



# CUDA Math API

## API Reference Manual

# Table of Contents

<b>Chapter 1. Modules</b> .....	<b>1</b>
1.1. FP8 Intrinsics.....	2
FP8 Conversion and Data Movement.....	2
C++ struct for handling fp8 data type of e5m2 kind.....	2
C++ struct for handling vector type of two fp8 values of e5m2 kind.....	2
C++ struct for handling vector type of four fp8 values of e5m2 kind.....	3
C++ struct for handling fp8 data type of e4m3 kind.....	3
C++ struct for handling vector type of two fp8 values of e4m3 kind.....	3
C++ struct for handling vector type of four fp8 values of e4m3 kind.....	3
1.1.1. FP8 Conversion and Data Movement.....	3
__nv_fp8_interpretation_t.....	3
__nv_saturation_t.....	3
__nv_fp8_storage_t.....	4
__nv_fp8x2_storage_t.....	4
__nv_fp8x4_storage_t.....	4
__nv_cvt_bfloat16raw2_to_fp8x2.....	4
__nv_cvt_bfloat16raw_to_fp8.....	4
__nv_cvt_double2_to_fp8x2.....	5
__nv_cvt_double_to_fp8.....	5
__nv_cvt_float2_to_fp8x2.....	6
__nv_cvt_float_to_fp8.....	6
__nv_cvt_fp8_to_halfraw.....	7
__nv_cvt_fp8x2_to_halfraw2.....	7
__nv_cvt_halfraw2_to_fp8x2.....	7
__nv_cvt_halfraw_to_fp8.....	8
1.1.2. C++ struct for handling fp8 data type of e5m2 kind.....	8
__nv_fp8_e5m2.....	8
__nv_fp8_e5m2::__x.....	8
__nv_fp8_e5m2::__nv_fp8_e5m2.....	8
__nv_fp8_e5m2::__nv_fp8_e5m2.....	9
__nv_fp8_e5m2::__nv_fp8_e5m2.....	9
__nv_fp8_e5m2::__nv_fp8_e5m2.....	9
__nv_fp8_e5m2::__nv_fp8_e5m2.....	9
__nv_fp8_e5m2::__nv_fp8_e5m2.....	9
__nv_fp8_e5m2::__nv_fp8_e5m2.....	10
__nv_fp8_e5m2::__nv_fp8_e5m2.....	10

__nv_fp8_e5m2::__nv_fp8_e5m2.....	10
__nv_fp8_e5m2::__nv_fp8_e5m2.....	10
__nv_fp8_e5m2::__nv_fp8_e5m2.....	10
__nv_fp8_e5m2::__nv_fp8_e5m2.....	11
__nv_fp8_e5m2::__nv_fp8_e5m2.....	11
__nv_fp8_e5m2::operator __half.....	11
__nv_fp8_e5m2::operator __nv_bfloat16.....	11
__nv_fp8_e5m2::operator bool.....	11
__nv_fp8_e5m2::operator char.....	11
__nv_fp8_e5m2::operator double.....	12
__nv_fp8_e5m2::operator float.....	12
__nv_fp8_e5m2::operator int.....	12
__nv_fp8_e5m2::operator long int.....	12
__nv_fp8_e5m2::operator long long int.....	12
__nv_fp8_e5m2::operator short int.....	12
__nv_fp8_e5m2::operator signed char.....	13
__nv_fp8_e5m2::operator unsigned char.....	13
__nv_fp8_e5m2::operator unsigned int.....	13
__nv_fp8_e5m2::operator unsigned long int.....	13
__nv_fp8_e5m2::operator unsigned long long int.....	13
__nv_fp8_e5m2::operator unsigned short int.....	14
1.1.3. C++ struct for handling vector type of two fp8 values of e5m2 kind.....	14
__nv_fp8x2_e5m2.....	14
__nv_fp8x2_e5m2::__x.....	14
__nv_fp8x2_e5m2::__nv_fp8x2_e5m2.....	14
__nv_fp8x2_e5m2::__nv_fp8x2_e5m2.....	14
__nv_fp8x2_e5m2::__nv_fp8x2_e5m2.....	15
__nv_fp8x2_e5m2::__nv_fp8x2_e5m2.....	15
__nv_fp8x2_e5m2::__nv_fp8x2_e5m2.....	15
__nv_fp8x2_e5m2::operator __half2.....	15
__nv_fp8x2_e5m2::operator float2.....	15
1.1.4. C++ struct for handling vector type of four fp8 values of e5m2 kind.....	15
__nv_fp8x4_e5m2.....	16
__nv_fp8x4_e5m2::__x.....	16
__nv_fp8x4_e5m2::__nv_fp8x4_e5m2.....	16
__nv_fp8x4_e5m2::__nv_fp8x4_e5m2.....	16
__nv_fp8x4_e5m2::__nv_fp8x4_e5m2.....	16
__nv_fp8x4_e5m2::__nv_fp8x4_e5m2.....	16

__nv_fp8x4_e5m2::__nv_fp8x4_e5m2.....	17
__nv_fp8x4_e5m2::operator float4.....	17
1.1.5. C++ struct for handling fp8 data type of e4m3 kind.....	17
__nv_fp8_e4m3.....	17
__nv_fp8_e4m3::__x.....	17
__nv_fp8_e4m3::__nv_fp8_e4m3.....	17
__nv_fp8_e4m3::__nv_fp8_e4m3.....	17
__nv_fp8_e4m3::__nv_fp8_e4m3.....	18
__nv_fp8_e4m3::__nv_fp8_e4m3.....	18
__nv_fp8_e4m3::__nv_fp8_e4m3.....	18
__nv_fp8_e4m3::__nv_fp8_e4m3.....	18
__nv_fp8_e4m3::__nv_fp8_e4m3.....	18
__nv_fp8_e4m3::__nv_fp8_e4m3.....	19
__nv_fp8_e4m3::__nv_fp8_e4m3.....	19
__nv_fp8_e4m3::__nv_fp8_e4m3.....	19
__nv_fp8_e4m3::__nv_fp8_e4m3.....	19
__nv_fp8_e4m3::__nv_fp8_e4m3.....	19
__nv_fp8_e4m3::__nv_fp8_e4m3.....	20
__nv_fp8_e4m3::operator __half.....	20
__nv_fp8_e4m3::operator __nv_bfloat16.....	20
__nv_fp8_e4m3::operator bool.....	20
__nv_fp8_e4m3::operator char.....	20
__nv_fp8_e4m3::operator double.....	20
__nv_fp8_e4m3::operator float.....	21
__nv_fp8_e4m3::operator int.....	21
__nv_fp8_e4m3::operator long int.....	21
__nv_fp8_e4m3::operator long long int.....	21
__nv_fp8_e4m3::operator short int.....	21
__nv_fp8_e4m3::operator signed char.....	21
__nv_fp8_e4m3::operator unsigned char.....	22
__nv_fp8_e4m3::operator unsigned int.....	22
__nv_fp8_e4m3::operator unsigned long int.....	22
__nv_fp8_e4m3::operator unsigned long long int.....	22
__nv_fp8_e4m3::operator unsigned short int.....	22
1.1.6. C++ struct for handling vector type of two fp8 values of e4m3 kind.....	23
__nv_fp8x2_e4m3.....	23
__nv_fp8x2_e4m3::__x.....	23
__nv_fp8x2_e4m3::__nv_fp8x2_e4m3.....	23

__nv_fp8x2_e4m3::__nv_fp8x2_e4m3.....	23
__nv_fp8x2_e4m3::__nv_fp8x2_e4m3.....	23
__nv_fp8x2_e4m3::__nv_fp8x2_e4m3.....	24
__nv_fp8x2_e4m3::__nv_fp8x2_e4m3.....	24
__nv_fp8x2_e4m3::operator __half2.....	24
__nv_fp8x2_e4m3::operator float2.....	24
1.1.7. C++ struct for handling vector type of four fp8 values of e4m3 kind.....	24
__nv_fp8x4_e4m3.....	24
__nv_fp8x4_e4m3::__x.....	24
__nv_fp8x4_e4m3::__nv_fp8x4_e4m3.....	25
__nv_fp8x4_e4m3::__nv_fp8x4_e4m3.....	25
__nv_fp8x4_e4m3::__nv_fp8x4_e4m3.....	25
__nv_fp8x4_e4m3::__nv_fp8x4_e4m3.....	25
__nv_fp8x4_e4m3::__nv_fp8x4_e4m3.....	25
__nv_fp8x4_e4m3::operator float4.....	25
1.2. Half Precision Intrinsics.....	26
__half.....	26
__half2.....	26
__half2_raw.....	26
__half_raw.....	26
Half Arithmetic Constants.....	26
Half Arithmetic Functions.....	27
Half2 Arithmetic Functions.....	27
Half Comparison Functions.....	27
Half2 Comparison Functions.....	27
Half Precision Conversion and Data Movement.....	27
Half Math Functions.....	27
Half2 Math Functions.....	27
__nv_half.....	27
__nv_half2.....	27
__nv_half2_raw.....	27
__nv_half_raw.....	27
half.....	27
half2.....	28
nv_half.....	28
nv_half2.....	28
1.2.1. Half Arithmetic Constants.....	28
CUDART_INF_FP16.....	28

CUDART_MAX_NORMAL_FP16.....	28
CUDART_MIN_DENORM_FP16.....	28
CUDART_NAN_FP16.....	28
CUDART_NEG_ZERO_FP16.....	28
CUDART_ONE_FP16.....	29
CUDART_ZERO_FP16.....	29
1.2.2. Half Arithmetic Functions.....	29
__habs.....	29
__hadd.....	29
__hadd_rn.....	30
__hadd_sat.....	30
__hdiv.....	30
__hfma.....	31
__hfma_relu.....	31
__hfma_sat.....	31
__hmul.....	32
__hmul_rn.....	32
__hmul_sat.....	32
__hneg.....	33
__hsub.....	33
__hsub_rn.....	33
__hsub_sat.....	33
atomicAdd.....	34
operator*.....	34
operator*=.....	35
operator+.....	35
operator+.....	35
operator++.....	35
operator++.....	35
operator+=.....	35
operator-.....	35
operator-.....	36
operator--.....	36
operator--.....	36
operator-=.....	36
operator/.....	36
operator/=.....	36
1.2.3. Half2 Arithmetic Functions.....	36

__h2div.....	37
__habs2.....	37
__hadd2.....	37
__hadd2_rn.....	37
__hadd2_sat.....	38
__hcmadd.....	38
__hfma2.....	39
__hfma2_relu.....	39
__hfma2_sat.....	39
__hmul2.....	40
__hmul2_rn.....	40
__hmul2_sat.....	40
__hneg2.....	41
__hsub2.....	41
__hsub2_rn.....	41
__hsub2_sat.....	42
atomicAdd.....	42
operator*.....	43
operator*=.....	43
operator+.....	43
operator+.....	43
operator++.....	43
operator++.....	44
operator+=.....	44
operator-.....	44
operator-.....	44
operator--.....	44
operator--.....	44
operator-=.....	44
operator/.....	45
operator/=.....	45
1.2.4. Half Comparison Functions.....	45
__heq.....	45
__hequ.....	46
__hge.....	46
__hgeu.....	47
__hgt.....	47
__hgtu.....	48

__hisinf.....	48
__hisnan.....	49
__hle.....	49
__hleu.....	49
__hlt.....	50
__hltu.....	50
__hmax.....	51
__hmax_nan.....	51
__hmin.....	51
__hmin_nan.....	52
__hne.....	52
__hneu.....	52
operator!=.....	53
operator<.....	53
operator<=.....	53
operator==.....	53
operator>.....	54
operator>=.....	54
1.2.5. Half2 Comparison Functions.....	54
__hbeq2.....	54
__hbequ2.....	55
__hbge2.....	55
__hbgeu2.....	56
__hbg2.....	56
__hbg2u2.....	57
__hble2.....	58
__hbleu2.....	58
__hb2lt.....	59
__hb2ltu2.....	59
__hbne2.....	60
__hbneu2.....	61
__heq2.....	61
__heq2_mask.....	62
__hequ2.....	62
__hequ2_mask.....	63
__hge2.....	63
__hge2_mask.....	64
__hgeu2.....	64



__hgeu2_mask.....	65
__hgt2.....	65
__hgt2_mask.....	66
__hgtu2.....	66
__hgtu2_mask.....	67
__hisnan2.....	67
__hle2.....	67
__hle2_mask.....	68
__hleu2.....	68
__hleu2_mask.....	69
__hlt2.....	69
__hlt2_mask.....	70
__hltu2.....	70
__hltu2_mask.....	71
__hmax2.....	71
__hmax2_nan.....	72
__hmin2.....	72
__hmin2_nan.....	72
__hne2.....	73
__hne2_mask.....	73
__hneu2.....	74
__hneu2_mask.....	74
operator!=.....	75
operator<.....	75
operator<=.....	75
operator==.....	75
operator>.....	75
operator>=.....	76
1.2.6. Half Precision Conversion and Data Movement.....	76
__double2half.....	76
__float22half2_rn.....	76
__float2half.....	77
__float2half2_rn.....	77
__float2half_rd.....	78
__float2half_rn.....	78
__float2half_ru.....	78
__float2half_rz.....	79
__floats2half2_rn.....	79

__half22float2.....	80
__half2char_rz.....	80
__half2float.....	81
__half2half2.....	81
__half2int_rd.....	81
__half2int_rn.....	82
__half2int_ru.....	82
__half2int_rz.....	83
__half2ll_rd.....	83
__half2ll_rn.....	83
__half2ll_ru.....	84
__half2ll_rz.....	84
__half2short_rd.....	85
__half2short_rn.....	85
__half2short_ru.....	86
__half2short_rz.....	86
__half2uchar_rz.....	86
__half2uint_rd.....	87
__half2uint_rn.....	87
__half2uint_ru.....	88
__half2uint_rz.....	88
__half2ull_rd.....	88
__half2ull_rn.....	89
__half2ull_ru.....	89
__half2ull_rz.....	90
__half2ushort_rd.....	90
__half2ushort_rn.....	91
__half2ushort_ru.....	91
__half2ushort_rz.....	92
__half_as_short.....	92
__half_as_ushort.....	93
__halves2half2.....	93
__high2float.....	94
__high2half.....	94
__high2half2.....	94
__highs2half2.....	95
__int2half_rd.....	95
__int2half_rn.....	96

__int2half_ru.....	96
__int2half_rz.....	96
__ldca.....	97
__ldca.....	97
__ldcg.....	97
__ldcg.....	98
__ldcs.....	98
__ldcs.....	98
__ldcv.....	98
__ldcv.....	99
__ldg.....	99
__ldg.....	99
__ldlu.....	99
__ldlu.....	100
__ll2half_rd.....	100
__ll2half_rn.....	100
__ll2half_ru.....	101
__ll2half_rz.....	101
__low2float.....	102
__low2half.....	102
__low2half2.....	102
__lowhigh2highlow.....	103
__lows2half2.....	103
__shfl_down_sync.....	104
__shfl_down_sync.....	105
__shfl_sync.....	105
__shfl_sync.....	106
__shfl_up_sync.....	107
__shfl_up_sync.....	108
__shfl_xor_sync.....	109
__shfl_xor_sync.....	110
__short2half_rd.....	110
__short2half_rn.....	111
__short2half_ru.....	111
__short2half_rz.....	112
__short_as_half.....	112
__stcg.....	112
__stcg.....	113

__stcs.....	113
__stcs.....	113
__stwb.....	113
__stwb.....	114
__stwt.....	114
__stwt.....	114
__uint2half_rd.....	114
__uint2half_rn.....	115
__uint2half_ru.....	115
__uint2half_rz.....	116
__ull2half_rd.....	116
__ull2half_rn.....	116
__ull2half_ru.....	117
__ull2half_rz.....	117
__ushort2half_rd.....	118
__ushort2half_rn.....	118
__ushort2half_ru.....	119
__ushort2half_rz.....	119
__ushort_as_half.....	120
make_half2.....	120
1.2.7. Half Math Functions.....	120
hceil.....	121
hcos.....	121
hexp.....	121
hexp10.....	122
hexp2.....	122
hfloor.....	122
hlog.....	123
hlog10.....	123
hlog2.....	124
hrcp.....	124
hrint.....	124
hrsqrt.....	125
hsin.....	125
hsqrt.....	126
htrunc.....	126
1.2.8. Half2 Math Functions.....	126
h2ceil.....	127

h2cos.....	127
h2exp.....	127
h2exp10.....	128
h2exp2.....	128
h2floor.....	129
h2log.....	129
h2log10.....	129
h2log2.....	130
h2rcp.....	130
h2rint.....	131
h2rsqrt.....	131
h2sin.....	131
h2sqrt.....	132
h2trunc.....	132
1.3. Bfloat16 Precision Intrinsics.....	132
__nv_bfloat16.....	133
__nv_bfloat162.....	133
__nv_bfloat162_raw.....	133
__nv_bfloat16_raw.....	133
Bfloat16 Arithmetic Constants.....	133
Bfloat16 Arithmetic Functions.....	133
Bfloat162 Arithmetic Functions.....	134
Bfloat16 Comparison Functions.....	134
Bfloat162 Comparison Functions.....	134
Bfloat16 Precision Conversion and Data Movement.....	134
Bfloat16 Math Functions.....	134
Bfloat162 Math Functions.....	134
nv_bfloat16.....	134
nv_bfloat162.....	134
1.3.1. Bfloat16 Arithmetic Constants.....	134
CUDA_RT_INF_BF16.....	134
CUDA_RT_MAX_NORMAL_BF16.....	135
CUDA_RT_MIN_DENORM_BF16.....	135
CUDA_RT_NAN_BF16.....	135
CUDA_RT_NEG_ZERO_BF16.....	135
CUDA_RT_ONE_BF16.....	135
CUDA_RT_ZERO_BF16.....	135
1.3.2. Bfloat16 Arithmetic Functions.....	135

__h2div.....	135
__habs.....	136
__hadd.....	136
__hadd_rn.....	136
__hadd_sat.....	136
__hdiv.....	137
__hfma.....	137
__hfma_relu.....	137
__hfma_sat.....	138
__hmul.....	138
__hmul_rn.....	139
__hmul_sat.....	139
__hneg.....	139
__hsub.....	140
__hsub_rn.....	140
__hsub_sat.....	140
atomicAdd.....	141
operator*.....	141
operator*.....	141
operator+.....	142
operator+.....	142
operator++.....	142
operator++.....	142
operator+=.....	142
operator-.....	142
operator-.....	143
operator--.....	143
operator--.....	143
operator-=.....	143
operator/.....	143
operator/=.....	143
1.3.3. Bfloat162 Arithmetic Functions.....	144
__habs2.....	144
__hadd2.....	144
__hadd2_rn.....	144
__hadd2_sat.....	145
__hcmadd.....	145
__hfma2.....	146

__hfma2_relu.....	146
__hfma2_sat.....	146
__hmul2.....	147
__hmul2_rn.....	147
__hmul2_sat.....	147
__hneg2.....	148
__hsub2.....	148
__hsub2_rn.....	148
__hsub2_sat.....	149
atomicAdd.....	149
operator*.....	150
operator*=.....	150
operator+.....	150
operator+.....	150
operator++.....	151
operator++.....	151
operator+=.....	151
operator-.....	151
operator-.....	151
operator--.....	151
operator--.....	152
operator-=.....	152
operator/.....	152
operator/=.....	152
1.3.4. Bfloat16 Comparison Functions.....	152
__heq.....	152
__hequ.....	153
__hge.....	153
__hgeu.....	154
__hgt.....	154
__hgtu.....	155
__hisinf.....	155
__hisnan.....	156
__hle.....	156
__hleu.....	157
__hlt.....	157
__hltu.....	158
__hmax.....	158

__hmax_nan.....	158
__hmin.....	159
__hmin_nan.....	159
__hne.....	159
__hneu.....	160
operator!=.....	160
operator<.....	160
operator<=.....	161
operator==.....	161
operator>.....	161
operator>=.....	161
1.3.5. Bfloat162 Comparison Functions.....	161
__hbeq2.....	162
__hbequ2.....	162
__hbge2.....	163
__hbgeu2.....	163
__hgt2.....	164
__hgtu2.....	165
__hble2.....	165
__hbleu2.....	166
__hblt2.....	166
__hbltu2.....	167
__hbne2.....	168
__hbneu2.....	168
__heq2.....	169
__heq2_mask.....	169
__hequ2.....	170
__hequ2_mask.....	170
__hge2.....	171
__hge2_mask.....	171
__hgeu2.....	172
__hgeu2_mask.....	172
__hgt2.....	173
__hgt2_mask.....	173
__hgtu2.....	174
__hgtu2_mask.....	174
__hisnan2.....	175
__hle2.....	175



__hle2_mask.....	176
__hleu2.....	176
__hleu2_mask.....	177
__hlt2.....	177
__hlt2_mask.....	178
__hltu2.....	178
__hltu2_mask.....	179
__hmax2.....	179
__hmax2_nan.....	180
__hmin2.....	180
__hmin2_nan.....	180
__hne2.....	181
__hne2_mask.....	181
__hneu2.....	182
__hneu2_mask.....	182
operator!=.....	183
operator<.....	183
operator<=.....	183
operator==.....	183
operator>.....	184
operator>=.....	184
1.3.6. Bfloat16 Precision Conversion and Data Movement.....	184
__bfloat162float2.....	184
__bfloat162bfloat162.....	185
__bfloat162char_rz.....	185
__bfloat162float.....	186
__bfloat162int_rd.....	186
__bfloat162int_rn.....	186
__bfloat162int_ru.....	187
__bfloat162int_rz.....	187
__bfloat162ll_rd.....	188
__bfloat162ll_rn.....	188
__bfloat162ll_ru.....	189
__bfloat162ll_rz.....	189
__bfloat162short_rd.....	190
__bfloat162short_rn.....	190
__bfloat162short_ru.....	191
__bfloat162short_rz.....	191

__bfloat162uchar_rz.....	192
__bfloat162uint_rd.....	192
__bfloat162uint_rn.....	193
__bfloat162uint_ru.....	193
__bfloat162uint_rz.....	194
__bfloat162ull_rd.....	194
__bfloat162ull_rn.....	195
__bfloat162ull_ru.....	195
__bfloat162ull_rz.....	196
__bfloat162ushort_rd.....	196
__bfloat162ushort_rn.....	197
__bfloat162ushort_ru.....	197
__bfloat162ushort_rz.....	198
__bfloat16_as_short.....	198
__bfloat16_as_ushort.....	199
__double2bfloat16.....	199
__float22bfloat162_rn.....	199
__float2bfloat16.....	200
__float2bfloat162_rn.....	200
__float2bfloat16_rd.....	201
__float2bfloat16_rn.....	201
__float2bfloat16_ru.....	202
__float2bfloat16_rz.....	202
__floats2bfloat162_rn.....	202
__halves2bfloat162.....	203
__high2bfloat16.....	203
__high2bfloat162.....	204
__high2float.....	204
__highs2bfloat162.....	205
__int2bfloat16_rd.....	205
__int2bfloat16_rn.....	206
__int2bfloat16_ru.....	206
__int2bfloat16_rz.....	206
__ldca.....	207
__ldca.....	207
__ldcg.....	207
__ldcg.....	208
__ldcs.....	208

__ldcs.....	208
__ldcv.....	208
__ldcv.....	209
__ldg.....	209
__ldg.....	209
__ldlu.....	209
__ldlu.....	210
__ll2bfloat16_rd.....	210
__ll2bfloat16_rn.....	210
__ll2bfloat16_ru.....	211
__ll2bfloat16_rz.....	211
__low2bfloat16.....	211
__low2bfloat162.....	212
__low2float.....	212
__lowhigh2highlow.....	213
__lows2bfloat162.....	213
__shfl_down_sync.....	214
__shfl_down_sync.....	215
__shfl_sync.....	215
__shfl_sync.....	216
__shfl_up_sync.....	217
__shfl_up_sync.....	218
__shfl_xor_sync.....	219
__shfl_xor_sync.....	220
__short2bfloat16_rd.....	220
__short2bfloat16_rn.....	221
__short2bfloat16_ru.....	221
__short2bfloat16_rz.....	222
__short_as_bfloat16.....	222
__stcg.....	222
__stcg.....	223
__stcs.....	223
__stcs.....	223
__stwb.....	223
__stwb.....	224
__stwt.....	224
__stwt.....	224
__uint2bfloat16_rd.....	224

__uint2bfloat16_rn.....	225
__uint2bfloat16_ru.....	225
__uint2bfloat16_rz.....	226
__ull2bfloat16_rd.....	226
__ull2bfloat16_rn.....	226
__ull2bfloat16_ru.....	227
__ull2bfloat16_rz.....	227
__ushort2bfloat16_rd.....	228
__ushort2bfloat16_rn.....	228
__ushort2bfloat16_ru.....	228
__ushort2bfloat16_rz.....	229
__ushort_as_bfloat16.....	229
make_bfloat162.....	230
1.3.7. Bfloat16 Math Functions.....	230
hceil.....	230
hcos.....	231
hexp.....	231
hexp10.....	231
hexp2.....	232
hfloor.....	232
hlog.....	233
hlog10.....	233
hlog2.....	233
hrcp.....	234
hrint.....	234
hrsqrt.....	235
hsin.....	235
hsqrt.....	235
htrunc.....	236
1.3.8. Bfloat162 Math Functions.....	236
h2ceil.....	236
h2cos.....	237
h2exp.....	237
h2exp10.....	238
h2exp2.....	238
h2floor.....	238
h2log.....	239
h2log10.....	239

h2log2.....	240
h2rcp.....	240
h2rint.....	240
h2rsqrt.....	241
h2sin.....	241
h2sqrt.....	242
h2trunc.....	242
1.4. Mathematical Functions.....	242
1.5. Single Precision Mathematical Functions.....	243
acosf.....	243
acoshf.....	243
asinf.....	244
asinhf.....	244
atan2f.....	245
atanf.....	246
atanhf.....	246
cbrtf.....	247
ceilf.....	247
copysignf.....	247
cosf.....	248
coshf.....	248
cospif.....	249
cyl_bessel_i0f.....	249
cyl_bessel_i1f.....	249
erfcf.....	250
erfcinvf.....	250
erfcxf.....	251
erff.....	251
erfinvf.....	252
exp10f.....	252
exp2f.....	253
expf.....	253
expm1f.....	254
fabsf.....	254
fdimf.....	255
fdividf.....	255
floorf.....	256
fmaf.....	256

fmaxf.....	257
fminf.....	257
fmodf.....	258
frexpf.....	258
hypotf.....	259
ilogbf.....	259
isfinite.....	260
isinf.....	260
isnan.....	261
j0f.....	261
j1f.....	261
jnf.....	262
ldexpf.....	262
lgammaf.....	263
llrintf.....	263
llroundf.....	264
log10f.....	264
log1pf.....	265
log2f.....	265
logbf.....	266
logf.....	266
lrintf.....	267
lroundf.....	267
max.....	267
min.....	268
modff.....	268
nanf.....	268
nearbyintf.....	269
nextafterf.....	269
norm3df.....	270
norm4df.....	270
normcdf.....	271
normcdfinvf.....	271
normf.....	272
powf.....	272
rcbrtf.....	273
remainderf.....	274
remquof.....	274

rhypotf.....	275
rintf.....	275
rnorm3df.....	276
rnorm4df.....	276
rnormf.....	277
roundf.....	277
rsqrtf.....	278
scalblnf.....	278
scalbnf.....	278
signbit.....	279
sincosf.....	279
sincospif.....	280
sinf.....	280
sinhf.....	281
sinpif.....	281
sqrtf.....	282
tanf.....	282
tanhf.....	283
tgammaf.....	283
truncf.....	284
y0f.....	284
y1f.....	284
ynf.....	285
1.6. Double Precision Mathematical Functions.....	286
acos.....	286
acosh.....	286
asin.....	287
asinh.....	287
atan.....	288
atan2.....	288
atanh.....	289
cbrt.....	289
ceil.....	290
copysign.....	290
cos.....	290
cosh.....	291
cospi.....	291
cyl_bessel_i0.....	292

cyl_bessel_i1.....	292
erf.....	292
erfc.....	293
erfcinv.....	293
erfcx.....	294
erfinv.....	294
exp.....	295
exp10.....	295
exp2.....	296
expm1.....	296
fabs.....	297
fdim.....	297
floor.....	298
fma.....	298
fmax.....	299
fmin.....	299
fmod.....	300
frexp.....	300
hypot.....	301
ilogb.....	301
isfinite.....	302
isinf.....	302
isnan.....	303
j0.....	303
j1.....	303
jn.....	304
ldexp.....	304
lgamma.....	305
llrint.....	305
llround.....	306
log.....	306
log10.....	306
log1p.....	307
log2.....	307
logb.....	308
lrint.....	308
lround.....	309
max.....	309



max.....	309
max.....	309
min.....	310
min.....	310
min.....	310
modf.....	310
nan.....	311
nearbyint.....	311
nextafter.....	312
norm.....	312
norm3d.....	313
norm4d.....	313
normcdf.....	314
normcdfinv.....	314
pow.....	315
rcbrt.....	316
remainder.....	316
remquo.....	317
rhypot.....	317
rint.....	318
rnorm.....	318
rnorm3d.....	319
rnorm4d.....	319
round.....	320
rsqrt.....	320
scalbln.....	321
scalbn.....	321
signbit.....	321
sin.....	322
sincos.....	322
sincospi.....	323
sinh.....	323
sinpi.....	324
sqrt.....	324
tan.....	325
tanh.....	325
tgamma.....	325
trunc.....	326

y0.....	326
y1.....	327
yn.....	327
1.7. Integer Mathematical Functions.....	328
abs.....	328
labs.....	328
llabs.....	328
llmax.....	329
llmin.....	329
max.....	329
max.....	329
max.....	330
max.....	330
max.....	330
max.....	330
max.....	330
max.....	331
max.....	331
max.....	331
max.....	331
max.....	331
min.....	332
min.....	332
min.....	332
min.....	332
min.....	333
min.....	333
min.....	333
min.....	333
min.....	333
min.....	334
min.....	334
min.....	334
ullmax.....	334
ullmin.....	334
umax.....	335
umin.....	335
1.8. Single Precision Intrinsics.....	335

__cosf.....	335
__exp10f.....	336
__expf.....	336
__fadd_rd.....	336
__fadd_rn.....	337
__fadd_ru.....	337
__fadd_rz.....	338
__fdiv_rd.....	338
__fdiv_rn.....	338
__fdiv_ru.....	339
__fdiv_rz.....	339
__fdividef.....	339
__fmaf_ieee_rd.....	340
__fmaf_ieee_rn.....	340
__fmaf_ieee_ru.....	340
__fmaf_ieee_rz.....	341
__fmaf_rd.....	341
__fmaf_rn.....	341
__fmaf_ru.....	342
__fmaf_rz.....	342
__fmul_rd.....	343
__fmul_rn.....	343
__fmul_ru.....	344
__fmul_rz.....	344
__frcp_rd.....	345
__frcp_rn.....	345
__frcp_ru.....	345
__frcp_rz.....	346
__frsqrtn_rn.....	346
__fsqrt_rd.....	347
__fsqrt_rn.....	347
__fsqrt_ru.....	347
__fsqrt_rz.....	348
__fsub_rd.....	348
__fsub_rn.....	349
__fsub_ru.....	349
__fsub_rz.....	349
__log10f.....	350

__log2f.....	350
__logf.....	351
__powf.....	351
__saturatef.....	351
__sincosf.....	352
__sinf.....	352
__tanf.....	353
1.9. Double Precision Intrinsic.....	353
__dadd_rd.....	353
__dadd_rn.....	354
__dadd_ru.....	354
__dadd_rz.....	354
__ddiv_rd.....	355
__ddiv_rn.....	355
__ddiv_ru.....	356
__ddiv_rz.....	356
__dmul_rd.....	356
__dmul_rn.....	357
__dmul_ru.....	357
__dmul_rz.....	358
__drcp_rd.....	358
__drcp_rn.....	359
__drcp_ru.....	359
__drcp_rz.....	359
__dsqrt_rd.....	360
__dsqrt_rn.....	360
__dsqrt_ru.....	361
__dsqrt_rz.....	361
__dsub_rd.....	362
__dsub_rn.....	362
__dsub_ru.....	362
__dsub_rz.....	363
__fma_rd.....	363
__fma_rn.....	364
__fma_ru.....	364
__fma_rz.....	365
1.10. Integer Intrinsic.....	366
__brev.....	366

__brevll.....	366
__byte_perm.....	366
__clz.....	367
__clzll.....	367
__dp2a_hi.....	368
__dp2a_hi.....	368
__dp2a_hi.....	368
__dp2a_hi.....	368
__dp2a_lo.....	369
__dp2a_lo.....	369
__dp2a_lo.....	369
__dp2a_lo.....	369
__dp4a.....	370
__dp4a.....	370
__dp4a.....	370
__dp4a.....	370
__ffs.....	370
__ffsll.....	371
__fns.....	371
__funnelshift_l.....	371
__funnelshift_lc.....	372
__funnelshift_r.....	372
__funnelshift_rc.....	373
__hadd.....	373
__mul24.....	373
__mul64hi.....	374
__mulhi.....	374
__popc.....	374
__popc.....	374
__rhadd.....	375
__sad.....	375
__uhadd.....	375
__umul24.....	376
__umul64hi.....	376
__umulhi.....	376
__urhadd.....	377
__usad.....	377
1.11. Type Casting Intrinsics.....	377

__double2float_rd.....	378
__double2float_rn.....	378
__double2float_ru.....	378
__double2float_rz.....	378
__double2hiint.....	379
__double2int_rd.....	379
__double2int_rn.....	379
__double2int_ru.....	380
__double2int_rz.....	380
__double2ll_rd.....	380
__double2ll_rn.....	380
__double2ll_ru.....	381
__double2ll_rz.....	381
__double2loint.....	381
__double2uint_rd.....	382
__double2uint_rn.....	382
__double2uint_ru.....	382
__double2uint_rz.....	382
__double2ull_rd.....	383
__double2ull_rn.....	383
__double2ull_ru.....	383
__double2ull_rz.....	384
__double_as_longlong.....	384
__float2int_rd.....	384
__float2int_rn.....	384
__float2int_ru.....	385
__float2int_rz.....	385
__float2ll_rd.....	385
__float2ll_rn.....	386
__float2ll_ru.....	386
__float2ll_rz.....	386
__float2uint_rd.....	386
__float2uint_rn.....	387
__float2uint_ru.....	387
__float2uint_rz.....	387
__float2ull_rd.....	388
__float2ull_rn.....	388
__float2ull_ru.....	388

__float2ull_rz.....	389
__float_as_int.....	389
__float_as_uint.....	389
__hiloint2double.....	389
__int2double_rn.....	390
__int2float_rd.....	390
__int2float_rn.....	390
__int2float_ru.....	391
__int2float_rz.....	391
__int_as_float.....	391
__ll2double_rd.....	391
__ll2double_rn.....	392
__ll2double_ru.....	392
__ll2double_rz.....	392
__ll2float_rd.....	393
__ll2float_rn.....	393
__ll2float_ru.....	393
__ll2float_rz.....	393
__longlong_as_double.....	394
__uint2double_rn.....	394
__uint2float_rd.....	394
__uint2float_rn.....	395
__uint2float_ru.....	395
__uint2float_rz.....	395
__uint_as_float.....	395
__ull2double_rd.....	396
__ull2double_rn.....	396
__ull2double_ru.....	396
__ull2double_rz.....	397
__ull2float_rd.....	397
__ull2float_rn.....	397
__ull2float_ru.....	398
__ull2float_rz.....	398
1.12. SIMD Intrinsics.....	398
__vabs2.....	398
__vabs4.....	399
__vabsdiffs2.....	399
__vabsdiffs4.....	399

__vabsdiffu2.....	400
__vabsdiffu4.....	400
__vabsss2.....	400
__vabsss4.....	401
__vadd2.....	401
__vadd4.....	401
__vaddss2.....	402
__vaddss4.....	402
__vaddus2.....	402
__vaddus4.....	403
__vavg2.....	403
__vavg4.....	403
__vavg2.....	404
__vavg4.....	404
__vcmpeq2.....	404
__vcmpeq4.....	405
__vcmpges2.....	405
__vcmpges4.....	405
__vcmpgeu2.....	406
__vcmpgeu4.....	406
__vcmpgts2.....	406
__vcmpgts4.....	407
__vcmpgtu2.....	407
__vcmpgtu4.....	407
__vcmples2.....	408
__vcmples4.....	408
__vcmpleu2.....	408
__vcmpleu4.....	409
__vcmplts2.....	409
__vcmplts4.....	409
__vcmpltu2.....	410
__vcmpltu4.....	410
__vcmpne2.....	410
__vcmpne4.....	411
__vhaddu2.....	411
__vhaddu4.....	411
__viaddmax_s16x2.....	412
__viaddmax_s16x2_relu.....	412



__viaddmax_s32.....	412
__viaddmax_s32_relu.....	413
__viaddmax_u16x2.....	413
__viaddmax_u32.....	413
__viadmin_s16x2.....	414
__viadmin_s16x2_relu.....	414
__viadmin_s32.....	414
__viadmin_s32_relu.....	415
__viadmin_u16x2.....	415
__viadmin_u32.....	415
__vibmax_s16x2.....	416
__vibmax_s32.....	416
__vibmax_u16x2.....	416
__vibmax_u32.....	417
__vibmin_s16x2.....	417
__vibmin_s32.....	418
__vibmin_u16x2.....	418
__vibmin_u32.....	418
__vimax3_s16x2.....	419
__vimax3_s16x2_relu.....	419
__vimax3_s32.....	420
__vimax3_s32_relu.....	420
__vimax3_u16x2.....	420
__vimax3_u32.....	421
__vimax_s16x2_relu.....	421
__vimax_s32_relu.....	421
__vimin3_s16x2.....	422
__vimin3_s16x2_relu.....	422
__vimin3_s32.....	422
__vimin3_s32_relu.....	423
__vimin3_u16x2.....	423
__vimin3_u32.....	423
__vimin_s16x2_relu.....	424
__vimin_s32_relu.....	424
__vmaxs2.....	424
__vmaxs4.....	425
__vmaxu2.....	425
__vmaxu4.....	425

__vmins2.....	426
__vmins4.....	426
__vminu2.....	426
__vminu4.....	427
__vneg2.....	427
__vneg4.....	427
__vnegss2.....	427
__vnegss4.....	428
__vsads2.....	428
__vsads4.....	428
__vsadu2.....	429
__vsadu4.....	429
__vseteq2.....	429
__vseteq4.....	430
__vsetges2.....	430
__vsetges4.....	430
__vsetgeu2.....	431
__vsetgeu4.....	431
__vsetgts2.....	431
__vsetgts4.....	432
__vsetgtu2.....	432
__vsetgtu4.....	432
__vsetles2.....	433
__vsetles4.....	433
__vsetleu2.....	433
__vsetleu4.....	434
__vsetlts2.....	434
__vsetlts4.....	434
__vsetltu2.....	435
__vsetltu4.....	435
__vsetne2.....	435
__vsetne4.....	436
__vsub2.....	436
__vsub4.....	436
__vsubss2.....	437
__vsubss4.....	437
__vsubus2.....	437
__vsubus4.....	438

Chapter 2. Data Structures..... 439

__half.....	440
__x.....	440
__half.....	440
__half.....	440
__half.....	440
__half.....	440
__half.....	441
__half.....	441
__half.....	441
__half.....	441
__half.....	441
__half.....	441
__half.....	442
__half.....	442
operator __half_raw.....	442
operator __half_raw.....	442
operator bool.....	442
operator char.....	442
operator float.....	443
operator int.....	443
operator long.....	443
operator long long.....	443
operator short.....	443
operator signed char.....	443
operator unsigned char.....	444
operator unsigned int.....	444
operator unsigned long.....	444
operator unsigned long long.....	444
operator unsigned short.....	444
operator=.....	445
operator=.....	445
operator=.....	445
operator=.....	445
operator=.....	445
operator=.....	446
operator=.....	446
operator=.....	446

operator=.....	446
operator=.....	446
operator=.....	446
__half2.....	447
x.....	447
y.....	447
__half2.....	447
__half2.....	447
__half2.....	447
__half2.....	447
operator __half2_raw.....	448
operator=.....	448
operator=.....	448
__half2_raw.....	448
__half_raw.....	448
__nv_bfloat16.....	449
__x.....	449
__nv_bfloat16.....	449
__nv_bfloat16.....	449
__nv_bfloat16.....	449
__nv_bfloat16.....	449
__nv_bfloat16.....	450
__nv_bfloat16.....	450
__nv_bfloat16.....	450
__nv_bfloat16.....	450
__nv_bfloat16.....	450
__nv_bfloat16.....	450
__nv_bfloat16.....	450
__nv_bfloat16.....	451
__nv_bfloat16.....	451
operator __nv_bfloat16_raw.....	451
operator __nv_bfloat16_raw.....	451
operator bool.....	451
operator char.....	451
operator float.....	452
operator int.....	452
operator long.....	452
operator long long.....	452
operator short.....	452

operator signed char.....	452
operator unsigned char.....	453
operator unsigned int.....	453
operator unsigned long.....	453
operator unsigned long long.....	453
operator unsigned short.....	453
operator=.....	454
operator=.....	454
operator=.....	454
operator=.....	454
operator=.....	454
operator=.....	454
operator=.....	455
operator=.....	455
operator=.....	455
operator=.....	455
operator=.....	455
__nv_bfloat162.....	456
x.....	456
y.....	456
__nv_bfloat162.....	456
__nv_bfloat162.....	456
__nv_bfloat162.....	456
__nv_bfloat162.....	456
operator __nv_bfloat162_raw.....	457
operator=.....	457
operator=.....	457
__nv_bfloat162_raw.....	457
__nv_bfloat16_raw.....	457
__nv_fp8_e4m3.....	458
__x.....	458
__nv_fp8_e4m3.....	458
__nv_fp8_e4m3.....	458
__nv_fp8_e4m3.....	458
__nv_fp8_e4m3.....	458
__nv_fp8_e4m3.....	459
__nv_fp8_e4m3.....	459
__nv_fp8_e4m3.....	459

__nv_fp8_e4m3.....	459
__nv_fp8_e4m3.....	459
__nv_fp8_e4m3.....	460
__nv_fp8_e4m3.....	460
__nv_fp8_e4m3.....	460
__nv_fp8_e4m3.....	460
operator __half.....	460
operator __nv_bfloat16.....	460
operator bool.....	461
operator char.....	461
operator double.....	461
operator float.....	461
operator int.....	461
operator long int.....	461
operator long long int.....	462
operator short int.....	462
operator signed char.....	462
operator unsigned char.....	462
operator unsigned int.....	462
operator unsigned long int.....	462
operator unsigned long long int.....	463
operator unsigned short int.....	463
__nv_fp8_e5m2.....	463
__x.....	463
__nv_fp8_e5m2.....	463
__nv_fp8_e5m2.....	463
__nv_fp8_e5m2.....	464
__nv_fp8_e5m2.....	464
__nv_fp8_e5m2.....	464
__nv_fp8_e5m2.....	464
__nv_fp8_e5m2.....	464
__nv_fp8_e5m2.....	464
__nv_fp8_e5m2.....	465
__nv_fp8_e5m2.....	465
__nv_fp8_e5m2.....	465
__nv_fp8_e5m2.....	465
__nv_fp8_e5m2.....	465
__nv_fp8_e5m2.....	465
operator __half.....	466

operator __nv_bfloat16.....	466
operator bool.....	466
operator char.....	466
operator double.....	466
operator float.....	466
operator int.....	467
operator long int.....	467
operator long long int.....	467
operator short int.....	467
operator signed char.....	467
operator unsigned char.....	467
operator unsigned int.....	468
operator unsigned long int.....	468
operator unsigned long long int.....	468
operator unsigned short int.....	468
__nv_fp8x2_e4m3.....	468
__x.....	468
__nv_fp8x2_e4m3.....	469
__nv_fp8x2_e4m3.....	469
__nv_fp8x2_e4m3.....	469
__nv_fp8x2_e4m3.....	469
__nv_fp8x2_e4m3.....	469
operator __half2.....	469
operator float2.....	470
__nv_fp8x2_e5m2.....	470
__x.....	470
__nv_fp8x2_e5m2.....	470
__nv_fp8x2_e5m2.....	470
__nv_fp8x2_e5m2.....	470
__nv_fp8x2_e5m2.....	471
__nv_fp8x2_e5m2.....	471
operator __half2.....	471
operator float2.....	471
__nv_fp8x4_e4m3.....	471
__x.....	471
__nv_fp8x4_e4m3.....	471
__nv_fp8x4_e4m3.....	472
__nv_fp8x4_e4m3.....	472

__nv_fp8x4_e4m3.....	472
__nv_fp8x4_e4m3.....	472
operator float4.....	472
__nv_fp8x4_e5m2.....	472
__x.....	473
__nv_fp8x4_e5m2.....	473
__nv_fp8x4_e5m2.....	473
__nv_fp8x4_e5m2.....	473
__nv_fp8x4_e5m2.....	473
__nv_fp8x4_e5m2.....	473
operator float4.....	474
<b>Chapter 3. Data Fields.....</b>	<b>475</b>



---

# Chapter 1. Modules

Here is a list of all modules:

- ▶ [FP8 Intrinsic](#)
  - ▶ [FP8 Conversion and Data Movement](#)
  - ▶ [C++ struct for handling fp8 data type of e5m2 kind.](#)
  - ▶ [C++ struct for handling vector type of two fp8 values of e5m2 kind.](#)
  - ▶ [C++ struct for handling vector type of four fp8 values of e5m2 kind.](#)
  - ▶ [C++ struct for handling fp8 data type of e4m3 kind.](#)
  - ▶ [C++ struct for handling vector type of two fp8 values of e4m3 kind.](#)
  - ▶ [C++ struct for handling vector type of four fp8 values of e4m3 kind.](#)
- ▶ [Half Precision Intrinsic](#)
  - ▶ [Half Arithmetic Constants](#)
  - ▶ [Half Arithmetic Functions](#)
  - ▶ [Half2 Arithmetic Functions](#)
  - ▶ [Half Comparison Functions](#)
  - ▶ [Half2 Comparison Functions](#)
  - ▶ [Half Precision Conversion and Data Movement](#)
  - ▶ [Half Math Functions](#)
  - ▶ [Half2 Math Functions](#)
- ▶ [Bfloat16 Precision Intrinsic](#)
  - ▶ [Bfloat16 Arithmetic Constants](#)
  - ▶ [Bfloat16 Arithmetic Functions](#)
  - ▶ [Bfloat162 Arithmetic Functions](#)
  - ▶ [Bfloat16 Comparison Functions](#)

- ▶ [Bfloat162 Comparison Functions](#)
- ▶ [Bfloat16 Precision Conversion and Data Movement](#)
- ▶ [Bfloat16 Math Functions](#)
- ▶ [Bfloat162 Math Functions](#)
- ▶ [Mathematical Functions](#)
- ▶ [Single Precision Mathematical Functions](#)
- ▶ [Double Precision Mathematical Functions](#)
- ▶ [Integer Mathematical Functions](#)
- ▶ [Single Precision Intrinsic](#)
- ▶ [Double Precision Intrinsic](#)
- ▶ [Integer Intrinsic](#)
- ▶ [Type Casting Intrinsic](#)
- ▶ [SIMD Intrinsic](#)

## 1.1. FP8 Intrinsic

This section describes fp8 intrinsic functions. To use these functions, include the header file `cuda_fp8.h` in your program. The following macros are available to help users selectively enable/disable various definitions present in the header file:

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

### FP8 Conversion and Data Movement

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

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

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

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

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

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

### 1.1.1. FP8 Conversion and Data Movement

FP8 Intrinsic

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

#### `enum __nv_fp8_interpretation_t`

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

##### Values

###### `__NV_E4M3`

Stands for fp8 numbers of e4m3 kind.

###### `__NV_E5M2`

Stands for fp8 numbers of e5m2 kind.

#### `enum __nv_saturation_t`

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

##### Values

###### `__NV_NOSAT`

Means no saturation to finite is performed when conversion results in rounding values outside the range of destination type. NOTE: for fp8 type of e4m3 kind, the results that are larger than the maximum representable finite number of the target format become NaN.

###### `__NV_SATFINITE`

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

### `typedef unsigned char __nv_fp8_storage_t`

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

### `typedef unsigned short int __nv_fp8x2_storage_t`

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

### `typedef unsigned int __nv_fp8x4_storage_t`

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

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

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

#### Returns

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

#### Description

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

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

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

#### Returns

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

## Description

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

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

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

## Returns

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

## Description

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

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

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

## Returns

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

## Description

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

```

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

```

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

### Returns

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

### Description

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

```

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

```

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

### Returns

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

### Description

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

```
__host__ device__ nv_cvt_fp8_to_halfraw (const
__nv_fp8_storage_t x, const __nv_fp8_interpretation_t
fp8_interpretation)
```

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

### Returns

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

### Description

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

```
__host__ device__ nv_cvt_fp8x2_to_halfraw2 (const
__nv_fp8x2_storage_t x, const __nv_fp8_interpretation_t
fp8_interpretation)
```

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

### Returns

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

### Description

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

```
__host__ device__ nv_fp8x2_storage_t
__nv_cvt_halfraw2_to_fp8x2 (const __half2_raw x, const
__nv_saturation_t saturate, const __nv_fp8_interpretation_t
fp8_interpretation)
```

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

### Returns

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

## Description

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

```
__host__ device __nv_fp8_storage_t
__nv_cvt_halfraw_to_fp8 (const __half_raw x, const
__nv_saturation_t saturate, const __nv_fp8_interpretation_t
fp8_interpretation)
```

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

## Returns

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

## Description

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

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

FP8 Intrinsics

```
struct __nv_fp8_e5m2
```

```
__nv_fp8_e5m2 datatype
```

```
__nv_fp8_storage_t :: __nv_fp8_e5m2 :: __x
```

Storage variable contains the  $fp8$  floating-point data.

```
__host__ device __nv_fp8_e5m2 :: __nv_fp8_e5m2 (const
long long int val)
```

## Description

Constructor from `long long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.



```
__host__ device __nv_fp8_e5m2::__nv_fp8_e5m2 (const long int val)
```

### Description

Constructor from `long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

```
__host__ device __nv_fp8_e5m2::__nv_fp8_e5m2 (const int val)
```

### Description

Constructor from `int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

```
__host__ device __nv_fp8_e5m2::__nv_fp8_e5m2 (const short int val)
```

### Description

Constructor from `short int` data type.

```
__host__ device __nv_fp8_e5m2::__nv_fp8_e5m2 (const unsigned long long int val)
```

### Description

Constructor from `unsigned long long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

```
__host__ device __nv_fp8_e5m2::__nv_fp8_e5m2 (const unsigned long int val)
```

### Description

Constructor from `unsigned long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

```
__host__ device __nv_fp8_e5m2::__nv_fp8_e5m2 (const
unsigned int val)
```

### Description

Constructor from `unsigned int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

```
__host__ device __nv_fp8_e5m2::__nv_fp8_e5m2 (const
unsigned short int val)
```

### Description

Constructor from `unsigned short int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

```
__host__ device __nv_fp8_e5m2::__nv_fp8_e5m2 (const
double f)
```

### Description

Constructor from `double` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

```
__host__ device __nv_fp8_e5m2::__nv_fp8_e5m2 (const
float f)
```

### Description

Constructor from `float` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

```
__host__ device __nv_fp8_e5m2::__nv_fp8_e5m2 (const
__nv_bfloat16 f)
```

### Description

Constructor from `__nv_bfloat16` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

```
__host__ device __nv_fp8_e5m2::__nv_fp8_e5m2 (const
__half f)
```

### Description

Constructor from `__half` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

```
__nv_fp8_e5m2::__nv_fp8_e5m2 ()
```

### Description

Constructor by default.

```
__host__ device __nv_fp8_e5m2::operator __half ()
```

### Description

Conversion operator to `__half` data type.

```
__host__ device __nv_fp8_e5m2::operator __nv_bfloat16
()
```

### Description

Conversion operator to `__nv_bfloat16` data type.

```
__host__ device __nv_fp8_e5m2::operator bool ()
```

### Description

Conversion operator to `bool` data type. `+0` and `-0` inputs convert to `false`. Non-zero inputs convert to `true`.

```
__host__ device __nv_fp8_e5m2::operator char ()
```

### Description

Conversion operator to an implementation defined `char` data type.

Detects signedness of the `char` type and proceeds accordingly, see further details in signed and unsigned char operators.

Clamps inputs to the output range. NaN inputs convert to `zero`.

```
__host__ device__ nv_fp8_e5m2::operator double ()
```

### Description

Conversion operator to `double` data type.

```
__host__ device__ nv_fp8_e5m2::operator float ()
```

### Description

Conversion operator to `float` data type.

```
__host__ device__ nv_fp8_e5m2::operator int ()
```

### Description

Conversion operator to `int` data type. Clamps too large inputs to the output range. NaN inputs convert to zero.

```
__host__ device__ nv_fp8_e5m2::operator long int ()
```

### Description

Conversion operator to `long int` data type. Clamps too large inputs to the output range. NaN inputs convert to zero if output type is 32-bit. NaN inputs convert to `0x8000000000000000ULL` if output type is 64-bit.

```
__host__ device__ nv_fp8_e5m2::operator long long int  
()
```

### Description

Conversion operator to `long long int` data type. Clamps too large inputs to the output range. NaN inputs convert to `0x8000000000000000LL`.

```
__host__ device__ nv_fp8_e5m2::operator short int ()
```

### Description

Conversion operator to `short int` data type. Clamps too large inputs to the output range. NaN inputs convert to zero.

`__host__ __device__ nv_fp8_e5m2::operator signed char ()`

### Description

Conversion operator to `signed char` data type. Clamps too large inputs to the output range. NaN inputs convert to zero.

`__host__ __device__ nv_fp8_e5m2::operator unsigned char ()`

### Description

Conversion operator to `unsigned char` data type. Clamps negative and too large inputs to the output range. NaN inputs convert to zero.

`__host__ __device__ nv_fp8_e5m2::operator unsigned int ()`

### Description

Conversion operator to `unsigned int` data type. Clamps negative and too large inputs to the output range. NaN inputs convert to zero.

`__host__ __device__ nv_fp8_e5m2::operator unsigned long int ()`

### Description

Conversion operator to `unsigned long int` data type. Clamps negative and too large inputs to the output range. NaN inputs convert to zero if output type is 32-bit. NaN inputs convert to `0x8000000000000000ULL` if output type is 64-bit.

`__host__ __device__ nv_fp8_e5m2::operator unsigned long long int ()`

### Description

Conversion operator to `unsigned long long int` data type. Clamps negative and too large inputs to the output range. NaN inputs convert to `0x8000000000000000ULL`.

```
__host__ device __nv_fp8_e5m2::operator unsigned
short int ()
```

### Description

Conversion operator to unsigned `short int` data type. Clamps negative and too large inputs to the output range. NaN inputs convert to zero.

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

FP8 Ininsics

```
struct __nv_fp8x2_e5m2
```

```
__nv_fp8x2_e5m2 datatype
```

```
__nv_fp8x2_storage_t :: __nv_fp8x2_e5m2::__x
```

Storage variable contains the vector of two `fp8` floating-point data values.

```
__host__ device __nv_fp8x2_e5m2::__nv_fp8x2_e5m2
(const double2 f)
```

### Description

Constructor from `double2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

```
__host__ device __nv_fp8x2_e5m2::__nv_fp8x2_e5m2
(const float2 f)
```

### Description

Constructor from `float2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

```
__host__ device__ nv_fp8x2_e5m2::__nv_fp8x2_e5m2
(const __nv_bfloat162 f)
```

### Description

Constructor from `__nv_bfloat162` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

```
__host__ device__ nv_fp8x2_e5m2::__nv_fp8x2_e5m2
(const __half2 f)
```

### Description

Constructor from `__half2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

```
__nv_fp8x2_e5m2::__nv_fp8x2_e5m2 ()
```

### Description

Constructor by default.

```
__host__ device__ nv_fp8x2_e5m2::operator __half2 ()
```

### Description

Conversion operator to `__half2` data type.

```
__host__ device__ nv_fp8x2_e5m2::operator float2 ()
```

### Description

Conversion operator to `float2` data type.

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

FP8 Intrinsics

```
struct __nv_fp8x4_e5m2
```

```
__nv_fp8x4_e5m2 datatype
```

```
__nv_fp8x4_storage_t :: __nv_fp8x4_e5m2 :: __x
```

Storage variable contains the vector of four fp8 floating-point data values.

```
__host__ device __nv_fp8x4_e5m2 :: __nv_fp8x4_e5m2  
(const double4 f)
```

### Description

Constructor from `double4` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

```
__host__ device __nv_fp8x4_e5m2 :: __nv_fp8x4_e5m2  
(const float4 f)
```

### Description

Constructor from `float4` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

```
__host__ device __nv_fp8x4_e5m2 :: __nv_fp8x4_e5m2  
(const __nv_bfloat162 flo, const __nv_bfloat162 fhi)
```

### Description

Constructor from a pair of `__nv_bfloat162` data type values, relies on `__NV_SATFINITE` behavior for out-of-range values.

```
__host__ device __nv_fp8x4_e5m2 :: __nv_fp8x4_e5m2  
(const __half2 flo, const __half2 fhi)
```

### Description

Constructor from a pair of `__half2` data type values, relies on `__NV_SATFINITE` behavior for out-of-range values.



```
__nv_fp8x4_e5m2::__nv_fp8x4_e5m2 ()
```

### Description

Constructor by default.

```
__host__ __device__ __nv_fp8x4_e5m2::operator float4 ()
```

### Description

Conversion operator to `float4` vector data type.

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

FP8 Intrinsics

```
struct __nv_fp8_e4m3
```

`__nv_fp8_e4m3` datatype

```
__nv_fp8_storage_t::__nv_fp8_e4m3::__x
```

Storage variable contains the `fp8` floating-point data.

```
__host__ __device__ __nv_fp8_e4m3::__nv_fp8_e4m3 (const
long long int val)
```

### Description

Constructor from `long long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

```
__host__ __device__ __nv_fp8_e4m3::__nv_fp8_e4m3 (const
long int val)
```

### Description

Constructor from `long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

```
__host__ device __nv_fp8_e4m3::__nv_fp8_e4m3 (const int val)
```

### Description

Constructor from `int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

```
__host__ device __nv_fp8_e4m3::__nv_fp8_e4m3 (const short int val)
```

### Description

Constructor from `short int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

```
__host__ device __nv_fp8_e4m3::__nv_fp8_e4m3 (const unsigned long long int val)
```

### Description

Constructor from `unsigned long long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

```
__host__ device __nv_fp8_e4m3::__nv_fp8_e4m3 (const unsigned long int val)
```

### Description

Constructor from `unsigned long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

```
__host__ device __nv_fp8_e4m3::__nv_fp8_e4m3 (const unsigned int val)
```

### Description

Constructor from `unsigned int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

```
__host__ device __nv_fp8_e4m3::__nv_fp8_e4m3 (const
unsigned short int val)
```

### Description

Constructor from `unsigned short int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

```
__host__ device __nv_fp8_e4m3::__nv_fp8_e4m3 (const
double f)
```

### Description

Constructor from `double` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

```
__host__ device __nv_fp8_e4m3::__nv_fp8_e4m3 (const
float f)
```

### Description

Constructor from `float` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

```
__host__ device __nv_fp8_e4m3::__nv_fp8_e4m3 (const
__nv_bfloat16 f)
```

### Description

Constructor from `__nv_bfloat16` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

```
__host__ device __nv_fp8_e4m3::__nv_fp8_e4m3 (const
__half f)
```

### Description

Constructor from `__half` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

```
__nv_fp8_e4m3::__nv_fp8_e4m3 ()
```

### Description

Constructor by default.

```
__host__ device __nv_fp8_e4m3::operator __half ()
```

### Description

Conversion operator to `__half` data type.

```
__host__ device __nv_fp8_e4m3::operator __nv_bfloat16  
()
```

### Description

Conversion operator to `__nv_bfloat16` data type.

```
__host__ device __nv_fp8_e4m3::operator bool ()
```

### Description

Conversion operator to `bool` data type. +0 and -0 inputs convert to `false`. Non-zero inputs convert to `true`.

```
__host__ device __nv_fp8_e4m3::operator char ()
```

### Description

Conversion operator to an implementation defined `char` data type.

Detects signedness of the `char` type and proceeds accordingly, see further details in signed and unsigned char operators.

Clamps inputs to the output range. NaN inputs convert to zero.

```
__host__ device __nv_fp8_e4m3::operator double ()
```

### Description

Conversion operator to `double` data type.

```
__host__ __device__ __nv_fp8_e4m3::operator float ()
```

### Description

Conversion operator to `float` data type.

```
__host__ __device__ __nv_fp8_e4m3::operator int ()
```

### Description

Conversion operator to `int` data type. NaN inputs convert to zero.

```
__host__ __device__ __nv_fp8_e4m3::operator long int ()
```

### Description

Conversion operator to `long int` data type. Clamps too large inputs to the output range. NaN inputs convert to zero if output type is 32-bit. NaN inputs convert to `0x8000000000000000ULL` if output type is 64-bit.

```
__host__ __device__ __nv_fp8_e4m3::operator long long int  
()
```

### Description

Conversion operator to `long long int` data type. NaN inputs convert to `0x8000000000000000LL`.

```
__host__ __device__ __nv_fp8_e4m3::operator short int ()
```

### Description

Conversion operator to `short int` data type. NaN inputs convert to zero.

```
__host__ __device__ __nv_fp8_e4m3::operator signed char ()
```

### Description

Conversion operator to `signed char` data type. Clamps too large inputs to the output range. NaN inputs convert to zero.

`__host__ __device__ nv_fp8_e4m3::operator unsigned char ()`

#### Description

Conversion operator to unsigned char data type. Clamps negative and too large inputs to the output range. NaN inputs convert to zero.

`__host__ __device__ nv_fp8_e4m3::operator unsigned int ()`

#### Description

Conversion operator to unsigned int data type. Clamps negative inputs to zero. NaN inputs convert to zero.

`__host__ __device__ nv_fp8_e4m3::operator unsigned long int ()`

#### Description

Conversion operator to unsigned long int data type. Clamps negative and too large inputs to the output range. NaN inputs convert to zero if output type is 32-bit. NaN inputs convert to 0x8000000000000000ULL if output type is 64-bit.

`__host__ __device__ nv_fp8_e4m3::operator unsigned long long int ()`

#### Description

Conversion operator to unsigned long long int data type. Clamps negative inputs to zero. NaN inputs convert to 0x8000000000000000ULL.

`__host__ __device__ nv_fp8_e4m3::operator unsigned short int ()`

#### Description

Conversion operator to unsigned short int data type. Clamps negative inputs to zero. NaN inputs convert to zero.

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

FP8 Intrinsic

```
struct __nv_fp8x2_e4m3
```

`__nv_fp8x2_e4m3` datatype

```
__nv_fp8x2_storage_t :: __nv_fp8x2_e4m3 :: __x
```

Storage variable contains the vector of two fp8 floating-point data values.

```
__host__ device __nv_fp8x2_e4m3 :: __nv_fp8x2_e4m3  
(const double2 f)
```

### Description

Constructor from `double2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

```
__host__ device __nv_fp8x2_e4m3 :: __nv_fp8x2_e4m3  
(const float2 f)
```

### Description

Constructor from `float2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

```
__host__ device __nv_fp8x2_e4m3 :: __nv_fp8x2_e4m3  
(const __nv_bfloat162 f)
```

### Description

Constructor from `__nv_bfloat162` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

```
__host__ device __nv_fp8x2_e4m3::__nv_fp8x2_e4m3
(const __half2 f)
```

### Description

Constructor from `__half2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

```
__nv_fp8x2_e4m3::__nv_fp8x2_e4m3 ()
```

### Description

Constructor by default.

```
__host__ device __nv_fp8x2_e4m3::operator __half2 ()
```

### Description

Conversion operator to `__half2` data type.

```
__host__ device __nv_fp8x2_e4m3::operator float2 ()
```

### Description

Conversion operator to `float2` data type.

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

FP8 Intrinsics

```
struct __nv_fp8x4_e4m3
```

`__nv_fp8x4_e4m3` datatype

```
__nv_fp8x4_storage_t::__nv_fp8x4_e4m3::__x
```

Storage variable contains the vector of four `fp8` floating-point data values.



```
__host__ device __nv_fp8x4_e4m3::__nv_fp8x4_e4m3
(const double4 f)
```

### Description

Constructor from `double4` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

```
__host__ device __nv_fp8x4_e4m3::__nv_fp8x4_e4m3
(const float4 f)
```

### Description

Constructor from `float4` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

```
__host__ device __nv_fp8x4_e4m3::__nv_fp8x4_e4m3
(const __nv_bfloat162 flo, const __nv_bfloat162 fhi)
```

### Description

Constructor from a pair of `__nv_bfloat162` data type values, relies on `__NV_SATFINITE` behavior for out-of-range values.

```
__host__ device __nv_fp8x4_e4m3::__nv_fp8x4_e4m3
(const __half2 flo, const __half2 fhi)
```

### Description

Constructor from a pair of `__half2` data type values, relies on `__NV_SATFINITE` behavior for out-of-range values.

```
__nv_fp8x4_e4m3::__nv_fp8x4_e4m3 ()
```

### Description

Constructor by default.

```
__host__ device __nv_fp8x4_e4m3::operator float4 ()
```

### Description

Conversion operator to `float4` vector data type.

## 1.2. Half Precision Intrinsics

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

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

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

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

### `struct __half`

`__half` data type

### `struct __half2`

`__half2` data type

### `struct __half2_raw`

`__half2_raw` data type

### `struct __half_raw`

`__half_raw` data type

## Half Arithmetic Constants

Half Arithmetic Functions

Half2 Arithmetic Functions

Half Comparison Functions

Half2 Comparison Functions

Half Precision Conversion and Data  
Movement

Half Math Functions

Half2 Math Functions

**typedef \_\_nv\_half**

This datatype is an `__nv_` prefixed alias.

**typedef \_\_nv\_half2**

This datatype is an `__nv_` prefixed alias.

**typedef \_\_nv\_half2\_raw**

This datatype is an `__nv_` prefixed alias.

**typedef \_\_nv\_half\_raw**

This datatype is an `__nv_` prefixed alias.

**typedef half**

This datatype is meant to be the first-class or fundamental implementation of the half-precision numbers format.

Should be implemented in the compiler in the future. Current implementation is a simple typedef to a respective user-level type with underscores.

## typedef half2

This datatype is meant to be the first-class or fundamental implementation of type for pairs of half-precision numbers.

Should be implemented in the compiler in the future. Current implementation is a simple typedef to a respective user-level type with underscores.

## typedef nv\_half

This datatype is an `nv_` prefixed alias.

## typedef nv\_half2

This datatype is an `nv_` prefixed alias.

### 1.2.1. Half Arithmetic Constants

Half Precision Intrinsics

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

```
#define CUDART_INF_FP16 __ushort_as_half((unsigned short)0x7C00U)
```

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

```
#define CUDART_MAX_NORMAL_FP16  
__ushort_as_half((unsigned short)0x7BFFU)
```

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

```
#define CUDART_MIN_DENORM_FP16  
__ushort_as_half((unsigned short)0x0001U)
```

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

```
#define CUDART_NAN_FP16 __ushort_as_half((unsigned short)0x7FFFU)
```

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

```
#define CUDART_NEG_ZERO_FP16  
__ushort_as_half((unsigned short)0x8000U)
```

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

```
#define CUDART_ONE_FP16 __ushort_as_half((unsigned short)0x3C00U)
```

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

```
#define CUDART_ZERO_FP16 __ushort_as_half((unsigned short)0x0000U)
```

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

## 1.2.2. Half Arithmetic Functions

Half Precision Intrinsic

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

```
__host__ __device__ __half habs (const __half a)
```

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

### Parameters

**a**

- `half`. Is only being read.

### Returns

`half`

► The absolute value of `a`.

### Description

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

```
__host__ __device__ __half hadd (const __half a, const __half b)
```

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

### Description

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

`__host__ __device__ hadd_rn (const __half a, const __half b)`

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

### Description

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

`__host__ __device__ hadd_sat (const __half a, const __half b)`

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

### Parameters

**a**

- `half`. Is only being read.

**b**

- `half`. Is only being read.

### Returns

`half`

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

### Description

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

`__host__ __device__ hdiv (const __half a, const __half b)`

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

### Description

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

`__device__ hfma (const __half a, const __half b, const __half c)`

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

### Description

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

`__device__ hfma_relu (const __half a, const __half b, const __half c)`

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

### Parameters

**a**

- `half`. Is only being read.

**b**

- `half`. Is only being read.

**c**

- `half`. Is only being read.

### Returns

`half`

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

### Description

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

`__device__ hfma_sat (const __half a, const __half b, const __half c)`

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

### Parameters

**a**

- `half`. Is only being read.

**b**

- `half`. Is only being read.

**c**

- half. Is only being read.

## Returns

half

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

## Description

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

`__host__ __device__ hmul (const __half a, const __half b)`

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

## Description

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

`__host__ __device__ hmul_rn (const __half a, const __half b)`

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

## Description

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

`__host__ __device__ hmul_sat (const __half a, const __half b)`

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

## Parameters

**a**

- half. Is only being read.

**b**

- half. Is only being read.

## Returns

half

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



## Description

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

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

Negates input `half` number and returns the result.

## Description

Negates input `half` number and returns the result.

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

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

## Description

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

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

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

## Description

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

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

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

## Parameters

**a**

- `half`. Is only being read.

**b**

- `half`. Is only being read.

## Returns

`half`

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

## Description

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

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

Adds `val` to the value stored at `address` in global or shared memory, and writes this value back to `address`. This operation is performed in one atomic operation.

## Parameters

### **address**

- `half*`. An address in global or shared memory.

### **val**

- `half`. The value to be added.

## Returns

`half`

- The old value read from `address`.

## Description

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



### **Note:**

For more details for this function see the Atomic Functions section in the CUDA C++ Programming Guide.

## `__host__ __device__ operator* (const __half lh, const __half rh)`

## Description

Performs `half` multiplication operation. See also [\\_\\_hmul\(\\_\\_half, \\_\\_half\)](#)

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

#### Description

Performs `half` compound assignment with multiplication operation.

`__host__ __device__ operator+ (const __half h)`

#### Description

Implements `half` unary plus operator, returns input value.

`__host__ __device__ operator+ (const __half lh, const __half rh)`

#### Description

Performs `half` addition operation. See also `__hadd(__half, __half)`

`__host__ __device__ operator++ (__half h, const int ignored)`

#### Description

Performs `half` postfix increment operation.

`__host__ __device__ operator++ (__half h)`

#### Description

Performs `half` prefix increment operation.

`__host__ __device__ operator+= (__half lh, const __half rh)`

#### Description

Performs `half` compound assignment with addition operation.

`__host__ __device__ operator- (const __half h)`

#### Description

Implements `half` unary minus operator. See also `__hneg(__half)`

`__host__ __device__ operator- (const __half lh, const __half rh)`

#### Description

Performs `half` subtraction operation. See also `__hsub(__half, __half)`

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

#### Description

Performs `half` postfix decrement operation.

`__host__ __device__ operator-- (__half h)`

#### Description

Performs `half` prefix decrement operation.

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

#### Description

Performs `half` compound assignment with subtraction operation.

`__host__ __device__ operator/ (const __half lh, const __half rh)`

#### Description

Performs `half` division operation. See also `__hdiv(__half, __half)`

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

#### Description

Performs `half` compound assignment with division operation.

### 1.2.3. Half2 Arithmetic Functions

Half Precision Ininsics

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

`__host__ __device__ h2div (const __half2 a, const __half2 b)`

Performs `half2` vector division in round-to-nearest-even mode.

### Description

Divides `half2` input vector `a` by input vector `b` in round-to-nearest-even mode.

`__host__ __device__ habs2 (const __half2 a)`

Calculates the absolute value of both halves of the input `half2` number and returns the result.

### Parameters

**a**

- `half2`. Is only being read.

### Returns

`half2`

► Returns `a` with the absolute value of both halves.

### Description

Calculates the absolute value of both halves of the input `half2` number and returns the result.

`__host__ __device__ hadd2 (const __half2 a, const __half2 b)`

Performs `half2` vector addition in round-to-nearest-even mode.

### Description

Performs `half2` vector add of inputs `a` and `b`, in round-to-nearest-even mode.

`__host__ __device__ hadd2_rn (const __half2 a, const __half2 b)`

Performs `half2` vector addition in round-to-nearest-even mode.

### Description

Performs `half2` vector add of inputs `a` and `b`, in round-to-nearest-even mode. Prevents floating-point contractions of `mul+add` into `fma`.

`__host__ __device__ hadd2_sat (const __half2 a, const __half2 b)`

Performs `half2` vector addition in round-to-nearest-even mode, with saturation to `[0.0, 1.0]`.

### Parameters

**a**

- `half2`. Is only being read.

**b**

- `half2`. Is only being read.

### Returns

`half2`

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

### Description

Performs `half2` vector add of inputs `a` and `b`, in round-to-nearest-even mode, and clamps the results to range `[0.0, 1.0]`. NaN results are flushed to `+0.0`.

`__device__ hcmadd (const __half2 a, const __half2 b, const __half2 c)`

Performs fast complex multiply-accumulate.

### Parameters

**a**

- `half2`. Is only being read.

**b**

- `half2`. Is only being read.

**c**

- `half2`. Is only being read.

### Returns

`half2`

- The result of complex multiply-accumulate operation on complex numbers `a`, `b`, and `c`

### Description

Interprets vector `half2` input pairs `a`, `b`, and `c` as complex numbers in `half` precision and performs complex multiply-accumulate operation:  $a*b + c$

`__device__ hfma2 (const __half2 a, const __half2 b, const __half2 c)`

Performs `half2` vector fused multiply-add in round-to-nearest-even mode.

### Description

Performs `half2` vector multiply on inputs `a` and `b`, then performs a `half2` vector add of the result with `c`, rounding the result once in round-to-nearest-even mode.

`__device__ hfma2_relu (const __half2 a, const __half2 b, const __half2 c)`

Performs `half2` vector fused multiply-add in round-to-nearest-even mode with relu saturation.

### Parameters

- a**  
- `half2`. Is only being read.
- b**  
- `half2`. Is only being read.
- c**  
- `half2`. Is only being read.

### Returns

`half2`

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

### Description

Performs `half2` vector multiply on inputs `a` and `b`, then performs a `half2` vector add of the result with `c`, rounding the result once in round-to-nearest-even mode. Then negative result is clamped to 0. NaN result is converted to canonical NaN.

`__device__ hfma2_sat (const __half2 a, const __half2 b, const __half2 c)`

Performs `half2` vector fused multiply-add in round-to-nearest-even mode, with saturation to `[0.0, 1.0]`.

### Parameters

- a**  
- `half2`. Is only being read.

**b**

- half2. Is only being read.

**c**

- half2. Is only being read.

## Returns

half2

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

## Description

Performs `half2` vector multiply on inputs a and b, then performs a `half2` vector add of the result with c, rounding the result once in round-to-nearest-even mode, and clamps the results to range [0.0, 1.0]. NaN results are flushed to +0.0.

`__host__ __device__ half2 hmul2 (const half2 a, const half2 b)`

Performs `half2` vector multiplication in round-to-nearest-even mode.

## Description

Performs `half2` vector multiplication of inputs a and b, in round-to-nearest-even mode.

`__host__ __device__ half2 hmul2_rn (const half2 a, const half2 b)`

Performs `half2` vector multiplication in round-to-nearest-even mode.

## Description

Performs `half2` vector multiplication of inputs a and b, in round-to-nearest-even mode. Prevents floating-point contractions of mul+add or sub into fma.

`__host__ __device__ half2 hmul2_sat (const half2 a, const half2 b)`

Performs `half2` vector multiplication in round-to-nearest-even mode, with saturation to [0.0, 1.0].

## Parameters

**a**

- half2. Is only being read.



**b**

- half2. Is only being read.

**Returns**

half2

- ▶ The result of elementwise multiplication of vectors a and b, with respect to saturation.

**Description**

Performs `half2` vector multiplication of inputs a and b, in round-to-nearest-even mode, and clamps the results to range [0.0, 1.0]. NaN results are flushed to +0.0.

`__host__ __device__ hneg2 (const __half2 a)`

Negates both halves of the input `half2` number and returns the result.

**Description**

Negates both halves of the input `half2` number a and returns the result.

`__host__ __device__ hsub2 (const __half2 a, const __half2 b)`

Performs `half2` vector subtraction in round-to-nearest-even mode.

**Description**

Subtracts `half2` input vector b from input vector a in round-to-nearest-even mode.

`__host__ __device__ hsub2_rn (const __half2 a, const __half2 b)`

Performs `half2` vector subtraction in round-to-nearest-even mode.

**Description**

Subtracts `half2` input vector b from input vector a in round-to-nearest-even mode. Prevents floating-point contractions of mul+sub into fma.

## `__host__ __device__ hsub2_sat (const __half2 a, const __half2 b)`

Performs `half2` vector subtraction in round-to-nearest-even mode, with saturation to [0.0, 1.0].

### Parameters

#### **a**

- `half2`. Is only being read.

#### **b**

- `half2`. Is only being read.

### Returns

`half2`

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

### Description

Subtracts `half2` input vector `b` from input vector `a` in round-to-nearest-even mode, and clamps the results to range [0.0, 1.0]. NaN results are flushed to +0.0.

## `__device__ atomicAdd (const __half2 *address, const __half2 val)`

Vector add `val` to the value stored at `address` in global or shared memory, and writes this value back to `address`. The atomicity of the add operation is guaranteed separately for each of the two `__half` elements; the entire `__half2` is not guaranteed to be atomic as a single 32-bit access.

### Parameters

#### **address**

- `half2*`. An address in global or shared memory.

#### **val**

- `half2`. The value to be added.

### Returns

`half2`

- The old value read from `address`.

## Description

The location of `address` must be in global or shared memory. This operation has undefined behavior otherwise. This operation is natively supported by devices of compute capability 6.x and higher, older devices use emulation path.



### Note:

For more details for this function see the Atomic Functions section in the CUDA C++ Programming Guide.

```
__host__ __device__ operator* (const __half2 lh, const
__half2 rh)
```

## Description

Performs packed `half` multiplication operation. See also [\\_\\_hmul2\(\\_\\_half2, \\_\\_half2\)](#)

```
__host__ __device__ operator* = (__half2 lh, const __half2 rh)
```

## Description

Performs packed `half` compound assignment with multiplication operation.

```
__host__ __device__ operator+ (const __half2 h)
```

## Description

Implements packed `half` unary plus operator, returns input value.

```
__host__ __device__ operator+ (const __half2 lh, const
__half2 rh)
```

## Description

Performs packed `half` addition operation. See also [\\_\\_hadd2\(\\_\\_half2, \\_\\_half2\)](#)

```
__host__ __device__ operator++ (__half2 h, const int ignored)
```

## Description

Performs packed `half` postfix increment operation.

`__host__ device __operator++ (__half2 h)`

#### Description

Performs packed `half` prefix increment operation.

`__host__ device __operator+= (__half2 lh, const __half2 rh)`

#### Description

Performs packed `half` compound assignment with addition operation.

`__host__ device __operator- (const __half2 h)`

#### Description

Implements packed `half` unary minus operator. See also `__hneg2(__half2)`

`__host__ device __operator- (const __half2 lh, const __half2 rh)`

#### Description

Performs packed `half` subtraction operation. See also `__hsub2(__half2, __half2)`

`__host__ device __operator-- (__half2 h, const int ignored)`

#### Description

Performs packed `half` postfix decrement operation.

`__host__ device __operator-- (__half2 h)`

#### Description

Performs packed `half` prefix decrement operation.

`__host__ device __operator-= (__half2 lh, const __half2 rh)`

#### Description

Performs packed `half` compound assignment with subtraction operation.

`__host__ __device__ operator/ (const __half2 lh, const __half2 rh)`

### Description

Performs packed `half` division operation. See also `h2div(__half2, __half2)`

`__host__ __device__ operator/= (__half2 lh, const __half2 rh)`

### Description

Performs packed `half` compound assignment with division operation.

## 1.2.4. Half Comparison Functions

Half Precision Ininsics

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

`__host__ __device__ bool __heq (const __half a, const __half b)`

Performs `half` if-equal comparison.

### Parameters

**a**

- `half`. Is only being read.

**b**

- `half`. Is only being read.

### Returns

`bool`

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

### Description

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

```
__host__ __device__ bool __hequ (const __half a, const
__half b)
```

Performs `half` unordered if-equal comparison.

### Parameters

**a**

- `half`. Is only being read.

**b**

- `half`. Is only being read.

### Returns

`bool`

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

### Description

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

```
__host__ __device__ bool __hge (const __half a, const __half
b)
```

Performs `half` greater-equal comparison.

### Parameters

**a**

- `half`. Is only being read.

**b**

- `half`. Is only being read.

### Returns

`bool`

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

### Description

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

```
__host__ device__ bool __hgeu (const __half a, const
__half b)
```

Performs `half` unordered greater-equal comparison.

### Parameters

**a**

- `half`. Is only being read.

**b**

- `half`. Is only being read.

### Returns

`bool`

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

### Description

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

```
__host__ device__ bool __hgt (const __half a, const __half
b)
```

Performs `half` greater-than comparison.

### Parameters

**a**

- `half`. Is only being read.

**b**

- `half`. Is only being read.

### Returns

`bool`

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

### Description

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

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

Performs `half` unordered greater-than comparison.

### Parameters

**a**

- `half`. Is only being read.

**b**

- `half`. Is only being read.

### Returns

`bool`

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

### Description

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

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

Checks if the input `half` number is infinite.

### Parameters

**a**

- `half`. Is only being read.

### Returns

`int`

- ▶ -1 iff `a` is equal to negative infinity,
- ▶ 1 iff `a` is equal to positive infinity,
- ▶ 0 otherwise.

### Description

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



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

Determine whether `half` argument is a NaN.

### Parameters

**a**

- `half`. Is only being read.

### Returns

`bool`

► true iff argument is NaN.

### Description

Determine whether `half` value `a` is a NaN.

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

Performs `half` less-equal comparison.

### Parameters

**a**

- `half`. Is only being read.

**b**

- `half`. Is only being read.

### Returns

`bool`

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

### Description

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

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

Performs `half` unordered less-equal comparison.

### Parameters

**a**

- `half`. Is only being read.

**b**

- half. Is only being read.

## Returns

bool

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

## Description

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

```
__host____device__ bool __hlt (const __half a, const __half b)
```

Performs `half` less-than comparison.

## Parameters

**a**

- half. Is only being read.

**b**

- half. Is only being read.

## Returns

bool

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

## Description

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

```
__host____device__ bool __hltu (const __half a, const __half b)
```

Performs `half` unordered less-than comparison.

## Parameters

**a**

- half. Is only being read.

**b**

- half. Is only being read.

## Returns

bool

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

### Description

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

`__host__ __device__ hmax (const __half a, const __half b)`

Calculates `half` maximum of two input values.

### Description

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

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

`__host__ __device__ hmax_nan (const __half a, const __half b)`

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

### Description

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

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

`__host__ __device__ hmin (const __half a, const __half b)`

Calculates `half` minimum of two input values.

### Description

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

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

```
__host__ __device__ hmin_nan (const __half a, const
__half b)
```

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

### Description

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

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

```
__host__ __device__ bool hne (const __half a, const __half
b)
```

Performs `half` not-equal comparison.

### Parameters

**a**

- half. Is only being read.

**b**

- half. Is only being read.

### Returns

bool

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

### Description

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

```
__host__ __device__ bool hneu (const __half a, const
__half b)
```

Performs `half` unordered not-equal comparison.

### Parameters

**a**

- half. Is only being read.

**b**

- half. Is only being read.

## Returns

bool

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

## Description

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

```
__host__ __device__ __CUDA_FP16_FORCEINLINE__ bool
operator!= (const __half lh, const __half rh)
```

## Description

Performs `half` unordered compare not-equal operation. See also [\\_\\_hneu\\_\\_\(\\_\\_half, \\_\\_half\)](#)

```
__host__ __device__ __CUDA_FP16_FORCEINLINE__ bool
operator< (const __half lh, const __half rh)
```

## Description

Performs `half` ordered less-than compare operation. See also [\\_\\_hlt\\_\\_\(\\_\\_half, \\_\\_half\)](#)

```
__host__ __device__ __CUDA_FP16_FORCEINLINE__ bool
operator<= (const __half lh, const __half rh)
```

## Description

Performs `half` ordered less-or-equal compare operation. See also [\\_\\_hle\\_\\_\(\\_\\_half, \\_\\_half\)](#)

```
__host__ __device__ __CUDA_FP16_FORCEINLINE__ bool
operator== (const __half lh, const __half rh)
```

## Description

Performs `half` ordered compare equal operation. See also [\\_\\_heq\\_\\_\(\\_\\_half, \\_\\_half\)](#)

```
__host__ __device__ __CUDA_FP16_FORCEINLINE__ bool
operator> (const __half lh, const __half rh)
```

### Description

Performs `half` ordered greater-than compare operation. See also `__hgt(__half, __half)`

```
__host__ __device__ __CUDA_FP16_FORCEINLINE__ bool
operator>= (const __half lh, const __half rh)
```

### Description

Performs `half` ordered greater-or-equal compare operation. See also `__hge(__half, __half)`

## 1.2.5. Half2 Comparison Functions

Half Precision Ininsics

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

```
__host__ __device__ bool __hbeq2 (const __half2 a, const
__half2 b)
```

Performs `half2` vector if-equal comparison and returns boolean true iff both `half` results are true, boolean false otherwise.

### Parameters

- a**  
- `half2`. Is only being read.
- b**  
- `half2`. Is only being read.

### Returns

`bool`

- ▶ true if both `half` results of if-equal comparison of vectors `a` and `b` are true;
- ▶ false otherwise.

### Description

Performs `half2` vector if-equal comparison of inputs `a` and `b`. The `bool` result is set to true only if both `half` if-equal comparisons evaluate to true, or false otherwise. NaN inputs generate false results.

`__host__ __device__ bool __hbequ2 (const __half2 a, const __half2 b)`

Performs `half2` vector unordered if-equal comparison and returns boolean true iff both `half` results are true, boolean false otherwise.

### Parameters

**a**

- `half2`. Is only being read.

**b**

- `half2`. Is only being read.

### Returns

bool

- ▶ true if both `half` results of unordered if-equal comparison of vectors `a` and `b` are true;
- ▶ false otherwise.

### Description

Performs `half2` vector if-equal comparison of inputs `a` and `b`. The bool result is set to true only if both `half` if-equal comparisons evaluate to true, or false otherwise. NaN inputs generate true results.

`__host__ __device__ bool __hbge2 (const __half2 a, const __half2 b)`

Performs `half2` vector greater-equal comparison and returns boolean true iff both `half` results are true, boolean false otherwise.

### Parameters

**a**

- `half2`. Is only being read.

**b**

- `half2`. Is only being read.

### Returns

bool

- ▶ true if both `half` results of greater-equal comparison of vectors `a` and `b` are true;
- ▶ false otherwise.

## Description

Performs `half2` vector greater-equal comparison of inputs `a` and `b`. The `bool` result is set to true only if both `half` greater-equal comparisons evaluate to true, or false otherwise. NaN inputs generate false results.

```
__host__ __device__ bool __hbgeu2 (const __half2 a, const
__half2 b)
```

Performs `half2` vector unordered greater-equal comparison and returns boolean true iff both `half` results are true, boolean false otherwise.

## Parameters

**a**

- `half2`. Is only being read.

**b**

- `half2`. Is only being read.

## Returns

`bool`

- ▶ true if both `half` results of unordered greater-equal comparison of vectors `a` and `b` are true;
- ▶ false otherwise.

## Description

Performs `half2` vector greater-equal comparison of inputs `a` and `b`. The `bool` result is set to true only if both `half` greater-equal comparisons evaluate to true, or false otherwise. NaN inputs generate true results.

```
__host__ __device__ bool __hbgt2 (const __half2 a, const
__half2 b)
```

Performs `half2` vector greater-than comparison and returns boolean true iff both `half` results are true, boolean false otherwise.

## Parameters

**a**

- `half2`. Is only being read.

**b**

- `half2`. Is only being read.



## Returns

bool

- ▶ true if both `half` results of greater-than comparison of vectors `a` and `b` are true;
- ▶ false otherwise.

## Description

Performs `half2` vector greater-than comparison of inputs `a` and `b`. The bool result is set to true only if both `half` greater-than comparisons evaluate to true, or false otherwise. NaN inputs generate false results.

```
__host__ __device__ bool __hbgtu2 (const __half2 a, const
__half2 b)
```

Performs `half2` vector unordered greater-than comparison and returns boolean true iff both `half` results are true, boolean false otherwise.

## Parameters

**a**

- `half2`. Is only being read.

**b**

- `half2`. Is only being read.

## Returns

bool

- ▶ true if both `half` results of unordered greater-than comparison of vectors `a` and `b` are true;
- ▶ false otherwise.

## Description

Performs `half2` vector greater-than comparison of inputs `a` and `b`. The bool result is set to true only if both `half` greater-than comparisons evaluate to true, or false otherwise. NaN inputs generate true results.

`__host__ __device__ bool __hble2 (const __half2 a, const __half2 b)`

Performs `half2` vector less-equal comparison and returns boolean true iff both `half` results are true, boolean false otherwise.

### Parameters

**a**

- `half2`. Is only being read.

**b**

- `half2`. Is only being read.

### Returns

bool

- ▶ true if both `half` results of less-equal comparison of vectors `a` and `b` are true;
- ▶ false otherwise.

### Description

Performs `half2` vector less-equal comparison of inputs `a` and `b`. The bool result is set to true only if both `half` less-equal comparisons evaluate to true, or false otherwise. NaN inputs generate false results.

`__host__ __device__ bool __hbleu2 (const __half2 a, const __half2 b)`

Performs `half2` vector unordered less-equal comparison and returns boolean true iff both `half` results are true, boolean false otherwise.

### Parameters

**a**

- `half2`. Is only being read.

**b**

- `half2`. Is only being read.

### Returns

bool

- ▶ true if both `half` results of unordered less-equal comparison of vectors `a` and `b` are true;
- ▶ false otherwise.

## Description

Performs `half2` vector less-equal comparison of inputs `a` and `b`. The `bool` result is set to `true` only if both `half` less-equal comparisons evaluate to `true`, or `false` otherwise. NaN inputs generate `true` results.

```
__host__ __device__ bool __hblt2 (const __half2 a, const
__half2 b)
```

Performs `half2` vector less-than comparison and returns boolean `true` iff both `half` results are `true`, boolean `false` otherwise.

## Parameters

**a**

- `half2`. Is only being read.

**b**

- `half2`. Is only being read.

## Returns

`bool`

- ▶ `true` if both `half` results of less-than comparison of vectors `a` and `b` are `true`;
- ▶ `false` otherwise.

## Description

Performs `half2` vector less-than comparison of inputs `a` and `b`. The `bool` result is set to `true` only if both `half` less-than comparisons evaluate to `true`, or `false` otherwise. NaN inputs generate `false` results.

```
__host__ __device__ bool __hbltu2 (const __half2 a, const
__half2 b)
```

Performs `half2` vector unordered less-than comparison and returns boolean `true` iff both `half` results are `true`, boolean `false` otherwise.

## Parameters

**a**

- `half2`. Is only being read.

**b**

- `half2`. Is only being read.

## Returns

bool

- ▶ true if both `half` results of unordered less-than comparison of vectors `a` and `b` are true;
- ▶ false otherwise.

## Description

Performs `half2` vector less-than comparison of inputs `a` and `b`. The bool result is set to true only if both `half` less-than comparisons evaluate to true, or false otherwise. NaN inputs generate true results.

```
__host__ __device__ bool __hlt2 (const __half2 a, const
__half2 b)
```

Performs `half2` vector not-equal comparison and returns boolean true iff both `half` results are true, boolean false otherwise.

## Parameters

**a**

- `half2`. Is only being read.

**b**

- `half2`. Is only being read.

## Returns

bool

- ▶ true if both `half` results of not-equal comparison of vectors `a` and `b` are true,
- ▶ false otherwise.

## Description

Performs `half2` vector not-equal comparison of inputs `a` and `b`. The bool result is set to true only if both `half` not-equal comparisons evaluate to true, or false otherwise. NaN inputs generate false results.

`__host__ __device__ bool __hbnneu2 (const __half2 a, const __half2 b)`

Performs `half2` vector unordered not-equal comparison and returns boolean true iff both `half` results are true, boolean false otherwise.

### Parameters

**a**

- `half2`. Is only being read.

**b**

- `half2`. Is only being read.

### Returns

bool

- ▶ true if both `half` results of unordered not-equal comparison of vectors `a` and `b` are true;
- ▶ false otherwise.

### Description

Performs `half2` vector not-equal comparison of inputs `a` and `b`. The bool result is set to true only if both `half` not-equal comparisons evaluate to true, or false otherwise. NaN inputs generate true results.

`__host__ __device__ __half2 heq2 (const __half2 a, const __half2 b)`

Performs `half2` vector if-equal comparison.

### Parameters

**a**

- `half2`. Is only being read.

**b**

- `half2`. Is only being read.

### Returns

`half2`

- ▶ The vector result of if-equal comparison of vectors `a` and `b`.

### Description

Performs `half2` vector if-equal comparison of inputs `a` and `b`. The corresponding `half` results are set to 1.0 for true, or 0.0 for false. NaN inputs generate false results.

`__host__ __device__ unsigned __heq2_mask (const __half2 a, const __half2 b)`

Performs half2 vector if-equal comparison.

### Parameters

**a**

- half2. Is only being read.

**b**

- half2. Is only being read.

### Returns

unsigned int

- The vector mask result of if-equal comparison of vectors a and b.

### Description

Performs half2 vector if-equal comparison of inputs a and b. The corresponding unsigned bits are set to 0xFFFF for true, or 0x0 for false. NaN inputs generate false results.

`__host__ __device__ __half2 hequ2 (const __half2 a, const __half2 b)`

Performs half2 vector unordered if-equal comparison.

### Parameters

**a**

- half2. Is only being read.

**b**

- half2. Is only being read.

### Returns

half2

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

### Description

Performs half2 vector if-equal comparison of inputs a and b. The corresponding half results are set to 1.0 for true, or 0.0 for false. NaN inputs generate true results.

`__host__ __device__ unsigned __hequ2_mask (const __half2 a, const __half2 b)`

Performs `half2` vector unordered if-equal comparison.

### Parameters

**a**

- `half2`. Is only being read.

**b**

- `half2`. Is only being read.

### Returns

unsigned int

- The vector mask result of unordered if-equal comparison of vectors `a` and `b`.

### Description

Performs `half2` vector if-equal comparison of inputs `a` and `b`. The corresponding unsigned bits are set to `0xFFFF` for true, or `0x0` for false. NaN inputs generate true results.

`__host__ __device__ __hge2 (const __half2 a, const __half2 b)`

Performs `half2` vector greater-equal comparison.

### Parameters

**a**

- `half2`. Is only being read.

**b**

- `half2`. Is only being read.

### Returns

`half2`

- The vector result of greater-equal comparison of vectors `a` and `b`.

### Description

Performs `half2` vector greater-equal comparison of inputs `a` and `b`. The corresponding `half` results are set to `1.0` for true, or `0.0` for false. NaN inputs generate false results.

`__host__ __device__ unsigned __hge2_mask (const __half2 a, const __half2 b)`

Performs `half2` vector greater-equal comparison.

### Parameters

**a**

- `half2`. Is only being read.

**b**

- `half2`. Is only being read.

### Returns

unsigned int

- The vector mask result of greater-equal comparison of vectors `a` and `b`.

### Description

Performs `half2` vector greater-equal comparison of inputs `a` and `b`. The corresponding unsigned bits are set to `0xFFFF` for true, or `0x0` for false. NaN inputs generate false results.

`__host__ __device__ __half2 __hgeu2 (const __half2 a, const __half2 b)`

Performs `half2` vector unordered greater-equal comparison.

### Parameters

**a**

- `half2`. Is only being read.

**b**

- `half2`. Is only being read.

### Returns

`half2`

- The `half2` vector result of unordered greater-equal comparison of vectors `a` and `b`.

### Description

Performs `half2` vector greater-equal comparison of inputs `a` and `b`. The corresponding `half` results are set to `1.0` for true, or `0.0` for false. NaN inputs generate true results.



`__host__ __device__ unsigned __hgeu2_mask (const __half2 a, const __half2 b)`

Performs `half2` vector unordered greater-equal comparison.

### Parameters

**a**

- `half2`. Is only being read.

**b**

- `half2`. Is only being read.

### Returns

unsigned int

- The vector mask result of unordered greater-equal comparison of vectors `a` and `b`.

### Description

Performs `half2` vector greater-equal comparison of inputs `a` and `b`. The corresponding unsigned bits are set to `0xFFFF` for true, or `0x0` for false. NaN inputs generate true results.

`__host__ __device__ __hgt2 (const __half2 a, const __half2 b)`

Performs `half2` vector greater-than comparison.

### Parameters

**a**

- `half2`. Is only being read.

**b**

- `half2`. Is only being read.

### Returns

`half2`

- The vector result of greater-than comparison of vectors `a` and `b`.

### Description

Performs `half2` vector greater-than comparison of inputs `a` and `b`. The corresponding `half` results are set to `1.0` for true, or `0.0` for false. NaN inputs generate false results.

`__host__ __device__ unsigned __hgt2_mask (const __half2 a, const __half2 b)`

Performs `half2` vector greater-than comparison.

### Parameters

**a**

- `half2`. Is only being read.

**b**

- `half2`. Is only being read.

### Returns

unsigned int

- The vector mask result of greater-than comparison of vectors `a` and `b`.

### Description

Performs `half2` vector greater-than comparison of inputs `a` and `b`. The corresponding unsigned bits are set to `0xFFFF` for true, or `0x0` for false. NaN inputs generate false results.

`__host__ __device__ __hgtu2 (const __half2 a, const __half2 b)`

Performs `half2` vector unordered greater-than comparison.

### Parameters

**a**

- `half2`. Is only being read.

**b**

- `half2`. Is only being read.

### Returns

`half2`

- The `half2` vector result of unordered greater-than comparison of vectors `a` and `b`.

### Description

Performs `half2` vector greater-than comparison of inputs `a` and `b`. The corresponding `half` results are set to `1.0` for true, or `0.0` for false. NaN inputs generate true results.

`__host__ __device__ unsigned __hgtu2_mask (const __half2 a, const __half2 b)`

Performs `half2` vector unordered greater-than comparison.

### Parameters

**a**

- `half2`. Is only being read.

**b**

- `half2`. Is only being read.

### Returns

unsigned int

- ▶ The vector mask result of unordered greater-than comparison of vectors `a` and `b`.

### Description

Performs `half2` vector greater-than comparison of inputs `a` and `b`. The corresponding unsigned bits are set to `0xFFFF` for true, or `0x0` for false. NaN inputs generate true results.

`__host__ __device__ __hisnan2 (const __half2 a)`

Determine whether `half2` argument is a NaN.

### Parameters

**a**

- `half2`. Is only being read.

### Returns

`half2`

- ▶ The `half2` with the corresponding `half` results set to 1.0 for NaN, 0.0 otherwise.

### Description

Determine whether each half of input `half2` number `a` is a NaN.

`__host__ __device__ __hle2 (const __half2 a, const __half2 b)`

Performs `half2` vector less-equal comparison.

### Parameters

**a**

- `half2`. Is only being read.

**b**

- half2. Is only being read.

**Returns**

half2

- ▶ The half2 result of less-equal comparison of vectors a and b.

**Description**

Performs half2 vector less-equal comparison of inputs a and b. The corresponding half results are set to 1.0 for true, or 0.0 for false. NaN inputs generate false results.

```
__host__ __device__ unsigned __hlf2_mask (const __half2  
a, const __half2 b)
```

Performs half2 vector less-equal comparison.

**Parameters****a**

- half2. Is only being read.

**b**

- half2. Is only being read.

**Returns**

unsigned int

- ▶ The vector mask result of less-equal comparison of vectors a and b.

**Description**

Performs half2 vector less-equal comparison of inputs a and b. The corresponding unsigned bits are set to 0xFFFF for true, or 0x0 for false. NaN inputs generate false results.

```
__host__ __device__ __hlf2 (const __half2 a, const __half2  
b)
```

Performs half2 vector unordered less-equal comparison.

**Parameters****a**

- half2. Is only being read.

**b**

- half2. Is only being read.

## Returns

half2

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

## Description

Performs half2 vector less-equal comparison of inputs a and b. The corresponding half results are set to 1.0 for true, or 0.0 for false. NaN inputs generate true results.

```
__host__ device__ unsigned __hleu2_mask (const __half2
a, const __half2 b)
```

Performs half2 vector unordered less-equal comparison.

## Parameters

**a**

- half2. Is only being read.

**b**

- half2. Is only being read.

## Returns

unsigned int

- The vector mask result of unordered less-equal comparison of vectors a and b.

## Description

Performs half2 vector less-equal comparison of inputs a and b. The corresponding unsigned bits are set to 0xFFFF for true, or 0x0 for false. NaN inputs generate true results.

```
__host__ device__ __hlt2 (const __half2 a, const __half2 b)
```

Performs half2 vector less-than comparison.

## Parameters

**a**

- half2. Is only being read.

**b**

- half2. Is only being read.

## Returns

half2

- The half2 vector result of less-than comparison of vectors a and b.

## Description

Performs `half2` vector less-than comparison of inputs `a` and `b`. The corresponding `half` results are set to 1.0 for true, or 0.0 for false. NaN inputs generate false results.

```
__host__ __device__ unsigned __hlt2_mask (const __half2 a,
const __half2 b)
```

Performs `half2` vector less-than comparison.

## Parameters

**a**

- `half2`. Is only being read.

**b**

- `half2`. Is only being read.

## Returns

unsigned int

- The vector mask result of less-than comparison of vectors `a` and `b`.

## Description

Performs `half2` vector less-than comparison of inputs `a` and `b`. The corresponding unsigned bits are set to 0xFFFF for true, or 0x0 for false. NaN inputs generate false results.

```
__host__ __device__ __hltu2 (const __half2 a, const __half2
b)
```

Performs `half2` vector unordered less-than comparison.

## Parameters

**a**

- `half2`. Is only being read.

**b**

- `half2`. Is only being read.

## Returns

`half2`

- The vector result of unordered less-than comparison of vectors `a` and `b`.

## Description

Performs `half2` vector less-than comparison of inputs `a` and `b`. The corresponding `half` results are set to 1.0 for true, or 0.0 for false. NaN inputs generate true results.

```
__host__ __device__ unsigned __hltu2_mask (const __half2
a, const __half2 b)
```

Performs `half2` vector unordered less-than comparison.

## Parameters

**a**

- `half2`. Is only being read.

**b**

- `half2`. Is only being read.

## Returns

unsigned int

- ▶ The vector mask result of unordered less-than comparison of vectors `a` and `b`.

## Description

Performs `half2` vector less-than comparison of inputs `a` and `b`. The corresponding unsigned bits are set to 0xFFFF for true, or 0x0 for false. NaN inputs generate true results.

```
__host__ __device__ __hmax2 (const __half2 a, const __half2
b)
```

Calculates `half2` vector maximum of two inputs.

## Description

Calculates `half2` vector  $\max(a, b)$ . Elementwise `half` operation is defined as  $(a > b) ? a : b$ .

- ▶ If either of inputs is NaN, the other input is returned.
- ▶ If both inputs are NaNs, then canonical NaN is returned.
- ▶ If values of both inputs are 0.0, then  $+0.0 > -0.0$
- ▶ The result of elementwise maximum of vectors `a` and `b`

`__host__ __device__ hmax2_nan (const __half2 a, const __half2 b)`

Calculates `half2` vector maximum of two inputs, NaNs pass through.

### Description

Calculates `half2` vector  $\max(a, b)$ . Elementwise `half` operation is defined as  $(a > b) ? a : b$ .

- ▶ If either of inputs is NaN, then canonical NaN is returned.
- ▶ If values of both inputs are 0.0, then  $+0.0 > -0.0$
- ▶ The result of elementwise maximum of vectors `a` and `b`, with NaNs pass through

`__host__ __device__ hmin2 (const __half2 a, const __half2 b)`

Calculates `half2` vector minimum of two inputs.

### Description

Calculates `half2` vector  $\min(a, b)$ . Elementwise `half` operation is defined as  $(a < b) ? a : b$ .

- ▶ If either of inputs is NaN, the other input is returned.
- ▶ If both inputs are NaNs, then canonical NaN is returned.
- ▶ If values of both inputs are 0.0, then  $+0.0 > -0.0$
- ▶ The result of elementwise minimum of vectors `a` and `b`

`__host__ __device__ hmin2_nan (const __half2 a, const __half2 b)`

Calculates `half2` vector minimum of two inputs, NaNs pass through.

### Description

Calculates `half2` vector  $\min(a, b)$ . Elementwise `half` operation is defined as  $(a < b) ? a : b$ .

- ▶ If either of inputs is NaN, then canonical NaN is returned.
- ▶ If values of both inputs are 0.0, then  $+0.0 > -0.0$
- ▶ The result of elementwise minimum of vectors `a` and `b`, with NaNs pass through



`__host__ device__ hne2 (const __half2 a, const __half2 b)`

Performs `half2` vector not-equal comparison.

### Parameters

**a**

- `half2`. Is only being read.

**b**

- `half2`. Is only being read.

### Returns

`half2`

- The vector result of not-equal comparison of vectors `a` and `b`.

### Description

Performs `half2` vector not-equal comparison of inputs `a` and `b`. The corresponding `half` results are set to 1.0 for true, or 0.0 for false. NaN inputs generate false results.

`__host__ device__ unsigned __hne2_mask (const __half2 a, const __half2 b)`

Performs `half2` vector not-equal comparison.

### Parameters

**a**

- `half2`. Is only being read.

**b**

- `half2`. Is only being read.

### Returns

unsigned int

- The vector mask result of not-equal comparison of vectors `a` and `b`.

### Description

Performs `half2` vector not-equal comparison of inputs `a` and `b`. The corresponding unsigned bits are set to 0xFFFF for true, or 0x0 for false. NaN inputs generate false results.

`__host__ __device__ hneu2 (const __half2 a, const __half2 b)`

Performs `half2` vector unordered not-equal comparison.

### Parameters

**a**

- `half2`. Is only being read.

**b**

- `half2`. Is only being read.

### Returns

`half2`

- The vector result of unordered not-equal comparison of vectors `a` and `b`.

### Description

Performs `half2` vector not-equal comparison of inputs `a` and `b`. The corresponding `half` results are set to 1.0 for true, or 0.0 for false. NaN inputs generate true results.

`__host__ __device__ unsigned __hneu2_mask (const __half2 a, const __half2 b)`

Performs `half2` vector unordered not-equal comparison.

### Parameters

**a**

- `half2`. Is only being read.

**b**

- `half2`. Is only being read.

### Returns

unsigned int

- The vector mask result of unordered not-equal comparison of vectors `a` and `b`.

### Description

Performs `half2` vector not-equal comparison of inputs `a` and `b`. The corresponding unsigned bits are set to 0xFFFF for true, or 0x0 for false. NaN inputs generate true results.

```
__host__ __device__ __CUDA_FP16_FORCEINLINE__ bool
operator!= (const __half2 lh, const __half2 rh)
```

### Description

Performs packed `half` unordered compare not-equal operation. See also [\\_\\_hbneu2\(\\_\\_half2, \\_\\_half2\)](#)

```
__host__ __device__ __CUDA_FP16_FORCEINLINE__ bool
operator< (const __half2 lh, const __half2 rh)
```

### Description

Performs packed `half` ordered less-than compare operation. See also [\\_\\_hblt2\(\\_\\_half2, \\_\\_half2\)](#)

```
__host__ __device__ __CUDA_FP16_FORCEINLINE__ bool
operator<= (const __half2 lh, const __half2 rh)
```

### Description

Performs packed `half` ordered less-or-equal compare operation. See also [\\_\\_hble2\(\\_\\_half2, \\_\\_half2\)](#)

```
__host__ __device__ __CUDA_FP16_FORCEINLINE__ bool
operator== (const __half2 lh, const __half2 rh)
```

### Description

Performs packed `half` ordered compare equal operation. See also [\\_\\_hbeq2\(\\_\\_half2, \\_\\_half2\)](#)

```
__host__ __device__ __CUDA_FP16_FORCEINLINE__ bool
operator> (const __half2 lh, const __half2 rh)
```

### Description

Performs packed `half` ordered greater-than compare operation. See also [\\_\\_hbgt2\(\\_\\_half2, \\_\\_half2\)](#)

`__host__ __device__ __CUDA_FP16_FORCEINLINE__ bool  
operator>= (const __half2 lh, const __half2 rh)`

### Description

Performs packed `half` ordered greater-or-equal compare operation. See also [\\_\\_hbge2\(\\_\\_half2, \\_\\_half2\)](#).

## 1.2.6. Half Precision Conversion and Data Movement

### Half Precision Ininsics

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

`__host__ __device__ __double2half (const double a)`

Converts double number to half precision in round-to-nearest-even mode and returns `half` with converted value.

### Parameters

**a**

- double. Is only being read.

### Returns

`half`

- ▶ a converted to half.

### Description

Converts double number a to half precision in round-to-nearest-even mode.

`__host__ __device__ __float2half2_rn (const float2 a)`

Converts both components of `float2` number to half precision in round-to-nearest-even mode and returns `half2` with converted values.

### Parameters

**a**

- `float2`. Is only being read.

### Returns

`half2`

- ▶ The `half2` which has corresponding halves equal to the converted `float2` components.

### Description

Converts both components of `float2` to half precision in round-to-nearest-even mode and combines the results into one `half2` number. Low 16 bits of the return value correspond to `a.x` and high 16 bits of the return value correspond to `a.y`.

### `__host__ __device__ float2half (const float a)`

Converts float number to half precision in round-to-nearest-even mode and returns `half` with converted value.

### Parameters

**a**

- float. Is only being read.

### Returns

`half`

- ▶ `a` converted to half.

### Description

Converts float number `a` to half precision in round-to-nearest-even mode.

### `__host__ __device__ float2half2_rn (const float a)`

Converts input to half precision in round-to-nearest-even mode and populates both halves of `half2` with converted value.

### Parameters

**a**

- float. Is only being read.

### Returns

`half2`

- ▶ The `half2` value with both halves equal to the converted half precision number.

### Description

Converts input `a` to half precision in round-to-nearest-even mode and populates both halves of `half2` with converted value.

## `__host__ __device__ float2half_rd (const float a)`

Converts float number to half precision in round-down mode and returns `half` with converted value.

### Parameters

**a**

- float. Is only being read.

### Returns

`half`

- ▶ a converted to half.

### Description

Converts float number a to half precision in round-down mode.

## `__host__ __device__ float2half_rn (const float a)`

Converts float number to half precision in round-to-nearest-even mode and returns `half` with converted value.

### Parameters

**a**

- float. Is only being read.

### Returns

`half`

- ▶ a converted to half.

### Description

Converts float number a to half precision in round-to-nearest-even mode.

## `__host__ __device__ float2half_ru (const float a)`

Converts float number to half precision in round-up mode and returns `half` with converted value.

### Parameters

**a**

- float. Is only being read.

## Returns

half

- ▶ a converted to half.

## Description

Converts float number a to half precision in round-up mode.

### `__host__ __device__ float2half_rz (const float a)`

Converts float number to half precision in round-towards-zero mode and returns `half` with converted value.

## Parameters

**a**

- float. Is only being read.

## Returns

half

- ▶ a converted to half.

## Description

Converts float number a to half precision in round-towards-zero mode.

### `__host__ __device__ floats2half2_rn (const float a, const float b)`

Converts both input floats to half precision in round-to-nearest-even mode and returns `half2` with converted values.

## Parameters

**a**

- float. Is only being read.

**b**

- float. Is only being read.

## Returns

half2

- ▶ The `half2` value with corresponding halves equal to the converted input floats.

## Description

Converts both input floats to half precision in round-to-nearest-even mode and combines the results into one `half2` number. Low 16 bits of the return value correspond to the input `a`, high 16 bits correspond to the input `b`.

```
__host__ __device__ float2 __half22float2 (const __half2 a)
```

Converts both halves of `half2` to `float2` and returns the result.

## Parameters

**a**

- `half2`. Is only being read.

## Returns

`float2`

- ▶ `a` converted to `float2`.

## Description

Converts both halves of `half2` input `a` to `float2` and returns the result.

```
__host__ __device__ signed char __half2char_rz (const __half h)
```

Convert a half to a signed char in round-towards-zero mode.

## Parameters

**h**

- `half`. Is only being read.

## Returns

signed char

- ▶ `h` converted to a signed char.

## Description

Convert the half-precision floating-point value `h` to a signed char integer in round-towards-zero mode. NaN inputs are converted to 0.



`__host__ __device__ float __half2float (const __half a)`

Converts `half` number to float.

### Parameters

**a**

- float. Is only being read.

### Returns

float

- ▶ a converted to float.

### Description

Converts half number a to float.

`__host__ __device__ half2 half2 (const __half a)`

Returns `half2` with both halves equal to the input value.

### Parameters

**a**

- half. Is only being read.

### Returns

half2

- ▶ The vector which has both its halves equal to the input a.

### Description

Returns `half2` number with both halves equal to the input a `half` number.

`__device__ int __half2int_rd (const __half h)`

Convert a half to a signed integer in round-down mode.

### Parameters

**h**

- half. Is only being read.

### Returns

int

- ▶ h converted to a signed integer.

## Description

Convert the half-precision floating-point value `h` to a signed integer in round-down mode. NaN inputs are converted to 0.

`__device__ int __half2int_rn (const __half h)`

Convert a half to a signed integer in round-to-nearest-even mode.

## Parameters

**h**

- half. Is only being read.

## Returns

int

- ▶ `h` converted to a signed integer.

## Description

Convert the half-precision floating-point value `h` to a signed integer in round-to-nearest-even mode. NaN inputs are converted to 0.

`__device__ int __half2int_ru (const __half h)`

Convert a half to a signed integer in round-up mode.

## Parameters

**h**

- half. Is only being read.

## Returns

int

- ▶ `h` converted to a signed integer.

## Description

Convert the half-precision floating-point value `h` to a signed integer in round-up mode. NaN inputs are converted to 0.

`__host__ __device__ int __half2int_rz (const __half h)`

Convert a half to a signed integer in round-towards-zero mode.

### Parameters

**h**

- half. Is only being read.

### Returns

int

- ▶ h converted to a signed integer.

### Description

Convert the half-precision floating-point value `h` to a signed integer in round-towards-zero mode. NaN inputs are converted to 0.

`__device__ long long int __half2ll_rd (const __half h)`

Convert a half to a signed 64-bit integer in round-down mode.

### Parameters

**h**

- half. Is only being read.

### Returns

long long int

- ▶ h converted to a signed 64-bit integer.

### Description

Convert the half-precision floating-point value `h` to a signed 64-bit integer in round-down mode. NaN inputs return a long long int with hex value of 0x8000000000000000.

`__device__ long long int __half2ll_rn (const __half h)`

Convert a half to a signed 64-bit integer in round-to-nearest-even mode.

### Parameters

**h**

- half. Is only being read.

## Returns

long long int

- ▶ `h` converted to a signed 64-bit integer.

## Description

Convert the half-precision floating-point value `h` to a signed 64-bit integer in round-to-nearest-even mode. NaN inputs return a long long int with hex value of 0x8000000000000000.

`__device__ long long int __half2ll_ru (const __half h)`

Convert a half to a signed 64-bit integer in round-up mode.

## Parameters

**h**

- half. Is only being read.

## Returns

long long int

- ▶ `h` converted to a signed 64-bit integer.

## Description

Convert the half-precision floating-point value `h` to a signed 64-bit integer in round-up mode. NaN inputs return a long long int with hex value of 0x8000000000000000.

`__host__ __device__ long long int __half2ll_rz (const __half h)`

Convert a half to a signed 64-bit integer in round-towards-zero mode.

## Parameters

**h**

- half. Is only being read.

## Returns

long long int

- ▶ `h` converted to a signed 64-bit integer.

## Description

Convert the half-precision floating-point value `h` to a signed 64-bit integer in round-towards-zero mode. NaN inputs return a long long int with hex value of 0x8000000000000000.

`__device__ short int __half2short_rd (const __half h)`

Convert a half to a signed short integer in round-down mode.

## Parameters

**h**

- half. Is only being read.

## Returns

short int

- ▶ `h` converted to a signed short integer.

## Description

Convert the half-precision floating-point value `h` to a signed short integer in round-down mode. NaN inputs are converted to 0.

`__device__ short int __half2short_rn (const __half h)`

Convert a half to a signed short integer in round-to-nearest-even mode.

## Parameters

**h**

- half. Is only being read.

## Returns

short int

- ▶ `h` converted to a signed short integer.

## Description

Convert the half-precision floating-point value `h` to a signed short integer in round-to-nearest-even mode. NaN inputs are converted to 0.

`__device__ short int __half2short_ru (const __half h)`

Convert a half to a signed short integer in round-up mode.

### Parameters

**h**

- half. Is only being read.

### Returns

short int

- ▶ h converted to a signed short integer.

### Description

Convert the half-precision floating-point value `h` to a signed short integer in round-up mode. NaN inputs are converted to 0.

`__host____device__ short int __half2short_rz (const __half h)`

Convert a half to a signed short integer in round-towards-zero mode.

### Parameters

**h**

- half. Is only being read.

### Returns

short int

- ▶ h converted to a signed short integer.

### Description

Convert the half-precision floating-point value `h` to a signed short integer in round-towards-zero mode. NaN inputs are converted to 0.

`__host____device__ unsigned char __half2uchar_rz (const __half h)`

Convert a half to an unsigned char in round-towards-zero mode.

### Parameters

**h**

- half. Is only being read.

## Returns

unsigned char

- ▶ `h` converted to an unsigned char.

## Description

Convert the half-precision floating-point value `h` to an unsigned char in round-towards-zero mode. NaN inputs are converted to 0.

`__device__ unsigned int __half2uint_rd (const __half h)`

Convert a half to an unsigned integer in round-down mode.

## Parameters

**h**

- half. Is only being read.

## Returns

unsigned int

- ▶ `h` converted to an unsigned integer.

## Description

Convert the half-precision floating-point value `h` to an unsigned integer in round-down mode. NaN inputs are converted to 0.

`__device__ unsigned int __half2uint_rn (const __half h)`

Convert a half to an unsigned integer in round-to-nearest-even mode.

## Parameters

**h**

- half. Is only being read.

## Returns

unsigned int

- ▶ `h` converted to an unsigned integer.

## Description

Convert the half-precision floating-point value `h` to an unsigned integer in round-to-nearest-even mode. NaN inputs are converted to 0.

`__device__ unsigned int __half2uint_ru (const __half h)`

Convert a half to an unsigned integer in round-up mode.

### Parameters

**h**

- half. Is only being read.

### Returns

unsigned int

- ▶ h converted to an unsigned integer.

### Description

Convert the half-precision floating-point value `h` to an unsigned integer in round-up mode. NaN inputs are converted to 0.

`__host__ __device__ unsigned int __half2uint_rz (const __half h)`

Convert a half to an unsigned integer in round-towards-zero mode.

### Parameters

**h**

- half. Is only being read.

### Returns

unsigned int

- ▶ h converted to an unsigned integer.

### Description

Convert the half-precision floating-point value `h` to an unsigned integer in round-towards-zero mode. NaN inputs are converted to 0.

`__device__ unsigned long long int __half2ull_rd (const __half h)`

Convert a half to an unsigned 64-bit integer in round-down mode.

### Parameters

**h**

- half. Is only being read.



## Returns

unsigned long long int

- ▶ `h` converted to an unsigned 64-bit integer.

## Description

Convert the half-precision floating-point value `h` to an unsigned 64-bit integer in round-down mode. NaN inputs return 0x8000000000000000.

```
__device__ unsigned long long int __half2ull_rn (const  
__half h)
```

Convert a half to an unsigned 64-bit integer in round-to-nearest-even mode.

## Parameters

**h**

- half. Is only being read.

## Returns

unsigned long long int

- ▶ `h` converted to an unsigned 64-bit integer.

## Description

Convert the half-precision floating-point value `h` to an unsigned 64-bit integer in round-to-nearest-even mode. NaN inputs return 0x8000000000000000.

```
__device__ unsigned long long int __half2ull_ru (const  
__half h)
```

Convert a half to an unsigned 64-bit integer in round-up mode.

## Parameters

**h**

- half. Is only being read.

## Returns

unsigned long long int

- ▶ `h` converted to an unsigned 64-bit integer.

## Description

Convert the half-precision floating-point value `h` to an unsigned 64-bit integer in round-up mode. NaN inputs return 0x8000000000000000.

```
__host__ __device__ unsigned long long int __half2ull_rz
(const __half h)
```

Convert a half to an unsigned 64-bit integer in round-towards-zero mode.

## Parameters

**h**

- half. Is only being read.

## Returns

unsigned long long int

- ▶ `h` converted to an unsigned 64-bit integer.

## Description

Convert the half-precision floating-point value `h` to an unsigned 64-bit integer in round-towards-zero mode. NaN inputs return 0x8000000000000000.

```
__device__ unsigned short int __half2ushort_rd (const
__half h)
```

Convert a half to an unsigned short integer in round-down mode.

## Parameters

**h**

- half. Is only being read.

## Returns

unsigned short int

- ▶ `h` converted to an unsigned short integer.

## Description

Convert the half-precision floating-point value `h` to an unsigned short integer in round-down mode. NaN inputs are converted to 0.

```
__device__ unsigned short int __half2ushort_rn (const  
__half h)
```

Convert a half to an unsigned short integer in round-to-nearest-even mode.

### Parameters

**h**

- half. Is only being read.

### Returns

unsigned short int

- ▶ h converted to an unsigned short integer.

### Description

Convert the half-precision floating-point value h to an unsigned short integer in round-to-nearest-even mode. NaN inputs are converted to 0.

```
__device__ unsigned short int __half2ushort_ru (const  
__half h)
```

Convert a half to an unsigned short integer in round-up mode.

### Parameters

**h**

- half. Is only being read.

### Returns

unsigned short int

- ▶ h converted to an unsigned short integer.

### Description

Convert the half-precision floating-point value h to an unsigned short integer in round-up mode. NaN inputs are converted to 0.

```
__host__ __device__ unsigned short int __half2ushort_rz
(const __half h)
```

Convert a half to an unsigned short integer in round-towards-zero mode.

### Parameters

**h**

- half. Is only being read.

### Returns

unsigned short int

- ▶ h converted to an unsigned short integer.

### Description

Convert the half-precision floating-point value `h` to an unsigned short integer in round-towards-zero mode. NaN inputs are converted to 0.

```
__host__ __device__ short int __half_as_short (const __half
h)
```

Reinterprets bits in a `half` as a signed short integer.

### Parameters

**h**

- half. Is only being read.

### Returns

short int

- ▶ The reinterpreted value.

### Description

Reinterprets the bits in the half-precision floating-point number `h` as a signed short integer.

```
__host__ __device__ unsigned short int __half_as_ushort
(const __half h)
```

Reinterprets bits in a `half` as an unsigned short integer.

### Parameters

**h**

- `half`. Is only being read.

### Returns

unsigned short int

► The reinterpreted value.

### Description

Reinterprets the bits in the half-precision floating-point `h` as an unsigned short number.

```
__host__ __device__ __half2 __half2half2 (const __half a, const
__half b)
```

Combines two `half` numbers into one `half2` number.

### Parameters

**a**

- `half`. Is only being read.

**b**

- `half`. Is only being read.

### Returns

`half2`

► The `half2` with one `half` equal to `a` and the other to `b`.

### Description

Combines two input `half` number `a` and `b` into one `half2` number. Input `a` is stored in low 16 bits of the return value, input `b` is stored in high 16 bits of the return value.

## `__host__ __device__ float __high2float (const __half2 a)`

Converts high 16 bits of `half2` to float and returns the result.

### Parameters

**a**

- `half2`. Is only being read.

### Returns

float

- ▶ The high 16 bits of `a` converted to float.

### Description

Converts high 16 bits of `half2` input `a` to 32-bit floating-point number and returns the result.

## `__host__ __device__ __half2 __high2half (const __half2 a)`

Returns high 16 bits of `half2` input.

### Parameters

**a**

- `half2`. Is only being read.

### Returns

`half`

- ▶ The high 16 bits of the input.

### Description

Returns high 16 bits of `half2` input `a`.

## `__host__ __device__ __half2 __high2half2 (const __half2 a)`

Extracts high 16 bits from `half2` input.

### Parameters

**a**

- `half2`. Is only being read.

### Returns

`half2`

- ▶ The `half2` with both halves equal to the high 16 bits of the input.

## Description

Extracts high 16 bits from `half2` input `a` and returns a new `half2` number which has both halves equal to the extracted bits.

`__host__ __device__ highs2half2 (const __half2 a, const __half2 b)`

Extracts high 16 bits from each of the two `half2` inputs and combines into one `half2` number.

## Parameters

**a**

- `half2`. Is only being read.

**b**

- `half2`. Is only being read.

## Returns

`half2`

- ▶ The high 16 bits of `a` and of `b`.

## Description

Extracts high 16 bits from each of the two `half2` inputs and combines into one `half2` number. High 16 bits from input `a` is stored in low 16 bits of the return value, high 16 bits from input `b` is stored in high 16 bits of the return value.

`__host__ __device__ int2half_rd (const int i)`

Convert a signed integer to a half in round-down mode.

## Parameters

**i**

- `int`. Is only being read.

## Returns

`half`

- ▶ `i` converted to `half`.

## Description

Convert the signed integer value `i` to a half-precision floating-point value in round-down mode.

## `__host__ __device__ int2half_rn (const int i)`

Convert a signed integer to a half in round-to-nearest-even mode.

### Parameters

**i**  
- int. Is only being read.

### Returns

half

► *i* converted to half.

### Description

Convert the signed integer value *i* to a half-precision floating-point value in round-to-nearest-even mode.

## `__host__ __device__ int2half_ru (const int i)`

Convert a signed integer to a half in round-up mode.

### Parameters

**i**  
- int. Is only being read.

### Returns

half

► *i* converted to half.

### Description

Convert the signed integer value *i* to a half-precision floating-point value in round-up mode.

## `__host__ __device__ int2half_rz (const int i)`

Convert a signed integer to a half in round-towards-zero mode.

### Parameters

**i**  
- int. Is only being read.

### Returns

half



- ▶ `i` converted to half.

### Description

Convert the signed integer value `i` to a half-precision floating-point value in round-towards-zero mode.

`__device__ ldca (const __half *ptr)`

Generates a ``ld.global.ca`` load instruction.

### Parameters

#### **ptr**

- memory location

### Returns

The value pointed by ``ptr``

`__device__ ldca (const __half2 *ptr)`

Generates a ``ld.global.ca`` load instruction.

### Parameters

#### **ptr**

- memory location

### Returns

The value pointed by ``ptr``

`__device__ ldcg (const __half *ptr)`

Generates a ``ld.global.cg`` load instruction.

### Parameters

#### **ptr**

- memory location

### Returns

The value pointed by ``ptr``

`__device__ ldcg (const __half2 *ptr)`

Generates a ``ld.global.cg`` load instruction.

### Parameters

#### **ptr**

- memory location

### Returns

The value pointed by ``ptr``

`__device__ ldcs (const __half *ptr)`

Generates a ``ld.global.cs`` load instruction.

### Parameters

#### **ptr**

- memory location

### Returns

The value pointed by ``ptr``

`__device__ ldcsc (const __half2 *ptr)`

Generates a ``ld.global.cs`` load instruction.

### Parameters

#### **ptr**

- memory location

### Returns

The value pointed by ``ptr``

`__device__ ldcvc (const __half *ptr)`

Generates a ``ld.global.cv`` load instruction.

### Parameters

#### **ptr**

- memory location

### Returns

The value pointed by ``ptr``

`__device__ ldcv (const __half2 *ptr)`

Generates a ``ld.global.cv`` load instruction.

### Parameters

#### **ptr**

- memory location

### Returns

The value pointed by ``ptr``

`__device__ ldg (const __half *ptr)`

Generates a ``ld.global.nc`` load instruction.

### Parameters

#### **ptr**

- memory location

### Returns

The value pointed by ``ptr``

`__device__ ldg (const __half2 *ptr)`

Generates a ``ld.global.nc`` load instruction.

### Parameters

#### **ptr**

- memory location

### Returns

The value pointed by ``ptr``

### Description

`defined(__CUDA_ARCH__) || (__CUDA_ARCH__ >= 300)`

`__device__ ldlu (const __half *ptr)`

Generates a ``ld.global.lu`` load instruction.

### Parameters

#### **ptr**

- memory location

## Returns

The value pointed by `ptr`

`__device__ ldlu (const __half2 *ptr)`

Generates a `ld.global.lu` load instruction.

## Parameters

### **ptr**

- memory location

## Returns

The value pointed by `ptr`

`__host__ __device__ ll2half_rd (const long long int i)`

Convert a signed 64-bit integer to a half in round-down mode.

## Parameters

### **i**

- long long int. Is only being read.

## Returns

half

- ▶ `i` converted to half.

## Description

Convert the signed 64-bit integer value `i` to a half-precision floating-point value in round-down mode.

`__host__ __device__ ll2half_rn (const long long int i)`

Convert a signed 64-bit integer to a half in round-to-nearest-even mode.

## Parameters

### **i**

- long long int. Is only being read.

## Returns

half

- ▶ `i` converted to half.

## Description

Convert the signed 64-bit integer value `i` to a half-precision floating-point value in round-to-nearest-even mode.

`__host__ __device__ ll2half_ru (const long long int i)`

Convert a signed 64-bit integer to a half in round-up mode.

## Parameters

**i**

- long long int. Is only being read.

## Returns

half

- ▶ `i` converted to half.

## Description

Convert the signed 64-bit integer value `i` to a half-precision floating-point value in round-up mode.

`__host__ __device__ ll2half_rz (const long long int i)`

Convert a signed 64-bit integer to a half in round-towards-zero mode.

## Parameters

**i**

- long long int. Is only being read.

## Returns

half

- ▶ `i` converted to half.

## Description

Convert the signed 64-bit integer value `i` to a half-precision floating-point value in round-towards-zero mode.

## `__host__ __device__ float __low2float (const __half2 a)`

Converts low 16 bits of `half2` to float and returns the result.

### Parameters

**a**

- `half2`. Is only being read.

### Returns

float

- ▶ The low 16 bits of `a` converted to float.

### Description

Converts low 16 bits of `half2` input `a` to 32-bit floating-point number and returns the result.

## `__host__ __device__ __half2 __low2half (const __half2 a)`

Returns low 16 bits of `half2` input.

### Parameters

**a**

- `half2`. Is only being read.

### Returns

`half`

- ▶ Returns `half` which contains low 16 bits of the input `a`.

### Description

Returns low 16 bits of `half2` input `a`.

## `__host__ __device__ __half2 __low2half2 (const __half2 a)`

Extracts low 16 bits from `half2` input.

### Parameters

**a**

- `half2`. Is only being read.

### Returns

`half2`

- ▶ The `half2` with both halves equal to the low 16 bits of the input.

## Description

Extracts low 16 bits from `half2` input `a` and returns a new `half2` number which has both halves equal to the extracted bits.

`__host__ __device__ lowhigh2highlow (const __half2 a)`

Swaps both halves of the `half2` input.

## Parameters

**a**

- `half2`. Is only being read.

## Returns

`half2`

► `a` with its halves being swapped.

## Description

Swaps both halves of the `half2` input and returns a new `half2` number with swapped halves.

`__host__ __device__ lows2half2 (const __half2 a, const __half2 b)`

Extracts low 16 bits from each of the two `half2` inputs and combines into one `half2` number.

## Parameters

**a**

- `half2`. Is only being read.

**b**

- `half2`. Is only being read.

## Returns

`half2`

► The low 16 bits of `a` and of `b`.

## Description

Extracts low 16 bits from each of the two `half2` inputs and combines into one `half2` number. Low 16 bits from input `a` is stored in low 16 bits of the return value, low 16 bits from input `b` is stored in high 16 bits of the return value.

## `__device__ shfl_down_sync (const unsigned mask, const __half var, const unsigned int delta, const int width)`

Exchange a variable between threads within a warp. Copy from a thread with higher ID relative to the caller.

### Parameters

#### **mask**

- unsigned int. Is only being read.

#### **var**

- half. Is only being read.

#### **delta**

- int. Is only being read.

#### **width**

- int. Is only being read.

### Returns

Returns the 2-byte word referenced by var from the source thread ID as half. If the source thread ID is out of range or the source thread has exited, the calling thread's own var is returned.

### Description

Calculates a source thread ID by adding delta to the caller's thread ID. The value of var held by the resulting thread ID is returned: this has the effect of shifting var down the warp by delta threads. If width is less than warpSize then each subsection of the warp behaves as a separate entity with a starting logical thread ID of 0. As for `__shfl_up_sync()`, the ID number of the source thread will not wrap around the value of width and so the upper delta threads will remain unchanged.



#### **Note:**

For more details for this function see the Warp Shuffle Functions section in the CUDA C++ Programming Guide.



## `__device__ shfl_down_sync (const unsigned mask, const __half var, const unsigned int delta, const int width)`

Exchange a variable between threads within a warp. Copy from a thread with higher ID relative to the caller.

### Parameters

#### **mask**

- unsigned int. Is only being read.

#### **var**

- half2. Is only being read.

#### **delta**

- int. Is only being read.

#### **width**

- int. Is only being read.

### Returns

Returns the 4-byte word referenced by var from the source thread ID as half2. If the source thread ID is out of range or the source thread has exited, the calling thread's own var is returned.

### Description

Calculates a source thread ID by adding delta to the caller's thread ID. The value of var held by the resulting thread ID is returned: this has the effect of shifting var down the warp by delta threads. If width is less than warpSize then each subsection of the warp behaves as a separate entity with a starting logical thread ID of 0. As for `shfl_up_sync()`, the ID number of the source thread will not wrap around the value of width and so the upper delta threads will remain unchanged.



#### **Note:**

For more details for this function see the Warp Shuffle Functions section in the CUDA C++ Programming Guide.

## `__device__ shfl_sync (const unsigned mask, const __half var, const int delta, const int width)`

Exchange a variable between threads within a warp. Direct copy from indexed thread.

### Parameters

#### **mask**

- unsigned int. Is only being read.

**var**

- half. Is only being read.

**delta**

- int. Is only being read.

**width**

- int. Is only being read.

**Returns**

Returns the 2-byte word referenced by var from the source thread ID as half. If the source thread ID is out of range or the source thread has exited, the calling thread's own var is returned.

**Description**

Returns the value of var held by the thread whose ID is given by delta. If width is less than warpSize then each subsection of the warp behaves as a separate entity with a starting logical thread ID of 0. If delta is outside the range [0:width-1], the value returned corresponds to the value of var held by the delta modulo width (i.e. within the same subsection). width must have a value which is a power of 2; results are undefined if width is not a power of 2, or is a number greater than warpSize.

**Note:**

For more details for this function see the Warp Shuffle Functions section in the CUDA C++ Programming Guide.

## **\_\_device\_\_ shfl\_sync (const unsigned mask, const \_\_half2 var, const int delta, const int width)**

Exchange a variable between threads within a warp. Direct copy from indexed thread.

**Parameters****mask**

- unsigned int. Is only being read.

**var**

- half2. Is only being read.

**delta**

- int. Is only being read.

**width**

- int. Is only being read.

## Returns

Returns the 4-byte word referenced by `var` from the source thread ID as `half2`. If the source thread ID is out of range or the source thread has exited, the calling thread's own `var` is returned.

## Description

Returns the value of `var` held by the thread whose ID is given by `delta`. If `width` is less than `warpSize` then each subsection of the warp behaves as a separate entity with a starting logical thread ID of 0. If `delta` is outside the range `[0:width-1]`, the value returned corresponds to the value of `var` held by the `delta` modulo `width` (i.e. within the same subsection). `width` must have a value which is a power of 2; results are undefined if `width` is not a power of 2, or is a number greater than `warpSize`.

## `__device__ shfl_up_sync (const unsigned mask, const __half var, const unsigned int delta, const int width)`

Exchange a variable between threads within a warp. Copy from a thread with lower ID relative to the caller.

## Parameters

### **mask**

- unsigned int. Is only being read.

### **var**

- `half`. Is only being read.

### **delta**

- int. Is only being read.

### **width**

- int. Is only being read.

## Returns

Returns the 2-byte word referenced by `var` from the source thread ID as `half`. If the source thread ID is out of range or the source thread has exited, the calling thread's own `var` is returned.

## Description

Calculates a source thread ID by subtracting `delta` from the caller's lane ID. The value of `var` held by the resulting lane ID is returned: in effect, `var` is shifted up the warp by `delta` threads. If `width` is less than `warpSize` then each subsection of the warp behaves as a separate entity with a starting logical thread ID of 0. The source thread index will not wrap around the value of `width`, so effectively the lower `delta` threads will be unchanged. `width` must have a value which

is a power of 2; results are undefined if width is not a power of 2, or is a number greater than warpSize.

**Note:**

For more details for this function see the Warp Shuffle Functions section in the CUDA C++ Programming Guide.

## `__device__ shfl_up_sync (const unsigned mask, const __half2 var, const unsigned int delta, const int width)`

Exchange a variable between threads within a warp. Copy from a thread with lower ID relative to the caller.

### Parameters

**mask**

- unsigned int. Is only being read.

**var**

- half2. Is only being read.

**delta**

- int. Is only being read.

**width**

- int. Is only being read.

### Returns

Returns the 4-byte word referenced by var from the source thread ID as half2. If the source thread ID is out of range or the source thread has exited, the calling thread's own var is returned.

### Description

Calculates a source thread ID by subtracting delta from the caller's lane ID. The value of var held by the resulting lane ID is returned: in effect, var is shifted up the warp by delta threads. If width is less than warpSize then each subsection of the warp behaves as a separate entity with a starting logical thread ID of 0. The source thread index will not wrap around the value of width, so effectively the lower delta threads will be unchanged. width must have a value which is a power of 2; results are undefined if width is not a power of 2, or is a number greater than warpSize.

**Note:**

For more details for this function see the Warp Shuffle Functions section in the CUDA C++ Programming Guide.

## `__device__ shfl_xor_sync (const unsigned mask, const __half var, const int delta, const int width)`

Exchange a variable between threads within a warp. Copy from a thread based on bitwise XOR of own thread ID.

### Parameters

#### **mask**

- unsigned int. Is only being read.

#### **var**

- half. Is only being read.

#### **delta**

- int. Is only being read.

#### **width**

- int. Is only being read.

### Returns

Returns the 2-byte word referenced by var from the source thread ID as half. If the source thread ID is out of range or the source thread has exited, the calling thread's own var is returned.

### Description

Calculates a source thread ID by performing a bitwise XOR of the caller's thread ID with mask: the value of var held by the resulting thread ID is returned. If width is less than warpSize then each group of width consecutive threads are able to access elements from earlier groups of threads, however if they attempt to access elements from later groups of threads their own value of var will be returned. This mode implements a butterfly addressing pattern such as is used in tree reduction and broadcast.



#### **Note:**

For more details for this function see the Warp Shuffle Functions section in the CUDA C++ Programming Guide.

## `__device__ shfl_xor_sync (const unsigned mask, const __half2 var, const int delta, const int width)`

Exchange a variable between threads within a warp. Copy from a thread based on bitwise XOR of own thread ID.

### Parameters

#### **mask**

- unsigned int. Is only being read.

#### **var**

- half2. Is only being read.

#### **delta**

- int. Is only being read.

#### **width**

- int. Is only being read.

### Returns

Returns the 4-byte word referenced by var from the source thread ID as half2. If the source thread ID is out of range or the source thread has exited, the calling thread's own var is returned.

### Description

Calculates a source thread ID by performing a bitwise XOR of the caller's thread ID with mask: the value of var held by the resulting thread ID is returned. If width is less than warpSize then each group of width consecutive threads are able to access elements from earlier groups of threads, however if they attempt to access elements from later groups of threads their own value of var will be returned. This mode implements a butterfly addressing pattern such as is used in tree reduction and broadcast.



#### **Note:**

For more details for this function see the Warp Shuffle Functions section in the CUDA C++ Programming Guide.

## `__host__ __device__ short2half_rd (const short int i)`

Convert a signed short integer to a half in round-down mode.

### Parameters

#### **i**

- short int. Is only being read.

## Returns

half

- ▶ `i` converted to half.

## Description

Convert the signed short integer value `i` to a half-precision floating-point value in round-down mode.

## `__host__ __device__ short2half_rn (const short int i)`

Convert a signed short integer to a half in round-to-nearest-even mode.

## Parameters

**i**

- short int. Is only being read.

## Returns

half

- ▶ `i` converted to half.

## Description

Convert the signed short integer value `i` to a half-precision floating-point value in round-to-nearest-even mode.

## `__host__ __device__ short2half_ru (const short int i)`

Convert a signed short integer to a half in round-up mode.

## Parameters

**i**

- short int. Is only being read.

## Returns

half

- ▶ `i` converted to half.

## Description

Convert the signed short integer value `i` to a half-precision floating-point value in round-up mode.

## `__host__ device __short2half_rz (const short int i)`

Convert a signed short integer to a half in round-towards-zero mode.

### Parameters

**i**

- short int. Is only being read.

### Returns

half

- ▶ `i` converted to half.

### Description

Convert the signed short integer value `i` to a half-precision floating-point value in round-towards-zero mode.

## `__host__ device __short_as_half (const short int i)`

Reinterprets bits in a signed short integer as a `half`.

### Parameters

**i**

- short int. Is only being read.

### Returns

half

- ▶ The reinterpreted value.

### Description

Reinterprets the bits in the signed short integer `i` as a half-precision floating-point number.

## `__device__ void __stcg (const __half *ptr, const __half value)`

Generates a ``st.global.cg`` store instruction.

### Parameters

**ptr**

- memory location

**value**

- the value to be stored



```
__device__ void __stcg (const __half2 *ptr, const __half2 value)
```

Generates a `st.global.cg` store instruction.

### Parameters

#### **ptr**

- memory location

#### **value**

- the value to be stored

```
__device__ void __stcs (const __half *ptr, const __half value)
```

Generates a `st.global.cs` store instruction.

### Parameters

#### **ptr**

- memory location

#### **value**

- the value to be stored

```
__device__ void __stcs (const __half2 *ptr, const __half2 value)
```

Generates a `st.global.cs` store instruction.

### Parameters

#### **ptr**

- memory location

#### **value**

- the value to be stored

```
__device__ void __stwb (const __half *ptr, const __half value)
```

Generates a `st.global.wb` store instruction.

### Parameters

#### **ptr**

- memory location

#### **value**

- the value to be stored

```
__device__ void __stwb (const __half2 *ptr, const __half2 value)
```

Generates a `st.global.wb` store instruction.

### Parameters

#### **ptr**

- memory location

#### **value**

- the value to be stored

```
__device__ void __stwt (const __half *ptr, const __half value)
```

Generates a `st.global.wt` store instruction.

### Parameters

#### **ptr**

- memory location

#### **value**

- the value to be stored

```
__device__ void __stwt (const __half2 *ptr, const __half2 value)
```

Generates a `st.global.wt` store instruction.

### Parameters

#### **ptr**

- memory location

#### **value**

- the value to be stored

```
__host__ __device__ uint2half_rd (const unsigned int i)
```

Convert an unsigned integer to a half in round-down mode.

### Parameters

#### **i**

- unsigned int. Is only being read.

### Returns

half

- ▶ `i` converted to half.

### Description

Convert the unsigned integer value `i` to a half-precision floating-point value in round-down mode.

`__host__ __device__ uint2half_rn (const unsigned int i)`

Convert an unsigned integer to a half in round-to-nearest-even mode.

### Parameters

**i**

- unsigned int. Is only being read.

### Returns

half

- ▶ `i` converted to half.

### Description

Convert the unsigned integer value `i` to a half-precision floating-point value in round-to-nearest-even mode.

`__host__ __device__ uint2half_ru (const unsigned int i)`

Convert an unsigned integer to a half in round-up mode.

### Parameters

**i**

- unsigned int. Is only being read.

### Returns

half

- ▶ `i` converted to half.

### Description

Convert the unsigned integer value `i` to a half-precision floating-point value in round-up mode.

## `__host__ device __uint2half_rz (const unsigned int i)`

Convert an unsigned integer to a half in round-towards-zero mode.

### Parameters

**i**  
- unsigned int. Is only being read.

### Returns

half

- ▶ `i` converted to half.

### Description

Convert the unsigned integer value `i` to a half-precision floating-point value in round-towards-zero mode.

## `__host__ device __ull2half_rd (const unsigned long long int i)`

Convert an unsigned 64-bit integer to a half in round-down mode.

### Parameters

**i**  
- unsigned long long int. Is only being read.

### Returns

half

- ▶ `i` converted to half.

### Description

Convert the unsigned 64-bit integer value `i` to a half-precision floating-point value in round-down mode.

## `__host__ device __ull2half_rn (const unsigned long long int i)`

Convert an unsigned 64-bit integer to a half in round-to-nearest-even mode.

### Parameters

**i**  
- unsigned long long int. Is only being read.

## Returns

half

- ▶ `i` converted to half.

## Description

Convert the unsigned 64-bit integer value `i` to a half-precision floating-point value in round-to-nearest-even mode.

`__host__ __device__ ull2half_ru (const unsigned long long int i)`

Convert an unsigned 64-bit integer to a half in round-up mode.

## Parameters

**i**

- unsigned long long int. Is only being read.

## Returns

half

- ▶ `i` converted to half.

## Description

Convert the unsigned 64-bit integer value `i` to a half-precision floating-point value in round-up mode.

`__host__ __device__ ull2half_rz (const unsigned long long int i)`

Convert an unsigned 64-bit integer to a half in round-towards-zero mode.

## Parameters

**i**

- unsigned long long int. Is only being read.

## Returns

half

- ▶ `i` converted to half.

## Description

Convert the unsigned 64-bit integer value `i` to a half-precision floating-point value in round-towards-zero mode.

`__host__ __device__ ushort2half_rd (const unsigned short int i)`

Convert an unsigned short integer to a half in round-down mode.

## Parameters

**i**

- unsigned short int. Is only being read.

## Returns

half

- ▶ `i` converted to half.

## Description

Convert the unsigned short integer value `i` to a half-precision floating-point value in round-down mode.

`__host__ __device__ ushort2half_rn (const unsigned short int i)`

Convert an unsigned short integer to a half in round-to-nearest-even mode.

## Parameters

**i**

- unsigned short int. Is only being read.

## Returns

half

- ▶ `i` converted to half.

## Description

Convert the unsigned short integer value `i` to a half-precision floating-point value in round-to-nearest-even mode.

## `__host__ __device__ ushort2half_ru (const unsigned short int i)`

Convert an unsigned short integer to a half in round-up mode.

### Parameters

**i**

- unsigned short int. Is only being read.

### Returns

half

► *i* converted to half.

### Description

Convert the unsigned short integer value *i* to a half-precision floating-point value in round-up mode.

## `__host__ __device__ ushort2half_rz (const unsigned short int i)`

Convert an unsigned short integer to a half in round-towards-zero mode.

### Parameters

**i**

- unsigned short int. Is only being read.

### Returns

half

► *i* converted to half.

### Description

Convert the unsigned short integer value *i* to a half-precision floating-point value in round-towards-zero mode.

## `__host__ __device__ ushort_as_half (const unsigned short int i)`

Reinterprets bits in an unsigned short integer as a `half`.

### Parameters

**i**

- unsigned short int. Is only being read.

### Returns

`half`

► The reinterpreted value.

### Description

Reinterprets the bits in the unsigned short integer `i` as a half-precision floating-point number.

## `__host__ __device__ __make_half2 (const __half x, const __half y)`

Vector function, combines two `__half` numbers into one `__half2` number.

### Parameters

**x**

- `half`. Is only being read.

**y**

- `half`. Is only being read.

### Returns

`__half2`

► The `__half2` vector with one half equal to `x` and the other to `y`.

### Description

Combines two input `__half` number `x` and `y` into one `__half2` number. Input `x` is stored in low 16 bits of the return value, input `y` is stored in high 16 bits of the return value.

## 1.2.7. Half Math Functions

Half Precision Intrinsics

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



## `__device__ hceil (const __half h)`

Calculate ceiling of the input argument.

### Parameters

**h**

- half. Is only being read.

### Returns

half

- ▶ The smallest integer value not less than h.

### Description

Compute the smallest integer value not less than h.

## `__device__ hcos (const __half a)`

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

### Parameters

**a**

- half. Is only being read.

### Returns

half

- ▶ The cosine of a.

### Description

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

## `__device__ hexp (const __half a)`

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

### Parameters

**a**

- half. Is only being read.

### Returns

half

- ▶ The natural exponential function on a.

## Description

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

### `__device__ hexp10 (const __half a)`

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

## Parameters

**a**

- `half`. Is only being read.

## Returns

`half`

- The decimal exponential function on `a`.

## Description

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

### `__device__ hexp2 (const __half a)`

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

## Parameters

**a**

- `half`. Is only being read.

## Returns

`half`

- The binary exponential function on `a`.

## Description

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

### `__device__ hfloor (const __half h)`

Calculate the largest integer less than or equal to `h`.

## Parameters

**h**

- `half`. Is only being read.

## Returns

half

- ▶ The largest integer value which is less than or equal to  $h$ .

## Description

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

## `__device__ hlog (const __half a)`

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

## Parameters

**a**

- `half`. Is only being read.

## Returns

half

- ▶ The natural logarithm of  $a$ .

## Description

Calculates `half` natural logarithm of input  $a$  in round-to-nearest-even mode.

## `__device__ hlog10 (const __half a)`

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

## Parameters

**a**

- `half`. Is only being read.

## Returns

half

- ▶ The decimal logarithm of  $a$ .

## Description

Calculates `half` decimal logarithm of input  $a$  in round-to-nearest-even mode.

## `__device__ hlog2 (const __half a)`

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

### Parameters

**a**

- `half`. Is only being read.

### Returns

`half`

- ▶ The binary logarithm of `a`.

### Description

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

## `__device__ hrcp (const __half a)`

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

### Parameters

**a**

- `half`. Is only being read.

### Returns

`half`

- ▶ The reciprocal of `a`.

### Description

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

## `__device__ hrint (const __half h)`

Round input to nearest integer value in half-precision floating-point number.

### Parameters

**h**

- `half`. Is only being read.

### Returns

`half`

- ▶ The nearest integer to `h`.

## Description

Round `h` to the nearest integer value in half-precision floating-point format, with halfway cases rounded to the nearest even integer value.

## `__device__ hrsqrt (const __half a)`

Calculates `half` reciprocal square root in round-to-nearest-even mode.

## Parameters

**a**

- `half`. Is only being read.

## Returns

`half`

- The reciprocal square root of `a`.

## Description

Calculates `half` reciprocal square root of input `a` in round-to-nearest-even mode.

## `__device__ hsin (const __half a)`

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

## Parameters

**a**

- `half`. Is only being read.

## Returns

`half`

- The sine of `a`.

## Description

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

## `__device__ hsqrt (const __half a)`

Calculates `half` square root in round-to-nearest-even mode.

### Parameters

**a**

- `half`. Is only being read.

### Returns

`half`

- ▶ The square root of `a`.

### Description

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

## `__device__ htrunc (const __half h)`

Truncate input argument to the integral part.

### Parameters

**h**

- `half`. Is only being read.

### Returns

`half`

- ▶ The truncated integer value.

### Description

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

## 1.2.8. Half2 Math Functions

Half Precision Intrinsics

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

## `__device__ h2ceil (const __half2 h)`

Calculate `half2` vector ceiling of the input argument.

### Parameters

**h**

- `half2`. Is only being read.

### Returns

`half2`

- ▶ The vector of smallest integers not less than `h`.

### Description

For each component of vector `h` compute the smallest integer value not less than `h`.

## `__device__ h2cos (const __half2 a)`

Calculates `half2` vector cosine in round-to-nearest-even mode.

### Parameters

**a**

- `half2`. Is only being read.

### Returns

`half2`

- ▶ The elementwise cosine on vector `a`.

### Description

Calculates `half2` cosine of input vector `a` in round-to-nearest-even mode.

## `__device__ h2exp (const __half2 a)`

Calculates `half2` vector exponential function in round-to-nearest-even mode.

### Parameters

**a**

- `half2`. Is only being read.

### Returns

`half2`

- ▶ The elementwise exponential function on vector `a`.

## Description

Calculates `half2` exponential function of input vector `a` in round-to-nearest-even mode.

## `__device__ h2exp10 (const __half2 a)`

Calculates `half2` vector decimal exponential function in round-to-nearest-even mode.

## Parameters

**a**

- `half2`. Is only being read.

## Returns

`half2`

- The elementwise decimal exponential function on vector `a`.

## Description

Calculates `half2` decimal exponential function of input vector `a` in round-to-nearest-even mode.

## `__device__ h2exp2 (const __half2 a)`

Calculates `half2` vector binary exponential function in round-to-nearest-even mode.

## Parameters

**a**

- `half2`. Is only being read.

## Returns

`half2`

- The elementwise binary exponential function on vector `a`.

## Description

Calculates `half2` binary exponential function of input vector `a` in round-to-nearest-even mode.



## `__device__ h2floor (const __half2 h)`

Calculate the largest integer less than or equal to  $h$ .

### Parameters

**h**

- half2. Is only being read.

### Returns

half2

- The vector of largest integers which is less than or equal to  $h$ .

### Description

For each component of vector  $h$  calculate the largest integer value which is less than or equal to  $h$ .

## `__device__ h2log (const __half2 a)`

Calculates `half2` vector natural logarithm in round-to-nearest-even mode.

### Parameters

**a**

- half2. Is only being read.

### Returns

half2

- The elementwise natural logarithm on vector  $a$ .

### Description

Calculates `half2` natural logarithm of input vector  $a$  in round-to-nearest-even mode.

## `__device__ h2log10 (const __half2 a)`

Calculates `half2` vector decimal logarithm in round-to-nearest-even mode.

### Parameters

**a**

- half2. Is only being read.

### Returns

half2

- ▶ The elementwise decimal logarithm on vector a.

### Description

Calculates `half2` decimal logarithm of input vector a in round-to-nearest-even mode.

### `__device__ h2log2 (const __half2 a)`

Calculates `half2` vector binary logarithm in round-to-nearest-even mode.

### Parameters

**a**

- `half2`. Is only being read.

### Returns

`half2`

- ▶ The elementwise binary logarithm on vector a.

### Description

Calculates `half2` binary logarithm of input vector a in round-to-nearest-even mode.

### `__device__ h2rcp (const __half2 a)`

Calculates `half2` vector reciprocal in round-to-nearest-even mode.

### Parameters

**a**

- `half2`. Is only being read.

### Returns

`half2`

- ▶ The elementwise reciprocal on vector a.

### Description

Calculates `half2` reciprocal of input vector a in round-to-nearest-even mode.

## `__device__ h2rint (const __half2 h)`

Round input to nearest integer value in half-precision floating-point number.

### Parameters

**h**

- half2. Is only being read.

### Returns

half2

- The vector of rounded integer values.

### Description

Round each component of `half2` vector `h` to the nearest integer value in half-precision floating-point format, with halfway cases rounded to the nearest even integer value.

## `__device__ h2rsqrt (const __half2 a)`

Calculates `half2` vector reciprocal square root in round-to-nearest-even mode.

### Parameters

**a**

- half2. Is only being read.

### Returns

half2

- The elementwise reciprocal square root on vector `a`.

### Description

Calculates `half2` reciprocal square root of input vector `a` in round-to-nearest-even mode.

## `__device__ h2sin (const __half2 a)`

Calculates `half2` vector sine in round-to-nearest-even mode.

### Parameters

**a**

- half2. Is only being read.

### Returns

half2

- ▶ The elementwise sine on vector `a`.

### Description

Calculates `half2` sine of input vector `a` in round-to-nearest-even mode.

### `__device__ h2sqrt (const __half2 a)`

Calculates `half2` vector square root in round-to-nearest-even mode.

### Parameters

**a**

- `half2`. Is only being read.

### Returns

`half2`

- ▶ The elementwise square root on vector `a`.

### Description

Calculates `half2` square root of input vector `a` in round-to-nearest-even mode.

### `__device__ h2trunc (const __half2 h)`

Truncate `half2` vector input argument to the integral part.

### Parameters

**h**

- `half2`. Is only being read.

### Returns

`half2`

- ▶ The truncated `h`.

### Description

Round each component of vector `h` to the nearest integer value that does not exceed `h` in magnitude.

## 1.3. Bfloat16 Precision Intrinsic

This section describes `nv_bfloat16` precision intrinsic functions. To use these functions, include the header file `cuda_bf16.h` in your program. All of the functions defined here are

available in device code. Some of the functions are also available to host compilers, please refer to respective functions' documentation for details.

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

- ▶ `hsin( __nv_bfloat16);`
- ▶ `hcos( __nv_bfloat16);`
- ▶ `h2sin( __nv_bfloat162);`
- ▶ `h2cos( __nv_bfloat162);`

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

- ▶ `CUDA_NO_BFLOAT16` - If defined, this macro will prevent the definition of additional type aliases in the global namespace, helping to avoid potential conflicts with symbols defined in the user program.
- ▶ `__CUDA_NO_BFLOAT16_CONVERSIONS__` - If defined, this macro will prevent the use of the C++ type conversions (converting constructors and conversion operators) that are common for built-in floating-point types, but may be undesirable for `__nv_bfloat16` which is essentially a user-defined type.
- ▶ `__CUDA_NO_BFLOAT16_OPERATORS__` and `__CUDA_NO_BFLOAT162_OPERATORS__` - If defined, these macros will prevent the inadvertent use of usual arithmetic and comparison operators. This enforces the storage-only type semantics and prevents C++ style computations on `__nv_bfloat16` and `__nv_bfloat162` types.

## struct `__nv_bfloat16`

`nv_bfloat16` datatype

## struct `__nv_bfloat162`

`nv_bfloat162` datatype

## struct `__nv_bfloat162_raw`

`__nv_bfloat162_raw` data type

## struct `__nv_bfloat16_raw`

`__nv_bfloat16_raw` data type

## Bfloat16 Arithmetic Constants

## Bfloat16 Arithmetic Functions

## Bfloat162 Arithmetic Functions

## Bfloat16 Comparison Functions

## Bfloat162 Comparison Functions

## Bfloat16 Precision Conversion and Data Movement

## Bfloat16 Math Functions

## Bfloat162 Math Functions

### `typedef nv_bfloat16`

This datatype is meant to be the first-class or fundamental implementation of the bfloat16 numbers format.

Should be implemented in the compiler in the future. Current implementation is a simple typedef to a respective user-level type with underscores.

### `typedef nv_bfloat162`

This datatype is meant to be the first-class or fundamental implementation of type for pairs of bfloat16 numbers.

Should be implemented in the compiler in the future. Current implementation is a simple typedef to a respective user-level type with underscores.

### 1.3.1. Bfloat16 Arithmetic Constants

Bfloat16 Precision Ininsics

To use these constants, include the header file `cuda_bf16.h` in your program.

```
#define CUDART_INF_BF16  
__ushort_as_bfloat16((unsigned short)0x7F80U)
```

Defines floating-point positive infinity value for the `nv_bfloat16` data type.

```
#define CUDART_MAX_NORMAL_BF16
__ushort_as_bfloat16((unsigned short)0x7F7FU)
```

Defines a maximum representable value for the `nv_bfloat16` data type.

```
#define CUDART_MIN_DENORM_BF16
__ushort_as_bfloat16((unsigned short)0x0001U)
```

Defines a minimum representable (denormalized) value for the `nv_bfloat16` data type.

```
#define CUDART_NAN_BF16
__ushort_as_bfloat16((unsigned short)0x7FFFU)
```

Defines canonical NaN value for the `nv_bfloat16` data type.

```
#define CUDART_NEG_ZERO_BF16
__ushort_as_bfloat16((unsigned short)0x8000U)
```

Defines a negative zero value for the `nv_bfloat16` data type.

```
#define CUDART_ONE_BF16
__ushort_as_bfloat16((unsigned short)0x3F80U)
```

Defines a value of 1.0 for the `nv_bfloat16` data type.

```
#define CUDART_ZERO_BF16
__ushort_as_bfloat16((unsigned short)0x0000U)
```

Defines a positive zero value for the `nv_bfloat16` data type.

### 1.3.2. Bfloat16 Arithmetic Functions

#### Bfloat16 Precision Intrinsics

To use these functions, include the header file `cuda_bf16.h` in your program.

```
__host__ __device__ __h2div (const __nv_bfloat162 a, const
__nv_bfloat162 b)
```

Performs `nv_bfloat162` vector division in round-to-nearest-even mode.

#### Description

Divides `nv_bfloat162` input vector `a` by input vector `b` in round-to-nearest-even mode.

`__host__ __device__ fabs (const __nv_bfloat16 a)`

Calculates the absolute value of input `nv_bfloat16` number and returns the result.

### Parameters

**a**

- `nv_bfloat16`. Is only being read.

### Returns

`nv_bfloat16`

► The absolute value of `a`.

### Description

Calculates the absolute value of input `nv_bfloat16` number and returns the result.

`__host__ __device__ hadd (const __nv_bfloat16 a, const __nv_bfloat16 b)`

Performs `nv_bfloat16` addition in round-to-nearest-even mode.

### Description

Performs `nv_bfloat16` addition of inputs `a` and `b`, in round-to-nearest-even mode.

`__host__ __device__ hadd_rn (const __nv_bfloat16 a, const __nv_bfloat16 b)`

Performs `nv_bfloat16` addition in round-to-nearest-even mode.

### Description

Performs `nv_bfloat16` addition of inputs `a` and `b`, in round-to-nearest-even mode. Prevents floating-point contractions of `mul+add` into `fma`.

`__host__ __device__ hadd_sat (const __nv_bfloat16 a, const __nv_bfloat16 b)`

Performs `nv_bfloat16` addition in round-to-nearest-even mode, with saturation to `[0.0, 1.0]`.

### Parameters

**a**

- `nv_bfloat16`. Is only being read.

**b**

- `nv_bfloat16`. Is only being read.



## Returns

`nv_bfloat16`

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

## Description

Performs `nv_bfloat16` add of inputs `a` and `b`, in round-to-nearest-even mode, and clamps the result to range `[0.0, 1.0]`. NaN results are flushed to `+0.0`.

```
__host__ __device__ hdiv (const __nv_bfloat16 a, const
__nv_bfloat16 b)
```

Performs `nv_bfloat16` division in round-to-nearest-even mode.

## Description

Divides `nv_bfloat16` input `a` by input `b` in round-to-nearest-even mode.

```
__device__ hfma (const __nv_bfloat16 a, const
__nv_bfloat16 b, const __nv_bfloat16 c)
```

Performs `nv_bfloat16` fused multiply-add in round-to-nearest-even mode.

## Description

Performs `nv_bfloat16` multiply on inputs `a` and `b`, then performs a `nv_bfloat16` add of the result with `c`, rounding the result once in round-to-nearest-even mode.

```
__device__ hfma_relu (const __nv_bfloat16 a, const
__nv_bfloat16 b, const __nv_bfloat16 c)
```

Performs `nv_bfloat16` fused multiply-add in round-to-nearest-even mode with relu saturation.

## Parameters

- a**
  - `nv_bfloat16`. Is only being read.
- b**
  - `nv_bfloat16`. Is only being read.
- c**
  - `nv_bfloat16`. Is only being read.

## Returns

`nv_bfloat16`

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

### Description

Performs `nv_bfloat16` multiply on inputs *a* and *b*, then performs a `nv_bfloat16` add of the result with *c*, rounding the result once in round-to-nearest-even mode. Then negative result is clamped to 0. NaN result is converted to canonical NaN.

```
__device__ hfma_sat (const __nv_bfloat16 a, const
__nv_bfloat16 b, const __nv_bfloat16 c)
```

Performs `nv_bfloat16` fused multiply-add in round-to-nearest-even mode, with saturation to [0.0, 1.0].

### Parameters

**a**

- `nv_bfloat16`. Is only being read.

**b**

- `nv_bfloat16`. Is only being read.

**c**

- `nv_bfloat16`. Is only being read.

### Returns

`nv_bfloat16`

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

### Description

Performs `nv_bfloat16` multiply on inputs *a* and *b*, then performs a `nv_bfloat16` add of the result with *c*, rounding the result once in round-to-nearest-even mode, and clamps the result to range [0.0, 1.0]. NaN results are flushed to +0.0.

```
__host__ __device__ hmul (const __nv_bfloat16 a, const
__nv_bfloat16 b)
```

Performs `nv_bfloat16` multiplication in round-to-nearest-even mode.

### Description

Performs `nv_bfloat16` multiplication of inputs *a* and *b*, in round-to-nearest-even mode.

`__host__ __device__ hmul_rn (const __nv_bfloat16 a, const __nv_bfloat16 b)`

Performs `nv_bfloat16` multiplication in round-to-nearest-even mode.

### Description

Performs `nv_bfloat16` multiplication of inputs `a` and `b`, in round-to-nearest-even mode. Prevents floating-point contractions of `mul+add` or `sub` into `fma`.

`__host__ __device__ hmul_sat (const __nv_bfloat16 a, const __nv_bfloat16 b)`

Performs `nv_bfloat16` multiplication in round-to-nearest-even mode, with saturation to `[0.0, 1.0]`.

### Parameters

**a**

- `nv_bfloat16`. Is only being read.

**b**

- `nv_bfloat16`. Is only being read.

### Returns

`nv_bfloat16`

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

### Description

Performs `nv_bfloat16` multiplication of inputs `a` and `b`, in round-to-nearest-even mode, and clamps the result to range `[0.0, 1.0]`. NaN results are flushed to `+0.0`.

`__host__ __device__ hneg (const __nv_bfloat16 a)`

Negates input `nv_bfloat16` number and returns the result.

### Description

Negates input `nv_bfloat16` number and returns the result.

```
__host__ device__ hsub (const __nv_bfloat16 a, const
__nv_bfloat16 b)
```

Performs `nv_bfloat16` subtraction in round-to-nearest-even mode.

### Description

Subtracts `nv_bfloat16` input `b` from input `a` in round-to-nearest-even mode.

```
__host__ device__ hsub_rn (const __nv_bfloat16 a, const
__nv_bfloat16 b)
```

Performs `nv_bfloat16` subtraction in round-to-nearest-even mode.

### Description

Subtracts `nv_bfloat16` input `b` from input `a` in round-to-nearest-even mode. Prevents floating-point contractions of `mul+sub` into `fma`.

```
__host__ device__ hsub_sat (const __nv_bfloat16 a,
const __nv_bfloat16 b)
```

Performs `nv_bfloat16` subtraction in round-to-nearest-even mode, with saturation to `[0.0, 1.0]`.

### Parameters

- a**
  - `nv_bfloat16`. Is only being read.
- b**
  - `nv_bfloat16`. Is only being read.

### Returns

`nv_bfloat16`

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

### Description

Subtracts `nv_bfloat16` input `b` from input `a` in round-to-nearest-even mode, and clamps the result to range `[0.0, 1.0]`. NaN results are flushed to `+0.0`.

`__device__ atomicAdd (const __nv_bfloat16 *address, const __nv_bfloat16 val)`

Adds `val` to the value stored at `address` in global or shared memory, and writes this value back to `address`. This operation is performed in one atomic operation.

### Parameters

#### **address**

- `__nv_bfloat16*`. An address in global or shared memory.

#### **val**

- `__nv_bfloat16`. The value to be added.

### Returns

`__nv_bfloat16`

- The old value read from `address`.

### Description

The location of `address` must be in global or shared memory. This operation has undefined behavior otherwise. This operation is natively supported by devices of compute capability 9.x and higher, older devices of compute capability 7.x and 8.x use emulation path.



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

`__host__ __device__ operator* (const __nv_bfloat16 lh, const __nv_bfloat16 rh)`

### Description

Performs `nv_bfloat16` multiplication operation. See also `__hmul(__nv_bfloat16, __nv_bfloat16)`

`__host__ __device__ operator*= (__nv_bfloat16 lh, const __nv_bfloat16 rh)`

### Description

Performs `nv_bfloat16` compound assignment with multiplication operation.

`__host__ __device__ operator+ (const __nv_bfloat16 h)`

### Description

Implements `nv_bfloat16` unary plus operator, returns input value.

`__host__ __device__ operator+ (const __nv_bfloat16 lh, const __nv_bfloat16 rh)`

### Description

Performs `nv_bfloat16` addition operation. See also `__hadd(__nv_bfloat16, __nv_bfloat16)`

`__host__ __device__ operator++ (__nv_bfloat16 h, const int ignored)`

### Description

Performs `nv_bfloat16` postfix increment operation.

`__host__ __device__ operator++ (__nv_bfloat16 h)`

### Description

Performs `nv_bfloat16` prefix increment operation.

`__host__ __device__ operator+= (__nv_bfloat16 lh, const __nv_bfloat16 rh)`

### Description

Performs `nv_bfloat16` compound assignment with addition operation.

`__host__ __device__ operator- (const __nv_bfloat16 h)`

### Description

Implements `nv_bfloat16` unary minus operator. See also `__hneg(__nv_bfloat16)`

`__host__ device __operator- (const __nv_bfloat16 lh, const __nv_bfloat16 rh)`

### Description

Performs `nv_bfloat16` subtraction operation. See also [\\_\\_hsub\(\\_\\_nv\\_bfloat16, \\_\\_nv\\_bfloat16\)](#)

`__host__ device __operator-- (__nv_bfloat16 h, const int ignored)`

### Description

Performs `nv_bfloat16` postfix decrement operation.

`__host__ device __operator-- (__nv_bfloat16 h)`

### Description

Performs `nv_bfloat16` prefix decrement operation.

`__host__ device __operator-= (__nv_bfloat16 lh, const __nv_bfloat16 rh)`

### Description

Performs `nv_bfloat16` compound assignment with subtraction operation.

`__host__ device __operator/ (const __nv_bfloat16 lh, const __nv_bfloat16 rh)`

### Description

Performs `nv_bfloat16` division operation. See also [\\_\\_hdiv\(\\_\\_nv\\_bfloat16, \\_\\_nv\\_bfloat16\)](#)

`__host__ device __operator/= (__nv_bfloat16 lh, const __nv_bfloat16 rh)`

### Description

Performs `nv_bfloat16` compound assignment with division operation.

### 1.3.3. Bfloat162 Arithmetic Functions

Bfloat16 Precision Intrinsic

To use these functions, include the header file `cuda_bf16.h` in your program.

`__host__ __device__ habs2 (const __nv_bfloat162 a)`

Calculates the absolute value of both halves of the input `nv_bfloat162` number and returns the result.

#### Parameters

**a**

- `nv_bfloat162`. Is only being read.

#### Returns

`bfloat2`

- Returns `a` with the absolute value of both halves.

#### Description

Calculates the absolute value of both halves of the input `nv_bfloat162` number and returns the result.

`__host__ __device__ hadd2 (const __nv_bfloat162 a, const __nv_bfloat162 b)`

Performs `nv_bfloat162` vector addition in round-to-nearest-even mode.

#### Description

Performs `nv_bfloat162` vector add of inputs `a` and `b`, in round-to-nearest-even mode.

`__host__ __device__ hadd2_rn (const __nv_bfloat162 a, const __nv_bfloat162 b)`

Performs `nv_bfloat162` vector addition in round-to-nearest-even mode.

#### Description

Performs `nv_bfloat162` vector add of inputs `a` and `b`, in round-to-nearest-even mode. Prevents floating-point contractions of `mul+add` into `fma`.



`__host__ __device__ hadd2_sat (const __nv_bfloat162 a, const __nv_bfloat162 b)`

Performs `nv_bfloat162` vector addition in round-to-nearest-even mode, with saturation to `[0.0, 1.0]`.

### Parameters

**a**

- `nv_bfloat162`. Is only being read.

**b**

- `nv_bfloat162`. Is only being read.

### Returns

`nv_bfloat162`

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

### Description

Performs `nv_bfloat162` vector add of inputs `a` and `b`, in round-to-nearest-even mode, and clamps the results to range `[0.0, 1.0]`. NaN results are flushed to `+0.0`.

`__device__ hcmadd (const __nv_bfloat162 a, const __nv_bfloat162 b, const __nv_bfloat162 c)`

Performs fast complex multiply-accumulate.

### Parameters

**a**

- `nv_bfloat162`. Is only being read.

**b**

- `nv_bfloat162`. Is only being read.

**c**

- `nv_bfloat162`. Is only being read.

### Returns

`nv_bfloat162`

► The result of complex multiply-accumulate operation on complex numbers `a`, `b`, and `c`

### Description

Interprets vector `nv_bfloat162` input pairs `a`, `b`, and `c` as complex numbers in `nv_bfloat16` precision and performs complex multiply-accumulate operation: `a*b + c`

`__device__ hfma2 (const __nv_bfloat162 a, const __nv_bfloat162 b, const __nv_bfloat162 c)`

Performs `nv_bfloat162` vector fused multiply-add in round-to-nearest-even mode.

### Description

Performs `nv_bfloat162` vector multiply on inputs `a` and `b`, then performs a `nv_bfloat162` vector add of the result with `c`, rounding the result once in round-to-nearest-even mode.

`__device__ hfma2_relu (const __nv_bfloat162 a, const __nv_bfloat162 b, const __nv_bfloat162 c)`

Performs `nv_bfloat162` vector fused multiply-add in round-to-nearest-even mode with relu saturation.

### Parameters

- a**
  - `nv_bfloat162`. Is only being read.
- b**
  - `nv_bfloat162`. Is only being read.
- c**
  - `nv_bfloat162`. Is only being read.

### Returns

`nv_bfloat162`

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

### Description

Performs `nv_bfloat162` vector multiply on inputs `a` and `b`, then performs a `nv_bfloat162` vector add of the result with `c`, rounding the result once in round-to-nearest-even mode. Then negative result is clamped to 0. NaN result is converted to canonical NaN.

`__device__ hfma2_sat (const __nv_bfloat162 a, const __nv_bfloat162 b, const __nv_bfloat162 c)`

Performs `nv_bfloat162` vector fused multiply-add in round-to-nearest-even mode, with saturation to `[0.0, 1.0]`.

### Parameters

- a**
  - `nv_bfloat162`. Is only being read.

**b**

- nv\_bfloat162. Is only being read.

**c**

- nv\_bfloat162. Is only being read.

**Returns**

nv\_bfloat162

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

**Description**

Performs nv\_bfloat162 vector multiply on inputs a and b, then performs a nv\_bfloat162 vector add of the result with c, rounding the result once in round-to-nearest-even mode, and clamps the results to range [0.0, 1.0]. NaN results are flushed to +0.0.

```
__host__ __device__ hmul2 (const __nv_bfloat162 a, const
__nv_bfloat162 b)
```

Performs nv\_bfloat162 vector multiplication in round-to-nearest-even mode.

**Description**

Performs nv\_bfloat162 vector multiplication of inputs a and b, in round-to-nearest-even mode.

```
__host__ __device__ hmul2_rn (const __nv_bfloat162 a,
const __nv_bfloat162 b)
```

Performs nv\_bfloat162 vector multiplication in round-to-nearest-even mode.

**Description**

Performs nv\_bfloat162 vector multiplication of inputs a and b, in round-to-nearest-even mode. Prevents floating-point contractions of mul+add or sub into fma.

```
__host__ __device__ hmul2_sat (const __nv_bfloat162 a,
const __nv_bfloat162 b)
```

Performs nv\_bfloat162 vector multiplication in round-to-nearest-even mode, with saturation to [0.0, 1.0].

**Parameters****a**

- nv\_bfloat162. Is only being read.

**b**

- nv\_bfloat162. Is only being read.

**Returns**

nv\_bfloat162

- ▶ The result of elementwise multiplication of vectors a and b, with respect to saturation.

**Description**

Performs nv\_bfloat162 vector multiplication of inputs a and b, in round-to-nearest-even mode, and clamps the results to range [0.0, 1.0]. NaN results are flushed to +0.0.

**\_\_host\_\_ \_\_device\_\_ hneg2 (const \_\_nv\_bfloat162 a)**

Negates both halves of the input nv\_bfloat162 number and returns the result.

**Description**

Negates both halves of the input nv\_bfloat162 number a and returns the result.

**\_\_host\_\_ \_\_device\_\_ hsub2 (const \_\_nv\_bfloat162 a, const \_\_nv\_bfloat162 b)**

Performs nv\_bfloat162 vector subtraction in round-to-nearest-even mode.

**Description**

Subtracts nv\_bfloat162 input vector b from input vector a in round-to-nearest-even mode.

**\_\_host\_\_ \_\_device\_\_ hsub2\_rn (const \_\_nv\_bfloat162 a, const \_\_nv\_bfloat162 b)**

Performs nv\_bfloat162 vector subtraction in round-to-nearest-even mode.

**Description**

Subtracts nv\_bfloat162 input vector b from input vector a in round-to-nearest-even mode. Prevents floating-point contractions of mul+sub into fma.

`__host__ __device__ hsub2_sat (const __nv_bfloat162 a, const __nv_bfloat162 b)`

Performs `nv_bfloat162` vector subtraction in round-to-nearest-even mode, with saturation to `[0.0, 1.0]`.

### Parameters

#### **a**

- `nv_bfloat162`. Is only being read.

#### **b**

- `nv_bfloat162`. Is only being read.

### Returns

`nv_bfloat162`

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

### Description

Subtracts `nv_bfloat162` input vector `b` from input vector `a` in round-to-nearest-even mode, and clamps the results to range `[0.0, 1.0]`. NaN results are flushed to `+0.0`.

`__device__ atomicAdd (const __nv_bfloat162 *address, const __nv_bfloat162 val)`

Vector add `val` to the value stored at `address` in global or shared memory, and writes this value back to `address`. The atomicity of the add operation is guaranteed separately for each of the two `nv_bfloat16` elements; the entire `__nv_bfloat162` is not guaranteed to be atomic as a single 32-bit access.

### Parameters

#### **address**

- `__nv_bfloat162*`. An address in global or shared memory.

#### **val**

- `__nv_bfloat162`. The value to be added.

### Returns

`__nv_bfloat162`

- The old value read from `address`.

## Description

The location of `address` must be in global or shared memory. This operation has undefined behavior otherwise. This operation is natively supported by devices of compute capability 9.x and higher, older devices use emulation path.



### Note:

For more details for this function see the Atomic Functions section in the CUDA C++ Programming Guide.

```
__host__ __device__ operator* (const __nv_bfloat162 lh,
const __nv_bfloat162 rh)
```

## Description

Performs packed `nv_bfloat16` multiplication operation. See also [\\_\\_hmul2\(\\_\\_nv\\_bfloat162, \\_\\_nv\\_bfloat162\)](#)

```
__host__ __device__ operator*= (__nv_bfloat162 lh, const
__nv_bfloat162 rh)
```

## Description

Performs packed `nv_bfloat16` compound assignment with multiplication operation.

```
__host__ __device__ operator+ (const __nv_bfloat162 h)
```

## Description

Implements packed `nv_bfloat16` unary plus operator, returns input value.

```
__host__ __device__ operator+ (const __nv_bfloat162 lh,
const __nv_bfloat162 rh)
```

## Description

Performs packed `nv_bfloat16` addition operation. See also [\\_\\_hadd2\(\\_\\_nv\\_bfloat162, \\_\\_nv\\_bfloat162\)](#)

`__host__ __device__ operator++ (__nv_bfloat162 h, const int ignored)`

#### Description

Performs packed `nv_bfloat16` postfix increment operation.

`__host__ __device__ operator++ (__nv_bfloat162 h)`

#### Description

Performs packed `nv_bfloat16` prefix increment operation.

`__host__ __device__ operator+= (__nv_bfloat162 lh, const __nv_bfloat162 rh)`

#### Description

Performs packed `nv_bfloat16` compound assignment with addition operation.

`__host__ __device__ operator- (const __nv_bfloat162 h)`

#### Description

Implements packed `nv_bfloat16` unary minus operator. See also [\\_\\_hneg2\(\\_\\_nv\\_bfloat162\)](#)

`__host__ __device__ operator- (const __nv_bfloat162 lh, const __nv_bfloat162 rh)`

#### Description

Performs packed `nv_bfloat16` subtraction operation. See also [\\_\\_hsub2\(\\_\\_nv\\_bfloat162, \\_\\_nv\\_bfloat162\)](#)

`__host__ __device__ operator-- (__nv_bfloat162 h, const int ignored)`

#### Description

Performs packed `nv_bfloat16` postfix decrement operation.

`__host__ __device__ operator-- (__nv_bfloat162 h)`

#### Description

Performs packed `nv_bfloat16` prefix decrement operation.

`__host__ __device__ operator-= (__nv_bfloat162 lh, const __nv_bfloat162 rh)`

#### Description

Performs packed `nv_bfloat16` compound assignment with subtraction operation.

`__host__ __device__ operator/ (const __nv_bfloat162 lh, const __nv_bfloat162 rh)`

#### Description

Performs packed `nv_bfloat16` division operation. See also [h2div\(\\_\\_nv\\_bfloat162, \\_\\_nv\\_bfloat162\)](#)

`__host__ __device__ operator/= (__nv_bfloat162 lh, const __nv_bfloat162 rh)`

#### Description

Performs packed `nv_bfloat16` compound assignment with division operation.

### 1.3.4. Bfloat16 Comparison Functions

#### Bfloat16 Precision Intrinsics

To use these functions, include the header file `cuda_bf16.h` in your program.

`__host__ __device__ bool __heq (const __nv_bfloat16 a, const __nv_bfloat16 b)`

Performs `nv_bfloat16` if-equal comparison.

#### Parameters

**a**

- `nv_bfloat16`. Is only being read.

**b**

- `nv_bfloat16`. Is only being read.



## Returns

bool

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

## Description

Performs `nv_bfloat16` if-equal comparison of inputs a and b. NaN inputs generate false results.

```
__host__ __device__ bool __hequ (const __nv_bfloat16 a,
const __nv_bfloat16 b)
```

Performs `nv_bfloat16` unordered if-equal comparison.

## Parameters

**a**

- `nv_bfloat16`. Is only being read.

**b**

- `nv_bfloat16`. Is only being read.

## Returns

bool

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

## Description

Performs `nv_bfloat16` if-equal comparison of inputs a and b. NaN inputs generate true results.

```
__host__ __device__ bool __hge (const __nv_bfloat16 a,
const __nv_bfloat16 b)
```

Performs `nv_bfloat16` greater-equal comparison.

## Parameters

**a**

- `nv_bfloat16`. Is only being read.

**b**

- `nv_bfloat16`. Is only being read.

## Returns

bool

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

### Description

Performs `nv_bfloat16` greater-equal comparison of inputs a and b. NaN inputs generate false results.

```
__host__ __device__ bool __hgeu (const __nv_bfloat16 a,
const __nv_bfloat16 b)
```

Performs `nv_bfloat16` unordered greater-equal comparison.

### Parameters

**a**

- `nv_bfloat16`. Is only being read.

**b**

- `nv_bfloat16`. Is only being read.

### Returns

bool

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

### Description

Performs `nv_bfloat16` greater-equal comparison of inputs a and b. NaN inputs generate true results.

```
__host__ __device__ bool __hgt (const __nv_bfloat16 a,
const __nv_bfloat16 b)
```

Performs `nv_bfloat16` greater-than comparison.

### Parameters

**a**

- `nv_bfloat16`. Is only being read.

**b**

- `nv_bfloat16`. Is only being read.

### Returns

bool

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

## Description

Performs `nv_bfloat16` greater-than comparison of inputs `a` and `b`. NaN inputs generate false results.

```
__host__ __device__ bool __hgtu (const __nv_bfloat16 a,
const __nv_bfloat16 b)
```

Performs `nv_bfloat16` unordered greater-than comparison.

## Parameters

**a**

- `nv_bfloat16`. Is only being read.

**b**

- `nv_bfloat16`. Is only being read.

## Returns

`bool`

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

## Description

Performs `nv_bfloat16` greater-than comparison of inputs `a` and `b`. NaN inputs generate true results.

```
__host__ __device__ int __hisinf (const __nv_bfloat16 a)
```

Checks if the input `nv_bfloat16` number is infinite.

## Parameters

**a**

- `nv_bfloat16`. Is only being read.

## Returns

`int`

- ▶ -1 iff `a` is equal to negative infinity,
- ▶ 1 iff `a` is equal to positive infinity,
- ▶ 0 otherwise.

## Description

Checks if the input `nv_bfloat16` number `a` is infinite.

`__host__ __device__ bool __hisnan (const __nv_bfloat16 a)`

Determine whether `nv_bfloat16` argument is a NaN.

### Parameters

**a**

- `nv_bfloat16`. Is only being read.

### Returns

bool

► true iff argument is NaN.

### Description

Determine whether `nv_bfloat16` value `a` is a NaN.

`__host__ __device__ bool __hle (const __nv_bfloat16 a, const __nv_bfloat16 b)`

Performs `nv_bfloat16` less-equal comparison.

### Parameters

**a**

- `nv_bfloat16`. Is only being read.

**b**

- `nv_bfloat16`. Is only being read.

### Returns

bool

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

### Description

Performs `nv_bfloat16` less-equal comparison of inputs `a` and `b`. NaN inputs generate false results.

```
__host__ __device__ bool __hleu (const __nv_bfloat16 a,
const __nv_bfloat16 b)
```

Performs `nv_bfloat16` unordered less-equal comparison.

### Parameters

**a**

- `nv_bfloat16`. Is only being read.

**b**

- `nv_bfloat16`. Is only being read.

### Returns

`bool`

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

### Description

Performs `nv_bfloat16` less-equal comparison of inputs `a` and `b`. NaN inputs generate true results.

```
__host__ __device__ bool __hlt (const __nv_bfloat16 a, const
__nv_bfloat16 b)
```

Performs `nv_bfloat16` less-than comparison.

### Parameters

**a**

- `nv_bfloat16`. Is only being read.

**b**

- `nv_bfloat16`. Is only being read.

### Returns

`bool`

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

### Description

Performs `nv_bfloat16` less-than comparison of inputs `a` and `b`. NaN inputs generate false results.

`__host__ __device__ bool __hltu (const __nv_bfloat16 a, const __nv_bfloat16 b)`

Performs `nv_bfloat16` unordered less-than comparison.

### Parameters

**a**

- `nv_bfloat16`. Is only being read.

**b**

- `nv_bfloat16`. Is only being read.

### Returns

`bool`

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

### Description

Performs `nv_bfloat16` less-than comparison of inputs `a` and `b`. NaN inputs generate true results.

`__host__ __device__ __hmax (const __nv_bfloat16 a, const __nv_bfloat16 b)`

Calculates `nv_bfloat16` maximum of two input values.

### Description

Calculates `nv_bfloat16`  $\max(a, b)$  defined as  $(a > b) ? a : b$ .

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

`__host__ __device__ __hmax_nan (const __nv_bfloat16 a, const __nv_bfloat16 b)`

Calculates `nv_bfloat16` maximum of two input values, NaNs pass through.

### Description

Calculates `nv_bfloat16`  $\max(a, b)$  defined as  $(a > b) ? a : b$ .

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

`__host__ device__ hmin (const __nv_bfloat16 a, const __nv_bfloat16 b)`

Calculates `nv_bfloat16` minimum of two input values.

### Description

Calculates `nv_bfloat16` `min(a, b)` defined as `(a < b) ? a : b`.

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

`__host__ device__ hmin_nan (const __nv_bfloat16 a, const __nv_bfloat16 b)`

Calculates `nv_bfloat16` minimum of two input values, NaNs pass through.

### Description

Calculates `nv_bfloat16` `min(a, b)` defined as `(a < b) ? a : b`.

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

`__host__ device__ bool __hne (const __nv_bfloat16 a, const __nv_bfloat16 b)`

Performs `nv_bfloat16` not-equal comparison.

### Parameters

- a**
  - `nv_bfloat16`. Is only being read.
- b**
  - `nv_bfloat16`. Is only being read.

### Returns

`bool`

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

### Description

Performs `nv_bfloat16` not-equal comparison of inputs `a` and `b`. NaN inputs generate false results.

```
__host__ device__ bool __hneu (const __nv_bfloat16 a,
const __nv_bfloat16 b)
```

Performs `nv_bfloat16` unordered not-equal comparison.

### Parameters

**a**

- `nv_bfloat16`. Is only being read.

**b**

- `nv_bfloat16`. Is only being read.

### Returns

bool

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

### Description

Performs `nv_bfloat16` not-equal comparison of inputs a and b. NaN inputs generate true results.

```
__host__ device__ __CUDA_BF16_FORCEINLINE__ bool
operator!= (const __nv_bfloat16 lh, const __nv_bfloat16 rh)
```

### Description

Performs `nv_bfloat16` unordered compare not-equal operation. See also [\\_\\_hneu\(\\_\\_nv\\_bfloat16, \\_\\_nv\\_bfloat16\)](#)

```
__host__ device__ __CUDA_BF16_FORCEINLINE__ bool
operator< (const __nv_bfloat16 lh, const __nv_bfloat16 rh)
```

### Description

Performs `nv_bfloat16` ordered less-than compare operation. See also [\\_\\_hlt\(\\_\\_nv\\_bfloat16, \\_\\_nv\\_bfloat16\)](#)



```
__host__ device__ __CUDA_BF16_FORCEINLINE__ bool
operator<= (const __nv_bfloat16 lh, const __nv_bfloat16 rh)
```

### Description

Performs `nv_bfloat16` ordered less-or-equal compare operation. See also [hle\(`\_\_nv\_bfloat16`, `\_\_nv\_bfloat16`\)](#)

```
__host__ device__ __CUDA_BF16_FORCEINLINE__ bool
operator== (const __nv_bfloat16 lh, const __nv_bfloat16 rh)
```

### Description

Performs `nv_bfloat16` ordered compare equal operation. See also [heq\(`\_\_nv\_bfloat16`, `\_\_nv\_bfloat16`\)](#)

```
__host__ device__ __CUDA_BF16_FORCEINLINE__ bool
operator> (const __nv_bfloat16 lh, const __nv_bfloat16 rh)
```

### Description

Performs `nv_bfloat16` ordered greater-than compare operation. See also [hgt\(`\_\_nv\_bfloat16`, `\_\_nv\_bfloat16`\)](#)

```
__host__ device__ __CUDA_BF16_FORCEINLINE__ bool
operator>= (const __nv_bfloat16 lh, const __nv_bfloat16 rh)
```

### Description

Performs `nv_bfloat16` ordered greater-or-equal compare operation. See also [hge\(`\_\_nv\_bfloat16`, `\_\_nv\_bfloat16`\)](#)

## 1.3.5. Bfloat162 Comparison Functions

### Bfloat16 Precision Ininsics

To use these functions, include the header file `cuda_bf16.h` in your program.

`__host__ __device__ bool __hbeq2 (const __nv_bfloat162 a, const __nv_bfloat162 b)`

Performs `nv_bfloat162` vector if-equal comparison and returns boolean true iff both `nv_bfloat16` results are true, boolean false otherwise.

### Parameters

**a**

- `nv_bfloat162`. Is only being read.

**b**

- `nv_bfloat162`. Is only being read.

### Returns

bool

- ▶ true if both `nv_bfloat16` results of if-equal comparison of vectors a and b are true;
- ▶ false otherwise.

### Description

Performs `nv_bfloat162` vector if-equal comparison of inputs a and b. The bool result is set to true only if both `nv_bfloat16` if-equal comparisons evaluate to true, or false otherwise. NaN inputs generate false results.

`__host__ __device__ bool __hbequ2 (const __nv_bfloat162 a, const __nv_bfloat162 b)`

Performs `nv_bfloat162` vector unordered if-equal comparison and returns boolean true iff both `nv_bfloat16` results are true, boolean false otherwise.

### Parameters

**a**

- `nv_bfloat162`. Is only being read.

**b**

- `nv_bfloat162`. Is only being read.

### Returns

bool

- ▶ true if both `nv_bfloat16` results of unordered if-equal comparison of vectors a and b are true;
- ▶ false otherwise.

## Description

Performs `nv_bfloat162` vector if-equal comparison of inputs `a` and `b`. The `bool` result is set to true only if both `nv_bfloat16` if-equal comparisons evaluate to true, or false otherwise. NaN inputs generate true results.

```
__host__ __device__ bool __hbge2 (const __nv_bfloat162 a,
const __nv_bfloat162 b)
```

Performs `nv_bfloat162` vector greater-equal comparison and returns boolean true iff both `nv_bfloat16` results are true, boolean false otherwise.

## Parameters

- a**  
- `nv_bfloat162`. Is only being read.
- b**  
- `nv_bfloat162`. Is only being read.

## Returns

`bool`

- ▶ true if both `nv_bfloat16` results of greater-equal comparison of vectors `a` and `b` are true;
- ▶ false otherwise.

## Description

Performs `nv_bfloat162` vector greater-equal comparison of inputs `a` and `b`. The `bool` result is set to true only if both `nv_bfloat16` greater-equal comparisons evaluate to true, or false otherwise. NaN inputs generate false results.

```
__host__ __device__ bool __hbgeu2 (const __nv_bfloat162 a,
const __nv_bfloat162 b)
```

Performs `nv_bfloat162` vector unordered greater-equal comparison and returns boolean true iff both `nv_bfloat16` results are true, boolean false otherwise.

## Parameters

- a**  
- `nv_bfloat162`. Is only being read.
- b**  
- `nv_bfloat162`. Is only being read.

## Returns

bool

- ▶ true if both `nv_bfloat16` results of unordered greater-equal comparison of vectors `a` and `b` are true;
- ▶ false otherwise.

## Description

Performs `nv_bfloat162` vector greater-equal comparison of inputs `a` and `b`. The bool result is set to true only if both `nv_bfloat16` greater-equal comparisons evaluate to true, or false otherwise. NaN inputs generate true results.

`__host__ __device__ bool __hbg2 (const __nv_bfloat162 a, const __nv_bfloat162 b)`

Performs `nv_bfloat162` vector greater-than comparison and returns boolean true iff both `nv_bfloat16` results are true, boolean false otherwise.

## Parameters

**a**

- `nv_bfloat162`. Is only being read.

**b**

- `nv_bfloat162`. Is only being read.

## Returns

bool

- ▶ true if both `nv_bfloat16` results of greater-than comparison of vectors `a` and `b` are true;
- ▶ false otherwise.

## Description

Performs `nv_bfloat162` vector greater-than comparison of inputs `a` and `b`. The bool result is set to true only if both `nv_bfloat16` greater-than comparisons evaluate to true, or false otherwise. NaN inputs generate false results.

## `__host__ __device__ bool __hbgtu2 (const __nv_bfloat162 a, const __nv_bfloat162 b)`

Performs `nv_bfloat162` vector unordered greater-than comparison and returns boolean true iff both `nv_bfloat16` results are true, boolean false otherwise.

### Parameters

- a**  
- `nv_bfloat162`. Is only being read.
- b**  
- `nv_bfloat162`. Is only being read.

### Returns

bool

- ▶ true if both `nv_bfloat16` results of unordered greater-than comparison of vectors `a` and `b` are true;
- ▶ false otherwise.

### Description

Performs `nv_bfloat162` vector greater-than comparison of inputs `a` and `b`. The bool result is set to true only if both `nv_bfloat16` greater-than comparisons evaluate to true, or false otherwise. NaN inputs generate true results.

## `__host__ __device__ bool __hble2 (const __nv_bfloat162 a, const __nv_bfloat162 b)`

Performs `nv_bfloat162` vector less-equal comparison and returns boolean true iff both `nv_bfloat16` results are true, boolean false otherwise.

### Parameters

- a**  
- `nv_bfloat162`. Is only being read.
- b**  
- `nv_bfloat162`. Is only being read.

### Returns

bool

- ▶ true if both `nv_bfloat16` results of less-equal comparison of vectors `a` and `b` are true;
- ▶ false otherwise.

## Description

Performs `nv_bfloat162` vector less-equal comparison of inputs `a` and `b`. The `bool` result is set to `true` only if both `nv_bfloat16` less-equal comparisons evaluate to `true`, or `false` otherwise. NaN inputs generate `false` results.

```
__host__ __device__ bool __hbleu2 (const __nv_bfloat162 a,
const __nv_bfloat162 b)
```

Performs `nv_bfloat162` vector unordered less-equal comparison and returns boolean `true` iff both `nv_bfloat16` results are `true`, boolean `false` otherwise.

## Parameters

- a**  
- `nv_bfloat162`. Is only being read.
- b**  
- `nv_bfloat162`. Is only being read.

## Returns

`bool`

- ▶ `true` if both `nv_bfloat16` results of unordered less-equal comparison of vectors `a` and `b` are `true`;
- ▶ `false` otherwise.

## Description

Performs `nv_bfloat162` vector less-equal comparison of inputs `a` and `b`. The `bool` result is set to `true` only if both `nv_bfloat16` less-equal comparisons evaluate to `true`, or `false` otherwise. NaN inputs generate `true` results.

```
__host__ __device__ bool __hblt2 (const __nv_bfloat162 a,
const __nv_bfloat162 b)
```

Performs `nv_bfloat162` vector less-than comparison and returns boolean `true` iff both `nv_bfloat16` results are `true`, boolean `false` otherwise.

## Parameters

- a**  
- `nv_bfloat162`. Is only being read.
- b**  
- `nv_bfloat162`. Is only being read.

## Returns

bool

- ▶ true if both `nv_bfloat16` results of less-than comparison of vectors `a` and `b` are true;
- ▶ false otherwise.

## Description

Performs `nv_bfloat162` vector less-than comparison of inputs `a` and `b`. The bool result is set to true only if both `nv_bfloat16` less-than comparisons evaluate to true, or false otherwise. NaN inputs generate false results.

`__host__ __device__ bool __hbltu2 (const __nv_bfloat162 a, const __nv_bfloat162 b)`

Performs `nv_bfloat162` vector unordered less-than comparison and returns boolean true iff both `nv_bfloat16` results are true, boolean false otherwise.

## Parameters

**a**

- `nv_bfloat162`. Is only being read.

**b**

- `nv_bfloat162`. Is only being read.

## Returns

bool

- ▶ true if both `nv_bfloat16` results of unordered less-than comparison of vectors `a` and `b` are true;
- ▶ false otherwise.

## Description

Performs `nv_bfloat162` vector less-than comparison of inputs `a` and `b`. The bool result is set to true only if both `nv_bfloat16` less-than comparisons evaluate to true, or false otherwise. NaN inputs generate true results.

`__host__ __device__ bool __hbne2 (const __nv_bfloat162 a, const __nv_bfloat162 b)`

Performs `nv_bfloat162` vector not-equal comparison and returns boolean true iff both `nv_bfloat16` results are true, boolean false otherwise.

### Parameters

**a**

- `nv_bfloat162`. Is only being read.

**b**

- `nv_bfloat162`. Is only being read.

### Returns

bool

- ▶ true if both `nv_bfloat16` results of not-equal comparison of vectors `a` and `b` are true,
- ▶ false otherwise.

### Description

Performs `nv_bfloat162` vector not-equal comparison of inputs `a` and `b`. The bool result is set to true only if both `nv_bfloat16` not-equal comparisons evaluate to true, or false otherwise. NaN inputs generate false results.

`__host__ __device__ bool __hbneu2 (const __nv_bfloat162 a, const __nv_bfloat162 b)`

Performs `nv_bfloat162` vector unordered not-equal comparison and returns boolean true iff both `nv_bfloat16` results are true, boolean false otherwise.

### Parameters

**a**

- `nv_bfloat162`. Is only being read.

**b**

- `nv_bfloat162`. Is only being read.

### Returns

bool

- ▶ true if both `nv_bfloat16` results of unordered not-equal comparison of vectors `a` and `b` are true;
- ▶ false otherwise.



## Description

Performs `nv_bfloat162` vector not-equal comparison of inputs `a` and `b`. The bool result is set to true only if both `nv_bfloat16` not-equal comparisons evaluate to true, or false otherwise. NaN inputs generate true results.

```
__host__ __device__ heq2 (const __nv_bfloat162 a, const
__nv_bfloat162 b)
```

Performs `nv_bfloat162` vector if-equal comparison.

## Parameters

**a**

- `nv_bfloat162`. Is only being read.

**b**

- `nv_bfloat162`. Is only being read.

## Returns

`nv_bfloat162`

- The vector result of if-equal comparison of vectors `a` and `b`.

## Description

Performs `nv_bfloat162` vector if-equal comparison of inputs `a` and `b`. The corresponding `nv_bfloat16` results are set to 1.0 for true, or 0.0 for false. NaN inputs generate false results.

```
__host__ __device__ unsigned __heq2_mask (const
__nv_bfloat162 a, const __nv_bfloat162 b)
```

Performs `nv_bfloat162` vector if-equal comparison.

## Parameters

**a**

- `nv_bfloat162`. Is only being read.

**b**

- `nv_bfloat162`. Is only being read.

## Returns

unsigned int

- The vector mask result of if-equal comparison of vectors `a` and `b`.

## Description

Performs `nv_bfloat162` vector if-equal comparison of inputs `a` and `b`. The corresponding unsigned bits are set to `0xFFFF` for true, or `0x0` for false. NaN inputs generate false results.

`__host__ __device__ hequ2 (const __nv_bfloat162 a, const __nv_bfloat162 b)`

Performs `nv_bfloat162` vector unordered if-equal comparison.

## Parameters

**a**

- `nv_bfloat162`. Is only being read.

**b**

- `nv_bfloat162`. Is only being read.

## Returns

`nv_bfloat162`

- The vector result of unordered if-equal comparison of vectors `a` and `b`.

## Description

Performs `nv_bfloat162` vector if-equal comparison of inputs `a` and `b`. The corresponding `nv_bfloat16` results are set to `1.0` for true, or `0.0` for false. NaN inputs generate true results.

`__host__ __device__ unsigned __hequ2_mask (const __nv_bfloat162 a, const __nv_bfloat162 b)`

Performs `nv_bfloat162` vector unordered if-equal comparison.

## Parameters

**a**

- `nv_bfloat162`. Is only being read.

**b**

- `nv_bfloat162`. Is only being read.

## Returns

unsigned int

- The vector mask result of unordered if-equal comparison of vectors `a` and `b`.

## Description

Performs `nv_bfloat162` vector if-equal comparison of inputs `a` and `b`. The corresponding unsigned bits are set to `0xFFFF` for true, or `0x0` for false. NaN inputs generate true results.

```
__host__ __device__ hge2 (const __nv_bfloat162 a, const
__nv_bfloat162 b)
```

Performs `nv_bfloat162` vector greater-equal comparison.

## Parameters

**a**

- `nv_bfloat162`. Is only being read.

**b**

- `nv_bfloat162`. Is only being read.

## Returns

`nv_bfloat162`

- The vector result of greater-equal comparison of vectors `a` and `b`.

## Description

Performs `nv_bfloat162` vector greater-equal comparison of inputs `a` and `b`. The corresponding `nv_bfloat16` results are set to `1.0` for true, or `0.0` for false. NaN inputs generate false results.

```
__host__ __device__ unsigned hge2_mask (const
__nv_bfloat162 a, const __nv_bfloat162 b)
```

Performs `nv_bfloat162` vector greater-equal comparison.

## Parameters

**a**

- `nv_bfloat162`. Is only being read.

**b**

- `nv_bfloat162`. Is only being read.

## Returns

unsigned int

- The vector mask result of greater-equal comparison of vectors `a` and `b`.

## Description

Performs `nv_bfloat162` vector greater-equal comparison of inputs `a` and `b`. The corresponding unsigned bits are set to `0xFFFF` for true, or `0x0` for false. NaN inputs generate false results.

```
__host__ __device__ hgeu2 (const __nv_bfloat162 a, const
__nv_bfloat162 b)
```

Performs `nv_bfloat162` vector unordered greater-equal comparison.

## Parameters

**a**

- `nv_bfloat162`. Is only being read.

**b**

- `nv_bfloat162`. Is only being read.

## Returns

`nv_bfloat162`

- The `nv_bfloat162` vector result of unordered greater-equal comparison of vectors `a` and `b`.

## Description

Performs `nv_bfloat162` vector greater-equal comparison of inputs `a` and `b`. The corresponding `nv_bfloat16` results are set to `1.0` for true, or `0.0` for false. NaN inputs generate true results.

```
__host__ __device__ unsigned __hgeu2_mask (const
__nv_bfloat162 a, const __nv_bfloat162 b)
```

Performs `nv_bfloat162` vector unordered greater-equal comparison.

## Parameters

**a**

- `nv_bfloat162`. Is only being read.

**b**

- `nv_bfloat162`. Is only being read.

## Returns

unsigned int

- The vector mask result of unordered greater-equal comparison of vectors `a` and `b`.

## Description

Performs `nv_bfloat162` vector greater-equal comparison of inputs `a` and `b`. The corresponding unsigned bits are set to `0xFFFF` for true, or `0x0` for false. NaN inputs generate true results.

```
__host__ __device__ hgt2 (const __nv_bfloat162 a, const
__nv_bfloat162 b)
```

Performs `nv_bfloat162` vector greater-than comparison.

## Parameters

**a**

- `nv_bfloat162`. Is only being read.

**b**

- `nv_bfloat162`. Is only being read.

## Returns

`nv_bfloat162`

- The vector result of greater-than comparison of vectors `a` and `b`.

## Description

Performs `nv_bfloat162` vector greater-than comparison of inputs `a` and `b`. The corresponding `nv_bfloat16` results are set to `1.0` for true, or `0.0` for false. NaN inputs generate false results.

```
__host__ __device__ unsigned __hgt2_mask (const
__nv_bfloat162 a, const __nv_bfloat162 b)
```

Performs `nv_bfloat162` vector greater-than comparison.

## Parameters

**a**

- `nv_bfloat162`. Is only being read.

**b**

- `nv_bfloat162`. Is only being read.

## Returns

unsigned int

- The vector mask result of greater-than comparison of vectors `a` and `b`.

## Description

Performs `nv_bfloat162` vector greater-than comparison of inputs `a` and `b`. The corresponding unsigned bits are set to `0xFFFF` for true, or `0x0` for false. NaN inputs generate false results.

```
__host__ __device__ hgtu2 (const __nv_bfloat162 a, const
__nv_bfloat162 b)
```

Performs `nv_bfloat162` vector unordered greater-than comparison.

## Parameters

**a**

- `nv_bfloat162`. Is only being read.

**b**

- `nv_bfloat162`. Is only being read.

## Returns

`nv_bfloat162`

- ▶ The `nv_bfloat162` vector result of unordered greater-than comparison of vectors `a` and `b`.

## Description

Performs `nv_bfloat162` vector greater-than comparison of inputs `a` and `b`. The corresponding `nv_bfloat16` results are set to `1.0` for true, or `0.0` for false. NaN inputs generate true results.

```
__host__ __device__ unsigned __hgtu2_mask (const
__nv_bfloat162 a, const __nv_bfloat162 b)
```

Performs `nv_bfloat162` vector unordered greater-than comparison.

## Parameters

**a**

- `nv_bfloat162`. Is only being read.

**b**

- `nv_bfloat162`. Is only being read.

## Returns

unsigned int

- ▶ The vector mask result of unordered greater-than comparison of vectors `a` and `b`.

## Description

Performs `nv_bfloat162` vector greater-than comparison of inputs `a` and `b`. The corresponding unsigned bits are set to `0xFFFF` for true, or `0x0` for false. NaN inputs generate true results.

`__host__ __device__ hisnan2 (const __nv_bfloat162 a)`

Determine whether `nv_bfloat162` argument is a NaN.

## Parameters

**a**

- `nv_bfloat162`. Is only being read.

## Returns

`nv_bfloat162`

- ▶ The `nv_bfloat162` with the corresponding `nv_bfloat16` results set to 1.0 for NaN, 0.0 otherwise.

## Description

Determine whether each `nv_bfloat16` of input `nv_bfloat162` number `a` is a NaN.

`__host__ __device__ hle2 (const __nv_bfloat162 a, const __nv_bfloat162 b)`

Performs `nv_bfloat162` vector less-equal comparison.

## Parameters

**a**

- `nv_bfloat162`. Is only being read.

**b**

- `nv_bfloat162`. Is only being read.

## Returns

`nv_bfloat162`

- ▶ The `nv_bfloat162` result of less-equal comparison of vectors `a` and `b`.

## Description

Performs `nv_bfloat162` vector less-equal comparison of inputs `a` and `b`. The corresponding `nv_bfloat16` results are set to 1.0 for true, or 0.0 for false. NaN inputs generate false results.

`__host__ __device__ unsigned __hlt2_mask (const __nv_bfloat162 a, const __nv_bfloat162 b)`

Performs `nv_bfloat162` vector less-equal comparison.

### Parameters

**a**

- `nv_bfloat162`. Is only being read.

**b**

- `nv_bfloat162`. Is only being read.

### Returns

unsigned int

- The vector mask result of less-equal comparison of vectors `a` and `b`.

### Description

Performs `nv_bfloat162` vector less-equal comparison of inputs `a` and `b`. The corresponding unsigned bits are set to `0xFFFF` for true, or `0x0` for false. NaN inputs generate false results.

`__host__ __device__ __hltu2 (const __nv_bfloat162 a, const __nv_bfloat162 b)`

Performs `nv_bfloat162` vector unordered less-equal comparison.

### Parameters

**a**

- `nv_bfloat162`. Is only being read.

**b**

- `nv_bfloat162`. Is only being read.

### Returns

`nv_bfloat162`

- The vector result of unordered less-equal comparison of vectors `a` and `b`.

### Description

Performs `nv_bfloat162` vector less-equal comparison of inputs `a` and `b`. The corresponding `nv_bfloat16` results are set to `1.0` for true, or `0.0` for false. NaN inputs generate true results.



`__host__ __device__ unsigned __hlt2_mask (const __nv_bfloat162 a, const __nv_bfloat162 b)`

Performs `nv_bfloat162` vector unordered less-equal comparison.

### Parameters

- a**  
- `nv_bfloat162`. Is only being read.
- b**  
- `nv_bfloat162`. Is only being read.

### Returns

unsigned int

- ▶ The vector mask result of unordered less-equal comparison of vectors `a` and `b`.

### Description

Performs `nv_bfloat162` vector less-equal comparison of inputs `a` and `b`. The corresponding unsigned bits are set to `0xFFFF` for true, or `0x0` for false. NaN inputs generate true results.

`__host__ __device__ __hlt2 (const __nv_bfloat162 a, const __nv_bfloat162 b)`

Performs `nv_bfloat162` vector less-than comparison.

### Parameters

- a**  
- `nv_bfloat162`. Is only being read.
- b**  
- `nv_bfloat162`. Is only being read.

### Returns

`nv_bfloat162`

- ▶ The `nv_bfloat162` vector result of less-than comparison of vectors `a` and `b`.

### Description

Performs `nv_bfloat162` vector less-than comparison of inputs `a` and `b`. The corresponding `nv_bfloat16` results are set to `1.0` for true, or `0.0` for false. NaN inputs generate false results.

`__host__ __device__ unsigned __hlt2_mask (const __nv_bfloat162 a, const __nv_bfloat162 b)`

Performs `nv_bfloat162` vector less-than comparison.

### Parameters

**a**

- `nv_bfloat162`. Is only being read.

**b**

- `nv_bfloat162`. Is only being read.

### Returns

unsigned int

- The vector mask result of less-than comparison of vectors `a` and `b`.

### Description

Performs `nv_bfloat162` vector less-than comparison of inputs `a` and `b`. The corresponding unsigned bits are set to `0xFFFF` for true, or `0x0` for false. NaN inputs generate false results.

`__host__ __device__ __hltu2 (const __nv_bfloat162 a, const __nv_bfloat162 b)`

Performs `nv_bfloat162` vector unordered less-than comparison.

### Parameters

**a**

- `nv_bfloat162`. Is only being read.

**b**

- `nv_bfloat162`. Is only being read.

### Returns

`nv_bfloat162`

- The vector result of unordered less-than comparison of vectors `a` and `b`.

### Description

Performs `nv_bfloat162` vector less-than comparison of inputs `a` and `b`. The corresponding `nv_bfloat16` results are set to `1.0` for true, or `0.0` for false. NaN inputs generate true results.

`__host__ __device__ unsigned __hltu2_mask (const __nv_bfloat162 a, const __nv_bfloat162 b)`

Performs `nv_bfloat162` vector unordered less-than comparison.

### Parameters

**a**

- `nv_bfloat162`. Is only being read.

**b**

- `nv_bfloat162`. Is only being read.

### Returns

unsigned int

- ▶ The vector mask result of unordered less-than comparison of vectors `a` and `b`.

### Description

Performs `nv_bfloat162` vector less-than comparison of inputs `a` and `b`. The corresponding unsigned bits are set to `0xFFFF` for true, or `0x0` for false. NaN inputs generate true results.

`__host__ __device__ __hmax2 (const __nv_bfloat162 a, const __nv_bfloat162 b)`

Calculates `nv_bfloat162` vector maximum of two inputs.

### Description

Calculates `nv_bfloat162` vector `max(a, b)`. Elementwise `nv_bfloat16` operation is defined as `(a > b) ? a : b`.

- ▶ If either of inputs is NaN, the other input is returned.
- ▶ If both inputs are NaNs, then canonical NaN is returned.
- ▶ If values of both inputs are 0.0, then `+0.0 > -0.0`
- ▶ The result of elementwise maximum of vectors `a` and `b`

`__host__ __device__ hmax2_nan (const __nv_bfloat162 a, const __nv_bfloat162 b)`

Calculates `nv_bfloat162` vector maximum of two inputs, NaNs pass through.

### Description

Calculates `nv_bfloat162` vector  $\max(a, b)$ . Elementwise `nv_bfloat16` operation is defined as  $(a > b) ? a : b$ .

- ▶ If either of inputs is NaN, then canonical NaN is returned.
- ▶ If values of both inputs are 0.0, then  $+0.0 > -0.0$
- ▶ The result of elementwise maximum of vectors `a` and `b`, with NaNs pass through

`__host__ __device__ hmin2 (const __nv_bfloat162 a, const __nv_bfloat162 b)`

Calculates `nv_bfloat162` vector minimum of two inputs.

### Description

Calculates `nv_bfloat162` vector  $\min(a, b)$ . Elementwise `nv_bfloat16` operation is defined as  $(a < b) ? a : b$ .

- ▶ If either of inputs is NaN, the other input is returned.
- ▶ If both inputs are NaNs, then canonical NaN is returned.
- ▶ If values of both inputs are 0.0, then  $+0.0 > -0.0$
- ▶ The result of elementwise minimum of vectors `a` and `b`

`__host__ __device__ hmin2_nan (const __nv_bfloat162 a, const __nv_bfloat162 b)`

Calculates `nv_bfloat162` vector minimum of two inputs, NaNs pass through.

### Description

Calculates `nv_bfloat162` vector  $\min(a, b)$ . Elementwise `nv_bfloat16` operation is defined as  $(a < b) ? a : b$ .

- ▶ If either of inputs is NaN, then canonical NaN is returned.
- ▶ If values of both inputs are 0.0, then  $+0.0 > -0.0$
- ▶ The result of elementwise minimum of vectors `a` and `b`, with NaNs pass through

`__host__ __device__ hne2 (const __nv_bfloat162 a, const __nv_bfloat162 b)`

Performs `nv_bfloat162` vector not-equal comparison.

### Parameters

**a**

- `nv_bfloat162`. Is only being read.

**b**

- `nv_bfloat162`. Is only being read.

### Returns

`nv_bfloat162`

- The vector result of not-equal comparison of vectors `a` and `b`.

### Description

Performs `nv_bfloat162` vector not-equal comparison of inputs `a` and `b`. The corresponding `nv_bfloat16` results are set to 1.0 for true, or 0.0 for false. NaN inputs generate false results.

`__host__ __device__ unsigned hne2_mask (const __nv_bfloat162 a, const __nv_bfloat162 b)`

Performs `nv_bfloat162` vector not-equal comparison.

### Parameters

**a**

- `nv_bfloat162`. Is only being read.

**b**

- `nv_bfloat162`. Is only being read.

### Returns

unsigned int

- The vector mask result of not-equal comparison of vectors `a` and `b`.

### Description

Performs `nv_bfloat162` vector not-equal comparison of inputs `a` and `b`. The corresponding unsigned bits are set to 0xFFFF for true, or 0x0 for false. NaN inputs generate false results.

`__host__ device__ hneu2 (const __nv_bfloat162 a, const __nv_bfloat162 b)`

Performs `nv_bfloat162` vector unordered not-equal comparison.

### Parameters

**a**

- `nv_bfloat162`. Is only being read.

**b**

- `nv_bfloat162`. Is only being read.

### Returns

`nv_bfloat162`

- The vector result of unordered not-equal comparison of vectors `a` and `b`.

### Description

Performs `nv_bfloat162` vector not-equal comparison of inputs `a` and `b`. The corresponding `nv_bfloat16` results are set to 1.0 for true, or 0.0 for false. NaN inputs generate true results.

`__host__ device__ unsigned__ hneu2_mask (const __nv_bfloat162 a, const __nv_bfloat162 b)`

Performs `nv_bfloat162` vector unordered not-equal comparison.

### Parameters

**a**

- `nv_bfloat162`. Is only being read.

**b**

- `nv_bfloat162`. Is only being read.

### Returns

unsigned int

- The vector mask result of unordered not-equal comparison of vectors `a` and `b`.

### Description

Performs `nv_bfloat162` vector not-equal comparison of inputs `a` and `b`. The corresponding unsigned bits are set to 0xFFFF for true, or 0x0 for false. NaN inputs generate true results.

```
__host__ device__ __CUDA_BF16_FORCEINLINE__ bool
operator!= (const __nv_bfloat162 lh, const __nv_bfloat162
rh)
```

### Description

Performs packed `nv_bfloat16` unordered compare not-equal operation. See also [\\_\\_hbnue2\\_\\_\(\\_\\_nv\\_bfloat162, \\_\\_nv\\_bfloat162\)](#)

```
__host__ device__ __CUDA_BF16_FORCEINLINE__ bool
operator< (const __nv_bfloat162 lh, const __nv_bfloat162 rh)
```

### Description

Performs packed `nv_bfloat16` ordered less-than compare operation. See also [\\_\\_hblt2\\_\\_\(\\_\\_nv\\_bfloat162, \\_\\_nv\\_bfloat162\)](#)

```
__host__ device__ __CUDA_BF16_FORCEINLINE__ bool
operator<= (const __nv_bfloat162 lh, const __nv_bfloat162
rh)
```

### Description

Performs packed `nv_bfloat16` ordered less-or-equal compare operation. See also [\\_\\_hble2\\_\\_\(\\_\\_nv\\_bfloat162, \\_\\_nv\\_bfloat162\)](#)

```
__host__ device__ __CUDA_BF16_FORCEINLINE__ bool
operator== (const __nv_bfloat162 lh, const __nv_bfloat162
rh)
```

### Description

Performs packed `nv_bfloat16` ordered compare equal operation. See also [\\_\\_hbeq2\\_\\_\(\\_\\_nv\\_bfloat162, \\_\\_nv\\_bfloat162\)](#)

```
__host__ __device__ __CUDA_BF16_FORCEINLINE__ bool
operator> (const __nv_bfloat162 lh, const __nv_bfloat162 rh)
```

### Description

Performs packed `nv_bfloat16` ordered greater-than compare operation. See also [\\_\\_hbgt2\(\\_\\_nv\\_bfloat162, \\_\\_nv\\_bfloat162\)](#)

```
__host__ __device__ __CUDA_BF16_FORCEINLINE__ bool
operator>= (const __nv_bfloat162 lh, const __nv_bfloat162
rh)
```

### Description

Performs packed `nv_bfloat16` ordered greater-or-equal compare operation. See also [\\_\\_hbge2\(\\_\\_nv\\_bfloat162, \\_\\_nv\\_bfloat162\)](#)

## 1.3.6. Bfloat16 Precision Conversion and Data Movement

Bfloat16 Precision Intrinsics

To use these functions, include the header file `cuda_bf16.h` in your program.

```
__host__ __device__ float2 __bfloat1622float2 (const
__nv_bfloat162 a)
```

Converts both halves of `nv_bfloat162` to `float2` and returns the result.

### Parameters

- a**
- `nv_bfloat162`. Is only being read.

### Returns

`float2`

- `a` converted to `float2`.

### Description

Converts both halves of `nv_bfloat162` input `a` to `float2` and returns the result.



```
__host__ device__ bfloat162bfloat162 (const
__nv_bfloat16 a)
```

Returns `nv_bfloat162` with both halves equal to the input value.

### Parameters

**a**

- `nv_bfloat16`. Is only being read.

### Returns

`nv_bfloat162`

► The vector which has both its halves equal to the input `a`.

### Description

Returns `nv_bfloat162` number with both halves equal to the input `a nv_bfloat16` number.

```
__host__ device__ signed char __bfloat162char_rz (const
__nv_bfloat16 h)
```

Convert a `nv_bfloat16` to a signed char in round-towards-zero mode.

### Parameters

**h**

- `nv_bfloat16`. Is only being read.

### Returns

signed char

► `h` converted to a signed char.

### Description

Convert the `nv_bfloat16` floating-point value `h` to a signed char in round-towards-zero mode. NaN inputs are converted to 0.

```
__host__ __device__ float __bfloat162float (const __nv_bfloat16 a)
```

Converts `nv_bfloat16` number to float.

### Parameters

**a**

- float. Is only being read.

### Returns

float

► a converted to float.

### Description

Converts `nv_bfloat16` number a to float.

```
__device__ int __bfloat162int_rd (const __nv_bfloat16 h)
```

Convert a `nv_bfloat16` to a signed integer in round-down mode.

### Parameters

**h**

- `nv_bfloat16`. Is only being read.

### Returns

int

► h converted to a signed integer.

### Description

Convert the `nv_bfloat16` floating-point value h to a signed integer in round-down mode. NaN inputs are converted to 0.

```
__device__ int __bfloat162int_rn (const __nv_bfloat16 h)
```

Convert a `nv_bfloat16` to a signed integer in round-to-nearest-even mode.

### Parameters

**h**

- `nv_bfloat16`. Is only being read.

## Returns

int

- ▶ `h` converted to a signed integer.

## Description

Convert the `nv_bfloat16` floating-point value `h` to a signed integer in round-to-nearest-even mode. NaN inputs are converted to 0.

```
__device__ int __bfloat162int_ru (const __nv_bfloat16 h)
```

Convert a `nv_bfloat16` to a signed integer in round-up mode.

## Parameters

**h**

- `nv_bfloat16`. Is only being read.

## Returns

int

- ▶ `h` converted to a signed integer.

## Description

Convert the `nv_bfloat16` floating-point value `h` to a signed integer in round-up mode. NaN inputs are converted to 0.

```
__host__ __device__ int __bfloat162int_rz (const  
__nv_bfloat16 h)
```

Convert a `nv_bfloat16` to a signed integer in round-towards-zero mode.

## Parameters

**h**

- `nv_bfloat16`. Is only being read.

## Returns

int

- ▶ `h` converted to a signed integer.

## Description

Convert the `nv_bfloat16` floating-point value `h` to a signed integer in round-towards-zero mode. NaN inputs are converted to 0.

```
__device__ long long int __bfloat162ll_rd (const
__nv_bfloat16 h)
```

Convert a `nv_bfloat16` to a signed 64-bit integer in round-down mode.

## Parameters

**h**

- `nv_bfloat16`. Is only being read.

## Returns

long long int

- ▶ `h` converted to a signed 64-bit integer.

## Description

Convert the `nv_bfloat16` floating-point value `h` to a signed 64-bit integer in round-down mode. NaN inputs return a long long int with hex value of 0x8000000000000000.

```
__device__ long long int __bfloat162ll_rn (const
__nv_bfloat16 h)
```

Convert a `nv_bfloat16` to a signed 64-bit integer in round-to-nearest-even mode.

## Parameters

**h**

- `nv_bfloat16`. Is only being read.

## Returns

long long int

- ▶ `h` converted to a signed 64-bit integer.

## Description

Convert the `nv_bfloat16` floating-point value `h` to a signed 64-bit integer in round-to-nearest-even mode. NaN inputs return a long long int with hex value of 0x8000000000000000.

```
__device__ long long int __bfloat162ll_ru (const
__nv_bfloat16 h)
```

Convert a `nv_bfloat16` to a signed 64-bit integer in round-up mode.

### Parameters

**h**

- `nv_bfloat16`. Is only being read.

### Returns

long long int

- ▶ `h` converted to a signed 64-bit integer.

### Description

Convert the `nv_bfloat16` floating-point value `h` to a signed 64-bit integer in round-up mode. NaN inputs return a long long int with hex value of 0x8000000000000000.

```
__host__ __device__ long long int __bfloat162ll_rz (const
__nv_bfloat16 h)
```

Convert a `nv_bfloat16` to a signed 64-bit integer in round-towards-zero mode.

### Parameters

**h**

- `nv_bfloat16`. Is only being read.

### Returns

long long int

- ▶ `h` converted to a signed 64-bit integer.

### Description

Convert the `nv_bfloat16` floating-point value `h` to a signed 64-bit integer in round-towards-zero mode. NaN inputs return a long long int with hex value of 0x8000000000000000.

```
__device__ short int __bfloat162short_rd (const  
__nv_bfloat16 h)
```

Convert a `nv_bfloat16` to a signed short integer in round-down mode.

### Parameters

**h**

- `nv_bfloat16`. Is only being read.

### Returns

short int

- ▶ `h` converted to a signed short integer.

### Description

Convert the `nv_bfloat16` floating-point value `h` to a signed short integer in round-down mode. NaN inputs are converted to 0.

```
__device__ short int __bfloat162short_rn (const  
__nv_bfloat16 h)
```

Convert a `nv_bfloat16` to a signed short integer in round-to-nearest-even mode.

### Parameters

**h**

- `nv_bfloat16`. Is only being read.

### Returns

short int

- ▶ `h` converted to a signed short integer.

### Description

Convert the `nv_bfloat16` floating-point value `h` to a signed short integer in round-to-nearest-even mode. NaN inputs are converted to 0.

```
__device__ short int __bfloat162short_ru (const  
__nv_bfloat16 h)
```

Convert a `nv_bfloat16` to a signed short integer in round-up mode.

### Parameters

**h**

- `nv_bfloat16`. Is only being read.

### Returns

short int

- ▶ `h` converted to a signed short integer.

### Description

Convert the `nv_bfloat16` floating-point value `h` to a signed short integer in round-up mode. NaN inputs are converted to 0.

```
__host__ __device__ short int __bfloat162short_rz (const  
__nv_bfloat16 h)
```

Convert a `nv_bfloat16` to a signed short integer in round-towards-zero mode.

### Parameters

**h**

- `nv_bfloat16`. Is only being read.

### Returns

short int

- ▶ `h` converted to a signed short integer.

### Description

Convert the `nv_bfloat16` floating-point value `h` to a signed short integer in round-towards-zero mode. NaN inputs are converted to 0.

```
__host__ __device__ unsigned char __bfloat162uchar_rz  
(const __nv_bfloat16 h)
```

Convert a `nv_bfloat16` to an unsigned char in round-towards-zero mode.

### Parameters

**h**

- `nv_bfloat16`. Is only being read.

### Returns

unsigned char

- `h` converted to an unsigned char.

### Description

Convert the `nv_bfloat16` floating-point value `h` to an unsigned char in round-towards-zero mode. NaN inputs are converted to 0.

```
__device__ unsigned int __bfloat162uint_rd (const  
__nv_bfloat16 h)
```

Convert a `nv_bfloat16` to an unsigned integer in round-down mode.

### Parameters

**h**

- `nv_bfloat16`. Is only being read.

### Returns

unsigned int

- `h` converted to an unsigned integer.

### Description

Convert the `nv_bfloat16` floating-point value `h` to an unsigned integer in round-down mode. NaN inputs are converted to 0.



```
__device__ unsigned int __bfloat162uint_rn (const
__nv_bfloat16 h)
```

Convert a `nv_bfloat16` to an unsigned integer in round-to-nearest-even mode.

### Parameters

**h**

- `nv_bfloat16`. Is only being read.

### Returns

unsigned int

- ▶ `h` converted to an unsigned integer.

### Description

Convert the `nv_bfloat16` floating-point value `h` to an unsigned integer in round-to-nearest-even mode. NaN inputs are converted to 0.

```
__device__ unsigned int __bfloat162uint_ru (const
__nv_bfloat16 h)
```

Convert a `nv_bfloat16` to an unsigned integer in round-up mode.

### Parameters

**h**

- `nv_bfloat16`. Is only being read.

### Returns

unsigned int

- ▶ `h` converted to an unsigned integer.

### Description

Convert the `nv_bfloat16` floating-point value `h` to an unsigned integer in round-up mode. NaN inputs are converted to 0.

```
__host__ __device__ unsigned int __bfloat162uint_rz (const
__nv_bfloat16 h)
```

Convert a `nv_bfloat16` to an unsigned integer in round-towards-zero mode.

### Parameters

**h**

- `nv_bfloat16`. Is only being read.

### Returns

unsigned int

- ▶ `h` converted to an unsigned integer.

### Description

Convert the `nv_bfloat16` floating-point value `h` to an unsigned integer in round-towards-zero mode. NaN inputs are converted to 0.

```
__device__ unsigned long long int __bfloat162ull_rd (const
__nv_bfloat16 h)
```

Convert a `nv_bfloat16` to an unsigned 64-bit integer in round-down mode.

### Parameters

**h**

- `nv_bfloat16`. Is only being read.

### Returns

unsigned long long int

- ▶ `h` converted to an unsigned 64-bit integer.

### Description

Convert the `nv_bfloat16` floating-point value `h` to an unsigned 64-bit integer in round-down mode. NaN inputs return `0x8000000000000000`.

```
__device__ unsigned long long int __bfloat162ull_rn (const
__nv_bfloat16 h)
```

Convert a `nv_bfloat16` to an unsigned 64-bit integer in round-to-nearest-even mode.

### Parameters

**h**

- `nv_bfloat16`. Is only being read.

### Returns

unsigned long long int

- ▶ `h` converted to an unsigned 64-bit integer.

### Description

Convert the `nv_bfloat16` floating-point value `h` to an unsigned 64-bit integer in round-to-nearest-even mode. NaN inputs return `0x8000000000000000`.

```
__device__ unsigned long long int __bfloat162ull_ru (const
__nv_bfloat16 h)
```

Convert a `nv_bfloat16` to an unsigned 64-bit integer in round-up mode.

### Parameters

**h**

- `nv_bfloat16`. Is only being read.

### Returns

unsigned long long int

- ▶ `h` converted to an unsigned 64-bit integer.

### Description

Convert the `nv_bfloat16` floating-point value `h` to an unsigned 64-bit integer in round-up mode. NaN inputs return `0x8000000000000000`.

```
__host__ __device__ unsigned long long int
__bfloat162ull_rz (const __nv_bfloat16 h)
```

Convert a `nv_bfloat16` to an unsigned 64-bit integer in round-towards-zero mode.

### Parameters

**h**

- `nv_bfloat16`. Is only being read.

### Returns

unsigned long long int

- ▶ `h` converted to an unsigned 64-bit integer.

### Description

Convert the `nv_bfloat16` floating-point value `h` to an unsigned 64-bit integer in round-towards-zero mode. NaN inputs return `0x8000000000000000`.

```
__device__ unsigned short int __bfloat162ushort_rd (const
__nv_bfloat16 h)
```

Convert a `nv_bfloat16` to an unsigned short integer in round-down mode.

### Parameters

**h**

- `nv_bfloat16`. Is only being read.

### Returns

unsigned short int

- ▶ `h` converted to an unsigned short integer.

### Description

Convert the `nv_bfloat16` floating-point value `h` to an unsigned short integer in round-down mode. NaN inputs are converted to 0.

```
__device__ unsigned short int __bfloat162ushort_rn (const  
__nv_bfloat16 h)
```

Convert a `nv_bfloat16` to an unsigned short integer in round-to-nearest-even mode.

### Parameters

**h**

- `nv_bfloat16`. Is only being read.

### Returns

unsigned short int

- ▶ `h` converted to an unsigned short integer.

### Description

Convert the `nv_bfloat16` floating-point value `h` to an unsigned short integer in round-to-nearest-even mode. NaN inputs are converted to 0.

```
__device__ unsigned short int __bfloat162ushort_ru (const  
__nv_bfloat16 h)
```

Convert a `nv_bfloat16` to an unsigned short integer in round-up mode.

### Parameters

**h**

- `nv_bfloat16`. Is only being read.

### Returns

unsigned short int

- ▶ `h` converted to an unsigned short integer.

### Description

Convert the `nv_bfloat16` floating-point value `h` to an unsigned short integer in round-up mode. NaN inputs are converted to 0.

```
__host__ __device__ unsigned short int
__bfloat162ushort_rz (const __nv_bfloat16 h)
```

Convert a `nv_bfloat16` to an unsigned short integer in round-towards-zero mode.

### Parameters

**h**

- `nv_bfloat16`. Is only being read.

### Returns

unsigned short int

- ▶ `h` converted to an unsigned short integer.

### Description

Convert the `nv_bfloat16` floating-point value `h` to an unsigned short integer in round-towards-zero mode. NaN inputs are converted to 0.

```
__host__ __device__ short int __bfloat16_as_short (const
__nv_bfloat16 h)
```

Reinterprets bits in a `nv_bfloat16` as a signed short integer.

### Parameters

**h**

- `nv_bfloat16`. Is only being read.

### Returns

short int

- ▶ The reinterpreted value.

### Description

Reinterprets the bits in the `nv_bfloat16` floating-point number `h` as a signed short integer.

`__host__ __device__ unsigned short int  
__bfloat16_as_ushort (const __nv_bfloat16 h)`

Reinterprets bits in a `nv_bfloat16` as an unsigned short integer.

### Parameters

**h**

- `nv_bfloat16`. Is only being read.

### Returns

unsigned short int

► The reinterpreted value.

### Description

Reinterprets the bits in the `nv_bfloat16` floating-point `h` as an unsigned short number.

`__host__ __device__ double2bfloat16 (const double a)`

Converts double number to `nv_bfloat16` precision in round-to-nearest-even mode and returns `nv_bfloat16` with converted value.

### Parameters

**a**

- double. Is only being read.

### Returns

`nv_bfloat16`

► `a` converted to `nv_bfloat16`.

### Description

Converts double number `a` to `nv_bfloat16` precision in round-to-nearest-even mode.

`__host__ __device__ float22bfloat162_rn (const float2 a)`

Converts both components of `float2` number to `nv_bfloat16` precision in round-to-nearest-even mode and returns `nv_bfloat162` with converted values.

### Parameters

**a**

- `float2`. Is only being read.

## Returns

`nv_bfloat162`

- ▶ The `nv_bfloat162` which has corresponding halves equal to the converted `float2` components.

## Description

Converts both components of `float2` to `nv_bfloat16` precision in round-to-nearest-even mode and combines the results into one `nv_bfloat162` number. Low 16 bits of the return value correspond to `a.x` and high 16 bits of the return value correspond to `a.y`.

## `__host__ __device__ float2bfloat16 (const float a)`

Converts `float` number to `nv_bfloat16` precision in round-to-nearest-even mode and returns `nv_bfloat16` with converted value.

## Parameters

**a**

- `float`. Is only being read.

## Returns

`nv_bfloat16`

- ▶ `a` converted to `nv_bfloat16`.

## Description

Converts `float` number `a` to `nv_bfloat16` precision in round-to-nearest-even mode.

## `__host__ __device__ float2bfloat162_rn (const float a)`

Converts input to `nv_bfloat16` precision in round-to-nearest-even mode and populates both halves of `nv_bfloat162` with converted value.

## Parameters

**a**

- `float`. Is only being read.

## Returns

`nv_bfloat162`

- ▶ The `nv_bfloat162` value with both halves equal to the converted `nv_bfloat16` precision number.



## Description

Converts input `a` to `nv_bfloat16` precision in round-to-nearest-even mode and populates both halves of `nv_bfloat162` with converted value.

## `__host__ __device__ float2bfloat16_rd (const float a)`

Converts float number to `nv_bfloat16` precision in round-down mode and returns `nv_bfloat16` with converted value.

## Parameters

**a**

- float. Is only being read.

## Returns

`nv_bfloat16`

- ▶ `a` converted to `nv_bfloat16`.

## Description

Converts float number `a` to `nv_bfloat16` precision in round-down mode.

## `__host__ __device__ float2bfloat16_rn (const float a)`

Converts float number to `nv_bfloat16` precision in round-to-nearest-even mode and returns `nv_bfloat16` with converted value.

## Parameters

**a**

- float. Is only being read.

## Returns

`nv_bfloat16`

- ▶ `a` converted to `nv_bfloat16`.

## Description

Converts float number `a` to `nv_bfloat16` precision in round-to-nearest-even mode.

## `__host__ device__ float2bfloat16_ru (const float a)`

Converts float number to `nv_bfloat16` precision in round-up mode and returns `nv_bfloat16` with converted value.

### Parameters

**a**

- float. Is only being read.

### Returns

`nv_bfloat16`

- ▶ a converted to `nv_bfloat16`.

### Description

Converts float number a to `nv_bfloat16` precision in round-up mode.

## `__host__ device__ float2bfloat16_rz (const float a)`

Converts float number to `nv_bfloat16` precision in round-towards-zero mode and returns `nv_bfloat16` with converted value.

### Parameters

**a**

- float. Is only being read.

### Returns

`nv_bfloat16`

- ▶ a converted to `nv_bfloat16`.

### Description

Converts float number a to `nv_bfloat16` precision in round-towards-zero mode.

## `__host__ device__ floats2bfloat162_rn (const float a, const float b)`

Converts both input floats to `nv_bfloat16` precision in round-to-nearest-even mode and returns `nv_bfloat162` with converted values.

### Parameters

**a**

- float. Is only being read.

**b**

- float. Is only being read.

**Returns**

nv\_bfloat162

► The nv\_bfloat162 value with corresponding halves equal to the converted input floats.

**Description**

Converts both input floats to nv\_bfloat16 precision in round-to-nearest-even mode and combines the results into one nv\_bfloat162 number. Low 16 bits of the return value correspond to the input a, high 16 bits correspond to the input b.

**\_\_host\_\_ device \_\_halves2bfloat162 (const  
\_\_nv\_bfloat16 a, const \_\_nv\_bfloat16 b)**

Combines two nv\_bfloat16 numbers into one nv\_bfloat162 number.

**Parameters****a**

- nv\_bfloat16. Is only being read.

**b**

- nv\_bfloat16. Is only being read.

**Returns**

nv\_bfloat162

► The nv\_bfloat162 with one nv\_bfloat16 equal to a and the other to b.

**Description**

Combines two input nv\_bfloat16 number a and b into one nv\_bfloat162 number. Input a is stored in low 16 bits of the return value, input b is stored in high 16 bits of the return value.

**\_\_host\_\_ device \_\_high2bfloat16 (const \_\_nv\_bfloat162  
a)**

Returns high 16 bits of nv\_bfloat162 input.

**Parameters****a**

- nv\_bfloat162. Is only being read.

## Returns

nv\_bfloat16

- ▶ The high 16 bits of the input.

## Description

Returns high 16 bits of `nv_bfloat162` input `a`.

```
__host__ __device__ high2bfloat162 (const __nv_bfloat162
a)
```

Extracts high 16 bits from `nv_bfloat162` input.

## Parameters

**a**

- `nv_bfloat162`. Is only being read.

## Returns

nv\_bfloat162

- ▶ The `nv_bfloat162` with both halves equal to the high 16 bits of the input.

## Description

Extracts high 16 bits from `nv_bfloat162` input `a` and returns a new `nv_bfloat162` number which has both halves equal to the extracted bits.

```
__host__ __device__ float __high2float (const
__nv_bfloat162 a)
```

Converts high 16 bits of `nv_bfloat162` to float and returns the result.

## Parameters

**a**

- `nv_bfloat162`. Is only being read.

## Returns

float

- ▶ The high 16 bits of `a` converted to float.

## Description

Converts high 16 bits of `nv_bfloat162` input `a` to 32-bit floating-point number and returns the result.

`__host__ __device__ highs2bfloat162 (const __nv_bfloat162 a, const __nv_bfloat162 b)`

Extracts high 16 bits from each of the two `nv_bfloat162` inputs and combines into one `nv_bfloat162` number.

## Parameters

**a**

- `nv_bfloat162`. Is only being read.

**b**

- `nv_bfloat162`. Is only being read.

## Returns

`nv_bfloat162`

- The high 16 bits of `a` and of `b`.

## Description

Extracts high 16 bits from each of the two `nv_bfloat162` inputs and combines into one `nv_bfloat162` number. High 16 bits from input `a` is stored in low 16 bits of the return value, high 16 bits from input `b` is stored in high 16 bits of the return value.

`__device__ int2bfloat16_rd (const int i)`

Convert a signed integer to a `nv_bfloat16` in round-down mode.

## Parameters

**i**

- `int`. Is only being read.

## Returns

`nv_bfloat16`

- `i` converted to `nv_bfloat16`.

## Description

Convert the signed integer value `i` to a `nv_bfloat16` floating-point value in round-down mode.

## `__host__ __device__ int2bfloat16_rn (const int i)`

Convert a signed integer to a `nv_bfloat16` in round-to-nearest-even mode.

### Parameters

**i**

- int. Is only being read.

### Returns

`nv_bfloat16`

- ▶ `i` converted to `nv_bfloat16`.

### Description

Convert the signed integer value `i` to a `nv_bfloat16` floating-point value in round-to-nearest-even mode.

## `__device__ int2bfloat16_ru (const int i)`

Convert a signed integer to a `nv_bfloat16` in round-up mode.

### Parameters

**i**

- int. Is only being read.

### Returns

`nv_bfloat16`

- ▶ `i` converted to `nv_bfloat16`.

### Description

Convert the signed integer value `i` to a `nv_bfloat16` floating-point value in round-up mode.

## `__device__ int2bfloat16_rz (const int i)`

Convert a signed integer to a `nv_bfloat16` in round-towards-zero mode.

### Parameters

**i**

- int. Is only being read.

### Returns

`nv_bfloat16`

- ▶ `i` converted to `nv_bfloat16`.

### Description

Convert the signed integer value `i` to a `nv_bfloat16` floating-point value in round-towards-zero mode.

`__device__ ldca (const __nv_bfloat16 *ptr)`

Generates a ``ld.global.ca`` load instruction.

### Parameters

#### **ptr**

- memory location

### Returns

The value pointed by ``ptr``

`__device__ ldca (const __nv_bfloat162 *ptr)`

Generates a ``ld.global.ca`` load instruction.

### Parameters

#### **ptr**

- memory location

### Returns

The value pointed by ``ptr``

`__device__ ldcg (const __nv_bfloat16 *ptr)`

Generates a ``ld.global.cg`` load instruction.

### Parameters

#### **ptr**

- memory location

### Returns

The value pointed by ``ptr``

`__device__ ldcg (const __nv_bfloat162 *ptr)`

Generates a ``ld.global.cg`` load instruction.

### Parameters

#### **ptr**

- memory location

### Returns

The value pointed by ``ptr``

`__device__ ldcv (const __nv_bfloat16 *ptr)`

Generates a ``ld.global.cv`` load instruction.

### Parameters

#### **ptr**

- memory location

### Returns

The value pointed by ``ptr``

`__device__ ldcg (const __nv_bfloat162 *ptr)`

Generates a ``ld.global.cs`` load instruction.

### Parameters

#### **ptr**

- memory location

### Returns

The value pointed by ``ptr``

`__device__ ldcv (const __nv_bfloat16 *ptr)`

Generates a ``ld.global.cg`` load instruction.

### Parameters

#### **ptr**

- memory location

### Returns

The value pointed by ``ptr``



`__device__ ldcv (const __nv_bfloat162 *ptr)`

Generates a ``ld.global.cv`` load instruction.

### Parameters

#### **ptr**

- memory location

### Returns

The value pointed by ``ptr``

`__device__ ldg (const __nv_bfloat16 *ptr)`

Generates a ``ld.global.nc`` load instruction.

### Parameters

#### **ptr**

- memory location

### Returns

The value pointed by ``ptr``

`__device__ ldg (const __nv_bfloat162 *ptr)`

Generates a ``ld.global.nc`` load instruction.

### Parameters

#### **ptr**

- memory location

### Returns

The value pointed by ``ptr``

`__device__ ldlu (const __nv_bfloat16 *ptr)`

Generates a ``ld.global.lu`` load instruction.

### Parameters

#### **ptr**

- memory location

### Returns

The value pointed by ``ptr``

`__device__ ldlu (const __nv_bfloat162 *ptr)`

Generates a `ld.global.lu` load instruction.

### Parameters

#### **ptr**

- memory location

### Returns

The value pointed by `ptr`

`__device__ ll2bfloat16_rd (const long long int i)`

Convert a signed 64-bit integer to a `nv_bfloat16` in round-down mode.

### Parameters

#### **i**

- long long int. Is only being read.

### Returns

`nv_bfloat16`

- ▶ `i` converted to `nv_bfloat16`.

### Description

Convert the signed 64-bit integer value `i` to a `nv_bfloat16` floating-point value in round-down mode.

`__host__ __device__ ll2bfloat16_rn (const long long int i)`

Convert a signed 64-bit integer to a `nv_bfloat16` in round-to-nearest-even mode.

### Parameters

#### **i**

- long long int. Is only being read.

### Returns

`nv_bfloat16`

- ▶ `i` converted to `nv_bfloat16`.

### Description

Convert the signed 64-bit integer value `i` to a `nv_bfloat16` floating-point value in round-to-nearest-even mode.

## `__device__ ll2bfloat16_ru (const long long int i)`

Convert a signed 64-bit integer to a `nv_bfloat16` in round-up mode.

### Parameters

**i**  
- long long int. Is only being read.

### Returns

`nv_bfloat16`

- ▶ `i` converted to `nv_bfloat16`.

### Description

Convert the signed 64-bit integer value `i` to a `nv_bfloat16` floating-point value in round-up mode.

## `__device__ ll2bfloat16_rz (const long long int i)`

Convert a signed 64-bit integer to a `nv_bfloat16` in round-towards-zero mode.

### Parameters

**i**  
- long long int. Is only being read.

### Returns

`nv_bfloat16`

- ▶ `i` converted to `nv_bfloat16`.

### Description

Convert the signed 64-bit integer value `i` to a `nv_bfloat16` floating-point value in round-towards-zero mode.

## `__host__ __device__ low2bfloat16 (const __nv_bfloat162 a)`

Returns low 16 bits of `nv_bfloat162` input.

### Parameters

**a**  
- `nv_bfloat162`. Is only being read.

## Returns

nv\_bfloat16

- ▶ Returns `nv_bfloat16` which contains low 16 bits of the input `a`.

## Description

Returns low 16 bits of `nv_bfloat162` input `a`.

```
__host__ __device__ low2bfloat162 (const __nv_bfloat162
a)
```

Extracts low 16 bits from `nv_bfloat162` input.

## Parameters

**a**

- `nv_bfloat162`. Is only being read.

## Returns

nv\_bfloat162

- ▶ The `nv_bfloat162` with both halves equal to the low 16 bits of the input.

## Description

Extracts low 16 bits from `nv_bfloat162` input `a` and returns a new `nv_bfloat162` number which has both halves equal to the extracted bits.

```
__host__ __device__ float __low2float (const __nv_bfloat162
a)
```

Converts low 16 bits of `nv_bfloat162` to `float` and returns the result.

## Parameters

**a**

- `nv_bfloat162`. Is only being read.

## Returns

float

- ▶ The low 16 bits of `a` converted to `float`.

## Description

Converts low 16 bits of `nv_bfloat162` input `a` to 32-bit floating-point number and returns the result.

```
__host__ __device__ lowhigh2highlow (const
__nv_bfloat162 a)
```

Swaps both halves of the `nv_bfloat162` input.

## Parameters

**a**

- `nv_bfloat162`. Is only being read.

## Returns

`nv_bfloat162`

- ▶ `a` with its halves being swapped.

## Description

Swaps both halves of the `nv_bfloat162` input and returns a new `nv_bfloat162` number with swapped halves.

```
__host__ __device__ lows2bfloat162 (const __nv_bfloat162
a, const __nv_bfloat162 b)
```

Extracts low 16 bits from each of the two `nv_bfloat162` inputs and combines into one `nv_bfloat162` number.

## Parameters

**a**

- `nv_bfloat162`. Is only being read.

**b**

- `nv_bfloat162`. Is only being read.

## Returns

`nv_bfloat162`

- ▶ The low 16 bits of `a` and of `b`.

## Description

Extracts low 16 bits from each of the two `nv_bfloat162` inputs and combines into one `nv_bfloat162` number. Low 16 bits from input `a` is stored in low 16 bits of the return value, low 16 bits from input `b` is stored in high 16 bits of the return value.

`__device__ shfl_down_sync (const unsigned mask, const __nv_bfloat16 var, const unsigned int delta, const int width)`

Exchange a variable between threads within a warp. Copy from a thread with higher ID relative to the caller.

## Parameters

### mask

- unsigned int. Is only being read.

### var

- `nv_bfloat16`. Is only being read.

### delta

- int. Is only being read.

### width

- int. Is only being read.

## Returns

Returns the 2-byte word referenced by `var` from the source thread ID as `nv_bfloat16`. If the source thread ID is out of range or the source thread has exited, the calling thread's own `var` is returned.

## Description

Calculates a source thread ID by adding `delta` to the caller's thread ID. The value of `var` held by the resulting thread ID is returned: this has the effect of shifting `var` down the warp by `delta` threads. If `width` is less than `warpSize` then each subsection of the warp behaves as a separate entity with a starting logical thread ID of 0. As for `shfl_up_sync()`, the ID number of the source thread will not wrap around the value of `width` and so the upper `delta` threads will remain unchanged.



### Note:

For more details for this function see the Warp Shuffle Functions section in the CUDA C++ Programming Guide.

`__device__ shfl_down_sync (const unsigned mask, const __nv_bfloat162 var, const unsigned int delta, const int width)`

Exchange a variable between threads within a warp. Copy from a thread with higher ID relative to the caller.

### Parameters

#### **mask**

- unsigned int. Is only being read.

#### **var**

- `nv_bfloat162`. Is only being read.

#### **delta**

- int. Is only being read.

#### **width**

- int. Is only being read.

### Returns

Returns the 4-byte word referenced by `var` from the source thread ID as `nv_bfloat162`. If the source thread ID is out of range or the source thread has exited, the calling thread's own `var` is returned.

### Description

Calculates a source thread ID by adding `delta` to the caller's thread ID. The value of `var` held by the resulting thread ID is returned: this has the effect of shifting `var` down the warp by `delta` threads. If `width` is less than `warpSize` then each subsection of the warp behaves as a separate entity with a starting logical thread ID of 0. As for `shfl_up_sync()`, the ID number of the source thread will not wrap around the value of `width` and so the upper `delta` threads will remain unchanged.



#### **Note:**

For more details for this function see the Warp Shuffle Functions section in the CUDA C++ Programming Guide.

`__device__ shfl_sync (const unsigned mask, const __nv_bfloat16 var, const int delta, const int width)`

Exchange a variable between threads within a warp. Direct copy from indexed thread.

### Parameters

#### **mask**

- unsigned int. Is only being read.

**var**

- nv\_bfloat16. Is only being read.

**delta**

- int. Is only being read.

**width**

- int. Is only being read.

## Returns

Returns the 2-byte word referenced by var from the source thread ID as nv\_bfloat16. If the source thread ID is out of range or the source thread has exited, the calling thread's own var is returned.

## Description

Returns the value of var held by the thread whose ID is given by delta. If width is less than warpSize then each subsection of the warp behaves as a separate entity with a starting logical thread ID of 0. If delta is outside the range [0:width-1], the value returned corresponds to the value of var held by the delta modulo width (i.e. within the same subsection). width must have a value which is a power of 2; results are undefined if width is not a power of 2, or is a number greater than warpSize.

**Note:**

For more details for this function see the Warp Shuffle Functions section in the CUDA C++ Programming Guide.

## `__device__ shfl_sync (const unsigned mask, const __nv_bfloat162 var, const int delta, const int width)`

Exchange a variable between threads within a warp. Direct copy from indexed thread.

## Parameters

**mask**

- unsigned int. Is only being read.

**var**

- nv\_bfloat162. Is only being read.

**delta**

- int. Is only being read.

**width**

- int. Is only being read.



## Returns

Returns the 4-byte word referenced by `var` from the source thread ID as `nv_bfloat162`. If the source thread ID is out of range or the source thread has exited, the calling thread's own `var` is returned.

## Description

Returns the value of `var` held by the thread whose ID is given by `delta`. If `width` is less than `warpSize` then each subsection of the warp behaves as a separate entity with a starting logical thread ID of 0. If `delta` is outside the range `[0:width-1]`, the value returned corresponds to the value of `var` held by the `delta` modulo `width` (i.e. within the same subsection). `width` must have a value which is a power of 2; results are undefined if `width` is not a power of 2, or is a number greater than `warpSize`.



### Note:

For more details for this function see the Warp Shuffle Functions section in the CUDA C++ Programming Guide.

`__device__ shfl_up_sync (const unsigned mask, const __nv_bfloat16 var, const unsigned int delta, const int width)`

Exchange a variable between threads within a warp. Copy from a thread with lower ID relative to the caller.

## Parameters

### mask

- unsigned int. Is only being read.

### var

- `nv_bfloat16`. Is only being read.

### delta

- int. Is only being read.

### width

- int. Is only being read.

## Returns

Returns the 2-byte word referenced by `var` from the source thread ID as `nv_bfloat16`. If the source thread ID is out of range or the source thread has exited, the calling thread's own `var` is returned.

## Description

Calculates a source thread ID by subtracting delta from the caller's lane ID. The value of var held by the resulting lane ID is returned: in effect, var is shifted up the warp by delta threads. If width is less than warpSize then each subsection of the warp behaves as a separate entity with a starting logical thread ID of 0. The source thread index will not wrap around the value of width, so effectively the lower delta threads will be unchanged. width must have a value which is a power of 2; results are undefined if width is not a power of 2, or is a number greater than warpSize.



### Note:

For more details for this function see the Warp Shuffle Functions section in the CUDA C++ Programming Guide.

**\_\_device\_\_ shfl\_up\_sync (const unsigned mask, const \_\_nv\_bfloat162 var, const unsigned int delta, const int width)**

Exchange a variable between threads within a warp. Copy from a thread with lower ID relative to the caller.

## Parameters

### mask

- unsigned int. Is only being read.

### var

- nv\_bfloat162. Is only being read.

### delta

- int. Is only being read.

### width

- int. Is only being read.

## Returns

Returns the 4-byte word referenced by var from the source thread ID as nv\_bfloat162. If the source thread ID is out of range or the source thread has exited, the calling thread's own var is returned.

## Description

Calculates a source thread ID by subtracting delta from the caller's lane ID. The value of var held by the resulting lane ID is returned: in effect, var is shifted up the warp by delta threads. If width is less than warpSize then each subsection of the warp behaves as a separate entity with a starting logical thread ID of 0. The source thread index will not wrap around the value of width, so effectively the lower delta threads will be unchanged. width must have a value which

is a power of 2; results are undefined if width is not a power of 2, or is a number greater than warpSize.

**Note:**

For more details for this function see the Warp Shuffle Functions section in the CUDA C++ Programming Guide.

## `__device__ shfl_xor_sync (const unsigned mask, const __nv_bfloat16 var, const int delta, const int width)`

Exchange a variable between threads within a warp. Copy from a thread based on bitwise XOR of own thread ID.

### Parameters

**mask**

- unsigned int. Is only being read.

**var**

- nv\_bfloat16. Is only being read.

**delta**

- int. Is only being read.

**width**

- int. Is only being read.

### Returns

Returns the 2-byte word referenced by var from the source thread ID as nv\_bfloat16. If the source thread ID is out of range or the source thread has exited, the calling thread's own var is returned.

### Description

Calculates a source thread ID by performing a bitwise XOR of the caller's thread ID with mask: the value of var held by the resulting thread ID is returned. If width is less than warpSize then each group of width consecutive threads are able to access elements from earlier groups of threads, however if they attempt to access elements from later groups of threads their own value of var will be returned. This mode implements a butterfly addressing pattern such as is used in tree reduction and broadcast.

**Note:**

For more details for this function see the Warp Shuffle Functions section in the CUDA C++ Programming Guide.

## `__device__ shfl_xor_sync (const unsigned mask, const __nv_bfloat162 var, const int delta, const int width)`

Exchange a variable between threads within a warp. Copy from a thread based on bitwise XOR of own thread ID.

### Parameters

#### **mask**

- unsigned int. Is only being read.

#### **var**

- `nv_bfloat162`. Is only being read.

#### **delta**

- int. Is only being read.

#### **width**

- int. Is only being read.

### Returns

Returns the 4-byte word referenced by `var` from the source thread ID as `nv_bfloat162`. If the source thread ID is out of range or the source thread has exited, the calling thread's own `var` is returned.

### Description

Calculates a source thread ID by performing a bitwise XOR of the caller's thread ID with `mask`: the value of `var` held by the resulting thread ID is returned. If `width` is less than `warpSize` then each group of `width` consecutive threads are able to access elements from earlier groups of threads, however if they attempt to access elements from later groups of threads their own value of `var` will be returned. This mode implements a butterfly addressing pattern such as is used in tree reduction and broadcast.



#### **Note:**

For more details for this function see the Warp Shuffle Functions section in the CUDA C++ Programming Guide.

## `__device__ short2bfloat16_rd (const short int i)`

Convert a signed short integer to a `nv_bfloat16` in round-down mode.

### Parameters

#### **i**

- short int. Is only being read.

## Returns

nv\_bfloat16

- ▶ `i` converted to nv\_bfloat16.

## Description

Convert the signed short integer value `i` to a nv\_bfloat16 floating-point value in round-down mode.

### `__host__ __device__ short2bfloat16_rn (const short int i)`

Convert a signed short integer to a nv\_bfloat16 in round-to-nearest-even mode.

## Parameters

**i**

- short int. Is only being read.

## Returns

nv\_bfloat16

- ▶ `i` converted to nv\_bfloat16.

## Description

Convert the signed short integer value `i` to a nv\_bfloat16 floating-point value in round-to-nearest-even mode.

### `__device__ short2bfloat16_ru (const short int i)`

Convert a signed short integer to a nv\_bfloat16 in round-up mode.

## Parameters

**i**

- short int. Is only being read.

## Returns

nv\_bfloat16

- ▶ `i` converted to nv\_bfloat16.

## Description

Convert the signed short integer value `i` to a nv\_bfloat16 floating-point value in round-up mode.

## `__device__ short2bfloat16_rz (const short int i)`

Convert a signed short integer to a `nv_bfloat16` in round-towards-zero mode.

### Parameters

**i**

- short int. Is only being read.

### Returns

`nv_bfloat16`

- ▶ `i` converted to `nv_bfloat16`.

### Description

Convert the signed short integer value `i` to a `nv_bfloat16` floating-point value in round-towards-zero mode.

## `__host__ __device__ short_as_bfloat16 (const short int i)`

Reinterprets bits in a signed short integer as a `nv_bfloat16`.

### Parameters

**i**

- short int. Is only being read.

### Returns

`nv_bfloat16`

- ▶ The reinterpreted value.

### Description

Reinterprets the bits in the signed short integer `i` as a `nv_bfloat16` floating-point number.

## `__device__ void __stcg (const __nv_bfloat16 *ptr, const __nv_bfloat16 value)`

Generates a ``st.global.cg`` store instruction.

### Parameters

**ptr**

- memory location

**value**

- the value to be stored

```
__device__ void __stcg (const __nv_bfloat162 *ptr, const
__nv_bfloat162 value)
```

Generates a `st.global.cg` store instruction.

### Parameters

#### **ptr**

- memory location

#### **value**

- the value to be stored

```
__device__ void __stcs (const __nv_bfloat16 *ptr, const
__nv_bfloat16 value)
```

Generates a `st.global.cs` store instruction.

### Parameters

#### **ptr**

- memory location

#### **value**

- the value to be stored

```
__device__ void __stcs (const __nv_bfloat162 *ptr, const
__nv_bfloat162 value)
```

Generates a `st.global.cs` store instruction.

### Parameters

#### **ptr**

- memory location

#### **value**

- the value to be stored

```
__device__ void __stwb (const __nv_bfloat16 *ptr, const
__nv_bfloat16 value)
```

Generates a `st.global.wb` store instruction.

### Parameters

#### **ptr**

- memory location

#### **value**

- the value to be stored

```
__device__ void __stwb (const __nv_bfloat162 *ptr, const
__nv_bfloat162 value)
```

Generates a `st.global.wb` store instruction.

### Parameters

#### **ptr**

- memory location

#### **value**

- the value to be stored

```
__device__ void __stwt (const __nv_bfloat16 *ptr, const
__nv_bfloat16 value)
```

Generates a `st.global.wt` store instruction.

### Parameters

#### **ptr**

- memory location

#### **value**

- the value to be stored

```
__device__ void __stwt (const __nv_bfloat162 *ptr, const
__nv_bfloat162 value)
```

Generates a `st.global.wt` store instruction.

### Parameters

#### **ptr**

- memory location

#### **value**

- the value to be stored

```
__device__ __uint2bfloat16_rd (const unsigned int i)
```

Convert an unsigned integer to a nv\_bfloat16 in round-down mode.

### Parameters

#### **i**

- unsigned int. Is only being read.

### Returns

nv\_bfloat16



- ▶ `i` converted to `nv_bfloat16`.

### Description

Convert the unsigned integer value `i` to a `nv_bfloat16` floating-point value in round-down mode.

### `__host__ __device__ uint2bfloat16_rn (const unsigned int i)`

Convert an unsigned integer to a `nv_bfloat16` in round-to-nearest-even mode.

### Parameters

**i**

- unsigned int. Is only being read.

### Returns

`nv_bfloat16`

- ▶ `i` converted to `nv_bfloat16`.

### Description

Convert the unsigned integer value `i` to a `nv_bfloat16` floating-point value in round-to-nearest-even mode.

### `__device__ uint2bfloat16_ru (const unsigned int i)`

Convert an unsigned integer to a `nv_bfloat16` in round-up mode.

### Parameters

**i**

- unsigned int. Is only being read.

### Returns

`nv_bfloat16`

- ▶ `i` converted to `nv_bfloat16`.

### Description

Convert the unsigned integer value `i` to a `nv_bfloat16` floating-point value in round-up mode.

## `__device__ uint2bfloat16_rz (const unsigned int i)`

Convert an unsigned integer to a `nv_bfloat16` in round-towards-zero mode.

### Parameters

**i**  
- unsigned int. Is only being read.

### Returns

`nv_bfloat16`

- ▶ `i` converted to `nv_bfloat16`.

### Description

Convert the unsigned integer value `i` to a `nv_bfloat16` floating-point value in round-towards-zero mode.

## `__device__ ull2bfloat16_rd (const unsigned long long int i)`

Convert an unsigned 64-bit integer to a `nv_bfloat16` in round-down mode.

### Parameters

**i**  
- unsigned long long int. Is only being read.

### Returns

`nv_bfloat16`

- ▶ `i` converted to `nv_bfloat16`.

### Description

Convert the unsigned 64-bit integer value `i` to a `nv_bfloat16` floating-point value in round-down mode.

## `__host__ __device__ ull2bfloat16_rn (const unsigned long long int i)`

Convert an unsigned 64-bit integer to a `nv_bfloat16` in round-to-nearest-even mode.

### Parameters

**i**  
- unsigned long long int. Is only being read.

## Returns

nv\_bfloat16

- ▶ `i` converted to nv\_bfloat16.

## Description

Convert the unsigned 64-bit integer value `i` to a nv\_bfloat16 floating-point value in round-to-nearest-even mode.

## \_\_device\_\_ ull2bfloat16\_ru (const unsigned long long int i)

Convert an unsigned 64-bit integer to a nv\_bfloat16 in round-up mode.

## Parameters

**i**

- unsigned long long int. Is only being read.

## Returns

nv\_bfloat16

- ▶ `i` converted to nv\_bfloat16.

## Description

Convert the unsigned 64-bit integer value `i` to a nv\_bfloat16 floating-point value in round-up mode.

## \_\_device\_\_ ull2bfloat16\_rz (const unsigned long long int i)

Convert an unsigned 64-bit integer to a nv\_bfloat16 in round-towards-zero mode.

## Parameters

**i**

- unsigned long long int. Is only being read.

## Returns

nv\_bfloat16

- ▶ `i` converted to nv\_bfloat16.

## Description

Convert the unsigned 64-bit integer value `i` to a nv\_bfloat16 floating-point value in round-towards-zero mode.

## `__device__ ushort2bfloat16_rd (const unsigned short int i)`

Convert an unsigned short integer to a `nv_bfloat16` in round-down mode.

### Parameters

**i**  
- unsigned short int. Is only being read.

### Returns

`nv_bfloat16`

► `i` converted to `nv_bfloat16`.

### Description

Convert the unsigned short integer value `i` to a `nv_bfloat16` floating-point value in round-down mode.

## `__host__ __device__ ushort2bfloat16_rn (const unsigned short int i)`

Convert an unsigned short integer to a `nv_bfloat16` in round-to-nearest-even mode.

### Parameters

**i**  
- unsigned short int. Is only being read.

### Returns

`nv_bfloat16`

► `i` converted to `nv_bfloat16`.

### Description

Convert the unsigned short integer value `i` to a `nv_bfloat16` floating-point value in round-to-nearest-even mode.

## `__device__ ushort2bfloat16_ru (const unsigned short int i)`

Convert an unsigned short integer to a `nv_bfloat16` in round-up mode.

### Parameters

**i**  
- unsigned short int. Is only being read.

## Returns

nv\_bfloat16

- ▶ `i` converted to nv\_bfloat16.

## Description

Convert the unsigned short integer value `i` to a nv\_bfloat16 floating-point value in round-up mode.

## \_\_device\_\_ ushort2bfloat16\_rz (const unsigned short int i)

Convert an unsigned short integer to a nv\_bfloat16 in round-towards-zero mode.

## Parameters

**i**

- unsigned short int. Is only being read.

## Returns

nv\_bfloat16

- ▶ `i` converted to nv\_bfloat16.

## Description

Convert the unsigned short integer value `i` to a nv\_bfloat16 floating-point value in round-towards-zero mode.

## \_\_host\_\_ device\_\_ ushort\_as\_bfloat16 (const unsigned short int i)

Reinterprets bits in an unsigned short integer as a nv\_bfloat16.

## Parameters

**i**

- unsigned short int. Is only being read.

## Returns

nv\_bfloat16

- ▶ The reinterpreted value.

## Description

Reinterprets the bits in the unsigned short integer `i` as a nv\_bfloat16 floating-point number.

**\_\_host\_\_ \_\_device\_\_ make\_bfloat162 (const \_\_nv\_bfloat16 x, const \_\_nv\_bfloat16 y)**

Vector function, combines two `nv_bfloat16` numbers into one `nv_bfloat162` number.

### Parameters

**x**

- `nv_bfloat16`. Is only being read.

**y**

- `nv_bfloat16`. Is only being read.

### Returns

`nv_bfloat162`

- ▶ The `nv_bfloat162` vector with one half equal to `x` and the other to `y`.

### Description

Combines two input `nv_bfloat16` number `x` and `y` into one `nv_bfloat162` number. Input `x` is stored in low 16 bits of the return value, input `y` is stored in high 16 bits of the return value.

## 1.3.7. Bfloat16 Math Functions

Bfloat16 Precision Ininsics

To use these functions, include the header file `cuda_bf16.h` in your program.

**\_\_device\_\_ hceil (const \_\_nv\_bfloat16 h)**

Calculate ceiling of the input argument.

### Parameters

**h**

- `nv_bfloat16`. Is only being read.

### Returns

`nv_bfloat16`

- ▶ The smallest integer value not less than `h`.

### Description

Compute the smallest integer value not less than `h`.

## `__device__ hcos (const __nv_bfloat16 a)`

Calculates `nv_bfloat16` cosine in round-to-nearest-even mode.

### Parameters

**a**

- `nv_bfloat16`. Is only being read.

### Returns

`nv_bfloat16`

- ▶ The cosine of `a`.

### Description

Calculates `nv_bfloat16` cosine of input `a` in round-to-nearest-even mode.

NOTE: this function's implementation calls `cosf(float)` function and is exposed to compiler optimizations. Specifically, `--use_fast_math` flag changes `cosf(float)` into an intrinsic `__cosf(float)`, which has less accurate numeric behavior.

## `__device__ hexp (const __nv_bfloat16 a)`

Calculates `nv_bfloat16` natural exponential function in round-to-nearest-even mode.

### Parameters

**a**

- `nv_bfloat16`. Is only being read.

### Returns

`nv_bfloat16`

- ▶ The natural exponential function on `a`.

### Description

Calculates `nv_bfloat16` natural exponential function of input `a` in round-to-nearest-even mode.

## `__device__ hexp10 (const __nv_bfloat16 a)`

Calculates `nv_bfloat16` decimal exponential function in round-to-nearest-even mode.

### Parameters

**a**

- `nv_bfloat16`. Is only being read.

## Returns

nv\_bfloat16

- ▶ The decimal exponential function on a.

## Description

Calculates nv\_bfloat16 decimal exponential function of input a in round-to-nearest-even mode.

## \_\_device\_\_ hexp2 (const \_\_nv\_bfloat16 a)

Calculates nv\_bfloat16 binary exponential function in round-to-nearest-even mode.

## Parameters

**a**

- nv\_bfloat16. Is only being read.

## Returns

nv\_bfloat16

- ▶ The binary exponential function on a.

## Description

Calculates nv\_bfloat16 binary exponential function of input a in round-to-nearest-even mode.

## \_\_device\_\_ hfloor (const \_\_nv\_bfloat16 h)

Calculate the largest integer less than or equal to h.

## Parameters

**h**

- nv\_bfloat16. Is only being read.

## Returns

nv\_bfloat16

- ▶ The largest integer value which is less than or equal to h.

## Description

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



## `__device__ hlog (const __nv_bfloat16 a)`

Calculates `nv_bfloat16` natural logarithm in round-to-nearest-even mode.

### Parameters

**a**

- `nv_bfloat16`. Is only being read.

### Returns

`nv_bfloat16`

- ▶ The natural logarithm of `a`.

### Description

Calculates `nv_bfloat16` natural logarithm of input `a` in round-to-nearest-even mode.

## `__device__ hlog10 (const __nv_bfloat16 a)`

Calculates `nv_bfloat16` decimal logarithm in round-to-nearest-even mode.

### Parameters

**a**

- `nv_bfloat16`. Is only being read.

### Returns

`nv_bfloat16`

- ▶ The decimal logarithm of `a`.

### Description

Calculates `nv_bfloat16` decimal logarithm of input `a` in round-to-nearest-even mode.

## `__device__ hlog2 (const __nv_bfloat16 a)`

Calculates `nv_bfloat16` binary logarithm in round-to-nearest-even mode.

### Parameters

**a**

- `nv_bfloat16`. Is only being read.

### Returns

`nv_bfloat16`

- ▶ The binary logarithm of `a`.

## Description

Calculates `nv_bfloat16` binary logarithm of input `a` in round-to-nearest-even mode.

## `__device__ hrcp (const __nv_bfloat16 a)`

Calculates `nv_bfloat16` reciprocal in round-to-nearest-even mode.

## Parameters

**a**

- `nv_bfloat16`. Is only being read.

## Returns

`nv_bfloat16`

- ▶ The reciprocal of `a`.

## Description

Calculates `nv_bfloat16` reciprocal of input `a` in round-to-nearest-even mode.

## `__device__ hrint (const __nv_bfloat16 h)`

Round input to nearest integer value in `nv_bfloat16` floating-point number.

## Parameters

**h**

- `nv_bfloat16`. Is only being read.

## Returns

`nv_bfloat16`

- ▶ The nearest integer to `h`.

## Description

Round `h` to the nearest integer value in `nv_bfloat16` floating-point format, with `bfloat16` way cases rounded to the nearest even integer value.

## `__device__ hrsqrt (const __nv_bfloat16 a)`

Calculates `nv_bfloat16` reciprocal square root in round-to-nearest-even mode.

### Parameters

**a**

- `nv_bfloat16`. Is only being read.

### Returns

`nv_bfloat16`

► The reciprocal square root of `a`.

### Description

Calculates `nv_bfloat16` reciprocal square root of input `a` in round-to-nearest-even mode.

## `__device__ hsin (const __nv_bfloat16 a)`

Calculates `nv_bfloat16` sine in round-to-nearest-even mode.

### Parameters

**a**

- `nv_bfloat16`. Is only being read.

### Returns

`nv_bfloat16`

► The sine of `a`.

### Description

Calculates `nv_bfloat16` sine of input `a` in round-to-nearest-even mode.

NOTE: this function's implementation calls `sinf(float)` function and is exposed to compiler optimizations. Specifically, `--use_fast_math` flag changes `sinf(float)` into an intrinsic `__sinf(float)`, which has less accurate numeric behavior.

## `__device__ hsqrt (const __nv_bfloat16 a)`

Calculates `nv_bfloat16` square root in round-to-nearest-even mode.

### Parameters

**a**

- `nv_bfloat16`. Is only being read.

## Returns

nv\_bfloat16

- ▶ The square root of a.

## Description

Calculates nv\_bfloat16 square root of input a in round-to-nearest-even mode.

## \_\_device\_\_ htrunc (const \_\_nv\_bfloat16 h)

Truncate input argument to the integral part.

## Parameters

**h**

- nv\_bfloat16. Is only being read.

## Returns

nv\_bfloat16

- ▶ The truncated integer value.

## Description

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

## 1.3.8. Bfloat162 Math Functions

Bfloat16 Precision Ininsics

To use these functions, include the header file `cuda_bf16.h` in your program.

## \_\_device\_\_ h2ceil (const \_\_nv\_bfloat162 h)

Calculate nv\_bfloat162 vector ceiling of the input argument.

## Parameters

**h**

- nv\_bfloat162. Is only being read.

## Returns

nv\_bfloat162

- ▶ The vector of smallest integers not less than h.

## Description

For each component of vector `h` compute the smallest integer value not less than `h`.

## `__device__ h2cos (const __nv_bfloat162 a)`

Calculates `nv_bfloat162` vector cosine in round-to-nearest-even mode.

## Parameters

**a**

- `nv_bfloat162`. Is only being read.

## Returns

`nv_bfloat162`

- The elementwise cosine on vector `a`.

## Description

Calculates `nv_bfloat162` cosine of input vector `a` in round-to-nearest-even mode.

NOTE: this function's implementation calls `cosf(float)` function and is exposed to compiler optimizations. Specifically, `--use_fast_math` flag changes `cosf(float)` into an intrinsic `__cosf(float)`, which has less accurate numeric behavior.

## `__device__ h2exp (const __nv_bfloat162 a)`

Calculates `nv_bfloat162` vector exponential function in round-to-nearest-even mode.

## Parameters

**a**

- `nv_bfloat162`. Is only being read.

## Returns

`nv_bfloat162`

- The elementwise exponential function on vector `a`.

## Description

Calculates `nv_bfloat162` exponential function of input vector `a` in round-to-nearest-even mode.

## `__device__ h2exp10 (const __nv_bfloat162 a)`

Calculates `nv_bfloat162` vector decimal exponential function in round-to-nearest-even mode.

### Parameters

**a**

- `nv_bfloat162`. Is only being read.

### Returns

`nv_bfloat162`

- ▶ The elementwise decimal exponential function on vector `a`.

### Description

Calculates `nv_bfloat162` decimal exponential function of input vector `a` in round-to-nearest-even mode.

## `__device__ h2exp2 (const __nv_bfloat162 a)`

Calculates `nv_bfloat162` vector binary exponential function in round-to-nearest-even mode.

### Parameters

**a**

- `nv_bfloat162`. Is only being read.

### Returns

`nv_bfloat162`

- ▶ The elementwise binary exponential function on vector `a`.

### Description

Calculates `nv_bfloat162` binary exponential function of input vector `a` in round-to-nearest-even mode.

## `__device__ h2floor (const __nv_bfloat162 h)`

Calculate the largest integer less than or equal to `h`.

### Parameters

**h**

- `nv_bfloat162`. Is only being read.

## Returns

nv\_bfloat162

- ▶ The vector of largest integers which is less than or equal to  $h$ .

## Description

For each component of vector  $h$  calculate the largest integer value which is less than or equal to  $h$ .

## \_\_device\_\_ h2log (const \_\_nv\_bfloat162 a)

Calculates nv\_bfloat162 vector natural logarithm in round-to-nearest-even mode.

## Parameters

**a**

- nv\_bfloat162. Is only being read.

## Returns

nv\_bfloat162

- ▶ The elementwise natural logarithm on vector  $a$ .

## Description

Calculates nv\_bfloat162 natural logarithm of input vector  $a$  in round-to-nearest-even mode.

## \_\_device\_\_ h2log10 (const \_\_nv\_bfloat162 a)

Calculates nv\_bfloat162 vector decimal logarithm in round-to-nearest-even mode.

## Parameters

**a**

- nv\_bfloat162. Is only being read.

## Returns

nv\_bfloat162

- ▶ The elementwise decimal logarithm on vector  $a$ .

## Description

Calculates nv\_bfloat162 decimal logarithm of input vector  $a$  in round-to-nearest-even mode.

## \_\_device\_\_ h2log2 (const \_\_nv\_bfloat162 a)

Calculates `nv_bfloat162` vector binary logarithm in round-to-nearest-even mode.

### Parameters

**a**

- `nv_bfloat162`. Is only being read.

### Returns

`nv_bfloat162`

- ▶ The elementwise binary logarithm on vector `a`.

### Description

Calculates `nv_bfloat162` binary logarithm of input vector `a` in round-to-nearest-even mode.

## \_\_device\_\_ h2rcp (const \_\_nv\_bfloat162 a)

Calculates `nv_bfloat162` vector reciprocal in round-to-nearest-even mode.

### Parameters

**a**

- `nv_bfloat162`. Is only being read.

### Returns

`nv_bfloat162`

- ▶ The elementwise reciprocal on vector `a`.

### Description

Calculates `nv_bfloat162` reciprocal of input vector `a` in round-to-nearest-even mode.

## \_\_device\_\_ h2rint (const \_\_nv\_bfloat162 h)

Round input to nearest integer value in `nv_bfloat16` floating-point number.

### Parameters

**h**

- `nv_bfloat162`. Is only being read.

### Returns

`nv_bfloat162`

- ▶ The vector of rounded integer values.



## Description

Round each component of `nv_bfloat162` vector `h` to the nearest integer value in `nv_bfloat16` floating-point format, with `bfloat16` cases rounded to the nearest even integer value.

## `__device__ h2rsqrt (const __nv_bfloat162 a)`

Calculates `nv_bfloat162` vector reciprocal square root in round-to-nearest-even mode.

## Parameters

**a**

- `nv_bfloat162`. Is only being read.

## Returns

`nv_bfloat162`

- ▶ The elementwise reciprocal square root on vector `a`.

## Description

Calculates `nv_bfloat162` reciprocal square root of input vector `a` in round-to-nearest-even mode.

## `__device__ h2sin (const __nv_bfloat162 a)`

Calculates `nv_bfloat162` vector sine in round-to-nearest-even mode.

## Parameters

**a**

- `nv_bfloat162`. Is only being read.

## Returns

`nv_bfloat162`

- ▶ The elementwise sine on vector `a`.

## Description

Calculates `nv_bfloat162` sine of input vector `a` in round-to-nearest-even mode.

NOTE: this function's implementation calls `sinf(float)` function and is exposed to compiler optimizations. Specifically, `--use_fast_math` flag changes `sinf(float)` into an intrinsic `__sinf(float)`, which has less accurate numeric behavior.

## `__device__ h2sqrt (const __nv_bfloat162 a)`

Calculates `nv_bfloat162` vector square root in round-to-nearest-even mode.

### Parameters

**a**

- `nv_bfloat162`. Is only being read.

### Returns

`nv_bfloat162`

- The elementwise square root on vector `a`.

### Description

Calculates `nv_bfloat162` square root of input vector `a` in round-to-nearest-even mode.

## `__device__ h2trunc (const __nv_bfloat162 h)`

Truncate `nv_bfloat162` vector input argument to the integral part.

### Parameters

**h**

- `nv_bfloat162`. Is only being read.

### Returns

`nv_bfloat162`

- The truncated `h`.

### Description

Round each component of vector `h` to the nearest integer value that does not exceed `h` in magnitude.

## 1.4. Mathematical Functions

CUDA mathematical functions are always available in device code.

Host implementations of the common mathematical functions are mapped in a platform-specific way to standard math library functions, provided by the host compiler and respective host libm where available. Some functions, not available with the host compilers, are implemented in `crt/math_functions.hpp` header file. For example, see [erfinv\(\)](#). Other, less common functions, like [rhypot\(\)](#), [cyl\\_bessel\\_i0\(\)](#) are only available in device code.

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

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

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

## 1.5. Single Precision Mathematical Functions

This section describes single precision mathematical functions. To use these functions you do not need to include any additional header files in your program.

### `__device__ float acosf(float x)`

Calculate the arc cosine of the input argument.

#### Returns

Result will be in radians, in the interval  $[0, \pi]$  for  $x$  inside  $[-1, +1]$ .

- ▶ `acosf(1)` returns  $+0$ .
- ▶ `acosf(x)` returns NaN for  $x$  outside  $[-1, +1]$ .

#### Description

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.

### `__device__ float acoshf(float x)`

Calculate the nonnegative inverse hyperbolic cosine of the input argument.

#### Returns

Result will be in the interval  $[0, +\infty]$ .

- ▶ `acoshf(1)` returns 0.
- ▶ `acoshf(x)` returns NaN for  $x$  in the interval  $[-\infty, 1)$ .
- ▶ `acoshf(+\infty)` returns  $+\infty$ .

## Description

Calculate the nonnegative inverse hyperbolic cosine of the input argument  $x$ .



### Note:

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

## `__device__ float asinf (float x)`

Calculate the arc sine of the input argument.

## Returns

Result will be in radians, in the interval  $[-\pi/2, +\pi/2]$  for  $x$  inside  $[-1, +1]$ .

- ▶ `asinf( $\pm 0$ )` returns  $\pm 0$ .
- ▶ `asinf(x)` returns NaN for  $x$  outside  $[-1, +1]$ .

## Description

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.

## `__device__ float asinhf (float x)`

Calculate the inverse hyperbolic sine of the input argument.

## Returns

- ▶ `asinhf( $\pm 0$ )` returns  $\pm 0$ .
- ▶ `asinhf( $\pm \infty$ )` returns  $\pm \infty$ .

## Description

Calculate the inverse hyperbolic sine of the input argument  $x$ .

**Note:**

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

## `__device__ float atan2f (float y, float x)`

Calculate the arc tangent of the ratio of first and second input arguments.

### Returns

Result will be in radians, in the interval  $[-\pi, +\pi]$ .

- ▶ `atan2f( ±0, -0)` returns  $\pm\pi$ .
- ▶ `atan2f( ±0, +0)` returns  $\pm 0$ .
- ▶ `atan2f( ±0, x)` returns  $\pm\pi$  for  $x < 0$ .
- ▶ `atan2f( ±0, x)` returns  $\pm 0$  for  $x > 0$ .
- ▶ `atan2f(y, ±0)` returns  $-\pi/2$  for  $y < 0$ .
- ▶ `atan2f(y, ±0)` returns  $\pi/2$  for  $y > 0$ .
- ▶ `atan2f( ±y, -∞)` returns  $\pm\pi$  for finite  $y > 0$ .
- ▶ `atan2f( ±y, +∞)` returns  $\pm 0$  for finite  $y > 0$ .
- ▶ `atan2f( ±∞, x)` returns  $\pm\pi/2$  for finite  $x$ .
- ▶ `atan2f( ±∞, -∞)` returns  $\pm 3\pi/4$ .
- ▶ `atan2f( ±∞, +∞)` returns  $\pm\pi/4$ .

### Description

Calculate the principal value of the arc tangent of the ratio of first and second input arguments  $y/x$ . The quadrant of the result is determined by the signs of inputs  $y$  and  $x$ .

**Note:**

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

## `__device__ float atanf (float x)`

Calculate the arc tangent of the input argument.

### Returns

Result will be in radians, in the interval  $[-\pi/2, +\pi/2]$ .

- ▶ `atanf( ±0 )` returns  $\pm 0$ .
- ▶ `atanf( ±∞ )` returns  $\pm \pi / 2$ .

### Description

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.

## `__device__ float atanhf (float x)`

Calculate the inverse hyperbolic tangent of the input argument.

### Returns

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

### Description

Calculate the inverse hyperbolic tangent of the input argument  $x$ .



#### Note:

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

## \_\_device\_\_ float cbrtf (float x)

Calculate the cube root of the input argument.

### Returns

Returns  $x^{1/3}$ .

- ▶ `cbrtf( ±0 )` returns  $±0$ .
- ▶ `cbrtf( ±∞ )` returns  $±∞$ .

### Description

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.

## \_\_device\_\_ float ceilf (float x)

Calculate ceiling of the input argument.

### Returns

Returns  $\lceil x \rceil$  expressed as a floating-point number.

- ▶ `ceilf( ±0 )` returns  $±0$ .
- ▶ `ceilf( ±∞ )` returns  $±∞$ .

### Description

Compute the smallest integer value not less than  $x$ .

## \_\_device\_\_ float copysignf (float x, float y)

Create value with given magnitude, copying sign of second value.

### Returns

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

### Description

Create a floating-point value with the magnitude  $x$  and the sign of  $y$ .

## `__device__ float cosf (float x)`

Calculate the cosine of the input argument.

### Returns

- ▶ `cosf( ±0 )` returns 1.
- ▶ `cosf( ±∞ )` returns NaN.

### Description

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.
- ▶ This function is affected by the `--use_fast_math` compiler flag. See the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section for a complete list of functions affected.

## `__device__ float coshf (float x)`

Calculate the hyperbolic cosine of the input argument.

### Returns

- ▶ `coshf( ±0 )` returns 1.
- ▶ `coshf( ±∞ )` returns  $+\infty$ .

### Description

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.



## \_\_device\_\_ float cospif (float x)

Calculate the cosine of the input argument  $\times \pi$ .

### Returns

- ▶ `cospif( ±0 )` returns 1.
- ▶ `cospif( ±∞ )` returns NaN.

### Description

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.

## \_\_device\_\_ float cyl\_bessel\_i0f (float x)

Calculate the value of the regular modified cylindrical Bessel function of order 0 for the input argument.

### Returns

Returns the value of the regular modified cylindrical Bessel function of order 0.

### Description

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.

## \_\_device\_\_ float cyl\_bessel\_i1f (float x)

Calculate the value of the regular modified cylindrical Bessel function of order 1 for the input argument.

### Returns

Returns the value of the regular modified cylindrical Bessel function of order 1.

## Description

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.

## `__device__ float erfcf (float x)`

Calculate the complementary error function of the input argument.

## Returns

- ▶ `erfcf( -∞ )` returns 2.
- ▶ `erfcf( +∞ )` returns +0.

## Description

Calculate the complementary error function of the input argument  $x$ ,  $1 - \text{erf}(x)$ .



### Note:

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

## `__device__ float erfcinvf (float x)`

Calculate the inverse complementary error function of the input argument.

## Returns

- ▶ `erfcinvf( ±0 )` returns  $+\infty$ .
- ▶ `erfcinvf(2)` returns  $-\infty$ .
- ▶ `erfcinvf(x)` returns NaN for  $x$  outside  $[0, 2]$ .

## Description

Calculate the inverse complementary error function  $\text{erfc}^{-1}(x)$ , of the input argument  $x$  in the interval  $[0, 2]$ .

**Note:**

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

## `__device__ float erfcxf (float x)`

Calculate the scaled complementary error function of the input argument.

### Returns

- ▶ `erfcxf( -∞ )` returns `+∞`.
- ▶ `erfcxf( +∞ )` returns `+0`.

### Description

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.

## `__device__ float erff (float x)`

Calculate the error function of the input argument.

### Returns

- ▶ `erff( ±0 )` returns `±0`.
- ▶ `erff( ±∞ )` returns `±1`.

### Description

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.

## \_\_device\_\_ float erfinvf (float x)

Calculate the inverse error function of the input argument.

### Returns

- ▶ `erfinvf( ±0 )` returns  $\pm 0$ .
- ▶ `erfinvf(1)` returns  $+\infty$ .
- ▶ `erfinvf(-1)` returns  $-\infty$ .
- ▶ `erfinvf(x)` returns NaN for  $x$  outside  $[-1, +1]$ .

### Description

Calculate the inverse error function  $\text{erf}^{-1}(x)$ , of the input argument  $x$  in the interval  $[-1, 1]$ .



#### Note:

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

## \_\_device\_\_ float exp10f (float x)

Calculate the base 10 exponential of the input argument.

### Returns

- ▶ `exp10f( ±0 )` returns 1.
- ▶ `exp10f(  $-\infty$  )` returns +0.
- ▶ `exp10f(  $+\infty$  )` returns  $+\infty$ .

### Description

Calculate  $10^x$ , the base 10 exponential of the input argument  $x$ .



#### Note:

- ▶ For accuracy information see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.
- ▶ This function is affected by the `--use_fast_math` compiler flag. See the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section for a complete list of functions affected.

## `__device__ float exp2f (float x)`

Calculate the base 2 exponential of the input argument.

### Returns

- ▶ `exp2f( ±0 )` returns 1.
- ▶ `exp2f( -∞ )` returns +0.
- ▶ `exp2f( +∞ )` returns +∞.

### Description

Calculate  $2^x$ , the base 2 exponential of the input argument  $x$ .



#### Note:

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

## `__device__ float expf (float x)`

Calculate the base  $e$  exponential of the input argument.

### Returns

- ▶ `expf( ±0 )` returns 1.
- ▶ `expf( -∞ )` returns +0.
- ▶ `expf( +∞ )` returns +∞.

### Description

Calculate  $e^x$ , the base  $e$  exponential of the input argument  $x$ .



#### Note:

- ▶ For accuracy information see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.
- ▶ This function is affected by the `--use_fast_math` compiler flag. See the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section for a complete list of functions affected.

## `__device__ float expm1f (float x)`

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

### Returns

- ▶ `expm1f( ±0 )` returns  $±0$ .
- ▶ `expm1f(  $-\infty$  )` returns  $-1$ .
- ▶ `expm1f(  $+\infty$  )` returns  $+\infty$ .

### Description

Calculate  $e^x - 1$ , the base  $e$  exponential of the input argument  $x$ , minus 1.



#### Note:

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

## `__device__ float fabsf (float x)`

Calculate the absolute value of its argument.

### Returns

Returns the absolute value of its argument.

- ▶ `fabsf( ±∞ )` returns  $+\infty$ .
- ▶ `fabsf( ±0 )` returns  $+0$ .
- ▶ `fabsf(NaN)` returns an unspecified NaN.

### Description

Calculate the absolute value of the input argument  $x$ .



#### Note:

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

## `__device__ float fdimf (float x, float y)`

Compute the positive difference between  $x$  and  $y$ .

### Returns

Returns the positive difference between  $x$  and  $y$ .

- ▶ `fdimf(x, y)` returns  $x - y$  if  $x > y$ .
- ▶ `fdimf(x, y)` returns  $+0$  if  $x \leq y$ .

### Description

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.

## `__device__ float fdivdef (float x, float y)`

Divide two floating-point values.

### Returns

Returns  $x / y$ .

### Description

Compute  $x$  divided by  $y$ . If `--use_fast_math` is specified, use `__fdivdef()` for higher performance, otherwise use normal division.



#### Note:

- ▶ For accuracy information see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.
- ▶ This function is affected by the `--use_fast_math` compiler flag. See the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section for a complete list of functions affected.

## \_\_device\_\_ float floorf (float x)

Calculate the largest integer less than or equal to  $x$ .

### Returns

Returns  $\lfloor x \rfloor$  expressed as a floating-point number.

- ▶ `floorf(  $\pm\infty$  )` returns  $\pm\infty$ .
- ▶ `floorf(  $\pm 0$  )` returns  $\pm 0$ .

### Description

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



#### Note:

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

## \_\_device\_\_ float fmaf (float x, float y, float z)

Compute  $x \times y + z$  as a single operation.

### Returns

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

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

### Description

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.



## `__device__ float fmaxf (float x, float y)`

Determine the maximum numeric value of the arguments.

### 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.

### Description

Determines the maximum numeric value of the arguments  $x$  and  $y$ . Treats NaN arguments as missing data. If one argument is a NaN and the other is legitimate numeric value, the numeric value is chosen.

**Note:**

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

## `__device__ float fminf (float x, float y)`

Determine the minimum numeric value of the arguments.

### Returns

Returns the minimum numeric value of the arguments  $x$  and  $y$ .

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

### Description

Determines the minimum numeric value of the arguments  $x$  and  $y$ . Treats NaN arguments as missing data. If one argument is a NaN and the other is legitimate numeric value, the numeric value is chosen.

**Note:**

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

## \_\_device\_\_ float fmodf (float x, float y)

Calculate the floating-point remainder of  $x / y$ .

### Returns

- ▶ Returns the floating-point remainder of  $x / y$ .
- ▶  $fmodf(\pm 0, y)$  returns  $\pm 0$  if  $y$  is not zero.
- ▶  $fmodf(x, \pm \infty)$  returns  $x$  if  $x$  is finite.
- ▶  $fmodf(x, y)$  returns NaN if  $x$  is  $\pm \infty$  or  $y$  is zero.
- ▶ If either argument is NaN, NaN is returned.

### Description

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.

## \_\_device\_\_ float frexpf (float x, int \*nptr)

Extract mantissa and exponent of a floating-point value.

### Returns

Returns the fractional component  $m$ .

- ▶  $frexpf(\pm 0, nptr)$  returns  $\pm 0$  and stores zero in the location pointed to by  $nptr$ .
- ▶  $frexpf(\pm \infty, nptr)$  returns  $\pm \infty$  and stores an unspecified value in the location to which  $nptr$  points.
- ▶  $frexpf(\text{NaN}, y)$  returns a NaN and stores an unspecified value in the location to which  $nptr$  points.

### Description

Decomposes the floating-point value  $x$  into a component  $m$  for the normalized fraction element and another term  $n$  for the exponent. The absolute value of  $m$  will be greater than or equal to

0.5 and less than 1.0 or it will be equal to 0;  $x = m \cdot 2^n$ . The integer exponent  $n$  will be stored in the location to which `next` points.

**Note:**

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

## `__device__ float hypotf (float x, float y)`

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

### Returns

Returns the length of the hypotenuse  $\sqrt{x^2 + y^2}$ .

- ▶ `hypotf(x,y)`, `hypotf(y,x)`, and `hypotf(x, -y)` are equivalent.
- ▶ `hypotf(x, ±0)` is equivalent to `fabsf(x)`.
- ▶ `hypotf(±∞, y)` returns  $+\infty$ , even if `y` is a NaN.

### Description

Calculates the length of the hypotenuse of a right triangle whose two sides have lengths `x` and `y` without undue overflow or underflow.

**Note:**

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

## `__device__ int ilogbf (float x)`

Compute the unbiased integer exponent of the argument.

### Returns

- ▶ If successful, returns the unbiased exponent of the argument.
- ▶ `ilogbf(±0)` returns `INT_MIN`.
- ▶ `ilogbf(NaN)` returns `INT_MIN`.
- ▶ `ilogbf(±∞)` returns `INT_MAX`.
- ▶ Note: above behavior does not take into account `FP_ILOGB0` nor `FP_ILOGBNAN`.

## Description

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.

## `__device__ __RETURN_TYPE isfinite (float a)`

Determine whether argument is finite.

### Returns

- ▶ With Visual Studio 2013 host compiler: `__RETURN_TYPE` is 'bool'. Returns true if and only if  $a$  is a finite value.
- ▶ With other host compilers: `__RETURN_TYPE` is 'int'. Returns a nonzero value if and only if  $a$  is a finite value.

## Description

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

## `__device__ __RETURN_TYPE isinf (float a)`

Determine whether argument is infinite.

### Returns

- ▶ With Visual Studio 2013 host compiler: `__RETURN_TYPE` is 'bool'. Returns true if and only if  $a$  is an infinite value.
- ▶ With other host compilers: `__RETURN_TYPE` is 'int'. Returns a nonzero value if and only if  $a$  is an infinite value.

## Description

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

## \_\_device\_\_ \_\_RETURN\_TYPE isnan (float a)

Determine whether argument is a NaN.

### Returns

- ▶ With Visual Studio 2013 host compiler: \_\_RETURN\_TYPE is 'bool'. Returns true if and only if a is a NaN value.
- ▶ With other host compilers: \_\_RETURN\_TYPE is 'int'. Returns a nonzero value if and only if a is a NaN value.

### Description

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

## \_\_device\_\_ float j0f (float x)

Calculate the value of the Bessel function of the first kind of order 0 for the input argument.

### Returns

Returns the value of the Bessel function of the first kind of order 0.

- ▶  $j0f(\pm\infty)$  returns +0.
- ▶  $j0f(\text{NaN})$  returns NaN.

### Description

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.

## \_\_device\_\_ float j1f (float x)

Calculate the value of the Bessel function of the first kind of order 1 for the input argument.

### Returns

Returns the value of the Bessel function of the first kind of order 1.

- ▶  $j1f(\pm 0)$  returns  $\pm 0$ .
- ▶  $j1f(\pm\infty)$  returns  $\pm 0$ .

- ▶ `j1f(NaN)` returns NaN.

### Description

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.

## `__device__ float jnf (int n, float x)`

Calculate the value of the Bessel function of the first kind of order  $n$  for the input argument.

### Returns

Returns the value of the Bessel function of the first kind of order  $n$ .

- ▶ `jnf(n, NaN)` returns NaN.
- ▶ `jnf(n, x)` returns NaN for  $n < 0$ .
- ▶ `jnf(n, +∞)` returns +0.

### Description

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.

## `__device__ float ldexpf (float x, int exp)`

Calculate the value of  $x \cdot 2^{exp}$ .

### Returns

- ▶ `ldexpf(x, exp)` is equivalent to `scalbnf(x, exp)`.

### Description

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.

## \_\_device\_\_ float lgammaf (float x)

Calculate the natural logarithm of the absolute value of the gamma function of the input argument.

### Returns

- ▶ `lgammaf(1)` returns +0.
- ▶ `lgammaf(2)` returns +0.
- ▶ `lgammaf(x)` returns  $+\infty$  if  $x \leq 0$  and  $x$  is an integer.
- ▶ `lgammaf(-∞)` returns  $+\infty$ .
- ▶ `lgammaf(+∞)` returns  $+\infty$ .

### Description

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.

## \_\_device\_\_ long long int llrintf (float x)

Round input to nearest integer value.

### Returns

Returns rounded integer value.

### Description

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

## `__device__ long long int llroundf (float x)`

Round to nearest integer value.

### Returns

Returns rounded integer value.

### Description

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



#### Note:

This function may be slower than alternate rounding methods. See [llrintf\(\)](#).

## `__device__ float log10f (float x)`

Calculate the base 10 logarithm of the input argument.

### Returns

- ▶  $\log_{10}f(\pm 0)$  returns  $-\infty$ .
- ▶  $\log_{10}f(1)$  returns  $+0$ .
- ▶  $\log_{10}f(x)$  returns NaN for  $x < 0$ .
- ▶  $\log_{10}f(+\infty)$  returns  $+\infty$ .

### Description

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.
- ▶ This function is affected by the `--use_fast_math` compiler flag. See the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section for a complete list of functions affected.



## \_\_device\_\_ float log1pf(float x)

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

### Returns

- ▶  $\log1pf(\pm 0)$  returns  $\pm 0$ .
- ▶  $\log1pf(-1)$  returns  $-\infty$ .
- ▶  $\log1pf(x)$  returns NaN for  $x < -1$ .
- ▶  $\log1pf(+\infty)$  returns  $+\infty$ .

### Description

Calculate the value of  $\log_e(1+x)$  of the input argument  $x$ .



#### Note:

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

## \_\_device\_\_ float log2f(float x)

Calculate the base 2 logarithm of the input argument.

### Returns

- ▶  $\log2f(\pm 0)$  returns  $-\infty$ .
- ▶  $\log2f(1)$  returns +0.
- ▶  $\log2f(x)$  returns NaN for  $x < 0$ .
- ▶  $\log2f(+\infty)$  returns  $+\infty$ .

### Description

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.

- ▶ This function is affected by the `--use_fast_math` compiler flag. See the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section for a complete list of functions affected.

## `__device__ float logbf (float x)`

Calculate the floating-point representation of the exponent of the input argument.

### Returns

- ▶ `logbf( ±0 )` returns  $-\infty$ .
- ▶ `logbf( ±∞ )` returns  $+\infty$ .

### Description

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.

## `__device__ float logf (float x)`

Calculate the natural logarithm of the input argument.

### Returns

- ▶ `logf( ±0 )` returns  $-\infty$ .
- ▶ `logf(1)` returns  $+0$ .
- ▶ `logf(x)` returns NaN for  $x < 0$ .
- ▶ `logf( +∞ )` returns  $+\infty$ .

### Description

Calculate the natural logarithm of the input argument  $x$ .



#### Note:

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

- ▶ This function is affected by the `--use_fast_math` compiler flag. See the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section for a complete list of functions affected.

## `__device__ long int lrintf (float x)`

Round input to nearest integer value.

### Returns

Returns rounded integer value.

### Description

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

## `__device__ long int lroundf (float x)`

Round to nearest integer value.

### Returns

Returns rounded integer value.

### Description

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



#### Note:

This function may be slower than alternate rounding methods. See [lrintf\(\)](#).

## `__device__ float max (const float a, const float b)`

Calculate the maximum value of the input `float` arguments.

### Description

Calculate the maximum value of the arguments `a` and `b`. Behavior is equivalent to [fmaxf\(\)](#) function.

Note, this is different from `std::` specification

## `__device__ float min (const float a, const float b)`

Calculate the minimum value of the input `float` arguments.

### Description

Calculate the minimum value of the arguments `a` and `b`. Behavior is equivalent to [fminf\(\)](#) function.

Note, this is different from `std::` specification

## `__device__ float modff (float x, float *iptr)`

Break down the input argument into fractional and integral parts.

### Returns

- ▶ `modff(±x, iptr)` returns a result with the same sign as `x`.
- ▶ `modff(±∞, iptr)` returns `±0` and stores `±∞` in the object pointed to by `iptr`.
- ▶ `modff(NaN, iptr)` stores a NaN in the object pointed to by `iptr` and returns a NaN.

### Description

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.

## `__device__ float nanf (const char *tagp)`

Returns "Not a Number" value.

### Returns

- ▶ `nanf(tagp)` returns NaN.

### Description

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.

## `__device__ float nearbyintf (float x)`

Round the input argument to the nearest integer.

### Returns

- ▶ `nearbyintf( ±0 )` returns  $\pm 0$ .
- ▶ `nearbyintf( ±∞ )` returns  $\pm \infty$ .

### Description

Round argument  $x$  to an integer value in single precision floating-point format. Uses round to nearest rounding, with ties rounding to even.

**Note:**

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

## `__device__ float nextafterf (float x, float y)`

Return next representable single-precision floating-point value after argument  $x$  in the direction of  $y$ .

### Returns

- ▶ `nextafterf(x, y) = y` if  $x$  equals  $y$ .
- ▶ `nextafterf(x, y) = NaN` if either  $x$  or  $y$  are NaN.

### Description

Calculate the next representable single-precision floating-point value following  $x$  in the direction of  $y$ . For example, if  $y$  is greater than  $x$ , [nextafterf\(\)](#) returns the smallest representable number greater than  $x$

**Note:**

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

## `__device__ float norm3df (float a, float b, float c)`

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

### Returns

Returns the length of the 3D vector  $\sqrt{a^2 + b^2 + c^2}$ .

- In the presence of an exactly infinite coordinate  $+\infty$  is returned, even if there are NaNs.

### Description

Calculates the length of three dimensional vector in Euclidean space without undue overflow or underflow.



#### Note:

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

## `__device__ float norm4df (float a, float b, float c, float d)`

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

### Returns

Returns the length of the 4D vector  $\sqrt{a^2 + b^2 + c^2 + d^2}$ .

- In the presence of an exactly infinite coordinate  $+\infty$  is returned, even if there are NaNs.

### Description

Calculates the length of four dimensional vector in Euclidean space without undue overflow or underflow.



#### Note:

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

## \_\_device\_\_ float normcdf (float x)

Calculate the standard normal cumulative distribution function.

### Returns

- ▶ `normcdf(+∞)` returns 1.
- ▶ `normcdf(-∞)` returns +0

### Description

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



#### Note:

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

## \_\_device\_\_ float normcdfinv (float x)

Calculate the inverse of the standard normal cumulative distribution function.

### Returns

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

### Description

Calculate the inverse of the standard normal cumulative distribution function for input argument  $x$ ,  $\Phi^{-1}(x)$ . The function is defined for input values in the interval  $(0, 1)$ .



#### Note:

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

## \_\_device\_\_ float normf (int dim, const float \*p)

Calculate the square root of the sum of squares of any number of coordinates.

### Returns

Returns the length of the dim-D vector  $\sqrt{p_0^2 + p_1^2 + \dots + p_{\text{dim}-1}^2}$ .

- ▶ In the presence of an exactly infinite coordinate  $+\infty$  is returned, even if there are NaNs.

### Description

Calculates the length of a vector  $p$ , dimension of which is passed as an argument without undue overflow or underflow.



#### Note:

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

## \_\_device\_\_ float powf (float x, float y)

Calculate the value of first argument to the power of second argument.

### Returns

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



- ▶  $\text{powf}(-\infty, y)$  returns  $+0$  for  $y < 0$  and not an odd integer.
- ▶  $\text{powf}(-\infty, y)$  returns  $-\infty$  for  $y$  an odd integer greater than  $0$ .
- ▶  $\text{powf}(-\infty, y)$  returns  $+\infty$  for  $y > 0$  and not an odd integer.
- ▶  $\text{powf}(+\infty, y)$  returns  $+0$  for  $y < 0$ .
- ▶  $\text{powf}(+\infty, y)$  returns  $+\infty$  for  $y > 0$ .

## Description

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.
- ▶ This function is affected by the `--use_fast_math` compiler flag. See the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section for a complete list of functions affected.

## `__device__ float rcbtrf(float x)`

Calculate reciprocal cube root function.

## Returns

- ▶  $\text{rcbrt}(\pm 0)$  returns  $\pm \infty$ .
- ▶  $\text{rcbrt}(\pm \infty)$  returns  $\pm 0$ .

## Description

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.

## `__device__ float remainderf (float x, float y)`

Compute single-precision floating-point remainder.

### Returns

- ▶ `remainderf(x, ±0)` returns NaN.
- ▶ `remainderf(±∞, y)` returns NaN.
- ▶ `remainderf(x, ±∞)` returns `x` for finite `x`.

### Description

Compute single-precision floating-point remainder  $r$  of dividing  $x$  by  $y$  for nonzero  $y$ . Thus  $r = x - ny$ . The value  $n$  is the integer value nearest  $\frac{x}{y}$ . In the case when  $|n - \frac{x}{y}| = \frac{1}{2}$ , the even  $n$  value is chosen.



#### Note:

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

## `__device__ float remquof (float x, float y, int *quo)`

Compute single-precision floating-point remainder and part of quotient.

### Returns

Returns the remainder.

- ▶ `remquof(x, ±0, quo)` returns NaN and stores an unspecified value in the location to which `quo` points.
- ▶ `remquof(±∞, y, quo)` returns NaN and stores an unspecified value in the location to which `quo` points.
- ▶ `remquof(x, y, quo)` returns NaN and stores an unspecified value in the location to which `quo` points if either of `x` or `y` is NaN.
- ▶ `remquof(x, ±∞, quo)` returns `x` and stores zero in the location to which `quo` points for finite `x`.

### Description

Compute a single-precision floating-point remainder in the same way as the [remainderf\(\)](#) function. Argument `quo` returns part of quotient upon division of `x` by `y`. Value `quo` has the

same sign as  $\frac{x}{y}$  and may not be the exact quotient but agrees with the exact quotient in the low order 3 bits.

**Note:**

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

## `__device__ float rhypotf (float x, float y)`

Calculate one over the square root of the sum of squares of two arguments.

### Returns

Returns one over the length of the hypotenuse  $\frac{1}{\sqrt{x^2 + y^2}}$ .

- ▶ `rhypotf(x,y)`, `rhypotf(y,x)`, and `rhypotf(x, -y)` are equivalent.
- ▶ `rhypotf( ±∞ ,y)` returns +0, even if `y` is a NaN.

### Description

Calculates one over the length of the hypotenuse of a right triangle whose two sides have lengths `x` and `y` without undue overflow or underflow.

**Note:**

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

## `__device__ float rintf (float x)`

Round input to nearest integer value in floating-point.

### Returns

Returns rounded integer value.

- ▶ `rintf( ±0 )` returns `±0`.
- ▶ `rintf( ±∞ )` returns `±∞`.

### Description

Round `x` to the nearest integer value in floating-point format, with halfway cases rounded to the nearest even integer value.

## \_\_device\_\_ float rnorm3df (float a, float b, float c)

Calculate one over the square root of the sum of squares of three coordinates.

### Returns

Returns one over the length of the 3D vector  $\frac{1}{\sqrt{a^2+b^2+c^2}}$ .

- In the presence of an exactly infinite coordinate +0 is returned, even if there are NaNs.

### Description

Calculates one over the length of three dimension vector in Euclidean space without undue overflow or underflow.



#### Note:

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

## \_\_device\_\_ float rnorm4df (float a, float b, float c, float d)

Calculate one over the square root of the sum of squares of four coordinates.

### Returns

Returns one over the length of the 3D vector  $\frac{1}{\sqrt{a^2+b^2+c^2+d^2}}$ .

- In the presence of an exactly infinite coordinate +0 is returned, even if there are NaNs.

### Description

Calculates one over the length of four dimension vector in Euclidean space without undue overflow or underflow.



#### Note:

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

## `__device__ float rnormf (int dim, const float *p)`

Calculate the reciprocal of square root of the sum of squares of any number of coordinates.

### Returns

Returns one over the length of the vector  $\frac{1}{\sqrt{p_0^2 + p_1^2 + \dots + p_{\text{dim}-1}^2}}$ .

- ▶ In the presence of an exactly infinite coordinate  $+\infty$  is returned, even if there are NaNs.

### Description

Calculates one over the length of vector  $p$ , dimension of which is passed as an argument, in Euclidean space without undue overflow or underflow.



#### Note:

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

## `__device__ float roundf (float x)`

Round to nearest integer value in floating-point.

### Returns

Returns rounded integer value.

- ▶ `roundf( ±0 )` returns  $\pm 0$ .
- ▶ `roundf( ±∞ )` returns  $\pm \infty$ .

### Description

Round  $x$  to the nearest integer value in floating-point format, with halfway cases rounded away from zero.



#### Note:

This function may be slower than alternate rounding methods. See [rintf\(\)](#).

## \_\_device\_\_ float rsqrtf (float x)

Calculate the reciprocal of the square root of the input argument.

### Returns

Returns  $1/\sqrt{x}$ .

- ▶ `rsqrtf(+∞)` returns `+0`.
- ▶ `rsqrtf(±0)` returns `±∞`.
- ▶ `rsqrtf(x)` returns NaN if `x` is less than 0.

### Description

Calculate the reciprocal of the nonnegative square root of `x`,  $1/\sqrt{x}$ .



#### Note:

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

## \_\_device\_\_ float scalblnf (float x, long int n)

Scale floating-point input by integer power of two.

### Returns

Returns  $x * 2^n$ .

- ▶ `scalblnf(±0, n)` returns `±0`.
- ▶ `scalblnf(x, 0)` returns `x`.
- ▶ `scalblnf(±∞, n)` returns `±∞`.

### Description

Scale `x` by  $2^n$  by efficient manipulation of the floating-point exponent.

## \_\_device\_\_ float scalbnf (float x, int n)

Scale floating-point input by integer power of two.

### Returns

Returns  $x * 2^n$ .

- ▶ `scalbnf( ±0, n)` returns  $±0$ .
- ▶ `scalbnf(x, 0)` returns  $x$ .
- ▶ `scalbnf( ±∞, n)` returns  $±∞$ .

### Description

Scale  $x$  by  $2^n$  by efficient manipulation of the floating-point exponent.

## \_\_device\_\_ \_\_RETURN\_TYPE signbit (float a)

Return the sign bit of the input.

### Returns

Reports the sign bit of all values including infinities, zeros, and NaNs.

- ▶ With Visual Studio 2013 host compiler: `__RETURN_TYPE` is 'bool'. Returns true if and only if  $a$  is negative.
- ▶ With other host compilers: `__RETURN_TYPE` is 'int'. Returns a nonzero value if and only if  $a$  is negative.

### Description

Determine whether the floating-point value  $a$  is negative.

## \_\_device\_\_ void sincosf (float x, float \*sptr, float \*cptr)

Calculate the sine and cosine of the first input argument.

### Returns

- ▶ none

### Description

Calculate the sine and cosine of the first input argument  $x$  (measured in radians). The results for sine and cosine are written into the second argument, `sptr`, and, respectively, third argument, `cptr`.

### See also:

[sinf\(\)](#) and [cosf\(\)](#).



#### Note:

- ▶ For accuracy information see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Single-Precision Floating-Point Functions section.
- ▶ This function is affected by the `--use_fast_math` compiler flag. See the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section for a complete list of functions affected.

## `__device__ void sincospif (float x, float *sptr, float *cptr)`

Calculate the sine and cosine of the first input argument  $\times \pi$ .

### Returns

- ▶ none

### Description

Calculate the sine and cosine of the first input argument,  $x$  (measured in radians),  $\times \pi$ . The results for sine and cosine are written into the second argument, `sptr`, and, respectively, third argument, `cptr`.

### See also:

[sinpif\(\)](#) and [cospif\(\)](#).



#### Note:

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

## `__device__ float sinf (float x)`

Calculate the sine of the input argument.

### Returns

- ▶ `sinf(  $\pm 0$  )` returns  $\pm 0$ .
- ▶ `sinf(  $\pm \infty$  )` returns NaN.

### Description

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.
- ▶ This function is affected by the `--use_fast_math` compiler flag. See the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section for a complete list of functions affected.

## `__device__ float sinhf (float x)`

Calculate the hyperbolic sine of the input argument.

### Returns

- ▶ `sinhf( ±0 )` returns `±0`.
- ▶ `sinhf( ±∞ )` returns `±∞`.

### Description

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.

## `__device__ float sinpif (float x)`

Calculate the sine of the input argument  $\times \pi$ .

### Returns

- ▶ `sinpif( ±0 )` returns `±0`.
- ▶ `sinpif( ±∞ )` returns NaN.

### Description

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.

## `__device__ float sqrtf (float x)`

Calculate the square root of the input argument.

### Returns

Returns  $\sqrt{x}$ .

- ▶ `sqrtf( ±0 )` returns  $\pm 0$ .
- ▶ `sqrtf( +∞ )` returns  $+\infty$ .
- ▶ `sqrtf(x)` returns NaN if  $x$  is less than 0.

### Description

Calculate the nonnegative square root of  $x$ ,  $\sqrt{x}$ .



#### Note:

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

## `__device__ float tanf (float x)`

Calculate the tangent of the input argument.

### Returns

- ▶ `tanf( ±0 )` returns  $\pm 0$ .
- ▶ `tanf( ±∞ )` returns NaN.

### Description

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.
- ▶ This function is affected by the `--use_fast_math` compiler flag. See the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section for a complete list of functions affected.

## \_\_device\_\_ float tanhf (float x)

Calculate the hyperbolic tangent of the input argument.

### Returns

- ▶  $\text{tanhf}(\pm 0)$  returns  $\pm 0$ .
- ▶  $\text{tanhf}(\pm \infty)$  returns  $\pm 1$ .

### Description

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.

## \_\_device\_\_ float tgammaf (float x)

Calculate the gamma function of the input argument.

### Returns

- ▶  $\text{tgammaf}(\pm 0)$  returns  $\pm \infty$ .
- ▶  $\text{tgammaf}(2)$  returns  $+1$ .
- ▶  $\text{tgammaf}(x)$  returns NaN if  $x < 0$  and  $x$  is an integer.
- ▶  $\text{tgammaf}(-\infty)$  returns NaN.
- ▶  $\text{tgammaf}(+\infty)$  returns  $+\infty$ .

### Description

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.

## `__device__ float truncf (float x)`

Truncate input argument to the integral part.

### Returns

Returns truncated integer value.

- ▶ `truncf( ±0 )` returns  $±0$ .
- ▶ `truncf( ±∞ )` returns  $±∞$ .

### Description

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

## `__device__ float y0f (float x)`

Calculate the value of the Bessel function of the second kind of order 0 for the input argument.

### Returns

Returns the value of the Bessel function of the second kind of order 0.

- ▶ `y0f( ±0 )` returns  $−∞$ .
- ▶ `y0f(x)` returns NaN for  $x < 0$ .
- ▶ `y0f( +∞ )` returns  $+0$ .
- ▶ `y0f(NaN)` returns NaN.

### Description

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.

## `__device__ float y1f (float x)`

Calculate the value of the Bessel function of the second kind of order 1 for the input argument.

### Returns

Returns the value of the Bessel function of the second kind of order 1.

- ▶  $y1f(\pm 0)$  returns  $-\infty$ .
- ▶  $y1f(x)$  returns NaN for  $x < 0$ .
- ▶  $y1f(+\infty)$  returns +0.
- ▶  $y1f(\text{NaN})$  returns NaN.

### Description

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.

## \_\_device\_\_ float ynf (int n, float x)

Calculate the value of the Bessel function of the second kind of order  $n$  for the input argument.

### Returns

Returns the value of the Bessel function of the second kind of order  $n$ .

- ▶  $ynf(n, x)$  returns NaN for  $n < 0$ .
- ▶  $ynf(n, \pm 0)$  returns  $-\infty$ .
- ▶  $ynf(n, x)$  returns NaN for  $x < 0$ .
- ▶  $ynf(n, +\infty)$  returns +0.
- ▶  $ynf(n, \text{NaN})$  returns NaN.

### Description

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.

## 1.6. Double Precision Mathematical Functions

This section describes double precision mathematical functions. To use these functions you do not need to include any additional header files in your program.

### `__device__ double acos (double x)`

Calculate the arc cosine of the input argument.

#### Returns

Result will be in radians, in the interval  $[0, \pi]$  for  $x$  inside  $[-1, +1]$ .

- ▶ `acos(1)` returns `+0`.
- ▶ `acos(x)` returns NaN for  $x$  outside  $[-1, +1]$ .

#### Description

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.

### `__device__ double acosh (double x)`

Calculate the nonnegative inverse hyperbolic cosine of the input argument.

#### Returns

Result will be in the interval  $[0, +\infty]$ .

- ▶ `acosh(1)` returns `0`.
- ▶ `acosh(x)` returns NaN for  $x$  in the interval  $[-\infty, 1)$ .
- ▶ `acosh(+\infty)` returns `+\infty`.

#### Description

Calculate the nonnegative inverse hyperbolic cosine of the input argument  $x$ .

**Note:**

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

## \_\_device\_\_ double asin (double x)

Calculate the arc sine of the input argument.

### Returns

Result will be in radians, in the interval  $[-\pi/2, +\pi/2]$  for  $x$  inside  $[-1, +1]$ .

- ▶ `asin( ±0 )` returns  $\pm 0$ .
- ▶ `asin(x)` returns NaN for  $x$  outside  $[-1, +1]$ .

### Description

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.

## \_\_device\_\_ double asinh (double x)

Calculate the inverse hyperbolic sine of the input argument.

### Returns

- ▶ `asinh( ±0 )` returns  $\pm 0$ .
- ▶ `asinh( ±∞ )` returns  $\pm \infty$ .

### Description

Calculate the inverse hyperbolic sine of the input argument  $x$ .

**Note:**

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

## `__device__ double atan (double x)`

Calculate the arc tangent of the input argument.

### Returns

Result will be in radians, in the interval  $[-\pi/2, +\pi/2]$ .

- ▶ `atan( ±0 )` returns  $\pm 0$ .
- ▶ `atan( ±∞ )` returns  $\pm \pi/2$ .

### Description

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.

## `__device__ double atan2 (double y, double x)`

Calculate the arc tangent of the ratio of first and second input arguments.

### Returns

Result will be in radians, in the interval  $[-\pi, +\pi]$ .

- ▶ `atan2( ±0 , -0)` returns  $\pm \pi$ .
- ▶ `atan2( ±0 , +0)` returns  $\pm 0$ .
- ▶ `atan2( ±0 , x)` returns  $\pm \pi$  for  $x < 0$ .
- ▶ `atan2( ±0 , x)` returns  $\pm 0$  for  $x > 0$ .
- ▶ `atan2(y , ±0)` returns  $-\pi/2$  for  $y < 0$ .
- ▶ `atan2(y , ±0)` returns  $\pi/2$  for  $y > 0$ .
- ▶ `atan2( ±y , -∞)` returns  $\pm \pi$  for finite  $y > 0$ .
- ▶ `atan2( ±y , +∞)` returns  $\pm 0$  for finite  $y > 0$ .
- ▶ `atan2( ±∞ , x)` returns  $\pm \pi/2$  for finite  $x$ .
- ▶ `atan2( ±∞ , -∞)` returns  $\pm 3\pi/4$ .
- ▶ `atan2( ±∞ , +∞)` returns  $\pm \pi/4$ .



## Description

Calculate the principal value of the arc tangent of the ratio of first and second input arguments  $y / x$ . The quadrant of the result is determined by the signs of inputs  $y$  and  $x$ .



### Note:

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

## `__device__ double atanh (double x)`

Calculate the inverse hyperbolic tangent of the input argument.

## Returns

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

## Description

Calculate the inverse hyperbolic tangent of the input argument  $x$ .



### Note:

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

## `__device__ double cbrt (double x)`

Calculate the cube root of the input argument.

## Returns

Returns  $x^{1/3}$ .

- ▶ `cbrt( ±0 )` returns  $\pm 0$ .
- ▶ `cbrt( ±∞ )` returns  $\pm \infty$ .

## Description

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.

## \_\_device\_\_ double ceil (double x)

Calculate ceiling of the input argument.

### Returns

Returns  $\lceil x \rceil$  expressed as a floating-point number.

- ▶  $\text{ceil}(\pm 0)$  returns  $\pm 0$ .
- ▶  $\text{ceil}(\pm \infty)$  returns  $\pm \infty$ .

### Description

Compute the smallest integer value not less than  $x$ .

## \_\_device\_\_ double copysign (double x, double y)

Create value with given magnitude, copying sign of second value.

### Returns

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

### Description

Create a floating-point value with the magnitude  $x$  and the sign of  $y$ .

## \_\_device\_\_ double cos (double x)

Calculate the cosine of the input argument.

### Returns

- ▶  $\cos(\pm 0)$  returns 1.
- ▶  $\cos(\pm \infty)$  returns NaN.

### Description

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.

## `__device__ double cosh (double x)`

Calculate the hyperbolic cosine of the input argument.

### Returns

- ▶ `cosh( ±0 )` returns 1.
- ▶ `cosh( ±∞ )` returns  $+\infty$ .

### Description

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.

## `__device__ double cospi (double x)`

Calculate the cosine of the input argument  $\times \pi$ .

### Returns

- ▶ `cospi( ±0 )` returns 1.
- ▶ `cospi( ±∞ )` returns NaN.

### Description

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.

## \_\_device\_\_ double cyl\_bessel\_i0 (double x)

Calculate the value of the regular modified cylindrical Bessel function of order 0 for the input argument.

### Returns

Returns the value of the regular modified cylindrical Bessel function of order 0.

### Description

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.

## \_\_device\_\_ double cyl\_bessel\_i1 (double x)

Calculate the value of the regular modified cylindrical Bessel function of order 1 for the input argument.

### Returns

Returns the value of the regular modified cylindrical Bessel function of order 1.

### Description

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.

## \_\_device\_\_ double erf (double x)

Calculate the error function of the input argument.

### Returns

- ▶  $\text{erf}(\pm 0)$  returns  $\pm 0$ .
- ▶  $\text{erf}(\pm \infty)$  returns  $\pm 1$ .

## Description

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.

## `__device__ double erfc (double x)`

Calculate the complementary error function of the input argument.

### Returns

- ▶ `erfc( -∞ )` returns 2.
- ▶ `erfc( +∞ )` returns +0.

## Description

Calculate the complementary error function of the input argument  $x$ ,  $1 - \text{erf}(x)$ .



### Note:

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

## `__device__ double erfcinv (double x)`

Calculate the inverse complementary error function of the input argument.

### Returns

- ▶ `erfcinv( ±0 )` returns  $+\infty$ .
- ▶ `erfcinv(2)` returns  $-\infty$ .
- ▶ `erfcinv(x)` returns NaN for  $x$  outside  $[0, 2]$ .

## Description

Calculate the inverse complementary error function  $\text{erfc}^{-1}(x)$ , of the input argument  $x$  in the interval  $[0, 2]$ .

**Note:**

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

## \_\_device\_\_ double erfcx (double x)

Calculate the scaled complementary error function of the input argument.

### Returns

- ▶  $\text{erfcx}(-\infty)$  returns  $+\infty$ .
- ▶  $\text{erfcx}(+\infty)$  returns  $+0$ .

### Description

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.

## \_\_device\_\_ double erfinv (double x)

Calculate the inverse error function of the input argument.

### Returns

- ▶  $\text{erfinv}(\pm 0)$  returns  $\pm 0$ .
- ▶  $\text{erfinv}(1)$  returns  $+\infty$ .
- ▶  $\text{erfinv}(-1)$  returns  $-\infty$ .
- ▶  $\text{erfinv}(x)$  returns NaN for  $x$  outside  $[-1, +1]$ .

### Description

Calculate the inverse error function  $\text{erf}^{-1}(x)$ , of the input argument  $x$  in the interval  $[-1, 1]$ .

**Note:**

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

## \_\_device\_\_ double exp (double x)

Calculate the base  $e$  exponential of the input argument.

### Returns

- ▶  $\text{exp}(\pm 0)$  returns 1.
- ▶  $\text{exp}(-\infty)$  returns +0.
- ▶  $\text{exp}(+\infty)$  returns  $+\infty$ .

### Description

Calculate  $e^x$ , the base  $e$  exponential of the input argument  $x$ .

**Note:**

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

## \_\_device\_\_ double exp10 (double x)

Calculate the base 10 exponential of the input argument.

### Returns

- ▶  $\text{exp10}(\pm 0)$  returns 1.
- ▶  $\text{exp10}(-\infty)$  returns +0.
- ▶  $\text{exp10}(+\infty)$  returns  $+\infty$ .

### Description

Calculate  $10^x$ , the base 10 exponential of the input argument  $x$ .

**Note:**

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

## \_\_device\_\_ double exp2 (double x)

Calculate the base 2 exponential of the input argument.

### Returns

- ▶  $\text{exp2}(\pm 0)$  returns 1.
- ▶  $\text{exp2}(-\infty)$  returns +0.
- ▶  $\text{exp2}(+\infty)$  returns  $+\infty$ .

### Description

Calculate  $2^x$ , the base 2 exponential of the input argument  $x$ .

**Note:**

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

## \_\_device\_\_ double expm1 (double x)

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

### Returns

- ▶  $\text{expm1}(\pm 0)$  returns  $\pm 0$ .
- ▶  $\text{expm1}(-\infty)$  returns -1.
- ▶  $\text{expm1}(+\infty)$  returns  $+\infty$ .

### Description

Calculate  $e^x - 1$ , the base  $e$  exponential of the input argument  $x$ , minus 1.

**Note:**

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



## `__device__ double fabs (double x)`

Calculate the absolute value of the input argument.

### Returns

Returns the absolute value of the input argument.

- ▶ `fabs( ±∞ )` returns  $+∞$ .
- ▶ `fabs( ±0 )` returns  $+0$ .

### Description

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.

## `__device__ double fdim (double x, double y)`

Compute the positive difference between `x` and `y`.

### Returns

Returns the positive difference between `x` and `y`.

- ▶ `fdim(x, y)` returns  $x - y$  if  $x > y$ .
- ▶ `fdim(x, y)` returns  $+0$  if  $x \leq y$ .

### Description

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.

## \_\_device\_\_ double floor (double x)

Calculate the largest integer less than or equal to  $x$ .

### Returns

Returns  $\lfloor x \rfloor$  expressed as a floating-point number.

- ▶  $\text{floor}(\pm\infty)$  returns  $\pm\infty$ .
- ▶  $\text{floor}(\pm 0)$  returns  $\pm 0$ .

### Description

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.

## \_\_device\_\_ double fma (double x, double y, double z)

Compute  $x \times y + z$  as a single operation.

### Returns

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

- ▶  $\text{fma}(\pm\infty, \pm 0, z)$  returns NaN.
- ▶  $\text{fma}(\pm 0, \pm\infty, z)$  returns NaN.
- ▶  $\text{fma}(x, y, -\infty)$  returns NaN if  $x \times y$  is an exact  $+\infty$ .
- ▶  $\text{fma}(x, y, +\infty)$  returns NaN if  $x \times y$  is an exact  $-\infty$ .

### Description

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.

## `__device__ double fmax (double, double)`

Determine the maximum numeric value of the arguments.

### 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.

### Description

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.

## `__device__ double fmin (double x, double y)`

Determine the minimum numeric value of the arguments.

### Returns

Returns the minimum numeric value of the arguments  $x$  and  $y$ .

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

### Description

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.

## `__device__ double fmod (double x, double y)`

Calculate the double-precision floating-point remainder of  $x / y$ .

### Returns

- ▶ Returns the floating-point remainder of  $x / y$ .
- ▶ `fmod( ±0, y)` returns  $±0$  if  $y$  is not zero.
- ▶ `fmod(x, ±∞)` returns  $x$  if  $x$  is finite.
- ▶ `fmod(x, y)` returns NaN if  $x$  is  $±∞$  or  $y$  is zero.
- ▶ If either argument is NaN, NaN is returned.

### Description

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.

## `__device__ double frexp (double x, int *nptr)`

Extract mantissa and exponent of a floating-point value.

### Returns

Returns the fractional component  $m$ .

- ▶ `frexp( ±0, nptr)` returns  $±0$  and stores zero in the location pointed to by `nptr`.
- ▶ `frexp( ±∞, nptr)` returns  $±∞$  and stores an unspecified value in the location to which `nptr` points.
- ▶ `frexp(NaN, y)` returns a NaN and stores an unspecified value in the location to which `nptr` points.

### Description

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.

## `__device__ double hypot (double x, double y)`

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

### Returns

Returns the length of the hypotenuse  $\sqrt{x^2 + y^2}$ .

- ▶ `hypot(x,y)`, `hypot(y,x)`, and `hypot(x, -y)` are equivalent.
- ▶ `hypot(x, ±0)` is equivalent to `fabs(x)`.
- ▶ `hypot( ±∞ ,y)` returns  $+\infty$ , even if `y` is a NaN.

### Description

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.

## `__device__ int ilogb (double x)`

Compute the unbiased integer exponent of the argument.

### Returns

- ▶ If successful, returns the unbiased exponent of the argument.
- ▶ `ilogb( ±0 )` returns `INT_MIN`.
- ▶ `ilogb(NaN)` returns `INT_MIN`.
- ▶ `ilogb( ±∞ )` returns `INT_MAX`.
- ▶ Note: above behavior does not take into account `FP_ILOGB0` nor `FP_ILOGBNAN`.

## Description

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.

## `__device__ __RETURN_TYPE isfinite (double a)`

Determine whether argument is finite.

### Returns

- ▶ With Visual Studio 2013 host compiler: `__RETURN_TYPE` is 'bool'. Returns true if and only if  $a$  is a finite value.
- ▶ With other host compilers: `__RETURN_TYPE` is 'int'. Returns a nonzero value if and only if  $a$  is a finite value.

## Description

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

## `__device__ __RETURN_TYPE isinf (double a)`

Determine whether argument is infinite.

### Returns

- ▶ With Visual Studio 2013 host compiler: Returns true if and only if  $a$  is an infinite value.
- ▶ With other host compilers: Returns a nonzero value if and only if  $a$  is an infinite value.

## Description

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

## `__device__ __RETURN_TYPE isnan (double a)`

Determine whether argument is a NaN.

### Returns

- ▶ With Visual Studio 2013 host compiler: `__RETURN_TYPE` is 'bool'. Returns true if and only if `a` is a NaN value.
- ▶ With other host compilers: `__RETURN_TYPE` is 'int'. Returns a nonzero value if and only if `a` is a NaN value.

### Description

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

## `__device__ double j0 (double x)`

Calculate the value of the Bessel function of the first kind of order 0 for the input argument.

### Returns

Returns the value of the Bessel function of the first kind of order 0.

- ▶ `j0( ±∞ )` returns `+0`.
- ▶ `j0(NaN)` returns `NaN`.

### Description

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.

## `__device__ double j1 (double x)`

Calculate the value of the Bessel function of the first kind of order 1 for the input argument.

### Returns

Returns the value of the Bessel function of the first kind of order 1.

- ▶ `j1( ±0 )` returns `±0`.
- ▶ `j1( ±∞ )` returns `±0`.

- ▶ `j1(NaN)` returns NaN.

### Description

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.

## `__device__ double jn (int n, double x)`

Calculate the value of the Bessel function of the first kind of order  $n$  for the input argument.

### Returns

Returns the value of the Bessel function of the first kind of order  $n$ .

- ▶ `jn(n, NaN)` returns NaN.
- ▶ `jn(n, x)` returns NaN for  $n < 0$ .
- ▶ `jn(n, +∞)` returns +0.

### Description

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.

## `__device__ double ldexp (double x, int exp)`

Calculate the value of  $x \cdot 2^{exp}$ .

### Returns

- ▶ `ldexp(x, exp)` is equivalent to `scalbn(x, exp)`.

### Description

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.

## \_\_device\_\_ double lgamma (double x)

Calculate the natural logarithm of the absolute value of the gamma function of the input argument.

### Returns

- ▶ `lgamma(1)` returns +0.
- ▶ `lgamma(2)` returns +0.
- ▶ `lgamma(x)` returns  $+\infty$  if  $x \leq 0$  and  $x$  is an integer.
- ▶ `lgamma(-∞)` returns  $+\infty$ .
- ▶ `lgamma(+∞)` returns  $+\infty$ .

### Description

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.

## \_\_device\_\_ long long int llrint (double x)

Round input to nearest integer value.

### Returns

Returns rounded integer value.

### Description

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

## `__device__ long long int llround (double x)`

Round to nearest integer value.

### Returns

Returns rounded integer value.

### Description

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



#### Note:

This function may be slower than alternate rounding methods. See [llrint\(\)](#).

## `__device__ double log (double x)`

Calculate the base  $e$  logarithm of the input argument.

### Returns

- ▶  $\log(\pm 0)$  returns  $-\infty$ .
- ▶  $\log(1)$  returns  $+0$ .
- ▶  $\log(x)$  returns NaN for  $x < 0$ .
- ▶  $\log(+\infty)$  returns  $+\infty$ .

### Description

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.

## `__device__ double log10 (double x)`

Calculate the base 10 logarithm of the input argument.

### Returns

- ▶  $\log_{10}(\pm 0)$  returns  $-\infty$ .

- ▶  $\log_{10}(1)$  returns +0.
- ▶  $\log_{10}(x)$  returns NaN for  $x < 0$ .
- ▶  $\log_{10}(+\infty)$  returns  $+\infty$ .

## Description

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.

## `__device__ double log1p (double x)`

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

## Returns

- ▶  $\log_{1p}(\pm 0)$  returns  $\pm 0$ .
- ▶  $\log_{1p}(-1)$  returns  $-\infty$ .
- ▶  $\log_{1p}(x)$  returns NaN for  $x < -1$ .
- ▶  $\log_{1p}(+\infty)$  returns  $+\infty$ .

## Description

Calculate the value of  $\log_e(1+x)$  of the input argument  $x$ .



### Note:

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

## `__device__ double log2 (double x)`

Calculate the base 2 logarithm of the input argument.

## Returns

- ▶  $\log_2(\pm 0)$  returns  $-\infty$ .
- ▶  $\log_2(1)$  returns +0.

- ▶  $\log_2(x)$  returns NaN for  $x < 0$ .
- ▶  $\log_2(+\infty)$  returns  $+\infty$ .

### Description

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.

## `__device__ double logb (double x)`

Calculate the floating-point representation of the exponent of the input argument.

### Returns

- ▶  $\log_b(\pm 0)$  returns  $-\infty$ .
- ▶  $\log_b(\pm \infty)$  returns  $+\infty$ .

### Description

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.

## `__device__ long int lrint (double x)`

Round input to nearest integer value.

### Returns

Returns rounded integer value.

### Description

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

## `__device__ long int lround (double x)`

Round to nearest integer value.

### Returns

Returns rounded integer value.

### Description

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



#### Note:

This function may be slower than alternate rounding methods. See [lrint\(\)](#).

## `__device__ double max (const double a, const float b)`

Calculate the maximum value of the input `double` and `float` arguments.

### Description

Convert `float` argument `b` to `double`, followed by [fmax\(\)](#).

Note, this is different from `std::` specification

## `__device__ double max (const float a, const double b)`

Calculate the maximum value of the input `float` and `double` arguments.

### Description

Convert `float` argument `a` to `double`, followed by [fmax\(\)](#).

Note, this is different from `std::` specification

## `__device__ double max (const double a, const double b)`

Calculate the maximum value of the input `float` arguments.

### Description

Calculate the maximum value of the arguments `a` and `b`. Behavior is equivalent to [fmax\(\)](#) function.

Note, this is different from `std::` specification

## `__device__ double min (const double a, const float b)`

Calculate the minimum value of the input `double` and `float` arguments.

### Description

Convert `float` argument `b` to `double`, followed by [fmin\(\)](#).

Note, this is different from `std::` specification

## `__device__ double min (const float a, const double b)`

Calculate the minimum value of the input `float` and `double` arguments.

### Description

Convert `float` argument `a` to `double`, followed by [fmin\(\)](#).

Note, this is different from `std::` specification

## `__device__ double min (const double a, const double b)`

Calculate the minimum value of the input `float` arguments.

### Description

Calculate the minimum value of the arguments `a` and `b`. Behavior is equivalent to [fmin\(\)](#) function.

Note, this is different from `std::` specification

## `__device__ double modf (double x, double *iptr)`

Break down the input argument into fractional and integral parts.

### Returns

- ▶ `modf( ±x, iptr)` returns a result with the same sign as `x`.
- ▶ `modf( ±∞, iptr)` returns `±0` and stores `±∞` in the object pointed to by `iptr`.
- ▶ `modf(NaN, iptr)` stores a NaN in the object pointed to by `iptr` and returns a NaN.

### Description

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.

## `__device__ double nan (const char *tagp)`

Returns "Not a Number" value.

### Returns

- ▶ `nan(tagp)` returns NaN.

### Description

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.

## `__device__ double nearbyint (double x)`

Round the input argument to the nearest integer.

### Returns

- ▶ `nearbyint( ±0 )` returns  $\pm 0$ .
- ▶ `nearbyint( ±∞ )` returns  $\pm \infty$ .

### Description

Round argument `x` to an integer value in double precision floating-point format. Uses round to nearest rounding, with ties rounding to even.

**Note:**

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

## `__device__ double nextafter (double x, double y)`

Return next representable double-precision floating-point value after argument  $x$  in the direction of  $y$ .

### Returns

- ▶  $\text{nextafter}(x, y) = y$  if  $x$  equals  $y$ .
- ▶  $\text{nextafter}(x, y) = \text{NaN}$  if either  $x$  or  $y$  are NaN.

### Description

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.

## `__device__ double norm (int dim, const double *p)`

Calculate the square root of the sum of squares of any number of coordinates.

### Returns

Returns the length of the dim-D vector  $\sqrt{p_0^2 + p_1^2 + \dots + p_{\text{dim}-1}^2}$ .

- ▶ In the presence of an exactly infinite coordinate  $+\infty$  is returned, even if there are NaNs.

### Description

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.



## `__device__ double norm3d (double a, double b, double c)`

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

### Returns

Returns the length of 3D vector  $\sqrt{a^2 + b^2 + c^2}$ .

- In the presence of an exactly infinite coordinate  $+\infty$  is returned, even if there are NaNs.

### Description

Calculate the length of three dimensional vector in Euclidean space without undue overflow or underflow.



#### Note:

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

## `__device__ double norm4d (double a, double b, double c, double d)`

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

### Returns

Returns the length of 4D vector  $\sqrt{a^2 + b^2 + c^2 + d^2}$ .

- In the presence of an exactly infinite coordinate  $+\infty$  is returned, even if there are NaNs.

### Description

Calculate the length of four dimensional vector in Euclidean space without undue overflow or underflow.



#### Note:

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

## \_\_device\_\_ double normcdf (double x)

Calculate the standard normal cumulative distribution function.

### Returns

- ▶ normcdf(  $+\infty$  ) returns 1.
- ▶ normcdf(  $-\infty$  ) returns +0.

### Description

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



#### Note:

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

## \_\_device\_\_ double normcdfinv (double x)

Calculate the inverse of the standard normal cumulative distribution function.

### Returns

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

### Description

Calculate the inverse of the standard normal cumulative distribution function for input argument  $x$ ,  $\Phi^{-1}(x)$ . The function is defined for input values in the interval  $(0, 1)$ .



#### Note:

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

## `__device__ double pow (double x, double y)`

Calculate the value of first argument to the power of second argument.

### Returns

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

### Description

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.

## `__device__ double rcbrt (double x)`

Calculate reciprocal cube root function.

### Returns

- ▶ `rcbrt( ±0 )` returns  $±∞$ .
- ▶ `rcbrt( ±∞ )` returns  $±0$ .

### Description

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.

## `__device__ double remainder (double x, double y)`

Compute double-precision floating-point remainder.

### Returns

- ▶ `remainder(x, ±0)` returns NaN.
- ▶ `remainder( ±∞, y)` returns NaN.
- ▶ `remainder(x, ±∞)` returns  $x$  for finite  $x$ .

### Description

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.

## `__device__ double remquo (double x, double y, int *quo)`

Compute double-precision floating-point remainder and part of quotient.

### Returns

Returns the remainder.

- ▶ `remquo(x, ±0, quo)` returns NaN and stores an unspecified value in the location to which `quo` points.
- ▶ `remquo(±∞, y, quo)` returns NaN and stores an unspecified value in the location to which `quo` points.
- ▶ `remquo(x, y, quo)` returns NaN and stores an unspecified value in the location to which `quo` points if either of `x` or `y` is NaN.
- ▶ `remquo(x, ±∞, quo)` returns `x` and stores zero in the location to which `quo` points for finite `x`.

### Description

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.

## `__device__ double rhypot (double x, double y)`

Calculate one over the square root of the sum of squares of two arguments.

### Returns

Returns one over the length of the hypotenuse  $\frac{1}{\sqrt{x^2 + y^2}}$ .

- ▶ `rhypot(x,y)`, `rhypot(y,x)`, and `rhypot(x, -y)` are equivalent.
- ▶ `rhypot(±∞,y)` returns +0, even if `y` is a NaN.

## Description

Calculate one over the length of the hypotenuse of a right triangle whose two sides have lengths  $x$  and  $y$  without undue overflow or underflow.



### Note:

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

## \_\_device\_\_ double rint (double x)

Round to nearest integer value in floating-point.

## Returns

Returns rounded integer value.

- ▶ `rint( ±0 )` returns  $±0$ .
- ▶ `rint( ±∞ )` returns  $±∞$ .

## Description

Round  $x$  to the nearest integer value in floating-point format, with halfway cases rounded to the nearest even integer value.

## \_\_device\_\_ double rnorm (int dim, const double \*p)

Calculate the reciprocal of square root of the sum of squares of any number of coordinates.

## Returns

Returns one over the length of the vector  $\frac{1}{\sqrt{p_0^2 + p_1^2 + \dots + p_{\text{dim}-1}^2}}$ .

- ▶ In the presence of an exactly infinite coordinate  $+0$  is returned, even if there are NaNs.

## Description

Calculates one over the length of vector  $p$ , dimension of which is passed as an argument, in Euclidean space without undue overflow or underflow.



### Note:

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

## `__device__ double rnorm3d (double a, double b, double c)`

Calculate one over the square root of the sum of squares of three coordinates.

### Returns

Returns one over the length of the 3D vector  $\frac{1}{\sqrt{a^2+b^2+c^2}}$ .

- In the presence of an exactly infinite coordinate `+0` is returned, even if there are NaNs.

### Description

Calculate one over the length of three dimensional vector in Euclidean space without undue overflow or underflow.



#### Note:

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

## `__device__ double rnorm4d (double a, double b, double c, double d)`

Calculate one over the square root of the sum of squares of four coordinates.

### Returns

Returns one over the length of the 4D vector  $\frac{1}{\sqrt{a^2+b^2+c^2+d^2}}$ .

- In the presence of an exactly infinite coordinate `+0` is returned, even if there are NaNs.

### Description

Calculate one over the length of four dimensional vector in Euclidean space without undue overflow or underflow.



#### Note:

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

## \_\_device\_\_ double round (double x)

Round to nearest integer value in floating-point.

### Returns

Returns rounded integer value.

- ▶ `round( ±0 )` returns `±0`.
- ▶ `round( ±∞ )` returns `±∞`.

### Description

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\(\)](#).

## \_\_device\_\_ double rsqrt (double x)

Calculate the reciprocal of the square root of the input argument.

### Returns

Returns  $1/\sqrt{x}$ .

- ▶ `rsqrt( +∞ )` returns `+0`.
- ▶ `rsqrt( ±0 )` returns `±∞`.
- ▶ `rsqrt(x)` returns NaN if `x` is less than 0.

### Description

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.



## `__device__ double scalbln (double x, long int n)`

Scale floating-point input by integer power of two.

### Returns

Returns  $x * 2^n$ .

- ▶ `scalbln( ±0 , n)` returns  $±0$ .
- ▶ `scalbln(x, 0)` returns  $x$ .
- ▶ `scalbln( ±∞ , n)` returns  $±∞$ .

### Description

Scale  $x$  by  $2^n$  by efficient manipulation of the floating-point exponent.

## `__device__ double scalbn (double x, int n)`

Scale floating-point input by integer power of two.

### Returns

Returns  $x * 2^n$ .

- ▶ `scalbn( ±0 , n)` returns  $±0$ .
- ▶ `scalbn(x, 0)` returns  $x$ .
- ▶ `scalbn( ±∞ , n)` returns  $±∞$ .

### Description

Scale  $x$  by  $2^n$  by efficient manipulation of the floating-point exponent.

## `__device__ __RETURN_TYPE signbit (double a)`

Return the sign bit of the input.

### Returns

Reports the sign bit of all values including infinities, zeros, and NaNs.

- ▶ With Visual Studio 2013 host compiler: `__RETURN_TYPE` is 'bool'. Returns true if and only if  $a$  is negative.
- ▶ With other host compilers: `__RETURN_TYPE` is 'int'. Returns a nonzero value if and only if  $a$  is negative.

## Description

Determine whether the floating-point value  $a$  is negative.

## `__device__ double sin (double x)`

Calculate the sine of the input argument.

## Returns

- ▶ `sin( ±0 )` returns  $\pm 0$ .
- ▶ `sin( ±∞ )` returns NaN.

## Description

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.

## `__device__ void sincos (double x, double *sptr, double *cptr)`

Calculate the sine and cosine of the first input argument.

## Returns

- ▶ none

## Description

Calculate the sine and cosine of the first input argument  $x$  (measured in radians). The results for sine and cosine are written into the second argument, `sptr`, and, respectively, third argument, `cptr`.

### See also:

[sin\(\)](#) and [cos\(\)](#).



### Note:

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

## `__device__ void sincospi (double x, double *sptr, double *cptr)`

Calculate the sine and cosine of the first input argument  $\times \pi$ .

### Returns

- ▶ none

### Description

Calculate the sine and cosine of the first input argument,  $x$  (measured in radians),  $\times \pi$ . The results for sine and cosine are written into the second argument, `sptr`, and, respectively, third argument, `cptr`.

### See also:

[sinpi\(\)](#) and [cospi\(\)](#).



#### Note:

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

## `__device__ double sinh (double x)`

Calculate the hyperbolic sine of the input argument.

### Returns

- ▶ `sinh( ±0 )` returns  $\pm 0$ .
- ▶ `sinh( ±∞ )` returns  $\pm \infty$ .

### Description

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.

## \_\_device\_\_ double sinpi (double x)

Calculate the sine of the input argument  $\times \pi$ .

### Returns

- ▶  $\text{sinpi}(\pm 0)$  returns  $\pm 0$ .
- ▶  $\text{sinpi}(\pm \infty)$  returns NaN.

### Description

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.

## \_\_device\_\_ double sqrt (double x)

Calculate the square root of the input argument.

### Returns

Returns  $\sqrt{x}$ .

- ▶  $\text{sqrt}(\pm 0)$  returns  $\pm 0$ .
- ▶  $\text{sqrt}(+\infty)$  returns  $+\infty$ .
- ▶  $\text{sqrt}(x)$  returns NaN if  $x$  is less than 0.

### Description

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.

## \_\_device\_\_ double tan (double x)

Calculate the tangent of the input argument.

### Returns

- ▶  $\tan(\pm 0)$  returns  $\pm 0$ .
- ▶  $\tan(\pm \infty)$  returns NaN.

### Description

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.

## \_\_device\_\_ double tanh (double x)

Calculate the hyperbolic tangent of the input argument.

### Returns

- ▶  $\tanh(\pm 0)$  returns  $\pm 0$ .
- ▶  $\tanh(\pm \infty)$  returns  $\pm 1$ .

### Description

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.

## \_\_device\_\_ double tgamma (double x)

Calculate the gamma function of the input argument.

### Returns

- ▶  $tgamma(\pm 0)$  returns  $\pm \infty$ .
- ▶  $tgamma(2)$  returns +1.

- ▶ `tgamma(x)` returns NaN if  $x < 0$  and  $x$  is an integer.
- ▶ `tgamma(-∞)` returns NaN.
- ▶ `tgamma(+∞)` returns  $+\infty$ .

### Description

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.

## \_\_device\_\_ double trunc (double x)

Truncate input argument to the integral part.

### Returns

Returns truncated integer value.

- ▶ `trunc(±0)` returns  $\pm 0$ .
- ▶ `trunc(±∞)` returns  $\pm \infty$ .

### Description

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

## \_\_device\_\_ double y0 (double x)

Calculate the value of the Bessel function of the second kind of order 0 for the input argument.

### Returns

Returns the value of the Bessel function of the second kind of order 0.

- ▶ `y0(±0)` returns  $-\infty$ .
- ▶ `y0(x)` returns NaN for  $x < 0$ .
- ▶ `y0(+∞)` returns  $+0$ .
- ▶ `y0(NaN)` returns NaN.

## Description

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.

## `__device__ double y1 (double x)`

Calculate the value of the Bessel function of the second kind of order 1 for the input argument.

## Returns

Returns the value of the Bessel function of the second kind of order 1.

- ▶  $y1(\pm 0)$  returns  $-\infty$ .
- ▶  $y1(x)$  returns NaN for  $x < 0$ .
- ▶  $y1(+\infty)$  returns  $+0$ .
- ▶  $y1(\text{NaN})$  returns NaN.

## Description

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.

## `__device__ double yn (int n, double x)`

Calculate the value of the Bessel function of the second kind of order  $n$  for the input argument.

## Returns


Returns the value of the Bessel function of the second kind of order  $n$ .

- ▶  $yn(n, x)$  returns NaN for  $n < 0$ .
- ▶  $yn(n, \pm 0)$  returns  $-\infty$ .

- ▶  $y_n(n, x)$  returns NaN for  $x < 0$ .
- ▶  $y_n(n, +\infty)$  returns +0.
- ▶  $y_n(n, \text{NaN})$  returns NaN.

### Description

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.

## 1.7. Integer Mathematical Functions

This section describes integer mathematical functions. To use these functions you do not need to include any additional header files in your program.

### `__device__ int abs (int a)`

Calculate the absolute value of the input `int` argument.

#### Description

Calculate the absolute value of the input argument `a`.

### `__device__ long int labs (long int a)`

Calculate the absolute value of the input `long int` argument.

#### Description

Calculate the absolute value of the input argument `a`.

### `__device__ long long int llabs (long long int a)`

Calculate the absolute value of the input `long long int` argument.

#### Description

Calculate the absolute value of the input argument `a`.



**\_\_device\_\_ long long int llmax (const long long int a, const long long int b)**

Calculate the maximum value of the input long long int arguments.

#### Description

Calculate the maximum value of the arguments a and b.

**\_\_device\_\_ long long int llmin (const long long int a, const long long int b)**

Calculate the minimum value of the input long long int arguments.

#### Description

Calculate the minimum value of the arguments a and b.

**\_\_device\_\_ unsigned long long int max (const unsigned long long int a, const long long int b)**

Calculate the maximum value of the input unsigned long long int and long long int arguments.

#### Description

Calculate the maximum value of the arguments a and b, perform integer promotion first.

**\_\_device\_\_ unsigned long long int max (const long long int a, const unsigned long long int b)**

Calculate the maximum value of the input long long int and unsigned long long int arguments.

#### Description

Calculate the maximum value of the arguments a and b, perform integer promotion first.

**\_\_device\_\_ unsigned long long int max (const unsigned long long int a, const unsigned long long int b)**

Calculate the maximum value of the input unsigned long long int arguments.

#### Description

Calculate the maximum value of the arguments a and b.

**\_\_device\_\_ long long int max (const long long int a, const long long int b)**

Calculate the maximum value of the input long long int arguments.

#### Description

Calculate the maximum value of the arguments a and b.

**\_\_device\_\_ unsigned long int max (const unsigned long int a, const long int b)**

Calculate the maximum value of the input unsigned long int and long int arguments.

#### Description

Calculate the maximum value of the arguments a and b, perform integer promotion first.

**\_\_device\_\_ unsigned long int max (const long int a, const unsigned long int b)**

Calculate the maximum value of the input long int and unsigned long int arguments.

#### Description

Calculate the maximum value of the arguments a and b, perform integer promotion first.

**\_\_device\_\_ unsigned long int max (const unsigned long int a, const unsigned long int b)**

Calculate the maximum value of the input unsigned long int arguments.

#### Description

Calculate the maximum value of the arguments a and b.

## `__device__ long int max (const long int a, const long int b)`

Calculate the maximum value of the input `long int` arguments.

### Description

Calculate the maximum value of the arguments `a` and `b`.

## `__device__ unsigned int max (const unsigned int a, const int b)`

Calculate the maximum value of the input `unsigned int` and `int` arguments.

### Description

Calculate the maximum value of the arguments `a` and `b`, perform integer promotion first.

## `__device__ unsigned int max (const int a, const unsigned int b)`

Calculate the maximum value of the input `int` and `unsigned int` arguments.

### Description

Calculate the maximum value of the arguments `a` and `b`, perform integer promotion first.

## `__device__ unsigned int max (const unsigned int a, const unsigned int b)`

Calculate the maximum value of the input `unsigned int` arguments.

### Description

Calculate the maximum value of the arguments `a` and `b`.

## `__device__ int max (const int a, const int b)`

Calculate the maximum value of the input `int` arguments.

### Description

Calculate the maximum value of the arguments `a` and `b`.

## `__device__ unsigned long long int min (const unsigned long long int a, const long long int b)`

Calculate the minimum value of the input `unsigned long long int` and `long long int` arguments.

### Description

Calculate the minimum value of the arguments `a` and `b`, perform integer promotion first.

## `__device__ unsigned long long int min (const long long int a, const unsigned long long int b)`

Calculate the minimum value of the input `long long int` and `unsigned long long int` arguments.

### Description

Calculate the minimum value of the arguments `a` and `b`, perform integer promotion first.

## `__device__ unsigned long long int min (const unsigned long long int a, const unsigned long long int b)`

Calculate the minimum value of the input `unsigned long long int` arguments.

### Description

Calculate the minimum value of the arguments `a` and `b`.

## `__device__ long long int min (const long long int a, const long long int b)`

Calculate the minimum value of the input `long long int` arguments.

### Description

Calculate the minimum value of the arguments `a` and `b`.

## `__device__ unsigned long int min (const unsigned long int a, const long int b)`

Calculate the minimum value of the input `unsigned long int` and `long int` arguments.

### Description

Calculate the minimum value of the arguments `a` and `b`, perform integer promotion first.

## `__device__ unsigned long int min (const long int a, const unsigned long int b)`

Calculate the minimum value of the input `long int` and `unsigned long int` arguments.

### Description

Calculate the minimum value of the arguments `a` and `b`, perform integer promotion first.

## `__device__ unsigned long int min (const unsigned long int a, const unsigned long int b)`

Calculate the minimum value of the input `unsigned long int` arguments.

### Description

Calculate the minimum value of the arguments `a` and `b`.

## `__device__ long int min (const long int a, const long int b)`

Calculate the minimum value of the input `long int` arguments.

### Description

Calculate the minimum value of the arguments `a` and `b`.

## `__device__ unsigned int min (const unsigned int a, const int b)`

Calculate the minimum value of the input `unsigned int` and `int` arguments.

### Description

Calculate the minimum value of the arguments `a` and `b`, perform integer promotion first.

## `__device__ unsigned int min (const int a, const unsigned int b)`

Calculate the minimum value of the input `int` and `unsigned int` arguments.

### Description

Calculate the minimum value of the arguments `a` and `b`, perform integer promotion first.

## `__device__ unsigned int min (const unsigned int a, const unsigned int b)`

Calculate the minimum value of the input `unsigned int` arguments.

### Description

Calculate the minimum value of the arguments `a` and `b`.

## `__device__ int min (const int a, const int b)`

Calculate the minimum value of the input `int` arguments.

### Description

Calculate the minimum value of the arguments `a` and `b`.

## `__device__ unsigned long long int ullmax (const unsigned long long int a, const unsigned long long int b)`

Calculate the maximum value of the input `unsigned long long int` arguments.

### Description

Calculate the maximum value of the arguments `a` and `b`.

## `__device__ unsigned long long int ullmin (const unsigned long long int a, const unsigned long long int b)`

Calculate the minimum value of the input `unsigned long long int` arguments.

### Description

Calculate the minimum value of the arguments `a` and `b`.

**\_\_device\_\_ unsigned int umax (const unsigned int a, const unsigned int b)**

Calculate the maximum value of the input unsigned int arguments.

#### Description

Calculate the maximum value of the arguments a and b.

**\_\_device\_\_ unsigned int umin (const unsigned int a, const unsigned int b)**

Calculate the minimum value of the input unsigned int arguments.

#### Description

Calculate the minimum value of the arguments a and b.

## 1.8. Single Precision Intrinsic Functions

This section describes single precision intrinsic functions that are only supported in device code. To use these functions you do not need to include any additional header files in your program.

**\_\_device\_\_ float \_\_cosf (float x)**

Calculate the fast approximate cosine of the input argument.

#### Returns

Returns the approximate cosine of x.

#### Description

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.

## `__device__ float __exp10f (float x)`

Calculate the fast approximate base 10 exponential of the input argument.

### Returns

Returns an approximation to  $10^x$ .

### Description

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.

## `__device__ float __expf (float x)`

Calculate the fast approximate base  $e$  exponential of the input argument.

### Returns

Returns an approximation to  $e^x$ .

### Description

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.

## `__device__ float __fadd_rd (float x, float y)`

Add two floating-point values in round-down mode.

### Returns

Returns  $x + y$ .

### Description

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

## `__device__ float __fadd_rn (float x, float y)`

Add two floating-point values in round-to-nearest-even mode.

### Returns

Returns  $x + y$ .

### Description

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

## `__device__ float __fadd_ru (float x, float y)`

Add two floating-point values in round-up mode.

### Returns

Returns  $x + y$ .

### Description

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

## `__device__ float __fadd_rz (float x, float y)`

Add two floating-point values in round-towards-zero mode.

### Returns

Returns  $x + y$ .

### Description

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

## `__device__ float __fdiv_rd (float x, float y)`

Divide two floating-point values in round-down mode.

### Returns

Returns  $x / y$ .

### Description

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.

## `__device__ float __fdiv_rn (float x, float y)`

Divide two floating-point values in round-to-nearest-even mode.

### Returns

Returns  $x / y$ .

### Description

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.

## `__device__ float __fdiv_ru (float x, float y)`

Divide two floating-point values in round-up mode.

### Returns

Returns  $x / y$ .

### Description

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.

## `__device__ float __fdiv_rz (float x, float y)`

Divide two floating-point values in round-towards-zero mode.

### Returns

Returns  $x / y$ .

### Description

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.

## `__device__ float __fdividef (float x, float y)`

Calculate the fast approximate division of the input arguments.

### Returns

Returns  $x / y$ .

- ▶ `__fdividef(∞, y)` returns NaN for  $2^{126} < |y| < 2^{128}$ .
- ▶ `__fdividef(x, y)` returns 0 for  $2^{126} < |y| < 2^{128}$  and finite  $x$ .

## Description

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.

## `__device__ float __fmaf_ieee_rd (float x, float y, float z)`

Compute fused multiply-add operation in round-down mode, ignore `-ftz=true` compiler flag.

## Description

Behavior is the same as `__fmaf_rd(x, y, z)`, the difference is in handling denormalized inputs and outputs: `-ftz` compiler flag has no effect.

## `__device__ float __fmaf_ieee_rn (float x, float y, float z)`

Compute fused multiply-add operation in round-to-nearest-even mode, ignore `-ftz=true` compiler flag.

## Description

Behavior is the same as `__fmaf_rn(x, y, z)`, the difference is in handling denormalized inputs and outputs: `-ftz` compiler flag has no effect.

## `__device__ float __fmaf_ieee_ru (float x, float y, float z)`

Compute fused multiply-add operation in round-up mode, ignore `-ftz=true` compiler flag.

## Description

Behavior is the same as `__fmaf_ru(x, y, z)`, the difference is in handling denormalized inputs and outputs: `-ftz` compiler flag has no effect.

## `__device__ float __fmaf_ieee_rz (float x, float y, float z)`

Compute fused multiply-add operation in round-towards-zero mode, ignore `-ftz=true` compiler flag.

### Description

Behavior is the same as `__fmaf_rz(x, y, z)`, the difference is in handling denormalized inputs and outputs: `-ftz` compiler flag has no effect.

## `__device__ float __fmaf_rd (float x, float y, float z)`

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

### Returns

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

- ▶ `fmaf( ±∞, ±0, z)` returns NaN.
- ▶ `fmaf( ±0, ±∞, z)` returns NaN.
- ▶ `fmaf(x, y, -∞)` returns NaN if  $x \times y$  is an exact  $+\infty$ .
- ▶ `fmaf(x, y, +∞)` returns NaN if  $x \times y$  is an exact  $-\infty$ .

### Description

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.

## `__device__ float __fmaf_rn (float x, float y, float z)`

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

### Returns

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

- ▶ `fmaf( ±∞, ±0, z)` returns NaN.
- ▶ `fmaf( ±0, ±∞, z)` returns NaN.

- ▶ `fmaf(x, y, -∞)` returns NaN if  $x \times y$  is an exact  $+\infty$ .
- ▶ `fmaf(x, y, +∞)` returns NaN if  $x \times y$  is an exact  $-\infty$ .

## Description

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.

## `__device__ float __fmaf_ru(float x, float y, float z)`

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

## Returns

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

- ▶ `fmaf(±∞, ±0, z)` returns NaN.
- ▶ `fmaf(±0, ±∞, z)` returns NaN.
- ▶ `fmaf(x, y, -∞)` returns NaN if  $x \times y$  is an exact  $+\infty$ .
- ▶ `fmaf(x, y, +∞)` returns NaN if  $x \times y$  is an exact  $-\infty$ .

## Description

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.

## `__device__ float __fmaf_rz(float x, float y, float z)`

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

## Returns


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

- ▶ `fmaf(±∞, ±0, z)` returns NaN.

- ▶ `fmaf( ±0 , ±∞ , z)` returns NaN.
- ▶ `fmaf(x, y, -∞)` returns NaN if  $x \times y$  is an exact  $+\infty$ .
- ▶ `fmaf(x, y, +∞)` returns NaN if  $x \times y$  is an exact  $-\infty$ .

## Description

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.

## `__device__ float __fmul_rd(float x, float y)`


Multiply two floating-point values in round-down mode.

### Returns

Returns  $x * y$ .

### Description

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

## `__device__ float __fmul_rn(float x, float y)`

Multiply two floating-point values in round-to-nearest-even mode.

### Returns

Returns  $x * y$ .

### Description

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

## `__device__ float __fmul_ru(float x, float y)`

Multiply two floating-point values in round-up mode.

### Returns

Returns  $x * y$ .

### Description

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

## `__device__ float __fmul_rz(float x, float y)`

Multiply two floating-point values in round-towards-zero mode.

### Returns

Returns  $x * y$ .

### Description

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



## `__device__ float __frcp_rd (float x)`

Compute  $\frac{1}{x}$  in round-down mode.

### Returns

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

### Description

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.

## `__device__ float __frcp_rn (float x)`

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

### Returns

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

### Description

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.

## `__device__ float __frcp_ru (float x)`

Compute  $\frac{1}{x}$  in round-up mode.

### Returns

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

## Description

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.

## `__device__ float __frcp_rz (float x)`

Compute  $\frac{1}{x}$  in round-towards-zero mode.

## Returns

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

## Description

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.

## `__device__ float __frsqrt_rn (float x)`

Compute  $1/\sqrt{x}$  in round-to-nearest-even mode.

## Returns

Returns  $1/\sqrt{x}$ .

## Description

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.

## `__device__ float __fsqrt_rd (float x)`

Compute  $\sqrt{x}$  in round-down mode.

### Returns

Returns  $\sqrt{x}$ .

### Description

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.

## `__device__ float __fsqrt_rn (float x)`

Compute  $\sqrt{x}$  in round-to-nearest-even mode.

### Returns

Returns  $\sqrt{x}$ .

### Description

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.

## `__device__ float __fsqrt_ru (float x)`

Compute  $\sqrt{x}$  in round-up mode.

### Returns

Returns  $\sqrt{x}$ .

## Description

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.

## `__device__ float __fsqrt_rz (float x)`

Compute  $\sqrt{x}$  in round-towards-zero mode.

## Returns

Returns  $\sqrt{x}$ .

## Description

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.

## `__device__ float __fsub_rd (float x, float y)`

Subtract two floating-point values in round-down mode.

## Returns

Returns  $x - y$ .

## Description

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

## `__device__ float __fsub_rn (float x, float y)`

Subtract two floating-point values in round-to-nearest-even mode.

### Returns

Returns  $x - y$ .

### Description

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

## `__device__ float __fsub_ru (float x, float y)`

Subtract two floating-point values in round-up mode.

### Returns

Returns  $x - y$ .

### Description

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

## `__device__ float __fsub_rz (float x, float y)`

Subtract two floating-point values in round-towards-zero mode.

### Returns

Returns  $x - y$ .

## Description

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

## `__device__ float __log10f (float x)`

Calculate the fast approximate base 10 logarithm of the input argument.

## Returns

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

## Description

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.

## `__device__ float __log2f (float x)`

Calculate the fast approximate base 2 logarithm of the input argument.

## Returns

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

## Description

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.

## `__device__ float __logf (float x)`

Calculate the fast approximate base  $e$  logarithm of the input argument.

### Returns

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

### Description

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.

## `__device__ float __powf (float x, float y)`

Calculate the fast approximate of  $x^y$ .

### Returns

Returns an approximation to  $x^y$ .

### Description

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.

## `__device__ float __saturatef (float x)`

Clamp the input argument to  $[+0.0, 1.0]$ .

### Returns

- ▶ `__saturatef(x)` returns 0 if  $x < 0$ .
- ▶ `__saturatef(x)` returns 1 if  $x > 1$ .
- ▶ `__saturatef(x)` returns  $x$  if  $0 \leq x \leq 1$ .

- ▶ `__saturatef(NaN)` returns 0.

### Description

Clamp the input argument `x` to be within the interval `[+0.0, 1.0]`.

**`__device__ void __sincosf (float x, float *sptr, float *cptr)`**

Calculate the fast approximate of sine and cosine of the first input argument.

### Returns

- ▶ none

### Description

Calculate the fast approximate of sine and cosine of the first input argument `x` (measured in radians). The results for sine and cosine are written into the second argument, `sptr`, and, respectively, third argument, `cptr`.



#### Note:

- ▶ For accuracy information see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Intrinsic Functions section.
- ▶ Denorm input/output is flushed to sign preserving 0.0.

**`__device__ float __sinf (float x)`**

Calculate the fast approximate sine of the input argument.

### Returns

Returns the approximate sine of `x`.

### Description

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.
- ▶ Output in the denormal range is flushed to sign preserving 0.0.



## `__device__ float __tanf (float x)`

Calculate the fast approximate tangent of the input argument.

### Returns

Returns the approximate tangent of  $x$ .

### Description

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.
- ▶ The result is computed as the fast divide of `__sinf()` by `__cosf()`. Denormal output is flushed to sign-preserving 0.0.

## 1.9. Double Precision Ininsics

This section describes double precision intrinsic functions that are only supported in device code. To use these functions you do not need to include any additional header files in your program.

## `__device__ double __dadd_rd (double x, double y)`

Add two floating-point values in round-down mode.

### Returns

Returns  $x + y$ .

### Description

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

## `__device__ double __dadd_rn (double x, double y)`

Add two floating-point values in round-to-nearest-even mode.

### Returns

Returns  $x + y$ .

### Description

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

## `__device__ double __dadd_ru (double x, double y)`

Add two floating-point values in round-up mode.

### Returns

Returns  $x + y$ .

### Description

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

## `__device__ double __dadd_rz (double x, double y)`

Add two floating-point values in round-towards-zero mode.

### Returns

Returns  $x + y$ .

## Description

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

## `__device__ double __ddiv_rd (double x, double y)`

Divide two floating-point values in round-down mode.

## Returns

Returns  $x / y$ .

## Description

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.
- ▶ Requires compute capability  $\geq 2.0$ .

## `__device__ double __ddiv_rn (double x, double y)`

Divide two floating-point values in round-to-nearest-even mode.

## Returns

Returns  $x / y$ .

## Description

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.

- ▶ Requires compute capability  $\geq 2.0$ .

## `__device__ double __ddiv_ru (double x, double y)`

Divide two floating-point values in round-up mode.

### Returns

Returns  $x / y$ .

### Description

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.
- ▶ Requires compute capability  $\geq 2.0$ .

## `__device__ double __ddiv_rz (double x, double y)`

Divide two floating-point values in round-towards-zero mode.

### Returns

Returns  $x / y$ .

### Description

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.
- ▶ Requires compute capability  $\geq 2.0$ .

## `__device__ double __dmul_rd (double x, double y)`

Multiply two floating-point values in round-down mode.

### Returns

Returns  $x * y$ .

## Description

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

## `__device__ double __dmul_rn (double x, double y)`

Multiply two floating-point values in round-to-nearest-even mode.

## Returns

Returns  $x * y$ .

## Description

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

## `__device__ double __dmul_ru (double x, double y)`

Multiply two floating-point values in round-up mode.

## Returns

Returns  $x * y$ .

## Description

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.

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

## `__device__ double __dmul_rz (double x, double y)`

Multiply two floating-point values in round-towards-zero mode.

### Returns

Returns  $x * y$ .

### Description

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

## `__device__ double __drcp_rd (double x)`

Compute  $\frac{1}{x}$  in round-down mode.

### Returns

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

### Description

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.
- ▶ Requires compute capability  $\geq 2.0$ .

## \_\_device\_\_ double \_\_drcp\_rn (double x)

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

### Returns

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

### Description

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.
- ▶ Requires compute capability  $\geq 2.0$ .

## \_\_device\_\_ double \_\_drcp\_ru (double x)

Compute  $\frac{1}{x}$  in round-up mode.

### Returns

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

### Description

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.
- ▶ Requires compute capability  $\geq 2.0$ .

## \_\_device\_\_ double \_\_drcp\_rz (double x)

Compute  $\frac{1}{x}$  in round-towards-zero mode.

### Returns

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

## Description

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.
- ▶ Requires compute capability  $\geq 2.0$ .

## \_\_device\_\_ double \_\_dsqrt\_rd (double x)

Compute  $\sqrt{x}$  in round-down mode.

## Returns

Returns  $\sqrt{x}$ .

## Description

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.
- ▶ Requires compute capability  $\geq 2.0$ .

## \_\_device\_\_ double \_\_dsqrt\_rn (double x)

Compute  $\sqrt{x}$  in round-to-nearest-even mode.

## Returns

Returns  $\sqrt{x}$ .

## Description

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.
- ▶ Requires compute capability  $\geq 2.0$ .

## `__device__ double __dsqrt_ru (double x)`

Compute  $\sqrt{x}$  in round-up mode.

### Returns

Returns  $\sqrt{x}$ .

### Description

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.
- ▶ Requires compute capability  $\geq 2.0$ .

## `__device__ double __dsqrt_rz (double x)`

Compute  $\sqrt{x}$  in round-towards-zero mode.

### Returns

Returns  $\sqrt{x}$ .

### Description

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.
- ▶ Requires compute capability  $\geq 2.0$ .

## `__device__ double __dsub_rd (double x, double y)`

Subtract two floating-point values in round-down mode.

### Returns

Returns  $x - y$ .

### Description

Subtracts two floating-point values  $x$  and  $y$  in round-down (to negative infinity) mode.



#### Note:

- ▶ For accuracy information see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.
- ▶ This operation will never be merged into a single multiply-add instruction.

## `__device__ double __dsub_rn (double x, double y)`

Subtract two floating-point values in round-to-nearest-even mode.

### Returns

Returns  $x - y$ .

### Description

Subtracts two floating-point values  $x$  and  $y$  in round-to-nearest-even mode.



#### Note:

- ▶ For accuracy information see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.
- ▶ This operation will never be merged into a single multiply-add instruction.

## `__device__ double __dsub_ru (double x, double y)`

Subtract two floating-point values in round-up mode.

### Returns

Returns  $x - y$ .

## Description

Subtracts two floating-point values  $x$  and  $y$  in round-up (to positive infinity) mode.



### Note:

- ▶ For accuracy information see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.
- ▶ This operation will never be merged into a single multiply-add instruction.

## `__device__ double __dsub_rz (double x, double y)`

Subtract two floating-point values in round-towards-zero mode.

## Returns

Returns  $x - y$ .

## Description

Subtracts two floating-point values  $x$  and  $y$  in round-towards-zero mode.



### Note:

- ▶ For accuracy information see the CUDA C++ Programming Guide, Mathematical Functions Appendix, Double-Precision Floating-Point Functions section.
- ▶ This operation will never be merged into a single multiply-add instruction.

## `__device__ double __fma_rd (double x, double y, double z)`

Compute  $x \times y + z$  as a single operation in round-down mode.

## Returns

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

- ▶ `fmaf( ±∞ , ±0 , z)` returns NaN.
- ▶ `fmaf( ±0 , ±∞ , z)` returns NaN.
- ▶ `fmaf(x, y, -∞)` returns NaN if  $x \times y$  is an exact  $+\infty$
- ▶ `fmaf(x, y, +∞)` returns NaN if  $x \times y$  is an exact  $-\infty$

## Description

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.

## `__device__ double __fma_rn (double x, double y, double z)`

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

## Returns

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

- ▶ `fmaf( ±∞ , ±0 , z )` returns NaN.
- ▶ `fmaf( ±0 , ±∞ , z )` returns NaN.
- ▶ `fmaf(x, y, -∞ )` returns NaN if  $x \times y$  is an exact  $+\infty$
- ▶ `fmaf(x, y, +∞ )` returns NaN if  $x \times y$  is an exact  $-\infty$

## Description

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.

## `__device__ double __fma_ru (double x, double y, double z)`

Compute  $x \times y + z$  as a single operation in round-up mode.

## Returns

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

- ▶ `fmaf( ±∞, ±0, z)` returns NaN.
- ▶ `fmaf( ±0, ±∞, z)` returns NaN.
- ▶ `fmaf(x, y, -∞)` returns NaN if  $x \times y$  is an exact  $+\infty$
- ▶ `fmaf(x, y, +∞)` returns NaN if  $x \times y$  is an exact  $-\infty$

### Description

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.

## `__device__ double __fma_rz (double x, double y, double z)`

Compute  $x \times y + z$  as a single operation in round-towards-zero mode.

### Returns

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

- ▶ `fmaf( ±∞, ±0, z)` returns NaN.
- ▶ `fmaf( ±0, ±∞, z)` returns NaN.
- ▶ `fmaf(x, y, -∞)` returns NaN if  $x \times y$  is an exact  $+\infty$
- ▶ `fmaf(x, y, +∞)` returns NaN if  $x \times y$  is an exact  $-\infty$

### Description

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.

## 1.10. Integer Intrinsics

This section describes integer intrinsic functions that are only supported in device code. To use these functions you do not need to include any additional header files in your program.

### `__device__ unsigned int __brev (unsigned int x)`

Reverse the bit order of a 32-bit unsigned integer.

#### Returns

Returns the bit-reversed value of `x`. i.e. bit `N` of the return value corresponds to bit `31-N` of `x`.

#### Description

Reverses the bit order of the 32-bit unsigned integer `x`.

### `__device__ unsigned long long int __brevll (unsigned long long int x)`

Reverse the bit order of a 64-bit unsigned integer.

#### Returns

Returns the bit-reversed value of `x`. i.e. bit `N` of the return value corresponds to bit `63-N` of `x`.

#### Description

Reverses the bit order of the 64-bit unsigned integer `x`.

### `__device__ unsigned int __byte_perm (unsigned int x, unsigned int y, unsigned int s)`

Return selected bytes from two 32-bit unsigned integers.

#### Returns

Returns a 32-bit integer consisting of four bytes from eight input bytes provided in the two input integers `x` and `y`, as specified by a selector, `s`.

#### Description

Create 8-byte source

► `uint64_t tmp64 = ((uint64_t)y << 32) | x;`

Extract selector bits

- ▶ `selector0 = (s >> 0) & 0x7;`
- ▶ `selector1 = (s >> 4) & 0x7;`
- ▶ `selector2 = (s >> 8) & 0x7;`
- ▶ `selector3 = (s >> 12) & 0x7;`

Return 4 selected bytes from 8-byte source:

- ▶ `res[07:00] = tmp64[selector0];`
- ▶ `res[15:08] = tmp64[selector1];`
- ▶ `res[23:16] = tmp64[selector2];`
- ▶ `res[31:24] = tmp64[selector3];`

## `__device__ int __clz (int x)`

Return the number of consecutive high-order zero bits in a 32-bit integer.

### Returns

Returns a value between 0 and 32 inclusive representing the number of zero bits.

### Description

Count the number of consecutive leading zero bits, starting at the most significant bit (bit 31) of `x`.

## `__device__ int __clzll (long long int x)`

Count the number of consecutive high-order zero bits in a 64-bit integer.

### Returns

Returns a value between 0 and 64 inclusive representing the number of zero bits.

### Description

Count the number of consecutive leading zero bits, starting at the most significant bit (bit 63) of `x`.

## `__device__ unsigned int __dp2a_hi (ushort2 srcA, uchar4 srcB, unsigned int c)`

Two-way unsigned `int16` by `int8` dot product with unsigned `int32` accumulate, taking the upper half of the second input.

### Description

Takes two packed 16-bit integers from `srcA` vector and two packed 8-bit integers from the upper 16 bits of `srcB` vector, then creates two pairwise  $8 \times 16$  products and adds them together to an unsigned 32-bit integer `c`.

## `__device__ int __dp2a_hi (short2 srcA, char4 srcB, int c)`

Two-way signed `int16` by `int8` dot product with `int32` accumulate, taking the upper half of the second input.

### Description

Takes two packed 16-bit integers from `srcA` vector and two packed 8-bit integers from the upper 16 bits of `srcB` vector, then creates two pairwise  $8 \times 16$  products and adds them together to a signed 32-bit integer `c`.

## `__device__ unsigned int __dp2a_hi (unsigned int srcA, unsigned int srcB, unsigned int c)`

Two-way unsigned `int16` by `int8` dot product with unsigned `int32` accumulate, taking the upper half of the second input.

### Description

Extracts two packed 16-bit integers from `srcA` and two packed 8-bit integers from the upper 16 bits of `srcB`, then creates two pairwise  $8 \times 16$  products and adds them together to an unsigned 32-bit integer `c`.

## `__device__ int __dp2a_hi (int srcA, int srcB, int c)`

Two-way signed `int16` by `int8` dot product with `int32` accumulate, taking the upper half of the second input.

### Description

Extracts two packed 16-bit integers from `srcA` and two packed 8-bit integers from the upper 16 bits of `srcB`, then creates two pairwise  $8 \times 16$  products and adds them together to a signed 32-bit integer `c`.



## `__device__ unsigned int __dp2a_lo (ushort2 srcA, uchar4 srcB, unsigned int c)`

Two-way unsigned `int16` by `int8` dot product with unsigned `int32` accumulate, taking the lower half of the second input.

### Description

Takes two packed 16-bit integers from `srcA` vector and two packed 8-bit integers from the lower 16 bits of `srcB` vector, then creates two pairwise `8x16` products and adds them together to an unsigned 32-bit integer `c`.

## `__device__ int __dp2a_lo (short2 srcA, char4 srcB, int c)`

Two-way signed `int16` by `int8` dot product with `int32` accumulate, taking the lower half of the second input.

### Description

Takes two packed 16-bit integers from `srcA` vector and two packed 8-bit integers from the lower 16 bits of `srcB` vector, then creates two pairwise `8x16` products and adds them together to a signed 32-bit integer `c`.

## `__device__ unsigned int __dp2a_lo (unsigned int srcA, unsigned int srcB, unsigned int c)`

Two-way unsigned `int16` by `int8` dot product with unsigned `int32` accumulate, taking the lower half of the second input.

### Description

Extracts two packed 16-bit integers from `srcA` and two packed 8-bit integers from the lower 16 bits of `srcB`, then creates two pairwise `8x16` products and adds them together to an unsigned 32-bit integer `c`.

## `__device__ int __dp2a_lo (int srcA, int srcB, int c)`

Two-way signed `int16` by `int8` dot product with `int32` accumulate, taking the lower half of the second input.

### Description

Extracts two packed 16-bit integers from `srcA` and two packed 8-bit integers from the lower 16 bits of `srcB`, then creates two pairwise `8x16` products and adds them together to a signed 32-bit integer `c`.

**\_\_device\_\_ unsigned int \_\_dp4a (uchar4 srcA, uchar4 srcB, unsigned int c)**

Four-way unsigned `int8` dot product with unsigned `int32` accumulate.

#### Description

Takes four pairs of packed byte-sized integers from `srcA` and `srcB` vectors, then creates four pairwise products and adds them together to an unsigned 32-bit integer `c`.

**\_\_device\_\_ int \_\_dp4a (char4 srcA, char4 srcB, int c)**

Four-way signed `int8` dot product with `int32` accumulate.

#### Description

Takes four pairs of packed byte-sized integers from `srcA` and `srcB` vectors, then creates four pairwise products and adds them together to a signed 32-bit integer `c`.

**\_\_device\_\_ unsigned int \_\_dp4a (unsigned int srcA, unsigned int srcB, unsigned int c)**

Four-way unsigned `int8` dot product with unsigned `int32` accumulate.

#### Description

Extracts four pairs of packed byte-sized integers from `srcA` and `srcB`, then creates four pairwise products and adds them together to an unsigned 32-bit integer `c`.

**\_\_device\_\_ int \_\_dp4a (int srcA, int srcB, int c)**

Four-way signed `int8` dot product with `int32` accumulate.

#### Description

Extracts four pairs of packed byte-sized integers from `srcA` and `srcB`, then creates four pairwise products and adds them together to a signed 32-bit integer `c`.

**\_\_device\_\_ int \_\_ffs (int x)**

Find the position of the least significant bit set to 1 in a 32-bit integer.

#### Returns

Returns a value between 0 and 32 inclusive representing the position of the first bit set.

- ▶ `__ffs(0)` returns 0.

## Description

Find the position of the first (least significant) bit set to 1 in  $x$ , where the least significant bit position is 1.

### `__device__ int __ffsll (long long int x)`

Find the position of the least significant bit set to 1 in a 64-bit integer.

## Returns

Returns a value between 0 and 64 inclusive representing the position of the first bit set.

- ▶ `__ffsll(0)` returns 0.

## Description

Find the position of the first (least significant) bit set to 1 in  $x$ , where the least significant bit position is 1.

### `__device__ unsigned __fns (unsigned mask, unsigned base, int offset)`

Find the position of the  $n$ -th set to 1 bit in a 32-bit integer.

## Returns

Returns a value between 0 and 32 inclusive representing the position of the  $n$ -th set bit.

- ▶ parameter `base` must be  $\leq 31$ , otherwise behavior is undefined.

## Description

Given a 32-bit value `mask` and an integer value `base` (between 0 and 31), find the  $n$ -th (given by `offset`) set bit in `mask` from the `base` bit. If not found, return `0xFFFFFFFF`.

See also <https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#integer-arithmetic-instructions-fns> for more information.

### `__device__ unsigned int __funnelshift_l (unsigned int lo, unsigned int hi, unsigned int shift)`

Concatenate `hi` : `lo`, shift left by `shift` & 31 bits, return the most significant 32 bits.

## Returns

Returns the most significant 32 bits of the shifted 64-bit value.

## Description

Shift the 64-bit value formed by concatenating argument `lo` and `hi` left by the amount specified by the argument `shift`. Argument `lo` holds bits 31:0 and argument `hi` holds bits 63:32 of the 64-bit source value. The source is shifted left by the wrapped value of `shift` (`shift & 31`). The most significant 32-bits of the result are returned.

## `__device__ unsigned int __funnelshift_lc (unsigned int lo, unsigned int hi, unsigned int shift)`

Concatenate `hi : lo`, shift left by `min(shift, 32)` bits, return the most significant 32 bits.

## Returns

Returns the most significant 32 bits of the shifted 64-bit value.

## Description

Shift the 64-bit value formed by concatenating argument `lo` and `hi` left by the amount specified by the argument `shift`. Argument `lo` holds bits 31:0 and argument `hi` holds bits 63:32 of the 64-bit source value. The source is shifted left by the clamped value of `shift` (`min(shift, 32)`). The most significant 32-bits of the result are returned.

## `__device__ unsigned int __funnelshift_r (unsigned int lo, unsigned int hi, unsigned int shift)`

Concatenate `hi : lo`, shift right by `shift & 31` bits, return the least significant 32 bits.

## Returns

Returns the least significant 32 bits of the shifted 64-bit value.

## Description

Shift the 64-bit value formed by concatenating argument `lo` and `hi` right by the amount specified by the argument `shift`. Argument `lo` holds bits 31:0 and argument `hi` holds bits 63:32 of the 64-bit source value. The source is shifted right by the wrapped value of `shift` (`shift & 31`). The least significant 32-bits of the result are returned.

## `__device__ unsigned int __funnelshift_rc (unsigned int lo, unsigned int hi, unsigned int shift)`

Concatenate `hi : lo`, shift right by `min(shift, 32)` bits, return the least significant 32 bits.

### Returns

Returns the least significant 32 bits of the shifted 64-bit value.

### Description

Shift the 64-bit value formed by concatenating argument `lo` and `hi` right by the amount specified by the argument `shift`. Argument `lo` holds bits 31:0 and argument `hi` holds bits 63:32 of the 64-bit source value. The source is shifted right by the clamped value of `shift` (`min(shift, 32)`). The least significant 32-bits of the result are returned.

## `__device__ int __hadd (int x, int y)`

Compute average of signed input arguments, avoiding overflow in the intermediate sum.

### Returns

Returns a signed integer value representing the signed average value of the two inputs.

### Description

Compute average of signed input arguments `x` and `y` as `( x + y ) >> 1`, avoiding overflow in the intermediate sum.

## `__device__ int __mul24 (int x, int y)`

Calculate the least significant 32 bits of the product of the least significant 24 bits of two integers.

### Returns

Returns the least significant 32 bits of the product `x * y`.

### Description

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.

## `__device__ long long int __mul64hi (long long int x, long long int y)`

Calculate the most significant 64 bits of the product of the two 64-bit integers.

### Returns

Returns the most significant 64 bits of the product  $x * y$ .

### Description

Calculate the most significant 64 bits of the 128-bit product  $x * y$ , where  $x$  and  $y$  are 64-bit integers.

## `__device__ int __mulhi (int x, int y)`

Calculate the most significant 32 bits of the product of the two 32-bit integers.

### Returns

Returns the most significant 32 bits of the product  $x * y$ .

### Description

Calculate the most significant 32 bits of the 64-bit product  $x * y$ , where  $x$  and  $y$  are 32-bit integers.

## `__device__ int __popc (unsigned int x)`

Count the number of bits that are set to 1 in a 32-bit integer.

### Returns

Returns a value between 0 and 32 inclusive representing the number of set bits.

### Description

Count the number of bits that are set to 1 in  $x$ .

## `__device__ int __popc ll (unsigned long long int x)`

Count the number of bits that are set to 1 in a 64-bit integer.

### Returns

Returns a value between 0 and 64 inclusive representing the number of set bits.

## Description

Count the number of bits that are set to 1 in  $x$ .

## `__device__ int __rhadd (int x, int y)`

Compute rounded average of signed input arguments, avoiding overflow in the intermediate sum.

## Returns

Returns a signed integer value representing the signed rounded average value of the two inputs.

## Description

Compute average of signed input arguments  $x$  and  $y$  as  $(x + y + 1) \gg 1$ , avoiding overflow in the intermediate sum.

## `__device__ unsigned int __sad (int x, int y, unsigned int z)`

Calculate  $|x - y| + z$ , the sum of absolute difference.

## Returns

Returns  $|x - y| + z$ .

## Description

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.

## `__device__ unsigned int __uhadd (unsigned int x, unsigned int y)`

Compute average of unsigned input arguments, avoiding overflow in the intermediate sum.

## Returns

Returns an unsigned integer value representing the unsigned average value of the two inputs.

### Description

Compute average of unsigned input arguments  $x$  and  $y$  as  $(x + y) \gg 1$ , avoiding overflow in the intermediate sum.

`__device__ unsigned int __umul24 (unsigned int x, unsigned int y)`

Calculate the least significant 32 bits of the product of the least significant 24 bits of two unsigned integers.

### Returns

Returns the least significant 32 bits of the product  $x * y$ .

### Description

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.

`__device__ unsigned long long int __umul64hi (unsigned long long int x, unsigned long long int y)`

Calculate the most significant 64 bits of the product of the two 64 unsigned bit integers.

### Returns

Returns the most significant 64 bits of the product  $x * y$ .

### Description

Calculate the most significant 64 bits of the 128-bit product  $x * y$ , where  $x$  and  $y$  are 64-bit unsigned integers.

`__device__ unsigned int __umulhi (unsigned int x, unsigned int y)`

Calculate the most significant 32 bits of the product of the two 32-bit unsigned integers.

### Returns

Returns the most significant 32 bits of the product  $x * y$ .



### Description

Calculate the most significant 32 bits of the 64-bit product  $x * y$ , where  $x$  and  $y$  are 32-bit unsigned integers.

`__device__ unsigned int __urhadd (unsigned int x, unsigned int y)`

Compute rounded average of unsigned input arguments, avoiding overflow in the intermediate sum.

### Returns

Returns an unsigned integer value representing the unsigned rounded average value of the two inputs.

### Description

Compute average of unsigned input arguments  $x$  and  $y$  as  $(x + y + 1) >> 1$ , avoiding overflow in the intermediate sum.

`__device__ unsigned int __usad (unsigned int x, unsigned int y, unsigned int z)`

Calculate  $|x - y| + z$ , the sum of absolute difference.

### Returns

Returns  $|x - y| + z$ .

### Description

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.

## 1.11. Type Casting Intrinsic

This section describes type casting intrinsic functions that are only supported in device code. To use these functions you do not need to include any additional header files in your program.

## `__device__ float __double2float_rd (double x)`

Convert a double to a float in round-down mode.

### Returns

Returns converted value.

### Description

Convert the double-precision floating-point value  $x$  to a single-precision floating-point value in round-down (to negative infinity) mode.

## `__device__ float __double2float_rn (double x)`

Convert a double to a float in round-to-nearest-even mode.

### Returns

Returns converted value.

### Description

Convert the double-precision floating-point value  $x$  to a single-precision floating-point value in round-to-nearest-even mode.

## `__device__ float __double2float_ru (double x)`

Convert a double to a float in round-up mode.

### Returns

Returns converted value.

### Description

Convert the double-precision floating-point value  $x$  to a single-precision floating-point value in round-up (to positive infinity) mode.

## `__device__ float __double2float_rz (double x)`

Convert a double to a float in round-towards-zero mode.

### Returns

Returns converted value.

### Description

Convert the double-precision floating-point value  $x$  to a single-precision floating-point value in round-towards-zero mode.

`__device__ int __double2hiint (double x)`

Reinterpret high 32 bits in a double as a signed integer.

### Returns

Returns reinterpreted value.

### Description

Reinterpret the high 32 bits in the double-precision floating-point value  $x$  as a signed integer.

`__device__ int __double2int_rd (double x)`

Convert a double to a signed int in round-down mode.

### Returns

Returns converted value.

### Description

Convert the double-precision floating-point value  $x$  to a signed integer value in round-down (to negative infinity) mode.

`__device__ int __double2int_rn (double x)`

Convert a double to a signed int in round-to-nearest-even mode.

### Returns

Returns converted value.

### Description

Convert the double-precision floating-point value  $x$  to a signed integer value in round-to-nearest-even mode.

## `__device__ int __double2int_ru (double x)`

Convert a double to a signed int in round-up mode.

### Returns

Returns converted value.

### Description

Convert the double-precision floating-point value  $x$  to a signed integer value in round-up (to positive infinity) mode.

## `__device__ int __double2int_rz (double x)`

Convert a double to a signed int in round-towards-zero mode.

### Returns

Returns converted value.

### Description

Convert the double-precision floating-point value  $x$  to a signed integer value in round-towards-zero mode.

## `__device__ long long int __double2ll_rd (double x)`

Convert a double to a signed 64-bit int in round-down mode.

### Returns

Returns converted value.

### Description

Convert the double-precision floating-point value  $x$  to a signed 64-bit integer value in round-down (to negative infinity) mode.

## `__device__ long long int __double2ll_rn (double x)`

Convert a double to a signed 64-bit int in round-to-nearest-even mode.

### Returns

Returns converted value.

### Description

Convert the double-precision floating-point value  $x$  to a signed 64-bit integer value in round-to-nearest-even mode.

`__device__ long long int __double2ll_ru (double x)`

Convert a double to a signed 64-bit int in round-up mode.

### Returns

Returns converted value.

### Description

Convert the double-precision floating-point value  $x$  to a signed 64-bit integer value in round-up (to positive infinity) mode.

`__device__ long long int __double2ll_rz (double x)`

Convert a double to a signed 64-bit int in round-towards-zero mode.

### Returns

Returns converted value.

### Description

Convert the double-precision floating-point value  $x$  to a signed 64-bit integer value in round-towards-zero mode.

`__device__ int __double2loint (double x)`

Reinterpret low 32 bits in a double as a signed integer.

### Returns

Returns reinterpreted value.

### Description

Reinterpret the low 32 bits in the double-precision floating-point value  $x$  as a signed integer.

## `__device__ unsigned int __double2uint_rd (double x)`

Convert a double to an unsigned int in round-down mode.

### Returns

Returns converted value.

### Description

Convert the double-precision floating-point value  $x$  to an unsigned integer value in round-down (to negative infinity) mode.

## `__device__ unsigned int __double2uint_rn (double x)`

Convert a double to an unsigned int in round-to-nearest-even mode.

### Returns

Returns converted value.

### Description

Convert the double-precision floating-point value  $x$  to an unsigned integer value in round-to-nearest-even mode.

## `__device__ unsigned int __double2uint_ru (double x)`

Convert a double to an unsigned int in round-up mode.

### Returns

Returns converted value.

### Description

Convert the double-precision floating-point value  $x$  to an unsigned integer value in round-up (to positive infinity) mode.

## `__device__ unsigned int __double2uint_rz (double x)`

Convert a double to an unsigned int in round-towards-zero mode.

### Returns

Returns converted value.

### Description

Convert the double-precision floating-point value  $x$  to an unsigned integer value in round-towards-zero mode.

`__device__ unsigned long long int __double2ull_rd(double x)`

Convert a double to an unsigned 64-bit int in round-down mode.

### Returns

Returns converted value.

### Description

Convert the double-precision floating-point value  $x$  to an unsigned 64-bit integer value in round-down (to negative infinity) mode.

`__device__ unsigned long long int __double2ull_rn(double x)`

Convert a double to an unsigned 64-bit int in round-to-nearest-even mode.

### Returns

Returns converted value.

### Description

Convert the double-precision floating-point value  $x$  to an unsigned 64-bit integer value in round-to-nearest-even mode.

`__device__ unsigned long long int __double2ull_ru(double x)`

Convert a double to an unsigned 64-bit int in round-up mode.

### Returns

Returns converted value.

### Description

Convert the double-precision floating-point value  $x$  to an unsigned 64-bit integer value in round-up (to positive infinity) mode.

## `__device__ unsigned long long int __double2ull_rz (double x)`

Convert a double to an unsigned 64-bit int in round-towards-zero mode.

### Returns

Returns converted value.

### Description

Convert the double-precision floating-point value  $x$  to an unsigned 64-bit integer value in round-towards-zero mode.

## `__device__ long long int __double_as_longlong (double x)`

Reinterpret bits in a double as a 64-bit signed integer.

### Returns

Returns reinterpreted value.

### Description

Reinterpret the bits in the double-precision floating-point value  $x$  as a signed 64-bit integer.

## `__device__ int __float2int_rd (float x)`

Convert a float to a signed integer in round-down mode.

### Returns

Returns converted value.

### Description

Convert the single-precision floating-point value  $x$  to a signed integer in round-down (to negative infinity) mode.

## `__device__ int __float2int_rn (float x)`

Convert a float to a signed integer in round-to-nearest-even mode.

### Returns

Returns converted value.



### Description

Convert the single-precision floating-point value  $x$  to a signed integer in round-to-nearest-even mode.

`__device__ int __float2int_ru (float)`

Convert a float to a signed integer in round-up mode.

### Returns

Returns converted value.

### Description

Convert the single-precision floating-point value  $x$  to a signed integer in round-up (to positive infinity) mode.

`__device__ int __float2int_rz (float x)`

Convert a float to a signed integer in round-towards-zero mode.

### Returns

Returns converted value.

### Description

Convert the single-precision floating-point value  $x$  to a signed integer in round-towards-zero mode.

`__device__ long long int __float2ll_rd (float x)`

Convert a float to a signed 64-bit integer in round-down mode.

### Returns

Returns converted value.

### Description

Convert the single-precision floating-point value  $x$  to a signed 64-bit integer in round-down (to negative infinity) mode.

## `__device__ long long int __float2ll_rn (float x)`

Convert a float to a signed 64-bit integer in round-to-nearest-even mode.

### Returns

Returns converted value.

### Description

Convert the single-precision floating-point value  $x$  to a signed 64-bit integer in round-to-nearest-even mode.

## `__device__ long long int __float2ll_ru (float x)`

Convert a float to a signed 64-bit integer in round-up mode.

### Returns

Returns converted value.

### Description

Convert the single-precision floating-point value  $x$  to a signed 64-bit integer in round-up (to positive infinity) mode.

## `__device__ long long int __float2ll_rz (float x)`

Convert a float to a signed 64-bit integer in round-towards-zero mode.

### Returns

Returns converted value.

### Description

Convert the single-precision floating-point value  $x$  to a signed 64-bit integer in round-towards-zero mode.

## `__device__ unsigned int __float2uint_rd (float x)`

Convert a float to an unsigned integer in round-down mode.

### Returns

Returns converted value.

### Description

Convert the single-precision floating-point value  $x$  to an unsigned integer in round-down (to negative infinity) mode.

`__device__ unsigned int __float2uint_rn (float x)`

Convert a float to an unsigned integer in round-to-nearest-even mode.

### Returns

Returns converted value.

### Description

Convert the single-precision floating-point value  $x$  to an unsigned integer in round-to-nearest-even mode.

`__device__ unsigned int __float2uint_ru (float x)`

Convert a float to an unsigned integer in round-up mode.

### Returns

Returns converted value.

### Description

Convert the single-precision floating-point value  $x$  to an unsigned integer in round-up (to positive infinity) mode.

`__device__ unsigned int __float2uint_rz (float x)`

Convert a float to an unsigned integer in round-towards-zero mode.

### Returns

Returns converted value.

### Description

Convert the single-precision floating-point value  $x$  to an unsigned integer in round-towards-zero mode.

`__device__ unsigned long long int __float2ull_rd (float x)`

Convert a float to an unsigned 64-bit integer in round-down mode.

#### Returns

Returns converted value.

#### Description

Convert the single-precision floating-point value  $x$  to an unsigned 64-bit integer in round-down (to negative infinity) mode.

`__device__ unsigned long long int __float2ull_rn (float x)`

Convert a float to an unsigned 64-bit integer in round-to-nearest-even mode.

#### Returns

Returns converted value.

#### Description

Convert the single-precision floating-point value  $x$  to an unsigned 64-bit integer in round-to-nearest-even mode.

`__device__ unsigned long long int __float2ull_ru (float x)`

Convert a float to an unsigned 64-bit integer in round-up mode.

#### Returns

Returns converted value.

#### Description

Convert the single-precision floating-point value  $x$  to an unsigned 64-bit integer in round-up (to positive infinity) mode.

`__device__ unsigned long long int __float2ull_rz (float x)`

Convert a float to an unsigned 64-bit integer in round-towards-zero mode.

### Returns

Returns converted value.

### Description

Convert the single-precision floating-point value  $x$  to an unsigned 64-bit integer in round-towards-zero mode.

`__device__ int __float_as_int (float x)`

Reinterpret bits in a float as a signed integer.

### Returns

Returns reinterpreted value.

### Description

Reinterpret the bits in the single-precision floating-point value  $x$  as a signed integer.

`__device__ unsigned int __float_as_uint (float x)`

Reinterpret bits in a float as a unsigned integer.

### Returns

Returns reinterpreted value.

### Description

Reinterpret the bits in the single-precision floating-point value  $x$  as a unsigned integer.

`__device__ double __hiloint2double (int hi, int lo)`

Reinterpret high and low 32-bit integer values as a double.

### Returns

Returns reinterpreted value.

### Description

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.

`__device__ double __int2double_rn (int x)`

Convert a signed int to a double.

### Returns

Returns converted value.

### Description

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

`__device__ float __int2float_rd (int x)`

Convert a signed integer to a float in round-down mode.

### Returns

Returns converted value.

### Description

Convert the signed integer value `x` to a single-precision floating-point value in round-down (to negative infinity) mode.

`__device__ float __int2float_rn (int x)`

Convert a signed integer to a float in round-to-nearest-even mode.

### Returns

Returns converted value.

### Description

Convert the signed integer value `x` to a single-precision floating-point value in round-to-nearest-even mode.

## `__device__ float __int2float_ru (int x)`

Convert a signed integer to a float in round-up mode.

### Returns

Returns converted value.

### Description

Convert the signed integer value  $x$  to a single-precision floating-point value in round-up (to positive infinity) mode.

## `__device__ float __int2float_rz (int x)`

Convert a signed integer to a float in round-towards-zero mode.

### Returns

Returns converted value.

### Description

Convert the signed integer value  $x$  to a single-precision floating-point value in round-towards-zero mode.

## `__device__ float __int_as_float (int x)`

Reinterpret bits in an integer as a float.

### Returns

Returns reinterpreted value.

### Description

Reinterpret the bits in the signed integer value  $x$  as a single-precision floating-point value.

## `__device__ double __ll2double_rd (long long int x)`

Convert a signed 64-bit int to a double in round-down mode.

### Returns

Returns converted value.

### Description

Convert the signed 64-bit integer value  $x$  to a double-precision floating-point value in round-down (to negative infinity) mode.

`__device__ double __ll2double_rn (long long int x)`

Convert a signed 64-bit int to a double in round-to-nearest-even mode.

### Returns

Returns converted value.

### Description

Convert the signed 64-bit integer value  $x$  to a double-precision floating-point value in round-to-nearest-even mode.

`__device__ double __ll2double_ru (long long int x)`

Convert a signed 64-bit int to a double in round-up mode.

### Returns

Returns converted value.

### Description

Convert the signed 64-bit integer value  $x$  to a double-precision floating-point value in round-up (to positive infinity) mode.

`__device__ double __ll2double_rz (long long int x)`

Convert a signed 64-bit int to a double in round-towards-zero mode.

### Returns

Returns converted value.

### Description

Convert the signed 64-bit integer value  $x$  to a double-precision floating-point value in round-towards-zero mode.



## `__device__ float __ll2float_rd (long long int x)`

Convert a signed integer to a float in round-down mode.

### Returns

Returns converted value.

### Description

Convert the signed integer value  $x$  to a single-precision floating-point value in round-down (to negative infinity) mode.

## `__device__ float __ll2float_rn (long long int x)`

Convert a signed 64-bit integer to a float in round-to-nearest-even mode.

### Returns

Returns converted value.

### Description

Convert the signed 64-bit integer value  $x$  to a single-precision floating-point value in round-to-nearest-even mode.

## `__device__ float __ll2float_ru (long long int x)`

Convert a signed integer to a float in round-up mode.

### Returns

Returns converted value.

### Description

Convert the signed integer value  $x$  to a single-precision floating-point value in round-up (to positive infinity) mode.

## `__device__ float __ll2float_rz (long long int x)`

Convert a signed integer to a float in round-towards-zero mode.

### Returns

Returns converted value.

### Description

Convert the signed integer value  $x$  to a single-precision floating-point value in round-towards-zero mode.

`__device__ double __longlong_as_double (long long int x)`

Reinterpret bits in a 64-bit signed integer as a double.

### Returns

Returns reinterpreted value.

### Description

Reinterpret the bits in the 64-bit signed integer value  $x$  as a double-precision floating-point value.

`__device__ double __uint2double_rn (unsigned int x)`

Convert an unsigned int to a double.

### Returns

Returns converted value.

### Description

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

`__device__ float __uint2float_rd (unsigned int x)`

Convert an unsigned integer to a float in round-down mode.

### Returns

Returns converted value.

### Description

Convert the unsigned integer value  $x$  to a single-precision floating-point value in round-down (to negative infinity) mode.

## `__device__ float __uint2float_rn (unsigned int x)`

Convert an unsigned integer to a float in round-to-nearest-even mode.

### Returns

Returns converted value.

### Description

Convert the unsigned integer value  $x$  to a single-precision floating-point value in round-to-nearest-even mode.

## `__device__ float __uint2float_ru (unsigned int x)`

Convert an unsigned integer to a float in round-up mode.

### Returns

Returns converted value.

### Description

Convert the unsigned integer value  $x$  to a single-precision floating-point value in round-up (to positive infinity) mode.

## `__device__ float __uint2float_rz (unsigned int x)`

Convert an unsigned integer to a float in round-towards-zero mode.

### Returns

Returns converted value.

### Description

Convert the unsigned integer value  $x$  to a single-precision floating-point value in round-towards-zero mode.

## `__device__ float __uint_as_float (unsigned int x)`

Reinterpret bits in an unsigned integer as a float.

### Returns

Returns reinterpreted value.

### Description

Reinterpret the bits in the unsigned integer value  $x$  as a single-precision floating-point value.

`__device__ double __ull2double_rd (unsigned long long int x)`

Convert an unsigned 64-bit int to a double in round-down mode.

### Returns

Returns converted value.

### Description

Convert the unsigned 64-bit integer value  $x$  to a double-precision floating-point value in round-down (to negative infinity) mode.

`__device__ double __ull2double_rn (unsigned long long int x)`

Convert an unsigned 64-bit int to a double in round-to-nearest-even mode.

### Returns

Returns converted value.

### Description

Convert the unsigned 64-bit integer value  $x$  to a double-precision floating-point value in round-to-nearest-even mode.

`__device__ double __ull2double_ru (unsigned long long int x)`

Convert an unsigned 64-bit int to a double in round-up mode.

### Returns

Returns converted value.

### Description

Convert the unsigned 64-bit integer value  $x$  to a double-precision floating-point value in round-up (to positive infinity) mode.

## `__device__ double __ull2double_rz (unsigned long long int x)`

Convert an unsigned 64-bit int to a double in round-towards-zero mode.

### Returns

Returns converted value.

### Description

Convert the unsigned 64-bit integer value  $x$  to a double-precision floating-point value in round-towards-zero mode.

## `__device__ float __ull2float_rd (unsigned long long int x)`

Convert an unsigned integer to a float in round-down mode.

### Returns

Returns converted value.

### Description

Convert the unsigned integer value  $x$  to a single-precision floating-point value in round-down (to negative infinity) mode.

## `__device__ float __ull2float_rn (unsigned long long int x)`

Convert an unsigned integer to a float in round-to-nearest-even mode.

### Returns

Returns converted value.

### Description

Convert the unsigned integer value  $x$  to a single-precision floating-point value in round-to-nearest-even mode.

`__device__ float __ull2float_ru (unsigned long long int x)`

Convert an unsigned integer to a float in round-up mode.

#### Returns

Returns converted value.

#### Description

Convert the unsigned integer value `x` to a single-precision floating-point value in round-up (to positive infinity) mode.

`__device__ float __ull2float_rz (unsigned long long int x)`

Convert an unsigned integer to a float in round-towards-zero mode.

#### Returns

Returns converted value.

#### Description

Convert the unsigned integer value `x` to a single-precision floating-point value in round-towards-zero mode.

## 1.12. SIMD Intrinsic

This section describes SIMD intrinsic functions that are only supported in device code. To use these functions you do not need to include any additional header files in your program.

`__device__ unsigned int __vabs2 (unsigned int a)`

Computes per-halfword absolute value.

#### Returns

Returns computed value.

#### Description

Splits 4 bytes of argument into 2 parts, each consisting of 2 bytes, then computes absolute value for each of parts. Partial results are recombined and returned as unsigned int.

## `__device__ unsigned int __vabs4 (unsigned int a)`

Computes per-byte absolute value.

### Returns

Returns computed value.

### Description

Splits argument by bytes. Computes absolute value of each byte. Partial results are recombined and returned as unsigned int.

## `__device__ unsigned int __vabsdiffs2 (unsigned int a, unsigned int b)`

Computes per-halfword sum of absolute difference of signed integer.

### Returns

Returns computed value.

### Description

Splits 4 bytes of each into 2 parts, each consisting of 2 bytes. For corresponding parts function computes absolute difference. Partial results are recombined and returned as unsigned int.

## `__device__ unsigned int __vabsdiffs4 (unsigned int a, unsigned int b)`

Computes per-byte absolute difference of signed integer.

### Returns

Returns computed value.

### Description

Splits 4 bytes of each into 4 parts, each consisting of 1 byte. For corresponding parts function computes absolute difference. Partial results are recombined and returned as unsigned int.

## `__device__ unsigned int __vabsdiffu2 (unsigned int a, unsigned int b)`

Performs per-halfword absolute difference of unsigned integer computation:  $|a - b|$ .

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts function computes absolute difference. Partial results are recombined and returned as unsigned int.

## `__device__ unsigned int __vabsdiffu4 (unsigned int a, unsigned int b)`

Computes per-byte absolute difference of unsigned integer.

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts function computes absolute difference. Partial results are recombined and returned as unsigned int.

## `__device__ unsigned int __vabsss2 (unsigned int a)`

Computes per-halfword absolute value with signed saturation.

### Returns

Returns computed value.

### Description

Splits 4 bytes of argument into 2 parts, each consisting of 2 bytes, then computes absolute value with signed saturation for each of parts. Partial results are recombined and returned as unsigned int.



## `__device__ unsigned int __vabsss4 (unsigned int a)`

Computes per-byte absolute value with signed saturation.

### Returns

Returns computed value.

### Description

Splits 4 bytes of argument into 4 parts, each consisting of 1 byte, then computes absolute value with signed saturation for each of parts. Partial results are recombined and returned as unsigned int.

## `__device__ unsigned int __vadd2 (unsigned int a, unsigned int b)`

Performs per-halfword (un)signed addition, with wrap-around:  $a + b$ .

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes, then performs unsigned addition on corresponding parts. Partial results are recombined and returned as unsigned int.

## `__device__ unsigned int __vadd4 (unsigned int a, unsigned int b)`

Performs per-byte (un)signed addition.

### Returns

Returns computed value.

### Description

Splits 'a' into 4 bytes, then performs unsigned addition on each of these bytes with the corresponding byte from 'b', ignoring overflow. Partial results are recombined and returned as unsigned int.

## `__device__ unsigned int __vaddss2 (unsigned int a, unsigned int b)`

Performs per-halfword addition with signed saturation.

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes, then performs addition with signed saturation on corresponding parts. Partial results are recombined and returned as unsigned int.

## `__device__ unsigned int __vaddss4 (unsigned int a, unsigned int b)`

Performs per-byte addition with signed saturation.

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte, then performs addition with signed saturation on corresponding parts. Partial results are recombined and returned as unsigned int.

## `__device__ unsigned int __vaddus2 (unsigned int a, unsigned int b)`

Performs per-halfword addition with unsigned saturation.

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes, then performs addition with unsigned saturation on corresponding parts.

## `__device__ unsigned int __vaddus4 (unsigned int a, unsigned int b)`

Performs per-byte addition with unsigned saturation.

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte, then performs addition with unsigned saturation on corresponding parts.

## `__device__ unsigned int __vavgs2 (unsigned int a, unsigned int b)`

Performs per-halfword signed rounded average computation.

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes, then computes signed rounded average of corresponding parts. Partial results are recombined and returned as unsigned int.

## `__device__ unsigned int __vavgs4 (unsigned int a, unsigned int b)`

Computes per-byte signed rounded average.

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. then computes signed rounded average of corresponding parts. Partial results are recombined and returned as unsigned int.

## `__device__ unsigned int __vavg2 (unsigned int a, unsigned int b)`

Performs per-halfword unsigned rounded average computation.

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes, then computes unsigned rounded average of corresponding parts. Partial results are recombined and returned as unsigned int.

## `__device__ unsigned int __vavg4 (unsigned int a, unsigned int b)`

Performs per-byte unsigned rounded average.

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. then computes unsigned rounded average of corresponding parts. Partial results are recombined and returned as unsigned int.

## `__device__ unsigned int __vcmpeq2 (unsigned int a, unsigned int b)`

Performs per-halfword (un)signed comparison.

### Returns

Returns 0xffff computed value.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts result is ffff if they are equal, and 0000 otherwise. For example `__vcmpeq2(0x1234aba5, 0x1234aba6)` returns `0xffff0000`.

## `__device__ unsigned int __vcmpeq4 (unsigned int a, unsigned int b)`

Performs per-byte (un)signed comparison.

### Returns

Returns 0xff if a = b, else returns 0.

### Description

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts result is ff if they are equal, and 00 otherwise. For example `__vcmpeq4(0x1234aba5, 0x1234aba6)` returns 0xffff00.

## `__device__ unsigned int __vcmpges2 (unsigned int a, unsigned int b)`

Performs per-halfword signed comparison:  $a \geq b ? 0xffff : 0$ .

### Returns

Returns 0xffff if  $a \geq b$ , else returns 0.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts result is ffff if 'a' part  $\geq$  'b' part, and 0000 otherwise. For example `__vcmpges2(0x1234aba5, 0x1234aba6)` returns 0xffff0000.

## `__device__ unsigned int __vcmpges4 (unsigned int a, unsigned int b)`

Performs per-byte signed comparison.

### Returns

Returns 0xff if  $a \geq b$ , else returns 0.

### Description

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts result is ff if 'a' part  $\geq$  'b' part, and 00 otherwise. For example `__vcmpges4(0x1234aba5, 0x1234aba6)` returns 0xffff00.

## `__device__ unsigned int __vcmpgeu2 (unsigned int a, unsigned int b)`

Performs per-halfword unsigned comparison:  $a \geq b ? 0xffff : 0$ .

### Returns

Returns 0xffff if  $a \geq b$ , else returns 0.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts result is ffff if 'a' part  $\geq$  'b' part, and 0000 otherwise. For example `__vcmpgeu2(0x1234aba5, 0x1234aba6)` returns 0xffff0000.

## `__device__ unsigned int __vcmpgeu4 (unsigned int a, unsigned int b)`

Performs per-byte unsigned comparison.

### Returns

Returns 0xff if  $a = b$ , else returns 0.

### Description

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts result is ff if 'a' part  $\geq$  'b' part, and 00 otherwise. For example `__vcmpgeu4(0x1234aba5, 0x1234aba6)` returns 0xffffff00.

## `__device__ unsigned int __vcmpgts2 (unsigned int a, unsigned int b)`

Performs per-halfword signed comparison:  $a > b ? 0xffff : 0$ .

### Returns

Returns 0xffff if  $a > b$ , else returns 0.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts result is ffff if 'a' part  $>$  'b' part, and 0000 otherwise. For example `__vcmpgts2(0x1234aba5, 0x1234aba6)` returns 0x00000000.

## `__device__ unsigned int __vcmpgts4 (unsigned int a, unsigned int b)`

Performs per-byte signed comparison.

### Returns

Returns 0xff if  $a > b$ , else returns 0.

### Description

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts result is ff if 'a' part > 'b' part, and 00 otherwise. For example `__vcmpgts4(0x1234aba5, 0x1234aba6)` returns 0x00000000.

## `__device__ unsigned int __vcmpgtu2 (unsigned int a, unsigned int b)`

Performs per-halfword unsigned comparison:  $a > b ? 0xffff : 0$ .

### Returns

Returns 0xffff if  $a > b$ , else returns 0.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts result is ffff if 'a' part > 'b' part, and 0000 otherwise. For example `__vcmpgtu2(0x1234aba5, 0x1234aba6)` returns 0x00000000.

## `__device__ unsigned int __vcmpgtu4 (unsigned int a, unsigned int b)`

Performs per-byte unsigned comparison.

### Returns

Returns 0xff if  $a > b$ , else returns 0.

### Description

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts result is ff if 'a' part > 'b' part, and 00 otherwise. For example `__vcmpgtu4(0x1234aba5, 0x1234aba6)` returns 0x00000000.

## `__device__ unsigned int __vcmples2 (unsigned int a, unsigned int b)`

Performs per-halfword signed comparison:  $a \leq b ? 0xffff : 0$ .

### Returns

Returns 0xffff if  $a \leq b$ , else returns 0.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts result is ffff if 'a' part  $\leq$  'b' part, and 0000 otherwise. For example `__vcmples2(0x1234aba5, 0x1234aba6)` returns 0xffffffff.

## `__device__ unsigned int __vcmples4 (unsigned int a, unsigned int b)`

Performs per-byte signed comparison.

### Returns

Returns 0xff if  $a \leq b$ , else returns 0.

### Description

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts result is ff if 'a' part  $\leq$  'b' part, and 00 otherwise. For example `__vcmples4(0x1234aba5, 0x1234aba6)` returns 0xffffffff.

## `__device__ unsigned int __vcmplesu2 (unsigned int a, unsigned int b)`

Performs per-halfword unsigned comparison:  $a \leq b ? 0xffff : 0$ .

### Returns

Returns 0xffff if  $a \leq b$ , else returns 0.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts result is ffff if 'a' part  $\leq$  'b' part, and 0000 otherwise. For example `__vcmplesu2(0x1234aba5, 0x1234aba6)` returns 0xffffffff.



## `__device__ unsigned int __vcmpleu4 (unsigned int a, unsigned int b)`

Performs per-byte unsigned comparison.

### Returns

Returns 0xff if  $a \leq b$ , else returns 0.

### Description

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts result is ff if 'a' part  $\leq$  'b' part, and 00 otherwise. For example `__vcmpleu4(0x1234aba5, 0x1234aba6)` returns 0xffffffff.

## `__device__ unsigned int __vcmplts2 (unsigned int a, unsigned int b)`

Performs per-halfword signed comparison:  $a < b ? 0xffff : 0$ .

### Returns

Returns 0xffff if  $a < b$ , else returns 0.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts result is ffff if 'a' part  $<$  'b' part, and 0000 otherwise. For example `__vcmplts2(0x1234aba5, 0x1234aba6)` returns 0x0000ffff.

## `__device__ unsigned int __vcmplts4 (unsigned int a, unsigned int b)`

Performs per-byte signed comparison.

### Returns

Returns 0xff if  $a < b$ , else returns 0.

### Description

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts result is ff if 'a' part  $<$  'b' part, and 00 otherwise. For example `__vcmplts4(0x1234aba5, 0x1234aba6)` returns 0x000000ff.

## `__device__ unsigned int __vcmpltu2 (unsigned int a, unsigned int b)`

Performs per-halfword unsigned comparison:  $a < b ? 0xffff : 0$ .

### Returns

Returns 0xffff if  $a < b$ , else returns 0.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts result is ffff if 'a' part < 'b' part, and 0000 otherwise. For example `__vcmpltu2(0x1234aba5, 0x1234aba6)` returns 0x0000ffff.

## `__device__ unsigned int __vcmpltu4 (unsigned int a, unsigned int b)`

Performs per-byte unsigned comparison.

### Returns

Returns 0xff if  $a < b$ , else returns 0.

### Description

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts result is ff if 'a' part < 'b' part, and 00 otherwise. For example `__vcmpltu4(0x1234aba5, 0x1234aba6)` returns 0x000000ff.

## `__device__ unsigned int __vcmpne2 (unsigned int a, unsigned int b)`

Performs per-halfword (un)signed comparison:  $a != b ? 0xffff : 0$ .

### Returns

Returns 0xffff if  $a != b$ , else returns 0.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts result is ffff if 'a' part != 'b' part, and 0000 otherwise. For example `__vcmpltu2(0x1234aba5, 0x1234aba6)` returns 0x0000ffff.

## `__device__ unsigned int __vcmpne4 (unsigned int a, unsigned int b)`

Performs per-byte (un)signed comparison.

### Returns

Returns 0xff if a != b, else returns 0.

### Description

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts result is ff if 'a' part != 'b' part, and 00 otherwise. For example `__vcmplt4(0x1234aba5, 0x1234aba6)` returns `0x000000ff`.

## `__device__ unsigned int __vhaddu2 (unsigned int a, unsigned int b)`

Performs per-halfword unsigned average computation.

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes, then computes unsigned average of corresponding parts. Partial results are recombined and returned as unsigned int.

## `__device__ unsigned int __vhaddu4 (unsigned int a, unsigned int b)`

Computes per-byte unsigned average.

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. then computes unsigned average of corresponding parts. Partial results are recombined and returned as unsigned int.

`__host__ __device__ unsigned int __viaddmax_s16x2`  
`(const unsigned int a, const unsigned int b, const`  
`unsigned int c)`

Performs per-halfword  $\max(a + b, c)$ .

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. These 2 byte parts are interpreted as signed shorts. For corresponding parts function performs an add and compare:  $\max(a\_part + b\_part, c\_part)$  Partial results are recombined and returned as unsigned int.

`__host__ __device__ unsigned int`  
`__viaddmax_s16x2_relu` `(const unsigned int a, const`  
`unsigned int b, const unsigned int c)`

Performs per-halfword  $\max(\max(a + b, c), 0)$ .

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. These 2 byte parts are interpreted as signed shorts. For corresponding parts function performs an add, followed by a max with relu:  $\max(\max(a\_part + b\_part, c\_part), 0)$  Partial results are recombined and returned as unsigned int.

`__host__ __device__ int __viaddmax_s32` `(const int a,`  
`const int b, const int c)`

Computes  $\max(a + b, c)$ .

### Returns

Returns computed value.

### Description

Calculates the sum of signed integers a and b and takes the max with c.

```
__host__ __device__ int __viaddmax_s32_relu (const
int a, const int b, const int c)
```

Computes  $\max(\max(a + b, c), 0)$ .

### Returns

Returns computed value.

### Description

Calculates the sum of signed integers  $a$  and  $b$  and takes the max with  $c$ . If the result is less than 0 then is returned.

```
__host__ __device__ unsigned int __viaddmax_u16x2
(const unsigned int a, const unsigned int b, const
unsigned int c)
```

Performs per-halfword  $\max(a + b, c)$ .

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. These 2 byte parts are interpreted as unsigned shorts. For corresponding parts function performs an add and compare:  $\max(a\_part + b\_part, c\_part)$  Partial results are recombined and returned as unsigned int.

```
__host__ __device__ unsigned int __viaddmax_u32
(const unsigned int a, const unsigned int b, const
unsigned int c)
```

Computes  $\max(a + b, c)$ .

### Returns

Returns computed value.

### Description

Calculates the sum of unsigned integers  $a$  and  $b$  and takes the max with  $c$ .

`__host__ __device__ unsigned int __viaddmin_s16x2`  
`(const unsigned int a, const unsigned int b, const`  
`unsigned int c)`

Performs per-halfword  $\min(a + b, c)$ .

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. These 2 byte parts are interpreted as signed shorts. For corresponding parts function performs an add and compare:  $\min(a\_part + b\_part, c\_part)$  Partial results are recombined and returned as unsigned int.

`__host__ __device__ unsigned int`  
`__viaddmin_s16x2_relu` `(const unsigned int a, const`  
`unsigned int b, const unsigned int c)`

Performs per-halfword  $\max(\min(a + b, c), 0)$ .

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. These 2 byte parts are interpreted as signed shorts. For corresponding parts function performs an add, followed by a min with relu:  $\max(\min(a\_part + b\_part), c\_part), 0)$  Partial results are recombined and returned as unsigned int.

`__host__ __device__ int __viaddmin_s32` `(const int a,`  
`const int b, const int c)`

Computes  $\min(a + b, c)$ .

### Returns

Returns computed value.

### Description

Calculates the sum of signed integers  $a$  and  $b$  and takes the min with  $c$ .

```
__host__ __device__ int __viaddmin_s32_relu (const
int a, const int b, const int c)
```

Computes  $\max(\min(a + b, c), 0)$ .

### Returns

Returns computed value.

### Description

Calculates the sum of signed integers  $a$  and  $b$  and takes the min with  $c$ . If the result is less than 0 then is returned.

```
__host__ __device__ unsigned int __viaddmin_u16x2
(const unsigned int a, const unsigned int b, const
unsigned int c)
```

Performs per-halfword  $\min(a + b, c)$ .

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. These 2 byte parts are interpreted as unsigned shorts. For corresponding parts function performs an add and compare:  $\min(a\_part + b\_part, c\_part)$  Partial results are recombined and returned as unsigned int.

```
__host__ __device__ unsigned int __viaddmin_u32
(const unsigned int a, const unsigned int b, const
unsigned int c)
```

Computes  $\min(a + b, c)$ .

### Returns

Returns computed value.

### Description

Calculates the sum of unsigned integers  $a$  and  $b$  and takes the min with  $c$ .

```
__host__ device__ unsigned int __vibmax_s16x2
(const unsigned int a, const unsigned int b, const bool
*pred_hi, const bool *pred_lo)
```

Performs per-halfword  $\max(a, b)$ , also sets the value pointed to by `pred_hi` and `pred_lo` to the per-halfword result of  $(a \geq b)$ .

### Returns

Returns computed values.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. These 2 byte parts are interpreted as signed shorts. For corresponding parts function performs a maximum ( $= \max(a\_part, b\_part)$ ). Partial results are recombined and returned as unsigned int. Sets the value pointed to by `pred_hi` to the value  $(a\_high\_part \geq b\_high\_part)$ . Sets the value pointed to by `pred_lo` to the value  $(a\_low\_part \geq b\_low\_part)$ .

```
__host__ device__ int __vibmax_s32 (const int a,
const int b, const bool *pred)
```

Computes  $\max(a, b)$ , also sets the value pointed to by `pred` to  $(a \geq b)$ .

### Returns

Returns computed values.

### Description

Calculates the maximum of `a` and `b` of two signed ints. Also sets the value pointed to by `pred` to the value  $(a \geq b)$ .

```
__host__ device__ unsigned int __vibmax_u16x2
(const unsigned int a, const unsigned int b, const bool
*pred_hi, const bool *pred_lo)
```

Performs per-halfword  $\max(a, b)$ , also sets the value pointed to by `pred_hi` and `pred_lo` to the per-halfword result of  $(a \geq b)$ .

### Returns

Returns computed values.



## Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. These 2 byte parts are interpreted as unsigned shorts. For corresponding parts function performs a maximum ( =  $\max(a\_part, b\_part)$  ). Partial results are recombined and returned as unsigned int. Sets the value pointed to by `pred_hi` to the value  $(a\_high\_part \geq b\_high\_part)$ . Sets the value pointed to by `pred_lo` to the value  $(a\_low\_part \geq b\_low\_part)$ .

```
__host__ __device__ unsigned int __vibmax_u32  
(const unsigned int a, const unsigned int b, const bool  
*pred)
```

Computes  $\max(a, b)$ , also sets the value pointed to by `pred` to  $(a \geq b)$ .

## Returns

Returns computed values.

## Description

Calculates the maximum of `a` and `b` of two unsigned ints. Also sets the value pointed to by `pred` to the value  $(a \geq b)$ .

```
__host__ __device__ unsigned int __vibmin_s16x2  
(const unsigned int a, const unsigned int b, const bool  
*pred_hi, const bool *pred_lo)
```

Performs per-halfword  $\min(a, b)$ , also sets the value pointed to by `pred_hi` and `pred_lo` to the per-halfword result of  $(a \leq b)$ .

## Returns

Returns computed values.

## Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. These 2 byte parts are interpreted as signed shorts. For corresponding parts function performs a maximum ( =  $\max(a\_part, b\_part)$  ). Partial results are recombined and returned as unsigned int. Sets the value pointed to by `pred_hi` to the value  $(a\_high\_part \leq b\_high\_part)$ . Sets the value pointed to by `pred_lo` to the value  $(a\_low\_part \leq b\_low\_part)$ .

`__host__ __device__ int __vibmin_s32 (const int a, const int b, const bool *pred)`

Computes  $\min(a, b)$ , also sets the value pointed to by `pred` to  $(a \leq b)$ .

### Returns

Returns computed values.

### Description

Calculates the minimum of `a` and `b` of two signed ints. Also sets the value pointed to by `pred` to the value  $(a \leq b)$ .

`__host__ __device__ unsigned int __vibmin_u16x2 (const unsigned int a, const unsigned int b, const bool *pred_hi, const bool *pred_lo)`

Performs per-halfword  $\min(a, b)$ , also sets the value pointed to by `pred_hi` and `pred_lo` to the per-halfword result of  $(a \leq b)$ .

### Returns

Returns computed values.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. These 2 byte parts are interpreted as unsigned shorts. For corresponding parts function performs a maximum  $( = \max(a\_part, b\_part) )$ . Partial results are recombined and returned as unsigned int. Sets the value pointed to by `pred_hi` to the value  $(a\_high\_part \leq b\_high\_part)$ . Sets the value pointed to by `pred_lo` to the value  $(a\_low\_part \leq b\_low\_part)$ .

`__host__ __device__ unsigned int __vibmin_u32 (const unsigned int a, const unsigned int b, const bool *pred)`

Computes  $\min(a, b)$ , also sets the value pointed to by `pred` to  $(a \leq b)$ .

### Returns

Returns computed values.

### Description

Calculates the minimum of a and b of two unsigned ints. Also sets the value pointed to by p to the value (a <= b).

```
__host__ __device__ unsigned int __vimax3_s16x2
(const unsigned int a, const unsigned int b, const
unsigned int c)
```

Performs per-halfword  $\max(\max(a, b), c)$ .

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. These 2 byte parts are interpreted as signed shorts. For corresponding parts function performs a 3-way max ( =  $\max(\max(a\_part, b\_part), c\_part)$  ). Partial results are recombined and returned as unsigned int.

```
__host__ __device__ unsigned int
__vimax3_s16x2_relu (const unsigned int a, const
unsigned int b, const unsigned int c)
```

Performs per-halfword  $\max(\max(\max(a, b), c), 0)$ .

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. These 2 byte parts are interpreted as signed shorts. For corresponding parts function performs a three-way max with relu ( =  $\max(a\_part, b\_part, c\_part, 0)$  ). Partial results are recombined and returned as unsigned int.

`__host__ __device__ int __vimax3_s32 (const int a, const int b, const int c)`

Computes  $\max(\max(a, b), c)$ .

### Returns

Returns computed value.

### Description

Calculates the 3-way max of signed integers a, b and c.

`__host__ __device__ int __vimax3_s32_relu (const int a, const int b, const int c)`

Computes  $\max(\max(\max(a, b), c), 0)$ .

### Returns

Returns computed value.

### Description

Calculates the maximum of three signed ints, if this is less than 0 then 0 is returned.

`__host__ __device__ unsigned int __vimax3_u16x2 (const unsigned int a, const unsigned int b, const unsigned int c)`

Performs per-halfword  $\max(\max(a, b), c)$ .

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. These 2 byte parts are interpreted as unsigned shorts. For corresponding parts function performs a 3-way max ( =  $\max(\max(a\_part, b\_part), c\_part)$  ). Partial results are recombined and returned as unsigned int.

`__host__ __device__ unsigned int __vimax3_u32`  
`(const unsigned int a, const unsigned int b, const`  
`unsigned int c)`

Computes  $\max(\max(a, b), c)$ .

#### Returns

Returns computed value.

#### Description

Calculates the 3-way max of unsigned integers a, b and c.

`__host__ __device__ unsigned int __vimax_s16x2_relu`  
`(const unsigned int a, const unsigned int b)`

Performs per-halfword  $\max(\max(a, b), 0)$ .

#### Returns

Returns computed value.

#### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. These 2 byte parts are interpreted as signed shorts. For corresponding parts function performs a max with relu ( =  $\max(a\_part, b\_part, 0)$  ). Partial results are recombined and returned as unsigned int.

`__host__ __device__ int __vimax_s32_relu` (const int  
`a, const int b)`

Computes  $\max(\max(a, b), 0)$ .

#### Returns

Returns computed value.

#### Description

Calculates the maximum of a and b of two signed ints, if this is less than 0 then 0 is returned.

`__host__ __device__ unsigned int __vmin3_s16x2`  
`(const unsigned int a, const unsigned int b, const`  
`unsigned int c)`

Performs per-halfword  $\min(\min(a, b), c)$ .

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. These 2 byte parts are interpreted as signed shorts. For corresponding parts function performs a 3-way min ( =  $\min(\min(a\_part, b\_part), c\_part)$  ). Partial results are recombined and returned as unsigned int.

`__host__ __device__ unsigned int`  
`__vmin3_s16x2_relu (const unsigned int a, const`  
`unsigned int b, const unsigned int c)`

Performs per-halfword  $\max(\min(\min(a, b), c), 0)$ .

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. These 2 byte parts are interpreted as signed shorts. For corresponding parts function performs a three-way min with relu ( =  $\max(\min(a\_part, b\_part, c\_part), 0)$  ). Partial results are recombined and returned as unsigned int.

`__host__ __device__ int __vmin3_s32 (const int a,`  
`const int b, const int c)`

Computes  $\min(\min(a, b), c)$ .

### Returns

Returns computed value.

### Description

Calculates the 3-way min of signed integers a, b and c.

```
__host__ __device__ int __vmin3_s32_relu (const int  
a, const int b, const int c)
```

Computes  $\max(\min(\min(a, b), c), 0)$ .

### Returns

Returns computed value.

### Description

Calculates the minimum of three signed ints, if this is less than 0 then 0 is returned.

```
__host__ __device__ unsigned int __vmin3_u16x2  
(const unsigned int a, const unsigned int b, const  
unsigned int c)
```

Performs per-halfword  $\min(\min(a, b), c)$ .

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. These 2 byte parts are interpreted as unsigned shorts. For corresponding parts function performs a 3-way min ( =  $\min(\min(a\_part, b\_part), c\_part)$  ). Partial results are recombined and returned as unsigned int.

```
__host__ __device__ unsigned int __vmin3_u32 (const  
unsigned int a, const unsigned int b, const unsigned  
int c)
```

Computes  $\min(\min(a, b), c)$ .

### Returns

Returns computed value.

### Description

Calculates the 3-way min of unsigned integers a, b and c.

```
__host__ __device__ unsigned int __vmin_s16x2_relu  
(const unsigned int a, const unsigned int b)
```

Performs per-halfword  $\max(\min(a, b), 0)$ .

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. These 2 byte parts are interpreted as signed shorts. For corresponding parts function performs a min with relu ( =  $\max(\min(a_{part}, b_{part}), 0)$  ). Partial results are recombined and returned as unsigned int.

```
__host__ __device__ int __vmin_s32_relu (const int a,  
const int b)
```

Computes  $\max(\min(a, b), 0)$ .

### Returns

Returns computed value.

### Description

Calculates the minimum of a and b of two signed ints, if this is less than 0 then 0 is returned.

```
__device__ unsigned int __vmaxs2 (unsigned int a,  
unsigned int b)
```

Performs per-halfword signed maximum computation.

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts function computes signed maximum. Partial results are recombined and returned as unsigned int.



## `__device__ unsigned int __vmaxs4 (unsigned int a, unsigned int b)`

Computes per-byte signed maximum.

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts function computes signed maximum. Partial results are recombined and returned as unsigned int.

## `__device__ unsigned int __vmaxu2 (unsigned int a, unsigned int b)`

Performs per-halfword unsigned maximum computation.

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts function computes unsigned maximum. Partial results are recombined and returned as unsigned int.

## `__device__ unsigned int __vmaxu4 (unsigned int a, unsigned int b)`

Computes per-byte unsigned maximum.

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts function computes unsigned maximum. Partial results are recombined and returned as unsigned int.

## `__device__ unsigned int __vmins2 (unsigned int a, unsigned int b)`

Performs per-halfword signed minimum computation.

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts function computes signed minimum. Partial results are recombined and returned as unsigned int.

## `__device__ unsigned int __vmins4 (unsigned int a, unsigned int b)`

Computes per-byte signed minimum.

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts function computes signed minimum. Partial results are recombined and returned as unsigned int.

## `__device__ unsigned int __vminu2 (unsigned int a, unsigned int b)`

Performs per-halfword unsigned minimum computation.

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts function computes unsigned minimum. Partial results are recombined and returned as unsigned int.

## `__device__ unsigned int __vminu4 (unsigned int a, unsigned int b)`

Computes per-byte unsigned minimum.

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts function computes unsigned minimum. Partial results are recombined and returned as unsigned int.

## `__device__ unsigned int __vneg2 (unsigned int a)`

Computes per-halfword negation.

### Returns

Returns computed value.

### Description

Splits 4 bytes of argument into 2 parts, each consisting of 2 bytes. For each part function computes negation. Partial results are recombined and returned as unsigned int.

## `__device__ unsigned int __vneg4 (unsigned int a)`

Performs per-byte negation.

### Returns

Returns computed value.

### Description

Splits 4 bytes of argument into 4 parts, each consisting of 1 byte. For each part function computes negation. Partial results are recombined and returned as unsigned int.

## `__device__ unsigned int __vnegss2 (unsigned int a)`

Computes per-halfword negation with signed saturation.

### Returns

Returns computed value.

### Description

Splits 4 bytes of argument into 2 parts, each consisting of 2 bytes. For each part function computes negation. Partial results are recombined and returned as unsigned int.

**\_\_device\_\_ unsigned int \_\_vnegss4 (unsigned int a)**

Performs per-byte negation with signed saturation.

### Returns

Returns computed value.

### Description

Splits 4 bytes of argument into 4 parts, each consisting of 1 byte. For each part function computes negation. Partial results are recombined and returned as unsigned int.

**\_\_device\_\_ unsigned int \_\_vsads2 (unsigned int a, unsigned int b)**

Performs per-halfword sum of absolute difference of signed.

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts function computes absolute difference and sum it up. Partial results are recombined and returned as unsigned int.

**\_\_device\_\_ unsigned int \_\_vsads4 (unsigned int a, unsigned int b)**

Computes per-byte sum of abs difference of signed.

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts function computes absolute difference and sum it up. Partial results are recombined and returned as unsigned int.

## `__device__ unsigned int __vsadu2 (unsigned int a, unsigned int b)`

Computes per-halfword sum of abs diff of unsigned.

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts function computes absolute differences and returns sum of those differences.

## `__device__ unsigned int __vsadu4 (unsigned int a, unsigned int b)`

Computes per-byte sum of abs difference of unsigned.

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts function computes absolute differences and returns sum of those differences.

## `__device__ unsigned int __vseteq2 (unsigned int a, unsigned int b)`

Performs per-halfword (un)signed comparison.

### Returns

Returns 1 if  $a = b$ , else returns 0.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts function performs comparison 'a' part == 'b' part. If both equalities are satisfied, function returns 1.

## `__device__ unsigned int __vseteq4 (unsigned int a, unsigned int b)`

Performs per-byte (un)signed comparison.

### Returns

Returns 1 if  $a = b$ , else returns 0.

### Description

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts function performs comparison 'a' part == 'b' part. If both equalities are satisfied, function returns 1.

## `__device__ unsigned int __vsetges2 (unsigned int a, unsigned int b)`

Performs per-halfword signed comparison.

### Returns

Returns 1 if  $a \geq b$ , else returns 0.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts function performs comparison 'a' part  $\geq$  'b' part. If both inequalities are satisfied, function returns 1.

## `__device__ unsigned int __vsetges4 (unsigned int a, unsigned int b)`

Performs per-byte signed comparison.

### Returns

Returns 1 if  $a \geq b$ , else returns 0.

### Description

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts function performs comparison 'a' part  $\geq$  'b' part. If both inequalities are satisfied, function returns 1.

## `__device__ unsigned int __vsetgeu2 (unsigned int a, unsigned int b)`

Performs per-halfword unsigned minimum unsigned comparison.

### Returns

Returns 1 if  $a \geq b$ , else returns 0.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts function performs comparison 'a' part  $\geq$  'b' part. If both inequalities are satisfied, function returns 1.

## `__device__ unsigned int __vsetgeu4 (unsigned int a, unsigned int b)`

Performs per-byte unsigned comparison.

### Returns

Returns 1 if  $a \geq b$ , else returns 0.

### Description

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts function performs comparison 'a' part  $\geq$  'b' part. If both inequalities are satisfied, function returns 1.

## `__device__ unsigned int __vsetgts2 (unsigned int a, unsigned int b)`

Performs per-halfword signed comparison.

### Returns

Returns 1 if  $a > b$ , else returns 0.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts function performs comparison 'a' part  $>$  'b' part. If both inequalities are satisfied, function returns 1.

## `__device__ unsigned int __vsetgts4 (unsigned int a, unsigned int b)`

Performs per-byte signed comparison.

### Returns

Returns 1 if  $a > b$ , else returns 0.

### Description

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts function performs comparison 'a' part > 'b' part. If both inequalities are satisfied, function returns 1.

## `__device__ unsigned int __vsetgtu2 (unsigned int a, unsigned int b)`

Performs per-halfword unsigned comparison.

### Returns

Returns 1 if  $a > b$ , else returns 0.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts function performs comparison 'a' part > 'b' part. If both inequalities are satisfied, function returns 1.

## `__device__ unsigned int __vsetgtu4 (unsigned int a, unsigned int b)`

Performs per-byte unsigned comparison.

### Returns

Returns 1 if  $a > b$ , else returns 0.

### Description

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts function performs comparison 'a' part > 'b' part. If both inequalities are satisfied, function returns 1.



## `__device__ unsigned int __vsetles2 (unsigned int a, unsigned int b)`

Performs per-halfword unsigned minimum computation.

### Returns

Returns 1 if  $a \leq b$ , else returns 0.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts function performs comparison 'a' part  $\leq$  'b' part. If both inequalities are satisfied, function returns 1.

## `__device__ unsigned int __vsetles4 (unsigned int a, unsigned int b)`

Performs per-byte signed comparison.

### Returns

Returns 1 if  $a \leq b$ , else returns 0.

### Description

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts function performs comparison 'a' part  $\leq$  'b' part. If both inequalities are satisfied, function returns 1.

## `__device__ unsigned int __vsetleu2 (unsigned int a, unsigned int b)`

Performs per-halfword signed comparison.

### Returns

Returns 1 if  $a \leq b$ , else returns 0.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts function performs comparison 'a' part  $\leq$  'b' part. If both inequalities are satisfied, function returns 1.

## `__device__ unsigned int __vsetleu4 (unsigned int a, unsigned int b)`

Performs per-byte unsigned comparison.

### Returns

Returns 1 if  $a \leq b$ , else returns 0.

### Description

Splits 4 bytes of each argument into 4 part, each consisting of 1 byte. For corresponding parts function performs comparison 'a' part  $\leq$  'b' part. If both inequalities are satisfied, function returns 1.

## `__device__ unsigned int __vsetlts2 (unsigned int a, unsigned int b)`

Performs per-halfword signed comparison.

### Returns

Returns 1 if  $a < b$ , else returns 0.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts function performs comparison 'a' part  $\leq$  'b' part. If both inequalities are satisfied, function returns 1.

## `__device__ unsigned int __vsetlts4 (unsigned int a, unsigned int b)`

Performs per-byte signed comparison.

### Returns

Returns 1 if  $a < b$ , else returns 0.

### Description

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts function performs comparison 'a' part  $\leq$  'b' part. If both inequalities are satisfied, function returns 1.

## `__device__ unsigned int __vsetltu2 (unsigned int a, unsigned int b)`

Performs per-halfword unsigned comparison.

### Returns

Returns 1 if  $a < b$ , else returns 0.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts function performs comparison 'a' part  $\leq$  'b' part. If both inequalities are satisfied, function returns 1.

## `__device__ unsigned int __vsetltu4 (unsigned int a, unsigned int b)`

Performs per-byte unsigned comparison.

### Returns

Returns 1 if  $a < b$ , else returns 0.

### Description

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts function performs comparison 'a' part  $\leq$  'b' part. If both inequalities are satisfied, function returns 1.

## `__device__ unsigned int __vsetne2 (unsigned int a, unsigned int b)`

Performs per-halfword (un)signed comparison.

### Returns

Returns 1 if  $a \neq b$ , else returns 0.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts function performs comparison 'a' part  $\neq$  'b' part. If both conditions are satisfied, function returns 1.

## `__device__ unsigned int __vsetne4 (unsigned int a, unsigned int b)`

Performs per-byte (un)signed comparison.

### Returns

Returns 1 if  $a \neq b$ , else returns 0.

### Description

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts function performs comparison 'a' part  $\neq$  'b' part. If both conditions are satisfied, function returns 1.

## `__device__ unsigned int __vsub2 (unsigned int a, unsigned int b)`

Performs per-halfword (un)signed subtraction, with wrap-around.

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts function performs subtraction. Partial results are recombined and returned as unsigned int.

## `__device__ unsigned int __vsub4 (unsigned int a, unsigned int b)`

Performs per-byte subtraction.

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts function performs subtraction. Partial results are recombined and returned as unsigned int.

## `__device__ unsigned int __vsubss2 (unsigned int a, unsigned int b)`

Performs per-halfword (un)signed subtraction, with signed saturation.

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts function performs subtraction with signed saturation. Partial results are recombined and returned as unsigned int.

## `__device__ unsigned int __vsubss4 (unsigned int a, unsigned int b)`

Performs per-byte subtraction with signed saturation.

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts function performs subtraction with signed saturation. Partial results are recombined and returned as unsigned int.

## `__device__ unsigned int __vsubus2 (unsigned int a, unsigned int b)`

Performs per-halfword subtraction with unsigned saturation.

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 2 parts, each consisting of 2 bytes. For corresponding parts function performs subtraction with unsigned saturation. Partial results are recombined and returned as unsigned int.

## `__device__ unsigned int __vsubus4 (unsigned int a, unsigned int b)`

Performs per-byte subtraction with unsigned saturation.

### Returns

Returns computed value.

### Description

Splits 4 bytes of each argument into 4 parts, each consisting of 1 byte. For corresponding parts function performs subtraction with unsigned saturation. Partial results are recombined and returned as unsigned int.

---

# Chapter 2. Data Structures

Here are the data structures with brief descriptions:

## half

\_\_half data type

## half2

\_\_half2 data type

## half2\_raw

\_\_half2\_raw data type

## half\_raw

\_\_half\_raw data type

## nv\_bfloat16

Nv\_bfloat16 datatype

## nv\_bfloat162

Nv\_bfloat162 datatype

## nv\_bfloat162\_raw

\_\_nv\_bfloat162\_raw data type

## nv\_bfloat16\_raw

\_\_nv\_bfloat16\_raw data type

## nv\_fp8\_e4m3

\_\_nv\_fp8\_e4m3 datatype

## nv\_fp8\_e5m2

\_\_nv\_fp8\_e5m2 datatype

## nv\_fp8x2\_e4m3

\_\_nv\_fp8x2\_e4m3 datatype

## nv\_fp8x2\_e5m2

\_\_nv\_fp8x2\_e5m2 datatype

## nv\_fp8x4\_e4m3

\_\_nv\_fp8x4\_e4m3 datatype

## nv\_fp8x4\_e5m2

\_\_nv\_fp8x4\_e5m2 datatype

## 2.1. `__half` Struct Reference

`__half` data type

This structure implements the datatype for storing half-precision floating-point numbers. The structure implements assignment, arithmetic and comparison operators, and type conversions. 16 bits are being used in total: 1 sign bit, 5 bits for the exponent, and the significand is being stored in 10 bits. The total precision is 11 bits. There are 15361 representable numbers within the interval [0.0, 1.0], endpoints included. On average we have  $\log_{10}(2^{11}) \sim 3.311$  decimal digits.

The objective here is to provide IEEE754-compliant implementation of `binary16` type and arithmetic with limitations due to device HW not supporting floating-point exceptions.

### unsigned short `__half::__x`

Protected storage variable contains the bits of floating-point data.

### `__half`

`__half` `cppOperationVisibility: visibility=public` `__host__` `__device__` `cppOperationConst: const=const` unsigned long long val

Construct `__half` from unsigned long long input using default round-to-nearest-even rounding mode.

### `__half`

`__half` `cppOperationVisibility: visibility=public` `__host__` `__device__` `cppOperationConst: const=const` long long val

Construct `__half` from long long input using default round-to-nearest-even rounding mode.

### `__half`

`__half` `cppOperationVisibility: visibility=public` `__host__` `__device__` `cppOperationConst: const=const` `cppOperationPrimitive: storage=unsignedlong` val

Construct `__half` from unsigned long input using default round-to-nearest-even rounding mode.

### `__half`

`__half` `cppOperationVisibility: visibility=public` `__host__` `__device__` `cppOperationConst: const=const` long val



Construct `__half` from `long` input using default round-to-nearest-even rounding mode.

## `__half`

`__half` `cppOperationVisibility: visibility=public` `__host__` `__device__` `cppOperationConst: const=const` `cppOperationPrimitive: storage=unsignedint` `val`

Construct `__half` from `unsigned int` input using default round-to-nearest-even rounding mode.

## `__half`

`__half` `cppOperationVisibility: visibility=public` `__host__` `__device__` `cppOperationConst: const=const` `cppOperationPrimitive: storage=int` `val`

Construct `__half` from `int` input using default round-to-nearest-even rounding mode.

## `__half`

`__half` `cppOperationVisibility: visibility=public` `__host__` `__device__` `cppOperationConst: const=const` `unsigned short` `val`

Construct `__half` from `unsigned short` integer input using default round-to-nearest-even rounding mode.

## `__half`

`__half` `cppOperationVisibility: visibility=public` `__host__` `__device__` `cppOperationConst: const=const` `short` `val`

Construct `__half` from `short` integer input using default round-to-nearest-even rounding mode.

## `__half`

`__half` `cppOperationVisibility: visibility=public` `__host__` `__device__` `cppOperationConst: const=const` `cppOperationPrimitive: storage=double` `f`

Construct `__half` from `double` input using default round-to-nearest-even rounding mode.

## `__half`

`__half` `cppOperationVisibility: visibility=public` `__host__` `__device__` `cppOperationConst: const=const` `cppOperationPrimitive: storage=float` `f`

Construct `__half` from `float` input using default round-to-nearest-even rounding mode.

## `__half`

`__half` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline` `__host__`  
`__device__` `cppOperationConst: const=const` `__half_raw` hr

Constructor from `__half_raw` .

## `__half`

`__half` `cppOperationVisibility: visibility=public` `__host__` `__device__`

Constructor by default.

## `__host__ __device__ operator __half_raw ()`

### Description

Type cast to `__half_raw` operator with `volatile` input.

## `__host__ __device__ operator __half_raw ()`

### Description

Type cast to `__half_raw` operator.

## `__host__ __device__ constexpr operator bool ()`

### Description

Conversion operator to `bool` data type. `+0` and `-0` inputs convert to `false`. Non-zero inputs convert to `true`.

## `__host__ __device__ operator char ()`

### Description

Conversion operator to an implementation defined `char` data type. Using round-toward-zero rounding mode.

Detects signedness of the `char` type and proceeds accordingly, see further details in signed and unsigned char operators.

`__host__ __device__ operator float ()`

#### Description

Type cast to `float` operator.

`__host__ __device__ operator int ()`

#### Description

Conversion operator to `int` data type. Using round-toward-zero rounding mode.

See [\\_\\_half2int\\_rz\(\\_\\_half\)](#) for further details

`__host__ __device__ operator long ()`

#### Description

Conversion operator to `long` data type. Using round-toward-zero rounding mode.

`__host__ __device__ operator long long ()`

#### Description

Conversion operator to `long long` data type. Using round-toward-zero rounding mode.

See [\\_\\_half2ll\\_rz\(\\_\\_half\)](#) for further details

`__host__ __device__ operator short ()`

#### Description

Conversion operator to `short` data type. Using round-toward-zero rounding mode.

See [\\_\\_half2short\\_rz\(\\_\\_half\)](#) for further details

`__host__ __device__ operator signed char ()`

#### Description

Conversion operator to `signed char` data type. Using round-toward-zero rounding mode.

See [\\_\\_half2char\\_rz\(\\_\\_half\)](#) for further details

## `__host__ device __operator unsigned char ()`

### Description

Conversion operator to `unsigned char` data type. Using round-toward-zero rounding mode.

See [\\_\\_half2uchar\\_rz\(\\_\\_half\)](#) for further details

## `__host__ device __operator unsigned int ()`

### Description

Conversion operator to `unsigned int` data type. Using round-toward-zero rounding mode.

See [\\_\\_half2uint\\_rz\(\\_\\_half\)](#) for further details

## `__host__ device __operator unsigned long ()`

### Description

Conversion operator to `unsigned long` data type. Using round-toward-zero rounding mode.

## `__host__ device __operator unsigned long long ()`

### Description

Conversion operator to `unsigned long long` data type. Using round-toward-zero rounding mode.

See [\\_\\_half2ull\\_rz\(\\_\\_half\)](#) for further details

## `__host__ device __operator unsigned short ()`

### Description

Conversion operator to `unsigned short` data type. Using round-toward-zero rounding mode.

See [\\_\\_half2ushort\\_rz\(\\_\\_half\)](#) for further details

`__host__ device __operator= (const unsigned long long val)`

#### Description

Type cast from `unsigned long long` assignment operator, using default round-to-nearest-even rounding mode.

`__host__ device __operator= (const long long val)`

#### Description

Type cast from `long long` assignment operator, using default round-to-nearest-even rounding mode.

`__host__ device __operator= (const unsigned int val)`

#### Description

Type cast from `unsigned int` assignment operator, using default round-to-nearest-even rounding mode.

`__host__ device __operator= (const int val)`

#### Description

Type cast from `int` assignment operator, using default round-to-nearest-even rounding mode.

`__host__ device __operator= (const unsigned short val)`

#### Description

Type cast from `unsigned short` assignment operator, using default round-to-nearest-even rounding mode.

`__host__ __device__ operator= (const short val)`

### Description

Type cast from `short` assignment operator, using default round-to-nearest-even rounding mode.

`__host__ __device__ operator= (const double f)`

### Description

Type cast to `__half` assignment operator from `double` input using default round-to-nearest-even rounding mode.

`__host__ __device__ operator= (const float f)`

### Description

Type cast to `__half` assignment operator from `float` input using default round-to-nearest-even rounding mode.

`__host__ __device__ operator= (const __half_raw hr)`

### Description

Assignment operator from volatile `__half_raw` to volatile `__half` .

`__host__ __device__ operator= (const __half_raw hr)`

### Description

Assignment operator from `__half_raw` to volatile `__half` .

`__host__ __device__ operator= (const __half_raw hr)`

### Description

Assignment operator from `__half_raw` .

## 2.2. `__half2` Struct Reference

`__half2` data type

This structure implements the datatype for storing two half-precision floating-point numbers. The structure implements assignment, arithmetic and comparison operators, and type conversions.

- ▶ NOTE: `__half2` is visible to non-nvcc host compilers

### `struct __half __half2::x`

Storage field holding lower `__half` part.

### `struct __half __half2::y`

Storage field holding upper `__half` part.

### `__half2`

`__half2` `cppOperationVisibility: visibility=public` `__host__` `__device__` `cppOperationConst: const=const` `__half2_raw` `h2r`

Constructor from `__half2_raw`

### `__half2`

`__half2` `cppOperationVisibility: visibility=public` `__host__` `__device__` `cppOperationConst: const=const` `__half2` `src`

Copy constructor

### `__half2`

`__half2` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline` `__host__` `__device__` `cppOperationConst: const=const` `__half` `a` `cppOperationConst: const=const` `__half` `b`

Constructor from two `__half` variables

### `__half2`

`__half2` `cppOperationVisibility: visibility=public` `__host__` `__device__`

Constructor by default.

`__host__ __device__ operator __half2_raw ()`

#### Description

Conversion operator to `__half2_raw`

`__host__ __device__ operator= (const __half2_raw h2r)`

#### Description

Assignment operator from `__half2_raw`

`__host__ __device__ operator= (const __half2 src)`

#### Description

Copy assignment operator

## 2.3. `__half2_raw` Struct Reference

`__half2_raw` data type

Type allows static initialization of `half2` until it becomes a builtin type.

- ▶ Note: this initialization is as a bit-field representation of `half2`, and not a conversion from `short2` to `half2`. Such representation will be deprecated in a future version of CUDA.
- ▶ Note: this is visible to non-nvcc compilers, including C-only compilations

## 2.4. `__half_raw` Struct Reference

`__half_raw` data type

Type allows static initialization of `half` until it becomes a builtin type.

- ▶ Note: this initialization is as a bit-field representation of `half`, and not a conversion from `short` to `half`. Such representation will be deprecated in a future version of CUDA.
- ▶ Note: this is visible to non-nvcc compilers, including C-only compilations



## 2.5. `__nv_bfloat16` Struct Reference

`nv_bfloat16` datatype

This structure implements the datatype for storing `nv_bfloat16` floating-point numbers. The structure implements assignment operators and type conversions. 16 bits are being used in total: 1 sign bit, 8 bits for the exponent, and the significand is being stored in 7 bits. The total precision is 8 bits.

### unsigned short `__nv_bfloat16::__x`

Protected storage variable contains the bits of floating-point data.

### `__nv_bfloat16`

`__nv_bfloat16` `cppOperationVisibility: visibility=public` `__host__` `__device__` unsigned long long val

Construct `__nv_bfloat16` from unsigned long long input using default round-to-nearest-even rounding mode.

### `__nv_bfloat16`

`__nv_bfloat16` `cppOperationVisibility: visibility=public` `__host__` `__device__` long long val

Construct `__nv_bfloat16` from long long input using default round-to-nearest-even rounding mode.

### `__nv_bfloat16`

`__nv_bfloat16` `cppOperationVisibility: visibility=public` `__host__` `__device__` `cppOperationConst: const=const` `cppOperationPrimitive: storage=unsignedlong` val

Construct `__nv_bfloat16` from unsigned long input using default round-to-nearest-even rounding mode.

### `__nv_bfloat16`

`__nv_bfloat16` `cppOperationVisibility: visibility=public` `__host__` `__device__` `cppOperationConst: const=const` long val

Construct `__nv_bfloat16` from long input using default round-to-nearest-even rounding mode.

## `__nv_bfloat16`

`__nv_bfloat16` `cppOperationVisibility: visibility=public` `__host__` `__device__`  
`cppOperationPrimitive: storage=unsignedint` val

Construct `__nv_bfloat16` from `unsigned int` input using default round-to-nearest-even rounding mode.

## `__nv_bfloat16`

`__nv_bfloat16` `cppOperationVisibility: visibility=public` `__host__` `__device__`  
`cppOperationPrimitive: storage=int` val

Construct `__nv_bfloat16` from `int` input using default round-to-nearest-even rounding mode.

## `__nv_bfloat16`

`__nv_bfloat16` `cppOperationVisibility: visibility=public` `__host__` `__device__` `unsigned short` val

Construct `__nv_bfloat16` from `unsigned short` integer input using default round-to-nearest-even rounding mode.

## `__nv_bfloat16`

`__nv_bfloat16` `cppOperationVisibility: visibility=public` `__host__` `__device__` `short` val

Construct `__nv_bfloat16` from `short` integer input using default round-to-nearest-even rounding mode.

## `__nv_bfloat16`

`__nv_bfloat16` `cppOperationVisibility: visibility=public` `__host__` `__device__`  
`cppOperationConst: const=const` `cppOperationPrimitive: storage=double` f

Construct `__nv_bfloat16` from `double` input using default round-to-nearest-even rounding mode.

## `__nv_bfloat16`

`__nv_bfloat16` `cppOperationVisibility: visibility=public` `__host__` `__device__`  
`cppOperationConst: const=const` `cppOperationPrimitive: storage=float` f

Construct `__nv_bfloat16` from `float` input using default round-to-nearest-even rounding mode.

## `__nv_bfloat16`

`__nv_bfloat16` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`__host__ __device__` `cppOperationConst: const=const` `__nv_bfloat16_raw` `hr`

Constructor from `__nv_bfloat16_raw` .

## `__nv_bfloat16`

`__nv_bfloat16` `cppOperationVisibility: visibility=public` `__host__ __device__`

Constructor by default.

## `__host__ __device__ operator __nv_bfloat16_raw ()`

### Description

Type cast to `__nv_bfloat16_raw` operator with `volatile` input.

## `__host__ __device__ operator __nv_bfloat16_raw ()`

### Description

Type cast to `__nv_bfloat16_raw` operator.

## `__host__ __device__ constexpr operator bool ()`

### Description

Conversion operator to `bool` data type. `+0` and `-0` inputs convert to `false`. Non-zero inputs convert to `true`.

## `__host__ __device__ operator char ()`

### Description

Conversion operator to an implementation defined `char` data type. Using round-toward-zero rounding mode.

Detects signedness of the `char` type and proceeds accordingly, see further details in signed and unsigned char operators.

`__host__ __device__ operator float ()`

#### Description

Type cast to `float` operator.

`__host__ __device__ operator int ()`

#### Description

Conversion operator to `int` data type. Using round-toward-zero rounding mode.

See [\\_\\_bfloat162int\\_rz\(\\_\\_nv\\_bfloat16\)](#) for further details

`__host__ __device__ operator long ()`

#### Description

Conversion operator to `long` data type. Using round-toward-zero rounding mode.

`__host__ __device__ operator long long ()`

#### Description

Conversion operator to `long long` data type. Using round-toward-zero rounding mode.

See [\\_\\_bfloat162ll\\_rz\(\\_\\_nv\\_bfloat16\)](#) for further details

`__host__ __device__ operator short ()`

#### Description

Conversion operator to `short` data type. Using round-toward-zero rounding mode.

See [\\_\\_bfloat162short\\_rz\(\\_\\_nv\\_bfloat16\)](#) for further details

`__host__ __device__ operator signed char ()`

#### Description

Conversion operator to `signed char` data type. Using round-toward-zero rounding mode.

See [\\_\\_bfloat162char\\_rz\(\\_\\_nv\\_bfloat16\)](#) for further details

## `__host__ device__ operator unsigned char ()`

### Description

Conversion operator to `unsigned char` data type. Using round-toward-zero rounding mode.

See [\\_\\_bfloat162uchar\\_rz\(\\_\\_nv\\_bfloat16\)](#) for further details

## `__host__ device__ operator unsigned int ()`

### Description

Conversion operator to `unsigned int` data type. Using round-toward-zero rounding mode.

See [\\_\\_bfloat162uint\\_rz\(\\_\\_nv\\_bfloat16\)](#) for further details

## `__host__ device__ operator unsigned long ()`

### Description

Conversion operator to `unsigned long` data type. Using round-toward-zero rounding mode.

## `__host__ device__ operator unsigned long long ()`

### Description

Conversion operator to `unsigned long long` data type. Using round-toward-zero rounding mode.

See [\\_\\_bfloat162ull\\_rz\(\\_\\_nv\\_bfloat16\)](#) for further details

## `__host__ device__ operator unsigned short ()`

### Description

Conversion operator to `unsigned short` data type. Using round-toward-zero rounding mode.

See [\\_\\_bfloat162ushort\\_rz\(\\_\\_nv\\_bfloat16\)](#) for further details

`__host__ device __operator= (unsigned long long val)`

#### Description

Type cast from `unsigned long long` assignment operator, using default round-to-nearest-even rounding mode.

`__host__ device __operator= (long long val)`

#### Description

Type cast from `long long` assignment operator, using default round-to-nearest-even rounding mode.

`__host__ device __operator= (unsigned int val)`

#### Description

Type cast from `unsigned int` assignment operator, using default round-to-nearest-even rounding mode.

`__host__ device __operator= (int val)`

#### Description

Type cast from `int` assignment operator, using default round-to-nearest-even rounding mode.

`__host__ device __operator= (unsigned short val)`

#### Description

Type cast from `unsigned short` assignment operator, using default round-to-nearest-even rounding mode.

`__host__ device __operator= (short val)`

#### Description

Type cast from `short` assignment operator, using default round-to-nearest-even rounding mode.

`__host__ __device__ operator= (const double f)`

### Description

Type cast to `__nv_bfloat16` assignment operator from `double` input using default round-to-nearest-even rounding mode.

`__host__ __device__ operator= (const float f)`

### Description

Type cast to `__nv_bfloat16` assignment operator from `float` input using default round-to-nearest-even rounding mode.

`__host__ __device__ operator= (const  
__nv_bfloat16_raw hr)`

### Description

Assignment operator from volatile `__nv_bfloat16_raw` to volatile `__nv_bfloat16` .

`__host__ __device__ operator= (const  
__nv_bfloat16_raw hr)`

### Description

Assignment operator from `__nv_bfloat16_raw` to volatile `__nv_bfloat16` .

`__host__ __device__ operator= (const  
__nv_bfloat16_raw hr)`

### Description

Assignment operator from `__nv_bfloat16_raw` .

## 2.6. `__nv_bfloat162` Struct Reference

`nv_bfloat162` datatype

This structure implements the datatype for storing two `nv_bfloat16` floating-point numbers. The structure implements assignment, arithmetic and comparison operators, and type conversions.

- ▶ NOTE: `__nv_bfloat162` is visible to non-nvcc host compilers

### `struct __nv_bfloat16 __nv_bfloat162::x`

Storage field holding lower `__nv_bfloat16` part.

### `struct __nv_bfloat16 __nv_bfloat162::y`

Storage field holding upper `__nv_bfloat16` part.

### `__nv_bfloat162`

`__nv_bfloat162` `cppOperationVisibility: visibility=public` `__host__` `__device__`  
`cppOperationConst: const=const` `__nv_bfloat162_raw` `h2r`

Constructor from `__nv_bfloat162_raw`

### `__nv_bfloat162`

`__nv_bfloat162` `cppOperationVisibility: visibility=public` `__host__` `__device__`  
`cppOperationConst: const=const` `__nv_bfloat162` `src`

Copy constructor

### `__nv_bfloat162`

`__nv_bfloat162` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`__host__` `__device__` `cppOperationConst: const=const` `__nv_bfloat16` `a`  
`cppOperationConst: const=const` `__nv_bfloat16` `b`

Constructor from two `__nv_bfloat16` variables

### `__nv_bfloat162`

`__nv_bfloat162` `cppOperationVisibility: visibility=public` `__host__` `__device__`

Constructor by default.



```
__host__ __device__ operator __nv_bfloat162_raw ()
```

### Description

Conversion operator to `__nv_bfloat162_raw`

```
__host__ __device__ operator= (const
__nv_bfloat162_raw h2r)
```

### Description

Assignment operator from `__nv_bfloat162_raw`

```
__host__ __device__ operator= (const __nv_bfloat162
src)
```

### Description

Copy assignment operator

## 2.7. `__nv_bfloat162_raw` Struct Reference

`__nv_bfloat162_raw` data type

Type allows static initialization of `nv_bfloat162` until it becomes a builtin type.

- ▶ Note: this initialization is as a bit-field representation of `nv_bfloat162`, and not a conversion from `short2` to `nv_bfloat162`. Such representation will be deprecated in a future version of CUDA.
- ▶ Note: this is visible to non-nvcc compilers, including C-only compilations

## 2.8. `__nv_bfloat16_raw` Struct Reference

`__nv_bfloat16_raw` data type

Type allows static initialization of `nv_bfloat16` until it becomes a builtin type.

- ▶ Note: this initialization is as a bit-field representation of `nv_bfloat16`, and not a conversion from `short` to `nv_bfloat16`. Such representation will be deprecated in a future version of CUDA.
- ▶ Note: this is visible to non-nvcc compilers, including C-only compilations

## 2.9. `__nv_fp8_e4m3` Struct Reference

`__nv_fp8_e4m3` datatype

This structure implements the datatype for storing `fp8` floating-point numbers of `e4m3` kind: with 1 sign, 4 exponent, 1 implicit and 3 explicit mantissa bits. The encoding doesn't support Infinity. NaNs are limited to `0x7F` and `0xFF` values.

The structure implements converting constructors and operators.

### `__nv_fp8_storage_t __nv_fp8_e4m3::__x`

Storage variable contains the `fp8` floating-point data.

### `__nv_fp8_e4m3`

`__nv_fp8_e4m3` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`cppConstructorExplicit: explicit=explicit` `__host__` `__device__` `cppOperationConst: const=const` long long int val

Constructor from long long int data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

### `__nv_fp8_e4m3`

`__nv_fp8_e4m3` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`cppConstructorExplicit: explicit=explicit` `__host__` `__device__` `cppOperationConst: const=const` `cppOperationPrimitive: storage=longint` val

Constructor from long int data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

### `__nv_fp8_e4m3`

`__nv_fp8_e4m3` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`cppConstructorExplicit: explicit=explicit` `__host__` `__device__` `cppOperationConst: const=const` `cppOperationPrimitive: storage=int` val

Constructor from int data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

### `__nv_fp8_e4m3`

`__nv_fp8_e4m3` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`cppConstructorExplicit: explicit=explicit` `__host__` `__device__` `cppOperationConst: const=const` short int val

Constructor from `short int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

### `__nv_fp8_e4m3`

`__nv_fp8_e4m3` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`cppConstructorExplicit: explicit=explicit` `__host__` `__device__` `cppOperationConst: const=const`  
`unsigned long long int val`

Constructor from `unsigned long long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

### `__nv_fp8_e4m3`

`__nv_fp8_e4m3` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`cppConstructorExplicit: explicit=explicit` `__host__` `__device__` `cppOperationConst: const=const`  
`unsigned long int val`

Constructor from `unsigned long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

### `__nv_fp8_e4m3`

`__nv_fp8_e4m3` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`cppConstructorExplicit: explicit=explicit` `__host__` `__device__` `cppOperationConst: const=const`  
`cppOperationPrimitive: storage=unsignedint` `val`

Constructor from `unsigned int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

### `__nv_fp8_e4m3`

`__nv_fp8_e4m3` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`cppConstructorExplicit: explicit=explicit` `__host__` `__device__` `cppOperationConst: const=const`  
`unsigned short int val`

Constructor from `unsigned short int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

### `__nv_fp8_e4m3`

`__nv_fp8_e4m3` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`cppConstructorExplicit: explicit=explicit` `__host__` `__device__` `cppOperationConst: const=const`  
`cppOperationPrimitive: storage=double` `f`

Constructor from `double` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

## `__nv_fp8_e4m3`

`__nv_fp8_e4m3` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`cppConstructorExplicit: explicit=explicit` `__host__` `__device__` `cppOperationConst: const=const`  
`cppOperationPrimitive: storage=float` `f`

Constructor from `float` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

## `__nv_fp8_e4m3`

`__nv_fp8_e4m3` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`cppConstructorExplicit: explicit=explicit` `__host__` `__device__` `cppOperationConst: const=const`  
`__nv_bfloat16` `f`

Constructor from `__nv_bfloat16` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

## `__nv_fp8_e4m3`

`__nv_fp8_e4m3` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`cppConstructorExplicit: explicit=explicit` `__host__` `__device__` `cppOperationConst: const=const`  
`__half` `f`

Constructor from `__half` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

## `__nv_fp8_e4m3`

`__nv_fp8_e4m3` `cppOperationVisibility: visibility=public`

Constructor by default.

## `__host__ __device__ operator __half ()`

### Description

Conversion operator to `__half` data type.

## `__host__ __device__ operator __nv_bfloat16 ()`

### Description

Conversion operator to `__nv_bfloat16` data type.

## `__host__ __device__ operator bool ()`

### Description

Conversion operator to `bool` data type. `+0` and `-0` inputs convert to `false`. Non-zero inputs convert to `true`.

## `__host__ __device__ operator char ()`

### Description

Conversion operator to an implementation defined `char` data type.

Detects signedness of the `char` type and proceeds accordingly, see further details in signed and unsigned char operators.

Clamps inputs to the output range. NaN inputs convert to zero.

## `__host__ __device__ operator double ()`

### Description

Conversion operator to `double` data type.

## `__host__ __device__ operator float ()`

### Description

Conversion operator to `float` data type.

## `__host__ __device__ operator int ()`

### Description

Conversion operator to `int` data type. NaN inputs convert to zero.

## `__host__ __device__ operator long int ()`

### Description

Conversion operator to `long int` data type. Clamps too large inputs to the output range. NaN inputs convert to zero if output type is 32-bit. NaN inputs convert to `0x8000000000000000ULL` if output type is 64-bit.

## `__host__ __device__ operator long long int ()`

### Description

Conversion operator to `long long int` data type. NaN inputs convert to `0x8000000000000000LL`.

## `__host__ __device__ operator short int ()`

### Description

Conversion operator to `short int` data type. NaN inputs convert to zero.

## `__host__ __device__ operator signed char ()`

### Description

Conversion operator to `signed char` data type. Clamps too large inputs to the output range. NaN inputs convert to zero.

## `__host__ __device__ operator unsigned char ()`

### Description

Conversion operator to `unsigned char` data type. Clamps negative and too large inputs to the output range. NaN inputs convert to zero.

## `__host__ __device__ operator unsigned int ()`

### Description

Conversion operator to `unsigned int` data type. Clamps negative inputs to zero. NaN inputs convert to zero.

## `__host__ __device__ operator unsigned long int ()`

### Description

Conversion operator to `unsigned long int` data type. Clamps negative and too large inputs to the output range. NaN inputs convert to zero if output type is 32-bit. NaN inputs convert to `0x8000000000000000ULL` if output type is 64-bit.

## `__host__ __device__ operator unsigned long long int ()`

### Description

Conversion operator to `unsigned long long int` data type. Clamps negative inputs to zero. NaN inputs convert to `0x8000000000000000ULL`.

## `__host__ __device__ operator unsigned short int ()`

### Description

Conversion operator to `unsigned short int` data type. Clamps negative inputs to zero. NaN inputs convert to zero.

## 2.10. `__nv_fp8_e5m2` Struct Reference

`__nv_fp8_e5m2` datatype

This structure implements the datatype for handling `fp8` floating-point numbers of `e5m2` kind: with 1 sign, 5 exponent, 1 implicit and 2 explicit mantissa bits.

The structure implements converting constructors and operators.

### `__nv_fp8_storage_t __nv_fp8_e5m2::__x`

Storage variable contains the `fp8` floating-point data.

### `__nv_fp8_e5m2`

`__nv_fp8_e5m2` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`cppConstructorExplicit: explicit=explicit` `__host__ __device__` `cppOperationConst: const=const` `long long int` `val`

Constructor from `long long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

### `__nv_fp8_e5m2`

`__nv_fp8_e5m2` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`cppConstructorExplicit: explicit=explicit` `__host__ __device__` `cppOperationConst: const=const` `cppOperationPrimitive: storage=longint` `val`

Constructor from `long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

## \_\_nv\_fp8\_e5m2

`__nv_fp8_e5m2` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`cppConstructorExplicit: explicit=explicit` `__host__` `__device__` `cppOperationConst: const=const`  
`cppOperationPrimitive: storage=int` `val`

Constructor from `int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

## \_\_nv\_fp8\_e5m2

`__nv_fp8_e5m2` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`cppConstructorExplicit: explicit=explicit` `__host__` `__device__` `cppOperationConst: const=const`  
`short int` `val`

Constructor from `short int` data type.

## \_\_nv\_fp8\_e5m2

`__nv_fp8_e5m2` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`cppConstructorExplicit: explicit=explicit` `__host__` `__device__` `cppOperationConst: const=const`  
`unsigned long long int` `val`

Constructor from `unsigned long long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

## \_\_nv\_fp8\_e5m2

`__nv_fp8_e5m2` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`cppConstructorExplicit: explicit=explicit` `__host__` `__device__` `cppOperationConst: const=const`  
`unsigned long int` `val`

Constructor from `unsigned long int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

## \_\_nv\_fp8\_e5m2

`__nv_fp8_e5m2` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`cppConstructorExplicit: explicit=explicit` `__host__` `__device__` `cppOperationConst: const=const`  
`cppOperationPrimitive: storage=unsignedint` `val`

Constructor from `unsigned int` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.



## \_\_nv\_fp8\_e5m2

`__nv_fp8_e5m2` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`cppConstructorExplicit: explicit=explicit` `__host__` `__device__` `cppOperationConst: const=const` unsigned short int val

Constructor from unsigned short int data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

## \_\_nv\_fp8\_e5m2

`__nv_fp8_e5m2` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`cppConstructorExplicit: explicit=explicit` `__host__` `__device__` `cppOperationConst: const=const` `cppOperationPrimitive: storage=double` f

Constructor from double data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

## \_\_nv\_fp8\_e5m2

`__nv_fp8_e5m2` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`cppConstructorExplicit: explicit=explicit` `__host__` `__device__` `cppOperationConst: const=const` `cppOperationPrimitive: storage=float` f

Constructor from float data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

## \_\_nv\_fp8\_e5m2

`__nv_fp8_e5m2` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`cppConstructorExplicit: explicit=explicit` `__host__` `__device__` `cppOperationConst: const=const` `__nv_bfloat16` f

Constructor from `__nv_bfloat16` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

## \_\_nv\_fp8\_e5m2

`__nv_fp8_e5m2` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`cppConstructorExplicit: explicit=explicit` `__host__` `__device__` `cppOperationConst: const=const` `__half` f

Constructor from `__half` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

## \_\_nv\_fp8\_e5m2

`__nv_fp8_e5m2` `cppOperationVisibility: visibility=public`

Constructor by default.

`__host__ device__ operator __half ()`

### Description

Conversion operator to `__half` data type.

`__host__ device__ operator __nv_bfloat16 ()`

### Description

Conversion operator to `__nv_bfloat16` data type.

`__host__ device__ operator bool ()`

### Description

Conversion operator to `bool` data type. +0 and -0 inputs convert to `false`. Non-zero inputs convert to `true`.

`__host__ device__ operator char ()`

### Description

Conversion operator to an implementation defined `char` data type.

Detects signedness of the `char` type and proceeds accordingly, see further details in signed and unsigned char operators.

Clamps inputs to the output range. NaN inputs convert to zero.

`__host__ device__ operator double ()`

### Description

Conversion operator to `double` data type.

`__host__ device__ operator float ()`

### Description

Conversion operator to `float` data type.

## `__host__ __device__ operator int ()`

### Description

Conversion operator to `int` data type. Clamps too large inputs to the output range. NaN inputs convert to `zero`.

## `__host__ __device__ operator long int ()`

### Description

Conversion operator to `long int` data type. Clamps too large inputs to the output range. NaN inputs convert to `zero` if output type is 32-bit. NaN inputs convert to `0x8000000000000000ULL` if output type is 64-bit.

## `__host__ __device__ operator long long int ()`

### Description

Conversion operator to `long long int` data type. Clamps too large inputs to the output range. NaN inputs convert to `0x8000000000000000LL`.

## `__host__ __device__ operator short int ()`

### Description

Conversion operator to `short int` data type. Clamps too large inputs to the output range. NaN inputs convert to `zero`.

## `__host__ __device__ operator signed char ()`

### Description

Conversion operator to `signed char` data type. Clamps too large inputs to the output range. NaN inputs convert to `zero`.

## `__host__ __device__ operator unsigned char ()`

### Description

Conversion operator to `unsigned char` data type. Clamps negative and too large inputs to the output range. NaN inputs convert to `zero`.

## `__host__ __device__ operator unsigned int ()`

### Description

Conversion operator to `unsigned int` data type. Clamps negative and too large inputs to the output range. NaN inputs convert to zero.

## `__host__ __device__ operator unsigned long int ()`

### Description

Conversion operator to `unsigned long int` data type. Clamps negative and too large inputs to the output range. NaN inputs convert to zero if output type is 32-bit. NaN inputs convert to `0x8000000000000000ULL` if output type is 64-bit.

## `__host__ __device__ operator unsigned long long int ()`

### Description

Conversion operator to `unsigned long long int` data type. Clamps negative and too large inputs to the output range. NaN inputs convert to `0x8000000000000000ULL`.

## `__host__ __device__ operator unsigned short int ()`

### Description

Conversion operator to `unsigned short int` data type. Clamps negative and too large inputs to the output range. NaN inputs convert to zero.

## 2.11. `__nv_fp8x2_e4m3` Struct Reference

### `__nv_fp8x2_e4m3` datatype

This structure implements the datatype for storage and operations on the vector of two `fp8` values of `e4m3` kind each: with 1 sign, 4 exponent, 1 implicit and 3 explicit mantissa bits. The encoding doesn't support Infinity. NaNs are limited to `0x7F` and `0xFF` values.

## `__nv_fp8x2_storage_t __nv_fp8x2_e4m3::__x`

Storage variable contains the vector of two `fp8` floating-point data values.

## \_\_nv\_fp8x2\_e4m3

`__nv_fp8x2_e4m3` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`cppConstructorExplicit: explicit=explicit` `__host__` `__device__` `cppOperationConst: const=const` `double2` `f`

Constructor from `double2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

## \_\_nv\_fp8x2\_e4m3

`__nv_fp8x2_e4m3` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`cppConstructorExplicit: explicit=explicit` `__host__` `__device__` `cppOperationConst: const=const` `float2` `f`

Constructor from `float2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

## \_\_nv\_fp8x2\_e4m3

`__nv_fp8x2_e4m3` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`cppConstructorExplicit: explicit=explicit` `__host__` `__device__` `cppOperationConst: const=const` `__nv_bfloat162` `f`

Constructor from `__nv_bfloat162` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

## \_\_nv\_fp8x2\_e4m3

`__nv_fp8x2_e4m3` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`cppConstructorExplicit: explicit=explicit` `__host__` `__device__` `cppOperationConst: const=const` `__half2` `f`

Constructor from `__half2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

## \_\_nv\_fp8x2\_e4m3

`__nv_fp8x2_e4m3` `cppOperationVisibility: visibility=public`

Constructor by default.

## \_\_host\_\_ \_\_device\_\_ operator \_\_half2 ()

### Description

Conversion operator to `__half2` data type.

## `__host__ __device__ operator float2 ()`

### Description

Conversion operator to `float2` data type.

## 2.12. `__nv_fp8x2_e5m2` Struct Reference

`__nv_fp8x2_e5m2` datatype

This structure implements the datatype for handling two `fp8` floating-point numbers of `e5m2` kind each: with 1 sign, 5 exponent, 1 implicit and 2 explicit mantissa bits.

The structure implements converting constructors and operators.

## `__nv_fp8x2_storage_t __nv_fp8x2_e5m2::__x`

Storage variable contains the vector of two `fp8` floating-point data values.

## `__nv_fp8x2_e5m2`

`__nv_fp8x2_e5m2` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`cppConstructorExplicit: explicit=explicit` `__host__ __device__` `cppOperationConst: const=const` `double2 f`

Constructor from `double2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

## `__nv_fp8x2_e5m2`

`__nv_fp8x2_e5m2` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`cppConstructorExplicit: explicit=explicit` `__host__ __device__` `cppOperationConst: const=const` `float2 f`

Constructor from `float2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

## `__nv_fp8x2_e5m2`

`__nv_fp8x2_e5m2` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`cppConstructorExplicit: explicit=explicit` `__host__ __device__` `cppOperationConst: const=const` `__nv_bfloat162 f`

Constructor from `__nv_bfloat162` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

## \_\_nv\_fp8x2\_e5m2

`__nv_fp8x2_e5m2` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`cppConstructorExplicit: explicit=explicit` `__host__` `__device__` `cppOperationConst: const=const` `__half2` `f`

Constructor from `__half2` data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

## \_\_nv\_fp8x2\_e5m2

`__nv_fp8x2_e5m2` `cppOperationVisibility: visibility=public`

Constructor by default.

## \_\_host\_\_ \_\_device\_\_ operator \_\_half2 ()

### Description

Conversion operator to `__half2` data type.

## \_\_host\_\_ \_\_device\_\_ operator float2 ()

### Description

Conversion operator to `float2` data type.

## 2.13. \_\_nv\_fp8x4\_e4m3 Struct Reference

`__nv_fp8x4_e4m3` datatype

This structure implements the datatype for storage and operations on the vector of four `fp8` values of `e4m3` kind each: with 1 sign, 4 exponent, 1 implicit and 3 explicit mantissa bits. The encoding doesn't support Infinity. NaNs are limited to `0x7F` and `0xFF` values.

## \_\_nv\_fp8x4\_storage\_t \_\_nv\_fp8x4\_e4m3::\_\_x

Storage variable contains the vector of four `fp8` floating-point data values.

## \_\_nv\_fp8x4\_e4m3

`__nv_fp8x4_e4m3` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`cppConstructorExplicit: explicit=explicit` `__host__` `__device__` `cppOperationConst: const=const` `double4` `f`

Constructor from `double4` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

## `__nv_fp8x4_e4m3`

`__nv_fp8x4_e4m3` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`cppConstructorExplicit: explicit=explicit` `__host__` `__device__` `cppOperationConst: const=const` `float4` `f`

Constructor from `float4` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

## `__nv_fp8x4_e4m3`

`__nv_fp8x4_e4m3` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`cppConstructorExplicit: explicit=explicit` `__host__` `__device__` `cppOperationConst: const=const` `__nv_bfloat162` `flo` `cppOperationConst: const=const` `__nv_bfloat162` `fhi`

Constructor from a pair of `__nv_bfloat162` data type values, relies on `__NV_SATFINITE` behavior for out-of-range values.

## `__nv_fp8x4_e4m3`

`__nv_fp8x4_e4m3` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`cppConstructorExplicit: explicit=explicit` `__host__` `__device__` `cppOperationConst: const=const` `__half2` `flo` `cppOperationConst: const=const` `__half2` `fhi`

Constructor from a pair of `__half2` data type values, relies on `__NV_SATFINITE` behavior for out-of-range values.

## `__nv_fp8x4_e4m3`

`__nv_fp8x4_e4m3` `cppOperationVisibility: visibility=public`

Constructor by default.

## `__host__ __device__ operator float4 ()`

### Description

Conversion operator to `float4` vector data type.

## 2.14. `__nv_fp8x4_e5m2` Struct Reference

`__nv_fp8x4_e5m2` datatype

This structure implements the datatype for handling four `fp8` floating-point numbers of `e5m2` kind each: with 1 sign, 5 exponent, 1 implicit and 2 explicit mantissa bits.



The structure implements converting constructors and operators.

## `__nv_fp8x4_storage_t __nv_fp8x4_e5m2::__x`

Storage variable contains the vector of four `fp8` floating-point data values.

## `__nv_fp8x4_e5m2`

`__nv_fp8x4_e5m2` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`cppConstructorExplicit: explicit=explicit` `__host__` `__device__` `cppOperationConst: const=const` `double4` `f`

Constructor from `double4` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

## `__nv_fp8x4_e5m2`

`__nv_fp8x4_e5m2` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`cppConstructorExplicit: explicit=explicit` `__host__` `__device__` `cppOperationConst: const=const` `float4` `f`

Constructor from `float4` vector data type, relies on `__NV_SATFINITE` behavior for out-of-range values.

## `__nv_fp8x4_e5m2`

`__nv_fp8x4_e5m2` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`cppConstructorExplicit: explicit=explicit` `__host__` `__device__` `cppOperationConst: const=const` `__nv_bfloat162` `flo` `cppOperationConst: const=const` `__nv_bfloat162` `fhi`

Constructor from a pair of `__nv_bfloat162` data type values, relies on `__NV_SATFINITE` behavior for out-of-range values.

## `__nv_fp8x4_e5m2`

`__nv_fp8x4_e5m2` `cppOperationVisibility: visibility=public` `cppOperationInline: inline=inline`  
`cppConstructorExplicit: explicit=explicit` `__host__` `__device__` `cppOperationConst: const=const` `__half2` `flo` `cppOperationConst: const=const` `__half2` `fhi`

Constructor from a pair of `__half2` data type values, relies on `__NV_SATFINITE` behavior for out-of-range values.

## `__nv_fp8x4_e5m2`

`__nv_fp8x4_e5m2` `cppOperationVisibility: visibility=public`

Constructor by default.

`__host__ __device__ operator float4 ()`

### Description

Conversion operator to `float4` vector data type.

---

## Chapter 3. Data Fields

Here is a list of all documented struct and union fields with links to the struct/union documentation for each field:

- \_\_half()**
  - [\\_\\_half](#)
- \_\_half2()**
  - [\\_\\_half2](#)
- \_\_nv\_bfloat16()**
  - [\\_\\_nv\\_bfloat16](#)
- \_\_nv\_bfloat162()**
  - [\\_\\_nv\\_bfloat162](#)
- \_\_nv\_fp8\_e4m3()**
  - [\\_\\_nv\\_fp8\\_e4m3](#)
- \_\_nv\_fp8\_e5m2()**
  - [\\_\\_nv\\_fp8\\_e5m2](#)
- \_\_nv\_fp8x2\_e4m3()**
  - [\\_\\_nv\\_fp8x2\\_e4m3](#)
- \_\_nv\_fp8x2\_e5m2()**
  - [\\_\\_nv\\_fp8x2\\_e5m2](#)
- \_\_nv\_fp8x4\_e4m3()**
  - [\\_\\_nv\\_fp8x4\\_e4m3](#)
- \_\_nv\_fp8x4\_e5m2()**
  - [\\_\\_nv\\_fp8x4\\_e5m2](#)
- \_\_x**
  - [\\_\\_nv\\_fp8x2\\_e4m3](#)
  - [\\_\\_half](#)
  - [\\_\\_nv\\_fp8x2\\_e5m2](#)
  - [\\_\\_nv\\_bfloat16](#)
  - [\\_\\_nv\\_fp8x4\\_e4m3](#)
  - [\\_\\_nv\\_fp8\\_e4m3](#)
  - [\\_\\_nv\\_fp8x4\\_e5m2](#)
  - [\\_\\_nv\\_fp8\\_e5m2](#)

## 0

**operator \_\_half()**nv\_fp8\_e5m2nv\_fp8\_e4m3**operator \_\_half2()**nv\_fp8x2\_e4m3nv\_fp8x2\_e5m2**operator \_\_half2\_raw()**half2**operator \_\_half\_raw()**half**operator \_\_nv\_bfloat16()**nv\_fp8\_e5m2nv\_fp8\_e4m3**operator \_\_nv\_bfloat162\_raw()**nv\_bfloat162**operator \_\_nv\_bfloat16\_raw()**nv\_bfloat16**operator bool()**nv\_bfloat16nv\_fp8\_e5m2nv\_fp8\_e4m3half**operator char()**nv\_fp8\_e5m2nv\_fp8\_e4m3halfnv\_bfloat16**operator double()**nv\_fp8\_e5m2nv\_fp8\_e4m3**operator float()**nv\_fp8\_e5m2nv\_fp8\_e4m3halfnv\_bfloat16**operator float2()**nv\_fp8x2\_e5m2nv\_fp8x2\_e4m3**operator float4()**nv\_fp8x4\_e5m2nv\_fp8x4\_e4m3

**operator int()**nv\_fp8\_e4m3halfnv\_bfloat16nv\_fp8\_e5m2**operator long()**halfnv\_bfloat16**operator long int()**nv\_fp8\_e5m2nv\_fp8\_e4m3**operator long long()**halfnv\_bfloat16**operator long long int()**nv\_fp8\_e5m2nv\_fp8\_e4m3**operator short()**halfnv\_bfloat16**operator short int()**nv\_fp8\_e5m2nv\_fp8\_e4m3**operator signed char()**nv\_fp8\_e5m2nv\_fp8\_e4m3halfnv\_bfloat16**operator unsigned char()**nv\_fp8\_e5m2nv\_fp8\_e4m3halfnv\_bfloat16**operator unsigned int()**nv\_fp8\_e5m2nv\_fp8\_e4m3halfnv\_bfloat16**operator unsigned long()**halfnv\_bfloat16**operator unsigned long int()**nv\_fp8\_e5m2

\_\_nv\_fp8\_e4m3**operator unsigned long long()**\_\_nv\_bfloat16\_\_half**operator unsigned long long int()**\_\_nv\_fp8\_e5m2\_\_nv\_fp8\_e4m3**operator unsigned short()**\_\_nv\_bfloat16\_\_half**operator unsigned short int()**\_\_nv\_fp8\_e4m3\_\_nv\_fp8\_e5m2**operator=()**\_\_half\_\_nv\_bfloat16\_\_half\_\_nv\_bfloat16\_\_half\_\_nv\_bfloat16\_\_half\_\_half2\_\_nv\_bfloat16\_\_nv\_bfloat162\_\_half\_\_nv\_bfloat162\_\_half\_\_half2\_\_nv\_bfloat16**X****x**\_\_half2\_\_nv\_bfloat162**Y****y**\_\_half2\_\_nv\_bfloat162

## 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.

## OpenCL

OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc.

## 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

© 2007-2023 NVIDIA Corporation & affiliates. All rights reserved.