Precision and Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs

White paper

TB-06711-001_v11.6 | March 2022

Table of Contents

List of Figures

Abstract

A number of issues related to floating point accuracy and compliance are a frequent source of confusion on both CPUs and GPUs. The purpose of this white paper is to discuss the most common issues related to NVIDIA GPUs and to supplement the documentation in the CUDA C++ Programming Guide.

Chapter 1. Introduction

Since the widespread adoption in 1985 of the IEEE Standard for *Binary Floating-Point Arithmetic* (IEEE 754-1985 [1]) virtually all mainstream computing systems have implemented the standard, including NVIDIA with the CUDA architecture. IEEE 754 standardizes how arithmetic results should be *approximated* in floating point. Whenever working with inexact results, programming decisions can affect accuracy. It is important to consider many aspects of floating point behavior in order to achieve the highest performance with the precision required for any specific application. This is especially true in a heterogeneous computing environment where operations will be performed on different types of hardware.

Understanding some of the intricacies of floating point and the specifics of how NVIDIA hardware handles floating point is obviously important to CUDA programmers striving to implement correct numerical algorithms. In addition, users of libraries such as *cuBLAS* and *cuFFT* will also find it informative to learn how NVIDIA handles floating point under the hood.

We review some of the basic properties of floating point calculations in Chapter 2. We also discuss the fused multiply-add operator, which was added to the IEEE 754 standard in 2008 [2] and is built into the hardware of NVIDIA GPUs. In Chapter 3 we work through an example of computing the dot product of two short vectors to illustrate how different choices of implementation affect the accuracy of the final result. In Chapter 4 we describe NVIDIA hardware versions and NVCC compiler options that affect floating point calculations. In Chapter 5 we consider some issues regarding the comparison of CPU and GPU results. Finally, in Chapter 6 we conclude with concrete recommendations to programmers that deal with numeric issues relating to floating point on the GPU.

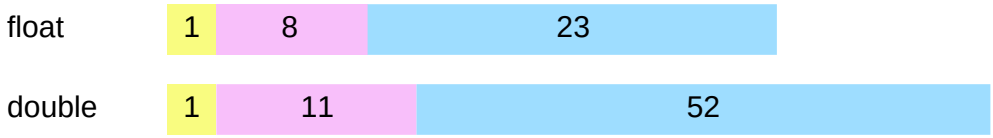
Chapter 2. Floating Point

2.1. Formats

Floating point encodings and functionality are defined in the IEEE 754 Standard [2] last revised in 2008. Goldberg [5] gives a good introduction to floating point and many of the issues that arise.

The standard mandates binary floating point data be encoded on three fields: a one bit sign field, followed by exponent bits encoding the exponent offset by a numeric bias specific to each format, and bits encoding the significand (or fraction).

In order to ensure consistent computations across platforms and to exchange floating point data, IEEE 754 defines basic and interchange formats. The 32 and 64 bit basic binary floating point formats correspond to the C data types `float` and `double`. Their corresponding representations have the following bit lengths:



For numerical data representing finite values, the sign is either negative or positive, the exponent field encodes the exponent in base 2, and the fraction field encodes the significand without the most significant non-zero bit. For example, the value -192 equals $(-1)^1 \times 2^7 \times 1.5$, and can be represented as having a negative sign, an exponent of 7, and a fractional part .5. The exponents are biased by 127 and 1023, respectively, to allow exponents to extend from negative to positive. Hence the exponent 7 is represented by bit strings with values 134 for float and 1030 for double. The integral part of 1. is implicit in the fraction.

float

1	10000110	.10000000000000000000000

double

1	10000000110	.10000000000000000...0000000

Also, encodings to represent infinity and not-a-number (NaN) data are reserved. The IEEE 754 Standard [2] describes floating point encodings in full.

Given that the fraction field uses a limited number of bits, not all real numbers can be represented exactly. For example the mathematical value of the fraction 2/3 represented in binary is 0.10101010... which has an infinite number of bits after the binary point. The value 2/3 must be rounded first in order to be represented as a floating point number with limited precision. The rules for rounding and the rounding modes are specified in IEEE 754. The most frequently used is the round-to-nearest-or-even mode (abbreviated as round-to-nearest). The value 2/3 rounded in this mode is represented in binary as:

float

0	01111110	.01010101010101010101011

double

0	01111111110	.01010101010101010...1010101

The sign is positive and the stored exponent value represents an exponent of -1.

2.2. Operations and Accuracy

The IEEE 754 standard requires support for a handful of operations. These include the arithmetic operations add, subtract, multiply, divide, square root, fused-multiply-add, remainder, conversion operations, scaling, sign operations, and comparisons. The results of these operations are guaranteed to be the same for all implementations of the standard, for a given format and rounding mode.

The rules and properties of mathematical arithmetic do not hold directly for floating point arithmetic because of floating point's limited precision. For example, the table below shows single precision values *A*, *B*, and *C*, and the mathematical exact value of their sum computed using different associativity.

$$A = 2^1 \times 1.00000000000000000000001$$
$$B = 2^0 \times 1.00000000000000000000001$$
$$C = 2^3 \times 1.00000000000000000000001$$
$$(A+B)+C = 2^3 \times 1.01100000000000000000001011$$
$$A+(B+C) = 2^3 \times 1.01100000000000000000001011$$

Mathematically, (*A* + *B*) + *C* does equal *A* + (*B* + *C*).

Let rn(*x*) denote one rounding step on *x*. Performing these same computations in single precision floating point arithmetic in round-to-nearest mode according to IEEE 754, we obtain:

$$A+B = 2^1 \times 1.10000000000000000000000110000...$$
$$\mathrm{rn}(A+B) = 2^1 \times 1.10000000000000000000010$$
$$B+C = 2^3 \times 1.00100000000000000000000100100...$$
$$\mathrm{rn}(B+C) = 2^3 \times 1.00100000000000000000001$$
$$A+B+C = 2^3 \times 1.01100000000000000000000101100...$$
$$\mathrm{rn}(\mathrm{rn}(A+B)+C) = 2^3 \times 1.01100000000000000000010$$
$$\mathrm{rn}(A+\mathrm{rn}(B+C)) = 2^3 \times 1.01100000000000000000001$$

For reference, the exact, mathematical results are computed as well in the table above. Not only are the results computed according to IEEE 754 different from the exact mathematical results, but also the results corresponding to the sum rn(rn(A + B) + C) and the sum rn(A + rn(B + C)) are different from each other. In this case, rn(A + rn(B + C)) is closer to the correct mathematical result than rn(rn(A + B) + C).

This example highlights that seemingly identical computations can produce different results even if all basic operations are computed in compliance with IEEE 754.

Here, the order in which operations are executed affects the accuracy of the result. The results are independent of the host system. These same results would be obtained using any microprocessor, CPU or GPU, which supports single precision floating point.

2.3. The Fused Multiply-Add (FMA)

In 2008 the IEEE 754 standard was revised to include the fused multiply-add operation (*FMA*). The FMA operation computes $\mathrm{rn}(X \times Y + Z)$ with only one rounding step. Without the FMA operation the result would have to be computed as $\mathrm{rn}(\mathrm{rn}(X \times Y) + Z)$ with two rounding steps, one for multiply and one for add. Because the FMA uses only a single rounding step the result is computed more accurately.

Let's consider an example to illustrate how the FMA operation works using decimal arithmetic first for clarity. Let's compute $x^2 - 1$ with four digits of precision after the decimal point, or a total of five digits of precision including the leading digit before the decimal point.

For $x = 1.0008$, the correct mathematical result is $x^2 - 1 = 1.60064 \times 10^{-4}$. The closest number using only four digits after the decimal point is 1.6006×10^{-4}. In

this case $\mathrm{rn}(x^2-1)=1.6006\times10^{-4}$ which corresponds to the fused multiply-add operation $\mathrm{rn}(x\times x+(-1))$. The alternative is to compute separate multiply and add steps. For the multiply, $x^2=1.00160064$, so $\mathrm{rn}(x^2)=1.0016$. The final result is $\mathrm{rn}(\mathrm{rn}(x^2)-1)=1.6000\times10^{-4}$.

Rounding the multiply and add separately yields a result that is off by 0.00064. The corresponding FMA computation is wrong by only 0.00004, and its result is closest to the correct mathematical answer. The results are summarized below:

$$
\begin{aligned}
x &= 1.0008 \\
x^2 &= 1.00160064 \\
x^2-1 &= 1.60064\times10^{-4} && \text{true value} \\
\mathrm{rn}(x^2-1) &= 1.6006\times10^{-4} && \text{fused multiply–add} \\
\mathrm{rn}(x^2) &= 1.0016\times10^{-4} \\
\mathrm{rn}(\mathrm{rn}(x^2)-1) &= 1.6000\times10^{-4} && \text{multiply, then add}
\end{aligned}
$$

Below is another example, using binary single precision values:

$$
\begin{aligned}
A &= 2^{0} \times 1.00000000000000000000001 \\
B &= -2^{0} \times 1.00000000000000000000010 \\
\mathrm{rn}(A\times A+B) &= 2^{-46} \times 1.00000000000000000000000 \\
\mathrm{rn}(\mathrm{rn}(A\times A)+B) &= 0
\end{aligned}
$$

In this particular case, computing $\mathrm{rn}(\mathrm{rn}(A\times A)+B)$ as an IEEE 754 multiply followed by an IEEE 754 add loses all bits of precision, and the computed result is 0. The alternative of computing the FMA $\mathrm{rn}(A\times A+B)$ provides a result equal to the mathematical value. In general, the fused-multiply-add operation generates more accurate results than computing one multiply followed by one add. The choice of whether or not to use the fused operation depends on whether the platform provides the operation and also on how the code is compiled.

Figure 1 shows CUDA C++ code and output corresponding to inputs *A* and *B* and operations from the example above. The code is executed on two different hardware platforms: an x86-class CPU using *SSE* in single precision, and an NVIDIA GPU with compute capability 2.0. At the time this paper is written (Spring 2011) there are no commercially available x86 CPUs which offer hardware FMA. Because of this, the computed result in single precision in SSE would be 0. NVIDIA GPUs with compute capability 2.0 do offer hardware FMAs, so the result of executing this code will be the more accurate one by default. However, both results are correct according to the IEEE 754 standard. The code fragment was compiled without any special intrinsics or compiler options for either platform.

The fused multiply-add helps avoid loss of precision during subtractive cancellation. Subtractive cancellation occurs during the addition of quantities of similar magnitude with opposite signs. In this case many of the leading bits cancel, leaving fewer meaningful bits of precision in the result. The fused multiply-add computes a double-width product during the multiplication. Thus even if subtractive cancellation occurs during the addition there are still enough valid bits remaining in the product to get a precise result with no loss of precision.

Figure 1. Multiply and Add Code Fragment and Output for x86 and NVIDIA Fermi GPU

```
union  {
    float  f;
    unsigned  int  i
} a,  b;
float  r;

a.i = 0x3F800001;
b.i = 0xBF800002;
r = a.f  * a.f  + b.f;

printf("a %.8g\n", a.f);
printf("b %.8g\n", b.f);
printf("r %.8g\n",   r);
```

x86-64 output:

```
a:  1.0000001
b: -1.0000002
r:  0
```

NVIDIA Fermi output:

```
a:  1.0000001
b: -1.0000002
r:  1.4210855e-14
```

Chapter 3. Dot Product: An Accuracy Example

Consider the problem of finding the dot product of two short vectors $\vec{a}$ and $\vec{b}$, both with four elements.

$$\vec{a} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} \quad \vec{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} \quad \vec{a} \cdot \vec{b} = a_1 b_1 + a_2 b_2 + a_3 b_3 + a_4 b_4$$

This operation is easy to write mathematically, but its implementation in software involves several choices. All of the strategies we will discuss use purely IEEE 754 compliant operations.

3.1. Example Algorithms

We present three algorithms which differ in how the multiplications, additions, and possibly fused multiply-adds are organized. These algorithms are presented in Figure 2, Figure 3, and Figure 4. Each of the three algorithms is represented graphically. Individual operation are shown as a circle with arrows pointing from arguments to operations.

The simplest way to compute the dot product is using a short loop as shown in Figure 2. The multiplications and additions are done separately.

Figure 2. Serial Method to Compute Vectors Dot Product

The serial method uses a simple loop with separate multiplies and adds to compute the do t product of the vectors. The final result can be represented as ((((a1 x b1) + (a2 x b2)) + (a3 x b3)) + (a4 x b4)).

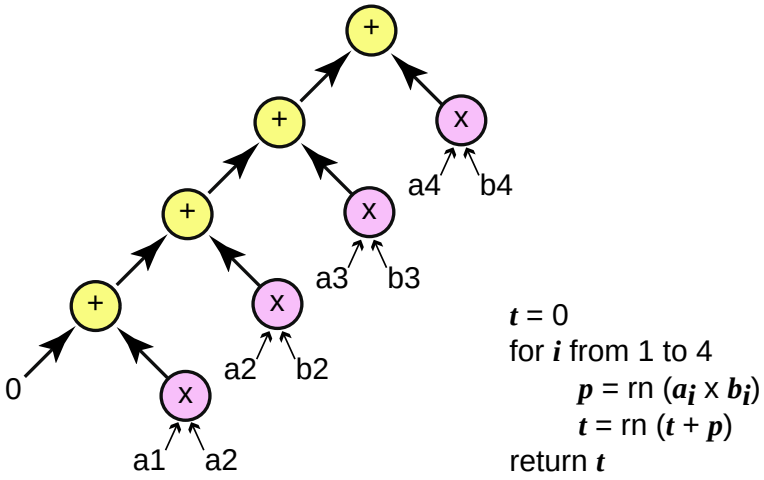
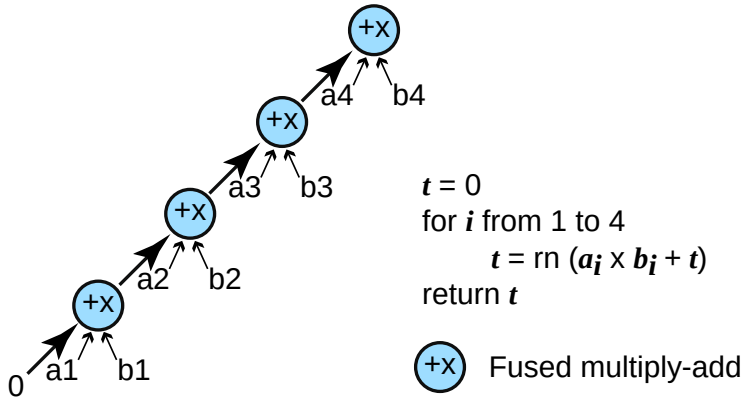


Figure 3. FMA Method to Compute Vector Dot Product

The FMA method uses a simple loop with fused multiply-adds to compute the dot product of the vectors. The final result can be represented as a4 x b4 = (a3 x b3 + (a2 x b2 + (a1 x b1 + 0))).



A simple improvement to the algorithm is to use the fused multiply-add to do the multiply and addition in one step to improve accuracy. Figure 3 shows this version.

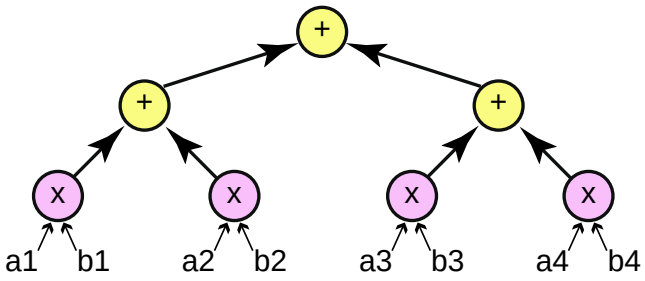
Yet another way to compute the dot product is to use a divide-and-conquer strategy in which we first find the dot products of the first half and the second half of the vectors, then combine these results using addition. This is a recursive strategy; the base case is the dot product of vectors of length 1 which is a single multiply. Figure 4 graphically illustrates this approach. We call this algorithm the parallel algorithm because the two sub-problems can be computed in parallel as they have no dependencies. The algorithm does not require a parallel implementation, however; it can still be implemented with a single thread.

3.2. Comparison

All three algorithms for computing a dot product use IEEE 754 arithmetic and can be implemented on any system that supports the IEEE standard. In fact, an implementation of the serial algorithm on multiple systems will give exactly the same result. So will implementations of the FMA or parallel algorithms. However, results computed by an implementation of the serial algorithm may differ from those computed by an implementation of the other two algorithms.

Figure 4. The Parallel Method to Reduce Individual Elements Products into a Final Sum

The parallel method uses a tree to reduce all the products of individual elements into a final sum. The final result can be represented as $((a_1 \times b_1) + (a_2 \times b_2)) + ((a_3 \times b_3) + (a_4 \times b_4))$.



$p1 = \text{rn}(a_1 \times b_1)$
$p2 = \text{rn}(a_2 \times b_2)$
$p3 = \text{rn}(a_3 \times b_3)$
$p4 = \text{rn}(a_4 \times b_4)$
$s_{left} = \text{rn}(p_1 + p_2)$
$s_{right} = \text{rn}(p_3 + p_4)$
$t = \text{rn}(s_{left} + s_{right})$
return t

Figure 5. Algorithms Results vs. the Correct Mathematical Dot Product

The three algorithms yield results slightly different from the correct mathematical dot product.

method	result	float value
exact	.0559587528435...	0x3D65350158...
serial	.0559588074	0x3D653510
FMA	.0559587515	0x3D653501
parallel	.0559587478	0x3D653500

For example, consider the vectors:

a = [1.907607, -.7862027, 1.147311, .9604002]
b = [-.9355000, -.6915108, 1.724470, -.7097529]

whose elements are randomly chosen values between -1 and 2. The accuracy of each algorithm corresponding to these inputs is shown in Figure 5.

The main points to notice from the table are that each algorithm yields a different result, and they are all slightly different from the correct mathematical dot product. In this example the FMA version is the most accurate, and the parallel algorithm is more accurate than the serial algorithm. In our experience these results are typical; fused multiply-add significantly

increases the accuracy of results, and parallel tree reductions for summation are usually much more accurate than serial summation.

Chapter 4. CUDA and Floating Point

NVIDIA has extended the capabilities of GPUs with each successive hardware generation. Current generations of the NVIDIA architecture such as *Tesla Kxx*, *GTX 8xx*, and *GTX 9xx*, support both single and double precision with *IEEE 754* precision and include hardware support for fused multiply-add in both single and double precision. In CUDA, the features supported by the GPU are encoded in the *compute capability* number. The runtime library supports a function call to determine the compute capability of a GPU at runtime; the *CUDA C ++ Programming Guide* also includes a table of compute capabilities for many different devices [7].

4.1. Compute Capability 2.0 and Above

Devices with compute capability *2.0 and above* support both single and double precision *IEEE 754* including fused multiply-add in both single and double precision. Operations such as square root and division will result in the floating point value closest to the correct mathematical result in both single and double precision, by default.

4.2. Rounding Modes

The *IEEE 754* standard defines four rounding modes: round-to-nearest, round towards positive, round towards negative, and round towards zero. CUDA supports all four modes. By default, operations use round-to-nearest. Compiler intrinsics like the ones listed in the tables below can be used to select other rounding modes for individual operations.

mode	interpretation
rn	round to nearest, ties to even
rz	round towards zero
ru	round towards $+\infty$
rd	round towards $-\infty$

`x + y`	addition

`__fadd_[rn | rz | ru | rd] (x, y)`	
`x * y` `__fmul_[rn | rz | ru | rd] (x, y)`	multiplication
`fmaf (x, y, z)` `__fmaf_[rn | rz | ru | rd] (x, y, z)`	FMA
`1.0f / x` `__frcp_[rn | rz | ru | rd] (x)`	reciprocal
`x / y` `__fdiv_[rn | rz | ru | rd] (x, y)`	division
`sqrtf(x)` `__fsqrt_[rn | rz | ru | rd] (x)`	square root

`x + y` `__dadd_[rn | rz | ru | rd] (x, y)`	addition
`x * y` `__dmul_[rn | rz | ru | rd] (x, y)`	multiplication
`fma (x, y, z)` `__fma_[rn | rz | ru | rd] (x, y, z)`	FMA
`1.0 / x` `__drcp_[rn | rz | ru | rd] (x)`	reciprocal
`x / y` `__ddiv_[rn | rz | ru | rd] (x, y)`	division
`sqrtf(x)` `__dsqrt_[rn | rz | ru | rd] (x)`	square root

4.3. Controlling Fused Multiply-add

In general, the fused multiply-add operation is faster and more accurate than performing separate multiply and add operations. However, on occasion you may wish to *disable* the merging of multiplies and adds into fused multiply-add instructions. To inhibit this optimization one can write the multiplies and additions using intrinsics with explicit rounding mode as shown in the previous tables. Operations written directly as intrinsics are guaranteed to remain independent and will not be merged into fused multiply-add instructions. It is also possible to disable FMA merging via a compiler flag.

4.4. Compiler Flags

Compiler flags relevant to *IEEE 754* operations are `-ftz={true|false}`, `-prec-div={true|false}`, and `-prec-sqrt={true|false}`. These flags control single precision operations on devices of compute capability of 2.0 or later.

mode	flags
IEEE 754 mode (default)	-ftz=false -prec-div=true -prec-sqrt=true
fast mode	-ftz=true -prec-div=false -prec-sqrt=false

The default *IEEE 754 mode* means that single precision operations are correctly rounded and support denormals, as per the IEEE 754 standard. In the *fast mode* denormal numbers are flushed to zero, and the operations division and square root are not computed to the nearest floating point value. The flags have no effect on double precision or on devices of compute capability below 2.0.

4.5. Differences from x86

NVIDIA GPUs differ from the x86 architecture in that rounding modes are encoded within each floating point instruction instead of dynamically using a floating point control word. Trap handlers for floating point exceptions are not supported. On the GPU there is no status flag to indicate when calculations have overflowed, underflowed, or have involved inexact arithmetic. Like *SSE*, the precision of each GPU operation is encoded in the instruction (for x87 the precision is controlled dynamically by the floating point control word).

Chapter 5. Considerations for a Heterogeneous World

5.1. Mathematical Function Accuracy

So far we have only considered simple math operations such as addition, multiplication, division, and square root. These operations are simple enough that computing the best floating point result (e.g., the closest in round-to-nearest) is reasonable. For other mathematical operations computing the best floating point result is harder.

The problem is called the *table maker's dilemma*. To guarantee the correctly rounded result, it is not generally enough to compute the function to a fixed high accuracy. There might still be rare cases where the error in the high accuracy result affects the rounding step at the lower accuracy.

It is possible to solve the dilemma for particular functions by doing mathematical analysis and formal proofs [4], but most math libraries choose instead to give up the guarantee of correct rounding. Instead they provide implementations of math functions and document bounds on the relative error of the functions over the input range. For example, the double precision `sin` function in CUDA is guaranteed to be accurate to within 2 units in the last place (ulp) of the correctly rounded result. In other words, the difference between the computed result and the mathematical result is at most ±2 with respect to the least significant bit position of the fraction part of the floating point result.

For most inputs the `sin` function produces the correctly rounded result. Take for example the C code sequence shown in Figure 6. We compiled the code sequence on a 64-bit x86 platform using gcc version 4.4.3 (Ubuntu 4.3.3-4ubuntu5).

This shows that the result of computing cos(5992555.0) using a common library differs depending on whether the code is compiled in 32-bit mode or 64-bit mode.

The consequence is that different math libraries cannot be expected to compute exactly the same result for a given input. This applies to GPU programming as well. Functions compiled for the GPU will use the NVIDIA CUDA math library implementation while functions compiled for the CPU will use the host compiler math library implementation (e.g., *glibc* on Linux). Because these implementations are independent and neither is guaranteed to be correctly rounded, the results will often differ slightly.

Figure 6. Cosine Computations using the `glibc` Math Library

The computation of cosine using the **`glibc`** Math Library yields different results when compiled with **`-m32`** and **`-m64`**.

```
volatile float x = 5992555.0;
printf("cos(%f): %.10g\n", x, cos(x));

gcc test.c -lm -m64
cos(5992555.000000): 3.320904615e-07

gcc test.c -lm -m32
cos(5992555.000000): 3.320904692e-07
```

5.2. x87 and SSE

One of the unfortunate realities of C compilers is that they are often poor at preserving IEEE 754 semantics of floating point operations [6]. This can be particularly confusing on platforms that support x87 and SSE operations. Just like CUDA operations, SSE operations are performed on single or double precision values, while x87 operations often use an additional internal 80-bit precision format. Sometimes the results of a computation using x87 can depend on whether an intermediate result was allocated to a register or stored to memory. Values stored to memory are rounded to the declared precision (e.g., single precision for `float` and double precision for `double`). Values kept in registers can remain in extended precision. Also, x87 instructions will often be used by default for 32-bit compiles but SSE instructions will be used by default for 64-bit compiles.

Because of these issues, guaranteeing a specific precision level on the CPU can sometimes be tricky. When comparing CPU results to results computed on the GPU, it is generally best to compare using SSE instructions. SSE instructions follow IEEE 754 for single and doubleprecision.

On 32-bit x86 targets without SSE it can be helpful to declare variables using `volatile` and force floating point values to be stored to memory (`/Op` in Visual Studio and `-ffloat-store` in `gcc`). This moves results from extended precision registers into memory, where the precision is precisely single or double precision. Alternately, the x87 control word can be updated to set the precision to 24 or 53 bits using the assembly instruction `fldcw` or a compiler option such as `-mpc32` or`-mpc64` in `gcc`.

5.3. Core Counts

As we have shown in Chapter 3, the final values computed using *IEEE 754* arithmetic can depend on implementation choices such as whether to use fused multiply-add or whether additions are organized in series or parallel. These differences affect computation on the CPU and on the GPU.

One way such differences can arise is from differences between the number of concurrent threads involved in a computation. On the GPU, a common design pattern is to have all threads in a block coordinate to do a parallel reduction on data within the block, followed by a serial reduction of the results from each block. Changing the number of threads per

block reorganizes the reduction; if the reduction is addition, then the change rearranges parentheses in the long string of additions.

Even if the same general strategy such as parallel reduction is used on the CPU and GPU, it is common to have widely different numbers of threads on the GPU compared to the CPU. For example, the GPU implementation might launch blocks with 128 threads per block, while the CPU implementation might use 4 threads in total.

5.4. Verifying GPU Results

The same inputs will give the same results for individual *IEEE 754* operations to a given precision on the CPU and GPU. As we have explained, there are many reasons why the same sequence of operations may not be performed on the CPU and GPU. The GPU has fused multiply-add while the CPU does not. Parallelizing algorithms may rearrange operations, yielding different numeric results. The CPU may be computing results in a precision higher than expected. Finally, many common mathematical functions are not required by the IEEE 754 standard to be correctly rounded so should not be expected to yield identical results between implementations.

When porting numeric code from the CPU to the GPU of course it makes sense to use the x86 CPU results as a reference. But differences between the CPU result and GPU result must be interpreted carefully. Differences are not automatically evidence that the result computed by the GPU is wrong or that there is a problem on the GPU.

Computing results in a high precision and then comparing to results computed in a lower precision can be helpful to see if the lower precision is adequate for a particular application. However, rounding high precision results to a lower precision is not equivalent to performing the entire computation in lower precision. This can sometimes be a problem when using x87 and comparing results against the GPU. The results of the CPU may be computed to an unexpectedly high extended precision for some or all of the operations. The GPU result will be computed using single or double precision only.

Chapter 6. Concrete Recommendations

The key points we have covered are the following:

Use the fused multiply-add operator.

The fused multiply-add operator on the GPU has high performance and increases the accuracy of computations. No special flags or function calls are needed to gain this benefit in CUDA programs. Understand that a hardware fused multiply-add operation is not yet available on the CPU, which can cause differences in numerical results.

Compare results carefully.

Even in the strict world of *IEEE 754* operations, minor details such as organization of parentheses or thread counts can affect the final result. Take this into account when doing comparisons between implementations.

Know the capabilities of your GPU.

The numerical capabilities are encoded in the compute capability number of your GPU. Devices of compute capability 2.0 and later are capable of single and double precision arithmetic following the IEEE 754 standard, and have hardware units for performing fused multiply-add in both single and double precision.

Take advantage of the CUDA math library functions.

These functions are documented in Appendix E of the *CUDA C++ Programming Guide* [7]. The math library includes all the math functions listed in the C99 standard [3] plus some additional useful functions. These functions have been tuned for a reasonable compromise between performance and accuracy.

We constantly strive to improve the quality of our math library functionality. Please let us know about any functions that you require that we do not provide, or if the accuracy or performance of any of our functions does not meet your needs. Leave comments in the *NVIDIA CUDA forum*[1] or join the *Registered Developer Program*[2] and file a bug with your feedback.

[1] http://forums.nvidia.com/index.php?showforum=62
[2] http://developer.nvidia.com/join-nvidia-registered-developer-program

Appendix A. Acknowledgements

This paper was authored by Nathan Whitehead and Alex Fit-Florea for NVIDIA Corporation.

Thanks to Ujval Kapasi, Kurt Wall, Paul Sidenblad, Massimiliano Fatica, Everett Phillips, Norbert Juffa, and Will Ramey for their helpful comments and suggestions.

Appendix B. References

[1] *ANSI/IEEE 754-1985. American National Standard - IEEE Standard for Binary Floating-Point Arithmetic. American National Standards Institute, Inc., New York, 1985.*

[2] *IEEE 754-2008. IEEE 754–2008 Standard for Floating-Point Arithmetic. August 2008.*

[3] *ISO/IEC 9899:1999(E). Programming languages - C. American National Standards Institute, Inc., New York, 1999.*

[4] *Catherine Daramy-Loirat, David Defour, Florent de Dinechin, Matthieu Gallet, Nicolas Gast, and Jean-Michel Muller. CR-LIBM: A library of correctly rounded elementary functions in double-precision, February 2005.*

[5] *David Goldberg. What every computer scientist should know about floating-point arithmetic. ACM Computing Surveys, March 1991.* Edited reprint available at: http://download.oracle.com/docs/cd/E19957-01/806-3568/ncg_goldberg.html.

[6] *David Monniaux. The pitfalls of verifying floating-point computations. ACM Transactions on Programming Languages and Systems, May 2008.*

[7] *NVIDIA. CUDA C++ Programming Guide Version 10.2, 2019.*

Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation ("NVIDIA") makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer ("Terms of Sale"). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

OpenCL

OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc.

Trademarks

NVIDIA and the NVIDIA logo are trademarks or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

NVIDIA Corporation | 2788 San Tomas Expressway, Santa Clara, CA 95051
https://www.nvidia.com