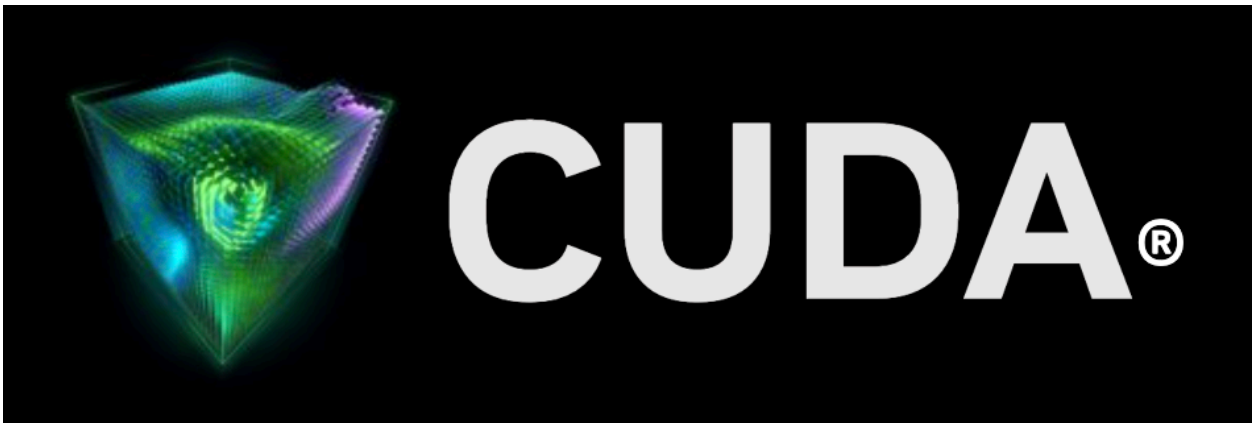

Libdevice Libary User's Guide

Release 13.4

NVIDIA Corporation

Sep 15, 2026

Contents

Chapter 1. Introduction

1.1. What Is libdevice?

The libdevice library is a collection of NVVM bitcode functions that implement common functions for NVIDIA GPU devices, including math primitives and bit-manipulation functions. These functions are optimized for particular GPU architectures, and are intended to be linked with an NVVM IR module during compilation to PTX.

This guide documents both the functions available in libdevice and the basic usage of the library from a compiler writer's perspective.

Chapter 2. Basic Usage

2.1. Linking with libdevice

The libdevice library ships as an LLVM bitcode library and is meant to be linked with the target module early in the compilation process. The standard process for linking with libdevice is to first link it with the target module, then run the standard LLVM optimization and code generation passes. This allows the optimizers to inline and perform analyses on the used library functions, and eliminate any used functions as dead code.

Users of libnvvm can link with libdevice by adding the appropriate libdevice module to the `nvvmProgram` object being compiled. In addition, the following options for `nvvmCompileProgram` affect the behavior of libdevice functions:

Table 1: Table 1. Supported Reflection Parameters

Parameter	Values	Description
`-ftz`	`0` (default)	preserve denormal values, when performing single-precision floating-point operations
	1	flush denormal values to zero, when performing single-precision floating-point operations
`-prec-div`	`0`	use a faster approximation for single-precision floating-point division and reciprocals
	1 (default)	use IEEE round-to-nearest mode for single-precision floating-point division and reciprocals
`-prec-sqrt`	`0`	use a faster approximation for single-precision floating-point square root
	1 (default)	use IEEE round-to-nearest mode for single-precision floating-point square root

The following pseudo-code shows an example of linking an NVVM IR module with the libdevice library using libnvvm:

```
nvvmProgram prog;
size_t libdeviceModSize;

const char *libdeviceMod = loadFile('/path/to/libdevice.*.bc',
                                    &libdeviceModSize);
const char *myIr = /* NVVM IR in text or binary format */;
```

(continues on next page)

(continued from previous page)

```
size_t myIrSize = /* size of myIr in bytes */;

// Create NVVM program object
nvvmCreateProgram(&prog);

// Add libdevice module to program
nvvmAddModuleToProgram(prog, libdeviceMod, libdeviceModSize);

// Add custom IR to program
nvvmAddModuleToProgram(prog, myIr, myIrSize);

// Declare compile options
const char *options[] = { "-ftz=1" };

// Compile the program
nvvmCompileProgram(prog, 1, options);
```

It is the responsibility of the client program to locate and read the libdevice library binary (represented by the `loadFile` function in the example).

Chapter 3. Functions

3.1. __nv_abs

i32 **@__nv_abs**(i32 %x)

Determine the absolute value of a 32-bit signed integer.

Determine the absolute value of the 32-bit signed integer x.

Returns

Returns the absolute value of the 32-bit signed integer x.

3.2. __nv_acos

double **@__nv_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].

- __nv_acos(1) returns +0.
- __nv_acos(x) returns NaN for x outside [-1, +1].

3.3. __nv_acosf

float **@__nv_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].

- __nv_acosf(1) returns +0.
- __nv_acosf(x) returns NaN for x outside [-1, +1].

3.4. __nv_acosh

double **@__nv_acosh**(double %x)

Calculate the nonnegative arc hyperbolic cosine of the input argument.

Calculate the nonnegative arc 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, $+\infty$].

- __nv_acosh(1) returns 0.
- __nv_acosh(x) returns NaN for x in the interval [$-\infty$, 1).

3.5. __nv_acoshf

float **@__nv_acoshf**(float %x)

Calculate the nonnegative arc hyperbolic cosine of the input argument.

Calculate the nonnegative arc 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, $+\infty$].

- __nv_acoshf(1) returns 0.
- __nv_acoshf(x) returns NaN for x in the interval [$-\infty$, 1).

3.6. __nv_asin

double **@__nv_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].

- __nv_asin(0) returns +0.
- __nv_asin(x) returns NaN for x outside [-1, +1].

3.7. __nv_asinf

float **@__nv_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 [- π /2, + π /2] for x inside [-1, +1].

- __nv_asinf(0) returns +0.
- __nv_asinf(x) returns NaN for x outside [-1, +1].

3.8. __nv_asinh

double **@__nv_asinh**(double %x)

Calculate the arc hyperbolic sine of the input argument.

Calculate the arc 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

- __nv_asinh(0) returns 1.

3.9. __nv_asinhf

float **@__nv_asinhf**(float %x)

Calculate the arc hyperbolic sine of the input argument.

Calculate the arc 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

- __nv_asinh(0) returns 1.

3.10. __nv_atan

double **@__nv_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].

- __nv_atan(0) returns +0.

3.11. __nv_atan2

double **@__nv_atan2**(double %x, double %y)

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 x / y. The quadrant of the result is determined by the signs of inputs x and y.

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].

- __nv_atan2(0, 1) returns +0.

3.12. __nv_atan2f

float **@__nv_atan2f**(float %x, float %y)

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 x / y. The quadrant of the result is determined by the signs of inputs x and y.

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].

- __nv_atan2f(0, 1) returns +0.

3.13. __nv_atanf

float **@__nv_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 [- π /2, + π /2].

- __nv_atan(0) returns +0.

3.14. __nv_atanh

double **@__nv_atanh**(double %x)

Calculate the arc hyperbolic tangent of the input argument.

Calculate the arc 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

- __nv_atanh(± 0) returns ± 0.
- __nv_atanh(± 1) returns $\pm \infty$.
- __nv_atanh(x) returns NaN for x outside interval [-1, 1].

3.15. __nv_atanhf

float **@__nv_atanhf**(float %x)

Calculate the arc hyperbolic tangent of the input argument.

Calculate the arc 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

- __nv_atanhf(± 0) returns ± 0.
- __nv_atanhf(± 1) returns $\pm \infty$.
- __nv_atanhf(x) returns NaN for x outside interval [-1, 1].

3.16. __nv_brev

i32 **@__nv_brev**(i32 %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.

3.17. __nv_brevll

i64 **@__nv_brevll**(i64 %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.

3.18. __nv_byte_perm

i32 **@__nv_byte_perm**(i32 %x, i32 %y, i32 %z)

Return selected bytes from two 32 bit unsigned integers.

__nv_byte_perm(x,y,s) 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.

The input bytes are indexed as follows: The selector indices are as follows (the upper 16-bits of the selector are not used):

Returns

The returned value r is computed to be: `result[n] := input[selector[n]]` where `result[n]` is the nth byte of r.

3.19. __nv_cbrt

double **@__nv_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 $x^{1/3}$.

- __nv_cbrt(± 0) returns ± 0.
- __nv_cbrt($\pm \infty$) returns $\pm \infty$.

3.20. __nv_cbrtf

float **@__nv_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 $x^{1/3}$.

- __nv_cbrtf(± 0) returns ± 0.
- __nv_cbrtf($\pm \infty$) returns $\pm \infty$.

3.21. __nv_ceil

double **@__nv_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.

- __nv_ceil(± 0) returns ± 0.
- __nv_ceil($\pm \infty$) returns $\pm \infty$.

3.22. __nv_ceilf

float **@__nv_ceilf**(float %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.

- __nv_ceilf(± 0) returns ± 0.

- __nv_ceilf($\pm\infty$) returns $\pm\infty$.

3.23. __nv_clz

i32 **@__nv_clz**(i32 %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.

3.24. __nv_clzll

i32 **@__nv_clzll**(i64 %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.

3.25. __nv_copysign

double **@__nv_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

Returns a value with the magnitude of x and the sign of y.

3.26. __nv_copysignf

float **@__nv_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

Returns a value with the magnitude of x and the sign of y.

3.27. __nv_cos

double **@__nv_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

- __nv_cos(± 0) returns 1.
- __nv_cos($\pm \infty$) returns NaN.

3.28. __nv_cosf

float **@__nv_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.

Returns

- __nv_cosf(± 0) returns 1.
- __nv_cosf($\pm \infty$) returns NaN.

3.29. __nv_cosh

double **@__nv_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

- __nv_cosh(0) returns 1.
- __nv_cosh($\pm\infty$) returns $+\infty$.

3.30. __nv_coshf

float **@__nv_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

- __nv_coshf(0) returns 1.
- __nv_coshf($\pm\infty$) returns $+\infty$.

3.31. __nv_cospi

double **@__nv_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

- __nv_cospi(± 0) returns 1.
- __nv_cospi($\pm\infty$) returns NaN.

3.32. __nv_cospif

float **@__nv_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

- __nv_cospif(± 0) returns 1.
- __nv_cospif($\pm\infty$) returns NaN.

3.33. __nv_cyl_bessel_i0

double **@__nv_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 reguler modified cylindrical Bessel function of order 0.

3.34. __nv_cyl_bessel_i0f

float **@__nv_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.

3.35. __nv_cyl_bessel_i1

double **@__nv_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 reguler modified cylindrical Bessel function of order 1.

3.36. __nv_cyl_bessel_i1f

float **@__nv_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 reguler modified cylindrical Bessel function.

3.37. __nv_dadd_rd

double **@__nv_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, Double-Precision Floating-Point Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns
Returns x + y.

3.38. __nv_dadd_rn

double **@__nv_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, Double-Precision Floating-Point Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns
Returns x + y.

3.39. __nv_dadd_ru

double **@__nv_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, Double-Precision Floating-Point Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns
Returns x + y.

3.40. __nv_dadd_rz

double **@__nv_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, Double-Precision Floating-Point Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns

Returns x + y.

3.41. __nv_ddiv_rd

double **@__nv_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, Double-Precision Floating-Point Functions section.

Note: Requires compute capability >= 2.0.

Returns

Returns x / y.

3.42. __nv_ddiv_rn

double **@__nv_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, Double-Precision Floating-Point Functions section.

Note: Requires compute capability >= 2.0.

Returns
Returns x / y.

3.43. __nv_ddiv_ru

double **@__nv_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, Double-Precision Floating-Point Functions section.

Note: Requires compute capability >= 2.0.

Returns
Returns x / y.

3.44. __nv_ddiv_rz

double **@__nv_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, Double-Precision Floating-Point Functions section.

Note: Requires compute capability >= 2.0.

Returns
Returns x / y.

3.45. __nv_dmul_rd

double **@__nv_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, Double-Precision Floating-Point Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns

Returns x * y.

3.46. __nv_dmul_rn

double **@__nv_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, Double-Precision Floating-Point Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns

Returns x * y.

3.47. __nv_dmul_ru

double **@__nv_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, Double-Precision Floating-Point Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns
Returns x * y.

3.48. __nv_dmul_rz

double **@__nv_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, Double-Precision Floating-Point Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns
Returns x * y.

3.49. __nv_double2float_rd

float **@__nv_double2float_rd**(double %d)

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.

3.50. __nv_double2float_rn

float **@__nv_double2float_rn**(double %d)

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.

3.51. __nv_double2float_ru

float **@__nv_double2float_ru**(double %d)

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.

3.52. __nv_double2float_rz

float **@__nv_double2float_rz**(double %d)

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.

3.53. __nv_double2hiint

i32 **@__nv_double2hiint**(double %d)

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.

3.54. __nv_double2int_rd

i32 **@__nv_double2int_rd**(double %d)

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.

Returns

Returns converted value.

3.55. __nv_double2int_rn

i32 **@__nv_double2int_rn**(double %d)

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.

Returns

Returns converted value.

3.56. __nv_double2int_ru

i32 **@__nv_double2int_ru**(double %d)

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.

Returns

Returns converted value.

3.57. __nv_double2int_rz

i32 **@__nv_double2int_rz**(double %d)

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.

Returns

Returns converted value.

3.58. __nv_double2ll_rd

i64 **@__nv_double2ll_rd**(double %f)

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.

Returns

Returns converted value.

3.59. __nv_double2ll_rn

i64 **@__nv_double2ll_rn**(double %f)

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.

Returns

Returns converted value.

3.60. __nv_double2ll_ru

i64 **@__nv_double2ll_ru**(double %f)

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.

Returns

Returns converted value.

3.61. __nv_double2ll_rz

i64 **@__nv_double2ll_rz**(double %f)

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.

Returns

Returns converted value.

3.62. __nv_double2loint

i32 **@__nv_double2loint**(double %d)

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.

3.63. __nv_double2uint_rd

i32 **@__nv_double2uint_rd**(double %d)

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.

Returns

Returns converted value.

3.64. __nv_double2uint_rn

i32 **@__nv_double2uint_rn**(double %d)

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.

Returns

Returns converted value.

3.65. __nv_double2uint_ru

i32 **@__nv_double2uint_ru**(double %d)

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.

Returns

Returns converted value.

3.66. __nv_double2uint_rz

i32 **@__nv_double2uint_rz**(double %d)

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.

Returns

Returns converted value.

3.67. __nv_double2ull_rd

i64 **@__nv_double2ull_rd**(double %f)

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.

Returns

Returns converted value.

3.68. __nv_double2ull_rn

i64 **@__nv_double2ull_rn**(double %f)

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.

Returns

Returns converted value.

3.69. __nv_double2ull_ru

i64 **@__nv_double2ull_ru**(double %f)

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.

Returns

Returns converted value.

3.70. __nv_double2ull_rz

i64 **@__nv_double2ull_rz**(double %f)

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.

Returns

Returns converted value.

3.71. __nv_double_as_longlong

i64 **@__nv_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.

3.72. __nv_drcp_rd

double **@__nv_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, Double-Precision Floating-Point Functions section.

Note: Requires compute capability >= 2.0.

Returns

Returns $\frac{1}{x}$.

3.73. __nv_drcp_rn

double **@__nv_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, Double-Precision Floating-Point Functions section.

Note: Requires compute capability >= 2.0.

Returns

Returns $\frac{1}{x}$.

3.74. __nv_drcp_ru

double **@__nv_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, Double-Precision Floating-Point Functions section.

Note: Requires compute capability >= 2.0.

Returns
Returns $\frac{1}{x}$.

3.75. __nv_drcp_rz

double **@__nv_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, Double-Precision Floating-Point Functions section.

Note: Requires compute capability >= 2.0.

Returns
Returns $\frac{1}{x}$.

3.76. __nv_dsqrt_rd

double **@__nv_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, Double-Precision Floating-Point Functions section.

Note: Requires compute capability >= 2.0.

Returns
Returns $\sqrt{x}$.

3.77. __nv_dsqrt_rn

double **@__nv_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, Double-Precision Floating-Point Functions section.

Note: Requires compute capability >= 2.0.

Returns
Returns $\sqrt{x}$.

3.78. __nv_dsqrt_ru

double **@__nv_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, Double-Precision Floating-Point Functions section.

Note: Requires compute capability >= 2.0.

Returns
Returns $\sqrt{x}$.

3.79. __nv_dsqrt_rz

double **@__nv_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, Double-Precision Floating-Point Functions section.

Note: Requires compute capability >= 2.0.

Returns

Returns $\sqrt{x}$.

3.80. __nv_dsub_rd

double **@__nv_dsub_rd**(double %a, double %b)

3.81. __nv_dsub_rn

double **@__nv_dsub_rn**(double %a, double %b)

3.82. __nv_dsub_ru

double **@__nv_dsub_ru**(double %a, double %b)

3.83. __nv_dsub_rz

double **@__nv_dsub_rz**(double %a, double %b)

3.84. __nv_erf

double **@__nv_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

- __nv_erf(± 0) returns ± 0.
- __nv_erf($\pm \infty$) returns ± 1.

3.85. __nv_erfc

double **@__nv_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

- __nv_erfc($-\infty$) returns 2.
- __nv_erfc($+\infty$) returns +0.

3.86. __nv_erfcf

float **@__nv_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

- __nv_erfcf($-\infty$) returns 2.
- __nv_erfcf($+\infty$) returns +0.

3.87. __nv_erfcinv

double **@__nv_erfcinv**(double %x)

Calculate the inverse complementary error function of the input argument.

Calculate the inverse complementary error function of the input argument y, for y in the interval [0, 2]. The inverse complementary error function find the value x that satisfies the equation y = erfc(x), for $0 \leq y \leq 2$, and $-\infty \leq x \leq \infty$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- __nv_erfcinv(0) returns $+\infty$.
- __nv_erfcinv(2) returns $-\infty$.

3.88. __nv_erfcinvf

float **@__nv_erfcinvf**(float %x)

Calculate the inverse complementary error function of the input argument.

Calculate the inverse complementary error function of the input argument y, for y in the interval [0, 2]. The inverse complementary error function find the value x that satisfies the equation y = erfc(x), for $0 \leq y \leq 2$, and $-\infty \leq x \leq \infty$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

- __nv_erfcinvf(0) returns $+\infty$.
- __nv_erfcinvf(2) returns $-\infty$.

3.89. __nv_erfcx

double **@__nv_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

- __nv_erfcx($-\infty$) returns $+\infty$
- __nv_erfcx($+\infty$) returns +0
- __nv_erfcx(x) returns $+\infty$ if the correctly calculated value is outside the double floating point range.

3.90. __nv_erfcxf

float **@__nv_erfcxf**(float %x)

Calculate the scaled complementary error function of the input argument.

Calculate the scaled complementary error function of the input argument x, $e^{x^2} \cdot \text{erfc}(x)$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

- __nv_erfcxf($-\infty$) returns $+\infty$
- __nv_erfcxf($+\infty$) returns +0
- __nv_erfcxf(x) returns $+\infty$ if the correctly calculated value is outside the double floating point range.

3.91. __nv_erff

float **@__nv_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

- __nv_erff(± 0) returns ± 0.
- __nv_erff($\pm \infty$) returns ± 1.

3.92. __nv_erfinv

double **@__nv_erfinv**(double %x)

Calculate the inverse error function of the input argument.

Calculate the inverse error function of the input argument y, for y in the interval [-1, 1]. The inverse error function finds the value x that satisfies the equation y = erf(x), for $-1 \leq y \leq 1$, and $-\infty \leq x \leq \infty$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- __nv_erfinv(1) returns $+\infty$.
- __nv_erfinv(-1) returns $-\infty$.

3.93. __nv_erfinvf

float **@__nv_erfinvf**(float %x)

Calculate the inverse error function of the input argument.

Calculate the inverse error function of the input argument y, for y in the interval [-1, 1]. The inverse error function finds the value x that satisfies the equation y = erf(x), for $-1 \leq y \leq 1$, and $-\infty \leq x \leq \infty$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

- __nv_erfinvf(1) returns $+\infty$.
- __nv_erfinvf(-1) returns $-\infty$.

3.94. __nv_exp

double **@__nv_exp**(double %x)

Calculate the base e exponential of the input argument.

Calculate 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

Returns e^x.

3.95. __nv_exp10

double **@__nv_exp10**(double %x)

Calculate the base 10 exponential of the input argument.

Calculate 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

Returns 10^x.

3.96. __nv_exp10f

float **@__nv_exp10f**(float %x)

Calculate the base 10 exponential of the input argument.

Calculate 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.

Returns

Returns 10^x.

3.97. __nv_exp2

double **@__nv_exp2**(double %x)

Calculate the base 2 exponential of the input argument.

Calculate 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

Returns 2^x.

3.98. __nv_exp2f

float **@__nv_exp2f**(float %x)

Calculate the base 2 exponential of the input argument.

Calculate 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

Returns 2^x.

3.99. __nv_expf

float **@__nv_expf**(float %x)

Calculate the base e exponential of the input argument.

Calculate 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.

Returns

Returns e^x.

3.100. __nv_expm1

double **@__nv_expm1**(double %x)

Calculate the base e exponential of the input argument, minus 1.

Calculate 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

Returns $e^x - 1$.

3.101. __nv_expm1f

float **@__nv_expm1f**(float %x)

Calculate the base e exponential of the input argument, minus 1.

Calculate 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

Returns $e^x - 1$.

3.102. __nv_fabs

double **@__nv_fabs**(double %f)

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.

- __nv_fabs($\pm\infty$) returns $+\infty$.
- __nv_fabs(± 0) returns 0.

3.103. __nv_fabsf

float **@__nv_fabsf**(float %f)

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.

- __nv_fabsf($\pm\infty$) returns $+\infty$.
- __nv_fabsf(± 0) returns 0.

3.104. __nv_fadd_rd

float **@__nv_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, Single-Precision Floating-Point Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns
Returns x + y.

3.105. __nv_fadd_rn

float **@__nv_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, Single-Precision Floating-Point Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns
Returns x + y.

3.106. __nv_fadd_ru

float **@__nv_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, Single-Precision Floating-Point Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns
Returns x + y.

3.107. __nv_fadd_rz

float **@__nv_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, Single-Precision Floating-Point Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns

Returns x + y.

3.108. __nv_fast_cosf

float **@__nv_fast_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.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: Input and output in the denormal range is flushed to sign preserving 0.0.

Returns

Returns the approximate cosine of x.

3.109. __nv_fast_exp10f

float **@__nv_fast_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.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: Most input and output values around denormal range are flushed to sign preserving 0.0.

Returns
Returns an approximation to 10^x.

3.110. __nv_fast_expf

float **@__nv_fast_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.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: Most input and output values around denormal range are flushed to sign preserving 0.0.

Returns
Returns an approximation to e^x.

3.111. __nv_fast_fdividef

float **@__nv_fast_fdividef**(float %x, float %y)

Calculate the fast approximate division of the input arguments.

Calculate the fast approximate division of x by y.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Returns
Returns x / y.

- __nv_fast_fdividef(∞ , y) returns NaN for $2^{126} < y < 2^{128}$.
- __nv_fast_fdividef(x, y) returns 0 for $2^{126} < y < 2^{128}$ and $x \neq \infty$.

3.112. __nv_fast_log10f

float **@__nv_fast_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.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: Most input and output values around denormal range are flushed to sign preserving 0.0.

Returns

Returns an approximation to $\log_{10}(x)$.

3.113. __nv_fast_log2f

float **@__nv_fast_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.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: Input and output in the denormal range is flushed to sign preserving 0.0.

Returns

Returns an approximation to $\log_2(x)$.

3.114. __nv_fast_logf

float **@__nv_fast_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.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: Most input and output values around denormal range are flushed to sign preserving 0.0.

Returns

Returns an approximation to $\log_e(x)$.

3.115. __nv_fast_powf

float **@__nv_fast_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.

Note: Most input and output values around denormal range are flushed to sign preserving 0.0.

Returns

Returns an approximation to x^y.

3.116. __nv_fast_sincosf

void **@__nv_fast_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, `zptr`.

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.

Returns

- none

3.117. __nv_fast_sinf

float **@__nv_fast_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.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.

Note: Input and output in the denormal range is flushed to sign preserving 0.0.

Returns

Returns the approximate sine of x.

3.118. __nv_fast_tanf

float **@__nv_fast_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 __nv_sinf() by __nv_cosf(). Denormal input and output are flushed to sign-preserving 0.0 at each step of the computation.

Returns

Returns the approximate tangent of x.

3.119. __nv_fast_tanhf

float **@__nv_fast_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.

3.120. __nv_fdim

double **@__nv_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, Single-Precision Floating-Point Functions section.

Returns
Returns the positive difference between x and y.

- __nv_fdim(x, y) returns x - y if x > y.
- __nv_fdim(x, y) returns +0 if x $\leq$ y.

3.121. __nv_fdimf

float **@__nv_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.

- __nv_fdimf(x, y) returns x - y if x > y.
- __nv_fdimf(x, y) returns +0 if x $\leq$ y.

3.122. __nv_fdiv_rd

float **@__nv_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, Single-Precision Floating-Point Functions section.

Returns

Returns x / y.

3.123. __nv_fdiv_rn

float **@__nv_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, Single-Precision Floating-Point Functions section.

Returns

Returns x / y.

3.124. __nv_fdiv_ru

float **@__nv_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, Single-Precision Floating-Point Functions section.

Returns

Returns x / y.

3.125. __nv_fdiv_rz

float **@__nv_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, Single-Precision Floating-Point Functions section.

Returns

Returns x / y.

3.126. __nv_ffs

i32 **@__nv_ffs**(i32 %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.

- __nv_ffs(0) returns 0.

3.127. __nv_ffsll

i32 **@__nv_ffsll**(i64 %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.

- __nv_ffsll(0) returns 0.

3.128. __nv_finitef

i32 **@__nv_finitef**(float %x)

Determine whether argument is finite.

Determine whether the floating-point value x is a finite value.

Returns

Returns a non-zero value if and only if x is a finite value.

3.129. __nv_float2half_rn

i16 **@__nv_float2half_rn**(float %f)

Convert a single-precision float to a half-precision float in round-to-nearest-even mode.

Convert the single-precision float value x to a half-precision floating point value represented in `unsigned short` format, in round-to-nearest-even mode.

Returns

Returns converted value.

3.130. __nv_float2int_rd

i32 **@__nv_float2int_rd**(float %in)

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.

Returns

Returns converted value.

3.131. __nv_float2int_rn

i32 **@__nv_float2int_rn**(float %in)

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.

Returns

Returns converted value.

3.132. __nv_float2int_ru

i32 **@__nv_float2int_ru**(float %in)

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.

Returns

Returns converted value.

3.133. __nv_float2int_rz

i32 **@__nv_float2int_rz**(float %in)

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.

Returns

Returns converted value.

3.134. __nv_float2ll_rd

i64 **@__nv_float2ll_rd**(float %f)

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.

Returns

Returns converted value.

3.135. __nv_float2ll_rn

i64 **@__nv_float2ll_rn**(float %f)

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.

Returns

Returns converted value.

3.136. __nv_float2ll_ru

i64 **@__nv_float2ll_ru**(float %f)

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.

Returns

Returns converted value.

3.137. __nv_float2ll_rz

i64 **@__nv_float2ll_rz**(float %f)

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.

Returns

Returns converted value.

3.138. __nv_float2uint_rd

i32 **@__nv_float2uint_rd**(float %in)

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.

Returns

Returns converted value.

3.139. __nv_float2uint_rn

i32 **@__nv_float2uint_rn**(float %in)

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.

Returns

Returns converted value.

3.140. __nv_float2uint_ru

i32 **@__nv_float2uint_ru**(float %in)

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.

Returns

Returns converted value.

3.141. __nv_float2uint_rz

i32 **@__nv_float2uint_rz**(float %in)

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.

Returns

Returns converted value.

3.142. __nv_float2ull_rd

i64 **@__nv_float2ull_rd**(float %f)

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.

Returns

Returns converted value.

3.143. __nv_float2ull_rn

i64 **@__nv_float2ull_rn**(float %f)

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.

Returns

Returns converted value.

3.144. __nv_float2ull_ru

i64 **@__nv_float2ull_ru**(float %f)

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.

Returns

Returns converted value.

3.145. __nv_float2ull_rz

i64 **@__nv_float2ull_rz**(float %f)

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.

Returns

Returns converted value.

3.146. __nv_float_as_int

i32 **@__nv_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.

3.147. __nv_float_as_uint

i32 **@__nv_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.

3.148. __nv_floor

double **@__nv_floor**(double %f)

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 the largest integer value which is less than or equal to x expressed as a floating-point number.

- __nv_floor($\pm\infty$) returns $\pm\infty$.
- __nv_floor(± 0) returns ± 0.

3.149. __nv_floorf

float **@__nv_floorf**(float %f)

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 the largest integer value which is less than or equal to x expressed as a floating-point number.

- __nv_floorf($\pm\infty$) returns $\pm\infty$.
- __nv_floorf(± 0) returns ± 0.

3.150. __nv_fma

double **@__nv_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.

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.

- __nv_fma($\pm\infty$, ± 0 , z) returns NaN.
- __nv_fma(± 0 , $\pm\infty$, z) returns NaN.
- __nv_fma(x, y, $-\infty$) returns NaN if $x \times y$ is an exact $+\infty$.
- __nv_fma(x, y, $+\infty$) returns NaN if $x \times y$ is an exact $-\infty$.

3.151. __nv_fma_rd

double **@__nv_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, Double-Precision Floating-Point Functions section.

Returns

Returns the rounded value of $x \times y + z$ as a single operation.

- __nv_fma_rd($\pm\infty$, ± 0 , z) returns NaN.
- __nv_fma_rd(± 0 , $\pm\infty$, z) returns NaN.
- __nv_fma_rd(x, y, $-\infty$) returns NaN if $x \times y$ is an exact $+\infty$
- __nv_fma_rd(x, y, $+\infty$) returns NaN if $x \times y$ is an exact $-\infty$

3.152. __nv_fma_rn

double **@__nv_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, Double-Precision Floating-Point Functions section.

Returns

Returns the rounded value of $x \times y + z$ as a single operation.

- __nv_fma_rn($\pm\infty$, ± 0 , z) returns NaN.
- __nv_fma_rn(± 0 , $\pm\infty$, z) returns NaN.
- __nv_fma_rn(x, y, $-\infty$) returns NaN if $x \times y$ is an exact $+\infty$
- __nv_fma_rn(x, y, $+\infty$) returns NaN if $x \times y$ is an exact $-\infty$

3.153. __nv_fma_ru

double **@__nv_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, Double-Precision Floating-Point Functions section.

Returns

Returns the rounded value of $x \times y + z$ as a single operation.

- __nv_fma_ru($\pm\infty$, ± 0 , z) returns NaN.
- __nv_fma_ru(± 0 , $\pm\infty$, z) returns NaN.
- __nv_fma_ru(x, y, $-\infty$) returns NaN if $x \times y$ is an exact $+\infty$
- __nv_fma_ru(x, y, $+\infty$) returns NaN if $x \times y$ is an exact $-\infty$

3.154. __nv_fma_rz

double **@__nv_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, Double-Precision Floating-Point Functions section.

Returns

Returns the rounded value of $x \times y + z$ as a single operation.

- __nv_fma_rz($\pm\infty$, ± 0 , z) returns NaN.
- __nv_fma_rz(± 0 , $\pm\infty$, z) returns NaN.

- __nv_fma_rz(x, y, $-\infty$) returns NaN if $x \times y$ is an exact $+\infty$
- __nv_fma_rz(x, y, $+\infty$) returns NaN if $x \times y$ is an exact $-\infty$

3.155. __nv_fmaf

float **@__nv_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.

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.

- __nv_fmaf($\pm\infty$, ± 0 , z) returns NaN.
- __nv_fmaf(± 0 , $\pm\infty$, z) returns NaN.
- __nv_fmaf(x, y, $-\infty$) returns NaN if $x \times y$ is an exact $+\infty$.
- __nv_fmaf(x, y, $+\infty$) returns NaN if $x \times y$ is an exact $-\infty$.

3.156. __nv_fmaf_ieee_rd

float **@__nv_fmaf_ieee_rd**(float %x, float %y, float %z)

DOCUMENTATION MISSING.

3.157. __nv_fmaf_ieee_rn

float **@__nv_fmaf_ieee_rn**(float %x, float %y, float %z)

DOCUMENTATION MISSING.

3.158. __nv_fmaf_ieee_ru

float **@__nv_fmaf_ieee_ru**(float %x, float %y, float %z)

DOCUMENTATION MISSING.

3.159. __nv_fmaf_ieee_rz

float **@__nv_fmaf_ieee_rz**(float %x, float %y, float %z)

DOCUMENTATION MISSING.

3.160. __nv_fmaf_rd

float **@__nv_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, Single-Precision Floating-Point Functions section.

Returns

Returns the rounded value of $x \times y + z$ as a single operation.

- __nv_fmaf_rd($\pm\infty$, ± 0 , z) returns NaN.
- __nv_fmaf_rd(± 0 , $\pm\infty$, z) returns NaN.
- __nv_fmaf_rd(x, y, $-\infty$) returns NaN if $x \times y$ is an exact $+\infty$.
- __nv_fmaf_rd(x, y, $+\infty$) returns NaN if $x \times y$ is an exact $-\infty$.

3.161. __nv_fmaf_rn

float **@__nv_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, Single-Precision Floating-Point Functions section.

Returns

Returns the rounded value of $x \times y + z$ as a single operation.

- __nv_fmaf_rn($\pm\infty$, ± 0 , z) returns NaN.
- __nv_fmaf_rn(± 0 , $\pm\infty$, z) returns NaN.
- __nv_fmaf_rn(x, y, $-\infty$) returns NaN if $x \times y$ is an exact $+\infty$.
- __nv_fmaf_rn(x, y, $+\infty$) returns NaN if $x \times y$ is an exact $-\infty$.

3.162. __nv_fmaf_ru

float **@__nv_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, Single-Precision Floating-Point Functions section.

Returns

Returns the rounded value of $x \times y + z$ as a single operation.

- __nv_fmaf_ru($\pm\infty$, ± 0 , z) returns NaN.
- __nv_fmaf_ru(± 0 , $\pm\infty$, z) returns NaN.
- __nv_fmaf_ru(x, y, $-\infty$) returns NaN if $x \times y$ is an exact $+\infty$.
- __nv_fmaf_ru(x, y, $+\infty$) returns NaN if $x \times y$ is an exact $-\infty$.

3.163. __nv_fmaf_rz

float **@__nv_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, Single-Precision Floating-Point Functions section.

Returns

Returns the rounded value of $x \times y + z$ as a single operation.

- __nv_fmaf_rz($\pm\infty$, ± 0 , z) returns NaN.
- __nv_fmaf_rz(± 0 , $\pm\infty$, z) returns NaN.

- __nv_fmaf_rz(x, y, $-\infty$) returns NaN if $x \times y$ is an exact $+\infty$.
- __nv_fmaf_rz(x, y, $+\infty$) returns NaN if $x \times y$ is an exact $-\infty$.

3.164. __nv_fmax

double **@__nv_fmax**(double %x, double %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, 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.

3.165. __nv_fmaxf

float **@__nv_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, 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.

3.166. __nv_fmin

double **@__nv_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 values of the arguments x and y.

- If both arguments are NaN, returns NaN.
- If one argument is NaN, returns the numeric argument.

3.167. __nv_fminf

float **@__nv_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, Double-Precision Floating-Point Functions section.

Returns

Returns the minimum numeric values of the arguments x and y.

- If both arguments are NaN, returns NaN.
- If one argument is NaN, returns the numeric argument.

3.168. __nv_fmod

double **@__nv_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.
- __nv_fmod(± 0 , y) returns ± 0 if y is not zero.
- __nv_fmod(x, $\pm \infty$) returns x if x is finite.
- __nv_fmod(x, y) returns NaN if x is $\pm \infty$ or y is zero.
- If either argument is NaN, NaN is returned.

3.169. __nv_fmodf

float **@__nv_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.
- __nv_fmodf(± 0 , y) returns ± 0 if y is not zero.
- __nv_fmodf(x, $\pm \infty$) returns x if x is finite.
- __nv_fmodf(x, y) returns NaN if x is $\pm \infty$ or y is zero.
- If either argument is NaN, NaN is returned.

3.170. __nv_fmul_rd

float **@__nv_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, Single-Precision Floating-Point Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns

Returns x * y.

3.171. __nv_fmul_rn

float **@__nv_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, Single-Precision Floating-Point Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns

Returns x * y.

3.172. __nv_fmul_ru

float **@__nv_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, Single-Precision Floating-Point Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns
Returns x * y.

3.173. __nv_fmul_rz

float **@__nv_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, Single-Precision Floating-Point Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns
Returns x * y.

3.174. __nv_frcp_rd

float **@__nv_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, Single-Precision Floating-Point Functions section.

Returns
Returns $\frac{1}{x}$.

3.175. __nv_frcp_rn

float **@__nv_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, Single-Precision Floating-Point Functions section.

Returns

Returns $\frac{1}{x}$.

3.176. __nv_frcp_ru

float **@__nv_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, Single-Precision Floating-Point Functions section.

Returns

Returns $\frac{1}{x}$.

3.177. __nv_frcp_rz

float **@__nv_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, Single-Precision Floating-Point Functions section.

Returns

Returns $\frac{1}{x}$.

3.178. __nv_frexp

double **@__nv_frexp**(double %x, i32* %b)

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.

- __nv_frexp(0, nptr) returns 0 for the fractional component and zero for the integer component.
- __nv_frexp(± 0 , nptr) returns ± 0 and stores zero in the location pointed to by nptr.
- __nv_frexp($\pm \infty$, nptr) returns $\pm \infty$ and stores an unspecified value in the location to which nptr points.
- __nv_frexp(NaN, y) returns a NaN and stores an unspecified value in the location to which nptr points.

3.179. __nv_frexpf

float **@__nv_frexpf**(float %x, i32* %b)

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, Single-Precision Floating-Point Functions section.

Returns

Returns the fractional component m.

- __nv_frexpf(0, nptr) returns 0 for the fractional component and zero for the integer component.
- __nv_frexpf(± 0 , nptr) returns ± 0 and stores zero in the location pointed to by nptr.

- __nv_frexpf($\pm\infty$, `nptr`) returns $\pm\infty$ and stores an unspecified value in the location to which `nptr` points.
- __nv_frexpf(NaN, y) returns a NaN and stores an unspecified value in the location to which `nptr` points.

3.180. __nv_frsqrt_rn

float **@__nv_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, Single-Precision Floating-Point Functions section.

Returns

Returns $1/\sqrt{x}$.

3.181. __nv_fsqrt_rd

float **@__nv_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, Single-Precision Floating-Point Functions section.

Returns

Returns $\sqrt{x}$.

3.182. __nv_fsqrt_rn

float **@__nv_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, Single-Precision Floating-Point Functions section.

Returns
Returns $\sqrt{x}$.

3.183. __nv_fsqrt_ru

float **@__nv_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, Single-Precision Floating-Point Functions section.

Returns
Returns $\sqrt{x}$.

3.184. __nv_fsqrt_rz

float **@__nv_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, Single-Precision Floating-Point Functions section.

Returns
Returns $\sqrt{x}$.

3.185. __nv_fsub_rd

float **@__nv_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, Single-Precision Floating-Point Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns
Returns x - y.

3.186. __nv_fsub_rn

float **@__nv_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, Single-Precision Floating-Point Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns
Returns x - y.

3.187. __nv_fsub_ru

float **@__nv_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, Single-Precision Floating-Point Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns
Returns x - y.

3.188. __nv_fsub_rz

float **@__nv_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, Single-Precision Floating-Point Functions section.

Note: This operation will never be merged into a single multiply-add instruction.

Returns

Returns x - y.

3.189. __nv_hadd

i32 **@__nv_hadd**(i32 %x, i32 %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.

3.190. __nv_half2float

float **@__nv_half2float**(i16 %h)

Convert a half-precision float to a single-precision float in round-to-nearest-even mode.

Convert the half-precision floating point value `x` represented in `unsigned short` format to a single-precision floating point value.

Returns

Returns converted value.

3.191. __nv_hiloint2double

double **@__nv_hiloint2double**(i32 %x, i32 %y)

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.

3.192. __nv_hypot

double **@__nv_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 $\sqrt{x^2+y^2}$. If the correct value would overflow, returns $+\infty$. If the correct value would underflow, returns 0. If one of the input arguments is 0, returns the other argument

3.193. __nv_hypotf

float **@__nv_hypotf**(float %x, float %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, Single-Precision Floating-Point Functions section.

Returns

Returns the length of the hypotenuse $\sqrt{x^2+y^2}$. If the correct value would overflow, returns $+\infty$. If the correct value would underflow, returns 0. If one of the input arguments is 0, returns the other argument

3.194. __nv_ilogb

i32 **@__nv_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.
- __nv_ilogb(0) returns `INT_MIN`.
- __nv_ilogb(NaN) returns `INT_MIN`.
- __nv_ilogb(x) returns `INT_MAX` if x is ∞ or the correct value is greater than `INT_MAX`.
- __nv_ilogb(x) return `INT_MIN` if the correct value is less than `INT_MIN`.

3.195. __nv_ilogbf

i32 **@__nv_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.
- __nv_ilogbf(0) returns `INT_MIN`.
- __nv_ilogbf(NaN) returns `INT_MIN`.
- __nv_ilogbf(x) returns `INT_MAX` if x is ∞ or the correct value is greater than `INT_MAX`.
- __nv_ilogbf(x) return `INT_MIN` if the correct value is less than `INT_MIN`.

3.196. __nv_int2double_rn

double **@__nv_int2double_rn**(i32 %i)

Convert a signed int to a double.

Convert the signed integer value x to a double-precision floating point value.

Returns

Returns converted value.

3.197. __nv_int2float_rd

float **@__nv_int2float_rd**(i32 %in)

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.

3.198. __nv_int2float_rn

float **@__nv_int2float_rn**(i32 %in)

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.

3.199. __nv_int2float_ru

float **@__nv_int2float_ru**(i32 %in)

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.

3.200. __nv_int2float_rz

float **@__nv_int2float_rz**(i32 %in)

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.

3.201. __nv_int_as_float

float **@__nv_int_as_float**(i32 %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.

3.202. __nv_isfinited

i32 **@__nv_isfinited**(double %x)

Determine whether argument is finite.

Determine whether the floating-point value x is a finite value (zero, subnormal, or normal and not infinity or NaN).

Returns

Returns a nonzero value if and only if x is a finite value.

3.203. __nv_isinfd

i32 **@__nv_isinfd**(double %x)

Determine whether argument is infinite.

Determine whether the floating-point value x is an infinite value (positive or negative).

Returns

Returns a nonzero value if and only if x is a infinite value.

3.204. __nv_isinff

i32 **@__nv_isinff**(float %x)

Determine whether argument is infinite.

Determine whether the floating-point value x is an infinite value (positive or negative).

Returns

Returns a nonzero value if and only if x is a infinite value.

3.205. __nv_isnand

i32 **@__nv_isnand**(double %x)

Determine whether argument is a NaN.

Determine whether the floating-point value x is a NaN.

Returns

Returns a nonzero value if and only if x is a NaN value.

3.206. __nv_isnanf

i32 **@__nv_isnanf**(float %x)

Determine whether argument is a NaN.

Determine whether the floating-point value x is a NaN.

Returns

Returns a nonzero value if and only if x is a NaN value.

3.207. __nv_j0

double **@__nv_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.

- __nv_j0($\pm\infty$) returns +0.

- __nv_j0(NaN) returns NaN.

3.208. __nv_j0f

float **@__nv_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.

- __nv_j0f($\pm\infty$) returns +0.
- __nv_j0f(NaN) returns NaN.

3.209. __nv_j1

double **@__nv_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.

- __nv_j1(± 0) returns ± 0.
- __nv_j1($\pm\infty$) returns ± 0.
- __nv_j1(NaN) returns NaN.

3.210. __nv_j1f

float **@__nv_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.

- __nv_j1f(± 0) returns ± 0.
- __nv_j1f($\pm \infty$) returns ± 0.
- __nv_j1f(NaN) returns NaN.

3.211. __nv_jn

double **@__nv_jn**(i32 %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.

- __nv_jn(n, NaN) returns NaN.
- __nv_jn(n, x) returns NaN for n < 0.
- __nv_jn(n, $+\infty$) returns +0.

3.212. __nv_jnf

float **@__nv_jnf**(i32 %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.

- __nv_jnf(n, NaN) returns NaN.
- __nv_jnf(n, x) returns NaN for n < 0.
- __nv_jnf(n, $+\infty$) returns +0.

3.213. __nv_ldexp

double **@__nv_ldexp**(double %x, i32 %y)

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

- __nv_ldexp(x) returns $\pm\infty$ if the correctly calculated value is outside the double floating point range.

3.214. __nv_ldexpf

float **@__nv_ldexpf**(float %x, i32 %y)

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

- __nv_ldexpf(x) returns $\pm\infty$ if the correctly calculated value is outside the double floating point range.

3.215. __nv_lgamma

double **@__nv_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 $\log_e \left|\int_0^\infty e^{-t} t^{x-1} dt\right|$

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- __nv_lgamma(1) returns +0.
- __nv_lgamma(2) returns +0.
- __nv_lgamma(x) returns $\pm\infty$ if the correctly calculated value is outside the double floating point range.
- __nv_lgamma(x) returns $+\infty$ if x $\leq$ 0.
- __nv_lgamma($-\infty$) returns $-\infty$.
- __nv_lgamma($+\infty$) returns $+\infty$.

3.216. __nv_lgammaf

float **@__nv_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 $\log_e \left|\int_0^\infty e^{-t} t^{x-1} dt\right|$

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

- __nv_lgammaf(1) returns +0.

- __nv_lgammaf(2) returns +0.
- __nv_lgammaf(x) returns $\pm\infty$ if the correctly calculated value is outside the double floating point range.
- __nv_lgammaf(x) returns $+\infty$ if x $\leq$ 0.
- __nv_lgammaf($-\infty$) returns $-\infty$.
- __nv_lgammaf($+\infty$) returns $+\infty$.

3.217. __nv_ll2double_rd

double **@__nv_ll2double_rd**(i64 %l)

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.

3.218. __nv_ll2double_rn

double **@__nv_ll2double_rn**(i64 %l)

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.

3.219. __nv_ll2double_ru

double **@__nv_ll2double_ru**(i64 %l)

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.

3.220. __nv_ll2double_rz

double **@__nv_ll2double_rz**(i64 %l)

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.

3.221. __nv_ll2float_rd

float **@__nv_ll2float_rd**(i64 %l)

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.

3.222. __nv_ll2float_rn

float **@__nv_ll2float_rn**(i64 %l)

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.

3.223. __nv_ll2float_ru

float **@__nv_ll2float_ru**(i64 %l)

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.

3.224. __nv_ll2float_rz

float **@__nv_ll2float_rz**(i64 %l)

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.

3.225. __nv_llabs

i64 **@__nv_llabs**(i64 %x)

Determine the absolute value of a 64-bit signed integer.

Determine the absolute value of the 64-bit signed integer x.

Returns

Returns the absolute value of the 64-bit signed integer x.

3.226. __nv_llmax

i64 **@__nv_llmax**(i64 %x, i64 %y)

Determine the maximum value of two 64-bit signed integers.

Determine the maximum value of the two 64-bit signed integers x and y.

Returns

Returns the maximum value of the two 64-bit signed integers x and y.

3.227. __nv_llmin

i64 **@__nv_llmin**(i64 %x, i64 %y)

Determine the minimum value of two 64-bit signed integers.

Determine the minimum value of the two 64-bit signed integers x and y.

Returns

Returns the minimum value of the two 64-bit signed integers x and y.

3.228. __nv_llrint

i64 **@__nv_llrint**(double %x)

Round input to nearest integer value.

Round x to the nearest integer value, with halfway cases rounded towards zero. If the result is outside the range of the return type, the result is undefined.

Returns

Returns rounded integer value.

3.229. __nv_llrintf

i64 **@__nv_llrintf**(float %x)

Round input to nearest integer value.

Round x to the nearest integer value, with halfway cases rounded towards zero. If the result is outside the range of the return type, the result is undefined.

Returns

Returns rounded integer value.

3.230. __nv_llround

i64 **@__nv_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 result is undefined.

Note: This function may be slower than alternate rounding methods. See ::llrint().

Returns

Returns rounded integer value.

3.231. __nv_llroundf

i64 **@__nv_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 result is undefined.

Note: This function may be slower than alternate rounding methods. See ::llrint().

Returns
Returns rounded integer value.

3.232. __nv_log

double **@__nv_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

- __nv_log(± 0) returns $-\infty$.
- __nv_log(1) returns +0.
- __nv_log(x) returns NaN for x < 0.
- __nv_log($+\infty$) returns $+\infty$

3.233. __nv_log10

double **@__nv_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

- __nv_log10(± 0) returns $-\infty$.
- __nv_log10(1) returns +0.
- __nv_log10(x) returns NaN for x < 0.
- __nv_log10($+\infty$) returns $+\infty$.

3.234. __nv_log10f

float **@__nv_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.

Returns

- __nv_log10f(± 0) returns $-\infty$.
- __nv_log10f(1) returns +0.
- __nv_log10f(x) returns NaN for x < 0.
- __nv_log10f($+\infty$) returns $+\infty$.

3.235. __nv_log1p

double **@__nv_log1p**(double %x)

Calculate the value of $log_e(1 + x)$ $\lfloor x \rfloor$.

Calculate the value of $log_e(1 + x)$ $\lfloor x \rfloor$ 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

- __nv_log1p(± 0) returns $-\infty$.
- __nv_log1p(-1) returns +0.
- __nv_log1p(x) returns NaN for x < -1.
- __nv_log1p($+\infty$) returns $+\infty$.

3.236. __nv_log1pf

float **@__nv_log1pf**(float %x)

Calculate the value of $log_e(1 + x)$ $\lfloor x \rfloor$.

Calculate the value of $log_e(1 + x)$ $\lfloor x \rfloor$ 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

- __nv_log1pf(± 0) returns $-\infty$.
- __nv_log1pf(-1) returns +0.
- __nv_log1pf(x) returns NaN for x < -1.
- __nv_log1pf($+\infty$) returns $+\infty$.

3.237. __nv_log2

double **@__nv_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

- __nv_log2(± 0) returns $-\infty$.
- __nv_log2(1) returns +0.
- __nv_log2(x) returns NaN for x < 0.
- __nv_log2($+\infty$) returns $+\infty$.

3.238. __nv_log2f

float **@__nv_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.

Returns

- __nv_log2f(± 0) returns $-\infty$.
- __nv_log2f(1) returns +0.
- __nv_log2f(x) returns NaN for x < 0.
- __nv_log2f($+\infty$) returns $+\infty$.

3.239. __nv_logb

double **@__nv_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

- __nv_logb ± 0 returns $-\infty$
- __nv_logb $\pm\infty$ returns $+\infty$

3.240. __nv_logbf

float **@__nv_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

- __nv_logbf ± 0 returns $-\infty$
- __nv_logbf $\pm\infty$ returns $+\infty$

3.241. __nv_logf

float **@__nv_logf**(float %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, Single-Precision Floating-Point Functions section.

Returns

- __nv_logf(± 0) returns $-\infty$.
- __nv_logf(1) returns +0.
- __nv_logf(x) returns NaN for x < 0.
- __nv_logf($+\infty$) returns $+\infty$

3.242. __nv_longlong_as_double

double **@__nv_longlong_as_double**(i64 %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.

3.243. __nv_max

i32 **@__nv_max**(i32 %x, i32 %y)

Determine the maximum value of two 32-bit signed integers.

Determine the maximum value of the two 32-bit signed integers x and y.

Returns

Returns the maximum value of the two 32-bit signed integers x and y.

3.244. __nv_min

i32 **@__nv_min**(i32 %x, i32 %y)

Determine the minimum value of two 32-bit signed integers.

Determine the minimum value of the two 32-bit signed integers x and y.

Returns

Returns the minimum value of the two 32-bit signed integers x and y.

3.245. __nv_modf

double **@__nv_modf**(double %x, double* %b)

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

- __nv_modf($\pm x$, `iptr`) returns a result with the same sign as x.
- __nv_modf($\pm\infty$, `iptr`) returns ± 0 and stores $\pm\infty$ in the object pointed to by `iptr`.
- __nv_modf(NaN, `iptr`) stores a NaN in the object pointed to by `iptr` and returns a NaN.

3.246. __nv_modff

float **@__nv_modff**(float %x, float* %b)

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

- __nv_modff($\pm x$, `iptr`) returns a result with the same sign as x.
- __nv_modff($\pm\infty$, `iptr`) returns ± 0 and stores $\pm\infty$ in the object pointed to by `iptr`.
- __nv_modff(NaN, `iptr`) stores a NaN in the object pointed to by `iptr` and returns a NaN.

3.247. __nv_mul24

i32 **@__nv_mul24**(i32 %x, i32 %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.

3.248. __nv_mul64hi

i64 **@__nv_mul64hi**(i64 %x, i64 %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.

3.249. __nv_mulhi

i32 **@__nv_mulhi**(i32 %x, i32 %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.

3.250. __nv_nan

double **@__nv_nan**(i8* %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

- __nv_nan(`tagp`) returns NaN.

3.251. __nv_nanf

float **@__nv_nanf**(i8* %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

- __nv_nanf(`tagp`) returns NaN.

3.252. __nv_nearbyint

double **@__nv_nearbyint**(double %x)

Round the input argument to the nearest integer.

Round argument x to an integer value in double precision floating-point format.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- __nv_nearbyint(± 0) returns ± 0.
- __nv_nearbyint($\pm \infty$) returns $\pm \infty$.

3.253. __nv_nearbyintf

float **@__nv_nearbyintf**(float %x)

Round the input argument to the nearest integer.

Round argument x to an integer value in double precision floating-point format.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

- __nv_nearbyintf(± 0) returns ± 0.
- __nv_nearbyintf($\pm \infty$) returns $\pm \infty$.

3.254. __nv_nextafter

double **@__nv_nextafter**(double %x, double %y)

Return next representable double-precision floating-point value after argument.

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

- __nv_nextafter($\pm\infty$, y) returns $\pm\infty$.

3.255. __nv_nextafterf

float **@__nv_nextafterf**(float %x, float %y)

Return next representable double-precision floating-point value after argument.

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, Single-Precision Floating-Point Functions section.

Returns

- __nv_nextafterf($\pm\infty$, y) returns $\pm\infty$.

3.256. __nv_norm

double **@__nv_norm**(i32 %dim, double const* %a)

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 $\sqrt{\sum_{i=1}^{dim} p_i^2}$. If the correct value would overflow, returns $+\infty$. If the correct value would underflow, returns 0. If two of the input arguments is 0, returns remaining argument

3.257. __nv_norm3d

double **@__nv_norm3d**(double %a, double %b, double %c)

Calculate the square root of the sum of squares of three coordinates of argument.

Calculate the length of three dimensional vector p in euclidean space 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 3D vector $\sqrt{p.x^2 + p.y^2 + p.z^2}$. If the correct value would overflow, returns $+\infty$. If the correct value would underflow, returns 0.

3.258. __nv_norm3df

float **@__nv_norm3df**(float %a, float %b, float %c)

Calculate the square root of the sum of squares of three coordinates of argument.

Calculate the length of three dimensional vector p 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 $\sqrt{p.x^2 + p.y^2 + p.z^2}$. If the correct value would overflow, returns $+\infty$. If the correct value would underflow, returns 0. If two of the input arguments is 0, returns remaining argument

3.259. __nv_norm4d

double **@__nv_norm4d**(double %a, double %b, double %c, double %d)

Calculate the square root of the sum of squares of four coordinates of argument.

Calculate the length of four dimensional vector p in euclidean space 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 4D vector $\sqrt{p.x^2 + p.y^2 + p.z^2 + p.t^2}$. If the correct value would overflow, returns $+\infty$. If the correct value would underflow, returns 0.

3.260. __nv_norm4df

float **@__nv_norm4df**(float %a, float %b, float %c, float %d)

Calculate the square root of the sum of squares of four coordinates of argument.

Calculate the length of four dimensional vector p 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 $\sqrt{p.x^2 + p.y^2 + p.z^2 + p.t^2}$. If the correct value would overflow, returns $+\infty$. If the correct value would underflow, returns 0. If two of the input arguments is 0, returns remaining argument

3.261. __nv_normcdf

double **@__nv_normcdf**(double %x)

Calculate the standard normal cumulative distribution function.

Calculate the cumulative distribution function of the standard normal distribution for input argument y, $\Phi(y)$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- __nv_normcdf($+\infty$) returns 1
- __nv_normcdf($-\infty$) returns +0

3.262. __nv_normcdff

float **@__nv_normcdff**(float %x)

Calculate the standard normal cumulative distribution function.

Calculate the cumulative distribution function of the standard normal distribution for input argument y, $\Phi(y)$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

- __nv_normcdff($+\infty$) returns 1
- __nv_normcdff($-\infty$) returns +0

3.263. __nv_normcdfinv

double **@__nv_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 y, $\Phi^{-1}(y)$. 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

- __nv_normcdfinv(0) returns $-\infty$.
- __nv_normcdfinv(1) returns $+\infty$.
- __nv_normcdfinv(x) returns NaN if x is not in the interval [0,1].

3.264. __nv_normcdfinvf

float **@__nv_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 y, $\Phi^{-1}(y)$. 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

- __nv_normcdfinvf(0) returns $-\infty$.
- __nv_normcdfinvf(1) returns $+\infty$.
- __nv_normcdfinvf(x) returns NaN if x is not in the interval [0,1].

3.265. __nv_normf

float **@__nv_normf**(i32 %dim, float const* %a)

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, Single-Precision Floating-Point Functions section.

Returns

Returns the length of the dim-D vector $\sqrt{\sum_{i=1}^{dim} p_i^2}$. If the correct value would overflow, returns $+\infty$. If the correct value would underflow, returns 0. If two of the input arguments is 0, returns remaining argument

3.266. __nv_popc

i32 **@__nv_popc**(i32 %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.

3.267. __nv_popcll

i32 **@__nv_popcll**(i64 %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.

3.268. __nv_pow

double **@__nv_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

- __nv_pow(± 0 , y) returns $\pm\infty$ for y an integer less than 0.
- __nv_pow(± 0 , y) returns ± 0 for y an odd integer greater than 0.
- __nv_pow(± 0 , y) returns +0 for y > 0 and not and odd integer.
- __nv_pow(-1, $\pm\infty$) returns 1.
- __nv_pow(+1, y) returns 1 for any y, even a NaN.
- __nv_pow(x, ± 0) returns 1 for any x, even a NaN.
- __nv_pow(x, y) returns a NaN for finite x < 0 and finite non-integer y.
- __nv_pow(x, $-\infty$) returns $+\infty$ for $|x| < 1$.
- __nv_pow(x, $-\infty$) returns +0 for $|x| > 1$.
- __nv_pow(x, $+\infty$) returns +0 for $|x| < 1$.
- __nv_pow(x, $+\infty$) returns $+\infty$ for $|x| > 1$.
- __nv_pow($-\infty$, y) returns -0 for y an odd integer less than 0.
- __nv_pow($-\infty$, y) returns +0 for y < 0 and not an odd integer.
- __nv_pow($-\infty$, y) returns $-\infty$ for y an odd integer greater than 0.
- __nv_pow($-\infty$, y) returns $+\infty$ for y > 0 and not an odd integer.
- __nv_pow($+\infty$, y) returns +0 for y < 0.
- __nv_pow($+\infty$, y) returns $+\infty$ for y > 0.

3.269. __nv_powf

float **@__nv_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.

Returns

- __nv_powf(± 0 , y) returns $\pm\infty$ for y an integer less than 0.
- __nv_powf(± 0 , y) returns ± 0 for y an odd integer greater than 0.
- __nv_powf(± 0 , y) returns +0 for y > 0 and not and odd integer.
- __nv_powf(-1, $\pm\infty$) returns 1.
- __nv_powf(+1, y) returns 1 for any y, even a NaN.
- __nv_powf(x, ± 0) returns 1 for any x, even a NaN.
- __nv_powf(x, y) returns a NaN for finite x < 0 and finite non-integer y.
- __nv_powf(x, $-\infty$) returns $+\infty$ for $|x| < 1$.
- __nv_powf(x, $-\infty$) returns +0 for $|x| > 1$.
- __nv_powf(x, $+\infty$) returns +0 for $|x| < 1$.
- __nv_powf(x, $+\infty$) returns $+\infty$ for $|x| > 1$.
- __nv_powf($-\infty$, y) returns -0 for y an odd integer less than 0.
- __nv_powf($-\infty$, y) returns +0 for y < 0 and not an odd integer.
- __nv_powf($-\infty$, y) returns $-\infty$ for y an odd integer greater than 0.
- __nv_powf($-\infty$, y) returns $+\infty$ for y > 0 and not an odd integer.
- __nv_powf($+\infty$, y) returns +0 for y < 0.
- __nv_powf($+\infty$, y) returns $+\infty$ for y > 0.

3.270. __nv_powi

double **@__nv_powi** (double %x, i32 %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

- __nv_powi(± 0 , y) returns $\pm\infty$ for y an integer less than 0.
- __nv_powi(± 0 , y) returns ± 0 for y an odd integer greater than 0.
- __nv_powi(± 0 , y) returns +0 for y > 0 and not and odd integer.
- __nv_powi(-1, $\pm\infty$) returns 1.
- __nv_powi(+1, y) returns 1 for any y, even a NaN.
- __nv_powi(x, ± 0) returns 1 for any x, even a NaN.
- __nv_powi(x, y) returns a NaN for finite x < 0 and finite non-integer y.
- __nv_powi(x, $-\infty$) returns $+\infty$ for $|x| < 1$.
- __nv_powi(x, $-\infty$) returns +0 for $|x| > 1$.
- __nv_powi(x, $+\infty$) returns +0 for $|x| < 1$.
- __nv_powi(x, $+\infty$) returns $+\infty$ for $|x| > 1$.
- __nv_powi($-\infty$, y) returns -0 for y an odd integer less than 0.
- __nv_powi($-\infty$, y) returns +0 for y < 0 and not an odd integer.
- __nv_powi($-\infty$, y) returns $-\infty$ for y an odd integer greater than 0.
- __nv_powi($-\infty$, y) returns $+\infty$ for y > 0 and not an odd integer.
- __nv_powi($+\infty$, y) returns +0 for y < 0.
- __nv_powi($+\infty$, y) returns $+\infty$ for y > 0.

3.271. __nv_powif

float **@__nv_powif**(float %x, i32 %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.

Returns

- __nv_powif(± 0 , y) returns $\pm\infty$ for y an integer less than 0.
- __nv_powif(± 0 , y) returns ± 0 for y an odd integer greater than 0.
- __nv_powif(± 0 , y) returns +0 for y > 0 and not and odd integer.
- __nv_powif(-1, $\pm\infty$) returns 1.
- __nv_powif(+1, y) returns 1 for any y, even a NaN.
- __nv_powif(x, ± 0) returns 1 for any x, even a NaN.

- __nv_powif(x, y) returns a NaN for finite x < 0 and finite non-integer y.
- __nv_powif(x, $-\infty$) returns $+\infty$ for $|x| < 1$.
- __nv_powif(x, $-\infty$) returns +0 for $|x| > 1$.
- __nv_powif(x, $+\infty$) returns +0 for $|x| < 1$.
- __nv_powif(x, $+\infty$) returns $+\infty$ for $|x| > 1$.
- __nv_powif($-\infty$, y) returns -0 for y an odd integer less than 0.
- __nv_powif($-\infty$, y) returns +0 for y < 0 and not an odd integer.
- __nv_powif($-\infty$, y) returns $-\infty$ for y an odd integer greater than 0.
- __nv_powif($-\infty$, y) returns $+\infty$ for y > 0 and not an odd integer.
- __nv_powif($+\infty$, y) returns +0 for y < 0.
- __nv_powif($+\infty$, y) returns $+\infty$ for y > 0.

3.272. __nv_rcbrt

double **@__nv_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

- __nv_rcbrt(± 0) returns $\pm\infty$.
- __nv_rcbrt($\pm\infty$) returns ± 0.

3.273. __nv_rcbrtf

float **@__nv_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

- __nv_rcbrtf(± 0) returns $\pm \infty$.
- __nv_rcbrtf($\pm \infty$) returns ± 0.

3.274. __nv_rcp64h

double **@__nv_rcp64h**(double %d)

DOCUMENTATION MISSING.

3.275. __nv_remainder

double **@__nv_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

- __nv_remainder(x, 0) returns NaN.
- __nv_remainder($\pm \infty$, y) returns NaN.
- __nv_remainder(x, $\pm \infty$) returns x for finite x.

3.276. __nv_remainderf

float **@__nv_remainderf**(float %x, float %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, Single-Precision Floating-Point Functions section.

Returns

- __nv_remainderf(x, 0) returns NaN.
- __nv_remainderf($\pm\infty$, y) returns NaN.
- __nv_remainderf(x, $\pm\infty$) returns x for finite x.

3.277. __nv_remquo

double **@__nv_remquo**(double %x, double %y, i32* %c)

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.

- __nv_remquo(x, 0, quo) returns NaN.
- __nv_remquo($\pm\infty$, y, quo) returns NaN.
- __nv_remquo(x, $\pm\infty$, quo) returns x.

3.278. __nv_remquof

float **@__nv_remquof**(float %x, float %y, i32* %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, Single-Precision Floating-Point Functions section.

Returns

Returns the remainder.

- __nv_remquof(x, 0, quo) returns NaN.
- __nv_remquof($\pm\infty$, y, quo) returns NaN.
- __nv_remquof(x, $\pm\infty$, quo) returns x.

3.279. __nv_rhadd

i32 **@__nv_rhadd**(i32 %x, i32 %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.

3.280. __nv_rhypot

double **@__nv_rhypot**(double %x, double %y)

Calculate the reciprocal square root of the sum of squares of two arguments.

Calculate reciprocal of 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 $\sqrt{x^2+y^2}$. If the correct value would overflow, returns $+\infty$. If the correct value would underflow, returns 0.

3.281. __nv_rhypotf

float **@__nv_rhypotf**(float %x, float %y)

Calculate the reciprocal square root of the sum of squares of two arguments.

Calculate the reciprocal 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 inverted length of the hypotenuse $\frac{1}{\sqrt{x^2+y^2}}$. If the correct value would overflow, returns $+\infty$ If the correct value would underflow, returns 0.

3.282. __nv_rint

double **@__nv_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.

3.283. __nv_rintf

float **@__nv_rintf**(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 to the nearest even integer value.

Returns

Returns rounded integer value.

3.284. __nv_rnorm

double **@__nv_rnorm**(i32 %dim, double const* %a)

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 agument, 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 $\frac{1}{\sqrt{\sum_{i=1}^{dim} p_i^2}}$. If the square root would overflow, returns 0. If the square root would underflow, returns $+\infty$.

3.285. __nv_rnorm3d

double **@__nv_rnorm3d**(double %a, double %b, double %c)

Calculate the reciprocal square root of the sum of squares of three coordinates of argument.

Calculate reciprocal of the length of three dimensional vector p in euclidean space 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 inverted length of the 3D vector $\sqrt{p.x^2 + p.y^2 + p.z^2}$. If the correct value would overflow, returns $+\infty$. If the correct value would underflow, returns 0. If two of the input arguments is 0, returns remaining argument

3.286. __nv_rnorm3df

float **@__nv_rnorm3df**(float %a, float %b, float %c)

Calculate the reciprocal square root of the sum of squares of three coordinates of argument.

Calculate the reciprocal length of three dimensional vector p in euclidean space 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 inverted length of the 3D vector $\frac{1}{\sqrt{p.x^2+p.y^2+p.z^2}}$. If the correct value would overflow, returns $+\infty$. If the correct value would underflow, returns 0.

3.287. __nv_rnorm4d

double **@__nv_rnorm4d**(double %a, double %b, double %c, double %d)

Calculate the reciprocal square root of the sum of squares of four coordinates of argument.

Calculate reciprocal of the length of four dimensional vector p in euclidean space 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 inverted length of the 3D vector $\sqrt{p.x^2 + p.y^2 + p.z^2 + p.t^2}$. If the correct value would overflow, returns $+\infty$. If the correct value would underflow, returns 0. If two of the input arguments is 0, returns remaining argument

3.288. __nv_rnorm4df

float **@__nv_rnorm4df**(float %a, float %b, float %c, float %d)

Calculate the reciprocal square root of the sum of squares of four coordinates of argument.

Calculate the reciprocal length of four dimensional vector p in euclidean space 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 inverted length of the 3D vector $\frac{1}{\sqrt{p.x^2+p.y^2+p.z^2+o.t^2}}$. If the correct value would overflow, returns $+\infty$. If the correct value would underflow, returns 0.

3.289. __nv_rnormf

float **@__nv_rnormf**(i32 %dim, float const* %a)

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 agument, 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 $\frac{1}{\sqrt{\sum_{i=1}^{dim} p_i^2}}$. If the square root would overflow, returns 0. If the square root would underflow, returns $+\infty$.

3.290. __nv_round

double **@__nv_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.

3.291. __nv_roundf

float **@__nv_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 ::rint().

Returns

Returns rounded integer value.

3.292. __nv_rsqrt

double **@__nv_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 $1/\sqrt{x}$.

- __nv_rsqrt($+\infty$) returns +0.
- __nv_rsqrt(± 0) returns $\pm\infty$.
- __nv_rsqrt(x) returns NaN if x is less than 0.

3.293. __nv_rsqrtf

float **@__nv_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, Double-Precision Floating-Point Functions section.

Returns

Returns $1/\sqrt{x}$.

- __nv_rsqrtf($+\infty$) returns +0.
- __nv_rsqrtf(± 0) returns $\pm\infty$.
- __nv_rsqrtf(x) returns NaN if x is less than 0.

3.294. __nv_sad

i32 **@__nv_sad**(i32 %x, i32 %y, i32 %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 $|x - y| + z$.

3.295. __nv_saturatef

float **@__nv_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

- __nv_saturatef(x) returns 0 if x < 0.
- __nv_saturatef(x) returns 1 if x > 1.
- __nv_saturatef(x) returns x if $0 \leq x \leq 1$.
- __nv_saturatef(NaN) returns 0.

3.296. __nv_scalbn

double **@__nv_scalbn**(double %x, i32 %y)

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.

- __nv_scalbn(± 0 , n) returns ± 0.
- __nv_scalbn(x, 0) returns x.
- __nv_scalbn($\pm \infty$, n) returns $\pm \infty$.

3.297. __nv_scalbnf

float **@__nv_scalbnf**(float %x, i32 %y)

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.

- __nv_scalbnf(± 0 , n) returns ± 0.
- __nv_scalbnf(x, 0) returns x.
- __nv_scalbnf($\pm \infty$, n) returns $\pm \infty$.

3.298. __nv_signbitd

i32 **@__nv_signbitd**(double %x)

Return the sign bit of the input.

Determine whether the floating-point value x is negative.

Returns

Returns a nonzero value if and only if x is negative. Reports the sign bit of all values including infinities, zeros, and NaNs.

3.299. __nv_signbitf

i32 **@__nv_signbitf**(float %x)

Return the sign bit of the input.

Determine whether the floating-point value x is negative.

Returns

Returns a nonzero value if and only if x is negative. Reports the sign bit of all values including infinities, zeros, and NaNs.

3.300. __nv_sin

double **@__nv_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

- __nv_sin(± 0) returns ± 0.
- __nv_sin($\pm \infty$) returns NaN.

3.301. __nv_sincos

void **@__nv_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, `zptr`.

See __nv_sin() and __nv_cos().

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- none

3.302. __nv_sincosf

void **@__nv_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, `zptr`.

See __nv_sinf() and __nv_cosf().

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

- none

3.303. __nv_sincospi

void **@__nv_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, `zptr`.

See __nv_sinpi() and __nv_cospi().

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- none

3.304. __nv_sincospif

void **@__nv_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, `zptr`.

See __nv_sinpif() and __nv_cospif().

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

- none

3.305. __nv_sinf

float **@__nv_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.

Returns

- __nv_sinf(± 0) returns ± 0.
- __nv_sinf($\pm\infty$) returns NaN.

3.306. __nv_sinh

double **@__nv_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

- __nv_sinh(± 0) returns ± 0.

3.307. __nv_sinhf

float **@__nv_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

- __nv_sinhf(± 0) returns ± 0.

3.308. __nv_sinpi

double **@__nv_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

- __nv_sinpi(± 0) returns ± 0.
- __nv_sinpi($\pm \infty$) returns NaN.

3.309. __nv_sinpif

float **@__nv_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

- __nv_sinpif(± 0) returns ± 0.
- __nv_sinpif($\pm\infty$) returns NaN.

3.310. __nv_sqrt

double **@__nv_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 $\sqrt{x}$.

- __nv_sqrt(± 0) returns ± 0.
- __nv_sqrt($+\infty$) returns $+\infty$.
- __nv_sqrt(x) returns NaN if x is less than 0.

3.311. __nv_sqrtf

float **@__nv_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, Double-Precision Floating-Point Functions section.

Returns

Returns $\sqrt{x}$.

- __nv_sqrtf(± 0) returns ± 0.
- __nv_sqrtf($+\infty$) returns $+\infty$.
- __nv_sqrtf(x) returns NaN if x is less than 0.

3.312. __nv_tan

double **@__nv_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

- __nv_tan(± 0) returns ± 0.
- __nv_tan($\pm \infty$) returns NaN.

3.313. __nv_tanf

float **@__nv_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.

Returns

- __nv_tanf(± 0) returns ± 0.
- __nv_tanf($\pm \infty$) returns NaN.

3.314. __nv_tanh

double **@__nv_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

- __nv_tanh(± 0) returns ± 0.

3.315. __nv_tanhf

float **@__nv_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

- __nv_tanhf(± 0) returns ± 0.

3.316. __nv_tgamma

double **@__nv_tgamma**(double %x)

Calculate the gamma function of the input argument.

Calculate the gamma function of the input argument x, namely the value of $\int_0^\infty e^{-t} t^{x-1} dt$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.

Returns

- __nv_tgamma(± 0) returns $\pm \infty$.

- __nv_tgamma(2) returns +0.
- __nv_tgamma(x) returns $\pm\infty$ if the correctly calculated value is outside the double floating point range.
- __nv_tgamma(x) returns NaN if x < 0.
- __nv_tgamma($-\infty$) returns NaN.
- __nv_tgamma($+\infty$) returns $+\infty$.

3.317. __nv_tgammaf

float **@__nv_tgammaf**(float %x)

Calculate the gamma function of the input argument.

Calculate the gamma function of the input argument x, namely the value of $\int_0^\infty e^{-t}t^{x-1}dt$.

Note: For accuracy information, see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.

Returns

- __nv_tgammaf(± 0) returns $\pm\infty$.
- __nv_tgammaf(2) returns +0.
- __nv_tgammaf(x) returns $\pm\infty$ if the correctly calculated value is outside the double floating point range.
- __nv_tgammaf(x) returns NaN if x < 0.
- __nv_tgammaf($-\infty$) returns NaN.
- __nv_tgammaf($+\infty$) returns $+\infty$.

3.318. __nv_trunc

double **@__nv_trunc**(double %x)

Truncate input argument to the integral part.

Round x to the nearest integer value that does not exceed x in magnitude.

Returns

Returns truncated integer value.

3.319. __nv_truncf

float **@__nv_truncf**(float %x)

Truncate input argument to the integral part.

Round x to the nearest integer value that does not exceed x in magnitude.

Returns

Returns truncated integer value.

3.320. __nv_uhadd

i32 **@__nv_uhadd**(i32 %x, i32 %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.

3.321. __nv_uint2double_rn

double **@__nv_uint2double_rn**(i32 %i)

Convert an unsigned int to a double.

Convert the unsigned integer value x to a double-precision floating point value.

Returns

Returns converted value.

3.322. __nv_uint2float_rd

float **@__nv_uint2float_rd**(i32 %in)

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.

3.323. __nv_uint2float_rn

float **@__nv_uint2float_rn**(i32 %in)

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.

3.324. __nv_uint2float_ru

float **@__nv_uint2float_ru**(i32 %in)

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.

3.325. __nv_uint2float_rz

float **@__nv_uint2float_rz**(i32 %in)

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.

3.326. __nv_uint_as_float

float **@__nv_uint_as_float**(i32 %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.

3.327. __nv_ull2double_rd

double **@__nv_ull2double_rd**(i64 %l)

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.

3.328. __nv_ull2double_rn

double **@__nv_ull2double_rn**(i64 %l)

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.

3.329. __nv_ull2double_ru

double **@__nv_ull2double_ru**(i64 %l)

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.

3.330. __nv_ull2double_rz

double **@__nv_ull2double_rz**(i64 %l)

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.

3.331. __nv_ull2float_rd

float **@__nv_ull2float_rd**(i64 %l)

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.

3.332. __nv_ull2float_rn

float **@__nv_ull2float_rn**(i64 %l)

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.

3.333. __nv_ull2float_ru

float **@__nv_ull2float_ru**(i64 %l)

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.

3.334. __nv_ull2float_rz

float **@__nv_ull2float_rz**(i64 %l)

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.

3.335. __nv_ullmax

i64 **@__nv_ullmax**(i64 %x, i64 %y)

Determine the maximum value of two 64-bit unsigned integers.

Determine the maximum value of the two 64-bit unsigned integers x and y.

Returns

Returns the maximum value of the two 64-bit unsigned integers x and y.

3.336. __nv_ullmin

i64 **@__nv_ullmin**(i64 %x, i64 %y)

Determine the minimum value of two 64-bit unsigned integers.

Determine the minimum value of the two 64-bit unsigned integers x and y.

Returns

Returns the minimum value of the two 64-bit unsigned integers x and y.

3.337. __nv_umax

i32 **@__nv_umax**(i32 %x, i32 %y)

Determine the maximum value of two 32-bit unsigned integers.

Determine the maximum value of the two 32-bit unsigned integers x and y.

Returns

Returns the maximum value of the two 32-bit unsigned integers x and y.

3.338. __nv_umin

i32 **@__nv_umin**(i32 %x, i32 %y)

Determine the minimum value of two 32-bit unsigned integers.

Determine the minimum value of the two 32-bit unsigned integers x and y.

Returns

Returns the minimum value of the two 32-bit unsigned integers x and y.

3.339. __nv_umul24

i32 **@__nv_umul24**(i32 %x, i32 %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.

3.340. __nv_umul64hi

i64 **@__nv_umul64hi**(i64 %x, i64 %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.

3.341. __nv_umulhi

i32 **@__nv_umulhi**(i32 %x, i32 %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.

3.342. __nv_urhadd

i32 **@__nv_urhadd**(i32 %x, i32 %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.

3.343. __nv_usad

i32 **@__nv_usad**(i32 %x, i32 %y, i32 %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 $|x - y| + z$.

3.344. __nv_vabs2

i32 **@__nv_vabs2**(i32 %a)

3.345. __nv_vabs4

i32 **@__nv_vabs4**(i32 %a)

3.346. __nv_vabsdiffs2

i32 **@__nv_vabsdiffs2**(i32 %a, i32 %b)

3.347. __nv_vabsdiffs4

i32 **@__nv_vabsdiffs4**(i32 %a, i32 %b)

3.348. __nv_vabsdiffu2

i32 **@__nv_vabsdiffu2**(i32 %a, i32 %b)

3.349. __nv_vabsdiffu4

i32 **@__nv_vabsdiffu4**(i32 %a, i32 %b)

3.350. __nv_vabsss2

i32 **@__nv_vabsss2**(i32 %a)

3.351. __nv_vabsss4

i32 **@__nv_vabsss4**(i32 %a)

3.352. __nv_vadd2

i32 **@__nv_vadd2**(i32 %a, i32 %b)

3.353. __nv_vadd4

i32 **@__nv_vadd4**(i32 %a, i32 %b)

3.354. __nv_vaddss2

i32 **@__nv_vaddss2**(i32 %a, i32 %b)

3.355. __nv_vaddss4

i32 **@__nv_vaddss4**(i32 %a, i32 %b)

3.356. __nv_vaddus2

i32 **@__nv_vaddus2**(i32 %a, i32 %b)

3.357. __nv_vaddus4

i32 **@__nv_vaddus4**(i32 %a, i32 %b)

3.358. __nv_vavgs2

i32 **@__nv_vavgs2**(i32 %a, i32 %b)

3.359. __nv_vavgs4

i32 **@__nv_vavgs4**(i32 %a, i32 %b)

3.360. __nv_vavgu2

i32 **@__nv_vavgu2**(i32 %a, i32 %b)

3.361. __nv_vavgu4

i32 **@__nv_vavgu4**(i32 %a, i32 %b)

3.362. __nv_vcmpeq2

i32 **@__nv_vcmpeq2**(i32 %a, i32 %b)

3.363. __nv_vcmpeq4

i32 **@__nv_vcmpeq4**(i32 %a, i32 %b)

3.364. __nv_vcmpges2

i32 **@__nv_vcmpges2**(i32 %a, i32 %b)

3.365. __nv_vcmpges4

i32 **@__nv_vcmpges4**(i32 %a, i32 %b)

3.366. __nv_vcmpgeu2

i32 **@__nv_vcmpgeu2**(i32 %a, i32 %b)

3.367. __nv_vcmpgeu4

i32 **@__nv_vcmpgeu4**(i32 %a, i32 %b)

3.368. __nv_vcmpgts2

i32 **@__nv_vcmpgts2**(i32 %a, i32 %b)

3.369. __nv_vcmpgts4

i32 **@__nv_vcmpgts4**(i32 %a, i32 %b)

3.370. __nv_vcmpgtu2

i32 **@__nv_vcmpgtu2**(i32 %a, i32 %b)

3.371. __nv_vcmpgtu4

i32 **@__nv_vcmpgtu4**(i32 %a, i32 %b)

3.372. __nv_vcmples2

i32 **@__nv_vcmples2**(i32 %a, i32 %b)

3.373. __nv_vcmples4

i32 **@__nv_vcmples4**(i32 %a, i32 %b)

3.374. __nv_vcmpleu2

i32 **@__nv_vcmpleu2**(i32 %a, i32 %b)

3.375. __nv_vcmpleu4

i32 **@__nv_vcmpleu4**(i32 %a, i32 %b)

3.376. __nv_vcmplts2

i32 **@__nv_vcmplts2**(i32 %a, i32 %b)

3.377. __nv_vcmplts4

i32 **@__nv_vcmplts4**(i32 %a, i32 %b)

3.378. __nv_vcmpltu2

i32 **@__nv_vcmpltu2**(i32 %a, i32 %b)

3.379. __nv_vcmpltu4

i32 **@__nv_vcmpltu4**(i32 %a, i32 %b)

3.380. __nv_vcmpne2

i32 **@__nv_vcmpne2**(i32 %a, i32 %b)

3.381. __nv_vcmpne4

i32 **@__nv_vcmpne4**(i32 %a, i32 %b)

3.382. __nv_vhaddu2

i32 **@__nv_vhaddu2**(i32 %a, i32 %b)

3.383. __nv_vhaddu4

i32 **@__nv_vhaddu4**(i32 %a, i32 %b)

3.384. __nv_vmaxs2

i32 **@__nv_vmaxs2**(i32 %a, i32 %b)

3.385. __nv_vmaxs4

i32 **@__nv_vmaxs4**(i32 %a, i32 %b)

3.386. __nv_vmaxu2

i32 **@__nv_vmaxu2**(i32 %a, i32 %b)

3.387. __nv_vmaxu4

i32 **@__nv_vmaxu4**(i32 %a, i32 %b)

3.388. __nv_vmins2

i32 **@__nv_vmins2**(i32 %a, i32 %b)

3.389. __nv_vmins4

i32 **@__nv_vmins4**(i32 %a, i32 %b)

3.390. __nv_vminu2

i32 **@__nv_vminu2**(i32 %a, i32 %b)

3.391. __nv_vminu4

i32 **@__nv_vminu4**(i32 %a, i32 %b)

3.392. __nv_vneg2

i32 **@__nv_vneg2**(i32 %a)

3.393. __nv_vneg4

i32 **@__nv_vneg4**(i32 %a)

3.394. __nv_vnegss2

i32 **@__nv_vnegss2**(i32 %a)

3.395. __nv_vnegss4

i32 **@__nv_vnegss4**(i32 %a)

3.396. __nv_vsads2

i32 **@__nv_vsads2**(i32 %a, i32 %b)

3.397. __nv_vsads4

i32 **@__nv_vsads4**(i32 %a, i32 %b)

3.398. __nv_vsadu2

i32 **@__nv_vsadu2**(i32 %a, i32 %b)

3.399. __nv_vsadu4

i32 **@__nv_vsadu4**(i32 %a, i32 %b)

3.400. __nv_vseteq2

i32 **@__nv_vseteq2**(i32 %a, i32 %b)

3.401. __nv_vseteq4

i32 **@__nv_vseteq4**(i32 %a, i32 %b)

3.402. __nv_vsetges2

i32 **@__nv_vsetges2**(i32 %a, i32 %b)

3.403. __nv_vsetges4

i32 **@__nv_vsetges4**(i32 %a, i32 %b)

3.404. __nv_vsetgeu2

i32 **@__nv_vsetgeu2**(i32 %a, i32 %b)

3.405. __nv_vsetgeu4

i32 **@__nv_vsetgeu4**(i32 %a, i32 %b)

3.406. __nv_vsetgts2

i32 **@__nv_vsetgts2**(i32 %a, i32 %b)

3.407. __nv_vsetgts4

i32 **@__nv_vsetgts4**(i32 %a, i32 %b)

3.408. __nv_vsetgtu2

i32 **@__nv_vsetgtu2**(i32 %a, i32 %b)

3.409. __nv_vsetgtu4

i32 **@__nv_vsetgtu4**(i32 %a, i32 %b)

3.410. __nv_vsetles2

i32 **@__nv_vsetles2**(i32 %a, i32 %b)

3.411. __nv_vsetles4

i32 **@__nv_vsetles4**(i32 %a, i32 %b)

3.412. __nv_vsetleu2

i32 **@__nv_vsetleu2**(i32 %a, i32 %b)

3.413. __nv_vsetleu4

i32 **@__nv_vsetleu4**(i32 %a, i32 %b)

3.414. __nv_vsetlts2

i32 **@__nv_vsetlts2**(i32 %a, i32 %b)

3.415. __nv_vsetlts4

i32 **@__nv_vsetlts4**(i32 %a, i32 %b)

3.416. __nv_vsetltu2

i32 **@__nv_vsetltu2**(i32 %a, i32 %b)

3.417. __nv_vsetltu4

i32 **@__nv_vsetltu4**(i32 %a, i32 %b)

3.418. __nv_vsetne2

i32 **@__nv_vsetne2**(i32 %a, i32 %b)

3.419. __nv_vsetne4

i32 **@__nv_vsetne4**(i32 %a, i32 %b)

3.420. __nv_vsub2

i32 **@__nv_vsub2**(i32 %a, i32 %b)

3.421. __nv_vsub4

i32 **@__nv_vsub4**(i32 %a, i32 %b)

3.422. __nv_vsubss2

i32 **@__nv_vsubss2**(i32 %a, i32 %b)

3.423. __nv_vsubss4

i32 **@__nv_vsubss4**(i32 %a, i32 %b)

3.424. __nv_vsubus2

i32 **@__nv_vsubus2**(i32 %a, i32 %b)

3.425. __nv_vsubus4

i32 **@__nv_vsubus4**(i32 %a, i32 %b)

3.426. __nv_y0

double **@__nv_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.

- __nv_y0(0) returns $-\infty$.
- __nv_y0(x) returns NaN for x < 0.
- __nv_y0($+\infty$) returns +0.
- __nv_y0(NaN) returns NaN.

3.427. __nv_y0f

float **@__nv_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.

- __nv_y0f(0) returns $-\infty$.
- __nv_y0f(x) returns NaN for x < 0.
- __nv_y0f($+\infty$) returns +0.
- __nv_y0f(NaN) returns NaN.

3.428. __nv_y1

double **@__nv_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.

- __nv_y1(0) returns $-\infty$.
- __nv_y1(x) returns NaN for x < 0.
- __nv_y1($+\infty$) returns +0.
- __nv_y1(NaN) returns NaN.

3.429. __nv_y1f

float **@__nv_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.

- __nv_y1f(0) returns $-\infty$.
- __nv_y1f(x) returns NaN for x < 0.
- __nv_y1f($+\infty$) returns +0.
- __nv_y1f(NaN) returns NaN.

3.430. __nv_yn

double **@__nv_yn**(i32 %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.

- __nv_yn(n, x) returns NaN for n < 0.
- __nv_yn(n, 0) returns $-\infty$.
- __nv_yn(n, x) returns NaN for x < 0.
- __nv_yn(n, $+\infty$) returns +0.
- __nv_yn(n, NaN) returns NaN.

3.431. __nv_ynf

float **@__nv_ynf**(i32 %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.

- __nv_ynf(n, x) returns NaN for n < 0.
- __nv_ynf(n, 0) returns $-\infty$.
- __nv_ynf(n, x) returns NaN for x < 0.
- __nv_ynf(n, $+\infty$) returns +0.
- __nv_ynf(n, NaN) returns NaN.

int *__nv_abs*(int x)

Determine the absolute value of a 32-bit signed integer.

double *__nv_acos*(double x)

Calculate the arc cosine of the input argument.

float ***__nv_acosf*****(float x)**
Calculate the arc cosine of the input argument.

double ***__nv_acosh*****(double x)**
Calculate the nonnegative arc hyperbolic cosine of the input argument.

float ***__nv_acoshf*****(float x)**
Calculate the nonnegative arc hyperbolic cosine of the input argument.

double ***__nv_asin*****(double x)**
Calculate the arc sine of the input argument.

float ***__nv_asinf*****(float x)**
Calculate the arc sine of the input argument.

double ***__nv_asinh*****(double x)**
Calculate the arc hyperbolic sine of the input argument.

float ***__nv_asinhf*****(float x)**
Calculate the arc hyperbolic sine of the input argument.

double ***__nv_atan*****(double x)**
Calculate the arc tangent of the input argument.

double ***__nv_atan2*****(double x, double y)**
Calculate the arc tangent of the ratio of first and second input arguments.

float ***__nv_atan2f*****(float x, float y)**
Calculate the arc tangent of the ratio of first and second input arguments.

float ***__nv_atanf*****(float x)**
Calculate the arc tangent of the input argument.

double ***__nv_atanh*****(double x)**
Calculate the arc hyperbolic tangent of the input argument.

float ***__nv_atanhf*****(float x)**
Calculate the arc hyperbolic tangent of the input argument.

unsigned int ***__nv_brev*****(unsigned int x)**
Reverse the bit order of a 32 bit unsigned integer.

unsigned long long ***__nv_brevll*****(unsigned long long x)**
Reverse the bit order of a 64 bit unsigned integer.

unsigned int ***__nv_byte_perm*****(unsigned int x, unsigned int y, unsigned int z)**
Return selected bytes from two 32 bit unsigned integers.

double ***__nv_cbrt*****(double x)**
Calculate the cube root of the input argument.

float ***__nv_cbrtf*****(float x)**
Calculate the cube root of the input argument.

double ***__nv_ceil*****(double x)**
Calculate ceiling of the input argument.

float ***__nv_ceilf*****(float x)**
Calculate ceiling of the input argument.

int ***__nv_clz*****(int x)**
Return the number of consecutive high-order zero bits in a 32 bit integer.

int ***__nv_clzll*****(long long x)**
Count the number of consecutive high-order zero bits in a 64 bit integer.

double ***__nv_copysign*****(double x, double y)**
Create value with given magnitude, copying sign of second value.

float ***__nv_copysignf*****(float x, float y)**
Create value with given magnitude, copying sign of second value.

double ***__nv_cos*****(double x)**
Calculate the cosine of the input argument.

float ***__nv_cosf*****(float x)**
Calculate the cosine of the input argument.

double ***__nv_cosh*****(double x)**
Calculate the hyperbolic cosine of the input argument.

float ***__nv_coshf*****(float x)**
Calculate the hyperbolic cosine of the input argument.

double ***__nv_cospi*****(double x)**
Calculate the cosine of the input argument $\times\pi$.

float ***__nv_cospif*****(float x)**
Calculate the cosine of the input argument $\times\pi$.

double ***__nv_cyl_bessel_i0*****(double x)**
Calculate the value of the regular modified cylindrical Bessel function of order 0 for the input argument.

float ***__nv_cyl_bessel_i0f*****(float x)**
Calculate the value of the regular modified cylindrical Bessel function of order 0 for the input argument.

double ***__nv_cyl_bessel_i1*****(double x)**
Calculate the value of the regular modified cylindrical Bessel function of order 1 for the input argument.

float ***__nv_cyl_bessel_i1f*****(float x)**
Calculate the value of the regular modified cylindrical Bessel function of order 1 for the input argument.

double ***__nv_dadd_rd*****(double x, double y)**
Add two floating point values in round-down mode.

double ***__nv_dadd_rn*****(double x, double y)**
Add two floating point values in round-to-nearest-even mode.

double ***__nv_dadd_ru*****(double x, double y)**
Add two floating point values in round-up mode.

double ***__nv_dadd_rz*****(double x, double y)**
Add two floating point values in round-towards-zero mode.

double ***__nv_ddiv_rd*****(double x, double y)**
Divide two floating point values in round-down mode.

double ***__nv_ddiv_rn*****(double x, double y)**
Divide two floating point values in round-to-nearest-even mode.

double ***__nv_ddiv_ru*****(double x, double y)**
Divide two floating point values in round-up mode.

double ***__nv_ddiv_rz*****(double x, double y)**
Divide two floating point values in round-towards-zero mode.

double ***__nv_dmul_rd*****(double x, double y)**
Multiply two floating point values in round-down mode.

double ***__nv_dmul_rn*****(double x, double y)**
Multiply two floating point values in round-to-nearest-even mode.

double ***__nv_dmul_ru*****(double x, double y)**
Multiply two floating point values in round-up mode.

double ***__nv_dmul_rz*****(double x, double y)**
Multiply two floating point values in round-towards-zero mode.

float ***__nv_double2float_rd*****(double d)**
Convert a double to a float in round-down mode.

float ***__nv_double2float_rn*****(double d)**
Convert a double to a float in round-to-nearest-even mode.

float ***__nv_double2float_ru*****(double d)**
Convert a double to a float in round-up mode.

float ***__nv_double2float_rz*****(double d)**
Convert a double to a float in round-towards-zero mode.

int ***__nv_double2hiint*****(double d)**
Reinterpret high 32 bits in a double as a signed integer.

int ***__nv_double2int_rd*****(double d)**
Convert a double to a signed int in round-down mode.

int ***__nv_double2int_rn*****(double d)**
Convert a double to a signed int in round-to-nearest-even mode.

int ***__nv_double2int_ru*****(double d)**
Convert a double to a signed int in round-up mode.

int ***__nv_double2int_rz*****(double d)**
Convert a double to a signed int in round-towards-zero mode.

long long ***__nv_double2ll_rd*****(double f)**
Convert a double to a signed 64-bit int in round-down mode.

long long ***__nv_double2ll_rn*****(double f)**
Convert a double to a signed 64-bit int in round-to-nearest-even mode.

long long ***__nv_double2ll_ru*****(double f)**
Convert a double to a signed 64-bit int in round-up mode.

long long ***__nv_double2ll_rz*****(double f)**
Convert a double to a signed 64-bit int in round-towards-zero mode.

int ***__nv_double2loint*****(double d)**
Reinterpret low 32 bits in a double as a signed integer.

unsigned int ***__nv_double2uint_rd*****(double d)**
Convert a double to an unsigned int in round-down mode.

unsigned int ***__nv_double2uint_rn*****(double d)**
Convert a double to an unsigned int in round-to-nearest-even mode.

unsigned int __nv_double2uint_ru(double d)
Convert a double to an unsigned int in round-up mode.

unsigned int __nv_double2uint_rz(double d)
Convert a double to an unsigned int in round-towards-zero mode.

unsigned long long __nv_double2ull_rd(double f)
Convert a double to an unsigned 64-bit int in round-down mode.

unsigned long long __nv_double2ull_rn(double f)
Convert a double to an unsigned 64-bit int in round-to-nearest-even mode.

unsigned long long __nv_double2ull_ru(double f)
Convert a double to an unsigned 64-bit int in round-up mode.

unsigned long long __nv_double2ull_rz(double f)
Convert a double to an unsigned 64-bit int in round-towards-zero mode.

long long __nv_double_as_longlong(double x)
Reinterpret bits in a double as a 64-bit signed integer.

double __nv_drcp_rd(double x)
Compute $\frac{1}{x}$ in round-down mode.

double __nv_drcp_rn(double x)
Compute $\frac{1}{x}$ in round-to-nearest-even mode.

double __nv_drcp_ru(double x)
Compute $\frac{1}{x}$ in round-up mode.

double __nv_drcp_rz(double x)
Compute $\frac{1}{x}$ in round-towards-zero mode.

double __nv_dsqrt_rd(double x)
Compute $\sqrt{x}$ in round-down mode.

double __nv_dsqrt_rn(double x)
Compute $\sqrt{x}$ in round-to-nearest-even mode.

double __nv_dsqrt_ru(double x)
Compute $\sqrt{x}$ in round-up mode.

double __nv_dsqrt_rz(double x)
Compute $\sqrt{x}$ in round-towards-zero mode.

double __nv_dsub_rd(double a, double b)

double __nv_dsub_rn(double a, double b)

double __nv_dsub_ru(double a, double b)

double __nv_dsub_rz(double a, double b)

double __nv_erf(double x)
Calculate the error function of the input argument.

double __nv_erfc(double x)
Calculate the complementary error function of the input argument.

float __nv_erfcf(float x)
Calculate the complementary error function of the input argument.

double __nv_erfcinv(double x)
Calculate the inverse complementary error function of the input argument.

float __nv_erfcinvf(float x)
Calculate the inverse complementary error function of the input argument.

double __nv_erfcx(double x)
Calculate the scaled complementary error function of the input argument.

float __nv_erfcxf(float x)
Calculate the scaled complementary error function of the input argument.

float __nv_erff(float x)
Calculate the error function of the input argument.

double __nv_erfinv(double x)
Calculate the inverse error function of the input argument.

float __nv_erfinvf(float x)
Calculate the inverse error function of the input argument.

double __nv_exp(double x)
Calculate the base e exponential of the input argument.

double __nv_exp10(double x)
Calculate the base 10 exponential of the input argument.

float __nv_exp10f(float x)
Calculate the base 10 exponential of the input argument.

double __nv_exp2(double x)
Calculate the base 2 exponential of the input argument.

float __nv_exp2f(float x)
Calculate the base 2 exponential of the input argument.

float __nv_expf(float x)
Calculate the base e exponential of the input argument.

double __nv_expm1(double x)
Calculate the base e exponential of the input argument, minus 1.

float __nv_expm1f(float x)
Calculate the base e exponential of the input argument, minus 1.

double __nv_fabs(double f)
Calculate the absolute value of the input argument.

float __nv_fabsf(float f)
Calculate the absolute value of the input argument.

float __nv_fadd_rd(float x, float y)
Add two floating point values in round-down mode.

float __nv_fadd_rn(float x, float y)
Add two floating point values in round-to-nearest-even mode.

float __nv_fadd_ru(float x, float y)
Add two floating point values in round-up mode.

float __nv_fadd_rz(float x, float y)
Add two floating point values in round-towards-zero mode.

float __nv_fast_cosf(float x)
Calculate the fast approximate cosine of the input argument.

float __nv_fast_exp10f(float x)
Calculate the fast approximate base 10 exponential of the input argument.

float __nv_fast_expf(float x)
Calculate the fast approximate base e exponential of the input argument.

float __nv_fast_fdividef(float x, float y)
Calculate the fast approximate division of the input arguments.

float __nv_fast_log10f(float x)
Calculate the fast approximate base 10 logarithm of the input argument.

float __nv_fast_log2f(float x)
Calculate the fast approximate base 2 logarithm of the input argument.

float __nv_fast_logf(float x)
Calculate the fast approximate base e logarithm of the input argument.

float __nv_fast_powf(float x, float y)
Calculate the fast approximate of x^y .

void __nv_fast_sincosf(float x, float *sptr, float *cptr)
Calculate the fast approximate of sine and cosine of the first input argument.

float __nv_fast_sinf(float x)
Calculate the fast approximate sine of the input argument.

float __nv_fast_tanf(float x)
Calculate the fast approximate tangent of the input argument.

float __nv_fast_tanhf(float x)
Calculate the fast approximate hyperbolic tangent of the input argument.

double __nv_fdim(double x, double y)
Compute the positive difference between x and y .

float __nv_fdimf(float x, float y)
Compute the positive difference between x and y .

float __nv_fdiv_rd(float x, float y)
Divide two floating point values in round-down mode.

float __nv_fdiv_rn(float x, float y)
Divide two floating point values in round-to-nearest-even mode.

float __nv_fdiv_ru(float x, float y)
Divide two floating point values in round-up mode.

float __nv_fdiv_rz(float x, float y)
Divide two floating point values in round-towards-zero mode.

int __nv_ffs(int x)
Find the position of the least significant bit set to 1 in a 32 bit integer.

int __nv_ffsll(long long int x)
Find the position of the least significant bit set to 1 in a 64 bit integer.

int ***__nv_finitef*****(float x)**
Determine whether argument is finite.

unsigned short ***__nv_float2half_rn*****(float f)**
Convert a single-precision float to a half-precision float in round-to-nearest-even mode.

int ***__nv_float2int_rd*****(float in)**
Convert a float to a signed integer in round-down mode.

int ***__nv_float2int_rn*****(float in)**
Convert a float to a signed integer in round-to-nearest-even mode.

int ***__nv_float2int_ru*****(float in)**
Convert a float to a signed integer in round-up mode.

int ***__nv_float2int_rz*****(float in)**
Convert a float to a signed integer in round-towards-zero mode.

long long ***__nv_float2ll_rd*****(float f)**
Convert a float to a signed 64-bit integer in round-down mode.

long long ***__nv_float2ll_rn*****(float f)**
Convert a float to a signed 64-bit integer in round-to-nearest-even mode.

long long ***__nv_float2ll_ru*****(float f)**
Convert a float to a signed 64-bit integer in round-up mode.

long long ***__nv_float2ll_rz*****(float f)**
Convert a float to a signed 64-bit integer in round-towards-zero mode.

unsigned int ***__nv_float2uint_rd*****(float in)**
Convert a float to an unsigned integer in round-down mode.

unsigned int ***__nv_float2uint_rn*****(float in)**
Convert a float to an unsigned integer in round-to-nearest-even mode.

unsigned int ***__nv_float2uint_ru*****(float in)**
Convert a float to an unsigned integer in round-up mode.

unsigned int ***__nv_float2uint_rz*****(float in)**
Convert a float to an unsigned integer in round-towards-zero mode.

unsigned long long ***__nv_float2ull_rd*****(float f)**
Convert a float to an unsigned 64-bit integer in round-down mode.

unsigned long long ***__nv_float2ull_rn*****(float f)**
Convert a float to an unsigned 64-bit integer in round-to-nearest-even mode.

unsigned long long ***__nv_float2ull_ru*****(float f)**
Convert a float to an unsigned 64-bit integer in round-up mode.

unsigned long long ***__nv_float2ull_rz*****(float f)**
Convert a float to an unsigned 64-bit integer in round-towards-zero mode.

int ***__nv_float_as_int*****(float x)**
Reinterpret bits in a float as a signed integer.

unsigned int ***__nv_float_as_uint*****(float x)**
Reinterpret bits in a float as a unsigned integer.

double ***__nv_floor*****(double f)**
Calculate the largest integer less than or equal to x .

float __nv_floorf(float f)
Calculate the largest integer less than or equal to x .

double __nv_fma(double x, double y, double z)
Compute $x \times y + z$ as a single operation.

double __nv_fma_rd(double x, double y, double z)
Compute $x \times y + z$ as a single operation in round-down mode.

double __nv_fma_rn(double x, double y, double z)
Compute $x \times y + z$ as a single operation in round-to-nearest-even mode.

double __nv_fma_ru(double x, double y, double z)
Compute $x \times y + z$ as a single operation in round-up mode.

double __nv_fma_rz(double x, double y, double z)
Compute $x \times y + z$ as a single operation in round-towards-zero mode.

float __nv_fmaf(float x, float y, float z)
Compute $x \times y + z$ as a single operation.

float __nv_fmaf_ieee_rd(float x, float y, float z)
DOCUMENTATION MISSING.

float __nv_fmaf_ieee_rn(float x, float y, float z)
DOCUMENTATION MISSING.

float __nv_fmaf_ieee_ru(float x, float y, float z)
DOCUMENTATION MISSING.

float __nv_fmaf_ieee_rz(float x, float y, float z)
DOCUMENTATION MISSING.

float __nv_fmaf_rd(float x, float y, float z)
Compute $x \times y + z$ as a single operation, in round-down mode.

float __nv_fmaf_rn(float x, float y, float z)
Compute $x \times y + z$ as a single operation, in round-to-nearest-even mode.

float __nv_fmaf_ru(float x, float y, float z)
Compute $x \times y + z$ as a single operation, in round-up mode.

float __nv_fmaf_rz(float x, float y, float z)
Compute $x \times y + z$ as a single operation, in round-towards-zero mode.

double __nv_fmax(double x, double y)
Determine the maximum numeric value of the arguments.

float __nv_fmaxf(float x, float y)
Determine the maximum numeric value of the arguments.

double __nv_fmin(double x, double y)
Determine the minimum numeric value of the arguments.

float __nv_fminf(float x, float y)
Determine the minimum numeric value of the arguments.

double __nv_fmod(double x, double y)
Calculate the double-precision floating-point remainder of x / y .

float __nv_fmodf(float x, float y)
Calculate the floating-point remainder of x / y .

float __nv_fmul_rd(float x, float y)
Multiply two floating point values in round-down mode.

float __nv_fmul_rn(float x, float y)
Multiply two floating point values in round-to-nearest-even mode.

float __nv_fmul_ru(float x, float y)
Multiply two floating point values in round-up mode.

float __nv_fmul_rz(float x, float y)
Multiply two floating point values in round-towards-zero mode.

float __nv_frcp_rd(float x)
Compute $\frac{1}{x}$ in round-down mode.

float __nv_frcp_rn(float x)
Compute $\frac{1}{x}$ in round-to-nearest-even mode.

float __nv_frcp_ru(float x)
Compute $\frac{1}{x}$ in round-up mode.

float __nv_frcp_rz(float x)
Compute $\frac{1}{x}$ in round-towards-zero mode.

double __nv_frexp(double x, int *b)
Extract mantissa and exponent of a floating-point value.

float __nv_frexpf(float x, int *b)
Extract mantissa and exponent of a floating-point value.

float __nv_frsqrt_rn(float x)
Compute $1/\sqrt{x}$ in round-to-nearest-even mode.

float __nv_fsqrt_rd(float x)
Compute $\sqrt{x}$ in round-down mode.

float __nv_fsqrt_rn(float x)
Compute $\sqrt{x}$ in round-to-nearest-even mode.

float __nv_fsqrt_ru(float x)
Compute $\sqrt{x}$ in round-up mode.

float __nv_fsqrt_rz(float x)
Compute $\sqrt{x}$ in round-towards-zero mode.

float __nv_fsub_rd(float x, float y)
Subtract two floating point values in round-down mode.

float __nv_fsub_rn(float x, float y)
Subtract two floating point values in round-to-nearest-even mode.

float __nv_fsub_ru(float x, float y)
Subtract two floating point values in round-up mode.

float __nv_fsub_rz(float x, float y)
Subtract two floating point values in round-towards-zero mode.

int __nv_hadd(int x, int y)
Compute average of signed input arguments, avoiding overflow in the intermediate sum.

float __nv_half2float(unsigned short h)
Convert a half-precision float to a single-precision float in round-to-nearest-even mode.

double ***__nv_hiloint2double*****(int x, int y)**
Reinterpret high and low 32-bit integer values as a double.

double ***__nv_hypot*****(double x, double y)**
Calculate the square root of the sum of squares of two arguments.

float ***__nv_hypotf*****(float x, float y)**
Calculate the square root of the sum of squares of two arguments.

int ***__nv_ilogb*****(double x)**
Compute the unbiased integer exponent of the argument.

int ***__nv_ilogbf*****(float x)**
Compute the unbiased integer exponent of the argument.

double ***__nv_int2double_rn*****(int i)**
Convert a signed int to a double.

float ***__nv_int2float_rd*****(int in)**
Convert a signed integer to a float in round-down mode.

float ***__nv_int2float_rn*****(int in)**
Convert a signed integer to a float in round-to-nearest-even mode.

float ***__nv_int2float_ru*****(int in)**
Convert a signed integer to a float in round-up mode.

float ***__nv_int2float_rz*****(int in)**
Convert a signed integer to a float in round-towards-zero mode.

float ***__nv_int_as_float*****(int x)**
Reinterpret bits in an integer as a float.

int ***__nv_isfinited*****(double x)**
Determine whether argument is finite.

int ***__nv_isinfd*****(double x)**
Determine whether argument is infinite.

int ***__nv_isinff*****(float x)**
Determine whether argument is infinite.

int ***__nv_isnand*****(double x)**
Determine whether argument is a NaN.

int ***__nv_isnanf*****(float x)**
Determine whether argument is a NaN.

double ***__nv_j0*****(double x)**
Calculate the value of the Bessel function of the first kind of order 0 for the input argument.

float ***__nv_j0f*****(float x)**
Calculate the value of the Bessel function of the first kind of order 0 for the input argument.

double ***__nv_j1*****(double x)**
Calculate the value of the Bessel function of the first kind of order 1 for the input argument.

float ***__nv_j1f*****(float x)**
Calculate the value of the Bessel function of the first kind of order 1 for the input argument.

double ***__nv_jn*****(int n, double x)**
Calculate the value of the Bessel function of the first kind of order n for the input argument.

float __nv_jnf(int n, float x)
Calculate the value of the Bessel function of the first kind of order n for the input argument.

double __nv_ldexp(double x, int y)
Calculate the value of $x \cdot 2^{exp}$.

float __nv_ldexpf(float x, int y)
Calculate the value of $x \cdot 2^{exp}$.

double __nv_lgamma(double x)
Calculate the natural logarithm of the absolute value of the gamma function of the input argument.

float __nv_lgammaf(float x)
Calculate the natural logarithm of the absolute value of the gamma function of the input argument.

double __nv_ll2double_rd(long long l)
Convert a signed 64-bit int to a double in round-down mode.

double __nv_ll2double_rn(long long l)
Convert a signed 64-bit int to a double in round-to-nearest-even mode.

double __nv_ll2double_ru(long long l)
Convert a signed 64-bit int to a double in round-up mode.

double __nv_ll2double_rz(long long l)
Convert a signed 64-bit int to a double in round-towards-zero mode.

float __nv_ll2float_rd(long long l)
Convert a signed integer to a float in round-down mode.

float __nv_ll2float_rn(long long l)
Convert a signed 64-bit integer to a float in round-to-nearest-even mode.

float __nv_ll2float_ru(long long l)
Convert a signed integer to a float in round-up mode.

float __nv_ll2float_rz(long long l)
Convert a signed integer to a float in round-towards-zero mode.

long long __nv_llabs(long long x)
Determine the absolute value of a 64-bit signed integer.

long long __nv_llmax(long long x, long long y)
Determine the maximum value of two 64-bit signed integers.

long long __nv_llmin(long long x, long long y)
Determine the minimum value of two 64-bit signed integers.

long long int __nv_llrint(double x)
Round input to nearest integer value.

long long int __nv_llrintf(float x)
Round input to nearest integer value.

long long int __nv_llround(double x)
Round to nearest integer value.

long long int __nv_llroundf(float x)
Round to nearest integer value.

double __nv_log(double x)
Calculate the base e logarithm of the input argument.

double __nv_log10(double x)
Calculate the base 10 logarithm of the input argument.

float __nv_log10f(float x)
Calculate the base 10 logarithm of the input argument.

double __nv_log1p(double x)
Calculate the value of $log_e(1 + x)$ $\lfloor x \rfloor$.

float __nv_log1pf(float x)
Calculate the value of $log_e(1 + x)$ $\lfloor x \rfloor$.

double __nv_log2(double x)
Calculate the base 2 logarithm of the input argument.

float __nv_log2f(float x)
Calculate the base 2 logarithm of the input argument.

double __nv_logb(double x)
Calculate the floating point representation of the exponent of the input argument.

float __nv_logbf(float x)
Calculate the floating point representation of the exponent of the input argument.

float __nv_logf(float x)
Calculate the base e logarithm of the input argument.

double __nv_longlong_as_double(long long x)
Reinterpret bits in a 64-bit signed integer as a double.

int __nv_max(int x, int y)
Determine the maximum value of two 32-bit signed integers.

int __nv_min(int x, int y)
Determine the minimum value of two 32-bit signed integers.

double __nv_modf(double x, double *b)
Break down the input argument into fractional and integral parts.

float __nv_modff(float x, float *b)
Break down the input argument into fractional and integral parts.

int __nv_mul24(int x, int y)
Calculate the least significant 32 bits of the product of the least significant 24 bits of two integers.

long long __nv_mul64hi(long long x, long long y)
Calculate the most significant 64 bits of the product of the two 64 bit integers.

int __nv_mulhi(int x, int y)
Calculate the most significant 32 bits of the product of the two 32 bit integers.

double __nv_nan(const signed char *tagp)
Returns "Not a Number" value.

float __nv_nanf(const signed char *tagp)
Returns "Not a Number" value.

double __nv_nearbyint(double x)
Round the input argument to the nearest integer.

float __nv_nearbyintf(float x)
Round the input argument to the nearest integer.

double ***__nv_nextafter*****(double x, double y)**
Return next representable double-precision floating-point value after argument.

float ***__nv_nextafterf*****(float x, float y)**
Return next representable double-precision floating-point value after argument.

double ***__nv_norm*****(int dim, double const *a)**
Calculate the square root of the sum of squares of any number of coordinates.

double ***__nv_norm3d*****(double a, double b, double c)**
Calculate the square root of the sum of squares of three coordinates of argument.

float ***__nv_norm3df*****(float a, float b, float c)**
Calculate the square root of the sum of squares of three coordinates of argument.

double ***__nv_norm4d*****(double a, double b, double c, double d)**
Calculate the square root of the sum of squares of four coordinates of argument.

float ***__nv_norm4df*****(float a, float b, float c, float d)**
Calculate the square root of the sum of squares of four coordinates of argument.

double ***__nv_normcdf*****(double x)**
Calculate the standard normal cumulative distribution function.

float ***__nv_normcdff*****(float x)**
Calculate the standard normal cumulative distribution function.

double ***__nv_normcdfinv*****(double x)**
Calculate the inverse of the standard normal cumulative distribution function.

float ***__nv_normcdfinvf*****(float x)**
Calculate the inverse of the standard normal cumulative distribution function.

float ***__nv_normf*****(int dim, float const *a)**
Calculate the square root of the sum of squares of any number of coordinates.

int ***__nv_popc*****(int x)**
Count the number of bits that are set to 1 in a 32 bit integer.

int ***__nv_popcll*****(long long x)**
Count the number of bits that are set to 1 in a 64 bit integer.

double ***__nv_pow*****(double x, double y)**
Calculate the value of first argument to the power of second argument.

float ***__nv_powf*****(float x, float y)**
Calculate the value of first argument to the power of second argument.

double ***__nv_powi*****(double x, int y)**
Calculate the value of first argument to the power of second argument.

float ***__nv_powif*****(float x, int y)**
Calculate the value of first argument to the power of second argument.

double ***__nv_rcbrt*****(double x)**
Calculate reciprocal cube root function.

float ***__nv_rcbrtf*****(float x)**
Calculate reciprocal cube root function.

double ***__nv_rcp64h*****(double d)**
DOCUMENTATION MISSING.

double *__nv_remainder*(double x, double y)
Compute double-precision floating-point remainder.

float *__nv_remainderf*(float x, float y)
Compute double-precision floating-point remainder.

double *__nv_remquo*(double x, double y, int *c)
Compute double-precision floating-point remainder and part of quotient.

float *__nv_remquof*(float x, float y, int *quo)
Compute double-precision floating-point remainder and part of quotient.

int *__nv_rhadd*(int x, int y)
Compute rounded average of signed input arguments, avoiding overflow in the intermediate sum.

double *__nv_rhypot*(double x, double y)
Calculate the reciprocal square root of the sum of squares of two arguments.

float *__nv_rhypotf*(float x, float y)
Calculate the reciprocal square root of the sum of squares of two arguments.

double *__nv_rint*(double x)
Round to nearest integer value in floating-point.

float *__nv_rintf*(float x)
Round to nearest integer value in floating-point.

double *__nv_rnorm*(int dim, double const *a)
Calculate the reciprocal of square root of the sum of squares of any number of coordinates.

double *__nv_rnorm3d*(double a, double b, double c)
Calculate the reciprocal square root of the sum of squares of three coordinates of argument.

float *__nv_rnorm3df*(float a, float b, float c)
Calculate the reciprocal square root of the sum of squares of three coordinates of argument.

double *__nv_rnorm4d*(double a, double b, double c, double d)
Calculate the reciprocal square root of the sum of squares of four coordinates of argument.

float *__nv_rnorm4df*(float a, float b, float c, float d)
Calculate the reciprocal square root of the sum of squares of four coordinates of argument.

float *__nv_rnormf*(int dim, float const *a)
Calculate the reciprocal of square root of the sum of squares of any number of coordinates.

double *__nv_round*(double x)
Round to nearest integer value in floating-point.

float *__nv_roundf*(float x)
Round to nearest integer value in floating-point.

double *__nv_rsqrt*(double x)
Calculate the reciprocal of the square root of the input argument.

float *__nv_rsqrtf*(float x)
Calculate the reciprocal of the square root of the input argument.

int *__nv_sad*(int x, int y, int z)
Calculate $|x - y| + z$, the sum of absolute difference.

float *__nv_saturatef*(float x)
Clamp the input argument to [+0.0, 1.0].

double __nv_scalbn(double x, int y)
Scale floating-point input by integer power of two.

float __nv_scalbnf(float x, int y)
Scale floating-point input by integer power of two.

int __nv_signbitd(double x)
Return the sign bit of the input.

int __nv_signbitf(float x)
Return the sign bit of the input.

double __nv_sin(double x)
Calculate the sine of the input argument.

void __nv_sincos(double x, double *sptr, double *cptr)
Calculate the sine and cosine of the first input argument.

void __nv_sincosf(float x, float *sptr, float *cptr)
Calculate the sine and cosine of the first input argument.

void __nv_sincospi(double x, double *sptr, double *cptr)
Calculate the sine and cosine of the first input argument $\times\pi$.

void __nv_sincospif(float x, float *sptr, float *cptr)
Calculate the sine and cosine of the first input argument $\times\pi$.

float __nv_sinf(float x)
Calculate the sine of the input argument.

double __nv_sinh(double x)
Calculate the hyperbolic sine of the input argument.

float __nv_sinhf(float x)
Calculate the hyperbolic sine of the input argument.

double __nv_sinpi(double x)
Calculate the sine of the input argument $\times\pi$.

float __nv_sinpif(float x)
Calculate the sine of the input argument $\times\pi$.

double __nv_sqrt(double x)
Calculate the square root of the input argument.

float __nv_sqrtf(float x)
Calculate the square root of the input argument.

double __nv_tan(double x)
Calculate the tangent of the input argument.

float __nv_tanf(float x)
Calculate the tangent of the input argument.

double __nv_tanh(double x)
Calculate the hyperbolic tangent of the input argument.

float __nv_tanhf(float x)
Calculate the hyperbolic tangent of the input argument.

double __nv_tgamma(double x)
Calculate the gamma function of the input argument.

float ***__nv_tgammaf*****(float x)**
Calculate the gamma function of the input argument.

double ***__nv_trunc*****(double x)**
Truncate input argument to the integral part.

float ***__nv_truncf*****(float x)**
Truncate input argument to the integral part.

unsigned int ***__nv_uhadd*****(unsigned int x, unsigned int y)**
Compute average of unsigned input arguments, avoiding overflow in the intermediate sum.

double ***__nv_uint2double_rn*****(unsigned int i)**
Convert an unsigned int to a double.

float ***__nv_uint2float_rd*****(unsigned int in)**
Convert an unsigned integer to a float in round-down mode.

float ***__nv_uint2float_rn*****(unsigned int in)**
Convert an unsigned integer to a float in round-to-nearest-even mode.

float ***__nv_uint2float_ru*****(unsigned int in)**
Convert an unsigned integer to a float in round-up mode.

float ***__nv_uint2float_rz*****(unsigned int in)**
Convert an unsigned integer to a float in round-towards-zero mode.

float ***__nv_uint_as_float*****(unsigned int x)**
Reinterpret bits in an unsigned integer as a float.

double ***__nv_ull2double_rd*****(unsigned long long l)**
Convert an unsigned 64-bit int to a double in round-down mode.

double ***__nv_ull2double_rn*****(unsigned long long l)**
Convert an unsigned 64-bit int to a double in round-to-nearest-even mode.

double ***__nv_ull2double_ru*****(unsigned long long l)**
Convert an unsigned 64-bit int to a double in round-up mode.

double ***__nv_ull2double_rz*****(unsigned long long l)**
Convert an unsigned 64-bit int to a double in round-towards-zero mode.

float ***__nv_ull2float_rd*****(unsigned long long l)**
Convert an unsigned integer to a float in round-down mode.

float ***__nv_ull2float_rn*****(unsigned long long l)**
Convert an unsigned integer to a float in round-to-nearest-even mode.

float ***__nv_ull2float_ru*****(unsigned long long l)**
Convert an unsigned integer to a float in round-up mode.

float ***__nv_ull2float_rz*****(unsigned long long l)**
Convert an unsigned integer to a float in round-towards-zero mode.

unsigned long long ***__nv_ullmax*****(unsigned long long x, unsigned long long y)**
Determine the maximum value of two 64-bit unsigned integers.

unsigned long long ***__nv_ullmin*****(unsigned long long x, unsigned long long y)**
Determine the minimum value of two 64-bit unsigned integers.

unsigned int ***__nv_umax*****(unsigned int x, unsigned int y)**
Determine the maximum value of two 32-bit unsigned integers.

unsigned int __nv_umin(unsigned int x, unsigned int y)
Determine the minimum value of two 32-bit unsigned integers.

unsigned int __nv_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.

unsigned long long __nv_umul64hi(unsigned long long x, unsigned long long y)
Calculate the most significant 64 bits of the product of the two 64 unsigned bit integers.

unsigned int __nv_umulhi(unsigned int x, unsigned int y)
Calculate the most significant 32 bits of the product of the two 32 bit unsigned integers.

unsigned int __nv_urhadd(unsigned int x, unsigned int y)
Compute rounded average of unsigned input arguments, avoiding overflow in the intermediate sum.

unsigned int __nv_usad(unsigned int x, unsigned int y, unsigned int z)
Calculate $|x - y| + z$, the sum of absolute difference.

unsigned int __nv_vabs2(unsigned int a)

unsigned int __nv_vabs4(unsigned int a)

unsigned int __nv_vabsdiffs2(unsigned int a, unsigned int b)

unsigned int __nv_vabsdiffs4(unsigned int a, unsigned int b)

unsigned int __nv_vabsdiffu2(unsigned int a, unsigned int b)

unsigned int __nv_vabsdiffu4(unsigned int a, unsigned int b)

unsigned int __nv_vabsss2(unsigned int a)

unsigned int __nv_vabsss4(unsigned int a)

unsigned int __nv_vadd2(unsigned int a, unsigned int b)

unsigned int __nv_vadd4(unsigned int a, unsigned int b)

unsigned int __nv_vaddss2(unsigned int a, unsigned int b)

unsigned int __nv_vaddss4(unsigned int a, unsigned int b)

unsigned int __nv_vaddus2(unsigned int a, unsigned int b)

unsigned int __nv_vaddus4(unsigned int a, unsigned int b)

unsigned int __nv_vavgs2(unsigned int a, unsigned int b)

unsigned int __nv_vavgs4(unsigned int a, unsigned int b)

unsigned int __nv_vavgu2(unsigned int a, unsigned int b)

unsigned int __nv_vavgu4(unsigned int a, unsigned int b)

unsigned int __nv_vcmpeq2(unsigned int a, unsigned int b)

unsigned int __nv_vcmpeq4(unsigned int a, unsigned int b)

unsigned int __nv_vcmpges2(unsigned int a, unsigned int b)

unsigned int __nv_vcmpges4(unsigned int a, unsigned int b)

unsigned int __nv_vcmpgeu2(unsigned int a, unsigned int b)

unsigned int __nv_vcmpgeu4(unsigned int a, unsigned int b)

unsigned int __nv_vcmpgts2(unsigned int a, unsigned int b)

unsigned int __nv_vcmpgts4(unsigned int a, unsigned int b)

unsigned int __nv_vcmpgtu2(unsigned int a, unsigned int b)

unsigned int __nv_vcmpgtu4(unsigned int a, unsigned int b)

unsigned int __nv_vcmples2(unsigned int a, unsigned int b)

unsigned int __nv_vcmples4(unsigned int a, unsigned int b)

unsigned int __nv_vcmpleu2(unsigned int a, unsigned int b)

unsigned int __nv_vcmpleu4(unsigned int a, unsigned int b)

unsigned int __nv_vcmplts2(unsigned int a, unsigned int b)

unsigned int __nv_vcmplts4(unsigned int a, unsigned int b)

unsigned int __nv_vcmpltu2(unsigned int a, unsigned int b)

unsigned int __nv_vcmpltu4(unsigned int a, unsigned int b)

unsigned int *__nv_vcmpne2*(unsigned int a, unsigned int b)

unsigned int *__nv_vcmpne4*(unsigned int a, unsigned int b)

unsigned int *__nv_vhaddu2*(unsigned int a, unsigned int b)

unsigned int *__nv_vhaddu4*(unsigned int a, unsigned int b)

unsigned int *__nv_vmaxs2*(unsigned int a, unsigned int b)

unsigned int *__nv_vmaxs4*(unsigned int a, unsigned int b)

unsigned int *__nv_vmaxu2*(unsigned int a, unsigned int b)

unsigned int *__nv_vmaxu4*(unsigned int a, unsigned int b)

unsigned int *__nv_vmins2*(unsigned int a, unsigned int b)

unsigned int *__nv_vmins4*(unsigned int a, unsigned int b)

unsigned int *__nv_vminu2*(unsigned int a, unsigned int b)

unsigned int *__nv_vminu4*(unsigned int a, unsigned int b)

unsigned int *__nv_vneg2*(unsigned int a)

unsigned int *__nv_vneg4*(unsigned int a)

unsigned int *__nv_vnegss2*(unsigned int a)

unsigned int *__nv_vnegss4*(unsigned int a)

unsigned int *__nv_vsads2*(unsigned int a, unsigned int b)

unsigned int *__nv_vsads4*(unsigned int a, unsigned int b)

unsigned int *__nv_vsadu2*(unsigned int a, unsigned int b)

unsigned int *__nv_vsadu4*(unsigned int a, unsigned int b)

unsigned int *__nv_vseteq2*(unsigned int a, unsigned int b)

unsigned int __nv_vseteq4(unsigned int a, unsigned int b)

unsigned int __nv_vsetges2(unsigned int a, unsigned int b)

unsigned int __nv_vsetges4(unsigned int a, unsigned int b)

unsigned int __nv_vsetgeu2(unsigned int a, unsigned int b)

unsigned int __nv_vsetgeu4(unsigned int a, unsigned int b)

unsigned int __nv_vsetgts2(unsigned int a, unsigned int b)

unsigned int __nv_vsetgts4(unsigned int a, unsigned int b)

unsigned int __nv_vsetgtu2(unsigned int a, unsigned int b)

unsigned int __nv_vsetgtu4(unsigned int a, unsigned int b)

unsigned int __nv_vsetles2(unsigned int a, unsigned int b)

unsigned int __nv_vsetles4(unsigned int a, unsigned int b)

unsigned int __nv_vsetleu2(unsigned int a, unsigned int b)

unsigned int __nv_vsetleu4(unsigned int a, unsigned int b)

unsigned int __nv_vsetlts2(unsigned int a, unsigned int b)

unsigned int __nv_vsetlts4(unsigned int a, unsigned int b)

unsigned int __nv_vsetltu2(unsigned int a, unsigned int b)

unsigned int __nv_vsetltu4(unsigned int a, unsigned int b)

unsigned int __nv_vsetne2(unsigned int a, unsigned int b)

unsigned int __nv_vsetne4(unsigned int a, unsigned int b)

unsigned int __nv_vsub2(unsigned int a, unsigned int b)

unsigned int __nv_vsub4(unsigned int a, unsigned int b)

unsigned int *__nv_vsubss2*(unsigned int a, unsigned int b)

unsigned int *__nv_vsubss4*(unsigned int a, unsigned int b)

unsigned int *__nv_vsubus2*(unsigned int a, unsigned int b)

unsigned int *__nv_vsubus4*(unsigned int a, unsigned int b)

double *__nv_y0*(double x)
Calculate the value of the Bessel function of the second kind of order 0 for the input argument.

float *__nv_y0f*(float x)
Calculate the value of the Bessel function of the second kind of order 0 for the input argument.

double *__nv_y1*(double x)
Calculate the value of the Bessel function of the second kind of order 1 for the input argument.

float *__nv_y1f*(float x)
Calculate the value of the Bessel function of the second kind of order 1 for the input argument.

double *__nv_yn*(int n, double x)
Calculate the value of the Bessel function of the second kind of order n for the input argument.

float *__nv_ynf*(int n, float x)
Calculate the value of the Bessel function of the second kind of order n for the input argument.

Chapter 4. Notices

4.1. 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

services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA's aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

4.2. OpenCL

OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc.

4.3. Trademarks

NVIDIA and the NVIDIA logo are trademarks or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

Index

Non-alphabetical