



# Libdevice User's Guide

# Table of Contents

<b>Chapter 1. Introduction.....</b>	<b>1</b>
1.1. What Is libdevice?.....	1
<b>Chapter 2. Basic Usage.....</b>	<b>2</b>
2.1. Linking with libdevice.....	2
<b>Chapter 3. Function Reference.....</b>	<b>4</b>
3.1. __nv_abs.....	4
3.2. __nv_acos.....	4
3.3. __nv_acosf.....	5
3.4. __nv_acosh.....	6
3.5. __nv_acoshf.....	6
3.6. __nv_asin.....	7
3.7. __nv_asinf.....	7
3.8. __nv_asinh.....	8
3.9. __nv_asinhf.....	9
3.10. __nv_atan.....	9
3.11. __nv_atan2.....	10
3.12. __nv_atan2f.....	10
3.13. __nv_atanf.....	11
3.14. __nv_atanh.....	11
3.15. __nv_atanhf.....	12
3.16. __nv_brev.....	13
3.17. __nv_brevll.....	13
3.18. __nv_byte_perm.....	14
3.19. __nv_cbrt.....	14
3.20. __nv_cbrtf.....	15
3.21. __nv_ceil.....	16
3.22. __nv_ceilf.....	16
3.23. __nv_clz.....	17
3.24. __nv_clzll.....	17
3.25. __nv_copysign.....	18
3.26. __nv_copysignf.....	18
3.27. __nv_cos.....	18
3.28. __nv_cosf.....	19
3.29. __nv_cosh.....	20
3.30. __nv_coshf.....	20

3.31. __nv_cospi.....	21
3.32. __nv_cospif.....	21
3.33. __nv_dadd_rd.....	22
3.34. __nv_dadd_rn.....	22
3.35. __nv_dadd_ru.....	23
3.36. __nv_dadd_rz.....	24
3.37. __nv_ddiv_rd.....	24
3.38. __nv_ddiv_rn.....	25
3.39. __nv_ddiv_ru.....	25
3.40. __nv_ddiv_rz.....	26
3.41. __nv_dmul_rd.....	26
3.42. __nv_dmul_rn.....	27
3.43. __nv_dmul_ru.....	28
3.44. __nv_dmul_rz.....	28
3.45. __nv_double2float_rd.....	29
3.46. __nv_double2float_rn.....	29
3.47. __nv_double2float_ru.....	30
3.48. __nv_double2float_rz.....	30
3.49. __nv_double2hiint.....	31
3.50. __nv_double2int_rd.....	31
3.51. __nv_double2int_rn.....	32
3.52. __nv_double2int_ru.....	32
3.53. __nv_double2int_rz.....	33
3.54. __nv_double2ll_rd.....	33
3.55. __nv_double2ll_rn.....	34
3.56. __nv_double2ll_ru.....	34
3.57. __nv_double2ll_rz.....	35
3.58. __nv_double2loint.....	35
3.59. __nv_double2uint_rd.....	36
3.60. __nv_double2uint_rn.....	36
3.61. __nv_double2uint_ru.....	37
3.62. __nv_double2uint_rz.....	37
3.63. __nv_double2ull_rd.....	38
3.64. __nv_double2ull_rn.....	38
3.65. __nv_double2ull_ru.....	39
3.66. __nv_double2ull_rz.....	39
3.67. __nv_double_as_longlong.....	40
3.68. __nv_drcp_rd.....	40

3.69. __nv_drcp_rn.....	41
3.70. __nv_drcp_ru.....	41
3.71. __nv_drcp_rz.....	42
3.72. __nv_dsqrt_rd.....	42
3.73. __nv_dsqrt_rn.....	43
3.74. __nv_dsqrt_ru.....	43
3.75. __nv_dsqrt_rz.....	44
3.76. __nv_erf.....	45
3.77. __nv_erfc.....	45
3.78. __nv_erfcf.....	46
3.79. __nv_erfcinv.....	46
3.80. __nv_erfcinvf.....	47
3.81. __nv_erfcx.....	48
3.82. __nv_erfcxf.....	48
3.83. __nv_erff.....	49
3.84. __nv_erfinv.....	50
3.85. __nv_erfinvf.....	50
3.86. __nv_exp.....	51
3.87. __nv_exp10.....	51
3.88. __nv_exp10f.....	52
3.89. __nv_exp2.....	52
3.90. __nv_exp2f.....	53
3.91. __nv_expf.....	54
3.92. __nv_expm1.....	54
3.93. __nv_expm1f.....	55
3.94. __nv_fabs.....	55
3.95. __nv_fabsf.....	56
3.96. __nv_fadd_rd.....	56
3.97. __nv_fadd_rn.....	57
3.98. __nv_fadd_ru.....	58
3.99. __nv_fadd_rz.....	58
3.100. __nv_fast_cosf.....	59
3.101. __nv_fast_exp10f.....	59
3.102. __nv_fast_expf.....	60
3.103. __nv_fast_fdividef.....	60
3.104. __nv_fast_log10f.....	61
3.105. __nv_fast_log2f.....	62
3.106. __nv_fast_logf.....	62

3.107. __nv_fast_powf.....	63
3.108. __nv_fast_sincosf.....	63
3.109. __nv_fast_sinf.....	64
3.110. __nv_fast_tanf.....	65
3.111. __nv_fdim.....	65
3.112. __nv_fdimf.....	66
3.113. __nv_fdiv_rd.....	66
3.114. __nv_fdiv_rn.....	67
3.115. __nv_fdiv_ru.....	68
3.116. __nv_fdiv_rz.....	68
3.117. __nv_ffs.....	69
3.118. __nv_ffsll.....	69
3.119. __nv_finitef.....	70
3.120. __nv_float2half_rn.....	70
3.121. __nv_float2int_rd.....	71
3.122. __nv_float2int_rn.....	71
3.123. __nv_float2int_ru.....	72
3.124. __nv_float2int_rz.....	72
3.125. __nv_float2ll_rd.....	73
3.126. __nv_float2ll_rn.....	73
3.127. __nv_float2ll_ru.....	74
3.128. __nv_float2ll_rz.....	74
3.129. __nv_float2uint_rd.....	75
3.130. __nv_float2uint_rn.....	75
3.131. __nv_float2uint_ru.....	76
3.132. __nv_float2uint_rz.....	76
3.133. __nv_float2ull_rd.....	77
3.134. __nv_float2ull_rn.....	77
3.135. __nv_float2ull_ru.....	78
3.136. __nv_float2ull_rz.....	78
3.137. __nv_float_as_int.....	79
3.138. __nv_floor.....	79
3.139. __nv_floorf.....	80
3.140. __nv_fma.....	80
3.141. __nv_fma_rd.....	81
3.142. __nv_fma_rn.....	82
3.143. __nv_fma_ru.....	82
3.144. __nv_fma_rz.....	83

3.145. __nv_fmaf.....	84
3.146. __nv_fmaf_rd.....	84
3.147. __nv_fmaf_rn.....	85
3.148. __nv_fmaf_ru.....	86
3.149. __nv_fmaf_rz.....	87
3.150. __nv_fmax.....	87
3.151. __nv_fmaxf.....	88
3.152. __nv_fmin.....	89
3.153. __nv_fminf.....	89
3.154. __nv_fmod.....	90
3.155. __nv_fmodf.....	91
3.156. __nv_fmul_rd.....	91
3.157. __nv_fmul_rn.....	92
3.158. __nv_fmul_ru.....	92
3.159. __nv_fmul_rz.....	93
3.160. __nv_frcp_rd.....	94
3.161. __nv_frcp_rn.....	94
3.162. __nv_frcp_ru.....	95
3.163. __nv_frcp_rz.....	95
3.164. __nv_frexp.....	96
3.165. __nv_frexp.....	97
3.166. __nv_frsqrt_rn.....	97
3.167. __nv_fsqrt_rd.....	98
3.168. __nv_fsqrt_rn.....	98
3.169. __nv_fsqrt_ru.....	99
3.170. __nv_fsqrt_rz.....	100
3.171. __nv_fsub_rd.....	100
3.172. __nv_fsub_rn.....	101
3.173. __nv_fsub_ru.....	101
3.174. __nv_fsub_rz.....	102
3.175. __nv_hadd.....	102
3.176. __nv_half2float.....	103
3.177. __nv_hiloint2double.....	103
3.178. __nv_hypot.....	104
3.179. __nv_hypotf.....	104
3.180. __nv_ilogb.....	105
3.181. __nv_ilogbf.....	106
3.182. __nv_int2double_rn.....	106

3.183.	__nv_int2float_rd.....	107
3.184.	__nv_int2float_rn.....	107
3.185.	__nv_int2float_ru.....	108
3.186.	__nv_int2float_rz.....	108
3.187.	__nv_int_as_float.....	109
3.188.	__nv_isfinitd.....	109
3.189.	__nv_isinfd.....	110
3.190.	__nv_isinff.....	110
3.191.	__nv_isnand.....	110
3.192.	__nv_isnanf.....	111
3.193.	__nv_j0.....	111
3.194.	__nv_j0f.....	112
3.195.	__nv_j1.....	113
3.196.	__nv_j1f.....	113
3.197.	__nv_jn.....	114
3.198.	__nv_jnf.....	115
3.199.	__nv_ldexp.....	115
3.200.	__nv_ldexpf.....	116
3.201.	__nv_lgamma.....	116
3.202.	__nv_lgammaf.....	117
3.203.	__nv_ll2double_rd.....	118
3.204.	__nv_ll2double_rn.....	118
3.205.	__nv_ll2double_ru.....	119
3.206.	__nv_ll2double_rz.....	119
3.207.	__nv_ll2float_rd.....	120
3.208.	__nv_ll2float_rn.....	120
3.209.	__nv_ll2float_ru.....	121
3.210.	__nv_ll2float_rz.....	121
3.211.	__nv_llabs.....	122
3.212.	__nv_llmax.....	122
3.213.	__nv_llmin.....	122
3.214.	__nv_llrint.....	123
3.215.	__nv_llrintf.....	123
3.216.	__nv_llround.....	124
3.217.	__nv_llroundf.....	124
3.218.	__nv_log.....	125
3.219.	__nv_log10.....	126
3.220.	__nv_log10f.....	126

3.221. __nv_log1p.....	127
3.222. __nv_log1pf.....	128
3.223. __nv_log2.....	128
3.224. __nv_log2f.....	129
3.225. __nv_logb.....	130
3.226. __nv_logbf.....	130
3.227. __nv_logf.....	131
3.228. __nv_longlong_as_double.....	131
3.229. __nv_max.....	132
3.230. __nv_min.....	132
3.231. __nv_modf.....	133
3.232. __nv_modff.....	133
3.233. __nv_mul24.....	134
3.234. __nv_mul64hi.....	134
3.235. __nv_mulhi.....	135
3.236. __nv_nan.....	135
3.237. __nv_nanf.....	136
3.238. __nv_nearbyint.....	136
3.239. __nv_nearbyintf.....	137
3.240. __nv_nextafter.....	138
3.241. __nv_nextafterf.....	138
3.242. __nv_normcdf.....	139
3.243. __nv_normcdf.....	139
3.244. __nv_normcdfinv.....	140
3.245. __nv_normcdfinvf.....	141
3.246. __nv_popc.....	141
3.247. __nv_popctl.....	142
3.248. __nv_pow.....	142
3.249. __nv_powf.....	143
3.250. __nv_powi.....	144
3.251. __nv_powif.....	145
3.252. __nv_rcbrt.....	147
3.253. __nv_rcbrtf.....	147
3.254. __nv_remainder.....	148
3.255. __nv_remainderf.....	148
3.256. __nv_remquo.....	149
3.257. __nv_remquof.....	150
3.258. __nv_rhadd.....	151



3.259. __nv_rint.....	151
3.260. __nv_rintf.....	152
3.261. __nv_round.....	152
3.262. __nv_roundf.....	153
3.263. __nv_rsqrt.....	153
3.264. __nv_rsqrtf.....	154
3.265. __nv_sad.....	155
3.266. __nv_saturatef.....	155
3.267. __nv_scalbn.....	156
3.268. __nv_scalbnf.....	156
3.269. __nv_signbitd.....	157
3.270. __nv_signbitf.....	157
3.271. __nv_sin.....	158
3.272. __nv_sincos.....	158
3.273. __nv_sincosf.....	159
3.274. __nv_sincospi.....	159
3.275. __nv_sincospif.....	160
3.276. __nv_sinf.....	161
3.277. __nv_sinh.....	161
3.278. __nv_sinhf.....	162
3.279. __nv_sinpi.....	162
3.280. __nv_sinpif.....	163
3.281. __nv_sqrt.....	163
3.282. __nv_sqrtf.....	164
3.283. __nv_tan.....	165
3.284. __nv_tanf.....	165
3.285. __nv_tanh.....	166
3.286. __nv_tanhf.....	166
3.287. __nv_tgamma.....	167
3.288. __nv_tgammaf.....	168
3.289. __nv_trunc.....	168
3.290. __nv_truncf.....	169
3.291. __nv_uhadd.....	169
3.292. __nv_uint2double_rn.....	170
3.293. __nv_uint2float_rd.....	170
3.294. __nv_uint2float_rn.....	171
3.295. __nv_uint2float_ru.....	171
3.296. __nv_uint2float_rz.....	172

3.297. __nv_ull2double_rd.....	172
3.298. __nv_ull2double_rn.....	173
3.299. __nv_ull2double_ru.....	173
3.300. __nv_ull2double_rz.....	174
3.301. __nv_ull2float_rd.....	174
3.302. __nv_ull2float_rn.....	175
3.303. __nv_ull2float_ru.....	175
3.304. __nv_ull2float_rz.....	176
3.305. __nv_ullmax.....	176
3.306. __nv_ullmin.....	177
3.307. __nv_umax.....	177
3.308. __nv_umin.....	177
3.309. __nv_umul24.....	178
3.310. __nv_umul64hi.....	178
3.311. __nv_umulhi.....	179
3.312. __nv_urhadd.....	179
3.313. __nv_usad.....	180
3.314. __nv_y0.....	180
3.315. __nv_y0f.....	181
3.316. __nv_y1.....	182
3.317. __nv_y1f.....	182
3.318. __nv_yn.....	183
3.319. __nv_ynf.....	184

# List of Tables

Table 1. Supported Reflection Parameters .....2



---

# Chapter 1. Introduction

## 1.1. What Is libdevice?

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

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

---

# Chapter 2. Basic Usage

## 2.1. Linking with libdevice

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

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

Table 1. Supported Reflection Parameters

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

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

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

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

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

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

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

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

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

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

---

# Chapter 3. Function Reference

This chapter describes all functions available in libdevice.

## 3.1. `__nv_abs`

### Prototype:

```
i32 @__nv_abs(i32 %x)
```

### Description:

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

### Returns:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.2. `__nv_acos`

### Prototype:

```
double @__nv_acos(double %x)
```

### Description:

Calculate the principal value of the arc cosine of the input argument  $x$ .

### Returns:

Result will be in radians, in the interval  $[0, \pi]$  for  $x$  inside  $[-1, +1]$ .



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

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.3. `__nv_acosf`

**Prototype:**

```
float @__nv_acosf(float %x)
```

**Description:**

Calculate the principal value of the arc cosine of the input argument  $x$ .

**Returns:**

Result will be in radians, in the interval  $[0, \pi]$  for  $x$  inside  $[-1, +1]$ .

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

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.4. `__nv_acosh`

### Prototype:

```
double @__nv_acosh(double %x)
```

### Description:

Calculate the nonnegative arc hyperbolic cosine of the input argument  $x$ .

### Returns:

Result will be in the interval  $[0, +\infty]$ .

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



### Note:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.5. `__nv_acoshf`

### Prototype:

```
float @__nv_acoshf(float %x)
```

### Description:

Calculate the nonnegative arc hyperbolic cosine of the input argument  $x$ .

### Returns:

Result will be in the interval  $[0, +\infty]$ .

- ▶ `__nv_acoshf(1)` returns 0.

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

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.6. `__nv_asin`

**Prototype:**

```
double @__nv_asin(double %x)
```

**Description:**

Calculate the principal value of the arc sine of the input argument  $x$ .

**Returns:**

Result will be in radians, in the interval  $[-\pi/2, +\pi/2]$  for  $x$  inside  $[-1, +1]$ .

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

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.7. `__nv_asinf`

**Prototype:**

```
float @__nv_asinf(float %x)
```

**Description:**

Calculate the principal value of the arc sine of the input argument  $x$ .

**Returns:**

Result will be in radians, in the interval  $[-\pi/2, +\pi/2]$  for  $x$  inside  $[-1, +1]$ .

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

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.8. `__nv_asinh`

**Prototype:**

```
double @__nv_asinh(double %x)
```

**Description:**

Calculate the arc hyperbolic sine of the input argument  $x$ .

**Returns:**

- ▶ `__nv_asinh(0)` returns 1.

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.9. `__nv_asinhf`

### Prototype:

```
float @__nv_asinhf(float %x)
```

### Description:

Calculate the arc hyperbolic sine of the input argument  $x$ .

### Returns:

- `__nv_asinh(0)` returns 1.



#### Note:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.10. `__nv_atan`

### Prototype:

```
double @__nv_atan(double %x)
```

### Description:

Calculate the principal value of the arc tangent of the input argument  $x$ .

### Returns:

Result will be in radians, in the interval  $[-\pi/2, +\pi/2]$ .

- `__nv_atan(0)` returns +0.



#### Note:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.11. `__nv_atan2`

### Prototype:

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

### Description:

Calculate the principal value of the arc tangent of the ratio of first and second input arguments  $x / y$ . The quadrant of the result is determined by the signs of inputs  $x$  and  $y$ .

### Returns:

Result will be in radians, in the interval  $[-\pi /, +\pi ]$ .

- `__nv_atan2(0, 1)` returns +0.



#### Note:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.12. `__nv_atan2f`

### Prototype:

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

### Description:

Calculate the principal value of the arc tangent of the ratio of first and second input arguments  $x / y$ . The quadrant of the result is determined by the signs of inputs  $x$  and  $y$ .

### Returns:

Result will be in radians, in the interval  $[-\pi /, +\pi ]$ .

- ▶ `__nv_atan2f(0, 1)` returns +0.

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.13. `__nv_atanf`

**Prototype:**

```
float @__nv_atanf(float %x)
```

**Description:**

Calculate the principal value of the arc tangent of the input argument  $x$ .

**Returns:**

Result will be in radians, in the interval  $[-\pi/2, +\pi/2]$ .

- ▶ `__nv_atan(0)` returns +0.

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.14. `__nv_atanh`

**Prototype:**

```
double @__nv_atanh(double %x)
```

**Description:**

Calculate the arc hyperbolic tangent of the input argument  $x$ .

**Returns:**

- ▶ `__nv_atanh( ±0 )` returns  $±0$ .
- ▶ `__nv_atanh( ±1 )` returns  $±∞$ .
- ▶ `__nv_atanh(x)` returns NaN for  $x$  outside interval  $[-1, 1]$ .

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.15. `__nv_atanhf`

**Prototype:**

```
float @__nv_atanhf(float %x)
```

**Description:**

Calculate the arc hyperbolic tangent of the input argument  $x$ .

**Returns:**

- ▶ `__nv_atanhf( ±0 )` returns  $±0$ .
- ▶ `__nv_atanhf( ±1 )` returns  $±∞$ .
- ▶ `__nv_atanhf(x)` returns NaN for  $x$  outside interval  $[-1, 1]$ .

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes



Compute 3.5: Yes

## 3.16. `__nv_brev`

### Prototype:

```
i32 @__nv_brev(i32 %x)
```

### Description:

Reverses the bit order of the 32 bit unsigned integer `x`.

### Returns:

Returns the bit-reversed value of `x`. i.e. bit `N` of the return value corresponds to bit `31-N` of `x`.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.17. `__nv_brevll`

### Prototype:

```
i64 @__nv_brevll(i64 %x)
```

### Description:

Reverses the bit order of the 64 bit unsigned integer `x`.

### Returns:

Returns the bit-reversed value of `x`. i.e. bit `N` of the return value corresponds to bit `63-N` of `x`.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.18. `__nv_byte_perm`

### Prototype:

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

### Description:

`__nv_byte_perm(x,y,s)` returns a 32-bit integer consisting of four bytes from eight input bytes provided in the two input integers `x` and `y`, as specified by a selector, `s`.

The input bytes are indexed as follows:

```
input[0] = x<7:0>   input[1] = x<15:8>
input[2] = x<23:16> input[3] = x<31:24>
input[4] = y<7:0>   input[5] = y<15:8>
input[6] = y<23:16> input[7] = y<31:24>
```

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

```
selector[0] = s<2:0>  selector[1] = s<6:4>
selector[2] = s<10:8> selector[3] = s<14:12>
```

### Returns:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.19. `__nv_cbrt`

### Prototype:

```
double @__nv_cbrt(double %x)
```

### Description:

Calculate the cube root of `x`,  $x^{1/3}$ .

**Returns:**

Returns  $x^{1/3}$ .

- ▶ `__nv_cbrt( ±0 )` returns  $±0$ .
- ▶ `__nv_cbrt( ±∞ )` returns  $±∞$ .

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.20. `__nv_cbrtf`

**Prototype:**

```
float @__nv_cbrtf(float %x)
```

**Description:**

Calculate the cube root of  $x$ ,  $x^{1/3}$ .

**Returns:**

Returns  $x^{1/3}$ .

- ▶ `__nv_cbrtf( ±0 )` returns  $±0$ .
- ▶ `__nv_cbrtf( ±∞ )` returns  $±∞$ .

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.21. `__nv_ceil`

### Prototype:

```
double @__nv_ceil(double %x)
```

### Description:

Compute the smallest integer value not less than  $x$ .

### Returns:

Returns  $[x]$  expressed as a floating-point number.

- ▶ `__nv_ceil( ±0 )` returns  $±0$ .
- ▶ `__nv_ceil( ±∞ )` returns  $±∞$ .

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.22. `__nv_ceilf`

### Prototype:

```
float @__nv_ceilf(float %x)
```

### Description:

Compute the smallest integer value not less than  $x$ .

### Returns:

Returns  $[x]$  expressed as a floating-point number.

- ▶ `__nv_ceilf( ±0 )` returns  $±0$ .
- ▶ `__nv_ceilf( ±∞ )` returns  $±∞$ .

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.23. `__nv_clz`

### Prototype:

```
i32 @__nv_clz(i32 %x)
```

### Description:

Count the number of consecutive leading zero bits, starting at the most significant bit (bit 31) of `x`.

### Returns:

Returns a value between 0 and 32 inclusive representing the number of zero bits.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.24. `__nv_clzll`

### Prototype:

```
i32 @__nv_clzll(i64 %x)
```

### Description:

Count the number of consecutive leading zero bits, starting at the most significant bit (bit 63) of `x`.

### Returns:

Returns a value between 0 and 64 inclusive representing the number of zero bits.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.25. `__nv_copysign`

### Prototype:

```
double __nv_copysign(double %x, double %y)
```

### Description:

Create a floating-point value with the magnitude `x` and the sign of `y`.

### Returns:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.26. `__nv_copysignf`

### Prototype:

```
float __nv_copysignf(float %x, float %y)
```

### Description:

Create a floating-point value with the magnitude `x` and the sign of `y`.

### Returns:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.27. `__nv_cos`

### Prototype:

```
double @__nv_cos(double %x)
```

**Description:**

Calculate the cosine of the input argument  $x$  (measured in radians).

**Returns:**

- ▶ `__nv_cos( ±0 )` returns 1.
- ▶ `__nv_cos( ±∞ )` returns NaN.

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.28. `__nv_cosf`

**Prototype:**

```
float @__nv_cosf(float %x)
```

**Description:**

Calculate the cosine of the input argument  $x$  (measured in radians).

**Returns:**

- ▶ `__nv_cosf( ±0 )` returns 1.
- ▶ `__nv_cosf( ±∞ )` returns NaN.

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.29. `__nv_cosh`

### Prototype:

```
double @__nv_cosh(double %x)
```

### Description:

Calculate the hyperbolic cosine of the input argument  $x$ .

### Returns:

- ▶ `__nv_cosh(0)` returns 1.
- ▶ `__nv_cosh( ±∞ )` returns  $+∞$ .



#### Note:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.30. `__nv_coshf`

### Prototype:

```
float @__nv_coshf(float %x)
```

### Description:

Calculate the hyperbolic cosine of the input argument  $x$ .

### Returns:

- ▶ `__nv_coshf(0)` returns 1.
- ▶ `__nv_coshf( ±∞ )` returns  $+∞$ .



#### Note:



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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.31. `__nv_cospi`

**Prototype:**

```
double @__nv_cospi(double %x)
```

**Description:**

Calculate the cosine of  $x \times \pi$  (measured in radians), where  $x$  is the input argument.

**Returns:**

- ▶ `__nv_cospi( ±0 )` returns 1.
- ▶ `__nv_cospi( ±∞ )` returns NaN.

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.32. `__nv_cospif`

**Prototype:**

```
float @__nv_cospif(float %x)
```

**Description:**

Calculate the cosine of  $x \times \pi$  (measured in radians), where  $x$  is the input argument.

**Returns:**

- ▶ `__nv_cospif( ±0 )` returns 1.
- ▶ `__nv_cospif( ±∞ )` returns NaN.

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.33. `__nv_dadd_rd`

**Prototype:**

```
double @__nv_dadd_rd(double %x, double %y)
```

**Description:**

Adds two floating point values `x` and `y` in round-down (to negative infinity) mode.

**Returns:**

Returns `x + y`.

**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.

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.34. `__nv_dadd_rn`

**Prototype:**

```
double @__nv_dadd_rn(double %x, double %y)
```

**Description:**

Adds two floating point values  $x$  and  $y$  in round-to-nearest-even mode.

**Returns:**

Returns  $x + y$ .

**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.

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.35. `__nv_dadd_ru`

**Prototype:**

```
double @__nv_dadd_ru(double %x, double %y)
```

**Description:**

Adds two floating point values  $x$  and  $y$  in round-up (to positive infinity) mode.

**Returns:**

Returns  $x + y$ .

**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.

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.36. `__nv_dadd_rz`

### Prototype:

```
double @__nv_dadd_rz(double %x, double %y)
```

### Description:

Adds two floating point values  $x$  and  $y$  in round-towards-zero mode.

### Returns:

Returns  $x + y$ .



#### 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.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.37. `__nv_ddiv_rd`

### Prototype:

```
double @__nv_ddiv_rd(double %x, double %y)
```

### Description:

Divides two floating point values  $x$  by  $y$  in round-down (to negative infinity) mode.

### Returns:

Returns  $x / y$ .



#### Note:

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

Requires compute capability  $\geq 2.0$ .

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.38. `__nv_ddiv_rn`

**Prototype:**

```
double @__nv_ddiv_rn(double %x, double %y)
```

**Description:**

Divides two floating point values  $x$  by  $y$  in round-to-nearest-even mode.

**Returns:**

Returns  $x / y$ .

**Note:**

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

Requires compute capability  $\geq 2.0$ .

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.39. `__nv_ddiv_ru`

**Prototype:**

```
double @__nv_ddiv_ru(double %x, double %y)
```

**Description:**

Divides two floating point values  $x$  by  $y$  in round-up (to positive infinity) mode.

**Returns:**

Returns  $x / y$ .

**Note:**

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

Requires compute capability  $\geq 2.0$ .

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.40. `__nv_ddiv_rz`

**Prototype:**

```
double @__nv_ddiv_rz(double %x, double %y)
```

**Description:**

Divides two floating point values  $x$  by  $y$  in round-towards-zero mode.

**Returns:**

Returns  $x / y$ .

**Note:**

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

Requires compute capability  $\geq 2.0$ .

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.41. `__nv_dmul_rd`

**Prototype:**

```
double @__nv_dmul_rd(double %x, double %y)
```

**Description:**

Multiplies two floating point values  $x$  and  $y$  in round-down (to negative infinity) mode.

**Returns:**

Returns  $x * y$ .

**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.

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.42. `__nv_dmul_rn`

**Prototype:**

```
double @__nv_dmul_rn(double %x, double %y)
```

**Description:**

Multiplies two floating point values  $x$  and  $y$  in round-to-nearest-even mode.

**Returns:**

Returns  $x * y$ .

**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.

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.43. `__nv_dmul_ru`

### Prototype:

```
double @__nv_dmul_ru(double %x, double %y)
```

### Description:

Multiplies two floating point values  $x$  and  $y$  in round-up (to positive infinity) mode.

### Returns:

Returns  $x * y$ .



#### 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.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.44. `__nv_dmul_rz`

### Prototype:

```
double @__nv_dmul_rz(double %x, double %y)
```

### Description:

Multiplies two floating point values  $x$  and  $y$  in round-towards-zero mode.

### Returns:

Returns  $x * y$ .



#### 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.



**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.45. `__nv_double2float_rd`

**Prototype:**

```
float __nv_double2float_rd(double x)
```

**Description:**

Convert the double-precision floating point value  $x$  to a single-precision floating point value in round-down (to negative infinity) mode.

**Returns:**

Returns converted value.

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.46. `__nv_double2float_rn`

**Prototype:**

```
float __nv_double2float_rn(double x)
```

**Description:**

Convert the double-precision floating point value  $x$  to a single-precision floating point value in round-to-nearest-even mode.

**Returns:**

Returns converted value.

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.47. `__nv_double2float_ru`

### Prototype:

```
float __nv_double2float_ru(double x)
```

### Description:

Convert the double-precision floating point value `x` to a single-precision floating point value in round-up (to positive infinity) mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.48. `__nv_double2float_rz`

### Prototype:

```
float __nv_double2float_rz(double x)
```

### Description:

Convert the double-precision floating point value `x` to a single-precision floating point value in round-towards-zero mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.49. `__nv_double2hiint`

### Prototype:

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

### Description:

Reinterpret the high 32 bits in the double-precision floating point value  $x$  as a signed integer.

### Returns:

Returns reinterpreted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.50. `__nv_double2int_rd`

### Prototype:

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

### Description:

Convert the double-precision floating point value  $x$  to a signed integer value in round-down (to negative infinity) mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.51. `__nv_double2int_rn`

### Prototype:

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

### Description:

Convert the double-precision floating point value  $x$  to a signed integer value in round-to-nearest-even mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.52. `__nv_double2int_ru`

### Prototype:

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

### Description:

Convert the double-precision floating point value  $x$  to a signed integer value in round-up (to positive infinity) mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.53. `__nv_double2int_rz`

### Prototype:

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

### Description:

Convert the double-precision floating point value  $x$  to a signed integer value in round-towards-zero mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.54. `__nv_double2ll_rd`

### Prototype:

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

### Description:

Convert the double-precision floating point value  $x$  to a signed 64-bit integer value in round-down (to negative infinity) mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.55. `__nv_double2ll_rn`

### Prototype:

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

### Description:

Convert the double-precision floating point value  $x$  to a signed 64-bit integer value in round-to-nearest-even mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.56. `__nv_double2ll_ru`

### Prototype:

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

### Description:

Convert the double-precision floating point value  $x$  to a signed 64-bit integer value in round-up (to positive infinity) mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.57. `__nv_double2ll_rz`

### Prototype:

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

### Description:

Convert the double-precision floating point value `x` to a signed 64-bit integer value in round-towards-zero mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.58. `__nv_double2loint`

### Prototype:

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

### Description:

Reinterpret the low 32 bits in the double-precision floating point value `x` as a signed integer.

### Returns:

Returns reinterpreted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.59. `__nv_double2uint_rd`

### Prototype:

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

### Description:

Convert the double-precision floating point value  $x$  to an unsigned integer value in round-down (to negative infinity) mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.60. `__nv_double2uint_rn`

### Prototype:

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

### Description:

Convert the double-precision floating point value  $x$  to an unsigned integer value in round-to-nearest-even mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes



## 3.61. `__nv_double2uint_ru`

### Prototype:

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

### Description:

Convert the double-precision floating point value  $x$  to an unsigned integer value in round-up (to positive infinity) mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.62. `__nv_double2uint_rz`

### Prototype:

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

### Description:

Convert the double-precision floating point value  $x$  to an unsigned integer value in round-towards-zero mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.63. `__nv_double2ull_rd`

### Prototype:

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

### Description:

Convert the double-precision floating point value  $x$  to an unsigned 64-bit integer value in round-down (to negative infinity) mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.64. `__nv_double2ull_rn`

### Prototype:

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

### Description:

Convert the double-precision floating point value  $x$  to an unsigned 64-bit integer value in round-to-nearest-even mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.65. `__nv_double2ull_ru`

### Prototype:

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

### Description:

Convert the double-precision floating point value  $x$  to an unsigned 64-bit integer value in round-up (to positive infinity) mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.66. `__nv_double2ull_rz`

### Prototype:

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

### Description:

Convert the double-precision floating point value  $x$  to an unsigned 64-bit integer value in round-towards-zero mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.67. `__nv_double_as_longlong`

### Prototype:

```
i64 @__nv_double_as_longlong(double %x)
```

### Description:

Reinterpret the bits in the double-precision floating point value `x` as a signed 64-bit integer.

### Returns:

Returns reinterpreted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.68. `__nv_drcp_rd`

### Prototype:

```
double @__nv_drcp_rd(double %x)
```

### Description:

Compute the reciprocal of `x` in round-down (to negative infinity) mode.

### Returns:

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



#### Note:

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

Requires compute capability  $\geq 2.0$ .

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.69. `__nv_drcp_rn`

### Prototype:

```
double @__nv_drcp_rn(double %x)
```

### Description:

Compute the reciprocal of  $x$  in round-to-nearest-even mode.

### Returns:

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



#### Note:

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

Requires compute capability  $\geq 2.0$ .

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.70. `__nv_drcp_ru`

### Prototype:

```
double @__nv_drcp_ru(double %x)
```

### Description:

Compute the reciprocal of  $x$  in round-up (to positive infinity) mode.

### Returns:

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



#### Note:

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

Requires compute capability  $\geq 2.0$ .

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.71. `__nv_drcp_rz`

**Prototype:**

```
double @__nv_drcp_rz(double %x)
```

**Description:**

Compute the reciprocal of  $x$  in round-towards-zero mode.

**Returns:**

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

**Note:**

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

Requires compute capability  $\geq 2.0$ .

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.72. `__nv_dsqrt_rd`

**Prototype:**

```
double @__nv_dsqrt_rd(double %x)
```

**Description:**

Compute the square root of  $x$  in round-down (to negative infinity) mode.

**Returns:**

Returns  $\sqrt{x}$ .

**Note:**

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

Requires compute capability  $\geq 2.0$ .

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.73. `__nv_dsqrt_rn`

**Prototype:**

```
double @__nv_dsqrt_rn(double %x)
```

**Description:**

Compute the square root of  $x$  in round-to-nearest-even mode.

**Returns:**

Returns  $\sqrt{x}$ .

**Note:**

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

Requires compute capability  $\geq 2.0$ .

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.74. `__nv_dsqrt_ru`

**Prototype:**

```
double @__nv_dsqrt_ru(double %x)
```

**Description:**

Compute the square root of  $x$  in round-up (to positive infinity) mode.

**Returns:**

Returns  $\sqrt{x}$ .

**Note:**

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

Requires compute capability  $\geq 2.0$ .

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.75. `__nv_dsqrt_rz`

**Prototype:**

```
double @__nv_dsqrt_rz(double %x)
```

**Description:**

Compute the square root of  $x$  in round-towards-zero mode.

**Returns:**

Returns  $\sqrt{x}$ .

**Note:**

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

Requires compute capability  $\geq 2.0$ .

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes



## 3.76. `__nv_erf`

### Prototype:

```
double @__nv_erf(double %x)
```

### Description:

Calculate the value of the error function for the input argument  $x$ ,  $\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$ .

### Returns:

- ▶ `__nv_erf( ±0 )` returns  $\pm 0$ .
- ▶ `__nv_erf( ±∞ )` returns  $\pm 1$ .



#### Note:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.77. `__nv_erfc`

### Prototype:

```
double @__nv_erfc(double %x)
```

### Description:

Calculate the complementary error function of the input argument  $x$ ,  $1 - \text{erf}(x)$ .

### Returns:

- ▶ `__nv_erfc( -∞ )` returns 2.

- ▶ `__nv_erfc(+∞)` returns +0.

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.78. `__nv_erfcf`

**Prototype:**

```
float @__nv_erfcf(float %x)
```

**Description:**

Calculate the complementary error function of the input argument `x`,  $1 - \text{erf}(x)$ .

**Returns:**

- ▶ `__nv_erfcf(-∞)` returns 2.
- ▶ `__nv_erfcf(+∞)` returns +0.

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.79. `__nv_erfcinv`

**Prototype:**

```
double @__nv_erfcinv(double %x)
```

**Description:**

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

**Returns:**

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

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.80. `__nv_erfcinvf`

**Prototype:**

```
float @__nv_erfcinvf(float %x)
```

**Description:**

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

**Returns:**

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

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.81. `__nv_erfcx`

### Prototype:

```
double @__nv_erfcx(double %x)
```

### Description:

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

### Returns:

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



#### Note:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.82. `__nv_erfcxf`

### Prototype:

```
float @__nv_erfcxf(float %x)
```

### Description:

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

### Returns:

- ▶ `__nv_erfcxf(-∞)` returns  $+\infty$
- ▶ `__nv_erfcxf(+∞)` returns  $+0$

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

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.83. `__nv_erff`

**Prototype:**

```
float @__nv_erff(float %x)
```

**Description:**

Calculate the value of the error function for the input argument  $x$ ,  $\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$ .

**Returns:**

- ▶ `__nv_erff( ±0 )` returns  $\pm 0$ .
- ▶ `__nv_erff( ±∞ )` returns  $\pm 1$ .

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.84. `__nv_erfinv`

### Prototype:

```
double @__nv_erfinv(double %x)
```

### Description:

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

### Returns:

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



### Note:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.85. `__nv_erfinvf`

### Prototype:

```
float @__nv_erfinvf(float %x)
```

### Description:

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

### Returns:

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

- ▶ `__nv_erfinvf(-1)` returns  $-\infty$ .

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.86. `__nv_exp`

**Prototype:**

```
double @__nv_exp(double %x)
```

**Description:**

Calculate the base  $e$  exponential of the input argument  $x$ .

**Returns:**

Returns  $e^x$ .

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.87. `__nv_exp10`

**Prototype:**

```
double @__nv_exp10(double %x)
```

**Description:**

Calculate the base 10 exponential of the input argument  $x$ .

**Returns:**

Returns  $10^x$ .



**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.88. `__nv_exp10f`

**Prototype:**

```
float @__nv_exp10f(float %x)
```

**Description:**

Calculate the base 10 exponential of the input argument  $x$ .

**Returns:**

Returns  $10^x$ .



**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.89. `__nv_exp2`

**Prototype:**

```
double @__nv_exp2(double %x)
```



**Description:**

Calculate the base 2 exponential of the input argument  $x$ .

**Returns:**

Returns  $2^x$ .

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.90. `__nv_exp2f`

**Prototype:**

```
float @__nv_exp2f(float %x)
```

**Description:**

Calculate the base 2 exponential of the input argument  $x$ .

**Returns:**

Returns  $2^x$ .

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.91. `__nv_expf`

### Prototype:

```
float @__nv_expf(float %x)
```

### Description:

Calculate the base  $e$  exponential of the input argument  $x$ .

### Returns:

Returns  $e^x$ .



#### Note:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.92. `__nv_expm1`

### Prototype:

```
double @__nv_expm1(double %x)
```

### Description:

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

### Returns:

Returns  $e^x - 1$ .



#### Note:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.93. `__nv_expm1f`

### Prototype:

```
float @__nv_expm1f(float %x)
```

### Description:

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

### Returns:

Returns  $e^x - 1$ .



#### Note:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.94. `__nv_fabs`

### Prototype:

```
double @__nv_fabs(double %f)
```

### Description:

Calculate the absolute value of the input argument  $x$ .

### Returns:

Returns the absolute value of the input argument.

- ▶ `__nv_fabs( ±∞ )` returns  $+∞$ .

- ▶ `__nv_fabs( ±0 )` returns 0.

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.95. `__nv_fabsf`

**Prototype:**

```
float @__nv_fabsf(float %f)
```

**Description:**

Calculate the absolute value of the input argument `x`.

**Returns:**

Returns the absolute value of the input argument.

- ▶ `__nv_fabsf( ±∞ )` returns  $+\infty$ .
- ▶ `__nv_fabsf( ±0 )` returns 0.

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.96. `__nv_fadd_rd`

**Prototype:**

```
float @__nv_fadd_rd(float %x, float %y)
```

**Description:**

Compute the sum of  $x$  and  $y$  in round-down (to negative infinity) mode.

**Returns:**

Returns  $x + y$ .

**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.

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.97. `__nv_fadd_rn`

**Prototype:**

```
float @__nv_fadd_rn(float %x, float %y)
```

**Description:**

Compute the sum of  $x$  and  $y$  in round-to-nearest-even rounding mode.

**Returns:**

Returns  $x + y$ .

**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.

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.98. `__nv_fadd_ru`

### Prototype:

```
float @__nv_fadd_ru(float %x, float %y)
```

### Description:

Compute the sum of  $x$  and  $y$  in round-up (to positive infinity) mode.

### Returns:

Returns  $x + y$ .



#### 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.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.99. `__nv_fadd_rz`

### Prototype:

```
float @__nv_fadd_rz(float %x, float %y)
```

### Description:

Compute the sum of  $x$  and  $y$  in round-towards-zero mode.

### Returns:

Returns  $x + y$ .



#### 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.

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.100. `__nv_fast_cosf`

**Prototype:**

```
float @__nv_fast_cosf(float %x)
```

**Description:**

Calculate the fast approximate cosine of the input argument  $x$ , measured in radians.

**Returns:**

Returns the approximate cosine of  $x$ .

**Note:**

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

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.101. `__nv_fast_exp10f`

**Prototype:**

```
float @__nv_fast_exp10f(float %x)
```

**Description:**

Calculate the fast approximate base 10 exponential of the input argument  $x$ ,  $10^x$ .

**Returns:**

Returns an approximation to  $10^x$ .

**Note:**

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

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.102. `__nv_fast_expf`

**Prototype:**

```
float @__nv_fast_expf(float %x)
```

**Description:**

Calculate the fast approximate base  $e$  exponential of the input argument  $x$ ,  $e^x$ .

**Returns:**

Returns an approximation to  $e^x$ .

**Note:**

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

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.103. `__nv_fast_fdividef`

**Prototype:**

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



**Description:**

Calculate the fast approximate division of  $x$  by  $y$ .

**Returns:**

Returns  $x / y$ .

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

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.104. `__nv_fast_log10f`

**Prototype:**

```
float @__nv_fast_log10f(float %x)
```

**Description:**

Calculate the fast approximate base 10 logarithm of the input argument  $x$ .

**Returns:**

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

**Note:**

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

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.105. `__nv_fast_log2f`

### Prototype:

```
float @__nv_fast_log2f(float %x)
```

### Description:

Calculate the fast approximate base 2 logarithm of the input argument  $x$ .

### Returns:

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



#### Note:

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

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.106. `__nv_fast_logf`

### Prototype:

```
float @__nv_fast_logf(float %x)
```

### Description:

Calculate the fast approximate base  $e$  logarithm of the input argument  $x$ .

### Returns:

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



#### Note:

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

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.107. `__nv_fast_powf`

**Prototype:**

```
float @__nv_fast_powf(float %x, float %y)
```

**Description:**

Calculate the fast approximate of  $x$ , the first input argument, raised to the power of  $y$ , the second input argument,  $x^y$ .

**Returns:**

Returns an approximation to  $x^y$ .

**Note:**

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

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.108. `__nv_fast_sincosf`

**Prototype:**

```
void @__nv_fast_sincosf(float %x, float* %sptr, float* %cptr)
```

**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, `zptr`.

**Returns:**

► none

**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.

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.109. `__nv_fast_sinf`

**Prototype:**

```
float __nv_fast_sinf(float x)
```

**Description:**

Calculate the fast approximate sine of the input argument `x`, measured in radians.

**Returns:**

Returns the approximate sine of `x`.

**Note:**

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

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.110. `__nv_fast_tanf`

### Prototype:

```
float @__nv_fast_tanf(float %x)
```

### Description:

Calculate the fast approximate tangent of the input argument `x`, measured in radians.

### Returns:

Returns the approximate tangent of `x`.



#### 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 `__nv_sinf()` by `__nv_cosf()`. Denormal input and output are flushed to sign-preserving 0.0 at each step of the computation.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.111. `__nv_fdim`

### Prototype:

```
double @__nv_fdim(double %x, double %y)
```

### Description:

Compute the positive difference between `x` and `y`. The positive difference is `x - y` when `x > y` and `+0` otherwise.

### Returns:

Returns the positive difference between `x` and `y`.

- ▶ `__nv_fdim(x, y)` returns `x - y` if `x > y`.

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

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.112. `__nv_fdimf`

**Prototype:**

```
float @__nv_fdimf(float %x, float %y)
```

**Description:**

Compute the positive difference between  $x$  and  $y$ . The positive difference is  $x - y$  when  $x > y$  and +0 otherwise.

**Returns:**

Returns the positive difference between  $x$  and  $y$ .

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

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.113. `__nv_fdiv_rd`

**Prototype:**

```
float @__nv_fdiv_rd(float %x, float %y)
```

**Description:**

Divide two floating point values  $x$  by  $y$  in round-down (to negative infinity) mode.

**Returns:**

Returns  $x / y$ .

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.114. `__nv_fdiv_rn`

**Prototype:**

```
float @__nv_fdiv_rn(float %x, float %y)
```

**Description:**

Divide two floating point values  $x$  by  $y$  in round-to-nearest-even mode.

**Returns:**

Returns  $x / y$ .

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.115. `__nv_fdiv_ru`

### Prototype:

```
float @__nv_fdiv_ru(float %x, float %y)
```

### Description:

Divide two floating point values  $x$  by  $y$  in round-up (to positive infinity) mode.

### Returns:

Returns  $x / y$ .



#### Note:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.116. `__nv_fdiv_rz`

### Prototype:

```
float @__nv_fdiv_rz(float %x, float %y)
```

### Description:

Divide two floating point values  $x$  by  $y$  in round-towards-zero mode.

### Returns:

Returns  $x / y$ .



#### Note:

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

### Library Availability:

Compute 2.0: Yes



Compute 3.0: Yes

Compute 3.5: Yes

## 3.117. `__nv_ffs`

### Prototype:

```
i32 @__nv_ffs(i32 %x)
```

### Description:

Find the position of the first (least significant) bit set to 1 in `x`, where the least significant bit position is 1.

### Returns:

Returns a value between 0 and 32 inclusive representing the position of the first bit set.

► `__nv_ffs(0)` returns 0.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.118. `__nv_ffsll`

### Prototype:

```
i32 @__nv_ffsll(i64 %x)
```

### Description:

Find the position of the first (least significant) bit set to 1 in `x`, where the least significant bit position is 1.

### Returns:

Returns a value between 0 and 64 inclusive representing the position of the first bit set.

► `__nv_ffsll(0)` returns 0.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.119. `__nv_finitef`

### Prototype:

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

### Description:

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

### Returns:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.120. `__nv_float2half_rn`

### Prototype:

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

### Description:

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

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.121. `__nv_float2int_rd`

### Prototype:

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

### Description:

Convert the single-precision floating point value  $x$  to a signed integer in round-down (to negative infinity) mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.122. `__nv_float2int_rn`

### Prototype:

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

### Description:

Convert the single-precision floating point value  $x$  to a signed integer in round-to-nearest-even mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.123. `__nv_float2int_ru`

### Prototype:

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

### Description:

Convert the single-precision floating point value  $x$  to a signed integer in round-up (to positive infinity) mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.124. `__nv_float2int_rz`

### Prototype:

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

### Description:

Convert the single-precision floating point value  $x$  to a signed integer in round-towards-zero mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.125. `__nv_float2ll_rd`

### Prototype:

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

### Description:

Convert the single-precision floating point value  $x$  to a signed 64-bit integer in round-down (to negative infinity) mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.126. `__nv_float2ll_rn`

### Prototype:

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

### Description:

Convert the single-precision floating point value  $x$  to a signed 64-bit integer in round-to-nearest-even mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.127. `__nv_float2ll_ru`

### Prototype:

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

### Description:

Convert the single-precision floating point value  $x$  to a signed 64-bit integer in round-up (to positive infinity) mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.128. `__nv_float2ll_rz`

### Prototype:

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

### Description:

Convert the single-precision floating point value  $x$  to a signed 64-bit integer in round-towards-zero mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.129. `__nv_float2uint_rd`

### Prototype:

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

### Description:

Convert the single-precision floating point value  $x$  to an unsigned integer in round-down (to negative infinity) mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.130. `__nv_float2uint_rn`

### Prototype:

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

### Description:

Convert the single-precision floating point value  $x$  to an unsigned integer in round-to-nearest-even mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.131. `__nv_float2uint_ru`

### Prototype:

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

### Description:

Convert the single-precision floating point value  $x$  to an unsigned integer in round-up (to positive infinity) mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.132. `__nv_float2uint_rz`

### Prototype:

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

### Description:

Convert the single-precision floating point value  $x$  to an unsigned integer in round-towards-zero mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes



## 3.133. `__nv_float2ull_rd`

### Prototype:

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

### Description:

Convert the single-precision floating point value  $x$  to an unsigned 64-bit integer in round-down (to negative infinity) mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.134. `__nv_float2ull_rn`

### Prototype:

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

### Description:

Convert the single-precision floating point value  $x$  to an unsigned 64-bit integer in round-to-nearest-even mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.135. `__nv_float2ull_ru`

### Prototype:

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

### Description:

Convert the single-precision floating point value  $x$  to an unsigned 64-bit integer in round-up (to positive infinity) mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.136. `__nv_float2ull_rz`

### Prototype:

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

### Description:

Convert the single-precision floating point value  $x$  to an unsigned 64-bit integer in round-towards\_zero mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.137. `__nv_float_as_int`

### Prototype:

```
i32 @__nv_float_as_int(float %x)
```

### Description:

Reinterpret the bits in the single-precision floating point value  $x$  as a signed integer.

### Returns:

Returns reinterpreted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.138. `__nv_floor`

### Prototype:

```
double @__nv_floor(double %f)
```

### Description:

Calculates the largest integer value which is less than or equal to  $x$ .

### Returns:

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

- ▶ `__nv_floor( ±∞ )` returns  $±∞$ .
- ▶ `__nv_floor( ±0 )` returns  $±0$ .



### Note:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.139. `__nv_floorf`

### Prototype:

```
float @__nv_floorf(float %f)
```

### Description:

Calculates the largest integer value which is less than or equal to  $x$ .

### Returns:

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

- ▶ `__nv_floorf(±∞)` returns  $±∞$ .
- ▶ `__nv_floorf(±0)` returns  $±0$ .



### Note:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.140. `__nv_fma`

### Prototype:

```
double @__nv_fma(double %x, double %y, double %z)
```

### 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.

### Returns:

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

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

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.141. `__nv_fma_rd`

**Prototype:**

```
double @__nv_fma_rd(double %x, double %y, double %z)
```

**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.

**Returns:**

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

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

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.142. `__nv_fma_rn`

### Prototype:

```
double @__nv_fma_rn(double %x, double %y, double %z)
```

### Description:

Computes the value of  $x \times y + z$  as a single ternary operation, rounding the result once in round-to-nearest-even mode.

### Returns:

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

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



### Note:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.143. `__nv_fma_ru`

### Prototype:

```
double @__nv_fma_ru(double %x, double %y, double %z)
```

### 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.

**Returns:**

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

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

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.144. `__nv_fma_rz`

**Prototype:**

```
double @__nv_fma_rz(double %x, double %y, double %z)
```

**Description:**

Computes the value of  $x \times y + z$  as a single ternary operation, rounding the result once in round-towards-zero mode.

**Returns:**

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

- ▶ `__nv_fma_rz( ±∞ , ±0 , z)` returns NaN.
- ▶ `__nv_fma_rz( ±0 , ±∞ , z)` returns NaN.
- ▶ `__nv_fma_rz(x, y, -∞)` returns NaN if  $x \times y$  is an exact  $+\infty$
- ▶ `__nv_fma_rz(x, y, +∞)` returns NaN if  $x \times y$  is an exact  $-\infty$

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.145. `__nv_fmaf`

**Prototype:**

```
float @__nv_fmaf(float %x, float %y, float %z)
```

**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.

**Returns:**

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

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

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.146. `__nv_fmaf_rd`

**Prototype:**

```
float @__nv_fmaf_rd(float %x, float %y, float %z)
```

**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.

**Returns:**

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

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



**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.147. `__nv_fmaf_rn`

**Prototype:**

```
float @__nv_fmaf_rn(float %x, float %y, float %z)
```

**Description:**

Computes the value of  $x \times y + z$  as a single ternary operation, rounding the result once in round-to-nearest-even mode.

**Returns:**

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

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

- ▶ `__nv_fmaf_ru(x, y, +∞)` returns NaN if  $x \times y$  is an exact  $-\infty$ .

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.148. `__nv_fmaf_ru`

**Prototype:**

```
float @__nv_fmaf_ru(float %x, float %y, float %z)
```

**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.

**Returns:**

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

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

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.149. `__nv_fmaf_rz`

### Prototype:

```
float @__nv_fmaf_rz(float %x, float %y, float %z)
```

### Description:

Computes the value of  $x \times y + z$  as a single ternary operation, rounding the result once in round-towards-zero mode.

### Returns:

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

- ▶ `__nv_fmaf_rz(±∞, ±0, z)` returns NaN.
- ▶ `__nv_fmaf_rz(±0, ±∞, z)` returns NaN.
- ▶ `__nv_fmaf_rz(x, y, -∞)` returns NaN if  $x \times y$  is an exact  $+\infty$ .
- ▶ `__nv_fmaf_rz(x, y, +∞)` returns NaN if  $x \times y$  is an exact  $-\infty$ .



### Note:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.150. `__nv_fmax`

### Prototype:

```
double @__nv_fmax(double %x, double %y)
```

### 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.

### 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.



**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.151. `__nv_fmaxf`

**Prototype:**

```
float @__nv_fmaxf(float %x, float %y)
```

**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.

**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.



**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.152. `__nv_fmin`

### Prototype:

```
double @__nv_fmin(double %x, double %y)
```

### 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.

### Returns:

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

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



#### Note:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.153. `__nv_fminf`

### Prototype:

```
float @__nv_fminf(float %x, float %y)
```

### 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.

### Returns:

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

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

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.154. `__nv_fmod`

**Prototype:**

```
double @__nv_fmod(double %x, double %y)
```

**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  $y$ .

**Returns:**

- ▶ Returns the floating-point remainder of  $x / y$ .
- ▶ `__nv_fmod(±0, y)` returns `±0` if  $y$  is not zero.
- ▶ `__nv_fmod(x, ±∞)` returns  $x$  if  $x$  is finite.
- ▶ `__nv_fmod(x, y)` returns NaN if  $x$  is `±∞` or  $y$  is zero.
- ▶ If either argument is NaN, NaN is returned.

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.155. `__nv_fmodf`

### Prototype:

```
float @__nv_fmodf(float %x, float %y)
```

### 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  $y$ .

### Returns:

- ▶ Returns the floating-point remainder of  $x / y$ .
- ▶ `__nv_fmodf(±0, y)` returns `±0` if  $y$  is not zero.
- ▶ `__nv_fmodf(x, ±∞)` returns  $x$  if  $x$  is finite.
- ▶ `__nv_fmodf(x, y)` returns NaN if  $x$  is `±∞` or  $y$  is zero.
- ▶ If either argument is NaN, NaN is returned.



### Note:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.156. `__nv_fmul_rd`

### Prototype:

```
float @__nv_fmul_rd(float %x, float %y)
```

### Description:

Compute the product of  $x$  and  $y$  in round-down (to negative infinity) mode.

**Returns:**

Returns  $x * y$ .

**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.

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.157. `__nv_fmul_rn`

**Prototype:**

```
float @__nv_fmul_rn(float %x, float %y)
```

**Description:**

Compute the product of  $x$  and  $y$  in round-to-nearest-even mode.

**Returns:**

Returns  $x * y$ .

**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.

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.158. `__nv_fmul_ru`

**Prototype:**



```
float @__nv_fmuls_ru(float %x, float %y)
```

**Description:**

Compute the product of  $x$  and  $y$  in round-up (to positive infinity) mode.

**Returns:**

Returns  $x * y$ .

**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.

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.159. `__nv_fmuls_rz`

**Prototype:**

```
float @__nv_fmuls_rz(float %x, float %y)
```

**Description:**

Compute the product of  $x$  and  $y$  in round-towards-zero mode.

**Returns:**

Returns  $x * y$ .

**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.

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.160. `__nv_frcp_rd`

### Prototype:

```
float @__nv_frcp_rd(float %x)
```

### Description:

Compute the reciprocal of  $x$  in round-down (to negative infinity) mode.

### Returns:

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



#### Note:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.161. `__nv_frcp_rn`

### Prototype:

```
float @__nv_frcp_rn(float %x)
```

### Description:

Compute the reciprocal of  $x$  in round-to-nearest-even mode.

### Returns:

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



#### Note:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.162. `__nv_frcp_ru`

### Prototype:

```
float @__nv_frcp_ru(float %x)
```

### Description:

Compute the reciprocal of  $x$  in round-up (to positive infinity) mode.

### Returns:

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



#### Note:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.163. `__nv_frcp_rz`

### Prototype:

```
float @__nv_frcp_rz(float %x)
```

### Description:

Compute the reciprocal of  $x$  in round-towards-zero mode.

### Returns:

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



#### Note:

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

#### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.164. `__nv_frexp`

#### Prototype:

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

#### 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.

#### Returns:

Returns the fractional component  $m$ .

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



#### Note:

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

#### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.165. `__nv_frexp`

### Prototype:

```
float @__nv_frexp(float %x, i32* %b)
```

### 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.

### Returns:

Returns the fractional component  $m$ .

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



### Note:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.166. `__nv_frsqrt_rn`

### Prototype:

```
float @__nv_frsqrt_rn(float %x)
```

### Description:

Compute the reciprocal square root of  $x$  in round-to-nearest-even mode.

**Returns:**

Returns  $1/\sqrt{x}$ .



**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.167. `__nv_fsqrt_rd`

**Prototype:**

```
float @__nv_fsqrt_rd(float %x)
```

**Description:**

Compute the square root of  $x$  in round-down (to negative infinity) mode.

**Returns:**

Returns  $\sqrt{x}$ .



**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.168. `__nv_fsqrt_rn`

**Prototype:**

```
float @__nv_fsqrt_rn(float %x)
```

**Description:**

Compute the square root of  $x$  in round-to-nearest-even mode.

**Returns:**

Returns  $\sqrt{x}$ .

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.169. `__nv_fsqrt_ru`

**Prototype:**

```
float @__nv_fsqrt_ru(float %x)
```

**Description:**

Compute the square root of  $x$  in round-up (to positive infinity) mode.

**Returns:**

Returns  $\sqrt{x}$ .

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.170. `__nv_fsqrt_rz`

### Prototype:

```
float @__nv_fsqrt_rz(float %x)
```

### Description:

Compute the square root of  $x$  in round-towards-zero mode.

### Returns:

Returns  $\sqrt{x}$ .



#### Note:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.171. `__nv_fsub_rd`

### Prototype:

```
float @__nv_fsub_rd(float %x, float %y)
```

### Description:

Compute the difference of  $x$  and  $y$  in round-down (to negative infinity) mode.

### Returns:

Returns  $x - y$ .



#### 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.

### Library Availability:



Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.172. `__nv_fsub_rn`

### Prototype:

```
float @__nv_fsub_rn(float %x, float %y)
```

### Description:

Compute the difference of `x` and `y` in round-to-nearest-even rounding mode.

### Returns:

Returns `x - y`.



#### 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.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.173. `__nv_fsub_ru`

### Prototype:

```
float @__nv_fsub_ru(float %x, float %y)
```

### Description:

Compute the difference of `x` and `y` in round-up (to positive infinity) mode.

### Returns:

Returns  $x - y$ .

**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.

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.174. `__nv_fsub_rz`

**Prototype:**

```
float @__nv_fsub_rz(float %x, float %y)
```

**Description:**

Compute the difference of  $x$  and  $y$  in round-towards-zero mode.

**Returns:**

Returns  $x - y$ .

**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.

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.175. `__nv_hadd`

**Prototype:**

```
i32 @__nv_hadd(i32 %x, i32 %y)
```

**Description:**

Compute average of signed input arguments  $x$  and  $y$  as  $(x + y) \gg 1$ , avoiding overflow in the intermediate sum.

**Returns:**

Returns a signed integer value representing the signed average value of the two inputs.

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.176. `__nv_half2float`

**Prototype:**

```
float __nv_half2float(int16_t h)
```

**Description:**

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

**Returns:**

Returns converted value.

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.177. `__nv_hiloint2double`

**Prototype:**

```
double __nv_hiloint2double(int32_t hi, int32_t lo)
```

**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.

**Returns:**

Returns reinterpreted value.

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.178. `__nv_hypot`

**Prototype:**

```
double @__nv_hypot(double %x, double %y)
```

**Description:**

Calculate the length of the hypotenuse of a right triangle whose two sides have lengths  $x$  and  $y$  without undue overflow or underflow.

**Returns:**

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

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.179. `__nv_hypotf`

**Prototype:**

```
float @__nv_hypotf(float %x, float %y)
```

**Description:**

Calculate the length of the hypotenuse of a right triangle whose two sides have lengths  $x$  and  $y$  without undue overflow or underflow.

**Returns:**

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



**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.180. `__nv_ilogb`

**Prototype:**

```
i32 @__nv_ilogb(double %x)
```

**Description:**

Calculates the unbiased integer exponent of the input argument  $x$ .

**Returns:**

- ▶ If successful, returns the unbiased exponent of the argument.
- ▶ `__nv_ilogb(0)` returns `INT_MIN`.
- ▶ `__nv_ilogb(NaN)` returns `INT_MIN`.
- ▶ `__nv_ilogb(x)` returns `INT_MAX` if  $x$  is  $\infty$  or the correct value is greater than `INT_MAX`.
- ▶ `__nv_ilogb(x)` return `INT_MIN` if the correct value is less than `INT_MIN`.



**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.181. `__nv_ilogbf`

### Prototype:

```
i32 @__nv_ilogbf(float %x)
```

### Description:

Calculates the unbiased integer exponent of the input argument  $x$ .

### Returns:

- ▶ If successful, returns the unbiased exponent of the argument.
- ▶ `__nv_ilogbf(0)` returns `INT_MIN`.
- ▶ `__nv_ilogbf(NaN)` returns `INT_MIN`.
- ▶ `__nv_ilogbf(x)` returns `INT_MAX` if  $x$  is  $\infty$  or the correct value is greater than `INT_MAX`.
- ▶ `__nv_ilogbf(x)` return `INT_MIN` if the correct value is less than `INT_MIN`.



### Note:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.182. `__nv_int2double_rn`

### Prototype:

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

### Description:

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

### Returns:

Returns converted value.

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.183. `__nv_int2float_rd`

**Prototype:**

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

**Description:**

Convert the signed integer value  $x$  to a single-precision floating point value in round-down (to negative infinity) mode.

**Returns:**

Returns converted value.

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.184. `__nv_int2float_rn`

**Prototype:**

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

**Description:**

Convert the signed integer value  $x$  to a single-precision floating point value in round-to-nearest-even mode.

**Returns:**

Returns converted value.

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.185. `__nv_int2float_ru`

### Prototype:

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

### Description:

Convert the signed integer value `x` to a single-precision floating point value in round-up (to positive infinity) mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.186. `__nv_int2float_rz`

### Prototype:

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

### Description:

Convert the signed integer value `x` to a single-precision floating point value in round-towards-zero mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes



## 3.187. `__nv_int_as_float`

### Prototype:

```
float @__nv_int_as_float(i32 %x)
```

### Description:

Reinterpret the bits in the signed integer value `x` as a single-precision floating point value.

### Returns:

Returns reinterpreted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.188. `__nv_isfinited`

### Prototype:

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

### Description:

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

### Returns:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.189. `__nv_isinfd`

### Prototype:

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

### Description:

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

### Returns:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.190. `__nv_isinff`

### Prototype:

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

### Description:

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

### Returns:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.191. `__nv_isnand`

### Prototype:

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

**Description:**

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

**Returns:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.192. `__nv_isnanf`

**Prototype:**

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

**Description:**

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

**Returns:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.193. `__nv_j0`

**Prototype:**

```
double @__nv_j0(double %x)
```

**Description:**

Calculate the value of the Bessel function of the first kind of order 0 for the input argument  $x$ ,  $J_0(x)$ .

**Returns:**

Returns the value of the Bessel function of the first kind of order 0.

- ▶ `__nv_j0( ± ∞ )` returns +0.
- ▶ `__nv_j0(NaN)` returns NaN.

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.194. `__nv_j0f`

**Prototype:**

```
float __nv_j0f(float %x)
```

**Description:**

Calculate the value of the Bessel function of the first kind of order 0 for the input argument  $x$ ,  $J_0(x)$ .

**Returns:**

Returns the value of the Bessel function of the first kind of order 0.

- ▶ `__nv_j0f( ± ∞ )` returns +0.
- ▶ `__nv_j0f(NaN)` returns NaN.

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.195. `__nv_j1`

### Prototype:

```
double @__nv_j1(double %x)
```

### Description:

Calculate the value of the Bessel function of the first kind of order 1 for the input argument  $x$ ,  $J_1(x)$ .

### Returns:

Returns the value of the Bessel function of the first kind of order 1.

- ▶ `__nv_j1(±0)` returns  $±0$ .
- ▶ `__nv_j1(±∞)` returns  $±0$ .
- ▶ `__nv_j1(NaN)` returns NaN.



### Note:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.196. `__nv_j1f`

### Prototype:

```
float @__nv_j1f(float %x)
```

### Description:

Calculate the value of the Bessel function of the first kind of order 1 for the input argument  $x$ ,  $J_1(x)$ .

### Returns:

Returns the value of the Bessel function of the first kind of order 1.

- ▶ `__nv_j1f( ±0 )` returns  $±0$ .
- ▶ `__nv_j1f( ±∞ )` returns  $±0$ .
- ▶ `__nv_j1f(NaN)` returns NaN.

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.197. `__nv_jn`

**Prototype:**

```
double @__nv_jn(i32 %n, double %x)
```

**Description:**

Calculate the value of the Bessel function of the first kind of order  $n$  for the input argument  $x$ ,  $J_n(x)$ .

**Returns:**

Returns the value of the Bessel function of the first kind of order  $n$ .

- ▶ `__nv_jn(n, NaN)` returns NaN.
- ▶ `__nv_jn(n, x)` returns NaN for  $n < 0$ .
- ▶ `__nv_jn(n, +∞)` returns  $+0$ .

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.198. `__nv_jnf`

### Prototype:

```
float @__nv_jnf(i32 %n, float %x)
```

### Description:

Calculate the value of the Bessel function of the first kind of order  $n$  for the input argument  $x$ ,  $J_n(x)$ .

### Returns:

Returns the value of the Bessel function of the first kind of order  $n$ .

- ▶ `__nv_jnf(n, NaN)` returns NaN.
- ▶ `__nv_jnf(n, x)` returns NaN for  $n < 0$ .
- ▶ `__nv_jnf(n, +∞)` returns +0.



### Note:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.199. `__nv_ldexp`

### Prototype:

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

### Description:

Calculate the value of  $x \cdot 2^{exp}$  of the input arguments  $x$  and  $exp$ .

### Returns:

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

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.200. `__nv_ldexpf`

**Prototype:**

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

**Description:**

Calculate the value of  $x \cdot 2^{exp}$  of the input arguments `x` and `exp`.

**Returns:**

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

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.201. `__nv_lgamma`

**Prototype:**

```
double @__nv_lgamma(double %x)
```



**Description:**

Calculate the natural logarithm of the absolute value of the gamma function of the input argument  $x$ , namely the value of  $\log_e\left(\int_0^\infty e^{-t}t^{x-1}dt\right)$

**Returns:**

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

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.202. `__nv_lgammaf`

**Prototype:**

```
float @__nv_lgammaf(float %x)
```

**Description:**

Calculate the natural logarithm of the absolute value of the gamma function of the input argument  $x$ , namely the value of  $\log_e\left(\int_0^\infty e^{-t}t^{x-1}dt\right)$

**Returns:**

- ▶ `__nv_lgammaf(1)` returns +0.
- ▶ `__nv_lgammaf(2)` returns +0.

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

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.203. `__nv_ll2double_rd`

**Prototype:**

```
double @__nv_ll2double_rd(i64 %1)
```

**Description:**

Convert the signed 64-bit integer value  $x$  to a double-precision floating point value in round-down (to negative infinity) mode.

**Returns:**

Returns converted value.

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.204. `__nv_ll2double_rn`

**Prototype:**

```
double @__nv_ll2double_rn(i64 %1)
```

**Description:**

Convert the signed 64-bit integer value  $x$  to a double-precision floating point value in round-to-nearest-even mode.

**Returns:**

Returns converted value.

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.205. `__nv_ll2double_ru`

**Prototype:**

```
double __nv_ll2double_ru(i64 %1)
```

**Description:**

Convert the signed 64-bit integer value  $x$  to a double-precision floating point value in round-up (to positive infinity) mode.

**Returns:**

Returns converted value.

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.206. `__nv_ll2double_rz`

**Prototype:**

```
double __nv_ll2double_rz(i64 %1)
```

**Description:**

Convert the signed 64-bit integer value  $x$  to a double-precision floating point value in round-towards-zero mode.

**Returns:**

Returns converted value.

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.207. `__nv_ll2float_rd`

**Prototype:**

```
float __nv_ll2float_rd(i64 %1)
```

**Description:**

Convert the signed integer value  $x$  to a single-precision floating point value in round-down (to negative infinity) mode.

**Returns:**

Returns converted value.

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.208. `__nv_ll2float_rn`

**Prototype:**

```
float __nv_ll2float_rn(i64 %1)
```

**Description:**

Convert the signed 64-bit integer value  $x$  to a single-precision floating point value in round-to-nearest-even mode.

**Returns:**

Returns converted value.

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.209. `__nv_ll2float_ru`

### Prototype:

```
float @__nv_ll2float_ru(i64 %1)
```

### Description:

Convert the signed integer value `x` to a single-precision floating point value in round-up (to positive infinity) mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.210. `__nv_ll2float_rz`

### Prototype:

```
float @__nv_ll2float_rz(i64 %1)
```

### Description:

Convert the signed integer value `x` to a single-precision floating point value in round-towards-zero mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.211. `__nv_llabs`

### Prototype:

```
i64 @__nv_llabs(i64 %x)
```

### Description:

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

### Returns:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.212. `__nv_llmax`

### Prototype:

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

### Description:

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

### Returns:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.213. `__nv_llmin`

### Prototype:

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

**Description:**

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

**Returns:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.214. `__nv_llrint`

**Prototype:**

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

**Description:**

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

**Returns:**

Returns rounded integer value.

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.215. `__nv_llrintf`

**Prototype:**

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

**Description:**

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

**Returns:**

Returns rounded integer value.

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.216. `__nv_llround`

**Prototype:**

```
i64 @__nv_llround(double %x)
```

**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 result is undefined.

**Returns:**

Returns rounded integer value.

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.217. `__nv_llroundf`

**Prototype:**

```
i64 @__nv_llroundf(float %x)
```

**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 result is undefined.

**Returns:**

Returns rounded integer value.

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.218. `__nv_log`

**Prototype:**

```
double @__nv_log(double %x)
```

**Description:**

Calculate the base  $e$  logarithm of the input argument  $x$ .

**Returns:**

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

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.219. `__nv_log10`

### Prototype:

```
double @__nv_log10(double %x)
```

### Description:

Calculate the base 10 logarithm of the input argument  $x$ .

### Returns:

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



### Note:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.220. `__nv_log10f`

### Prototype:

```
float @__nv_log10f(float %x)
```

### Description:

Calculate the base 10 logarithm of the input argument  $x$ .

### Returns:

- ▶ `__nv_log10f(±0)` returns  $-\infty$ .
- ▶ `__nv_log10f(1)` returns  $+0$ .

- ▶ `__nv_log10f(x)` returns NaN for  $x < 0$ .
- ▶ `__nv_log10f(+∞)` returns  $+∞$ .

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.221. `__nv_log1p`

**Prototype:**

```
double @__nv_log1p(double %x)
```

**Description:**

Calculate the value of  $\log_e(1+x)$  of the input argument  $x$ .

**Returns:**

- ▶ `__nv_log1p(±0)` returns  $-∞$ .
- ▶ `__nv_log1p(-1)` returns  $+0$ .
- ▶ `__nv_log1p(x)` returns NaN for  $x < -1$ .
- ▶ `__nv_log1p(+∞)` returns  $+∞$ .

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.222. `__nv_log1pf`

### Prototype:

```
float @__nv_log1pf(float %x)
```

### Description:

Calculate the value of  $\log_e(1+x)$  of the input argument  $x$ .

### Returns:

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



#### Note:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.223. `__nv_log2`

### Prototype:

```
double @__nv_log2(double %x)
```

### Description:

Calculate the base 2 logarithm of the input argument  $x$ .

### Returns:

- ▶ `__nv_log2(±0)` returns  $-\infty$ .
- ▶ `__nv_log2(1)` returns  $+0$ .

- ▶ `__nv_log2(x)` returns NaN for  $x < 0$ .
- ▶ `__nv_log2(+∞)` returns  $+∞$ .

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.224. `__nv_log2f`

**Prototype:**

```
float @__nv_log2f(float %x)
```

**Description:**

Calculate the base 2 logarithm of the input argument  $x$ .

**Returns:**

- ▶ `__nv_log2f(±0)` returns  $-∞$ .
- ▶ `__nv_log2f(1)` returns  $+0$ .
- ▶ `__nv_log2f(x)` returns NaN for  $x < 0$ .
- ▶ `__nv_log2f(+∞)` returns  $+∞$ .

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.225. `__nv_logb`

### Prototype:

```
double @__nv_logb(double %x)
```

### Description:

Calculate the floating point representation of the exponent of the input argument  $x$ .

### Returns:

- ▶ `__nv_logb ±0` returns  $-\infty$
- ▶ `__nv_logb ±∞` returns  $+\infty$



#### Note:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.226. `__nv_logbf`

### Prototype:

```
float @__nv_logbf(float %x)
```

### Description:

Calculate the floating point representation of the exponent of the input argument  $x$ .

### Returns:

- ▶ `__nv_logbf ±0` returns  $-\infty$
- ▶ `__nv_logbf ±∞` returns  $+\infty$



#### Note:

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.227. `__nv_logf`

**Prototype:**

```
float __nv_logf(float %x)
```

**Description:**

Calculate the base  $e$  logarithm of the input argument  $x$ .

**Returns:**

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

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.228. `__nv_longlong_as_double`

**Prototype:**

```
double __nv_longlong_as_double(i64 %x)
```

**Description:**

Reinterpret the bits in the 64-bit signed integer value `x` as a double-precision floating point value.

**Returns:**

Returns reinterpreted value.

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.229. `__nv_max`

**Prototype:**

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

**Description:**

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

**Returns:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.230. `__nv_min`

**Prototype:**

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

**Description:**

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

**Returns:**

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



**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.231. `__nv_modf`

**Prototype:**

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

**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  $x$ .

**Returns:**

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

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.232. `__nv_modff`

**Prototype:**

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

**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  $x$ .

**Returns:**

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

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.233. `__nv_mul24`

**Prototype:**

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

**Returns:**

Returns the least significant 32 bits of the product `x * y`.

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.234. `__nv_mul64hi`

**Prototype:**

```
i64 @__nv_mul64hi(i64 %x, i64 %y)
```

**Description:**

Calculate the most significant 64 bits of the 128-bit product  $x * y$ , where  $x$  and  $y$  are 64-bit integers.

**Returns:**

Returns the most significant 64 bits of the product  $x * y$ .

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.235. `__nv_mulhi`

**Prototype:**

```
i32 @__nv_mulhi(i32 %x, i32 %y)
```

**Description:**

Calculate the most significant 32 bits of the 64-bit product  $x * y$ , where  $x$  and  $y$  are 32-bit integers.

**Returns:**

Returns the most significant 32 bits of the product  $x * y$ .

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.236. `__nv_nan`

**Prototype:**

```
double @__nv_nan(i8* %tagp)
```

**Description:**

Return a representation of a quiet NaN. Argument `tagp` selects one of the possible representations.

**Returns:**

- ▶ `__nv_nan(tagp)` returns NaN.

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.237. `__nv_nanf`

**Prototype:**

```
float @__nv_nanf(i8* %tagp)
```

**Description:**

Return a representation of a quiet NaN. Argument `tagp` selects one of the possible representations.

**Returns:**

- ▶ `__nv_nanf(tagp)` returns NaN.

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.238. `__nv_nearbyint`

**Prototype:**

```
double @__nv_nearbyint(double %x)
```

**Description:**

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

**Returns:**

- ▶ `__nv_nearbyint( ±0 )` returns  $±0$ .
- ▶ `__nv_nearbyint( ±∞ )` returns  $±∞$ .

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.239. `__nv_nearbyintf`

**Prototype:**

```
float @__nv_nearbyintf(float %x)
```

**Description:**

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

**Returns:**

- ▶ `__nv_nearbyintf( ±0 )` returns  $±0$ .
- ▶ `__nv_nearbyintf( ±∞ )` returns  $±∞$ .

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.240. `__nv_nextafter`

### Prototype:

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

### 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  $x$ .

### Returns:

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



#### Note:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.241. `__nv_nextafterf`

### Prototype:

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

### 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  $x$ .

### Returns:

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



#### Note:

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.242. `__nv_normcdf`

**Prototype:**

```
double @__nv_normcdf(double %x)
```

**Description:**

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

**Returns:**

- ▶ `__nv_normcdf(+∞)` returns 1
- ▶ `__nv_normcdf(-∞)` returns +0

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.243. `__nv_normcdfff`

**Prototype:**

```
float @__nv_normcdfff(float %x)
```

**Description:**

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

**Returns:**

- ▶ `__nv_normcdfff(+∞)` returns 1
- ▶ `__nv_normcdfff(-∞)` returns +0



**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.244. `__nv_normcdfinv`

**Prototype:**

```
double @__nv_normcdfinv(double %x)
```

**Description:**

Calculate the inverse of the standard normal cumulative distribution function for input argument  $y$ ,  $\Phi^{-1}(y)$ . The function is defined for input values in the interval  $(0, 1)$ .

**Returns:**

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



**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes



## 3.245. `__nv_normcdfinvf`

### Prototype:

```
float @__nv_normcdfinvf(float %x)
```

### Description:

Calculate the inverse of the standard normal cumulative distribution function for input argument  $y$ ,  $\Phi^{-1}(y)$ . The function is defined for input values in the interval  $(0, 1)$ .

### Returns:

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



### Note:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.246. `__nv_popc`

### Prototype:

```
i32 @__nv_popc(i32 %x)
```

### Description:

Count the number of bits that are set to 1 in  $x$ .

### Returns:

Returns a value between 0 and 32 inclusive representing the number of set bits.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.247. `__nv_popc11`

### Prototype:

```
i32 @__nv_popc11(i64 %x)
```

### Description:

Count the number of bits that are set to 1 in  $x$ .

### Returns:

Returns a value between 0 and 64 inclusive representing the number of set bits.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.248. `__nv_pow`

### Prototype:

```
double @__nv_pow(double %x, double %y)
```

### Description:

Calculate the value of  $x$  to the power of  $y$

### Returns:

- ▶ `__nv_pow(±0, y)` returns  $±∞$  for  $y$  an integer less than 0.
- ▶ `__nv_pow(±0, y)` returns  $±0$  for  $y$  an odd integer greater than 0.
- ▶ `__nv_pow(±0, y)` returns +0 for  $y > 0$  and not an odd integer.
- ▶ `__nv_pow(-1, ±∞)` returns 1.
- ▶ `__nv_pow(+1, y)` returns 1 for any  $y$ , even a NaN.
- ▶ `__nv_pow(x, ±0)` returns 1 for any  $x$ , even a NaN.

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

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.249. `__nv_powf`

**Prototype:**

```
float @__nv_powf(float %x, float %y)
```

**Description:**

Calculate the value of  $x$  to the power of  $y$

**Returns:**

- ▶ `__nv_powf(±0, y)` returns  $±\infty$  for  $y$  an integer less than  $0$ .
- ▶ `__nv_powf(±0, y)` returns  $±0$  for  $y$  an odd integer greater than  $0$ .
- ▶ `__nv_powf(±0, y)` returns  $+0$  for  $y > 0$  and not an odd integer.

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

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.250. `__nv_powi`

**Prototype:**

```
double @__nv_powi(double %x, i32 %y)
```

**Description:**

Calculate the value of `x` to the power of `y`

**Returns:**

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

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.251. `__nv_powif`

**Prototype:**

```
float @__nv_powif(float %x, i32 %y)
```

**Description:**

Calculate the value of  $x$  to the power of  $y$ .

**Returns:**

- ▶ `__nv_powif( ±0 , y )` returns  $±∞$  for  $y$  an integer less than 0.
- ▶ `__nv_powif( ±0 , y )` returns  $±0$  for  $y$  an odd integer greater than 0.
- ▶ `__nv_powif( ±0 , y )` returns +0 for  $y > 0$  and not an odd integer.
- ▶ `__nv_powif(-1, ±∞ )` returns 1.
- ▶ `__nv_powif(+1, y)` returns 1 for any  $y$ , even a NaN.
- ▶ `__nv_powif(x, ±0 )` returns 1 for any  $x$ , even a NaN.
- ▶ `__nv_powif(x, y)` returns a NaN for finite  $x < 0$  and finite non-integer  $y$ .
- ▶ `__nv_powif(x, -∞ )` returns  $+∞$  for  $|x| < 1$ .
- ▶ `__nv_powif(x, -∞ )` returns +0 for  $|x| > 1$ .
- ▶ `__nv_powif(x, +∞ )` returns +0 for  $|x| < 1$ .
- ▶ `__nv_powif(x, +∞ )` returns  $+∞$  for  $|x| > 1$ .
- ▶ `__nv_powif( -∞ , y )` returns -0 for  $y$  an odd integer less than 0.
- ▶ `__nv_powif( -∞ , y )` returns +0 for  $y < 0$  and not an odd integer.
- ▶ `__nv_powif( -∞ , y )` returns  $-∞$  for  $y$  an odd integer greater than 0.
- ▶ `__nv_powif( -∞ , y )` returns  $+∞$  for  $y > 0$  and not an odd integer.
- ▶ `__nv_powif( +∞ , y )` returns +0 for  $y < 0$ .
- ▶ `__nv_powif( +∞ , y )` returns  $+∞$  for  $y > 0$ .

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.252. `__nv_rcbrt`

### Prototype:

```
double @__nv_rcbrt(double %x)
```

### Description:

Calculate reciprocal cube root function of  $x$

### Returns:

- ▶ `__nv_rcbrt( ±0 )` returns  $±∞$ .
- ▶ `__nv_rcbrt( ±∞ )` returns  $±0$ .



#### Note:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.253. `__nv_rcbrtf`

### Prototype:

```
float @__nv_rcbrtf(float %x)
```

### Description:

Calculate reciprocal cube root function of  $x$

### Returns:

- ▶ `__nv_rcbrtf( ±0 )` returns  $±∞$ .
- ▶ `__nv_rcbrtf( ±∞ )` returns  $±0$ .



#### Note:

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.254. `__nv_remainder`

**Prototype:**

```
double @__nv_remainder(double %x, double %y)
```

**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  $\left|n - \frac{x}{y}\right| = \frac{1}{2}$ , the even  $n$  value is chosen.

**Returns:**

- ▶ `__nv_remainder(x, 0)` returns NaN.
- ▶ `__nv_remainder(±∞, y)` returns NaN.
- ▶ `__nv_remainder(x, ±∞)` returns  $x$  for finite  $x$ .

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.255. `__nv_remainderf`

**Prototype:**

```
float @__nv_remainderf(float %x, float %y)
```



**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  $\left|n - \frac{x}{y}\right| = \frac{1}{2}$ , the even  $n$  value is chosen.

**Returns:**

- ▶ `__nv_remainderf(x, 0)` returns NaN.
- ▶ `__nv_remainderf(±∞, y)` returns NaN.
- ▶ `__nv_remainderf(x, ±∞)` returns  $x$  for finite  $x$ .

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.256. `__nv_remquo`

**Prototype:**

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

**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.

**Returns:**

Returns the remainder.

- ▶ `__nv_remquo(x, 0, quo)` returns NaN.
- ▶ `__nv_remquo(±∞, y, quo)` returns NaN.

- ▶ `__nv_remquof(x, ±∞, quo)` returns `x`.

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.257. `__nv_remquof`

**Prototype:**

```
float @__nv_remquof(float %x, float %y, i32* %quo)
```

**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.

**Returns:**

Returns the remainder.

- ▶ `__nv_remquof(x, 0, quo)` returns NaN.
- ▶ `__nv_remquof(±∞, y, quo)` returns NaN.
- ▶ `__nv_remquof(x, ±∞, quo)` returns `x`.

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.258. `__nv_rhadd`

### Prototype:

```
i32 @__nv_rhadd(i32 %x, i32 %y)
```

### Description:

Compute average of signed input arguments  $x$  and  $y$  as  $(x + y + 1) \gg 1$ , avoiding overflow in the intermediate sum.

### Returns:

Returns a signed integer value representing the signed rounded average value of the two inputs.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.259. `__nv_rint`

### Prototype:

```
double @__nv_rint(double %x)
```

### Description:

Round  $x$  to the nearest integer value in floating-point format, with halfway cases rounded to the nearest even integer value.

### Returns:

Returns rounded integer value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.260. `__nv_rintf`

### Prototype:

```
float @__nv_rintf(float %x)
```

### Description:

Round  $x$  to the nearest integer value in floating-point format, with halfway cases rounded to the nearest even integer value.

### Returns:

Returns rounded integer value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.261. `__nv_round`

### Prototype:

```
double @__nv_round(double %x)
```

### Description:

Round  $x$  to the nearest integer value in floating-point format, with halfway cases rounded away from zero.

### Returns:

Returns rounded integer value.



#### Note:

This function may be slower than alternate rounding methods. See `rint()`.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.262. `__nv_roundf`

### Prototype:

```
float @__nv_roundf(float %x)
```

### Description:

Round  $x$  to the nearest integer value in floating-point format, with halfway cases rounded away from zero.

### Returns:

Returns rounded integer value.



### Note:

This function may be slower than alternate rounding methods. See `rint()`.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.263. `__nv_rsqrt`

### Prototype:

```
double @__nv_rsqrt(double %x)
```

### Description:

Calculate the reciprocal of the nonnegative square root of  $x$ ,  $1/\sqrt{x}$ .

### Returns:

Returns  $1/\sqrt{x}$ .

- ▶ `__nv_rsqrt(+∞)` returns  $+0$ .
- ▶ `__nv_rsqrt(±0)` returns  $±∞$ .

- ▶ `__nv_rsqrt(x)` returns NaN if  $x$  is less than 0.

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.264. `__nv_rsqrtf`

**Prototype:**

```
float @__nv_rsqrtf(float %x)
```

**Description:**

Calculate the reciprocal of the nonnegative square root of  $x$ ,  $1/\sqrt{x}$ .

**Returns:**

Returns  $1/\sqrt{x}$ .

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

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.265. `__nv_sad`

### Prototype:

```
i32 @__nv_sad(i32 %x, i32 %y, i32 %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.

### Returns:

Returns  $|x - y| + z$ .

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.266. `__nv_saturatef`

### Prototype:

```
float @__nv_saturatef(float %x)
```

### Description:

Clamp the input argument  $x$  to be within the interval  $[+0.0, 1.0]$ .

### Returns:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.267. `__nv_scalbn`

### Prototype:

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

### Description:

Scale  $x$  by  $2^n$  by efficient manipulation of the floating-point exponent.

### Returns:

Returns  $x * 2^n$ .

- ▶ `__nv_scalbn(±0, n)` returns  $±0$ .
- ▶ `__nv_scalbn(x, 0)` returns  $x$ .
- ▶ `__nv_scalbn(±∞, n)` returns  $±∞$ .

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.268. `__nv_scalbnf`

### Prototype:

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

### Description:

Scale  $x$  by  $2^n$  by efficient manipulation of the floating-point exponent.

### Returns:

Returns  $x * 2^n$ .

- ▶ `__nv_scalbnf(±0, n)` returns  $±0$ .
- ▶ `__nv_scalbnf(x, 0)` returns  $x$ .
- ▶ `__nv_scalbnf(±∞, n)` returns  $±∞$ .



**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.269. `__nv_signbitd`

**Prototype:**

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

**Description:**

Determine whether the floating-point value `x` is negative.

**Returns:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.270. `__nv_signbitf`

**Prototype:**

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

**Description:**

Determine whether the floating-point value `x` is negative.

**Returns:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.271. `__nv_sin`

### Prototype:

```
double @__nv_sin(double %x)
```

### Description:

Calculate the sine of the input argument  $x$  (measured in radians).

### Returns:

- ▶ `__nv_sin( ±0 )` returns  $±0$ .
- ▶ `__nv_sin( ±∞ )` returns NaN.



#### Note:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.272. `__nv_sincos`

### Prototype:

```
void @__nv_sincos(double %x, double* %sptr, double* %cptr)
```

### 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, `zptr`.

### Returns:

- ▶ none

See `__nv_sin()` and `__nv_cos()`.

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.273. `__nv_sincosf`

**Prototype:**

```
void @__nv_sincosf(float %x, float* %sptr, float* %cptr)
```

**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, `zptr`.

**Returns:**

► none

See `__nv_sinf()` and `__nv_cosf()`.

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.274. `__nv_sincospi`

**Prototype:**

```
void @__nv_sincospi(double %x, double* %sptr, double* %cptr)
```

**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, `zptr`.

**Returns:**

- ▶ none

See `__nv_sinpil()` and `__nv_cospil()`.

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.275. `__nv_sincospif`

**Prototype:**

```
void __nv_sincospif(float x, float* sptr, float* cptr)
```

**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, `zptr`.

**Returns:**

- ▶ none

See `__nv_sinpif()` and `__nv_cospif()`.

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.276. `__nv_sinf`

### Prototype:

```
float @__nv_sinf(float %x)
```

### Description:

Calculate the sine of the input argument  $x$  (measured in radians).

### Returns:

- ▶ `__nv_sinf( ±0 )` returns  $±0$ .
- ▶ `__nv_sinf( ±∞ )` returns NaN.



#### Note:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.277. `__nv_sinh`

### Prototype:

```
double @__nv_sinh(double %x)
```

### Description:

Calculate the hyperbolic sine of the input argument  $x$ .

### Returns:

- ▶ `__nv_sinh( ±0 )` returns  $±0$ .



#### Note:

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.278. `__nv_sinhf`

**Prototype:**

```
float @__nv_sinhf(float %x)
```

**Description:**

Calculate the hyperbolic sine of the input argument  $x$ .

**Returns:**

► `__nv_sinhf(±0)` returns  $±0$ .

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.279. `__nv_sinpi`

**Prototype:**

```
double @__nv_sinpi(double %x)
```

**Description:**

Calculate the sine of  $x \times \pi$  (measured in radians), where  $x$  is the input argument.

**Returns:**

- ▶ `__nv_sinpif( ±0 )` returns  $\pm 0$ .
- ▶ `__nv_sinpif( ±∞ )` returns NaN.

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.280. `__nv_sinpif`

**Prototype:**

```
float @__nv_sinpif(float %x)
```

**Description:**

Calculate the sine of  $x \times \pi$  (measured in radians), where  $x$  is the input argument.

**Returns:**

- ▶ `__nv_sinpif( ±0 )` returns  $\pm 0$ .
- ▶ `__nv_sinpif( ±∞ )` returns NaN.

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.281. `__nv_sqrt`

**Prototype:**

```
double @__nv_sqrt(double %x)
```

**Description:**

Calculate the nonnegative square root of  $x$ ,  $\sqrt{x}$ .

**Returns:**

Returns  $\sqrt{x}$ .

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

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.282. `__nv_sqrtf`

**Prototype:**

```
float @__nv_sqrtf(float %x)
```

**Description:**

Calculate the nonnegative square root of  $x$ ,  $\sqrt{x}$ .

**Returns:**

Returns  $\sqrt{x}$ .

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

**Note:**



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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.283. `__nv_tan`

**Prototype:**

```
double @__nv_tan(double %x)
```

**Description:**

Calculate the tangent of the input argument  $x$  (measured in radians).

**Returns:**

- ▶ `__nv_tan(±0)` returns  $±0$ .
- ▶ `__nv_tan(±∞)` returns NaN.

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.284. `__nv_tanf`

**Prototype:**

```
float @__nv_tanf(float %x)
```

**Description:**

Calculate the tangent of the input argument  $x$  (measured in radians).

**Returns:**

- ▶ `__nv_tanf( ±0 )` returns  $±0$ .
- ▶ `__nv_tanf( ±∞ )` returns NaN.

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.285. `__nv_tanh`

**Prototype:**

```
double @__nv_tanh(double %x)
```

**Description:**

Calculate the hyperbolic tangent of the input argument `x`.

**Returns:**

- ▶ `__nv_tanh( ±0 )` returns  $±0$ .

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.286. `__nv_tanhf`

**Prototype:**

```
float @__nv_tanhf(float %x)
```

**Description:**

Calculate the hyperbolic tangent of the input argument  $x$ .

**Returns:**

- ▶ `__nv_tanhf( ±0 )` returns  $±0$ .

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.287. `__nv_tgamma`

**Prototype:**

```
double @__nv_tgamma(double %x)
```

**Description:**

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

**Returns:**

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

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.288. `__nv_tgammaf`

**Prototype:**

```
float __nv_tgammaf(float %x)
```

**Description:**

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

**Returns:**

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

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.289. `__nv_trunc`

**Prototype:**

```
double @__nv_trunc(double %x)
```

**Description:**

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

**Returns:**

Returns truncated integer value.

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.290. `__nv_truncf`

**Prototype:**

```
float @__nv_truncf(float %x)
```

**Description:**

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

**Returns:**

Returns truncated integer value.

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.291. `__nv_uhadd`

**Prototype:**

```
i32 @__nv_uhadd(i32 %x, i32 %y)
```

**Description:**

Compute average of unsigned input arguments  $x$  and  $y$  as  $(x + y) \gg 1$ , avoiding overflow in the intermediate sum.

**Returns:**

Returns an unsigned integer value representing the unsigned average value of the two inputs.

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.292. `__nv_uint2double_rn`

**Prototype:**

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

**Description:**

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

**Returns:**

Returns converted value.

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.293. `__nv_uint2float_rd`

**Prototype:**

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

**Description:**

Convert the unsigned integer value  $x$  to a single-precision floating point value in round-down (to negative infinity) mode.

**Returns:**

Returns converted value.

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.294. `__nv_uint2float_rn`

### Prototype:

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

### Description:

Convert the unsigned integer value `x` to a single-precision floating point value in round-to-nearest-even mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.295. `__nv_uint2float_ru`

### Prototype:

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

### Description:

Convert the unsigned integer value `x` to a single-precision floating point value in round-up (to positive infinity) mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.296. `__nv_uint2float_rz`

### Prototype:

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

### Description:

Convert the unsigned integer value  $x$  to a single-precision floating point value in round-towards-zero mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.297. `__nv_ull2double_rd`

### Prototype:

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

### Description:

Convert the unsigned 64-bit integer value  $x$  to a double-precision floating point value in round-down (to negative infinity) mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes



## 3.298. `__nv_ull2double_rn`

### Prototype:

```
double @__nv_ull2double_rn(i64 %1)
```

### Description:

Convert the unsigned 64-bit integer value `x` to a double-precision floating point value in round-to-nearest-even mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.299. `__nv_ull2double_ru`

### Prototype:

```
double @__nv_ull2double_ru(i64 %1)
```

### Description:

Convert the unsigned 64-bit integer value `x` to a double-precision floating point value in round-up (to positive infinity) mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.300. `__nv_ull2double_rz`

### Prototype:

```
double @__nv_ull2double_rz(i64 %1)
```

### Description:

Convert the unsigned 64-bit integer value `x` to a double-precision floating point value in round-towards-zero mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.301. `__nv_ull2float_rd`

### Prototype:

```
float @__nv_ull2float_rd(i64 %1)
```

### Description:

Convert the unsigned integer value `x` to a single-precision floating point value in round-down (to negative infinity) mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.302. `__nv_ull2float_rn`

### Prototype:

```
float @__nv_ull2float_rn(i64 %1)
```

### Description:

Convert the unsigned integer value  $x$  to a single-precision floating point value in round-to-nearest-even mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.303. `__nv_ull2float_ru`

### Prototype:

```
float @__nv_ull2float_ru(i64 %1)
```

### Description:

Convert the unsigned integer value  $x$  to a single-precision floating point value in round-up (to positive infinity) mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.304. `__nv_ull2float_rz`

### Prototype:

```
float @__nv_ull2float_rz(i64 %1)
```

### Description:

Convert the unsigned integer value `x` to a single-precision floating point value in round-towards-zero mode.

### Returns:

Returns converted value.

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.305. `__nv_ullmax`

### Prototype:

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

### Description:

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

### Returns:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.306. `__nv_ullmin`

### Prototype:

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

### Description:

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

### Returns:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.307. `__nv_umax`

### Prototype:

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

### Description:

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

### Returns:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.308. `__nv_uamin`

### Prototype:

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

**Description:**

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

**Returns:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.309. `__nv_umul24`

**Prototype:**

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

**Returns:**

Returns the least significant 32 bits of the product  $x * y$ .

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.310. `__nv_umul64hi`

**Prototype:**

```
i64 @__nv_umul64hi(i64 %x, i64 %y)
```

**Description:**

Calculate the most significant 64 bits of the 128-bit product  $x * y$ , where  $x$  and  $y$  are 64-bit unsigned integers.

**Returns:**

Returns the most significant 64 bits of the product  $x * y$ .

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.311. `__nv_umulhi`

**Prototype:**

```
i32 @__nv_umulhi(i32 %x, i32 %y)
```

**Description:**

Calculate the most significant 32 bits of the 64-bit product  $x * y$ , where  $x$  and  $y$  are 32-bit unsigned integers.

**Returns:**

Returns the most significant 32 bits of the product  $x * y$ .

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.312. `__nv_urhadd`

**Prototype:**

```
i32 @__nv_urhadd(i32 %x, i32 %y)
```

**Description:**

Compute average of unsigned input arguments  $x$  and  $y$  as  $(x + y + 1) >> 1$ , avoiding overflow in the intermediate sum.

**Returns:**

Returns an unsigned integer value representing the unsigned rounded average value of the two inputs.

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.313. `__nv_usad`

**Prototype:**

```
i32 @__nv_usad(i32 %x, i32 %y, i32 %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.

**Returns:**

Returns  $|x - y| + z$ .

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.314. `__nv_y0`

**Prototype:**

```
double @__nv_y0(double %x)
```

**Description:**

Calculate the value of the Bessel function of the second kind of order 0 for the input argument  $x$ ,  $Y_0(x)$ .

**Returns:**

Returns the value of the Bessel function of the second kind of order 0.



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

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.315. `__nv_y0f`

**Prototype:**

```
float __nv_y0f(float %x)
```

**Description:**

Calculate the value of the Bessel function of the second kind of order 0 for the input argument  $x$ ,  $Y_0(x)$ .

**Returns:**

Returns the value of the Bessel function of the second kind of order 0.

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

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.316. `__nv_y1`

### Prototype:

```
double @__nv_y1(double %x)
```

### Description:

Calculate the value of the Bessel function of the second kind of order 1 for the input argument  $x$ ,  $Y_1(x)$ .

### Returns:

Returns the value of the Bessel function of the second kind of order 1.

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



### Note:

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

### Library Availability:

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.317. `__nv_y1f`

### Prototype:

```
float @__nv_y1f(float %x)
```

### Description:

Calculate the value of the Bessel function of the second kind of order 1 for the input argument  $x$ ,  $Y_1(x)$ .

**Returns:**

Returns the value of the Bessel function of the second kind of order 1.

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

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.318. `__nv_yn`

**Prototype:**

```
double @__nv_yn(i32 %n, double %x)
```

**Description:**

Calculate the value of the Bessel function of the second kind of order  $n$  for the input argument  $x$ ,  $Y_n(x)$ .

**Returns:**

Returns the value of the Bessel function of the second kind of order  $n$ .

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

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 3.319. `__nv_ynf`

**Prototype:**

```
float __nv_ynf(int32_t n, float x)
```

**Description:**

Calculate the value of the Bessel function of the second kind of order  $n$  for the input argument  $x$ ,  $Y_n(x)$ .

**Returns:**

Returns the value of the Bessel function of the second kind of order  $n$ .

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

**Note:**

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

**Library Availability:**

Compute 2.0: Yes

Compute 3.0: Yes

Compute 3.5: Yes

## 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

© 2013-2022 NVIDIA Corporation & affiliates. All rights reserved.