float x)

Reinterpret bits in a float as a signed integer.

Reinterpret the bits in the single-precision floating-point value x as a signed integer.

Returns
Returns reinterpreted value.

__device__ unsigned int **__float_as_uint**(float x)

Reinterpret bits in a float as a unsigned integer.

Reinterpret the bits in the single-precision floating-point value x as a unsigned integer.

Returns
Returns reinterpreted value.

__device__ double **__hiloint2double**(int hi, int lo)

Reinterpret high and low 32-bit integer values as a double.

Reinterpret the integer value of `hi` as the high 32 bits of a double-precision floating-point value and the integer value of `lo` as the low 32 bits of the same double-precision floating-point value.

Returns

Returns reinterpreted value.

__device__ double **__int2double_rn**(int x)

Convert a signed int to a double.

Convert the signed integer value x to a double-precision floating-point value.

Returns

Returns converted value.

__device__ float **__int2float_rd**(int x)

Convert a signed integer to a float in round-down mode.

Convert the signed integer value x to a single-precision floating-point value in round-down (to negative infinity) mode.

Returns

Returns converted value.

__device__ float **__int2float_rn**(int x)

Convert a signed integer to a float in round-to-nearest-even mode.

Convert the signed integer value x to a single-precision floating-point value in round-to-nearest-even mode.

Returns

Returns converted value.

__device__ float **__int2float_ru**(int x)

Convert a signed integer to a float in round-up mode.

Convert the signed integer value x to a single-precision floating-point value in round-up (to positive infinity) mode.

Returns

Returns converted value.

__device__ float **__int2float_rz**(int x)

Convert a signed integer to a float in round-towards-zero mode.

Convert the signed integer value x to a single-precision floating-point value in round-towards-zero mode.

Returns

Returns converted value.

__device__ float **__int_as_float**(int x)

Reinterpret bits in an integer as a float.

Reinterpret the bits in the signed integer value x as a single-precision floating-point value.

Returns

Returns reinterpreted value.

__device__ double **__ll2double_rd**(long long int x)

Convert a signed 64-bit int to a double in round-down mode.

Convert the signed 64-bit integer value x to a double-precision floating-point value in round-down (to negative infinity) mode.

Returns

Returns converted value.

__device__ double **__ll2double_rn**(long long int x)

Convert a signed 64-bit int to a double in round-to-nearest-even mode.

Convert the signed 64-bit integer value x to a double-precision floating-point value in round-to-nearest-even mode.

Returns

Returns converted value.

__device__ double **__ll2double_ru**(long long int x)

Convert a signed 64-bit int to a double in round-up mode.

Convert the signed 64-bit integer value x to a double-precision floating-point value in round-up (to positive infinity) mode.

Returns

Returns converted value.

__device__ double **__ll2double_rz**(long long int x)

Convert a signed 64-bit int to a double in round-towards-zero mode.

Convert the signed 64-bit integer value x to a double-precision floating-point value in round-towards-zero mode.

Returns

Returns converted value.

__device__ float **__ll2float_rd**(long long int x)

Convert a signed integer to a float in round-down mode.

Convert the signed integer value x to a single-precision floating-point value in round-down (to negative infinity) mode.

Returns

Returns converted value.

__device__ float **__ll2float_rn**(long long int x)

Convert a signed 64-bit integer to a float in round-to-nearest-even mode.

Convert the signed 64-bit integer value x to a single-precision floating-point value in round-to-nearest-even mode.

Returns

Returns converted value.

__device__ float **__ll2float_ru**(long long int x)

Convert a signed integer to a float in round-up mode.

Convert the signed integer value x to a single-precision floating-point value in round-up (to positive infinity) mode.

Returns

Returns converted value.

__device__ float **__ll2float_rz**(long long int x)

Convert a signed integer to a float in round-towards-zero mode.

Convert the signed integer value x to a single-precision floating-point value in round-towards-zero mode.

Returns

Returns converted value.

__device__ double **__longlong_as_double**(long long int x)

Reinterpret bits in a 64-bit signed integer as a double.

Reinterpret the bits in the 64-bit signed integer value x as a double-precision floating-point value.

Returns

Returns reinterpreted value.

__device__ double **__uint2double_rn**(unsigned int x)

Convert an unsigned int to a double.

Convert the unsigned integer value x to a double-precision floating-point value.

Returns

Returns converted value.

__device__ float **__uint2float_rd**(unsigned int x)

Convert an unsigned integer to a float in round-down mode.

Convert the unsigned integer value x to a single-precision floating-point value in round-down (to negative infinity) mode.

Returns

Returns converted value.

__device__ float **__uint2float_rn**(unsigned int x)

Convert an unsigned integer to a float in round-to-nearest-even mode.

Convert the unsigned integer value x to a single-precision floating-point value in round-to-nearest-even mode.

Returns

Returns converted value.

__device__ float **__uint2float_ru**(unsigned int x)

Convert an unsigned integer to a float in round-up mode.

Convert the unsigned integer value x to a single-precision floating-point value in round-up (to positive infinity) mode.

Returns

Returns converted value.

__device__ float **__uint2float_rz**(unsigned int x)

Convert an unsigned integer to a float in round-towards-zero mode.

Convert the unsigned integer value x to a single-precision floating-point value in round-towards-zero mode.

Returns

Returns converted value.

__device__ float **__uint_as_float**(unsigned int x)

Reinterpret bits in an unsigned integer as a float.

Reinterpret the bits in the unsigned integer value x as a single-precision floating-point value.

Returns

Returns reinterpreted value.

__device__ double **__ull2double_rd**(unsigned long long int x)

Convert an unsigned 64-bit int to a double in round-down mode.

Convert the unsigned 64-bit integer value x to a double-precision floating-point value in round-down (to negative infinity) mode.

Returns

Returns converted value.

__device__ double **__ull2double_rn**(unsigned long long int x)

Convert an unsigned 64-bit int to a double in round-to-nearest-even mode.

Convert the unsigned 64-bit integer value x to a double-precision floating-point value in round-to-nearest-even mode.

Returns

Returns converted value.

__device__ double **__ull2double_ru**(unsigned long long int x)

Convert an unsigned 64-bit int to a double in round-up mode.

Convert the unsigned 64-bit integer value x to a double-precision floating-point value in round-up (to positive infinity) mode.

Returns

Returns converted value.

__device__ double **__ull2double_rz**(unsigned long long int x)

Convert an unsigned 64-bit int to a double in round-towards-zero mode.

Convert the unsigned 64-bit integer value x to a double-precision floating-point value in round-towards-zero mode.

Returns

Returns converted value.

__device__ float **__ull2float_rd**(unsigned long long int x)

Convert an unsigned integer to a float in round-down mode.

Convert the unsigned integer value x to a single-precision floating-point value in round-down (to negative infinity) mode.

Returns

Returns converted value.

__device__ float **__ull2float_rn**(unsigned long long int x)

Convert an unsigned integer to a float in round-to-nearest-even mode.

Convert the unsigned integer value x to a single-precision floating-point value in round-to-nearest-even mode.

Returns

Returns converted value.

__device__ float **__ull2float_ru**(unsigned long long int x)

Convert an unsigned integer to a float in round-up mode.

Convert the unsigned integer value x to a single-precision floating-point value in round-up (to positive infinity) mode.

Returns

Returns converted value.

__device__ float **__ull2float_rz**(unsigned long long int x)

Convert an unsigned integer to a float in round-towards-zero mode.

Convert the unsigned integer value x to a single-precision floating-point value in round-towards-zero mode.

Returns

Returns converted value.

Chapter 12. Integer Mathematical Functions

This section describes integer mathematical functions.

To use these functions, you do not need to include any additional header file in your program.

Functions

__device__ long int *abs*(long int a)
Calculate the absolute value of the input `long int` argument.

__device__ int *abs*(int a)
Calculate the absolute value of the input `int` argument.

__device__ long long int *abs*(long long int a)
Calculate the absolute value of the input `long long int` argument.

__device__ long int *labs*(long int a)
Calculate the absolute value of the input `long int` argument.

__device__ long long int *llabs*(long long int a)
Calculate the absolute value of the input `long long int` argument.

__device__ long long int *llmax*(const long long int a, const long long int b)
Calculate the maximum value of the input `long long int` arguments.

__device__ long long int *llmin*(const long long int a, const long long int b)
Calculate the minimum value of the input `long long int` arguments.

__device__ unsigned long int *max*(const long int a, const unsigned long int b)
Calculate the maximum value of the input `long int` and `unsigned long int` arguments.

__device__ unsigned long long int *max*(const unsigned long long int a, const unsigned long long int b)
Calculate the maximum value of the input `unsigned long long int` arguments.

__device__ unsigned int *max*(const unsigned int a, const int b)
Calculate the maximum value of the input `unsigned int` and `int` arguments.

__device__ unsigned long long int *max*(const long long int a, const unsigned long long int b)
Calculate the maximum value of the input `long long int` and `unsigned long long int` arguments.

__device__ unsigned long int *max*(const unsigned long int a, const unsigned long int b)
Calculate the maximum value of the input `unsigned long int` arguments.

__device__ long long int *max*(const long long int a, const long long int b)
Calculate the maximum value of the input long long int arguments.

__device__ unsigned long long int *max*(const unsigned long long int a, const long long int b)
Calculate the maximum value of the input unsigned long long int and long long int arguments.

__device__ unsigned long int *max*(const unsigned long int a, const long int b)
Calculate the maximum value of the input unsigned long int and long int arguments.

__device__ long int *max*(const long int a, const long int b)
Calculate the maximum value of the input long int arguments.

__device__ int *max*(const int a, const int b)
Calculate the maximum value of the input int arguments.

__device__ unsigned int *max*(const unsigned int a, const unsigned int b)
Calculate the maximum value of the input unsigned int arguments.

__device__ unsigned int *max*(const int a, const unsigned int b)
Calculate the maximum value of the input int and unsigned int arguments.

__device__ unsigned long int *min*(const long int a, const unsigned long int b)
Calculate the minimum value of the input long int and unsigned long int arguments.

__device__ unsigned long long int *min*(const unsigned long long int a, const unsigned long long int b)
Calculate the minimum value of the input unsigned long long int arguments.

__device__ unsigned long long int *min*(const unsigned long long int a, const long long int b)
Calculate the minimum value of the input unsigned long long int and long long int arguments.

__device__ int *min*(const int a, const int b)
Calculate the minimum value of the input int arguments.

__device__ unsigned int *min*(const unsigned int a, const int b)
Calculate the minimum value of the input unsigned int and int arguments.

__device__ unsigned long long int *min*(const long long int a, const unsigned long long int b)
Calculate the minimum value of the input long long int and unsigned long long int arguments.

__device__ long long int *min*(const long long int a, const long long int b)
Calculate the minimum value of the input long long int arguments.

__device__ unsigned int *min*(const int a, const unsigned int b)
Calculate the minimum value of the input int and unsigned int arguments.

__device__ long int *min*(const long int a, const long int b)
Calculate the minimum value of the input long int arguments.

__device__ unsigned int *min*(const unsigned int a, const unsigned int b)
Calculate the minimum value of the input unsigned int arguments.

__device__ unsigned long int *min*(const unsigned long int a, const long int b)
Calculate the minimum value of the input unsigned long int and long int arguments.

__device__ unsigned long int *min*(const unsigned long int a, const unsigned long int b)
Calculate the minimum value of the input unsigned long int arguments.

__device__ unsigned long long int *ullmax*(const unsigned long long int a, const unsigned long long int b)
Calculate the maximum value of the input `unsigned long long int` arguments.

__device__ unsigned long long int *ullmin*(const unsigned long long int a, const unsigned long long int b)
Calculate the minimum value of the input `unsigned long long int` arguments.

__device__ unsigned int *umax*(const unsigned int a, const unsigned int b)
Calculate the maximum value of the input `unsigned int` arguments.

__device__ unsigned int *umin*(const unsigned int a, const unsigned int b)
Calculate the minimum value of the input `unsigned int` arguments.

12.1. Functions

__device__ long int **abs** (long int a)
Calculate the absolute value of the input `long int` argument.

Calculate the absolute value of the input argument a.

Returns
Returns the absolute value of the input argument.

- abs(`LONG_MIN`) is `Undefined`

__device__ int **abs** (int a)
Calculate the absolute value of the input `int` argument.

Calculate the absolute value of the input argument a.

Returns
Returns the absolute value of the input argument.

- abs(`INT_MIN`) is `Undefined`

__device__ long long int **abs** (long long int a)
Calculate the absolute value of the input `long long int` argument.

Calculate the absolute value of the input argument a.

Returns
Returns the absolute value of the input argument.

- abs(`LLONG_MIN`) is `Undefined`

__device__ long int **labs** (long int a)
Calculate the absolute value of the input `long int` argument.

Calculate the absolute value of the input argument a.

Returns
Returns the absolute value of the input argument.

- labs(`LONG_MIN`) is `Undefined`

__device__ long long int **llabs** (long long int a)
Calculate the absolute value of the input `long long int` argument.

Calculate the absolute value of the input argument a.

Returns

Returns the absolute value of the input argument.

- llabs(`LLONG_MIN`) is `Undefined`

__device__ long long int **llmax**(const long long int a, const long long int b)

Calculate the maximum value of the input `long long int` arguments.

Calculate the maximum value of the arguments a and b.

__device__ long long int **llmin**(const long long int a, const long long int b)

Calculate the minimum value of the input `long long int` arguments.

Calculate the minimum value of the arguments a and b.

__device__ unsigned long int **max**(const long int a, const unsigned long int b)

Calculate the maximum value of the input `long int` and `unsigned long int` arguments.

Calculate the maximum value of the arguments a and b, perform integer promotion first.

__device__ unsigned long long int **max**(const unsigned long long int a, const unsigned long long int b)

Calculate the maximum value of the input `unsigned long long int` arguments.

Calculate the maximum value of the arguments a and b.

__device__ unsigned int **max**(const unsigned int a, const int b)

Calculate the maximum value of the input `unsigned int` and `int` arguments.

Calculate the maximum value of the arguments a and b, perform integer promotion first.

__device__ unsigned long long int **max**(const long long int a, const unsigned long long int b)

Calculate the maximum value of the input `long long int` and `unsigned long long int` arguments.

Calculate the maximum value of the arguments a and b, perform integer promotion first.

__device__ unsigned long int **max**(const unsigned long int a, const unsigned long int b)

Calculate the maximum value of the input `unsigned long int` arguments.

Calculate the maximum value of the arguments a and b.

__device__ long long int **max**(const long long int a, const long long int b)

Calculate the maximum value of the input `long long int` arguments.

Calculate the maximum value of the arguments a and b.

__device__ unsigned long long int **max**(const unsigned long long int a, const long long int b)

Calculate the maximum value of the input `unsigned long long int` and `long long int` arguments.

Calculate the maximum value of the arguments a and b, perform integer promotion first.

__device__ unsigned long int **max**(const unsigned long int a, const long int b)

Calculate the maximum value of the input `unsigned long int` and `long int` arguments.

Calculate the maximum value of the arguments a and b, perform integer promotion first.

__device__ long int **max**(const long int a, const long int b)

Calculate the maximum value of the input `long int` arguments.

Calculate the maximum value of the arguments a and b.

__device__ int **max**(const int a, const int b)

Calculate the maximum value of the input `int` arguments.

Calculate the maximum value of the arguments a and b.

__device__ unsigned int **max**(const unsigned int a, const unsigned int b)

Calculate the maximum value of the input `unsigned int` arguments.

Calculate the maximum value of the arguments a and b.

__device__ unsigned int **max**(const int a, const unsigned int b)

Calculate the maximum value of the input `int` and `unsigned int` arguments.

Calculate the maximum value of the arguments a and b, perform integer promotion first.

__device__ unsigned long int **min**(const long int a, const unsigned long int b)

Calculate the minimum value of the input `long int` and `unsigned long int` arguments.

Calculate the minimum value of the arguments a and b, perform integer promotion first.

__device__ unsigned long long int **min**(const unsigned long long int a, const unsigned long long int b)

Calculate the minimum value of the input `unsigned long long int` arguments.

Calculate the minimum value of the arguments a and b.

__device__ unsigned long long int **min**(const unsigned long long int a, const long long int b)

Calculate the minimum value of the input `unsigned long long int` and `long long int` arguments.

Calculate the minimum value of the arguments a and b, perform integer promotion first.

__device__ int **min**(const int a, const int b)

Calculate the minimum value of the input `int` arguments.

Calculate the minimum value of the arguments a and b.

__device__ unsigned int **min**(const unsigned int a, const int b)

Calculate the minimum value of the input `unsigned int` and `int` arguments.

Calculate the minimum value of the arguments a and b, perform integer promotion first.

__device__ unsigned long long int **min**(const long long int a, const unsigned long long int b)

Calculate the minimum value of the input `long long int` and `unsigned long long int` arguments.

Calculate the minimum value of the arguments a and b, perform integer promotion first.

__device__ long long int **min**(const long long int a, const long long int b)

Calculate the minimum value of the input `long long int` arguments.

Calculate the minimum value of the arguments a and b.

__device__ unsigned int **min**(const int a, const unsigned int b)

Calculate the minimum value of the input `int` and `unsigned int` arguments.

Calculate the minimum value of the arguments a and b, perform integer promotion first.

__device__ long int **min**(const long int a, const long int b)

Calculate the minimum value of the input `long int` arguments.

Calculate the minimum value of the arguments a and b.

__device__ unsigned int **min**(const unsigned int a, const unsigned int b)

Calculate the minimum value of the input `unsigned int` arguments.

Calculate the minimum value of the arguments a and b.

__device__ unsigned long int **min**(const unsigned long int a, const long int b)

Calculate the minimum value of the input `unsigned long int` and `long int` arguments.

Calculate the minimum value of the arguments a and b, perform integer promotion first.

__device__ unsigned long int **min**(const unsigned long int a, const unsigned long int b)

Calculate the minimum value of the input `unsigned long int` arguments.

Calculate the minimum value of the arguments a and b.

__device__ unsigned long long int **ullmax**(const unsigned long long int a, const unsigned long long int b)

Calculate the maximum value of the input `unsigned long long int` arguments.

Calculate the maximum value of the arguments a and b.

__device__ unsigned long long int **ullmin**(const unsigned long long int a, const unsigned long long int b)

Calculate the minimum value of the input `unsigned long long int` arguments.

Calculate the minimum value of the arguments a and b.

__device__ unsigned int **umax**(const unsigned int a, const unsigned int b)

Calculate the maximum value of the input `unsigned int` arguments.

Calculate the maximum value of the arguments a and b.

__device__ unsigned int **umin**(const unsigned int a, const unsigned int b)

Calculate the minimum value of the input `unsigned int` arguments.

Calculate the minimum value of the arguments a and b.

Chapter 13. Integer Intrinsics

This section describes integer intrinsic functions.

All of these functions are supported in device code. For some of the functions, host-specific implementations are also provided. For example, see `__nv_bswap16()`. To use these functions, you do not need to include any additional header file in your program.

Functions

__device__ unsigned int *__brev*(unsigned int x)
Reverse the bit order of a 32-bit unsigned integer.

__device__ unsigned long long int *__brevll*(unsigned long long int x)
Reverse the bit order of a 64-bit unsigned integer.

__device__ unsigned int *__byte_perm*(unsigned int x, unsigned int y, unsigned int s)
Return selected bytes from two 32-bit unsigned integers.

__device__ int *__clz*(int x)
Return the number of consecutive high-order zero bits in a 32-bit integer.

__device__ int *__clzll*(long long int x)
Count the number of consecutive high-order zero bits in a 64-bit integer.

__device__ int *__dp2a_hi*(int srcA, int srcB, int c)
Two-way `signed int16` by `int8` dot product with `int32` accumulate, taking the upper half of the second input.

__device__ unsigned int *__dp2a_hi*(unsigned int srcA, unsigned int srcB, unsigned int c)
Two-way `unsigned int16` by `int8` dot product with `unsigned int32` accumulate, taking the upper half of the second input.

__device__ unsigned int *__dp2a_hi*(ushort2 srcA, uchar4 srcB, unsigned int c)
Two-way `unsigned int16` by `int8` dot product with `unsigned int32` accumulate, taking the upper half of the second input.

__device__ int *__dp2a_hi*(short2 srcA, char4 srcB, int c)
Two-way `signed int16` by `int8` dot product with `int32` accumulate, taking the upper half of the second input.

__device__ unsigned int *__dp2a_lo*(ushort2 srcA, uchar4 srcB, unsigned int c)
Two-way `unsigned int16` by `int8` dot product with `unsigned int32` accumulate, taking the lower half of the second input.

__device__ int *__dp2a_lo*(short2 srcA, char4 srcB, int c)
Two-way `signed int16` by `int8` dot product with `int32` accumulate, taking the lower half of the second input.

__device__ unsigned int *__dp2a_lo*(unsigned int srcA, unsigned int srcB, unsigned int c)
Two-way `unsigned int16` by `int8` dot product with `unsigned int32` accumulate, taking the lower half of the second input.

__device__ int *__dp2a_lo*(int srcA, int srcB, int c)
Two-way `signed int16` by `int8` dot product with `int32` accumulate, taking the lower half of the second input.

__device__ unsigned int *__dp4a*(uchar4 srcA, uchar4 srcB, unsigned int c)
Four-way `unsigned int8` dot product with `unsigned int32` accumulate.

__device__ unsigned int *__dp4a*(unsigned int srcA, unsigned int srcB, unsigned int c)
Four-way `unsigned int8` dot product with `unsigned int32` accumulate.

__device__ int *__dp4a*(int srcA, int srcB, int c)
Four-way `signed int8` dot product with `int32` accumulate.

__device__ int *__dp4a*(char4 srcA, char4 srcB, int c)
Four-way `signed int8` dot product with `int32` accumulate.

__device__ int *__ffs*(int x)
Find the position of the least significant bit set to 1 in a 32-bit integer.

__device__ int *__ffsll*(long long int x)
Find the position of the least significant bit set to 1 in a 64-bit integer.

__device__ unsigned *__fns*(unsigned mask, unsigned base, int offset)
Find the position of the n-th set to 1 bit in a 32-bit integer.

__device__ unsigned int *__funnelshift_l*(unsigned int lo, unsigned int hi, unsigned int shift)
Concatenate `hi` : `lo` , shift left by `shift` & 31 bits, return the most significant 32 bits.

__device__ unsigned int *__funnelshift_lc*(unsigned int lo, unsigned int hi, unsigned int shift)
Concatenate `hi` : `lo` , shift left by min(`shift` , 32) bits, return the most significant 32 bits.

__device__ unsigned int *__funnelshift_r*(unsigned int lo, unsigned int hi, unsigned int shift)
Concatenate `hi` : `lo` , shift right by `shift` & 31 bits, return the least significant 32 bits.

__device__ unsigned int *__funnelshift_rc*(unsigned int lo, unsigned int hi, unsigned int shift)
Concatenate `hi` : `lo` , shift right by min(`shift` , 32) bits, return the least significant 32 bits.

__device__ int *__hadd*(int x, int y)
Compute average of signed input arguments, avoiding overflow in the intermediate sum.

__device__ int *__mul24*(int x, int y)
Calculate the least significant 32 bits of the product of the least significant 24 bits of two integers.

__device__ long long int *__mul64hi*(long long int x, long long int y)
Calculate the most significant 64 bits of the product of the two 64-bit integers.

__device__ int *__mulhi*(int x, int y)
Calculate the most significant 32 bits of the product of the two 32-bit integers.

__host__ __device__ unsigned short *__nv_bswap16*(unsigned short x)
Reverse the order of bytes of the 16-bit unsigned integer.

__host__ __device__ unsigned int *__nv_bswap32*(unsigned int x)
Reverse the order of bytes of the 32-bit unsigned integer.

__host__ __device__ unsigned long long *__nv_bswap64*(unsigned long long x)
Reverse the order of bytes of the 64-bit unsigned integer.

__device__ int *__popc*(unsigned int x)
Count the number of bits that are set to 1 in a 32-bit integer.

__device__ int *__popcll*(unsigned long long int x)
Count the number of bits that are set to 1 in a 64-bit integer.

__device__ int *__rhadd*(int x, int y)
Compute rounded average of signed input arguments, avoiding overflow in the intermediate sum.

__device__ unsigned int *__sad*(int x, int y, unsigned int z)
Calculate $|x - y| + z$, the sum of absolute difference.

__device__ unsigned int *__uhadd*(unsigned int x, unsigned int y)
Compute average of unsigned input arguments, avoiding overflow in the intermediate sum.

__device__ unsigned int *__umul24*(unsigned int x, unsigned int y)
Calculate the least significant 32 bits of the product of the least significant 24 bits of two unsigned integers.

__device__ unsigned long long int *__umul64hi*(unsigned long long int x, unsigned long long int y)
Calculate the most significant 64 bits of the product of the two 64 unsigned bit integers.

__device__ unsigned int *__umulhi*(unsigned int x, unsigned int y)
Calculate the most significant 32 bits of the product of the two 32-bit unsigned integers.

__device__ unsigned int *__urhadd*(unsigned int x, unsigned int y)
Compute rounded average of unsigned input arguments, avoiding overflow in the intermediate sum.

__device__ unsigned int *__usad*(unsigned int x, unsigned int y, unsigned int z)
Calculate $|x - y| + z$, the sum of absolute difference.

13.1. Functions

__device__ unsigned int **`__brev`**(unsigned int x)

Reverse the bit order of a 32-bit unsigned integer.

Reverses the bit order of the 32-bit unsigned integer x.

Returns
Returns the bit-reversed value of x. i.e. bit N of the return value corresponds to bit 31-N of x.

__device__ unsigned long long int **`__brevll`**(unsigned long long int x)

Reverse the bit order of a 64-bit unsigned integer.

Reverses the bit order of the 64-bit unsigned integer x.

Returns
Returns the bit-reversed value of x. i.e. bit N of the return value corresponds to bit 63-N of x.

__device__ unsigned int **`__byte_perm`**(unsigned int x, unsigned int y, unsigned int s)

Return selected bytes from two 32-bit unsigned integers.

Create 8-byte source

- uint64_t `tmp64` = ((uint64_t)y << 32) | x;

Extract selector bits

- `selector0` = (s >> 0) & 0x7;
- `selector1` = (s >> 4) & 0x7;
- `selector2` = (s >> 8) & 0x7;
- `selector3` = (s >> 12) & 0x7;

Return 4 selected bytes from 8-byte source:

- `res`[07:00] = `tmp64`[`selector0`];
- `res`[15:08] = `tmp64`[`selector1`];
- `res`[23:16] = `tmp64`[`selector2`];
- `res`[31:24] = `tmp64`[`selector3`];

Returns

Returns a 32-bit integer consisting of four bytes from eight input bytes provided in the two input integers `x` and `y`, as specified by a selector, `s`.

__device__ int **`__clz`**(int x)

Return the number of consecutive high-order zero bits in a 32-bit integer.

Count the number of consecutive leading zero bits, starting at the most significant bit (bit 31) of x.

Returns

Returns a value between 0 and 32 inclusive representing the number of zero bits.

__device__ int **`__clzll`**(long long int x)

Count the number of consecutive high-order zero bits in a 64-bit integer.

Count the number of consecutive leading zero bits, starting at the most significant bit (bit 63) of x.

Returns

Returns a value between 0 and 64 inclusive representing the number of zero bits.

__device__ int **`__dp2a_hi`**(int srcA, int srcB, int c)

Two-way `signed int16` by `int8` dot product with `int32` accumulate, taking the upper half of the second input.

Extracts two packed 16-bit integers from `scrA` and two packed 8-bit integers from the upper 16 bits of `srcB`, then creates two pairwise 8x16 products and adds them together to a signed 32-bit integer `c`.

__device__ unsigned int **`__dp2a_hi`**(unsigned int srcA, unsigned int srcB, unsigned int c)

Two-way `unsigned int16` by `int8` dot product with `unsigned int32` accumulate, taking the upper half of the second input.

Extracts two packed 16-bit integers from `scrA` and two packed 8-bit integers from the upper 16 bits of `srcB`, then creates two pairwise 8x16 products and adds them together to an unsigned 32-bit integer `c`.

__device__ unsigned int **__dp2a_hi**(ushort2 srcA, uchar4 srcB, unsigned int c)

Two-way `unsigned int16` by `int8` dot product with `unsigned int32` accumulate, taking the upper half of the second input.

Takes two packed 16-bit integers from `scrA` vector and two packed 8-bit integers from the upper 16 bits of `srcB` vector, then creates two pairwise 8x16 products and adds them together to an unsigned 32-bit integer c.

__device__ int **__dp2a_hi**(short2 srcA, char4 srcB, int c)

Two-way `signed int16` by `int8` dot product with `int32` accumulate, taking the upper half of the second input.

Takes two packed 16-bit integers from `scrA` vector and two packed 8-bit integers from the upper 16 bits of `srcB` vector, then creates two pairwise 8x16 products and adds them together to a signed 32-bit integer c.

__device__ unsigned int **__dp2a_lo**(ushort2 srcA, uchar4 srcB, unsigned int c)

Two-way `unsigned int16` by `int8` dot product with `unsigned int32` accumulate, taking the lower half of the second input.

Takes two packed 16-bit integers from `scrA` vector and two packed 8-bit integers from the lower 16 bits of `srcB` vector, then creates two pairwise 8x16 products and adds them together to an unsigned 32-bit integer c.

__device__ int **__dp2a_lo**(short2 srcA, char4 srcB, int c)

Two-way `signed int16` by `int8` dot product with `int32` accumulate, taking the lower half of the second input.

Takes two packed 16-bit integers from `scrA` vector and two packed 8-bit integers from the lower 16 bits of `srcB` vector, then creates two pairwise 8x16 products and adds them together to a signed 32-bit integer c.

__device__ unsigned int **__dp2a_lo**(unsigned int srcA, unsigned int srcB, unsigned int c)

Two-way `unsigned int16` by `int8` dot product with `unsigned int32` accumulate, taking the lower half of the second input.

Extracts two packed 16-bit integers from `scrA` and two packed 8-bit integers from the lower 16 bits of `srcB`, then creates two pairwise 8x16 products and adds them together to an unsigned 32-bit integer c.

__device__ int **__dp2a_lo**(int srcA, int srcB, int c)

Two-way `signed int16` by `int8` dot product with `int32` accumulate, taking the lower half of the second input.

Extracts two packed 16-bit integers from `scrA` and two packed 8-bit integers from the lower 16 bits of `srcB`, then creates two pairwise 8x16 products and adds them together to a signed 32-bit integer c.

__device__ unsigned int **__dp4a**(uchar4 srcA, uchar4 srcB, unsigned int c)

Four-way `unsigned int8` dot product with `unsigned int32` accumulate.

Takes four pairs of packed byte-sized integers from `scrA` and `srcB` vectors, then creates four pairwise products and adds them together to an unsigned 32-bit integer c.

__device__ unsigned int **__dp4a**(unsigned int srcA, unsigned int srcB, unsigned int c)

Four-way `unsigned int8` dot product with `unsigned int32` accumulate.

Extracts four pairs of packed byte-sized integers from `scrA` and `srcB`, then creates four pairwise products and adds them together to an unsigned 32-bit integer c.

__device__ int **__dp4a**(int srcA, int srcB, int c)

Four-way `signed int8` dot product with `int32` accumulate.

Extracts four pairs of packed byte-sized integers from `scrA` and `srcB`, then creates four pairwise products and adds them together to a signed 32-bit integer `c`.

__device__ int **__dp4a**(char4 srcA, char4 srcB, int c)

Four-way `signed int8` dot product with `int32` accumulate.

Takes four pairs of packed byte-sized integers from `scrA` and `srcB` vectors, then creates four pairwise products and adds them together to a signed 32-bit integer `c`.

__device__ int **__ffs**(int x)

Find the position of the least significant bit set to 1 in a 32-bit integer.

Find the position of the first (least significant) bit set to 1 in `x`, where the least significant bit position is 1.

Returns

Returns a value between 0 and 32 inclusive representing the position of the first bit set.

- __ffs(0) returns 0.

__device__ int **__ffsll**(long long int x)

Find the position of the least significant bit set to 1 in a 64-bit integer.

Find the position of the first (least significant) bit set to 1 in `x`, where the least significant bit position is 1.

Returns

Returns a value between 0 and 64 inclusive representing the position of the first bit set.

- __ffsll(0) returns 0.

__device__ unsigned **__fns**(unsigned mask, unsigned base, int offset)

Find the position of the n-th set to 1 bit in a 32-bit integer.

Given a 32-bit value `mask` and an integer value `base` (between 0 and 31), find the n-th (given by `offset`) set bit in `mask` from the `base` bit. If not found, return 0xFFFFFFFF.

See also https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#integer-arithmetic-instructions-fns for more information.

Returns

Returns a value between 0 and 32 inclusive representing the position of the n-th set bit.

- parameter `base` must be <=31, otherwise behavior is undefined.

__device__ unsigned int **__funnelshift_l**(unsigned int lo, unsigned int hi, unsigned int shift)

Concatenate `hi` : `lo`, shift left by `shift` & 31 bits, return the most significant 32 bits.

Shift the 64-bit value formed by concatenating argument `lo` and `hi` left by the amount specified by the argument `shift`. Argument `lo` holds bits 31:0 and argument `hi` holds bits 63:32 of the 64-bit source value. The source is shifted left by the wrapped value of `shift` (`shift` & 31). The most significant 32-bits of the result are returned.

Returns

Returns the most significant 32 bits of the shifted 64-bit value.

__device__ unsigned int **`__funnelshift_lc`**(unsigned int lo, unsigned int hi, unsigned int shift)

Concatenate `hi` : `lo`, shift left by min(`shift`, 32) bits, return the most significant 32 bits.

Shift the 64-bit value formed by concatenating argument `lo` and `hi` left by the amount specified by the argument `shift`. Argument `lo` holds bits 31:0 and argument `hi` holds bits 63:32 of the 64-bit source value. The source is shifted left by the clamped value of `shift` (min(`shift`, 32)). The most significant 32-bits of the result are returned.

Returns

Returns the most significant 32 bits of the shifted 64-bit value.

__device__ unsigned int **`__funnelshift_r`**(unsigned int lo, unsigned int hi, unsigned int shift)

Concatenate `hi` : `lo`, shift right by `shift` & 31 bits, return the least significant 32 bits.

Shift the 64-bit value formed by concatenating argument `lo` and `hi` right by the amount specified by the argument `shift`. Argument `lo` holds bits 31:0 and argument `hi` holds bits 63:32 of the 64-bit source value. The source is shifted right by the wrapped value of `shift` (`shift` & 31). The least significant 32-bits of the result are returned.

Returns

Returns the least significant 32 bits of the shifted 64-bit value.

__device__ unsigned int **`__funnelshift_rc`**(unsigned int lo, unsigned int hi, unsigned int shift)

Concatenate `hi` : `lo`, shift right by min(`shift`, 32) bits, return the least significant 32 bits.

Shift the 64-bit value formed by concatenating argument `lo` and `hi` right by the amount specified by the argument `shift`. Argument `lo` holds bits 31:0 and argument `hi` holds bits 63:32 of the 64-bit source value. The source is shifted right by the clamped value of `shift` (min(`shift`, 32)). The least significant 32-bits of the result are returned.

Returns

Returns the least significant 32 bits of the shifted 64-bit value.

__device__ int **`__hadd`**(int x, int y)

Compute average of signed input arguments, avoiding overflow in the intermediate sum.

Compute average of signed input arguments x and y as (x + y) >> 1, avoiding overflow in the intermediate sum.

Returns

Returns a signed integer value representing the signed average value of the two inputs.

__device__ int **`__mul24`**(int x, int y)

Calculate the least significant 32 bits of the product of the least significant 24 bits of two integers.

Calculate the least significant 32 bits of the product of the least significant 24 bits of x and y. The high order 8 bits of x and y are ignored.

Returns

Returns the least significant 32 bits of the product x * y.

__device__ long long int **`__mul64hi`**(long long int x, long long int y)

Calculate the most significant 64 bits of the product of the two 64-bit integers.

Calculate the most significant 64 bits of the 128-bit product x * y, where x and y are 64-bit integers.

Returns

Returns the most significant 64 bits of the product x * y.

__device__ int **__mulhi**(int x, int y)

Calculate the most significant 32 bits of the product of the two 32-bit integers.

Calculate the most significant 32 bits of the 64-bit product x * y, where x and y are 32-bit integers.

Returns

Returns the most significant 32 bits of the product x * y.

__host__ __device__ unsigned short **__nv_bswap16**(unsigned short x)

Reverse the order of bytes of the 16-bit unsigned integer.

Reverse the order of bytes of x . Only supported in MSVC and other host compilers which define the `__GNUC__` macro, such as GCC and CLANG.

Returns

Returns x with the order of bytes reversed.

__host__ __device__ unsigned int **__nv_bswap32**(unsigned int x)

Reverse the order of bytes of the 32-bit unsigned integer.

Reverse the order of bytes of x . Only supported in MSVC and other host compilers which define the `__GNUC__` macro, such as GCC and CLANG.

Returns

Returns x with the order of bytes reversed.

__host__ __device__ unsigned long long **__nv_bswap64**(unsigned long long x)

Reverse the order of bytes of the 64-bit unsigned integer.

Reverse the order of bytes of x . Only supported in MSVC and other host compilers which define the `__GNUC__` macro, such as GCC and CLANG.

Returns

Returns x with the order of bytes reversed.

__device__ int **__popc**(unsigned int x)

Count the number of bits that are set to 1 in a 32-bit integer.

Count the number of bits that are set to 1 in x.

Returns

Returns a value between 0 and 32 inclusive representing the number of set bits.

__device__ int **__popcll**(unsigned long long int x)

Count the number of bits that are set to 1 in a 64-bit integer.

Count the number of bits that are set to 1 in x.

Returns

Returns a value between 0 and 64 inclusive representing the number of set bits.

__device__ int **__rhadd**(int x, int y)

Compute rounded average of signed input arguments, avoiding overflow in the intermediate sum.

Compute average of signed input arguments x and y as (x + y + 1) >> 1, avoiding overflow in the intermediate sum.

Returns

Returns a signed integer value representing the signed rounded average value of the two inputs.

__device__ unsigned int **__sad**(int x, int y, unsigned int z)

Calculate $|x - y| + z$, the sum of absolute difference.

Calculate $|x - y| + z$, the 32-bit sum of the third argument z plus and the absolute value of the difference between the first argument, x, and second argument, y.

Inputs x and y are signed 32-bit integers, input z is a 32-bit unsigned integer.

Returns

Returns $|x - y| + z$.

__device__ unsigned int **__uhadd**(unsigned int x, unsigned int y)

Compute average of unsigned input arguments, avoiding overflow in the intermediate sum.

Compute average of unsigned input arguments x and y as (x + y) >> 1, avoiding overflow in the intermediate sum.

Returns

Returns an unsigned integer value representing the unsigned average value of the two inputs.

__device__ unsigned int **__umul24**(unsigned int x, unsigned int y)

Calculate the least significant 32 bits of the product of the least significant 24 bits of two unsigned integers.

Calculate the least significant 32 bits of the product of the least significant 24 bits of x and y. The high order 8 bits of x and y are ignored.

Returns

Returns the least significant 32 bits of the product x * y.

__device__ unsigned long long int **__umul64hi**(unsigned long long int x, unsigned long long int y)

Calculate the most significant 64 bits of the product of the two 64 unsigned bit integers.

Calculate the most significant 64 bits of the 128-bit product x * y, where x and y are 64-bit unsigned integers.

Returns

Returns the most significant 64 bits of the product x * y.

__device__ unsigned int **__umulhi**(unsigned int x, unsigned int y)

Calculate the most significant 32 bits of the product of the two 32-bit unsigned integers.

Calculate the most significant 32 bits of the 64-bit product x * y, where x and y are 32-bit unsigned integers.

Returns

Returns the most significant 32 bits of the product x * y.

__device__ unsigned int **__urhadd**(unsigned int x, unsigned int y)

Compute rounded average of unsigned input arguments, avoiding overflow in the intermediate sum.

Compute average of unsigned input arguments x and y as (x + y + 1) >> 1, avoiding overflow in the intermediate sum.

Returns

Returns an unsigned integer value representing the unsigned rounded average value of the two inputs.

__device__ unsigned int **__usad**(unsigned int x, unsigned int y, unsigned int z)

Calculate $|x - y| + z$, the sum of absolute difference.

Calculate $|x - y| + z$, the 32-bit sum of the third argument z plus and the absolute value of the difference between the first argument, x, and second argument, y.

Inputs x, y, and z are unsigned 32-bit integers.

Returns

Returns $|x - y| + z$.

Chapter 14. SIMD Intrinsics

This section describes SIMD intrinsic functions that are only supported in device code.

To use these functions, you do not need to include any additional header file in your program.

Functions

__device__ unsigned int *__vabs2*(unsigned int a)
Computes per-halfword absolute value: |a|.

__device__ unsigned int *__vabs4*(unsigned int a)
Computes per-byte absolute value: |a|.

__device__ unsigned int *__vabsdiffs2*(unsigned int a, unsigned int b)
Computes per-halfword absolute difference of signed integer: |a - b|.

__device__ unsigned int *__vabsdiffs4*(unsigned int a, unsigned int b)
Computes per-byte absolute difference of signed integer: |a - b|.

__device__ unsigned int *__vabsdiffu2*(unsigned int a, unsigned int b)
Computes per-halfword absolute difference of unsigned integer: |a - b|.

__device__ unsigned int *__vabsdiffu4*(unsigned int a, unsigned int b)
Computes per-byte absolute difference of unsigned integer: |a - b|.

__device__ unsigned int *__vabsss2*(unsigned int a)
Computes per-halfword absolute value with signed saturation: |a|.

__device__ unsigned int *__vabsss4*(unsigned int a)
Computes per-byte absolute value with signed saturation: |a|.

__device__ unsigned int *__vadd2*(unsigned int a, unsigned int b)
Performs per-halfword (un)signed addition, with wrap-around: a + b.

__device__ unsigned int *__vadd4*(unsigned int a, unsigned int b)
Performs per-byte (un)signed addition: a + b.

__device__ unsigned int *__vaddss2*(unsigned int a, unsigned int b)
Performs per-halfword addition with signed saturation: a + b.

__device__ unsigned int *__vaddss4*(unsigned int a, unsigned int b)
Performs per-byte addition with signed saturation: a + b.

__device__ unsigned int *__vaddus2*(unsigned int a, unsigned int b)
Performs per-halfword addition with unsigned saturation: a + b.

__device__ unsigned int *__vaddus4*(unsigned int a, unsigned int b)
Performs per-byte addition with unsigned saturation: a + b.

__device__ unsigned int *__vavgs2*(unsigned int a, unsigned int b)
Performs per-halfword signed rounded average computation.

__device__ unsigned int *__vavgs4*(unsigned int a, unsigned int b)
Computes per-byte signed rounded average.

__device__ unsigned int *__vavgu2*(unsigned int a, unsigned int b)
Performs per-halfword unsigned rounded average computation.

__device__ unsigned int *__vavgu4*(unsigned int a, unsigned int b)
Performs per-byte unsigned rounded average.

__device__ unsigned int *__vcmpeq2*(unsigned int a, unsigned int b)
Performs per-halfword (un)signed comparison: a == b ? 0xffff : 0.

__device__ unsigned int *__vcmpeq4*(unsigned int a, unsigned int b)
Performs per-byte (un)signed comparison: a == b ? 0xff : 0.

__device__ unsigned int *__vcmpges2*(unsigned int a, unsigned int b)
Performs per-halfword signed comparison: a >= b ? 0xffff : 0.

__device__ unsigned int *__vcmpges4*(unsigned int a, unsigned int b)
Performs per-byte signed comparison: a >= b ? 0xff : 0.

__device__ unsigned int *__vcmpgeu2*(unsigned int a, unsigned int b)
Performs per-halfword unsigned comparison: a >= b ? 0xffff : 0.

__device__ unsigned int *__vcmpgeu4*(unsigned int a, unsigned int b)
Performs per-byte unsigned comparison: a >= b ? 0xff : 0.

__device__ unsigned int *__vcmpgts2*(unsigned int a, unsigned int b)
Performs per-halfword signed comparison: a > b ? 0xffff : 0.

__device__ unsigned int *__vcmpgts4*(unsigned int a, unsigned int b)
Performs per-byte signed comparison: a > b ? 0xff : 0.

__device__ unsigned int *__vcmpgtu2*(unsigned int a, unsigned int b)
Performs per-halfword unsigned comparison: a > b ? 0xffff : 0.

__device__ unsigned int *__vcmpgtu4*(unsigned int a, unsigned int b)
Performs per-byte unsigned comparison: a > b ? 0xff : 0.

__device__ unsigned int *__vcmples2*(unsigned int a, unsigned int b)
Performs per-halfword signed comparison: a <= b ? 0xffff : 0.

__device__ unsigned int *__vcmples4*(unsigned int a, unsigned int b)
Performs per-byte signed comparison: a <= b ? 0xff : 0.

__device__ unsigned int *__vcmpleu2*(unsigned int a, unsigned int b)
Performs per-halfword unsigned comparison: a <= b ? 0xffff : 0.

__device__ unsigned int *__vcmpleu4*(unsigned int a, unsigned int b)
Performs per-byte unsigned comparison: a <= b ? 0xff : 0.

__device__ unsigned int *__vcmplts2*(unsigned int a, unsigned int b)
Performs per-halfword signed comparison: a < b ? 0xffff : 0.

__device__ unsigned int *__vcmplts4*(unsigned int a, unsigned int b)
Performs per-byte signed comparison: a < b ? 0xff : 0.

__device__ unsigned int *__vcmpltu2*(unsigned int a, unsigned int b)
Performs per-halfword unsigned comparison: a < b ? 0xffff : 0.

__device__ unsigned int *__vcmpltu4*(unsigned int a, unsigned int b)
Performs per-byte unsigned comparison: a < b ? 0xff : 0.

__device__ unsigned int *__vcmpne2*(unsigned int a, unsigned int b)
Performs per-halfword (un)signed comparison: a != b ? 0xffff : 0.

__device__ unsigned int *__vcmpne4*(unsigned int a, unsigned int b)
Performs per-byte (un)signed comparison: a != b ? 0xff : 0.

__device__ unsigned int *__vhaddu2*(unsigned int a, unsigned int b)
Performs per-halfword unsigned average computation.

__device__ unsigned int *__vhaddu4*(unsigned int a, unsigned int b)
Computes per-byte unsigned average.

__host__ __device__ unsigned int *__viaddmax_s16x2*(const unsigned int a, const unsigned int b, const unsigned int c)
Performs per-halfword max(a + b, c)

__host__ __device__ unsigned int *__viaddmax_s16x2_relu*(const unsigned int a, const unsigned int b, const unsigned int c)
Performs per-halfword max(max(a + b, c), 0)

__host__ __device__ int *__viaddmax_s32*(const int a, const int b, const int c)
Computes max(a + b, c)

__host__ __device__ int *__viaddmax_s32_relu*(const int a, const int b, const int c)
Computes max(max(a + b, c), 0)

__host__ __device__ unsigned int *__viaddmax_u16x2*(const unsigned int a, const unsigned int b, const unsigned int c)
Performs per-halfword max(a + b, c)

__host__ __device__ unsigned int *__viaddmax_u32*(const unsigned int a, const unsigned int b, const unsigned int c)
Computes max(a + b, c)

__host__ __device__ unsigned int *__viaddmin_s16x2*(const unsigned int a, const unsigned int b, const unsigned int c)
Performs per-halfword min(a + b, c)

__host__ __device__ unsigned int *__viaddmin_s16x2_relu*(const unsigned int a, const unsigned int b, const unsigned int c)
Performs per-halfword max(min(a + b, c), 0)

__host__ __device__ int *__viaddmin_s32*(const int a, const int b, const int c)
Computes min(a + b, c)

__host__ __device__ int *__viaddmin_s32_relu*(const int a, const int b, const int c)
Computes max(min(a + b, c), 0)

__host__ __device__ unsigned int *__viaddmin_u16x2*(const unsigned int a, const unsigned int b, const unsigned int c)
Performs per-halfword min(a + b, c)

__host__ __device__ unsigned int *__viaddmin_u32*(const unsigned int a, const unsigned int b, const unsigned int c)
Computes min(a + b, c)

__host__ __device__ unsigned int *__vibmax_s16x2*(const unsigned int a, const unsigned int b, bool *const pred_hi, bool *const pred_lo)

Performs per-halfword max(a, b), also sets the value pointed to by pred_hi and pred_lo to the per-halfword result of (a >= b).

__host__ __device__ int *__vibmax_s32*(const int a, const int b, bool *const pred)
Computes max(a, b), also sets the value pointed to by pred to (a >= b).

__host__ __device__ unsigned int *__vibmax_u16x2*(const unsigned int a, const unsigned int b, bool *const pred_hi, bool *const pred_lo)
Performs per-halfword max(a, b), also sets the value pointed to by pred_hi and pred_lo to the per-halfword result of (a >= b).

__host__ __device__ unsigned int *__vibmax_u32*(const unsigned int a, const unsigned int b, bool *const pred)
Computes max(a, b), also sets the value pointed to by pred to (a >= b).

__host__ __device__ unsigned int *__vibmin_s16x2*(const unsigned int a, const unsigned int b, bool *const pred_hi, bool *const pred_lo)
Performs per-halfword min(a, b), also sets the value pointed to by pred_hi and pred_lo to the per-halfword result of (a <= b).

__host__ __device__ int *__vibmin_s32*(const int a, const int b, bool *const pred)
Computes min(a, b), also sets the value pointed to by pred to (a <= b).

__host__ __device__ unsigned int *__vibmin_u16x2*(const unsigned int a, const unsigned int b, bool *const pred_hi, bool *const pred_lo)
Performs per-halfword min(a, b), also sets the value pointed to by pred_hi and pred_lo to the per-halfword result of (a <= b).

__host__ __device__ unsigned int *__vibmin_u32*(const unsigned int a, const unsigned int b, bool *const pred)
Computes min(a, b), also sets the value pointed to by pred to (a <= b).

__host__ __device__ unsigned int *__vimax3_s16x2*(const unsigned int a, const unsigned int b, const unsigned int c)
Performs per-halfword max(max(a, b), c)

__host__ __device__ unsigned int *__vimax3_s16x2_relu*(const unsigned int a, const unsigned int b, const unsigned int c)
Performs per-halfword max(max(max(a, b), c), 0)

__host__ __device__ int *__vimax3_s32*(const int a, const int b, const int c)
Computes max(max(a, b), c)

__host__ __device__ int *__vimax3_s32_relu*(const int a, const int b, const int c)
Computes max(max(max(a, b), c), 0)

__host__ __device__ unsigned int *__vimax3_u16x2*(const unsigned int a, const unsigned int b, const unsigned int c)
Performs per-halfword max(max(a, b), c)

__host__ __device__ unsigned int *__vimax3_u32*(const unsigned int a, const unsigned int b, const unsigned int c)
Computes max(max(a, b), c)

__host__ __device__ unsigned int *__vimax_s16x2_relu*(const unsigned int a, const unsigned int b)
Performs per-halfword max(max(a, b), 0)

__host__ __device__ int *__vimax_s32_relu*(const int a, const int b)
Computes max(max(a, b), 0)

__host__ __device__ unsigned int *__vimin3_s16x2*(const unsigned int a, const unsigned int b, const unsigned int c)
Performs per-halfword min(min(a, b), c)

__host__ __device__ unsigned int *__vimin3_s16x2_relu*(const unsigned int a, const unsigned int b, const unsigned int c)
Performs per-halfword max(min(min(a, b), c), 0)

__host__ __device__ int *__vimin3_s32*(const int a, const int b, const int c)
Computes min(min(a, b), c)

__host__ __device__ int *__vimin3_s32_relu*(const int a, const int b, const int c)
Computes max(min(min(a, b), c), 0)

__host__ __device__ unsigned int *__vimin3_u16x2*(const unsigned int a, const unsigned int b, const unsigned int c)
Performs per-halfword min(min(a, b), c)

__host__ __device__ unsigned int *__vimin3_u32*(const unsigned int a, const unsigned int b, const unsigned int c)
Computes min(min(a, b), c)

__host__ __device__ unsigned int *__vimin_s16x2_relu*(const unsigned int a, const unsigned int b)
Performs per-halfword max(min(a, b), 0)

__host__ __device__ int *__vimin_s32_relu*(const int a, const int b)
Computes max(min(a, b), 0)

__device__ unsigned int *__vmaxs2*(unsigned int a, unsigned int b)
Performs per-halfword signed maximum computation.

__device__ unsigned int *__vmaxs4*(unsigned int a, unsigned int b)
Computes per-byte signed maximum.

__device__ unsigned int *__vmaxu2*(unsigned int a, unsigned int b)
Performs per-halfword unsigned maximum computation.

__device__ unsigned int *__vmaxu4*(unsigned int a, unsigned int b)
Computes per-byte unsigned maximum.

__device__ unsigned int *__vmins2*(unsigned int a, unsigned int b)
Performs per-halfword signed minimum computation.

__device__ unsigned int *__vmins4*(unsigned int a, unsigned int b)
Computes per-byte signed minimum.

__device__ unsigned int *__vminu2*(unsigned int a, unsigned int b)
Performs per-halfword unsigned minimum computation.

__device__ unsigned int *__vminu4*(unsigned int a, unsigned int b)
Computes per-byte unsigned minimum.

__device__ unsigned int *__vneg2*(unsigned int a)
Computes per-halfword negation.

__device__ unsigned int *__vneg4*(unsigned int a)
Performs per-byte negation.

__device__ unsigned int *__vnegss2*(unsigned int a)
Computes per-halfword negation with signed saturation.

__device__ unsigned int *__vnegss4*(unsigned int a)
Performs per-byte negation with signed saturation.

__device__ unsigned int *__vsads2*(unsigned int a, unsigned int b)
Performs per-halfword sum of absolute difference of signed.

__device__ unsigned int *__vsads4*(unsigned int a, unsigned int b)
Computes per-byte sum of abs difference of signed.

__device__ unsigned int *__vsadu2*(unsigned int a, unsigned int b)
Computes per-halfword sum of abs diff of unsigned.

__device__ unsigned int *__vsadu4*(unsigned int a, unsigned int b)
Computes per-byte sum of abs difference of unsigned.

__device__ unsigned int *__vseteq2*(unsigned int a, unsigned int b)
Performs per-halfword (un)signed comparison: returns 1 if both parts compare equal.

__device__ unsigned int *__vseteq4*(unsigned int a, unsigned int b)
Performs per-byte (un)signed comparison: returns 1 if all 4 pairs compare equal.

__device__ unsigned int *__vsetges2*(unsigned int a, unsigned int b)
Performs per-halfword signed comparison: returns 1 if both parts compare greater than or equal.

__device__ unsigned int *__vsetges4*(unsigned int a, unsigned int b)
Performs per-byte signed comparison: returns 1 if all 4 pairs compare greater than or equal.

__device__ unsigned int *__vsetgeu2*(unsigned int a, unsigned int b)
Performs per-halfword unsigned comparison: returns 1 if both parts compare greater than or equal.

__device__ unsigned int *__vsetgeu4*(unsigned int a, unsigned int b)
Performs per-byte unsigned comparison: returns 1 if all 4 pairs compare greater than or equal.

__device__ unsigned int *__vsetgts2*(unsigned int a, unsigned int b)
Performs per-halfword signed comparison: returns 1 if both parts compare greater than.

__device__ unsigned int *__vsetgts4*(unsigned int a, unsigned int b)
Performs per-byte signed comparison: returns 1 if all 4 pairs compare greater than.

__device__ unsigned int *__vsetgtu2*(unsigned int a, unsigned int b)
Performs per-halfword unsigned comparison: returns 1 if both parts compare greater than.

__device__ unsigned int *__vsetgtu4*(unsigned int a, unsigned int b)
Performs per-byte unsigned comparison: returns 1 if all 4 pairs compare greater than.

__device__ unsigned int *__vsetles2*(unsigned int a, unsigned int b)
Performs per-halfword unsigned comparison: returns 1 if both parts compare less than or equal.

__device__ unsigned int *__vsetles4*(unsigned int a, unsigned int b)
Performs per-byte signed comparison: returns 1 if all 4 pairs compare less than or equal.

__device__ unsigned int *__vsetleu2*(unsigned int a, unsigned int b)
Performs per-halfword signed comparison: returns 1 if both parts compare less than or equal.

__device__ unsigned int *__vsetleu4*(unsigned int a, unsigned int b)
Performs per-byte unsigned comparison: returns 1 if all 4 pairs compare less than or equal.

__device__ unsigned int *__vsetlts2*(unsigned int a, unsigned int b)
Performs per-halfword signed comparison: returns 1 if both parts compare less than.

__device__ unsigned int *__vsetlts4*(unsigned int a, unsigned int b)
Performs per-byte signed comparison: returns 1 if all 4 pairs compare less than.

__device__ unsigned int *__vsetltu2*(unsigned int a, unsigned int b)
Performs per-halfword unsigned comparison: returns 1 if both parts compare less than.

__device__ unsigned int *__vsetltu4*(unsigned int a, unsigned int b)
Performs per-byte unsigned comparison: returns 1 if all 4 pairs compare less than.

__device__ unsigned int *__vsetne2*(unsigned int a, unsigned int b)
Performs per-halfword (un)signed comparison: returns 1 if both parts compare not equal.

__device__ unsigned int *__vsetne4*(unsigned int a, unsigned int b)
Performs per-byte (un)signed comparison: returns 1 if all 4 pairs compare not equal.

__device__ unsigned int *__vsub2*(unsigned int a, unsigned int b)
Performs per-halfword (un)signed subtraction, with wrap-around: a - b.

__device__ unsigned int *__vsub4*(unsigned int a, unsigned int b)
Performs per-byte subtraction: a - b.

__device__ unsigned int *__vsubss2*(unsigned int a, unsigned int b)
Performs per-halfword (un)signed subtraction, with signed saturation: a - b.

__device__ unsigned int *__vsubss4*(unsigned int a, unsigned int b)
Performs per-byte subtraction with signed saturation: a - b.

__device__ unsigned int *__vsubus2*(unsigned int a, unsigned int b)
Performs per-halfword subtraction with unsigned saturation: a - b.

__device__ unsigned int *__vsubus4*(unsigned int a, unsigned int b)
Performs per-byte subtraction with unsigned saturation: a - b.

14.1. Functions

__device__ unsigned int **`__vabs2`**(unsigned int a)

Computes per-halfword absolute value: |a|.

Splits 4 bytes of argument into 2 parts, each consisting of 2 bytes, then computes absolute value for each of parts. Partial results are recombined and returned as unsigned int.

Returns
Returns computed value.

__device__ unsigned int **`__vabs4`**(unsigned int a)

Computes per-byte absolute value: |a|.

Splits argument by bytes. Computes absolute value of each byte. Partial results are recombined and returned as unsigned int.

Returns
Returns computed value.

__device__ unsigned int **`__vabsdiffs2`**(unsigned int a, unsigned int b)

Computes per-halfword absolute difference of signed integer: |a - b|.

Splits 4 bytes of each into 2 parts, each consisting of 2 bytes. For corresponding parts function computes absolute difference. Partial results are recombined and returned as unsigned int.

Returns
Returns computed value.

__device__ unsigned int **__vabsdiffs4**(unsigned int a, unsigned int b)

Computes per-byte absolute difference of signed integer: |a - b|.

Splits 4 bytes of each into 4 parts, each consisting of 1 byte. For corresponding parts function computes absolute difference. Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__device__ unsigned int **__vabsdiffu2**(unsigned int a, unsigned int b)

Computes per-halfword absolute difference of unsigned integer: |a - b|.

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts function computes absolute difference. Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__device__ unsigned int **__vabsdiffu4**(unsigned int a, unsigned int b)

Computes per-byte absolute difference of unsigned integer: |a - b|.

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts function computes absolute difference. Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__device__ unsigned int **__vabsss2**(unsigned int a)

Computes per-halfword absolute value with signed saturation: |a|.

Splits 4 bytes of argument into 2 parts, each consisting of 2 bytes, then computes absolute value with signed saturation for each of parts. Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__device__ unsigned int **__vabsss4**(unsigned int a)

Computes per-byte absolute value with signed saturation: |a|.

Splits 4 bytes of argument into 4 parts, each consisting of 1 byte, then computes absolute value with signed saturation for each of parts. Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__device__ unsigned int **__vadd2**(unsigned int a, unsigned int b)

Performs per-halfword (un)signed addition, with wrap-around: a + b.

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes, then performs unsigned addition on corresponding parts. Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__device__ unsigned int **__vadd4**(unsigned int a, unsigned int b)

Performs per-byte (un)signed addition: a + b.

Splits 'a' into 4 bytes, then performs unsigned addition on each of these bytes with the corresponding byte from 'b', ignoring overflow. Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__device__ unsigned int **__vaddss2**(unsigned int a, unsigned int b)

Performs per-halfword addition with signed saturation: a + b.

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes, then performs addition with signed saturation on corresponding parts. Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__device__ unsigned int **__vaddss4**(unsigned int a, unsigned int b)

Performs per-byte addition with signed saturation: a + b.

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte, then performs addition with signed saturation on corresponding parts. Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__device__ unsigned int **__vaddus2**(unsigned int a, unsigned int b)

Performs per-halfword addition with unsigned saturation: a + b.

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes, then performs addition with unsigned saturation on corresponding parts.

Returns

Returns computed value.

__device__ unsigned int **__vaddus4**(unsigned int a, unsigned int b)

Performs per-byte addition with unsigned saturation: a + b.

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte, then performs addition with unsigned saturation on corresponding parts.

Returns

Returns computed value.

__device__ unsigned int **__vavgs2**(unsigned int a, unsigned int b)

Performs per-halfword signed rounded average computation.

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes, then computes signed rounded average of corresponding parts. Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__device__ unsigned int **__vavgs4**(unsigned int a, unsigned int b)

Computes per-byte signed rounded average.

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. then computes signed rounded average of corresponding parts. Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__device__ unsigned int **__vavgu2**(unsigned int a, unsigned int b)

Performs per-halfword unsigned rounded average computation.

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes, then computes unsigned rounded average of corresponding parts. Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__device__ unsigned int **__vavgu4**(unsigned int a, unsigned int b)

Performs per-byte unsigned rounded average.

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. then computes unsigned rounded average of corresponding parts. Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__device__ unsigned int **__vcmpeq2**(unsigned int a, unsigned int b)

Performs per-halfword (un)signed comparison: a == b ? 0xffff : 0.

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts result is ffff if they are equal, and 0000 otherwise. For example __vcmpeq2(0x1234aba5, 0x1234aba6) returns 0xffff0000.

Returns

Returns 0xffff computed value.

__device__ unsigned int **__vcmpeq4**(unsigned int a, unsigned int b)

Performs per-byte (un)signed comparison: a == b ? 0xff : 0.

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts result is ff if they are equal, and 00 otherwise. For example __vcmpeq4(0x1234aba5, 0x1234aba6) returns 0xffffff00.

Returns

Returns 0xff if a = b, else returns 0.

__device__ unsigned int **__vcmpges2**(unsigned int a, unsigned int b)

Performs per-halfword signed comparison: a >= b ? 0xffff : 0.

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts result is ffff if 'a' part >= 'b' part, and 0000 otherwise. For example __vcmpges2(0x1234aba5, 0x1234aba6) returns 0xffff0000.

Returns

Returns 0xffff if a >= b, else returns 0.

__device__ unsigned int **__vcmpges4**(unsigned int a, unsigned int b)

Performs per-byte signed comparison: a >= b ? 0xff : 0.

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts result is ff if 'a' part >= 'b' part, and 00 otherwise. For example __vcmpges4(0x1234aba5, 0x1234aba6) returns 0xffffff00.

Returns

Returns 0xff if a >= b, else returns 0.

__device__ unsigned int **__vcmpgeu2**(unsigned int a, unsigned int b)

Performs per-halfword unsigned comparison: a >= b ? 0xffff : 0.

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts result is ffff if 'a' part >= 'b' part, and 0000 otherwise. For example __vcmpgeu2(0x1234aba5, 0x1234aba6) returns 0xffff0000.

Returns

Returns 0xffff if a >= b, else returns 0.

__device__ unsigned int **__vcmpgeu4**(unsigned int a, unsigned int b)

Performs per-byte unsigned comparison: a >= b ? 0xff : 0.

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts result is ff if 'a' part >= 'b' part, and 00 otherwise. For example __vcmpgeu4(0x1234aba5, 0x1234aba6) returns 0xffffff00.

Returns

Returns 0xff if a >= b, else returns 0.

__device__ unsigned int **__vcmpgts2**(unsigned int a, unsigned int b)

Performs per-halfword signed comparison: a > b ? 0xffff : 0.

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts result is ffff if 'a' part > 'b' part, and 0000 otherwise. For example __vcmpgts2(0x1234aba5, 0x1234aba6) returns 0x00000000.

Returns

Returns 0xffff if a > b, else returns 0.

__device__ unsigned int **__vcmpgts4**(unsigned int a, unsigned int b)

Performs per-byte signed comparison: a > b ? 0xff : 0.

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts result is ff if 'a' part > 'b' part, and 00 otherwise. For example __vcmpgts4(0x1234aba5, 0x1234aba6) returns 0x00000000.

Returns

Returns 0xff if a > b, else returns 0.

__device__ unsigned int **__vcmpgtu2**(unsigned int a, unsigned int b)

Performs per-halfword unsigned comparison: a > b ? 0xffff : 0.

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts result is ffff if 'a' part > 'b' part, and 0000 otherwise. For example __vcmpgtu2(0x1234aba5, 0x1234aba6) returns 0x00000000.

Returns

Returns 0xffff if a > b, else returns 0.

__device__ unsigned int **__vcmpgtu4**(unsigned int a, unsigned int b)

Performs per-byte unsigned comparison: a > b ? 0xff : 0.

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts result is ff if 'a' part > 'b' part, and 00 otherwise. For example __vcmpgtu4(0x1234aba5, 0x1234aba6) returns 0x00000000.

Returns

Returns 0xff if a > b, else returns 0.

__device__ unsigned int **__vcmples2**(unsigned int a, unsigned int b)

Performs per-halfword signed comparison: a <= b ? 0xffff : 0.

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts result is ffff if 'a' part <= 'b' part, and 0000 otherwise. For example __vcmples2(0x1234aba5, 0x1234aba6) returns 0xffffffff.

Returns

Returns 0xffff if a <= b, else returns 0.

__device__ unsigned int **__vcmples4**(unsigned int a, unsigned int b)

Performs per-byte signed comparison: a <= b ? 0xff : 0.

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts result is ff if 'a' part <= 'b' part, and 00 otherwise. For example __vcmples4(0x1234aba5, 0x1234aba6) returns 0xffffffff.

Returns

Returns 0xff if a <= b, else returns 0.

__device__ unsigned int **__vcmpleu2**(unsigned int a, unsigned int b)

Performs per-halfword unsigned comparison: a <= b ? 0xffff : 0.

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts result is ffff if 'a' part <= 'b' part, and 0000 otherwise. For example __vcmpleu2(0x1234aba5, 0x1234aba6) returns 0xffffffff.

Returns

Returns 0xffff if a <= b, else returns 0.

__device__ unsigned int **__vcmpleu4**(unsigned int a, unsigned int b)

Performs per-byte unsigned comparison: a <= b ? 0xff : 0.

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts result is ff if 'a' part <= 'b' part, and 00 otherwise. For example __vcmpleu4(0x1234aba5, 0x1234aba6) returns 0xffffffff.

Returns

Returns 0xff if a <= b, else returns 0.

__device__ unsigned int **__vcmplts2**(unsigned int a, unsigned int b)

Performs per-halfword signed comparison: a < b ? 0xffff : 0.

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts result is ffff if 'a' part < 'b' part, and 0000 otherwise. For example __vcmplts2(0x1234aba5, 0x1234aba6) returns 0x0000ffff.

Returns

Returns 0xffff if a < b, else returns 0.

__device__ unsigned int **__vcmplts4**(unsigned int a, unsigned int b)

Performs per-byte signed comparison: a < b ? 0xff : 0.

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts result is ff if 'a' part < 'b' part, and 00 otherwise. For example __vcmplts4(0x1234aba5, 0x1234aba6) returns 0x000000ff.

Returns

Returns 0xff if a < b, else returns 0.

__device__ unsigned int **__vcmpltu2**(unsigned int a, unsigned int b)

Performs per-halfword unsigned comparison: a < b ? 0xffff : 0.

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts result is ffff if 'a' part < 'b' part, and 0000 otherwise. For example __vcmpltu2(0x1234aba5, 0x1234aba6) returns 0x0000ffff.

Returns

Returns 0xffff if a < b, else returns 0.

__device__ unsigned int **__vcmpltu4**(unsigned int a, unsigned int b)

Performs per-byte unsigned comparison: a < b ? 0xff : 0.

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts result is ff if 'a' part < 'b' part, and 00 otherwise. For example __vcmpltu4(0x1234aba5, 0x1234aba6) returns 0x000000ff.

Returns

Returns 0xff if a < b, else returns 0.

__device__ unsigned int **__vcmpne2**(unsigned int a, unsigned int b)

Performs per-halfword (un)signed comparison: a != b ? 0xffff : 0.

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts result is ffff if 'a' part != 'b' part, and 0000 otherwise. For example __vcmplts2(0x1234aba5, 0x1234aba6) returns 0x0000ffff.

Returns

Returns 0xffff if a != b, else returns 0.

__device__ unsigned int **__vcmpne4**(unsigned int a, unsigned int b)

Performs per-byte (un)signed comparison: a != b ? 0xff : 0.

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts result is ff if 'a' part != 'b' part, and 00 otherwise. For example __vcmplts4(0x1234aba5, 0x1234aba6) returns 0x000000ff.

Returns

Returns 0xff if a != b, else returns 0.

__device__ unsigned int **__vhaddu2**(unsigned int a, unsigned int b)

Performs per-halfword unsigned average computation.

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes, then computes unsigned average of corresponding parts. Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__device__ unsigned int **__vhaddu4**(unsigned int a, unsigned int b)

Computes per-byte unsigned average.

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. then computes unsigned average of corresponding parts. Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__host__ __device__ unsigned int **__viaddmax_s16x2**(const unsigned int a, const unsigned int b, const unsigned int c)

Performs per-halfword max(a + b, c)

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. These 2 byte parts are interpreted as signed shorts. For corresponding parts function performs an add and compare: max(a_part + b_part), c_part) Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__host__ __device__ unsigned int **`__viaddmax_s16x2_relu`**(const unsigned int a, const unsigned int b, const unsigned int c)

Performs per-halfword max(max(a + b, c), 0)

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. These 2 byte parts are interpreted as signed shorts. For corresponding parts function performs an add, followed by a max with relu: max(max(a_part + b_part), c_part), 0) Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__host__ __device__ int **`__viaddmax_s32`**(const int a, const int b, const int c)

Computes max(a + b, c)

Calculates the sum of signed integers a and b and takes the max with c.

Returns

Returns computed value.

__host__ __device__ int **`__viaddmax_s32_relu`**(const int a, const int b, const int c)

Computes max(max(a + b, c), 0)

Calculates the sum of signed integers a and b and takes the max with c. If the result is less than `0` then `0` is returned.

Returns

Returns computed value.

__host__ __device__ unsigned int **`__viaddmax_u16x2`**(const unsigned int a, const unsigned int b, const unsigned int c)

Performs per-halfword max(a + b, c)

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. These 2 byte parts are interpreted as unsigned shorts. For corresponding parts function performs an add and compare: max(a_part + b_part), c_part) Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__host__ __device__ unsigned int **`__viaddmax_u32`**(const unsigned int a, const unsigned int b, const unsigned int c)

Computes max(a + b, c)

Calculates the sum of unsigned integers a and b and takes the max with c.

Returns

Returns computed value.

__host__ __device__ unsigned int **__viaddmin_s16x2**(const unsigned int a, const unsigned int b, const unsigned int c)

Performs per-halfword min(a + b, c)

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. These 2 byte parts are interpreted as signed shorts. For corresponding parts function performs an add and compare: min(a_part + b_part), c_part) Partial results are recombined and returned as unsigned int.

Returns
Returns computed value.

__host__ __device__ unsigned int **__viaddmin_s16x2_relu**(const unsigned int a, const unsigned int b, const unsigned int c)

Performs per-halfword max(min(a + b, c), 0)

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. These 2 byte parts are interpreted as signed shorts. For corresponding parts function performs an add, followed by a min with relu: max(min(a_part + b_part), c_part), 0) Partial results are recombined and returned as unsigned int.

Returns
Returns computed value.

__host__ __device__ int **__viaddmin_s32**(const int a, const int b, const int c)

Computes min(a + b, c)

Calculates the sum of signed integers a and b and takes the min with c.

Returns
Returns computed value.

__host__ __device__ int **__viaddmin_s32_relu**(const int a, const int b, const int c)

Computes max(min(a + b, c), 0)

Calculates the sum of signed integers a and b and takes the min with c. If the result is less than `0` then `0` is returned.

Returns
Returns computed value.

__host__ __device__ unsigned int **__viaddmin_u16x2**(const unsigned int a, const unsigned int b, const unsigned int c)

Performs per-halfword min(a + b, c)

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. These 2 byte parts are interpreted as unsigned shorts. For corresponding parts function performs an add and compare: min(a_part + b_part), c_part) Partial results are recombined and returned as unsigned int.

Returns
Returns computed value.

__host__ __device__ unsigned int **__viaddmin_u32**(const unsigned int a, const unsigned int b, const unsigned int c)

Computes min(a + b, c)

Calculates the sum of unsigned integers a and b and takes the min with c.

Returns
Returns computed value.

__host__ __device__ unsigned int **__vibmax_s16x2**(const unsigned int a, const unsigned int b, bool *const pred_hi, bool *const pred_lo)

Performs per-halfword max(a, b), also sets the value pointed to by pred_hi and pred_lo to the per-halfword result of (a >= b).

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. These 2 byte parts are interpreted as signed shorts. For corresponding parts function performs a maximum (= max(a_part, b_part)). Partial results are recombined and returned as unsigned int. Sets the value pointed to by `pred_hi` to the value (a_high_part >= b_high_part). Sets the value pointed to by `pred_lo` to the value (a_low_part >= b_low_part).

Returns
Returns computed values.

__host__ __device__ int **__vibmax_s32**(const int a, const int b, bool *const pred)

Computes max(a, b), also sets the value pointed to by pred to (a >= b).

Calculates the maximum of a and b of two signed ints. Also sets the value pointed to by `pred` to the value (a >= b).

Returns
Returns computed values.

__host__ __device__ unsigned int **__vibmax_u16x2**(const unsigned int a, const unsigned int b, bool *const pred_hi, bool *const pred_lo)

Performs per-halfword max(a, b), also sets the value pointed to by pred_hi and pred_lo to the per-halfword result of (a >= b).

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. These 2 byte parts are interpreted as unsigned shorts. For corresponding parts function performs a maximum (= max(a_part, b_part)). Partial results are recombined and returned as unsigned int. Sets the value pointed to by `pred_hi` to the value (a_high_part >= b_high_part). Sets the value pointed to by `pred_lo` to the value (a_low_part >= b_low_part).

Returns
Returns computed values.

__host__ __device__ unsigned int **__vibmax_u32**(const unsigned int a, const unsigned int b, bool *const pred)

Computes max(a, b), also sets the value pointed to by pred to (a >= b).

Calculates the maximum of a and b of two unsigned ints. Also sets the value pointed to by `pred` to the value (a >= b).

Returns
Returns computed values.

__host__ __device__ unsigned int **__vibmin_s16x2**(const unsigned int a, const unsigned int b, bool *const pred_hi, bool *const pred_lo)

Performs per-halfword min(a, b), also sets the value pointed to by pred_hi and pred_lo to the per-halfword result of (a <= b).

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. These 2 byte parts are interpreted as signed shorts. For corresponding parts function performs a maximum (= max(a_part, b_part)). Partial results are recombined and returned as unsigned int. Sets the value pointed to by `pred_hi` to the value (a_high_part <= b_high_part). Sets the value pointed to by `pred_lo` to the value (a_low_part <= b_low_part).

Returns
Returns computed values.

__host__ __device__ int **__vibmin_s32**(const int a, const int b, bool *const pred)

Computes min(a, b), also sets the value pointed to by pred to (a <= b).

Calculates the minimum of a and b of two signed ints. Also sets the value pointed to by `pred` to the value (a <= b).

Returns

Returns computed values.

__host__ __device__ unsigned int **__vibmin_u16x2**(const unsigned int a, const unsigned int b, bool *const pred_hi, bool *const pred_lo)

Performs per-halfword min(a, b), also sets the value pointed to by pred_hi and pred_lo to the per-halfword result of (a <= b).

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. These 2 byte parts are interpreted as unsigned shorts. For corresponding parts function performs a maximum (= max(a_part, b_part)). Partial results are recombined and returned as unsigned int. Sets the value pointed to by `pred_hi` to the value (a_high_part <= b_high_part). Sets the value pointed to by `pred_lo` to the value (a_low_part <= b_low_part).

Returns

Returns computed values.

__host__ __device__ unsigned int **__vibmin_u32**(const unsigned int a, const unsigned int b, bool *const pred)

Computes min(a, b), also sets the value pointed to by pred to (a <= b).

Calculates the minimum of a and b of two unsigned ints. Also sets the value pointed to by `pred` to the value (a <= b).

Returns

Returns computed values.

__host__ __device__ unsigned int **__vimax3_s16x2**(const unsigned int a, const unsigned int b, const unsigned int c)

Performs per-halfword max(max(a, b), c)

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. These 2 byte parts are interpreted as signed shorts. For corresponding parts function performs a 3-way max (= max(max(a_part, b_part), c_part)). Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__host__ __device__ unsigned int **__vimax3_s16x2_relu**(const unsigned int a, const unsigned int b, const unsigned int c)

Performs per-halfword max(max(max(a, b), c), 0)

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. These 2 byte parts are interpreted as signed shorts. For corresponding parts function performs a three-way max with relu (= max(a_part, b_part, c_part, 0)). Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__host__ __device__ int **__vimax3_s32**(const int a, const int b, const int c)

Computes max(max(a, b), c)

Calculates the 3-way max of signed integers a, b and c.

Returns

Returns computed value.

__host__ __device__ int **__vimax3_s32_relu**(const int a, const int b, const int c)

Computes max(max(max(a, b), c), 0)

Calculates the maximum of three signed ints, if this is less than `0` then `0` is returned.

Returns

Returns computed value.

__host__ __device__ unsigned int **__vimax3_u16x2**(const unsigned int a, const unsigned int b, const unsigned int c)

Performs per-halfword max(max(a, b), c)

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. These 2 byte parts are interpreted as unsigned shorts. For corresponding parts function performs a 3-way max (= max(max(a_part, b_part), c_part)). Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__host__ __device__ unsigned int **__vimax3_u32**(const unsigned int a, const unsigned int b, const unsigned int c)

Computes max(max(a, b), c)

Calculates the 3-way max of unsigned integers a, b and c.

Returns

Returns computed value.

__host__ __device__ unsigned int **__vimax_s16x2_relu**(const unsigned int a, const unsigned int b)

Performs per-halfword max(max(a, b), 0)

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. These 2 byte parts are interpreted as signed shorts. For corresponding parts function performs a max with relu (= max(a_part, b_part, 0)). Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__host__ __device__ int **__vimax_s32_relu**(const int a, const int b)

Computes max(max(a, b), 0)

Calculates the maximum of a and b of two signed ints, if this is less than `0` then `0` is returned.

Returns

Returns computed value.

__host__ __device__ unsigned int **__vimin3_s16x2**(const unsigned int a, const unsigned int b, const unsigned int c)

Performs per-halfword min(min(a, b), c)

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. These 2 byte parts are interpreted as signed shorts. For corresponding parts function performs a 3-way min (= min(min(a_part, b_part), c_part)). Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__host__ __device__ unsigned int **__vimin3_s16x2_relu**(const unsigned int a, const unsigned int b, const unsigned int c)

Performs per-halfword max(min(min(a, b), c), 0)

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. These 2 byte parts are interpreted as signed shorts. For corresponding parts function performs a three-way min with relu (= max(min(a_part, b_part, c_part), 0)). Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__host__ __device__ int **__vimin3_s32**(const int a, const int b, const int c)

Computes min(min(a, b), c)

Calculates the 3-way min of signed integers a, b and c.

Returns

Returns computed value.

__host__ __device__ int **__vimin3_s32_relu**(const int a, const int b, const int c)

Computes max(min(min(a, b), c), 0)

Calculates the minimum of three signed ints, if this is less than `0` then `0` is returned.

Returns

Returns computed value.

__host__ __device__ unsigned int **__vimin3_u16x2**(const unsigned int a, const unsigned int b, const unsigned int c)

Performs per-halfword min(min(a, b), c)

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. These 2 byte parts are interpreted as unsigned shorts. For corresponding parts function performs a 3-way min (= min(min(a_part, b_part), c_part)). Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__host__ __device__ unsigned int **__vimin3_u32**(const unsigned int a, const unsigned int b, const unsigned int c)

Computes min(min(a, b), c)

Calculates the 3-way min of unsigned integers a, b and c.

Returns

Returns computed value.

__host__ __device__ unsigned int **__vimin_s16x2_relu**(const unsigned int a, const unsigned int b)

Performs per-halfword max(min(a, b), 0)

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. These 2 byte parts are interpreted as signed shorts. For corresponding parts function performs a min with relu (= max(min(a_part, b_part), 0)). Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__host__ __device__ int **__vimin_s32_relu**(const int a, const int b)

Computes max(min(a, b), 0)

Calculates the minimum of a and b of two signed ints, if this is less than `0` then `0` is returned.

Returns

Returns computed value.

__device__ unsigned int **__vmaxs2**(unsigned int a, unsigned int b)

Performs per-halfword signed maximum computation.

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts function computes signed maximum. Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__device__ unsigned int **__vmaxs4**(unsigned int a, unsigned int b)

Computes per-byte signed maximum.

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts function computes signed maximum. Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__device__ unsigned int **__vmaxu2**(unsigned int a, unsigned int b)

Performs per-halfword unsigned maximum computation.

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts function computes unsigned maximum. Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__device__ unsigned int **__vmaxu4**(unsigned int a, unsigned int b)

Computes per-byte unsigned maximum.

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts function computes unsigned maximum. Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__device__ unsigned int **__vmins2**(unsigned int a, unsigned int b)

Performs per-halfword signed minimum computation.

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts function computes signed minimum. Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__device__ unsigned int **__vmins4**(unsigned int a, unsigned int b)

Computes per-byte signed minimum.

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts function computes signed minimum. Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__device__ unsigned int **__vminu2**(unsigned int a, unsigned int b)

Performs per-halfword unsigned minimum computation.

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts function computes unsigned minimum. Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__device__ unsigned int **__vminu4**(unsigned int a, unsigned int b)

Computes per-byte unsigned minimum.

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts function computes unsigned minimum. Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__device__ unsigned int **__vneg2**(unsigned int a)

Computes per-halfword negation.

Splits 4 bytes of argument into 2 parts, each consisting of 2 bytes. For each part function computes negation. Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__device__ unsigned int **__vneg4**(unsigned int a)

Performs per-byte negation.

Splits 4 bytes of argument into 4 parts, each consisting of 1 byte. For each part function computes negation. Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__device__ unsigned int **__vnegss2**(unsigned int a)

Computes per-halfword negation with signed saturation.

Splits 4 bytes of argument into 2 parts, each consisting of 2 bytes. For each part function computes negation. Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__device__ unsigned int **__vnegss4**(unsigned int a)

Performs per-byte negation with signed saturation.

Splits 4 bytes of argument into 4 parts, each consisting of 1 byte. For each part function computes negation. Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__device__ unsigned int **__vsads2**(unsigned int a, unsigned int b)

Performs per-halfword sum of absolute difference of signed.

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts function computes absolute difference and sum it up. Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__device__ unsigned int **__vsads4**(unsigned int a, unsigned int b)

Computes per-byte sum of abs difference of signed.

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts function computes absolute difference and sum it up. Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__device__ unsigned int **__vsadu2**(unsigned int a, unsigned int b)

Computes per-halfword sum of abs diff of unsigned.

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts function computes absolute differences and returns sum of those differences.

Returns

Returns computed value.

__device__ unsigned int **__vsadu4**(unsigned int a, unsigned int b)

Computes per-byte sum of abs difference of unsigned.

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts function computes absolute differences and returns sum of those differences.

Returns

Returns computed value.

__device__ unsigned int **__vseteq2**(unsigned int a, unsigned int b)

Performs per-halfword (un)signed comparison: returns 1 if both parts compare equal.

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts function performs comparison ‘a’ part == ‘b’ part. If both equalities are satisfied, function returns 1.

Returns

Returns 1 if a = b, else returns 0.

__device__ unsigned int **__vseteq4**(unsigned int a, unsigned int b)

Performs per-byte (un)signed comparison: returns 1 if all 4 pairs compare equal.

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts function performs comparison ‘a’ part == ‘b’ part. If both equalities are satisfied, function returns 1.

Returns

Returns 1 if a = b, else returns 0.

__device__ unsigned int **__vsetges2**(unsigned int a, unsigned int b)

Performs per-halfword signed comparison: returns 1 if both parts compare greater than or equal.

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts function performs comparison 'a' part >= 'b' part. If both inequalities are satisfied, function returns 1.

Returns

Returns 1 if a >= b, else returns 0.

__device__ unsigned int **__vsetges4**(unsigned int a, unsigned int b)

Performs per-byte signed comparison: returns 1 if all 4 pairs compare greater than or equal.

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts function performs comparison 'a' part >= 'b' part. If both inequalities are satisfied, function returns 1.

Returns

Returns 1 if a >= b, else returns 0.

__device__ unsigned int **__vsetgeu2**(unsigned int a, unsigned int b)

Performs per-halfword unsigned comparison: returns 1 if both parts compare greater than or equal.

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts function performs comparison 'a' part >= 'b' part. If both inequalities are satisfied, function returns 1.

Returns

Returns 1 if a >= b, else returns 0.

__device__ unsigned int **__vsetgeu4**(unsigned int a, unsigned int b)

Performs per-byte unsigned comparison: returns 1 if all 4 pairs compare greater than or equal.

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts function performs comparison 'a' part >= 'b' part. If both inequalities are satisfied, function returns 1.

Returns

Returns 1 if a >= b, else returns 0.

__device__ unsigned int **__vsetgts2**(unsigned int a, unsigned int b)

Performs per-halfword signed comparison: returns 1 if both parts compare greater than.

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts function performs comparison 'a' part > 'b' part. If both inequalities are satisfied, function returns 1.

Returns

Returns 1 if a > b, else returns 0.

__device__ unsigned int **__vsetgts4**(unsigned int a, unsigned int b)

Performs per-byte signed comparison: returns 1 if all 4 pairs compare greater than.

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts function performs comparison 'a' part > 'b' part. If both inequalities are satisfied, function returns 1.

Returns

Returns 1 if a > b, else returns 0.

__device__ unsigned int **__vsetgtu2**(unsigned int a, unsigned int b)

Performs per-halfword unsigned comparison: returns 1 if both parts compare greater than.

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts function performs comparison 'a' part > 'b' part. If both inequalities are satisfied, function returns 1.

Returns

Returns 1 if a > b, else returns 0.

__device__ unsigned int **__vsetgtu4**(unsigned int a, unsigned int b)

Performs per-byte unsigned comparison: returns 1 if all 4 pairs compare greater than.

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts function performs comparison 'a' part > 'b' part. If both inequalities are satisfied, function returns 1.

Returns

Returns 1 if a > b, else returns 0.

__device__ unsigned int **__vsetles2**(unsigned int a, unsigned int b)

Performs per-halfword unsigned comparison: returns 1 if both parts compare less than or equal.

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts function performs comparison 'a' part <= 'b' part. If both inequalities are satisfied, function returns 1.

Returns

Returns 1 if a <= b, else returns 0.

__device__ unsigned int **__vsetles4**(unsigned int a, unsigned int b)

Performs per-byte signed comparison: returns 1 if all 4 pairs compare less than or equal.

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts function performs comparison 'a' part <= 'b' part. If both inequalities are satisfied, function returns 1.

Returns

Returns 1 if a <= b, else returns 0.

__device__ unsigned int **__vsetleu2**(unsigned int a, unsigned int b)

Performs per-halfword signed comparison: returns 1 if both parts compare less than or equal.

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts function performs comparison 'a' part <= 'b' part. If both inequalities are satisfied, function returns 1.

Returns

Returns 1 if a <= b, else returns 0.

__device__ unsigned int **__vsetleu4**(unsigned int a, unsigned int b)

Performs per-byte unsigned comparison: returns 1 if all 4 pairs compare less than or equal.

Splits 4 bytes of each argument into 4 part, each consisting of 1 byte. For corresponding parts function performs comparison 'a' part <= 'b' part. If both inequalities are satisfied, function returns 1.

Returns

Returns 1 if a <= b, else returns 0.

__device__ unsigned int **__vsetlts2**(unsigned int a, unsigned int b)

Performs per-halfword signed comparison: returns 1 if both parts compare less than.

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts function performs comparison 'a' part <= 'b' part. If both inequalities are satisfied, function returns 1.

Returns

Returns 1 if a < b, else returns 0.

__device__ unsigned int **__vsetlts4**(unsigned int a, unsigned int b)

Performs per-byte signed comparison: returns 1 if all 4 pairs compare less than.

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts function performs comparison 'a' part <= 'b' part. If both inequalities are satisfied, function returns 1.

Returns

Returns 1 if a < b, else returns 0.

__device__ unsigned int **__vsetltu2**(unsigned int a, unsigned int b)

Performs per-halfword unsigned comparison: returns 1 if both parts compare less than.

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts function performs comparison 'a' part <= 'b' part. If both inequalities are satisfied, function returns 1.

Returns

Returns 1 if a < b, else returns 0.

__device__ unsigned int **__vsetltu4**(unsigned int a, unsigned int b)

Performs per-byte unsigned comparison: returns 1 if all 4 pairs compare less than.

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts function performs comparison 'a' part <= 'b' part. If both inequalities are satisfied, function returns 1.

Returns

Returns 1 if a < b, else returns 0.

__device__ unsigned int **__vsetne2**(unsigned int a, unsigned int b)

Performs per-halfword (un)signed comparison: returns 1 if both parts compare not equal.

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts function performs comparison 'a' part != 'b' part. If both conditions are satisfied, function returns 1.

Returns

Returns 1 if a != b, else returns 0.

__device__ unsigned int **__vsetne4**(unsigned int a, unsigned int b)

Performs per-byte (un)signed comparison: returns 1 if all 4 pairs compare not equal.

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts function performs comparison 'a' part != 'b' part. If both conditions are satisfied, function returns 1.

Returns

Returns 1 if a != b, else returns 0.

__device__ unsigned int **__vsub2**(unsigned int a, unsigned int b)

Performs per-halfword (un)signed subtraction, with wrap-around: a - b.

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts function performs subtraction. Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__device__ unsigned int **__vsub4** (unsigned int a, unsigned int b)

Performs per-byte subtraction: a - b.

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts function performs subtraction. Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__device__ unsigned int **__vsubss2** (unsigned int a, unsigned int b)

Performs per-halfword (un)signed subtraction, with signed saturation: a - b.

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts function performs subtraction with signed saturation. Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__device__ unsigned int **__vsubss4** (unsigned int a, unsigned int b)

Performs per-byte subtraction with signed saturation: a - b.

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts function performs subtraction with signed saturation. Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__device__ unsigned int **__vsubus2** (unsigned int a, unsigned int b)

Performs per-halfword subtraction with unsigned saturation: a - b.

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts function performs subtraction with unsigned saturation. Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

__device__ unsigned int **__vsubus4** (unsigned int a, unsigned int b)

Performs per-byte subtraction with unsigned saturation: a - b.

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts function performs subtraction with unsigned saturation. Partial results are recombined and returned as unsigned int.

Returns

Returns computed value.

Chapter 15. Structs

15.1. __half

struct **__half**

__half data type

This structure implements the datatype for storing half-precision floating-point numbers. The structure implements assignment, arithmetic and comparison operators, and type conversions. 16 bits are being used in total: 1 sign bit, 5 bits for the exponent, and the significand is being stored in 10 bits. The total precision is 11 bits. There are 15361 representable numbers within the interval [0.0, 1.0], endpoints included. On average we have log10(2**11) ~ 3.311 decimal digits.

The objective here is to provide IEEE754-compliant implementation of `binary16` type and arithmetic with limitations due to device HW not supporting floating-point exceptions.

Public Functions

__half() = default

Constructor by default.

Emtpy default constructor, result is uninitialized.

__host__ __device__ inline constexpr **__half**(const *__half_raw* &hr)

Constructor from `__half_raw`.

__host__ __device__ explicit **__half**(const *__nv_bfloat16* f)

Construct `__half` from `__nv_bfloat16` input using default round-to-nearest-even rounding mode.

Need to include the header file `cuda_bf16.h`

__host__ __device__ inline **__half**(const double f)

Construct `__half` from `double` input using default round-to-nearest-even rounding mode.

See also:

__double2half(double) for further details.

__host__ __device__ inline **__half**(const float f)

Construct `__half` from `float` input using default round-to-nearest-even rounding mode.

See also:

__float2half(float) for further details.

__host__ __device__ inline **__half**(const int val)

Construct `__half` from `int` input using default round-to-nearest-even rounding mode.

__host__ __device__ inline **__half**(const long long val)

Construct `__half` from `long long` input using default round-to-nearest-even rounding mode.

__host__ __device__ inline **__half**(const long val)

Construct `__half` from `long` input using default round-to-nearest-even rounding mode.

__host__ __device__ inline **__half**(const short val)

Construct `__half` from `short` integer input using default round-to-nearest-even rounding mode.

__host__ __device__ inline **__half**(const unsigned int val)

Construct `__half` from `unsigned int` input using default round-to-nearest-even rounding mode.

__host__ __device__ inline **__half**(const unsigned long long val)

Construct `__half` from `unsigned long long` input using default round-to-nearest-even rounding mode.

__host__ __device__ inline **__half**(const unsigned long val)

Construct `__half` from `unsigned long` input using default round-to-nearest-even rounding mode.

__host__ __device__ inline **__half**(const unsigned short val)

Construct `__half` from `unsigned short` integer input using default round-to-nearest-even rounding mode.

__host__ __device__ **operator** ***__half_raw***() const

Type cast to `__half_raw` operator.

__host__ __device__ **operator** ***__half_raw***() volatile const

Type cast to `__half_raw` operator with `volatile` input.

__host__ __device__ inline constexpr **operator bool**() const

Conversion operator to `bool` data type.

+0 and -0 inputs convert to `false`. Non-zero inputs convert to `true`.

__host__ __device__ **operator char**() const

Conversion operator to an implementation defined `char` data type.

Using round-toward-zero rounding mode.

Detects signedness of the `char` type and proceeds accordingly, see further details in *__half2char_rz(__half)* and *__half2uchar_rz(__half)*.

__host__ __device__ **operator float**() const

Type cast to `float` operator.

__host__ __device__ **operator int**() const

Conversion operator to `int` data type.

Using round-toward-zero rounding mode.

See also:

__half2int_rz(__half) for further details.

__host__ __device__ **operator long**() const

Conversion operator to `long` data type.

Using round-toward-zero rounding mode.

Detects size of the `long` type and proceeds accordingly, see further details in *__half2int_rz(__half)* and *__half2ll_rz(__half)*.

__host__ __device__ **operator long long**() const

Conversion operator to `long long` data type.

Using round-toward-zero rounding mode.

See also:

__half2ll_rz(__half) for further details.

__host__ __device__ **operator short**() const

Conversion operator to `short` data type.

Using round-toward-zero rounding mode.

See also:

__half2short_rz(__half) for further details.

__host__ __device__ **operator signed char**() const

Conversion operator to `signed char` data type.

Using round-toward-zero rounding mode.

See also:

__half2char_rz(__half) for further details.

__host__ __device__ **operator unsigned char**() const

Conversion operator to `unsigned char` data type.

Using round-toward-zero rounding mode.

See also:

__half2uchar_rz(__half) for further details.

__host__ __device__ **operator unsigned int**() const

Conversion operator to `unsigned int` data type.

Using round-toward-zero rounding mode.

See also:

__half2uint_rz(__half) for further details.

__host__ __device__ **operator unsigned long**() const

Conversion operator to `unsigned long` data type.

Using round-toward-zero rounding mode.

Detects size of the `unsigned long` type and proceeds accordingly, see further details in *__half2uint_rz(__half)* and *__half2ull_rz(__half)*.

__host__ __device__ **operator unsigned long long**() const

Conversion operator to `unsigned long long` data type.

Using round-toward-zero rounding mode.

See also:

__half2ull_rz(__half) for further details.

__host__ __device__ **operator unsigned short**() const

Conversion operator to `unsigned short` data type.

Using round-toward-zero rounding mode.

See also:

__half2ushort_rz(__half) for further details.

__host__ __device__ *__half* &**operator=**(const *__half_raw* &hr)

Assignment operator from *`__half_raw`*.

__host__ __device__ volatile *__half* &**operator=**(const *__half_raw* &hr) volatile

Assignment operator from *`__half_raw`* to `volatile` *`__half`*.

__host__ __device__ *__half* &**operator=**(const double f)

Type cast to *`__half`* assignment operator from `double` input using default round-to-nearest-even rounding mode.

See also:

__double2half(double) for further details.

__host__ __device__ *__half* &**operator=**(const float f)

Type cast to *`__half`* assignment operator from `float` input using default round-to-nearest-even rounding mode.

See also:

__float2half(float) for further details.

__host__ __device__ *__half* &**operator=**(const int val)

Type cast from `int` assignment operator, using default round-to-nearest-even rounding mode.

__host__ __device__ *__half* &**operator=**(const long long val)

Type cast from `long long` assignment operator, using default round-to-nearest-even rounding mode.

__host__ __device__ *__half* &**operator=**(const short val)

Type cast from `short` assignment operator, using default round-to-nearest-even rounding mode.

__host__ __device__ *__half* &**operator=**(const unsigned int val)

Type cast from `unsigned int` assignment operator, using default round-to-nearest-even rounding mode.

__host__ __device__ *__half* &**operator=**(const unsigned long long val)

Type cast from `unsigned long long` assignment operator, using default round-to-nearest-even rounding mode.

__host__ __device__ *__half* &**operator=**(const unsigned short val)

Type cast from `unsigned short` assignment operator, using default round-to-nearest-even rounding mode.

__host__ __device__ volatile *__half* &**operator=**(volatile const *__half_raw* &hr) volatile

Assignment operator from `volatile` *`__half_raw`* to `volatile` *`__half`*.

15.2. __half2

struct **__half2**

__half2 data type

This structure implements the datatype for storing two half-precision floating-point numbers. The structure implements assignment, arithmetic and comparison operators, and type conversions.

- NOTE: *__half2* is visible to non-nvcc host compilers

Public Functions

__half2() = default

Constructor by default.

Emtpy default constructor, result is uninitialized.

__host__ __device__ inline constexpr **__half2**(const *__half* &a, const *__half* &b)

Constructor from two *`__half`* variables.

__host__ __device__ inline **__half2**(const *__half2* &&src)

Move constructor, available for C++11 and later dialects.

__host__ __device__ inline **__half2**(const *__half2* &src)

Copy constructor.

__host__ __device__ inline **__half2**(const *__half2_raw* &h2r)

Constructor from *__half2_raw*.

__host__ __device__ **operator** ***__half2_raw***() const

Conversion operator to *__half2_raw*.

__host__ __device__ *__half2* &**operator=**(const *__half2* &&src)

Move assignment operator, available for C++11 and later dialects.

__host__ __device__ *__half2* &**operator=**(const *__half2* &src)

Copy assignment operator.

__host__ __device__ *__half2* &**operator=**(const *__half2_raw* &h2r)

Assignment operator from *__half2_raw*.

Public Members

__half **x**

Storage field holding lower *__half* part.

__half **y**

Storage field holding upper *__half* part.

15.3. __half2_raw

struct **__half2_raw**

__half2_raw data type

Type allows static initialization of `half2` until it becomes a built-in type.

- Note: this initialization is as a bit-field representation of `half2`, and not a conversion from `short2` to `half2`. Such representation will be deprecated in a future version of CUDA.
- Note: this is visible to non-nvcc compilers, including C-only compilations

Public Members

unsigned short **x**

Storage field contains bits of the lower `half` part.

unsigned short **y**

Storage field contains bits of the upper `half` part.

15.4. __half_raw

struct **__half_raw**

__half_raw data type

Type allows static initialization of `half` until it becomes a built-in type.

- Note: this initialization is as a bit-field representation of `half`, and not a conversion from `short` to `half`. Such representation will be deprecated in a future version of CUDA.
- Note: this is visible to non-nvcc compilers, including C-only compilations

Public Members

unsigned short **x**

Storage field contains bits representation of the `half` floating-point number.

15.5. __nv_bfloat16

struct **__nv_bfloat16**

nv_bfloat16 datatype

This structure implements the datatype for storing nv_bfloat16 floating-point numbers. The structure implements assignment operators and type conversions. 16 bits are being used in total: 1 sign bit, 8 bits for the exponent, and the significand is being stored in 7 bits. The total precision is 8 bits.

Public Functions

__nv_bfloat16() = default

Constructor by default.

Emtpy default constructor, result is uninitialized.

__host__ __device__ inline explicit **__nv_bfloat16**(const *__half* f)

Construct *__nv_bfloat16* from *__half* input using default round-to-nearest-even rounding mode.

__host__ __device__ inline constexpr **__nv_bfloat16**(const *__nv_bfloat16_raw* &hr)

Constructor from *__nv_bfloat16_raw*.

__host__ __device__ inline **__nv_bfloat16**(const double f)

Construct *__nv_bfloat16* from `double` input using default round-to-nearest-even rounding mode.

__host__ __device__ inline **__nv_bfloat16**(const float f)

Construct *__nv_bfloat16* from `float` input using default round-to-nearest-even rounding mode.

__host__ __device__ inline **__nv_bfloat16**(const long val)

Construct *__nv_bfloat16* from `long` input using default round-to-nearest-even rounding mode.

__host__ __device__ inline **__nv_bfloat16**(const unsigned long val)

Construct *__nv_bfloat16* from `unsigned long` input using default round-to-nearest-even rounding mode.

__host__ __device__ inline **__nv_bfloat16**(int val)

Construct *__nv_bfloat16* from `int` input using default round-to-nearest-even rounding mode.

__host__ __device__ inline **__nv_bfloat16**(long long val)

Construct *__nv_bfloat16* from `long long` input using default round-to-nearest-even rounding mode.

__host__ __device__ inline **__nv_bfloat16**(short val)

Construct *__nv_bfloat16* from `short` integer input using default round-to-nearest-even rounding mode.

__host__ __device__ inline **__nv_bfloat16**(unsigned int val)

Construct *__nv_bfloat16* from `unsigned int` input using default round-to-nearest-even rounding mode.

__host__ __device__ inline **__nv_bfloat16**(unsigned long long val)

Construct *__nv_bfloat16* from `unsigned long long` input using default round-to-nearest-even rounding mode.

__host__ __device__ inline **__nv_bfloat16**(unsigned short val)

Construct *__nv_bfloat16* from `unsigned short` integer input using default round-to-nearest-even rounding mode.

__host__ __device__ **operator** ***__nv_bfloat16_raw***() const

Type cast to *__nv_bfloat16_raw* operator.

__host__ __device__ **operator** ***__nv_bfloat16_raw***() volatile const

Type cast to *__nv_bfloat16_raw* operator with `volatile` input.

__host__ __device__ inline constexpr **operator bool**() const

Conversion operator to `bool` data type.

+0 and -0 inputs convert to `false`. Non-zero inputs convert to `true`.

__host__ __device__ **operator char**() const

Conversion operator to an implementation defined `char` data type.

Using round-toward-zero rounding mode.

Detects signedness of the `char` type and proceeds accordingly, see further details in signed and unsigned char operators.

__host__ __device__ **operator float**() const

Type cast to `float` operator.

__host__ __device__ **operator int**() const
: Conversion operator to `int` data type.

  Using round-toward-zero rounding mode.

  See *__bfloat162int_rz(__nv_bfloat16)* for further details

__host__ __device__ **operator long**() const
: Conversion operator to `long` data type.

  Using round-toward-zero rounding mode.

__host__ __device__ **operator long long**() const
: Conversion operator to `long long` data type.

  Using round-toward-zero rounding mode.

  See *__bfloat162ll_rz(__nv_bfloat16)* for further details

__host__ __device__ **operator short**() const
: Conversion operator to `short` data type.

  Using round-toward-zero rounding mode.

  See *__bfloat162short_rz(__nv_bfloat16)* for further details

__host__ __device__ **operator signed char**() const
: Conversion operator to `signed char` data type.

  Using round-toward-zero rounding mode.

  See *__bfloat162char_rz(__nv_bfloat16)* for further details

__host__ __device__ **operator unsigned char**() const
: Conversion operator to `unsigned char` data type.

  Using round-toward-zero rounding mode.

  See *__bfloat162uchar_rz(__nv_bfloat16)* for further details

__host__ __device__ **operator unsigned int**() const
: Conversion operator to `unsigned int` data type.

  Using round-toward-zero rounding mode.

  See *__bfloat162uint_rz(__nv_bfloat16)* for further details

__host__ __device__ **operator unsigned long**() const
: Conversion operator to `unsigned long` data type.

  Using round-toward-zero rounding mode.

__host__ __device__ **operator unsigned long long**() const
: Conversion operator to `unsigned long long` data type.

  Using round-toward-zero rounding mode.

  See *__bfloat162ull_rz(__nv_bfloat16)* for further details

__host__ __device__ **operator unsigned short**() const
: Conversion operator to `unsigned short` data type.

  Using round-toward-zero rounding mode.

  See *__bfloat162ushort_rz(__nv_bfloat16)* for further details __host__ __device__ *__nv_bfloat16* &**operator=**(const *__nv_bfloat16_raw* &hr)

Assignment operator from `__nv_bfloat16_raw`.

__host__ __device__ volatile *__nv_bfloat16* &**operator=**(const *__nv_bfloat16_raw* &hr) volatile

Assignment operator from `__nv_bfloat16_raw` to `volatile __nv_bfloat16`.

__host__ __device__ *__nv_bfloat16* &**operator=**(const double f)

Type cast to `__nv_bfloat16` assignment operator from `double` input using default round-to-nearest-even rounding mode.

__host__ __device__ *__nv_bfloat16* &**operator=**(const float f)

Type cast to `__nv_bfloat16` assignment operator from `float` input using default round-to-nearest-even rounding mode.

__host__ __device__ volatile *__nv_bfloat16* &**operator=**(volatile const *__nv_bfloat16_raw* &hr) volatile

Assignment operator from `volatile __nv_bfloat16_raw` to `volatile __nv_bfloat16`.

__host__ __device__ *__nv_bfloat16* &**operator=**(int val)

Type cast from `int` assignment operator, using default round-to-nearest-even rounding mode.

__host__ __device__ *__nv_bfloat16* &**operator=**(long long val)

Type cast from `long long` assignment operator, using default round-to-nearest-even rounding mode.

__host__ __device__ *__nv_bfloat16* &**operator=**(short val)

Type cast from `short` assignment operator, using default round-to-nearest-even rounding mode.

__host__ __device__ *__nv_bfloat16* &**operator=**(unsigned int val)

Type cast from `unsigned int` assignment operator, using default round-to-nearest-even rounding mode.

__host__ __device__ *__nv_bfloat16* &**operator=**(unsigned long long val)

Type cast from `unsigned long long` assignment operator, using default round-to-nearest-even rounding mode.

__host__ __device__ *__nv_bfloat16* &**operator=**(unsigned short val)

Type cast from `unsigned short` assignment operator, using default round-to-nearest-even rounding mode.

15.6. __nv_bfloat162

struct **`__nv_bfloat162`**

nv_bfloat162 datatype

This structure implements the datatype for storing two nv_bfloat16 floating-point numbers. The structure implements assignment, arithmetic and comparison operators, and type conversions.

- NOTE: *__nv_bfloat162* is visible to non-nvcc host compilers

Public Functions

__nv_bfloat162( ) = default

Constructor by default.

Emtpy default constructor, result is uninitialized.

__host__ __device__ **__nv_bfloat162**(*__nv_bfloat162* &&src)

Move constructor, available for C++11 and later dialects.

__host__ __device__ inline constexpr **__nv_bfloat162**(const *__nv_bfloat16* &a, const *__nv_bfloat16* &b)

Constructor from two `__nv_bfloat16` variables.

__host__ __device__ **__nv_bfloat162**(const *__nv_bfloat162* &src)

Copy constructor.

__host__ __device__ **__nv_bfloat162**(const *__nv_bfloat162_raw* &h2r)

Constructor from `__nv_bfloat162_raw`.

__host__ __device__ **operator __nv_bfloat162_raw**( ) const

Conversion operator to `__nv_bfloat162_raw`.

__host__ __device__ *__nv_bfloat162* &**operator=**(*__nv_bfloat162* &&src)

Move assignment operator, available for C++11 and later dialects.

__host__ __device__ *__nv_bfloat162* &**operator=**(const *__nv_bfloat162* &src)

Copy assignment operator.

__host__ __device__ *__nv_bfloat162* &**operator=**(const *__nv_bfloat162_raw* &h2r)

Assignment operator from `__nv_bfloat162_raw`.

Public Members

__nv_bfloat16 **x**

Storage field holding lower `__nv_bfloat16` part.

__nv_bfloat16 **y**

Storage field holding upper `__nv_bfloat16` part.

15.7. __nv_bfloat162_raw

struct **__nv_bfloat162_raw**

__nv_bfloat162_raw data type

Type allows static initialization of `nv_bfloat162` until it becomes a built-in type.

- Note: this initialization is as a bit-field representation of `nv_bfloat162`, and not a conversion from `short2` to `nv_bfloat162`. Such representation will be deprecated in a future version of CUDA.

- Note: this is visible to non-nvcc compilers, including C-only compilations

Public Members

unsigned short **x**

Storage field contains bits of the lower `nv_bfloat16` part.

unsigned short **y**

Storage field contains bits of the upper `nv_bfloat16` part.

15.8. __nv_bfloat16_raw

struct **`__nv_bfloat16_raw`**

__nv_bfloat16_raw data type

Type allows static initialization of `nv_bfloat16` until it becomes a built-in type.

- Note: this initialization is as a bit-field representation of `nv_bfloat16`, and not a conversion from `short` to `nv_bfloat16`. Such representation will be deprecated in a future version of CUDA.
- Note: this is visible to non-nvcc compilers, including C-only compilations

Public Members

unsigned short **x**

Storage field contains bits representation of the `nv_bfloat16` floating-point number.

15.9. __nv_fp4_e2m1

struct **`__nv_fp4_e2m1`**

__nv_fp4_e2m1 datatype

This structure implements the datatype for handling `fp4` floating-point numbers of `e2m1` kind: with 1 sign, 2 exponent, 1 implicit and 1 explicit mantissa bits. This encoding does not support Inf/NaN.

The structure implements converting constructors and operators.

Public Functions

__host__ __device__ inline **__nv_fp4_e2m1**()

Constructor by default.

__host__ __device__ inline explicit **__nv_fp4_e2m1**(const *__half* f)

Constructor from *__half* data type, relies on `__NV_SATFINITE` behavior for out-of-range values and `cudaRoundNearest` rounding mode.

__host__ __device__ inline explicit **__nv_fp4_e2m1**(const *__nv_bfloat16* f)

Constructor from *__nv_bfloat16* data type, relies on `__NV_SATFINITE` behavior for out-of-range values and `cudaRoundNearest` rounding mode.

__host__ __device__ inline explicit **__nv_fp4_e2m1**(const double f)

Constructor from `double` data type, relies on `__NV_SATFINITE` behavior for out-of-range values and `cudaRoundNearest` rounding mode.

__host__ __device__ inline explicit **__nv_fp4_e2m1**(const float f)

Constructor from `float` data type, relies on `__NV_SATFINITE` behavior for out-of-range values and `cudaRoundNearest` rounding mode.

__host__ __device__ inline explicit **__nv_fp4_e2m1**(const int val)

Constructor from `int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp4_e2m1**(const long int val)

Constructor from `long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp4_e2m1**(const long long int val)

Constructor from `long long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp4_e2m1**(const short int val)

Constructor from `short int` data type.

__host__ __device__ inline explicit **__nv_fp4_e2m1**(const unsigned int val)

Constructor from `unsigned int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp4_e2m1**(const unsigned long int val)

Constructor from `unsigned long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp4_e2m1**(const unsigned long long int val)

Constructor from `unsigned long long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp4_e2m1**(const unsigned short int val)

Constructor from `unsigned short int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

Public Members

__nv_fp4_storage_t **__x**

Storage variable contains the `fp4` floating-point data.

15.10. __nv_fp4x2_e2m1

struct **__nv_fp4x2_e2m1**

__nv_fp4x2_e2m1 datatype

This structure implements the datatype for handling two `fp4` floating-point numbers of `e2m1` kind each.

The structure implements converting constructors and operators.

Public Functions

__host__ __device__ inline **__nv_fp4x2_e2m1**()

Constructor by default.

__host__ __device__ inline explicit **__nv_fp4x2_e2m1**(const *__half2* f)

Constructor from *__half2* data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp4x2_e2m1**(const *__nv_bfloat162* f)

Constructor from *__nv_bfloat162* data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp4x2_e2m1**(const double2 f)

Constructor from `double2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp4x2_e2m1**(const float2 f)

Constructor from `float2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

Public Members

__nv_fp4x2_storage_t **__x**

Storage variable contains the vector of two `fp4` floating-point data values.

15.11. __nv_fp4x4_e2m1

struct **__nv_fp4x4_e2m1**

__nv_fp4x4_e2m1 datatype

This structure implements the datatype for handling four `fp4` floating-point numbers of `e2m1` kind each.

The structure implements converting constructors and operators.

Public Functions

__host__ __device__ inline **__nv_fp4x4_e2m1**()

Constructor by default.

__host__ __device__ inline explicit **__nv_fp4x4_e2m1**(const *__half2* flo, const *__half2* fhi)

Constructor from a pair of *__half2* data type values, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp4x4_e2m1**(const *__nv_bfloat162* flo, const *__nv_bfloat162* fhi)

Constructor from a pair of *__nv_bfloat162* data type values, relies on `__NV_SATFINITE` behavior for out-of-range values.

inline explicit __NV_SILENCE_DEPRECATION_BEGIN __host__ __device__ **__nv_fp4x4_e2m1**(const double4 f)

Constructor from `double4` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

inline explicit __NV_SILENCE_DEPRECATION_END __host__ __device__ **__nv_fp4x4_e2m1**(const double4_16a f)

Constructor from `double4_16a` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp4x4_e2m1**(const double4_32a f)

Constructor from `double4_32a` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp4x4_e2m1**(const float4 f)

Constructor from `float4` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

Public Members

__nv_fp4x4_storage_t **__x**

Storage variable contains the vector of four `fp4` floating-point data values.

15.12. __nv_fp6_e2m3

struct **__nv_fp6_e2m3**

__nv_fp6_e2m3 datatype

This structure implements the datatype for storing `fp6` floating-point numbers of `e2m3` kind: with 1 sign, 2 exponent, 1 implicit and 3 explicit mantissa bits. This encoding does not support Inf/NaN.

The structure implements converting constructors and operators.

Public Functions

__host__ __device__ inline **__nv_fp6_e2m3**()

Constructor by default.

__host__ __device__ inline explicit **__nv_fp6_e2m3**(const *__half* f)

Constructor from *__half* data type, relies on `__NV_SATFINITE` behavior for out-of-range values and `cudaRoundNearest` rounding mode.

__host__ __device__ inline explicit **__nv_fp6_e2m3**(const *__nv_bfloat16* f)

Constructor from *__nv_bfloat16* data type, relies on `__NV_SATFINITE` behavior for out-of-range values and `cudaRoundNearest` rounding mode.

__host__ __device__ inline explicit **__nv_fp6_e2m3**(const double f)

Constructor from `double` data type, relies on `__NV_SATFINITE` behavior for out-of-range values and `cudaRoundNearest` rounding mode.

__host__ __device__ inline explicit **__nv_fp6_e2m3**(const float f)

Constructor from `float` data type, relies on `__NV_SATFINITE` behavior for out-of-range values and `cudaRoundNearest` rounding mode.

__host__ __device__ inline explicit **__nv_fp6_e2m3**(const int val)

Constructor from `int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp6_e2m3**(const long int val)

Constructor from `long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp6_e2m3**(const long long int val)

Constructor from `long long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp6_e2m3**(const short int val)

Constructor from `short int` data type.

__host__ __device__ inline explicit **__nv_fp6_e2m3**(const unsigned int val)

Constructor from `unsigned int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp6_e2m3**(const unsigned long int val)

Constructor from `unsigned long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp6_e2m3**(const unsigned long long int val)

Constructor from `unsigned long long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp6_e2m3**(const unsigned short int val)

Constructor from `unsigned short int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

Public Members

__nv_fp6_storage_t **__x**

Storage variable contains the `fp6` floating-point data.

15.13. __nv_fp6_e3m2

struct **__nv_fp6_e3m2**

__nv_fp6_e3m2 datatype

This structure implements the datatype for handling `fp6` floating-point numbers of `e3m2` kind: with 1 sign, 3 exponent, 1 implicit and 2 explicit mantissa bits. This encoding does not support Inf/NaN.

The structure implements converting constructors and operators.

Public Functions

__host__ __device__ inline **__nv_fp6_e3m2**()

Constructor by default.

__host__ __device__ inline explicit **__nv_fp6_e3m2**(const *__half* f)

Constructor from *`__half`* data type, relies on `__NV_SATFINITE` behavior for out-of-range values and `cudaRoundNearest` rounding mode.

__host__ __device__ inline explicit **__nv_fp6_e3m2**(const *__nv_bfloat16* f)

Constructor from *`__nv_bfloat16`* data type, relies on `__NV_SATFINITE` behavior for out-of-range values and `cudaRoundNearest` rounding mode.

__host__ __device__ inline explicit **__nv_fp6_e3m2**(const double f)

Constructor from `double` data type, relies on `__NV_SATFINITE` behavior for out-of-range values and `cudaRoundNearest` rounding mode.

__host__ __device__ inline explicit **__nv_fp6_e3m2**(const float f)

Constructor from `float` data type, relies on `__NV_SATFINITE` behavior for out-of-range values and `cudaRoundNearest` rounding mode.

__host__ __device__ inline explicit **__nv_fp6_e3m2**(const int val)

Constructor from `int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp6_e3m2**(const long int val)

Constructor from `long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp6_e3m2**(const long long int val)

Constructor from `long long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp6_e3m2**(const short int val)

Constructor from `short int` data type.

__host__ __device__ inline explicit **__nv_fp6_e3m2**(const unsigned int val)

Constructor from `unsigned int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp6_e3m2**(const unsigned long int val)

Constructor from `unsigned long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp6_e3m2**(const unsigned long long int val)

Constructor from `unsigned long long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp6_e3m2**(const unsigned short int val)

Constructor from `unsigned short int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

Public Members

__nv_fp6_storage_t **__x**

Storage variable contains the `fp6` floating-point data.

15.14. __nv_fp6x2_e2m3

struct **__nv_fp6x2_e2m3**

__nv_fp6x2_e2m3 datatype

This structure implements the datatype for handling two `fp6` floating-point numbers of `e2m3` kind each.

The structure implements converting constructors and operators.

Public Functions

__host__ __device__ inline **__nv_fp6x2_e2m3**()

Constructor by default.

__host__ __device__ inline explicit **__nv_fp6x2_e2m3**(const *__half2* f)

Constructor from `__half2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp6x2_e2m3**(const *__nv_bfloat162* f)

Constructor from `__nv_bfloat162` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp6x2_e2m3**(const double2 f)

Constructor from `double2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp6x2_e2m3**(const float2 f)

Constructor from `float2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

Public Members

__nv_fp6x2_storage_t **__x**

Storage variable contains the vector of two `fp6` floating-point data values.

15.15. __nv_fp6x2_e3m2

struct **__nv_fp6x2_e3m2**

__nv_fp6x2_e3m2 datatype

This structure implements the datatype for handling two `fp6` floating-point numbers of `e3m2` kind each.

The structure implements converting constructors and operators.

Public Functions

__host__ __device__ inline **__nv_fp6x2_e3m2**()

Constructor by default.

__host__ __device__ inline explicit **__nv_fp6x2_e3m2**(const *__half2* f)

Constructor from `__half2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp6x2_e3m2**(const *__nv_bfloat162* f)

Constructor from `__nv_bfloat162` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp6x2_e3m2**(const double2 f)

Constructor from `double2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp6x2_e3m2**(const float2 f)

Constructor from `float2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

Public Members

__nv_fp6x2_storage_t **__x**

Storage variable contains the vector of two `fp6` floating-point data values.

15.16. __nv_fp6x4_e2m3

struct **__nv_fp6x4_e2m3**

__nv_fp6x4_e2m3 datatype

This structure implements the datatype for handling four `fp6` floating-point numbers of `e2m3` kind each.

The structure implements converting constructors and operators.

Public Functions

__host__ __device__ inline **__nv_fp6x4_e2m3**()

Constructor by default.

__host__ __device__ inline explicit **__nv_fp6x4_e2m3**(const *__half2* flo, const *__half2* fhi)

Constructor from a pair of *`__half2`* data type values, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp6x4_e2m3**(const *__nv_bfloat162* flo, const *__nv_bfloat162* fhi)

Constructor from a pair of *`__nv_bfloat162`* data type values, relies on `__NV_SATFINITE` behavior for out-of-range values.

inline explicit __NV_SILENCE_DEPRECATION_BEGIN __host__ __device__ **__nv_fp6x4_e2m3**(const double4 f)

Constructor from `double4` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

inline explicit __NV_SILENCE_DEPRECATION_END __host__ __device__ **__nv_fp6x4_e2m3**(const double4_16a f)

Constructor from `double4_16a` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **`__nv_fp6x4_e2m3`**(const double4_32a f)

Constructor from `double4_32a` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **`__nv_fp6x4_e2m3`**(const float4 f)

Constructor from `float4` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

Public Members

__nv_fp6x4_storage_t **`__x`**

Storage variable contains the vector of four `fp6` floating-point data values.

15.17. __nv_fp6x4_e3m2

struct **`__nv_fp6x4_e3m2`**

__nv_fp6x4_e3m2 datatype

This structure implements the datatype for handling four `fp6` floating-point numbers of `e3m2` kind each.

The structure implements converting constructors and operators.

Public Functions

__host__ __device__ inline **`__nv_fp6x4_e3m2`**()

Constructor by default.

__host__ __device__ inline explicit **`__nv_fp6x4_e3m2`**(const *__half2* flo, const *__half2* fhi)

Constructor from a pair of *`__half2`* data type values, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **`__nv_fp6x4_e3m2`**(const *__nv_bfloat162* flo, const *__nv_bfloat162* fhi)

Constructor from a pair of *`__nv_bfloat162`* data type values, relies on `__NV_SATFINITE` behavior for out-of-range values.

inline explicit __NV_SILENCE_DEPRECATION_BEGIN __host__ __device__ **`__nv_fp6x4_e3m2`**(const double4 f)

Constructor from `double4` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

inline explicit __NV_SILENCE_DEPRECATION_END __host__ __device__ **__nv_fp6x4_e3m2**(const double4_16a f)

Constructor from double4_16a vector data type, relies on __NV_SATFINITE behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp6x4_e3m2**(const double4_32a f)

Constructor from double4_32a vector data type, relies on __NV_SATFINITE behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp6x4_e3m2**(const float4 f)

Constructor from float4 vector data type, relies on __NV_SATFINITE behavior for out-of-range values.

Public Members

__nv_fp6x4_storage_t **__x**

Storage variable contains the vector of four fp6 floating-point data values.

15.18. __nv_fp8_e4m3

struct **__nv_fp8_e4m3**

__nv_fp8_e4m3 datatype

This structure implements the datatype for storing fp8 floating-point numbers of e4m3 kind: with 1 sign, 4 exponent, 1 implicit and 3 explicit mantissa bits. The encoding doesn't support Infinity. NaNs are limited to 0x7F and 0xFF values.

The structure implements converting constructors and operators.

Public Functions

__nv_fp8_e4m3() = default

Constructor by default.

__host__ __device__ inline explicit **__nv_fp8_e4m3**(const *__half* f)

Constructor from *__half* data type, relies on __NV_SATFINITE behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp8_e4m3**(const *__nv_bfloat16* f)

Constructor from *__nv_bfloat16* data type, relies on __NV_SATFINITE behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp8_e4m3**(const double f)

Constructor from double data type, relies on __NV_SATFINITE behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp8_e4m3**(const float f)

Constructor from `float` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp8_e4m3**(const int val)

Constructor from `int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp8_e4m3**(const long int val)

Constructor from `long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp8_e4m3**(const long long int val)

Constructor from `long long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp8_e4m3**(const short int val)

Constructor from `short int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp8_e4m3**(const unsigned int val)

Constructor from `unsigned int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp8_e4m3**(const unsigned long int val)

Constructor from `unsigned long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp8_e4m3**(const unsigned long long int val)

Constructor from `unsigned long long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp8_e4m3**(const unsigned short int val)

Constructor from `unsigned short int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **operator** ***__half***() const

Conversion operator to *__half* data type.

__host__ __device__ inline explicit **operator** ***__nv_bfloat16***() const

Conversion operator to *__nv_bfloat16* data type.

__host__ __device__ inline explicit **operator** **bool**() const

Conversion operator to `bool` data type.

+0 and -0 inputs convert to `false`. Non-zero inputs convert to `true`.

__host__ __device__ inline explicit **operator** **char**() const

Conversion operator to an implementation defined `char` data type.

Detects signedness of the `char` type and proceeds accordingly, see further details in signed and unsigned char operators.

Clamps inputs to the output range. NaN inputs convert to `zero`.

__host__ __device__ inline explicit **operator** **double**() const

Conversion operator to `double` data type.

__host__ __device__ inline explicit **operator float**() const

Conversion operator to `float` data type.

__host__ __device__ inline explicit **operator int**() const

Conversion operator to `int` data type.

`NaN` inputs convert to `zero`.

__host__ __device__ inline explicit **operator long int**() const

Conversion operator to `long int` data type.

Clamps too large inputs to the output range. `NaN` inputs convert to `zero` if output type is 32-bit. `NaN` inputs convert to `0x8000000000000000ULL` if output type is 64-bit.

__host__ __device__ inline explicit **operator long long int**() const

Conversion operator to `long long int` data type.

`NaN` inputs convert to `0x8000000000000000LL`.

__host__ __device__ inline explicit **operator short int**() const

Conversion operator to `short int` data type.

`NaN` inputs convert to `zero`.

__host__ __device__ inline explicit **operator signed char**() const

Conversion operator to `signed char` data type.

Clamps too large inputs to the output range. `NaN` inputs convert to `zero`.

__host__ __device__ inline explicit **operator unsigned char**() const

Conversion operator to `unsigned char` data type.

Clamps negative and too large inputs to the output range. `NaN` inputs convert to `zero`.

__host__ __device__ inline explicit **operator unsigned int**() const

Conversion operator to `unsigned int` data type.

Clamps negative inputs to zero. `NaN` inputs convert to `zero`.

__host__ __device__ inline explicit **operator unsigned long int**() const

Conversion operator to `unsigned long int` data type.

Clamps negative and too large inputs to the output range. `NaN` inputs convert to `zero` if output type is 32-bit. `NaN` inputs convert to `0x8000000000000000ULL` if output type is 64-bit.

__host__ __device__ inline explicit **operator unsigned long long int**() const

Conversion operator to `unsigned long long int` data type.

Clamps negative inputs to zero. `NaN` inputs convert to `0x8000000000000000ULL`.

__host__ __device__ inline explicit **operator unsigned short int**() const

Conversion operator to `unsigned short int` data type.

Clamps negative inputs to zero. `NaN` inputs convert to `zero`.

Public Members

__nv_fp8_storage_t **__x**

Storage variable contains the `fp8` floating-point data.

15.19. __nv_fp8_e5m2

struct **__nv_fp8_e5m2**

__nv_fp8_e5m2 datatype

This structure implements the datatype for handling `fp8` floating-point numbers of `e5m2` kind: with 1 sign, 5 exponent, 1 implicit and 2 explicit mantissa bits.

The structure implements converting constructors and operators.

Public Functions

__nv_fp8_e5m2() = default

Constructor by default.

__host__ __device__ inline explicit **__nv_fp8_e5m2**(const *__half* f)

Constructor from *__half* data type, relies on __NV_SATFINITE behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp8_e5m2**(const *__nv_bfloat16* f)

Constructor from *__nv_bfloat16* data type, relies on __NV_SATFINITE behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp8_e5m2**(const double f)

Constructor from `double` data type, relies on __NV_SATFINITE behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp8_e5m2**(const float f)

Constructor from `float` data type, relies on __NV_SATFINITE behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp8_e5m2**(const int val)

Constructor from `int` data type, relies on __NV_SATFINITE behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp8_e5m2**(const long int val)

Constructor from `long int` data type, relies on __NV_SATFINITE behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp8_e5m2**(const long long int val)

Constructor from `long long int` data type, relies on __NV_SATFINITE behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp8_e5m2**(const short int val)

Constructor from `short int` data type.

__host__ __device__ inline explicit **__nv_fp8_e5m2**(const unsigned int val)

Constructor from `unsigned int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp8_e5m2**(const unsigned long int val)

Constructor from `unsigned long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp8_e5m2**(const unsigned long long int val)

Constructor from `unsigned long long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp8_e5m2**(const unsigned short int val)

Constructor from `unsigned short int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **operator __half**() const

Conversion operator to `__half` data type.

__host__ __device__ inline explicit **operator __nv_bfloat16**() const

Conversion operator to `__nv_bfloat16` data type.

__host__ __device__ inline explicit **operator bool**() const

Conversion operator to `bool` data type.

+0 and -0 inputs convert to `false`. Non-zero inputs convert to `true`.

__host__ __device__ inline explicit **operator char**() const

Conversion operator to an implementation defined `char` data type.

Detects signedness of the `char` type and proceeds accordingly, see further details in signed and unsigned char operators.

Clamps inputs to the output range. `NaN` inputs convert to `zero`.

__host__ __device__ inline explicit **operator double**() const

Conversion operator to `double` data type.

__host__ __device__ inline explicit **operator float**() const

Conversion operator to `float` data type.

__host__ __device__ inline explicit **operator int**() const

Conversion operator to `int` data type.

Clamps too large inputs to the output range. `NaN` inputs convert to `zero`.

__host__ __device__ inline explicit **operator long int**() const

Conversion operator to `long int` data type.

Clamps too large inputs to the output range. `NaN` inputs convert to `zero` if output type is 32-bit. `NaN` inputs convert to `0x8000000000000000ULL` if output type is 64-bit.

__host__ __device__ inline explicit **operator long long int**() const

Conversion operator to `long long int` data type.

Clamps too large inputs to the output range. `NaN` inputs convert to `0x8000000000000000LL`.

__host__ __device__ inline explicit **operator short int**() const

Conversion operator to `short int` data type.

Clamps too large inputs to the output range. NaN inputs convert to `zero`.

__host__ __device__ inline explicit **operator signed char**() const

Conversion operator to `signed char` data type.

Clamps too large inputs to the output range. NaN inputs convert to `zero`.

__host__ __device__ inline explicit **operator unsigned char**() const

Conversion operator to `unsigned char` data type.

Clamps negative and too large inputs to the output range. NaN inputs convert to `zero`.

__host__ __device__ inline explicit **operator unsigned int**() const

Conversion operator to `unsigned int` data type.

Clamps negative and too large inputs to the output range. NaN inputs convert to `zero`.

__host__ __device__ inline explicit **operator unsigned long int**() const

Conversion operator to `unsigned long int` data type.

Clamps negative and too large inputs to the output range. NaN inputs convert to `zero` if output type is 32-bit. NaN inputs convert to `0x8000000000000000ULL` if output type is 64-bit.

__host__ __device__ inline explicit **operator unsigned long long int**() const

Conversion operator to `unsigned long long int` data type.

Clamps negative and too large inputs to the output range. NaN inputs convert to `0x8000000000000000ULL`.

__host__ __device__ inline explicit **operator unsigned short int**() const

Conversion operator to `unsigned short int` data type.

Clamps negative and too large inputs to the output range. NaN inputs convert to `zero`.

Public Members

__nv_fp8_storage_t **__x**

Storage variable contains the `fp8` floating-point data.

15.20. __nv_fp8_e8m0

struct **__nv_fp8_e8m0**

__nv_fp8_e8m0 datatype

This structure implements the datatype for handling 8-bit scale factors of `e8m0` kind: interpreted as powers of two with biased exponent. Bias equals to 127, so numbers 0 through 254 represent 2^-127 through 2^127. Number `0xFF` = 255 is reserved for NaN.

The structure implements converting constructors and operators.

Public Functions

__nv_fp8_e8m0() = default

Constructor by default.

__host__ __device__ inline explicit **__nv_fp8_e8m0**(const *__half* f)

Constructor from `__half` data type, relies on `__NV_SATFINITE` behavior for large input values and `cudaRoundPosInf` for rounding.

See also:

__nv_cvt_float_to_e8m0 for further details

__host__ __device__ inline explicit **__nv_fp8_e8m0**(const *__nv_bfloat16* f)

Constructor from `__nv_bfloat16` data type, relies on `__NV_SATFINITE` behavior for large input values and `cudaRoundPosInf` for rounding.

See also:

__nv_cvt_bfloat16raw_to_e8m0 for further details

__host__ __device__ inline explicit **__nv_fp8_e8m0**(const double f)

Constructor from `double` data type, relies on `__NV_SATFINITE` behavior for large input values and `cudaRoundPosInf` for rounding.

See also:

__nv_cvt_double_to_e8m0 for further details

__host__ __device__ inline explicit **__nv_fp8_e8m0**(const float f)

Constructor from `float` data type, relies on `__NV_SATFINITE` behavior behavior for large input values and `cudaRoundPosInf` for rounding.

See also:

__nv_cvt_float_to_e8m0 for further details

__host__ __device__ inline explicit **__nv_fp8_e8m0**(const int val)

Constructor from `int` data type, relies on `cudaRoundPosInf` rounding.

__host__ __device__ inline explicit **__nv_fp8_e8m0**(const long int val)

Constructor from `long int` data type, relies on `cudaRoundPosInf` rounding.

__host__ __device__ inline explicit **__nv_fp8_e8m0**(const long long int val)

Constructor from `long long int` data type, relies on `cudaRoundPosInf` rounding.

__host__ __device__ inline explicit **__nv_fp8_e8m0**(const short int val)

Constructor from `short int` data type, relies on `cudaRoundPosInf` rounding.

__host__ __device__ inline explicit **__nv_fp8_e8m0**(const unsigned int val)

Constructor from `unsigned int` data type, relies on `cudaRoundPosInf` rounding.

__host__ __device__ inline explicit **__nv_fp8_e8m0**(const unsigned long int val)

Constructor from `unsigned long int` data type, relies on `cudaRoundPosInf` rounding.

__host__ __device__ inline explicit **__nv_fp8_e8m0**(const unsigned long long int val)

Constructor from unsigned long long int data type, relies on cudaRoundPosInf rounding.

__host__ __device__ inline explicit **__nv_fp8_e8m0**(const unsigned short int val)

Constructor from unsigned short int data type, relies on cudaRoundPosInf rounding.

__host__ __device__ inline explicit **operator __half**() const

Conversion operator to *__half* data type.

__host__ __device__ inline explicit **operator __nv_bfloat16**() const

Conversion operator to *__nv_bfloat16* data type.

__host__ __device__ inline explicit **operator bool**() const

Conversion operator to bool data type.

All values in input range are non-zero, so result is always true.

__host__ __device__ inline explicit **operator char**() const

Conversion operator to an implementation defined char data type.

Detects signedness of the char type and proceeds accordingly, see further details in signed and unsigned char operators.

Clamps inputs to the output range. NaN inputs convert to zero.

__host__ __device__ inline explicit **operator double**() const

Conversion operator to double data type.

__host__ __device__ inline explicit **operator float**() const

Conversion operator to float data type.

__host__ __device__ inline explicit **operator int**() const

Conversion operator to int data type.

Clamps too large inputs to the output range. NaN inputs convert to zero.

__host__ __device__ inline explicit **operator long int**() const

Conversion operator to long int data type.

Clamps too large inputs to the output range. NaN inputs convert to zero if output type is 32-bit. NaN inputs convert to 0x8000000000000000ULL if output type is 64-bit.

__host__ __device__ inline explicit **operator long long int**() const

Conversion operator to long long int data type.

Clamps too large inputs to the output range. NaN inputs convert to 0x8000000000000000LL.

__host__ __device__ inline explicit **operator short int**() const

Conversion operator to short int data type.

Clamps too large inputs to the output range. NaN inputs convert to zero.

__host__ __device__ inline explicit **operator signed char**() const

Conversion operator to signed char data type.

Clamps too large inputs to the output range. NaN inputs convert to zero.

__host__ __device__ inline explicit **operator unsigned char**() const

Conversion operator to unsigned char data type.

Clamps too large inputs to the output range. NaN inputs convert to zero.

__host__ __device__ inline explicit **operator unsigned int**() const

Conversion operator to unsigned int data type.

Clamps too large inputs to the output range. NaN inputs convert to zero.

__host__ __device__ inline explicit **operator unsigned long int**() const

Conversion operator to unsigned long int data type.

Clamps too large inputs to the output range. NaN inputs convert to zero if output type is 32-bit. NaN inputs convert to 0x8000000000000000ULL if output type is 64-bit.

__host__ __device__ inline explicit **operator unsigned long long int**() const

Conversion operator to unsigned long long int data type.

Clamps too large inputs to the output range. NaN inputs convert to 0x8000000000000000ULL.

__host__ __device__ inline explicit **operator unsigned short int**() const

Conversion operator to unsigned short int data type.

Clamps too large inputs to the output range. NaN inputs convert to zero.

Public Members

__nv_fp8_storage_t **__x**

Storage variable contains the 8-bit scale data.

15.21. __nv_fp8x2_e4m3

struct **__nv_fp8x2_e4m3**

__nv_fp8x2_e4m3 datatype

This structure implements the datatype for storage and operations on the vector of two fp8 values of e4m3 kind each: with 1 sign, 4 exponent, 1 implicit and 3 explicit mantissa bits. The encoding doesn't support Infinity. NaNs are limited to 0x7F and 0xFF values.

Public Functions

__nv_fp8x2_e4m3() = default

Constructor by default.

__host__ __device__ inline explicit **__nv_fp8x2_e4m3**(const *__half2* f)

Constructor from *__half2* data type, relies on __NV_SATFINITE behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp8x2_e4m3**(const *__nv_bfloat162* f)

Constructor from `__nv_bfloat162` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp8x2_e4m3**(const double2 f)

Constructor from `double2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp8x2_e4m3**(const float2 f)

Constructor from `float2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **operator __half2**() const

Conversion operator to `__half2` data type.

__host__ __device__ inline explicit **operator float2**() const

Conversion operator to `float2` data type.

Public Members

__nv_fp8x2_storage_t **__x**

Storage variable contains the vector of two `fp8` floating-point data values.

15.22. __nv_fp8x2_e5m2

struct **__nv_fp8x2_e5m2**

__nv_fp8x2_e5m2 datatype

This structure implements the datatype for handling two `fp8` floating-point numbers of `e5m2` kind each: with 1 sign, 5 exponent, 1 implicit and 2 explicit mantissa bits.

The structure implements converting constructors and operators.

Public Functions

__nv_fp8x2_e5m2() = default

Constructor by default.

__host__ __device__ inline explicit **__nv_fp8x2_e5m2**(const *__half2* f)

Constructor from `__half2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp8x2_e5m2**(const *__nv_bfloat162* f)

Constructor from `__nv_bfloat162` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp8x2_e5m2**(const double2 f)

Constructor from `double2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp8x2_e5m2**(const float2 f)

Constructor from `float2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **operator __half2**() const

Conversion operator to `__half2` data type.

__host__ __device__ inline explicit **operator float2**() const

Conversion operator to `float2` data type.

Public Members

__nv_fp8x2_storage_t **__x**

Storage variable contains the vector of two `fp8` floating-point data values.

15.23. __nv_fp8x2_e8m0

struct **__nv_fp8x2_e8m0**

__nv_fp8x2_e8m0 datatype

This structure implements the datatype for storage and operations on the vector of two scale factors of `e8m0` kind each.

Public Functions

__nv_fp8x2_e8m0() = default

Constructor by default.

__host__ __device__ inline explicit **__nv_fp8x2_e8m0**(const *__half2* f)

Constructor from `__half2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp8x2_e8m0**(const *__nv_bfloat162* f)

Constructor from `__nv_bfloat162` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp8x2_e8m0**(const double2 f)

Constructor from `double2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp8x2_e8m0**(const float2 f)

Constructor from `float2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **operator __half2**() const

Conversion operator to `__half2` data type.

__host__ __device__ inline explicit **operator __nv_bfloat162**() const

Conversion operator to `__nv_bfloat162` data type.

__host__ __device__ inline explicit **operator float2**() const

Conversion operator to `float2` data type.

Public Members

__nv_fp8x2_storage_t **__x**

Storage variable contains the vector of two scale factor values.

15.24. __nv_fp8x4_e4m3

struct **__nv_fp8x4_e4m3**

__nv_fp8x4_e4m3 datatype

This structure implements the datatype for storage and operations on the vector of four `fp8` values of `e4m3` kind each: with 1 sign, 4 exponent, 1 implicit and 3 explicit mantissa bits. The encoding doesn't support Infinity. NaNs are limited to 0x7F and 0xFF values.

Public Functions

__nv_fp8x4_e4m3() = default

Constructor by default.

__host__ __device__ inline explicit **__nv_fp8x4_e4m3**(const *__half2* flo, const *__half2* fhi)

Constructor from a pair of `__half2` data type values, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp8x4_e4m3**(const *__nv_bfloat162* flo, const *__nv_bfloat162* fhi)

Constructor from a pair of `__nv_bfloat162` data type values, relies on `__NV_SATFINITE` behavior for out-of-range values.

inline explicit __NV_SILENCE_DEPRECATION_BEGIN __host__ __device__ **__nv_fp8x4_e4m3**(const double4 f)

Constructor from `double4` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

inline explicit __NV_SILENCE_DEPRECATION_END __host__ __device__ **__nv_fp8x4_e4m3**(const double4_16a f)

Constructor from `double4_16a` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp8x4_e4m3**(const double4_32a f)

Constructor from `double4_32a` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp8x4_e4m3**(const float4 f)

Constructor from `float4` vector data type, relies on __NV_SATFINITE behavior for out-of-range values.

__host__ __device__ inline explicit **operator float4**() const

Conversion operator to `float4` vector data type.

Public Members

__nv_fp8x4_storage_t **__x**

Storage variable contains the vector of four `fp8` floating-point data values.

15.25. __nv_fp8x4_e5m2

struct **__nv_fp8x4_e5m2**

__nv_fp8x4_e5m2 datatype

This structure implements the datatype for handling four `fp8` floating-point numbers of `e5m2` kind each: with 1 sign, 5 exponent, 1 implicit and 2 explicit mantissa bits.

The structure implements converting constructors and operators.

Public Functions

__nv_fp8x4_e5m2() = default

Constructor by default.

__host__ __device__ inline explicit **__nv_fp8x4_e5m2**(const *__half2* flo, const *__half2* fhi)

Constructor from a pair of *`__half2`* data type values, relies on __NV_SATFINITE behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp8x4_e5m2**(const *__nv_bfloat162* flo, const *__nv_bfloat162* fhi)

Constructor from a pair of *`__nv_bfloat162`* data type values, relies on __NV_SATFINITE behavior for out-of-range values.

inline explicit __NV_SILENCE_DEPRECATION_BEGIN __host__ __device__ **__nv_fp8x4_e5m2**(const double4 f)

Constructor from `double4` vector data type, relies on __NV_SATFINITE behavior for out-of-range values.

inline explicit __NV_SILENCE_DEPRECATION_END __host__ __device__ **__nv_fp8x4_e5m2**(const double4_16a f)

Constructor from `double4_16a` vector data type, relies on __NV_SATFINITE behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp8x4_e5m2**(const double4_32a f)

Constructor from `double4_32a` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp8x4_e5m2**(const float4 f)

Constructor from `float4` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **operator float4**() const

Conversion operator to `float4` vector data type.

Public Members

__nv_fp8x4_storage_t **__x**

Storage variable contains the vector of four `fp8` floating-point data values.

15.26. __nv_fp8x4_e8m0

struct **__nv_fp8x4_e8m0**

__nv_fp8x4_e8m0 datatype

This structure implements the datatype for storage and operations on the vector of scale factors of `e8m0` kind each.

Public Functions

__nv_fp8x4_e8m0() = default

Constructor by default.

__host__ __device__ inline explicit **__nv_fp8x4_e8m0**(const *__half2* flo, const *__half2* fhi)

Constructor from a pair of *`__half2`* data type values, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **__nv_fp8x4_e8m0**(const *__nv_bfloat162* flo, const *__nv_bfloat162* fhi)

Constructor from a pair of *`__nv_bfloat162`* data type values, relies on `__NV_SATFINITE` behavior for out-of-range values.

inline explicit __NV_SILENCE_DEPRECATION_BEGIN __host__ __device__ **__nv_fp8x4_e8m0**(const double4 f)

Constructor from `double4` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

inline explicit __NV_SILENCE_DEPRECATION_END __host__ __device__ **__nv_fp8x4_e8m0**(const double4_16a f)

Constructor from `double4_16a` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **`__nv_fp8x4_e8m0`**(const double4_32a f)

Constructor from `double4_32a` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **`__nv_fp8x4_e8m0`**(const float4 f)

Constructor from `float4` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

__host__ __device__ inline explicit **`operator float4`**() const

Conversion operator to `float4` vector data type.

Public Members

__nv_fp8x4_storage_t **`__x`**

Storage variable contains the vector of four scale factor values.

__half

__half data type

__half2

__half2 data type

__half2_raw

__half2_raw data type

__half_raw

__half_raw data type

__nv_bfloat16

nv_bfloat16 datatype

__nv_bfloat162

nv_bfloat162 datatype

__nv_bfloat162_raw

__nv_bfloat162_raw data type

__nv_bfloat16_raw

__nv_bfloat16_raw data type

__nv_fp4_e2m1

__nv_fp4_e2m1 datatype

__nv_fp4x2_e2m1

__nv_fp4x2_e2m1 datatype

__nv_fp4x4_e2m1

__nv_fp4x4_e2m1 datatype

__nv_fp6_e2m3

__nv_fp6_e2m3 datatype

__nv_fp6_e3m2

__nv_fp6_e3m2 datatype

__nv_fp6x2_e2m3
    __nv_fp6x2_e2m3 datatype

__nv_fp6x2_e3m2
    __nv_fp6x2_e3m2 datatype

__nv_fp6x4_e2m3
    __nv_fp6x4_e2m3 datatype

__nv_fp6x4_e3m2
    __nv_fp6x4_e3m2 datatype

__nv_fp8_e4m3
    __nv_fp8_e4m3 datatype

__nv_fp8_e5m2
    __nv_fp8_e5m2 datatype

__nv_fp8_e8m0
    __nv_fp8_e8m0 datatype

__nv_fp8x2_e4m3
    __nv_fp8x2_e4m3 datatype

__nv_fp8x2_e5m2
    __nv_fp8x2_e5m2 datatype

__nv_fp8x2_e8m0
    __nv_fp8x2_e8m0 datatype

__nv_fp8x4_e4m3
    __nv_fp8x4_e4m3 datatype

__nv_fp8x4_e5m2
    __nv_fp8x4_e5m2 datatype

__nv_fp8x4_e8m0
    __nv_fp8x4_e8m0 datatype

Chapter 16. Notices

16.1. Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation ("NVIDIA") makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer ("Terms of Sale"). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

NVIDIA products are not designed, authorized, or warranted to be suitable for use in medical, military, aircraft, space, or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer's own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer's sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer's product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this document. NVIDIA accepts no liability related to any default, damage, costs, or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or

16.2. OpenCL

16.3. Trademarks

Copyright