Libdevice User’s Guide
# Table of Contents

## Chapter 1. Introduction

1.1. What Is libdevice? ........................................................................................................ 1

## Chapter 2. Basic Usage

2.1. Linking with libdevice ................................................................................................... 2

## Chapter 3. Function Reference

3.1. __nv_abs.................................................................................................................................... 4
3.2. __nv_acos.................................................................................................................................. 4
3.3. __nv_acosf............................................................................................................................... 5
3.4. __nv_acosh............................................................................................................................. 5
3.5. __nv_acoshf............................................................................................................................ 6
3.6. __nv_asin................................................................................................................................... 7
3.7. __nv_asinf.................................................................................................................................. 7
3.8. __nv_asinh................................................................................................................................. 8
3.9. __nv_asinhf............................................................................................................................ 8
3.10. __nv_atan................................................................................................................................. 9
3.11. __nv_atan2............................................................................................................................... 9
3.12. __nv_atan2f............................................................................................................................ 10
3.13. __nv_atanf............................................................................................................................... 11
3.14. __nv_atanh............................................................................................................................ 11
3.15. __nv_atanhf........................................................................................................................... 12
3.16. __nv_brev............................................................................................................................... 12
3.17. __nv_brevll............................................................................................................................. 13
3.18. __nv_byte_perm..................................................................................................................... 13
3.19. __nv_cbrt................................................................................................................................ 14
3.20. __nv_cbrtf............................................................................................................................ 14
3.21. __nvceil.................................................................................................................................. 15
3.22. __nvceilf............................................................................................................................... 15
3.23. __nv_clz................................................................................................................................ 16
3.24. __nv_clzll............................................................................................................................... 16
3.25. __nv_copysign....................................................................................................................... 17
3.26. __nv_copysignf..................................................................................................................... 17
3.27. __nv_cos................................................................................................................................ 18
3.28. __nv_cosf............................................................................................................................... 18
3.29. __nv_cosh.............................................................................................................................. 19
3.30. __nv_coshf............................................................................................................................ 19
3.31. __nv_cospi..............................................................................................................................20
3.32. __nv_cospif.............................................................................................................................20
3.33. __nv_dadd_rd..........................................................................................................................21
3.34. __nv_dadd_rn..........................................................................................................................21
3.35. __nv_dadd_ru..........................................................................................................................22
3.36. __nv_dadd_rz..........................................................................................................................23
3.37. __nv_ddiv_rd..........................................................................................................................23
3.38. __nv_ddiv_rn..........................................................................................................................24
3.39. __nv_ddiv_ru..........................................................................................................................24
3.40. __nv_ddiv_rz............................................................................................................................25
3.41. __nv_dmul_rd..........................................................................................................................25
3.42. __nv_dmul_rn..........................................................................................................................26
3.43. __nv_dmul_ru..........................................................................................................................26
3.44. __nv_dmul_rz..........................................................................................................................27
3.45. __nv_double2float_rd.............................................................................................................27
3.46. __nv_double2float_rn.............................................................................................................28
3.47. __nv_double2float_ru.............................................................................................................28
3.48. __nv_double2float_rz.............................................................................................................29
3.49. __nv_double2hiint.................................................................................................................29
3.50. __nv_double2int_rd...............................................................................................................30
3.51. __nv_double2int_rn...............................................................................................................30
3.52. __nv_double2int_ru...............................................................................................................30
3.53. __nv_double2int_rz...............................................................................................................31
3.54. __nv_double2ll_rd..................................................................................................................31
3.55. __nv_double2ll_rn..................................................................................................................32
3.56. __nv_double2ll_ru..................................................................................................................32
3.57. __nv_double2ll_rz..................................................................................................................33
3.58. __nv_double2loint..................................................................................................................33
3.59. __nv_double2uint_rd.............................................................................................................33
3.60. __nv_double2uint_rn.............................................................................................................34
3.61. __nv_double2uint_ru.............................................................................................................34
3.62. __nv_double2uint_rz.............................................................................................................35
3.63. __nv_double2ull_rd...............................................................................................................35
3.64. __nv_double2ull_rn...............................................................................................................36
3.65. __nv_double2ull_ru...............................................................................................................36
3.66. __nv_double2ull_rz...............................................................................................................36
3.67. __nv_double_as_longlong....................................................................................................37
3.68. __nv_drcp_rd.........................................................................................................................37
3.69. __nv_drcp_rn......................................................................................................................... 38
3.70. __nv_drcp_ru......................................................................................................................... 38
3.71. __nv_drcp_rz..........................................................................................................................39
3.72. __nv_dsqrt_rd........................................................................................................................40
3.73. __nv_dsqrt_rn........................................................................................................................40
3.74. __nv_dsqrt_ru........................................................................................................................41
3.75. __nv_dsqrt_rz........................................................................................................................ 41
3.76. __nv_erf..................................................................................................................................42
3.77. __nv_erfc................................................................................................................................42
3.78. __nv_erfcf...............................................................................................................................43
3.79. __nv_erfcinv........................................................................................................................... 43
3.80. __nv_erfcinvf..........................................................................................................................44
3.81. __nv_erfcx..............................................................................................................................45
3.82. __nv_erfcxf.............................................................................................................................45
3.83. __nv_erff............................................................................................................................... 46
3.84. __nv_erfinv.............................................................................................................................46
3.85. __nv_erfinvf............................................................................................................................47
3.86. __nv_exp................................................................................................................................ 48
3.87. __nv_exp10.............................................................................................................................48
3.88. __nv_exp10f............................................................................................................................ 49
3.89. __nv_exp2...............................................................................................................................49
3.90. __nv_exp2f..............................................................................................................................50
3.91. __nv_expf...............................................................................................................................50
3.92. __nv_expm1........................................................................................................................... 51
3.93. __nv_expm1f............................................................................................................................51
3.94. __nv_fabs...............................................................................................................................52
3.95. __nv_fabsf..............................................................................................................................52
3.96. __nv_fadd_rd......................................................................................................................... 53
3.97. __nv_fadd_rn..........................................................................................................................53
3.98. __nv_fadd_ru..........................................................................................................................54
3.99. __nv_fadd_rz........................................................................................................................... 54
3.100. __nv_fast_cosf..................................................................................................................... 55
3.101. __nv_fast_exp10f................................................................................................................. 56
3.102. __nv_fast_expf.......................................................................................................................56
3.103. __nv_fast_fdivdef................................................................................................................ 57
3.104. __nv_fast_log10f.................................................................................................................. 57
3.105. __nv_fast_log2f....................................................................................................................58
3.106. __nv_fast_logf....................................................................................................................... 58
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>__nv_log1p</td>
<td>119</td>
</tr>
<tr>
<td>__nv_log1pf</td>
<td>120</td>
</tr>
<tr>
<td>__nv_log2</td>
<td>120</td>
</tr>
<tr>
<td>__nv_log2f</td>
<td>121</td>
</tr>
<tr>
<td>__nv_logb</td>
<td>121</td>
</tr>
<tr>
<td>__nv_logbf</td>
<td>122</td>
</tr>
<tr>
<td>__nv_logf</td>
<td>123</td>
</tr>
<tr>
<td>__nv_longlong_as_double</td>
<td>123</td>
</tr>
<tr>
<td>__nv_max</td>
<td>124</td>
</tr>
<tr>
<td>__nv_min</td>
<td>124</td>
</tr>
<tr>
<td>__nv_modf</td>
<td>124</td>
</tr>
<tr>
<td>__nv_modff</td>
<td>125</td>
</tr>
<tr>
<td>__nv_mul24</td>
<td>126</td>
</tr>
<tr>
<td>__nv_mul64hi</td>
<td>126</td>
</tr>
<tr>
<td>__nv_mulhi</td>
<td>127</td>
</tr>
<tr>
<td>__nv_nan</td>
<td>127</td>
</tr>
<tr>
<td>__nv_nanf</td>
<td>128</td>
</tr>
<tr>
<td>__nv_nearbyint</td>
<td>128</td>
</tr>
<tr>
<td>__nv_nearbyintf</td>
<td>129</td>
</tr>
<tr>
<td>__nv_nextafter</td>
<td>129</td>
</tr>
<tr>
<td>__nv_nextafterf</td>
<td>130</td>
</tr>
<tr>
<td>__nv_normcdf</td>
<td>130</td>
</tr>
<tr>
<td>__nv_normcddf</td>
<td>131</td>
</tr>
<tr>
<td>__nv_normcdfinv</td>
<td>132</td>
</tr>
<tr>
<td>__nv_normcdfinvf</td>
<td>132</td>
</tr>
<tr>
<td>__nv_popc</td>
<td>133</td>
</tr>
<tr>
<td>__nv_popcll</td>
<td>133</td>
</tr>
<tr>
<td>__nv_pow</td>
<td>134</td>
</tr>
<tr>
<td>__nv_powf</td>
<td>135</td>
</tr>
<tr>
<td>__nv_powi</td>
<td>136</td>
</tr>
<tr>
<td>__nv_powif</td>
<td>137</td>
</tr>
<tr>
<td>__nv_rcbrt</td>
<td>138</td>
</tr>
<tr>
<td>__nv_rcbrtf</td>
<td>138</td>
</tr>
<tr>
<td>__nv_remainder</td>
<td>139</td>
</tr>
<tr>
<td>__nv_remainderf</td>
<td>139</td>
</tr>
<tr>
<td>__nv_remquo</td>
<td>140</td>
</tr>
<tr>
<td>__nv_remquof</td>
<td>141</td>
</tr>
<tr>
<td>__nv_rhadd</td>
<td>141</td>
</tr>
</tbody>
</table>
3.297. __nv_ull2double_rd........................................................................................................... 162
3.298. __nv_ull2double_rn........................................................................................................... 162
3.299. __nv_ull2double_ru........................................................................................................... 162
3.300. __nv_ull2double_rz........................................................................................................... 163
3.301. __nv_ull2float_rd............................................................................................................... 163
3.302. __nv_ull2float_rn............................................................................................................... 164
3.303. __nv_ull2float_ru............................................................................................................... 164
3.304. __nv_ull2float_rz............................................................................................................... 165
3.305. __nv_ullmax....................................................................................................................... 165
3.306. __nv_ullmin........................................................................................................................ 165
3.307. __nv_umax......................................................................................................................... 166
3.308. __nv_umin.......................................................................................................................... 166
3.309. __nv_umul24...................................................................................................................... 167
3.310. __nv_umul64hi................................................................................................................... 167
3.311. __nv_umulhi....................................................................................................................... 168
3.312. __nv_urhadd....................................................................................................................... 169
3.313. __nv_usad.......................................................................................................................... 169
3.314. __nv_y0............................................................................................................................... 169
3.315. __nv_y0f.............................................................................................................................. 170
3.316. __nv_y1............................................................................................................................... 170
3.317. __nv_y1f.............................................................................................................................. 171
3.318. __nv_yn............................................................................................................................... 172
3.319. __nv_ynf............................................................................................................................. 172
List of Tables

Table 1. Supported Reflection Parameters .......................................................................................2
Chapter 1. Introduction

1.1. What Is libdevice?

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

This guide documents both the functions available in libdevice and the basic usage of the library from a compiler writer’s perspective.
Chapter 2. Basic Usage

2.1. Linking with libdevice

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

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ftz</td>
<td>0</td>
<td>preserve denormal values, when performing single-precision floating-point operations</td>
</tr>
<tr>
<td></td>
<td>[default]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>flush denormal values to zero, when performing single-precision floating-point operations</td>
</tr>
<tr>
<td>-prec-div</td>
<td>0</td>
<td>use a faster approximation for single-precision floating-point division and reciprocals</td>
</tr>
<tr>
<td></td>
<td>[default]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>use IEEE round-to-nearest mode for single-precision floating-point division and reciprocals</td>
</tr>
<tr>
<td>-prec-sqrt</td>
<td>0</td>
<td>use a faster approximation for single-precision floating-point square root</td>
</tr>
<tr>
<td></td>
<td>[default]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>use IEEE round-to-nearest mode for single-precision floating-point square root</td>
</tr>
</tbody>
</table>

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

```c
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, π] for x inside [-1, +1].

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

**Note:** For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

### 3.3. __nv_acosf

**Prototype:**

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

### 3.4. __nv_acosh

**Prototype:**

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

**Library Availability:**
Compute 2.0: Yes  
Compute 3.0: Yes  
Compute 3.5: Yes

### 3.5. \(__nv_acoshf\)

**Prototype:**

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**
Compute 2.0: Yes  
Compute 3.0: Yes  
Compute 3.5: Yes
3.6.  __nv_asin

Prototype:

\[
\text{double } @\text{__nv_asin}(\text{double } \%x)
\]

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 \([-1, +1]\).

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

Note:
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

Library Availability:
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.7.  __nv_asinf

Prototype:

\[
\text{float } @\text{__nv_asinf}(\text{float } \%x)
\]

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 \([-1, +1]\).

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

Note:
3.8. __nv_asinh

Prototype:

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

Library Availability:
- Compute 2.0: Yes
- Compute 3.0: Yes
- Compute 3.5: Yes

3.9. __nv_asinhf

Prototype:

```c
float __nv_asinhf(float x)
```

Description:
Calculate the arc hyperbolic sine of the input argument \( x \).

Returns:
3.10. __nv_asinh

Prototype:

```c
double __nv_asinh(double x)
```

Description:
Calculate the inverse hyperbolic sine of the input argument `x`.

Returns:
Result will be in radians, in the interval [-1, +1].

Note:
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

Library Availability:
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.10. __nv_atan

Prototype:

```c
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 [-\(\pi/2\), +\(\pi/2\)].

Note:
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

Library Availability:
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.11. __nv_atan2

Prototype:

```c
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 \([- \pi, + \pi]\).

- \( \text{__nv_atan2}(0, 1) \) returns +0.

**Note:**

For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

**Library Availability:**

- Compute 2.0: Yes
- Compute 3.0: Yes
- Compute 3.5: Yes

### 3.12. \text{__nv_atan2f}

**Prototype:**

```
float \text{__nv_atan2f}(float \text{x, float \text{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 \([- \pi, + \pi]\).

- \( \text{__nv_atan2f}(0, 1) \) returns +0.

**Note:**

For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**

- Compute 2.0: Yes
- Compute 3.0: Yes
- Compute 3.5: Yes
3.13. __nv_atanf

Prototype:

```c
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 \([-\pi/2, +\pi/2]\).
- \( \text{__nv_atan}(0) \) returns +0.

Note:
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

Library Availability:
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.14. __nv_atanh

Prototype:

```c
double __nv_atanh(double x)
```

Description:
Calculate the arc hyperbolic tangent of the input argument \( x \).

Returns:
- \( \text{__nv_atanh}(\pm 0) \) returns \( \pm 0 \).
- \( \text{__nv_atanh}(\pm 1) \) returns \( \pm \infty \).
- \( \text{__nv_atanh}(x) \) returns NaN for \( x \) outside interval \([-1, 1]\).

Note:
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.
3.15. __nv_atanhf

Prototype:

```c
float __nv_atanhf(float %x)
```

Description:
Calculate the arc hyperbolic tangent of the input argument \( x \).

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

Note:
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

Library Availability:
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.16. __nv_brev

Prototype:

```c
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_bревll

**Prototype:**

```
i64 __nv_bревll(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>
```

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>
```

**Returns:**
The returned value r is computed to be: \( \text{result}[n] := \text{input}[\text{selector}[n]] \) where \( \text{result}[n] \) is the nth byte of r.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

### 3.19. \texttt{__nv_cbrt}

**Prototype:**
```
double @__nv_cbrt(double %x)
```

**Description:**
Calculate the cube root of \( x \), \( x^{1/3} \).

**Returns:**
Returns \( x^{1/3} \).
- \( \texttt{__nv_cbrt}(\pm 0) \) returns \( \pm 0 \).
- \( \texttt{__nv_cbrt}(\pm \infty) \) returns \( \pm \infty \).

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

### 3.20. \texttt{__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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**
- Compute 2.0: Yes
- Compute 3.0: Yes
- Compute 3.5: Yes

### 3.21. `__nv_ceil`

**Prototype:**

```c
double __nv_ceil(double x)
```

**Description:**
Compute the smallest integer value not less than $x$.

**Returns:**
Returns $\lfloor x \rfloor$ 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:**

```c
float __nv_ceilf(float x)
```

**Description:**
Compute the smallest integer value not less than \( x \).

**Returns:**

Returns \([x]\) expressed as a floating-point number.

- \( \text{__nv_ceilf}(\pm 0) \) returns \( \pm 0 \).
- \( \text{__nv_ceilf}(\pm \infty) \) returns \( \pm \infty \).

**Library Availability:**

Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

### 3.23. \( \text{__nv_clz} \)

**Prototype:**

\[
\text{int32} \ @\text{__nv_clz(int32} \ %x)\]

**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. \( \text{__nv_clzll} \)

** Prototype:**

\[
\text{int32} \ @\text{__nv_clzll(int64} \ %x)\]

**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:**

```c
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:**

```c
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:

```c
double __nv_cos(double %x)
```

Description:
Calculate the cosine of the input argument \( x \) (measured in radians).

Returns:
- \( __nv_cos(\pm 0) \) returns 1.
- \( __nv_cos(\pm \infty) \) returns NaN.

Note:
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

Library Availability:
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.28.  __nv_cosf

Prototype:

```c
float __nv_cosf(float %x)
```

Description:
Calculate the cosine of the input argument \( x \) (measured in radians).

Returns:
- \( __nv_cosf(\pm 0) \) returns 1.
- \( __nv_cosf(\pm \infty) \) returns NaN.

Note:
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.
3.29. **__nv_cosh**

**Prototype:**

```c
double __nv_cosh(double %x)
```

**Description:**
Calculate the hyperbolic cosine of the input argument \( x \).

**Returns:**
- \( \text{__nv_cosh}(0) \) returns 1.
- \( \text{__nv_cosh}(\pm \infty) \) returns \( +\infty \).

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

**Library Availability:**
- Compute 2.0: Yes
- Compute 3.0: Yes
- Compute 3.5: Yes

---

3.30. **__nv_coshf**

**Prototype:**

```c
float __nv_coshf(float %x)
```

**Description:**
Calculate the hyperbolic cosine of the input argument \( x \).

**Returns:**
- \( \text{__nv_coshf}(0) \) returns 1.
function returns +∞.

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.31. **__nv_cospis**

**Prototype:**
```c
double __nv_cospis(double %x)
```

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

**Returns:**
- __nv_cospis($\pm 0$) returns 1.
- __nv_cospis($\pm \infty$) returns NaN.

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.32. **__nv_cospisf**

**Prototype:**
```c
float __nv_cospisf(float %x)
```

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

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

Note:
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

Library Availability:
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.33. `__nv_dadd_rd`

Prototype:

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.
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:

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.
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:**
```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.
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:**

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

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:**

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

Requires compute capability >= 2.0.

**Library Availability:**
3.38. __nv_ddiv_rn

**Prototype:**

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.
Requires compute capability >= 2.0.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.39. __nv_ddiv_ru

**Prototype:**

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.
3.40. \texttt{__nv_ddiv_rz}

**Prototype:**

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

Requires compute capability $\geq 2.0$.

**Library Availability:**

- Compute 2.0: Yes
- Compute 3.0: Yes
- Compute 3.5: Yes

3.41. \texttt{__nv_dmul_rd}

**Prototype:**

```c
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 \( x \times y \).

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.
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:**
```c
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 \( x \times y \).

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.
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:**
```c
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 $x \times y$.

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.
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:**
```c
double __nv_dmul_rz(double %x, double %y)
```

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

**Returns:**
Returns $x \times y$.

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.
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 %d)`

**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 %d)`

**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 %d)`

**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:**

```c
float __nv_double2float_rz(double %d)
```

**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:**

```c
int __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.
3.55.  __nv_double2ll_rn

Prototype:

```c
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:

```c
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:

```c
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:

```c
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:

```c
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:**

```c
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:**

```c
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.
3.62.  __nv_double2uint_rz

Prototype:

```c
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:

```c
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:**

```c
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:**

```c
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:**

```c
i64 __nv_double2ull_rz(double %f)
```
**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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.
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:
```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.
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:
```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.
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:
```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.
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:**

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

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:**

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

Requires compute capability \( \geq 2.0 \).

**Library Availability:**
3.74. __nv_dsqrt_ru

Prototype:

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.
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:

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.
Requires compute capability \( \geq 2.0 \).
3.76. \texttt{\_nv\_erf}

Prototype:

\begin{verbatim}
double __nv_erf(double \%x)
\end{verbatim}

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:

- \( \_nv\_erf( \pm 0 ) \) returns \( \pm 0 \).
- \( \_nv\_erf( \pm \infty ) \) returns \( \pm 1 \).

Note:

For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

Library Availability:

- Compute 2.0: Yes
- Compute 3.0: Yes
- Compute 3.5: Yes

3.77. \texttt{\_nv\_erfc}

Prototype:

\begin{verbatim}
double __nv_erfc(double \%x)
\end{verbatim}

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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

**Library Availability:**
- Compute 2.0: Yes
- Compute 3.0: Yes
- Compute 3.5: Yes

#### 3.78. __nv_erfcf

**Prototype:**
```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**
- Compute 2.0: Yes
- Compute 3.0: Yes
- Compute 3.5: Yes

#### 3.79. __nv_erfcinv

**Prototype:**
```c
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 \( y = \text{erfc}(x) \), for \( 0 \leq y \leq 2 \), and \( -\infty \leq x \leq \infty \).

**Returns:**
- \( \text{__nv_erfcinv}(0) \) returns \(+\infty\).
- \( \text{__nv_erfcinv}(2) \) returns \(-\infty\).

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

**Library Availability:**
- Compute 2.0: Yes
- Compute 3.0: Yes
- Compute 3.5: Yes

### 3.80. \texttt{__nv_erfcinvf}

**Prototype:**
```c
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 \( y = \text{erfc}(x) \), for \( 0 \leq y \leq 2 \), and \( -\infty \leq x \leq \infty \).

**Returns:**
- \( \text{__nv_erfcinvf}(0) \) returns \(+\infty\).
- \( \text{__nv_erfcinvf}(2) \) returns \(-\infty\).

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**
- Compute 2.0: Yes
- Compute 3.0: Yes
3.81. **__nv_erfcx**

**Prototype:**

```c
double __nv_erfcx(double %x)
```

**Description:**

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

**Returns:**

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

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

**Library Availability:**

- Compute 2.0: Yes
- Compute 3.0: Yes
- Compute 3.5: Yes

3.82. **__nv_erfcxf**

**Prototype:**

```c
float __nv_erfcxf(float %x)
```

**Description:**

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

**Returns:**

- \( \text{__nv_erfcxf}( -\infty ) \) returns +\( \infty \)
- \( \text{__nv_erfcxf}( +\infty ) \) returns +0
Function Reference

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

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

### 3.83. __nv_erff

**Prototype:**

```c
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 ±0.
- __nv_erff(±∞) returns ±1.

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

### 3.84. __nv_erfinv

**Prototype:**

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

Library Availability:
- Compute 2.0: Yes
- Compute 3.0: Yes
- Compute 3.5: Yes

3.85. __nv_erfinvf

Prototype:
```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

Library Availability:
- Compute 2.0: Yes
- Compute 3.0: Yes
3.86. **__nv_exp**

**Prototype:**

```c
double __nv_exp(double %x)
```

**Description:**

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

**Returns:**

Returns $e^x$.

**Note:**

For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

**Library Availability:**

Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.87. **__nv_exp10**

**Prototype:**

```c
double __nv_exp10(double %x)
```

**Description:**

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

**Returns:**

Returns $10^x$.

**Note:**

For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

**Library Availability:**

Compute 2.0: Yes
Compute 3.0: Yes
3.88.  __nv_exp10f

Prototype:

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

Description:
Calculate the base 10 exponential of the input argument \( x \).

Returns:
Returns \( 10^x \).

Note:
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

Library Availability:
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.89.  __nv_exp2

Prototype:

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

Description:
Calculate the base 2 exponential of the input argument \( x \).

Returns:
Returns \( 2^x \).

Note:
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

Library Availability:
Compute 2.0: Yes
3.90. __nv_exp2f

Prototype:

```c
float __nv_exp2f(float x)
```

Description:
Calculate the base 2 exponential of the input argument \( x \).

Returns:
Returns \( 2^x \).

Note:
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

Library Availability:
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.91. __nv_expf

Prototype:

```c
float __nv_expf(float x)
```

Description:
Calculate the base \( e \) exponential of the input argument \( x \).

Returns:
Returns \( e^x \).

Note:
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

Library Availability:
Compute 2.0: Yes
3.92. __nv_expm1

Prototype:

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

Library Availability:
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.93. __nv_expm1f

Prototype:

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

Library Availability:
Compute 2.0: Yes
3.94. __nv_fabs

**Prototype:**

```c
double __nv_fabs(double %f)
```

**Description:**

Calculate the absolute value of the input argument \( x \).

**Returns:**

Returns the absolute value of the input argument.

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

**Note:**

For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

**Library Availability:**

Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.95. __nv_fabsf

**Prototype:**

```c
float __nv_fabsf(float %f)
```

**Description:**

Calculate the absolute value of the input argument \( x \).

**Returns:**

Returns the absolute value of the input argument.

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

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

### 3.96. __nv_fadd_rd

**Prototype:**

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

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:**

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.
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:
```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.
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:
```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.
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 for this function see the CUDA C++ Programming Guide, Appendix E.2, Table 9.
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:

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.2, Table 9.
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:

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.2, Table 9.
Most input and output values around denormal range are flushed to sign preserving 0.0.

Library Availability:
3.103. **__nv_fast_fdividef**

**Prototype:**

```c
float __nv_fast_fdividef(float x, float y)
```

**Description:**

Calculate the fast approximate division of \( x \) by \( y \).

**Returns:**

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

**Note:**

For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.2, Table 9.

**Library Availability:**

- Compute 2.0: Yes
- Compute 3.0: Yes
- Compute 3.5: Yes

3.104. **__nv_fast_log10f**

**Prototype:**

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.2, Table 9.
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:**

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.2, Table 9.
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:**

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.2, Table 9.
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:**
```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.2, Table 9.
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:

```c
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, \( \text{sptr} \), and, respectively, third argument, \( \text{cptr} \).

Returns:
- none

Note:
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.2, Table 9.
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:

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.2, Table 9.
Input and output in the denormal range is flushed to sign preserving 0.0.
**3.110. __nv_fast_tanf**

**Prototype:**

\[ \text{float } \_\_\text{nv\_fast\_tanf}(\text{float } \%x) \]

**Description:**
Calculation the fast approximate tangent of the input argument \( x \), measured in radians.

**Returns:**
Returns the approximate tangent of \( x \).

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.2, Table 9. The result is computed as the fast divide of \( \_\_\text{nv\_sinf}() \) by \( \_\_\text{nv\_cosf}() \). 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:**

\[ \text{double } \_\_\text{nv\_fdim}(\text{double } \%x, \text{double } \%y) \]

**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 \( y \).

- \( \_\_\text{nv\_fdim}(x, y) \) returns \( x - y \) if \( x > y \).
__nv_fdim(x, y) returns +0 if $x \leq y$.

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

### 3.112. __nv_fdimf

**Prototype:**

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

### 3.113. __nv_fdiv_rd

**Prototype:**

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

Library Availability:
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.114. __nv_fdiv_rn

Prototype:

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

Library Availability:
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.115. __nv_fdiv_ru

Prototype:

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

Library Availability:
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.116. __nv_fdiv_rz

Prototype:
```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

Library Availability:
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.117. __nv_ffs

Prototype:
```c
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:**
```c
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:**
```c
i32 __nv_finitef(float x)
```

**Description:**
Determine whether the floating-point value x is a finite value.
3.120. __nv_float2half_rn

**Prototype:**

```c
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:**

```c
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
3.122. __nv_float2int_rn

Prototype:

```c
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:

```c
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
### __nv_float2int_rz

**Prototype:**

```c
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

### __nv_float2ll_rd

**Prototype:**

```c
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

### __nv_float2ll_rn

**Prototype:**

```c
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:**

```c
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:**

```c
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:

```c
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:

```c
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:

```c
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:

```c
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:

```c
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:**

```c
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:**

```c
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:**

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

### 3.139. `__nv_floorf`

**Prototype:**
```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
3.140. __nv_fma

Prototype:

```c
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.

- \( \_\_nv\_fma( \pm \infty, \pm 0, z) \) returns NaN.
- \( \_\_nv\_fma( \pm 0, \pm \infty, z) \) returns NaN.
- \( \_\_nv\_fma(x, y, -\infty) \) returns NaN if \( x \times y \) is an exact \( +\infty \).
- \( \_\_nv\_fma(x, y, +\infty) \) returns NaN if \( x \times y \) is an exact \( -\infty \).

Note:
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

Library Availability:
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.141. __nv_fma_rd

Prototype:

```c
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.
Function Reference

Libdevice User’s Guide
Part 000  vRelease Version  |  76

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

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes
3.143. __nv_fma_ru

Prototype:

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

Note:
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

Library Availability:
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.144. __nv_fma_rz

Prototype:

```c
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($\pm \infty$, $\pm 0$, $z$) returns NaN.


**__nv_fma_rz(±0, ±∞, z) returns NaN.**

**__nv_fma_rz(x, y, −∞) returns NaN if x y is an exact +∞**

**__nv_fma_rz(x, y, +∞) returns NaN if x y is an exact −∞**

**Note:**

For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

**Library Availability:**

Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

---

**3.145. __nv_fmaf**

**Prototype:**

```c
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 y is an exact +∞.**
- **__nv_fmaf(x, y, +∞) returns NaN if x y is an exact −∞.**

**Note:**

For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**

Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes
3.146. __nv_fmaf_rd

Prototype:

```c
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.

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

Note:

For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

Library Availability:

Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.147. __nv_fmaf_rn

Prototype:

```c
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.

- \(\text{__nv_fmaf_rn} (\pm \infty, \pm 0, z)\) 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_rn(x, y, +∞) returns NaN if \( x \times y \) is an exact -∞.

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

### 3.148. __nv_fmaf_ru

**Prototype:**

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

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes
3.149. __nv_fmaf_rz

Prototype:

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

Library Availability:
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.150. __nv_fmax

Prototype:

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

### 3.151. __nv_fmaxf

**Prototype:**

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes
3.152. __nv_fmin

Prototype:

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

Library Availability:
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.153. __nv_fminf

Prototype:

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

### 3.154. __nv_fmod

**Prototype:**
```c
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 \( x / y \).
- \( __nv_fmod(\pm0, y) \) returns \( \pm0 \) if \( y \) is not zero.
- \( __nv_fmod(x, \pm\infty) \) returns \( x \) if \( x \) is finite.
- \( __nv_fmod(x, y) \) returns NaN if \( x \) is \( \pm\infty \) or \( y \) is zero.
- If either argument is NaN, NaN is returned.

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes
3.155. **__nv_fmodf**

**Prototype:**

```c
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 \( x - n \times y \), 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 \( x / y \).
- \( __nv_fmodf(\pm 0, y) \) returns \( \pm 0 \) if \( y \) is not zero.
- \( __nv_fmodf(x, \pm \infty) \) returns \( x \) if \( x \) is finite.
- \( __nv_fmodf(x, y) \) returns NaN if \( x \) is \( \pm \infty \) or \( y \) is zero.
- If either argument is NaN, NaN is returned.

**Note:**

For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**

- Compute 2.0: Yes
- Compute 3.0: Yes
- Compute 3.5: Yes

3.156. **__nv_fmul_rd**

**Prototype:**

```c
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 \( x \times y \).

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.
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:**

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

**Description:**
Compute the product of \( x \) and \( y \) in round-to-nearest-even mode.

**Returns:**
Returns \( x \times y \).

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.
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:**

```c
float @__nv_fmul_ru(float %x, float %y)
```

**Description:**

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

**Returns:**

Returns $x \times y$.

**Note:**

For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

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_fmul_rz**

**Prototype:**

```c
float __nv_fmul_rz(float x, float y)
```

**Description:**

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

**Returns:**

Returns $x \times y$.

**Note:**

For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

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:**

```c
float __nv_frcp_rd(float x)
```
Function Reference

**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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

### 3.161. __nv_frcp_rn

**Prototype:**

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes
3.162. __nv_frcp_ru

**Prototype:**

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**

Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.163. __nv_frcp_rz

**Prototype:**

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**

Compute 2.0: Yes
Compute 3.0: Yes
3.164. __nv_frexp

Prototype:

```c
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 \( m \).

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

Note:
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

Library Availability:
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.165. __nv_frexpf

Prototype:

```c
float __nv_frexpf(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 \( \text{nptr} \) points.

**Returns:**

Returns the fractional component \( m \).

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

**Note:**

For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**

Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

### 3.166. \text{__nv_frsqrt_rn}

**Prototype:**

```cpp
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**

Compute 2.0: Yes
Compute 3.0: Yes
**3.167. __nv_fsqrt_rd**

**Prototype:**

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**

Compute 2.0: Yes  
Compute 3.0: Yes  
Compute 3.5: Yes

---

**3.168. __nv_fsqrt_rn**

**Prototype:**

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**

Compute 2.0: Yes
3.169. __nv_fsqrt_ru

Prototype:

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

Library Availability:
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.170. __nv_fsqrt_rz

Prototype:

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

Library Availability:
3.171. __nv_fsub_rd

Prototype:

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

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:

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.
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:**

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

### 3.174. `__nv_fsub_rz`

**Prototype:**

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

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:**

```c
int32_t __nv_hadd(int32_t x, int32_t y)
```

**Description:**
Compute average of signed input arguments \( x \) and \( y \) as \( \frac{x + y}{2} \), 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:**

```c
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:
```c
double __nv_hiloint2double(i32 %x, i32 %y)
```

Description:
Reinterpret the integer value of \texttt{hi} as the high 32 bits of a double-precision floating point value and the integer value of \texttt{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:
```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

**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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.180. **__nvilogb**

**Prototype:**

```
i32 __nvilogb(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) returns INT_MIN if the correct value is less than INT_MIN.

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

### 3.181. __nv_ilogbf

**Prototype:**

```c
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) returns INT_MIN if the correct value is less than INT_MIN.

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes
### 3.182. __nv_int2double_rn

**Prototype:**

```c
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:**

```c
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:**

```c
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:**
```c
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:**
```c
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.
3.187. **__nv_int_as_float**

**Prototype:**

```c
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_isfinitelyed**

**Prototype:**

```c
i32 __nv_isfinitelyed(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:

```c
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:

```c
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:

```c
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:**
```c
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:**
```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

### 3.194. __nv_j0f

**Prototype:**

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

### 3.195. __nv_j1

**Prototype:**

```c
float __nv_j1(float x)
```
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.196. __nv_j1f

**Prototype:**

```c
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:**
3.197. __nv_jn

Prototype:

```
double __nv_jn(int 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 \( n < 0 \).
- \( \_nv\_jn(n, +\infty) \) returns +0.

Note:

For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

Library Availability:

Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.198. __nv_jnf

Prototype:

```
float __nv_jnf(int 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 \( n \).

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

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**
- Compute 2.0: Yes
- Compute 3.0: Yes
- Compute 3.5: Yes

### 3.199. \text{__nv_ldexp}

**Prototype:**
```
double @__nv_ldexp(double %x, i32 %y)
```

**Description:**
Calculate the value of \( x \cdot 2^{\text{exp}} \) of the input arguments \( x \) and \( \text{exp} \).

**Returns:**
- \( \text{__nv_ldexp}(x) \) returns \( \pm \infty \) if the correctly calculated value is outside the double floating point range.

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

**Library Availability:**
- Compute 2.0: Yes
- Compute 3.0: Yes
- Compute 3.5: Yes
3.200. __nv_ldexpf

Prototype:

```c
float __nv_ldexpf(float x, int y)
```

Description:
Calculate the value of \( x \cdot 2^\text{exp} \) of the input arguments \( x \) and \( \text{exp} \).

Returns:
- __nv_ldexpf(x) returns ±∞ if the correctly calculated value is outside the double floating point range.

Note:
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

Library Availability:
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.201. __nv_lgamma

Prototype:

```c
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 ±∞ if the correctly calculated value is outside the double floating point range.
- __nv_lgamma(x) returns +∞ if \( x \leq 0 \).
- __nv_lgamma(-∞) returns -∞.
__nv_lgamma( +∞) returns +∞.

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

### 3.202. __nv_lgammaf

**Prototype:**

```c
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 ±∞ if the correctly calculated value is outside the double floating point range.
- __nv_lgammaf\([x]\) returns +∞ if \( x \leq 0 \).
- __nv_lgammaf\([-\infty]\) returns -∞.
- __nv_lgammaf\([+\infty]\) returns +∞.

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes
3.203. __nv_ll2double_rd

Prototype:

```c
double __nv_ll2double_rd(i64 %l)
```

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:

```c
double __nv_ll2double_rn(i64 %l)
```

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:

```c
double __nv_ll2double_ru(i64 %l)
```
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:
```c
double __nv_ll2double_rz(i64 %l)
```

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:
```c
float __nv_ll2float_rd(i64 %l)
```

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:

```c
float __nv_ll2float_rn(i64 %l)
```

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:

```c
float __nv_ll2float_ru(i64 %l)
```

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:**

```c
float __nv_ll2float_rz(i64 %l)
```

**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:**

```c
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:**

```c
i64 __nv_llmax(i64 %x, i64 %y)
```
**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.
### 3.215. __nv_llrintf

**Prototype:**

```c
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:**

```c
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
3.217. __nv_llroundf

Prototype:

```c
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 lllrint().

Library Availability:
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.218. __nv_log

Prototype:

```c
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(\( +\infty \)) returns +\( \infty \).

Note:
3.219. __nv_log10

Prototype:

```c
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(+\infty) returns $+\infty$.

Note:
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

Library Availability:
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.220. __nv_log10f

Prototype:

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

Library Availability:
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

### 3.221. __nv_log1p

**Prototype:**

```c
double __nv_log1p(double %x)
```

**Description:**

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

**Returns:**

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

Note:
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

Library Availability:
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes
3.222. __nv_log1pf

Prototype:

```c
float __nv_log1pf(float x)
```

Description:

Calculate the value of $\log_2(1 + x)$ of the input argument $x$.

Returns:

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

Note:

For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

Library Availability:

Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.223. __nv_log2

Prototype:

```c
double __nv_log2(double x)
```

Description:

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

Returns:

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

Function Reference

__nv_log2( +∞ ) returns +∞.

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

### 3.224. __nv_log2f

**Prototype:**

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

### 3.225. __nv_logb

**Prototype:**

```c
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 ± \infty` returns $+\infty$

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

### 3.226. __nv_logbf

**Prototype:**
```c
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 ± \infty` returns $+\infty$

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes
3.227. __nv_logf

Prototype:

```c
float@g__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(+$\infty$) returns $+\infty$

Note:
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

Library Availability:
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.228. __nv_longlong_as_double

Prototype:

```c
double@g__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
3.229. __nv_max

Prototype:

```c
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:

```c
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:

```c
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:**

- $\texttt{__nv\_modf(} x, \texttt{iptr)}$ returns a result with the same sign as $x$.
- $\texttt{__nv\_modf(} \pm \infty, \texttt{iptr)}$ returns $\pm 0$ and stores $\pm \infty$ in the object pointed to by $iptr$.
- $\texttt{__nv\_modf(} \text{NaN}, \texttt{iptr)}$ stores a NaN in the object pointed to by $iptr$ and returns a NaN.

**Note:**

For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

**Library Availability:**

Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

### 3.232. \texttt{__nv\_modff}

**Prototype:**

```c
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:**

- $\texttt{__nv\_modff(} x, \texttt{iptr)}$ returns a result with the same sign as $x$.
- $\texttt{__nv\_modff(} \pm \infty, \texttt{iptr)}$ returns $\pm 0$ and stores $\pm \infty$ in the object pointed to by $iptr$.
- $\texttt{__nv\_modff(} \text{NaN}, \texttt{iptr)}$ stores a NaN in the object pointed to by $iptr$ and returns a NaN.

**Note:**

For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**

Compute 2.0: Yes
Compute 3.0: 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 \( x \times y \).

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 \( x \times y \), where \( x \) and \( y \) are 64-bit integers.

Returns:
Returns the most significant 64 bits of the product \( x \times y \).

Library Availability:
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes
3.235. **__nv_mulhi**

**Prototype:**

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

**Description:**
Calculate the most significant 32 bits of the 64-bit product \(x \times y\), where \(x\) and \(y\) are 32-bit integers.

**Returns:**
Returns the most significant 32 bits of the product \(x \times y\).

**Library Availability:**
Compute 2.0: Yes  
Compute 3.0: Yes  
Compute 3.5: Yes

---

3.236. **__nv_nan**

**Prototype:**

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

**Description:**
Return a representation of a quiet NaN. Argument \(\text{tagp}\) selects one of the possible representations.

**Returns:**
- \(\text{__nv_nan(tagp)}\) returns NaN.

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

**Library Availability:**
Compute 2.0: Yes  
Compute 3.0: Yes  
Compute 3.5: Yes
3.237. __nv_nanf

Prototype:

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

Library Availability:
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.238. __nv_nearbyint

Prototype:

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

Library Availability:
3.239. **__nv_nearbyintf**

**Prototype:**

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**
- Compute 2.0: Yes
- Compute 3.0: Yes
- Compute 3.5: Yes

3.240. **__nv_nextafter**

**Prototype:**

```c
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(±∞, y) returns ±∞.

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

### 3.241. __nv_nextafterf

**Prototype:**

```c
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(±∞, y) returns ±∞.

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

### 3.242. __nv_normcdf

**Prototype:**

```c
double __nv_normcdf(double x)
```

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

**Returns:**
- \( \texttt{__nv_normcdf( +\infty )} \) returns 1
- \( \texttt{__nv_normcdf( -\infty )} \) returns +0

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

**Library Availability:**
- Compute 2.0: Yes
- Compute 3.0: Yes
- Compute 3.5: Yes

### 3.243. \texttt{__nv_normcdff}

**Prototype:**

```c
float @__nv_normcdff(float %x)
```

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

**Returns:**
- \( \texttt{__nv_normcdff( +\infty )} \) returns 1
- \( \texttt{__nv_normcdff( -\infty )} \) returns +0

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**
- Compute 2.0: Yes
- Compute 3.0: Yes
- Compute 3.5: Yes
3.244. __nv_normcdfinv

Prototype:

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

Library Availability:
- Compute 2.0: Yes
- Compute 3.0: Yes
- Compute 3.5: Yes

3.245. __nv_normcdfinvf

Prototype:

```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**
Compute 2.0: Yes  
Compute 3.0: Yes  
Compute 3.5: Yes

### 3.246. __nv_popc

**Prototype:**

```c
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_popcll

**Prototype:**

```c
i32 __nv_popcll(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:**
### 3.248. __nv_pow

**Prototype:**

```c
double __nv_pow(double x, double y)
```

**Description:**

Calculate the value of `x` to the power of `y`.

**Returns:**

- `__nv_pow(x, ±0, y)` returns `±∞` for `y` an integer less than 0.
- `__nv_pow(x, ±0, y)` returns `±0` for `y` an odd integer greater than 0.
- `__nv_pow(x, ±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 `+∞` for `|x| < 1`.
- `__nv_pow(x, −∞)` returns `+0` for `|x| > 1`.
- `__nv_pow(x, +∞)` returns `+0` for `|x| < 1`.
- `__nv_pow(x, +∞)` returns `+∞` 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 `−∞` for `y` an odd integer greater than 0.
- `__nv_pow(−∞, y)` returns `+∞` for `y > 0` and not an odd integer.
- `__nv_pow(+∞, y)` returns `+0` for `y < 0`.
- `__nv_pow(+∞, y)` returns `+∞` for `y > 0`.

**Note:**

For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

**Library Availability:**

- Compute 2.0: Yes
- Compute 3.0: Yes
- Compute 3.5: Yes
3.249. __nv_powf

Prototype:

```c
float __nv_powf(float %x, float %y)
```

Description:
Calculate the value of \(x\) to the power of \(y\).

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

Note:
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

Library Availability:
- Compute 2.0: Yes
- Compute 3.0: Yes
3.250. __nv_powi

Prototype:

```c
double __nv_powi(double %x, i32 %y)
```

Description:
Calculate the value of \( x \) to the power of \( y \)

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

Note:
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

Library Availability:
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes
3.251. __nv_powif

Prototype:

```c
float __nv_powif(float x, int32 y)
```

Description:
Calculate the value of $x$ to the power of $y$.

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

Note:
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

Library Availability:
- Compute 2.0: Yes
- Compute 3.0: Yes
- Compute 3.5: Yes
3.252. \texttt{__nv_rcbrt}

**Prototype:**

\begin{verbatim}
double __nv_rcbrt(double %x)
\end{verbatim}

**Description:**

Calculate reciprocal cube root function of \( x \)

**Returns:**

- \( \texttt{__nv_rcbrt(\pm 0)} \) returns \( \pm \infty \).
- \( \texttt{__nv_rcbrt(\pm \infty)} \) returns \( \pm 0 \).

**Note:**

For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

**Library Availability:**

Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.253. \texttt{__nv_rcbrtf}

**Prototype:**

\begin{verbatim}
float __nv_rcbrtf(float %x)
\end{verbatim}

**Description:**

Calculate reciprocal cube root function of \( x \)

**Returns:**

- \( \texttt{__nv_rcbrtf(\pm 0)} \) returns \( \pm \infty \).
- \( \texttt{__nv_rcbrtf(\pm \infty)} \) returns \( \pm 0 \).

**Note:**

For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**
3.254. __nv_remainder

Prototype:

\[
\text{double } \text{__nv_remainder}(\text{double } \%x, \text{double } \%y)
\]

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 \( |n - \frac{x}{y}| = \frac{1}{2} \), the even \( n \) value is chosen.

Returns:

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

Note:
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

Library Availability:
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.255. __nv_remainderf

Prototype:

\[
\text{float } \text{__nv_remainderf}(\text{float } \%x, \text{float } \%y)
\]

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 \( |n - \frac{x}{y}| = \frac{1}{2} \), the even \( n \) value is chosen.
Returns:

- \(__nv\_remainderf(x, 0)\) returns NaN.
- \(__nv\_remainderf(\pm \infty, y)\) returns NaN.
- \(__nv\_remainderf(x, \pm \infty)\) returns x for finite x.

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

### 3.256. \(__nv\_remquo\)

**Prototype:**

```c
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.
- \(__nv\_remquo(\pm \infty, y, quo)\) returns NaN.
- \(__nv\_remquo(x, \pm \infty, quo)\) returns x.

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes
3.257. **__nv_remquof**

**Prototype:**

```c
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 `x` 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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**

- Compute 2.0: Yes
- Compute 3.0: Yes
- Compute 3.5: Yes

3.258. **__nv_rhadd**

**Prototype:**

```c
i32 __nv_rhadd(i32 %x, i32 %y)
```

**Description:**

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

**Returns:**

Returns a signed integer value representing the signed rounded average value of the two inputs.
Library Availability:
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.259. __nv_rint

Prototype:
```c
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:
```c
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:

```c
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:

```c
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
3.263. __nv_rsqrts

Prototype:

```c
double __nv_rsqrts(double %x)
```

Description:

Calculate the reciprocal of the nonnegative square root of \( x \), \( \frac{1}{\sqrt{x}} \).

Returns:

Returns \( \frac{1}{\sqrt{x}} \).

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

Note:

For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

Library Availability:

- Compute 2.0: Yes
- Compute 3.0: Yes
- Compute 3.5: Yes

3.264. __nv_rsqrtsf

Prototype:

```c
float __nv_rsqrtsf(float %x)
```

Description:

Calculate the reciprocal of the nonnegative square root of \( x \), \( \frac{1}{\sqrt{x}} \).

Returns:

Returns \( \frac{1}{\sqrt{x}} \).

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

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

### 3.265. __nv_sad

**Prototype:**
```c
i32 __nv_sad(i32 %x, i32 %y, i32 %z)
```

**Description:**
Calculate \(|x - y| + z\), the 32-bit sum of the third argument z plus 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:**
```c
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:**

```c
double __nv_scalbn(double %x, i32 %y)
```

**Description:**
Scale \( x \) by \( 2^n \) by efficient manipulation of the floating-point exponent.

**Returns:**
- Returns \( x \times 2^n \).
- \( __nv_scalbn(\pm 0, n) \) returns \( \pm 0 \).
- \( __nv_scalbn(x, 0) \) returns \( x \).
- \( __nv_scalbn(\pm \infty, n) \) returns \( \pm \infty \).

**Library Availability:**
- Compute 2.0: Yes
- Compute 3.0: Yes
- Compute 3.5: Yes

### 3.268. __nv_scalbnf

**Prototype:**

```c
float __nv_scalbnf(float %x, i32 %y)
```

**Description:**
Scale \( x \) by \( 2^n \) by efficient manipulation of the floating-point exponent.

**Returns:**

Returns $x \times 2^n$.

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

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

### 3.269. __nv_signbitd

**Prototype:**
```c
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:**
```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

### 3.273. __nv_sincosf

**Prototype:**

```c
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, \( cptr \).

**Returns:**

- none

See __nv_sinf() and __nv_cosf().

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

### 3.274. __nv_sincospi

**Prototype:**

```c
void __nv_sincospi(double x, double* sptr, double* cptr)
```

See __nv_sin() and __nv_cos().
Description:
Calculate the sine and cosine of the first input argument, \( x \) (measured in radians), \( x \pi \). The results for sine and cosine are written into the second argument, \( sptr \), and, respectively, third argument, \( zptr \).

Returns:
- none

See \( \_\_nv\_sin\_pi() \) and \( \_\_nv\_cos\_pi() \).

Note:
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

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), \( x \pi \). The results for sine and cosine are written into the second argument, \( sptr \), and, respectively, third argument, \( zptr \).

Returns:
- none

See \( \_\_nv\_sin\_pi() \) and \( \_\_nv\_cos\_pi() \).

Note:
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

Library Availability:
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes
3.276. \texttt{__nv_sinf}

Prototype:

\[ \texttt{float \_\_nv_sinf(float \%x)} \]

Description:

Calculate the sine of the input argument \( x \) [measured in radians].

Returns:

\begin{itemize}
  \item \texttt{__nv_sinf( \pm 0) \text{ returns } \pm 0.}
  \item \texttt{__nv_sinf( \pm \infty) \text{ returns } NaN.}
\end{itemize}

Note:

For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

Library Availability:

Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.277. \texttt{__nv_sinh}

Prototype:

\[ \texttt{double \_\_nv_sinh(double \%x)} \]

Description:

Calculate the hyperbolic sine of the input argument \( x \).

Returns:

\begin{itemize}
  \item \texttt{__nv_sinh( \pm 0) \text{ returns } \pm 0.}
\end{itemize}

Note:

For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

Library Availability:

Compute 2.0: Yes
3.278. __nv_sinhf

**Prototype:**

```c
float __nv_sinhf(float x)
```

**Description:**

Calculate the hyperbolic sine of the input argument \( x \).

**Returns:**

- \( __nv_sinhf( \pm 0 ) \) returns \( \pm 0 \).

**Note:**

For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**

- Compute 2.0: Yes
- Compute 3.0: Yes
- Compute 3.5: Yes

3.279. __nv_sinpi

**Prototype:**

```c
double __nv_sinpi(double x)
```

**Description:**

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

**Returns:**

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

**Note:**

For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.
3.280. \texttt{__nv_sinpif}

**Prototype:**

```c
float __nv_sinpif(float x)
```

**Description:**
Calculate the sine of $x \times \pi$ (measured in radians), where $x$ is the input argument.

**Returns:**
- \texttt{__nv_sinpif(\pm 0)} returns $\pm 0$.
- \texttt{__nv_sinpif(\pm \infty)} returns NaN.

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**Library Availability:**
- Compute 2.0: Yes
- Compute 3.0: Yes
- Compute 3.5: Yes

3.281. \texttt{__nv_sqrt}

**Prototype:**

```c
double __nv_sqrt(double x)
```

**Description:**
Calculate the nonnegative square root of $x$, $\sqrt{x}$.

**Returns:**
Returns $\sqrt{x}$.
- \texttt{__nv_sqrt(\pm 0)} returns $\pm 0$. 

**Library Availability:**
- Compute 2.0: Yes
- Compute 3.0: Yes
- Compute 3.5: Yes
Function Reference

- \( \texttt{__nv_sqrt( } + \infty \text{) returns } +\infty \).
- \( \texttt{__nv_sqrt(} x \texttt{) returns NaN if } x \texttt{ is less than 0.} \\

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

**Library Availability:**
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

### 3.282. \texttt{__nv_sqrtf}

**Prototype:**

```c
float \* __nv_sqrtf(float %x)
```

**Description:**
Calculate the nonnegative square root of \( x, \sqrt{x} \).

**Returns:**
Returns \( \sqrt{x} \).

- \( \texttt{__nv_sqrtf( } \pm 0 \text{) returns } \pm 0 \).
- \( \texttt{__nv_sqrtf(} + \infty \texttt{) returns } +\infty \).
- \( \texttt{__nv_sqrtf(} x \texttt{) returns NaN if } x \texttt{ is less than 0.} \\

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

**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:
- \( \_\_nv\_tan\left( \pm 0 \right) \) returns \( \pm 0 \).
- \( \_\_nv\_tan\left( \pm \infty \right) \) returns NaN.

Note:
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

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:
- \( \_\_nv\_tanf\left( \pm 0 \right) \) returns \( \pm 0 \).
- \( \_\_nv\_tanf\left( \pm \infty \right) \) returns NaN.

Note:
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

Library Availability:
3.285. __nv_tanh

Prototype:

```c
double @__nv_tanh(double %x)
```

Description:
Calculate the hyperbolic tangent of the input argument \( x \).

Returns:
- \( \_\_n\_v\_t\_a\_n\_h( \pm 0 ) \) returns \( \pm 0 \).

Note:
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

Library Availability:
- Compute 2.0: Yes
- Compute 3.0: Yes
- Compute 3.5: Yes

3.286. __nv_tanhf

Prototype:

```c
float @__nv_tanhf(float %x)
```

Description:
Calculate the hyperbolic tangent of the input argument \( x \).

Returns:
- \( \_\_n\_v\_t\_a\_n\_h\_f( \pm 0 ) \) returns \( \pm 0 \).

Note:
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.
Library Availability:
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.287. __nv_tgamma

 Prototype:

\[
\text{double } \_\_\text{nv}_\text{tgamma}(\text{double } x)
\]

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

Returns:
- \( \_\_\text{nv}_\text{tgamma}(\pm 0) \) returns \( \pm \infty \).
- \( \_\_\text{nv}_\text{tgamma}(2) \) returns +0.
- \( \_\_\text{nv}_\text{tgamma}(x) \) returns \( \pm \infty \) if the correctly calculated value is outside the double floating point range.
- \( \_\_\text{nv}_\text{tgamma}(x) \) returns NaN if \( x < 0 \).
- \( \_\_\text{nv}_\text{tgamma}( -\infty ) \) returns NaN.
- \( \_\_\text{nv}_\text{tgamma}( +\infty ) \) returns +\( \infty \).

Note:
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

Library Availability:
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.288. __nv_tgammaf

 Prototype:

\[
\text{float } \_\_\text{nv}_\text{tgammaf}(\text{float } x)
\]

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

Returns:
- \( \text{__nv_tgammaf(} \pm 0 \text{)} \) returns \( \pm \infty \).
- \( \text{__nv_tgammaf(}2\text{)} \) returns +0.
- \( \text{__nv_tgammaf(}x\text{)} \) returns \( \pm \infty \) if the correctly calculated value is outside the double floating point range.
- \( \text{__nv_tgammaf(}x\text{)} \) returns NaN if \( x < 0 \).
- \( \text{__nv_tgammaf(} - \infty \text{)} \) returns NaN.
- \( \text{__nv_tgammaf(} + \infty \text{)} \) returns +\( \infty \).

Note:
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

Library Availability:
- Compute 2.0: Yes
- Compute 3.0: Yes
- Compute 3.5: Yes

3.289. \text{__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:

```c
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:

```c
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:

```c
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:
3.295. __nv_uint2float_ru

Prototype:

```c
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:

```c
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 %l)

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 %l)
### 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:**
```c
double __nv_ull2double_rz(i64 %l)
```

**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:**
```c
float __nv_ull2float_rd(i64 %l)
```

**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:**

```c
float __nv_ull2float_rn(i64 %l)
```

**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:**

```c
float __nv_ull2float_ru(i64 %l)
```

**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. \texttt{__nv_ull2float_rz}

\textbf{Prototype:}

\begin{verbatim}
float __nv_ull2float_rz(i64 %l)
\end{verbatim}

\textbf{Description:}

Convert the unsigned integer value \(x\) to a single-precision floating point value in round-towards-zero mode.

\textbf{Returns:}

Returns converted value.

\textbf{Library Availability:}

Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.305. \texttt{__nv_ullmax}

\textbf{Prototype:}

\begin{verbatim}
i64 __nv_ullmax(i64 %x, i64 %y)
\end{verbatim}

\textbf{Description:}

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

\textbf{Returns:}

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

\textbf{Library Availability:}

Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.306. \texttt{__nv_ullmin}

\textbf{Prototype:}

\begin{verbatim}
i64 __nv_ullmin(i64 %x, i64 %y)
\end{verbatim}
**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:**

```c
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_umin

**Prototype:**

```c
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:**
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 \(x \times y\).

**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 \(x \times y\), where \(x\) and \(y\) are 64-bit unsigned integers.

**Returns:**
Returns the most significant 64 bits of the product \(x \times y\).

**Library Availability:**
Compute 2.0: Yes  
Compute 3.0: Yes  
Compute 3.5: Yes
3.311. __nv_umulhi

Prototype:

```c
i32 @__nv_umulhi(i32 %x, i32 %y)
```

Description:
Calculate the most significant 32 bits of the 64-bit product \( x \times y \), where \( x \) and \( y \) are 32-bit unsigned integers.

Returns:
Returns the most significant 32 bits of the product \( x \times y \).

Library Availability:
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.312. __nv_urhadd

Prototype:

```c
i32 @__nv_urhadd(i32 %x, i32 %y)
```

Description:
Compute average of unsigned input arguments \( x \) and \( y \) as \( (x + y + 1) \gg 1 \), 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:

```c
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:

```c
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 \(x < 0\).
- \(__nv_y0( +\infty)\) returns +0.
- \(__nv_y0(NaN)\) returns NaN.

Note:

For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.
3.315. __nv_y0f

Prototype:

```c
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( +\infty ) returns +0.
- __nv_y0f(NaN) returns NaN.

Note:
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

Library Availability:
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.316. __nv_y1

Prototype:

```c
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.

- \( \texttt{__nv}_y1[0] \) returns \(-\infty\).
- \( \texttt{__nv}_y1[x] \) returns NaN for \( x < 0 \).
- \( \texttt{__nv}_y1[ + \infty ] \) returns +0.
- \( \texttt{__nv}_y1[\text{NaN}] \) returns NaN.

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

**Library Availability:**
- Compute 2.0: Yes
- Compute 3.0: Yes
- Compute 3.5: Yes

### 3.317. \texttt{__nv}_y1f

**Prototype:**
```c
float @\texttt{__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.

- \( \texttt{__nv}_y1f[0] \) returns \(-\infty\).
- \( \texttt{__nv}_y1f[x] \) returns NaN for \( x < 0 \).
- \( \texttt{__nv}_y1f[ + \infty ] \) returns +0.
- \( \texttt{__nv}_y1f[\text{NaN}] \) returns NaN.

**Note:**
For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.
Library Availability:
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.318. __nv_yn

Prototype:
```c
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 for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 7.

Library Availability:
Compute 2.0: Yes
Compute 3.0: Yes
Compute 3.5: Yes

3.319. __nv_ynf

Prototype:
```c
float __nv_ynf(i32 %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 \( n \).

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

**Note:**

For accuracy information for this function see the CUDA C++ Programming Guide, Appendix E.1, Table 6.

**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.

VESA DisplayPort

DisplayPort and DisplayPort Compliance Logo, DisplayPort Compliance Logo for Dual-mode Sources, and DisplayPort Compliance Logo for Active Cables are trademarks owned by the Video Electronics Standards Association in the United States and other countries.

HDMI

HDMI, the HDMI logo, and High-Definition Multimedia Interface are trademarks or registered trademarks of HDMI Licensing LLC.

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

© 2013-2019 NVIDIA Corporation. All rights reserved.