Analysis of Performance/Accuracy Tradeoffs for Floating Point Applications on GPUs

A thesis

presented to

the Faculty of Graduate School

at University of Missouri-Columbia

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Huyen Nguyen

Dr. Michela Becchi, Thesis Advisor

May 2016

The undersigned, appointed by the dean of the Graduate School, have examined the thesis entitled

ANALYSIS OF PERFORMANCE/ACCURACY TRADEOFFS FOR

FLOATING POINT APPLICATIONS ON GPUS

presented by Huyen Nguyen, a candidate for the degree of Master of Science, and hereby certify that, in their opinion, it is worthy of acceptance.

Professor Michela Becchi

Professor Grant Scott

Professor Filiz Bunyak Ersoy

This work has been supported by NSF award CCF-1513603 and by equipment donations from Nvidia Corporation.

ACKNOWLEDGEMENTS

First and foremost, I would like to express my sincere gratitude to my research advisor, Michela Becchi. Thank you for your invaluable guidance and advice during my graduate studies. Your ideas and contributions have had a major influence on my thesis. From you, I have learned how to participate in groups, how to work more efficiently, and especially how to do better research.

I would like to thank all friends in the NPS lab at Mizzou. The knowledge you have shared and the happy environment you have created have made my time in the lab very enjoyable. You are an indispensable factor contributing to the success of my studies.

I am thankful to my Vietnamese friends at Mizzou for bringing me a happy life at Columbia, MO. You have helped me integrate into the community quickly and easily. Thanks for becoming my friends during these years; our friendships will last forever. I want to especially thank the family of Thong Thai and Nguyen Doan, the family of Tu Vu and Lan Tran; you have made my life at Mizzou healthier with your delicious meals.

Finally, I cannot thank my family enough for their love and support. I thank my parents for encouraging me to join graduate school; you always make my life better. I also thank my parents-in-law for supporting me while away from home. I extremely thank my husband – Nam Pham for taking care of our daughters and encouraging me through my years in graduate school. I am very happy to have you by my side.

Thank you!

Huyen

University of Missouri-Columbia

May 2016

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

ANALYSIS OF PERFORMANCE/ACCURACY TRADEOFFS FOR

FLOATING POINT APPLICATIONS ON GPUS

Huyen Nguyen

Dr. Michela Becchi, Thesis Supervisor

ABSTRACT

Floating-point computations produce approximate results, which can lead to inaccuracy problems. Existing work addresses two issues: first, the design of high precision floating-point representations; second, the study of methods to trade-off accuracy and performance of serial CPU applications. However, a comprehensive study of the trade-off between accuracy and performance for multithreaded applications is missing. In my thesis, I study this trade-off on GPU. In particular, my study covers the use of different floating-point precisions (i.e., single and double floating-point precision in IEEE 754 standard, GNU Multiple Precision, and composite floating-point precision) on a variety of real-world and synthetic benchmark applications. I explore how the use of multithreading and instruction-level parallelism on GPU can allow the use of higher precision arithmetic to improve accuracy without paying in terms of execution time. As a result of my analysis, I provide insights to guide users to the selection of the arithmetic precision leading to a good performance/accuracy tradeoff depending on the arithmetic operations used in their program (addition, multiplication, division), the degree of multithreading of their program, and its arithmetic intensity.

Keywords – floating-point arithmetic; parallel computing; multithreading.

Chapter 1 Introduction

Applications relying on floating-point arithmetic are common in various fields such as graphics, finance, engineering and science. Floating-point numbers are an approximation of real numbers, and therefore their use can lead to inaccuracy problems, which are often ignored by programmers. The following simple example illustrates an inaccuracy problem due to the use of floating-point arithmetic. Let us consider a large array of floating-point numbers with two properties: (i) the accurate summation of all values in the array is zero; and (ii) the absolute value of the elements in the array belongs either to a subset of very small numbers, or to a subset of very large numbers. In other words, the elements in the array may have substantially different orders of magnitude. Table 1 shows the results obtained by summing all elements of 8M-element array with these properties using single and double floating-point precision on CPU in a sequential fashion. In different columns we shows the result for arrays constructed using different intervals.

Table 1: The accuracy of the global summation on CPU

Precision	*Intervals: ($10^{-3,}$ 10^{-2}) & (10^{2}, 10^{3}).*	*Intervals: ($10^{-4,}$ 10^{-3}) & (10^{3}, 10^{4}).*	*Intervals: ($10^{-6,}$ 10^{-5}) & (10^{5}, 10^{6}).*
Single (32 bits)	-3.57E+01	-2.49E+02	-3.77E+03
Double (64 bits)	1.52E-08	-2.89E-06	-4.26E-05

As can be seen, in all cases the result computed is inaccurate (i.e., it is different from zero). Further, the accuracy of the computation decreases as the difference in order of magnitude of the elements increases. While the use of higher arithmetic precision can improve accuracy, it may lead to degraded performance. For some applications (such as medical studies, climate modeling, quantum theory) result accuracy is paramount; others can tolerate a moderate loss of accuracy, especially if better accuracy comes at a cost of degraded performance. While there have been studies addressing this problem for sequential CPU computation [8][9], very little work has considered the effect of multithreading on the performance/accuracy tradeoff. The goal of this thesis work is to provide insights into performance/accuracy tradeoffs on GPU. In particular, we consider different floating-point precisions, including: single and double floating-point precision in IEEE 754 standard, GNU Multiple Precision (GMP), and composite floating-point precision defined by M. Taufer et al. [1]. Our study focuses on NVIDIA devices and uses NVIDIA CUDA as GPU programming language. However, our results can be generalized to other parallel architectures using the IEEE 754 floating point standard, as well as different parallel programming models.

Our study shows that on GPU higher precision may lead to a better tradeoff between execution time and accuracy, and provides directions on the selection of the arithmetic precision that optimizes this tradeoff. Our study covers real-world benchmark applications with different computational characteristics. As a side result, our analysis of these benchmark applications led us to the identifications of inaccuracy problems due to the use of the composite precision library designed by M. Taufer et al. [1]. To complement our analysis, we construct two micro-benchmarks: the first aims to provide better insights and a deeper examination of the effect of the arithmetic intensity and the degree of multithreading on the performance/accuracy tradeoffs, and the second aims to analyze the behavior of the multiplication and division operations using different arithmetic precisions, with a focus on composite precision.

1.1 Contributions

In this thesis we make the following contributions:

- A detailed investigation of the performance/accuracy tradeoff for different floating-point precisions – single precision (*float*), double precision (*double*), composite precision (*float2* based on float, and *double2* based on double), and multiple precision (*GMP*) [7] using global summation in [1] and two benchmark applications from the Rodinia Benchmark Suite [2]. We modified these applications to use all types of precisions above and run on NVIDIA GPUs.
- Micro-benchmarks to: (1) analyze the effect of arithmetic intensity on the performance/accuracy tradeoff on GPU, and (2) explain the behavior of composite precision multiplication and division.
- Insights to guide users to the selection of the arithmetic precision leading to a good performance/accuracy tradeoff depending on the arithmetic operations used in their program (addition, multiplication, division), the degree of multithreading of their program, and its arithmetic intensity.

1.2 Organization

The remainder of the thesis is organized into nine chapters. In Chapter 2 we provide background information. In Chapter 3 we discuss our research questions and methodology. In Chapter 4 we discuss the related work. In Chapters 5 we present and discuss results on global summation program. Chapter 6 describes the compute-intensive micro-benchmark we designed to provide an in-depth discussion of the effect of the arithmetic intensity and multithreading on the performance/accuracy tradeoff and the results obtained by this analysis. In Chapter 7 we study the LU decomposition and Gaussian Elimination benchmarks In Chapter 8 we describe the *multiplication* and *do-undo* micro-benchmarks designed to study different types of multiplications and divisions on GPUs, and discuss the results of this analysis. In Chapter 9 we conclude and discuss future research directions.

Chapter 2 Background

This chapter provides background information on floating-point arithmetic and on the GPU architecture. Section 2.1 provides a definition of floating-point arithmetic, and describes its notation and the IEEE 754 standard. In addition, it provides a discussion of the sources of floating-point inaccuracy. Section 2.2 introduces the composite floating-point precision library. Section 2.3 describes the GNU Multiple Precision library (GMP) and the CUDA Multiple precision (CUMP) libraries. Section 2.4 introduces NVIDIA GPUs including their hardware and software architecture and their floating-point support.

2.1 Floating Point Arithmetic

2.1.1 Floating Point Definition and Notation

Definition: According to Wikipedia, "in computing, 'floating-point' is the formulaic representation that approximates a real number in computer language so as to support a trade-off between range and precision" [15]. The floating-point arithmetic can be less or more accurate depending on the computer hardware and configuration.

Floating-point notation: Floating-point notation is a representation of real numbers using a scientific notation [6] that allows handling very small and very large numbers. Generally, a floating-point number is composed of three parts: a *sign* that indicates whether the number is positive or negative, a *significand* (aka *mantissa*) that contains significant digits, and an *exponent* that indicates the position of the radix point (decimal point in base 10 or binary point in base 2). The scientific notation of a number is as indicated in formula (1) below.

$$number = -1^{sign} * significand * base^{exponent} \quad (1)$$

The *base* is the *base* of the number system in use. The floating point number for *base* two is called "binary floating-point", for base ten it is called "decimal floating-point." For instance, 12.5 could be represented in "decimal floating-point" notation as *significand* 1.25 with an *exponent* of 1 and a *sign* of 0, and in "binary floating point" notation as *significand* 1.1001 with an exponent of 3 and a *sign* of 0.

The floating-point notation has two advantages over integer data-types. First, real values between integers (real numbers) can be represented. Second, a floating-point number can represent a very large range of values because of the scaling factor (or the sliding window technique). Using sliding windows of precision, some of the less important digits can be sacrificed to let the machine represent a very large number. For example, the number 123,456,321,123 cannot be represented using the integer data-type, but it can be represented in "decimal floating-point" as a *significand* 1.2345632 (the window slides to the left) with an *exponent* of 11 and a *sign* of zero; the machine only needs to store 1.2345632, 11 and 0. The same rule applies to very small numbers; in this case, the window is slid to the right. For example, the number 0.00000001234 can be stored as a *significand* 1.234 with an *exponent* of -8 and a *sign* of zero.

A disadvantage of the floating-point representation in binary machines is that some numbers do not have an exact binary representation. Although a many numbers have a compact and exact decimal representation, they have very long or infinite expansion in binary. As a result, many decimal numbers cannot be represented exactly in binary format, and they have to be stored as approximated binary floating-point numbers in the machine. For example, there is no exact value of decimal 0.1 which has the 32-bit binary representation as 0 01111011 10011001100110011001101 with the exponent 01111011 (123 in decimal) and infinite digits of *significand* 1001 1001 1001 1001 1001 …

The advantage of the sliding window technique is also a disadvantage of the floating-point representation, since it causes the truncation of some of the least significant bits. The number 123,456,321,123 is represented by 1.2345632 x 10^{11}, and the last digits "1123" are lost.

2.1.2 IEEE 754 standard for Floating Point Numbers

IEEE 754 is the IEEE standard for binary floating-point arithmetic established in 1985, and commonly used in most of modern hardware and programming languages. According to the IEEE 754 standard, a floating-point number is encoded into 3 components:

- A one-bit *sign* field: this leftmost bit indicates whether the number is positive (*sign* bit = 0) or negative (*sign* bit = 1).
- An *exponent* field: the exponent can be negative as well as positive (signed). To store the exponent as an unsigned integer, the technique "biasing" is used and the stored exponent is called "biased exponent". In this technique, before storing into a floating-point format, a positive bias number (127 for single, 1023 for double or $2^{n-1} - 1$ for n-bit exponent) is added to the real exponent. For instance, if the exponent is 3, the stored exponent field is 130 for single, and 1026 for double.
- A *significand* field: these bits combined with the implicit leading significant bit with value 1 except for subnormal numbers and zero give the true *significand*.

The distributions of these fields for 32-bit single precision (*float* type in C) and 64-bit double precision (*double* type in C) are indicated in Table 2.

Table 2: Format of single and double precisions in IEEE 754 standard

Format	***Total bits***	***Sign***	***Exponent***	***Significand***	***Smallest***	***Largest***
Single precision	32	1	8	23	~$1.2 * 10^{-38}$	~$3.4 * 10^{38}$
Double Precision	64	1	11	52	~$2.23 * 10^{-308}$	~$1.8 * 10^{308}$

The floating-point notation has a specific representation for the following special numbers: *zero* is represented with *exponent* 0 and *significand* 0; *infinity* is represented with *exponent* 255 and *significand* 0; *NaN (not-a-number)* is represented with stored 255 and non-zero *significand*.

The value of an IEEE floating point number is computed using formula (2) below, in which *m* is the number of significant bits, and *e* is the stored exponent.

$$(-1)^{sign} * (1 + \sum_{i=1}^{m} b_{m-i}\, 2^{-i}) * 2^{(e-bias\ number)} \quad (2)$$

For example:

0 **01111100** 01000000000000000000000 = 1 * (1+ 2^{-2})*$2^{(124-127)}$ = 0.15625

2.1.3 Sources of Numerical Errors

Numerical errors happening during the computations are the combined effect of two types of errors: *round-off errors* (due to limited precision of representation) and *truncation errors* (limited time of computation).

2.1.3.1 Round-off errors

Round-off errors occur because it is impossible to represent exactly all real numbers in binary format. Each machine-hardware uses a finite number of precision bits (*n* bits) to store and manipulate (finite or infinite) real numbers that require more than *n* bits, leading to an approximation.

Although the round-off error can be small for a given numerical step, it can be accumulated and become significant after a number of iterations. Our research focuses on this kind of error.

2.1.3.2 Truncation errors

A truncation error corresponds to the fact that a process is terminated, or a mathematical procedure is approximated after a certain finite number of steps, and the approximation result differs from the mathematical result.

For example, when using logarithms, exponentials, trigonometric functions, hyperbolic functions, the infinite term ∞ is replaced with a finite term *n* ($\sum_{i=0}^{\infty} a_i x^i \rightarrow \sum_{i=0}^{n} a_i x^i$). The truncation error is $\sum_{i=n+1}^{\infty} a_i x^i$.

2.2 Composite floating point number library

Because of the hardware limitation, traditional floating-point numbers cause numerical errors that cannot be accepted in many applications requiring high accuracy. To improve the accuracy of applications, many scientists have explored techniques to extend the available precision in software. In 1971, Dekker [3] introduced a technique for expressing multi-length floating-point arithmetic in terms of single-length floating-point arithmetic. In particular, he illustrated a method to split a floating-point number into two half-length floating-point numbers.

The pseudo-code of the splitting method is below:

```
Error-free split of a floating-point into two parts
[hx, tx] split (float x) {
    c = fl($2^{t1} + 1$); // t1 = 12 for single or t1 = 27 for double
    p = fl(x X c)
    hx = fl(p - (p - x));

    tx = fl(x - hx);

}
```

Dekker's splitting method has been used in several studies, including Thall's work on the use of extended-precision floating-point numbers for GPU computation [4]. More recently, Taufer et al. [1] have redefined the extended-precision arithmetic described in [4] and introduced the composite floating-point arithmetic library that we use in this thesis.

The single composite floating-point precision representation (*float2*) proposed in [1] is a data structure consisting of two single floating-point numbers: a *value* component and an *error* component. The single precision value of a composite precision number is the addition of its value and error components.

Similarly, the double composite floating-point precision (*double2*) uses two double floating-point numbers to represent a number. The data structures of single and double composite floating-point numbers are presented in Table 3.

Table 3: Data structure for single and double precision composite arithmetic.

`struct float2{`
`        float x; // x2.value`
`        float y; //x2.error`
`} x2;`
`float x2 = x2.x + x2.y; // x2.value + x2.error;`
	`struct double2{`
	`        double x; // x2.value`
	`        double y; //x2.error`
	`} x2;`
	`double x2 = x2.x + x2.y; // x2.value + x2.error;`

In the composite precision library in [1], multiple single precision additions, subtractions, multiplications, as well as reciprocal are used to perform single composite precision addition, subtraction, multiplication, and division. Not only do the composite precision algorithms perform the calculation, but they also keep track of the error. Table 4 shows that single composite precision addition and subtraction require four single precision additions and four single precision subtractions; single composite multiplication requires four single precision multiplications and two single precision additions; single composite division needs a single floating-point reciprocal, four single precision multiplications, one single precision addition, and one single precision subtraction. These algorithms obviously slow down the performance because of the extra operations.

Table 4: Algorithms for the single composite floating-point arithmetic

Addition

Pseudo Code	Implementation
`float2 x2,y2,z2`	`float2 x2,y2,z2`
`z2 = x2 + y2`	`float t`
	`z2.value= x2.value + y2.value`
	`t = z2.value - x2.value`
	`z2.error = x2.value`
	`- (z2.value -t)`
	`+ (y2.value -t)`
	`+ x2.error`
	`+ y2.value`

Subtraction

Pseudo Code	Implementation
`float2 x2,y2,z2`	`float2 x2,y2,z2`
`z2 = x2 + y2`	`float t`
	`z2.value= x2.value + y2.value`
	`t = z2.value - x2.value`
	`z2.error = x2.value`
	`- (z2.value -t)`
	`+ (y2.value -t)`
	`+ x2.error`
	`+ y2.value`

Multiplication

Pseudo Code	Implementation
`float2 x2,y2,z2`	`float2 x2,y2,z2`
`z2 = x2 * y2`	`float t`
	`z2.value= x2.value * y2.value`
	`z2.error= x2.value * y2.error`
	`+ x2.error * y2.value`
	`+ x2.error * y2.error`

Division

Pseudo Code	Implementation
`float2 x2,y2,z2`	`float2 x2,y2,z2`
`z2 = x2 / y2`	`float t, s, diff`
	`t = (1/y2.value)`
	`s = t * x2.value`
	`diff = x2.value`
	`- (s * y2.value)`
	`z2.value = s + t * diff`
	`z2.error = t * diff`

2.3 GNU Multiple Precision (GMP) and CUMP

2.3.1 GMP – The GNU Multiple Precision Arithmetic Library

GMP [7] is a free library for *arbitrary-precision arithmetic*, operating on integers, rational numbers, and floating-point numbers. Arbitrary-precision arithmetic, also called big-num arithmetic, multiple precision arithmetic, or sometimes infinite-precision arithmetic, indicates that calculations are performed on numbers whose digits of precision are limited by the available memory of the host system. In principle, arbitrary-precision arithmetic should be able to allocate additional space dynamically whenever the accurate representation of a variable requires it. However, the current version of the GMP library

(GMP 6.0.0) supports the automatic expansion of the precision only of integer and rational numbers. The precision of floating point numbers in GMP has to be chosen statically, and the size of the variables doesn't change after initialization.

GMP floating-point numbers are stored in objects of type *mpf_t* and functions operating on them have the *mpf_* prefix. The GMP floating-point representation is illustrated in Figure 1.

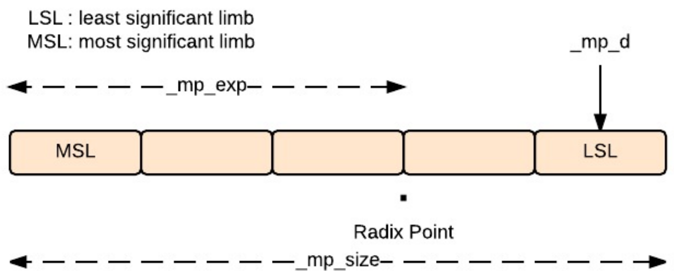


Figure 1: GMP floating point number representation

The GMP floating-point representation is based on the following definitions:

- *limb*: the part of a multi-precision number that fits in a single word. Normally a limb contains 32 or 64 bits. The C type of limb is *mp_limb_t.*
- *_mp_size:* the number of current limbs used to represent a number, or the negative of that if the represented number is negative. If the number is zero: *_mp_size* and *_mp_exp* are *zero, _mp_d* is unused.
- *_mp_prec:* the precision of the mantissa, in limbs. At initialization, given number of precision bits, GMP library computes *mp_prec,* and *(mp_prec + 1)* limbs are allocated to *_mp_d.* If there is a carry while implementing GMP floating point operations, the carry value is stored in the extra limb. *_mp_size* can be smaller than *_mp_prec,* if a represented number need less limbs; *_mp_size* can be larger than *__mp_prec* when we use all *(mp_prec + 1*) allocated limbs.

- *_mp_d:* A pointer to the array of limbs, which is the absolute value of the mantissa. *_mp_d*[0] points to the least significant limb and *_mp_d*[*ABS(_mp_size)-1*] points to the most significant.
- *_mp_exp:* The exponent, in limbs, decides the position of the implied radix point. If *_mp_exp* is zero, the radix point is just above the most significant limb. If *_mp_exp* is positive, the radix point offset is between the most significant limbs and the least significant limbs. Negative exponents shows that a radix point is further above the highest limb.

2.3.2 CUMP

The CUDA Multiple Precision Arithmetic Library (CUMP) [10] is a free library for arbitrary precision arithmetic on CUDA, operating on floating point numbers. It is based on the GNU MP library (GMP). CUMP provides functions for host and device codes, the former operating on CPU and the latter operating on GPU.

CUMP supports only a restricted amount of the functionality of the original GMP library. Specifically, in the current version CUMP only supports three arithmetic operations: addition, subtraction, and multiplication.

2.4 Introductions to GPUs

Graphic Processing Units (GPUs) were originally designed for graphics processing. Nowadays, modern GPUs are not only a very powerful computer-graphics and image-processing engine, but also an efficient accelerator for parallel computing and data intensive applications. Thanks to the rapid increase in their computational power and programmability (through the advent of the CUDA programming model), more and more scientists have progressively accelerated their applications on GPUs. In this section, we give a brief introduction to NVIDIA GPUs that we use in our research.

2.4.1 NVIDIA GPU Architecture

Modern NVIDIA GPUs are composed of multiple highly threaded Streaming Processors (SMs). Figure 2 shows the architecture of a Fermi GPU that consists of 16 SMs.

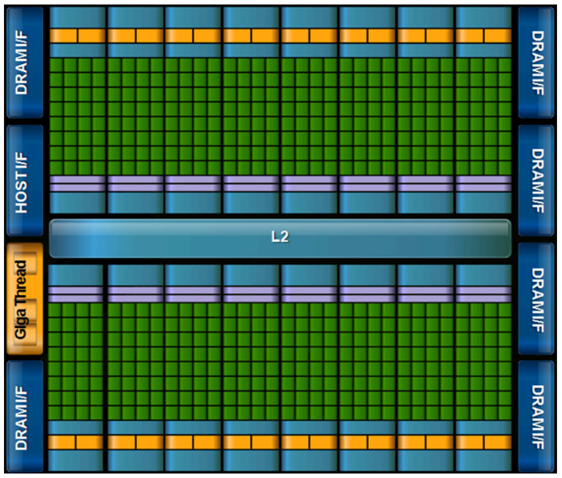


Figure 2: NVIDIA Fermi Architecture

The general architecture of a single SM contains:

- Several Streaming Processors (SPs) (also called single-precision CUDA cores),
- Double-precision units (DFUs),
- Special function units (SFUs),
- Load/store units (LD/ST),
- A register file,
- A shared memory/cache.

Figure 3 shows the design of a SM in the Fermi architecture.

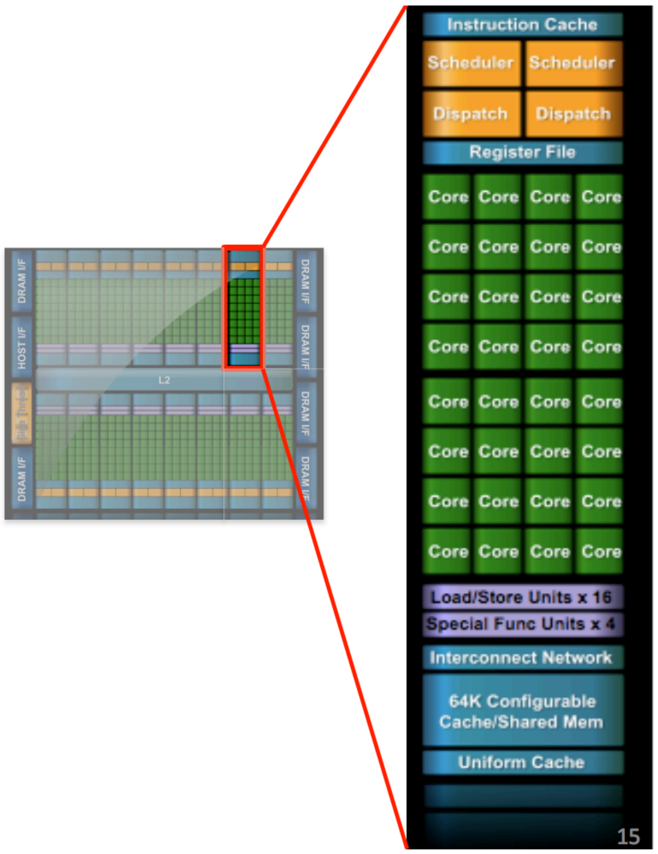


Figure 3: NVIDIA Fermi Streaming Multiprocessor

Table 5 provides a brief comparison between the NVIDIA Fermi and Kepler GPU architectures. From Fermi to Kepler, the number of CUDA cores has increased by a factor 6.42 (from 448 to 2880 cores). Higher clock rates allow faster instruction execution by each core, but lead to higher power consumption. To limit the power consumption, the clock frequency has been slightly reduced moving from Fermi to Kepler GPUs.

Table 5: Fermi vs. Kepler features

	Fermi GF100 [11][13] (Tesla C2070)	*Kepler GK110 [12][14] (Tesla K40c)*
Number of SMs	14	15
SPs per SM	32	192
DFUs per SM	16	64
SFUs per SM	4	32
LD/STs per SM	16	32
Registers per block	32768	65536
Shared memory per block	49152 bytes	49152 bytes
GPU max clock rate	1147 MHz (1.15 GHz)	745 MHz (0.75 GHz)
Peak double-precision floating point performance	515 GFLOPS	1.43 TFLOPS
Peak single-precision floating point performance	1 TFLOPS	4.29 TFLOPS
Warp Schedulers per SM	2	4
Dispatch unit per SM	2	8
Max of active threads per SM	1536	2048
Max of threads per block	1024	1024
Max of active blocks per SM	8	16

2.4.2 CUDA programming model

For both Kepler and Fermi GPUs, a CUDA application generally spawns hundreds to thousands of threads to populate the SM and hide memory accesses/computation pipeline latency. From the programmer's perspective, the work is partitions across threads, the threads are grouped into thread blocks, and the thread blocks are grouped into grids. Each block is mapped to one SM at execution time, and threads within a block are split into warps. The warp is the basic scheduling unit of NVIDIA GPUs, and contains 32 threads. Because the scheduler issues instructions in warps, the block-size (number of threads per block) should be a multiple of the *warp-size* (32 threads) to avoid wasting threads. For example, if the kernel configuration has a *block-size* 16 threads, the instruction is still issued to 32 cores, and 16 cores are wasted. The *grid-size* can be assigned based on the number of active threads and blocks on a SM.

In addition, when considering occupancy and massive parallelism, the registers and shared memory resource are also significant. If a kernel requires too many registers, it limits the number of active threads. This limitation will be explored in Chapter 5.

2.4.3 Floating point for NVIDIA GPUs

The capabilities of NVIDIA GPUs have been expanded in each hardware generation from only supporting single precision in early NVIDIA architectures, to fully supporting IEEE single, double precision, and including FMA (Fused-Multiply-Add) operations in modern generations such as Fermi and Kepler GPUs. CUDA classifies different GPU generations using the compute-capability number [16]; the compute capability of a GPU device can be queried by a specific CUDA function call. Below, we detail the GPU support provided in GPU with different compute capabilities.

- Compute capability 1.2 and below only support single precision floating point arithmetic. Moreover, on these GPU devices not all single precision operations are IEEE compliant, possibly leading to high level of inaccuracy.
- Compute capability 1.3 supports both single and double precision arithmetic, and offers double-precision FMA operations. Single precision in these devices keeps unchanged from previous compute capabilities. Double precision in compute capability 1.3 devices is IEEE compliant. Double-precision FMA operations combine a multiply and an addition with only one rounding step; the resulting operation is faster and more accurate than separate multiply and addition. Compute capability 1.3 does not support single precision FMA.
- Compute capability 2.0 and above fully support IEEE compliant single and double precision arithmetic, and include both single and double FMA operations.

The experiments in this thesis are performed on Tesla C2050/C2070 (Fermi GPU with compute capability 2.0) and Tesla K40C (Kepler GPU with compute capability 3.5), both supporting IEEE-compliant single and double precision arithmetic. The newest NVIDIA

GPU architecture – called Maxwell – also supports 16-bit precision, but we will not focus on this device. Table 5 shows additional features related to floating-point support on Fermi and Kepler architectures; namely: the number of single/double precision units (SPs/DFUs) per SM (SPs/DFUs in Fermi is 32/16, in Kepler is 192/64), and the peak performance of floating-point operations.

Table 6 reports the latencies of 32-bit floating-point operations over Fermi and Kepler NVIDIA GPUs based on the research in [17].

Table 6: Latencies (clock cycles) of math data-path 32-bit floating-point operations over Fermi and Kepler NVIDIA GPUs

Operation	*Fermi GF106*	*Kepler GK104*
ADD, SUB	16	9
MAX, MIN	20	9
MAD	18	9
MUL	16	9
DIV	1038	758
__fadd_*()	16	9
__fmul_*()	16	9
__fdivdef()	95	41
__sinf(), __cosf()	42	18
__tanf()	124	58
__exp2f()	98	49
__expf(), __exp10f()	114	58
__log2f()	46	22
__logf(), __log10f()	94	49
__powf()	143	62
sqrt()	216	181

Chapter 3 Research Questions and Methodology

3.1 Research questions

In this thesis, we focus on answering the following questions:

- How GPU multithreading affects the tradeoff between accuracy and performance?
- How the arithmetic intensity of a program affects the performance/accuracy tradeoff on GPU?
- How the kind of floating-point operation executed in a program affects the accuracy?
- Why and when one precision should be replaced by another precision?
- How the use of division in CUDA programming affects the accuracy and performance of the program?

3.2 Methodology

To answer the above questions, it is important to thoughtfully select appropriate benchmark applications. Naturally, the combination of these benchmarks should reflect all angles of our research. We selected the following benchmark applications.

- *Global summation* [1]: this numerical benchmark performs the summation of an array of values whose "accurate" sum is known to be equal to zero. Our goal is to observe the accuracy of different precisions, and this benchmark gives us a good reference – the zero "correct" sum. We modified the global summation benchmark described in [1] to use single precision, double precision, composite precision and GMP for both original single-block version and modified multi-block version. Overall, this benchmark allows answering most of the questions related to addition and subtraction operations.

- *LU Decomposition (LUD)* and *Gaussian Elimination* benchmarks from the Rodinia benchmark suite [2]: the Rodinia benchmark suite is very popular in high performance computing. LUD and Gaussian Elimination are two applications that use floating-point numbers. These applications use the combination of different operations (addition, subtraction, multiplication, division), and allow us to evaluate the general accuracy/performance tradeoff of the different floating-point precisions in consideration. These applications have multi-block configurations that we can directly use to analyze the performance of the programs at different degrees of multithreading. We modified these applications to use different floating-point precisions. Besides using the input matrices publicly available for these benchmarks, we generate random matrices that include elements with various orders of magnitude so as to study their effect on the accuracy of the results.
- *Do/undo* benchmark [1]: this application performs a combination of multiplication and division operations. Specifically, a reference x variable is repeatedly multiplied and divided by a sequence of y variables. The expected result of the computation is the original x. This do/undo benchmark enables us to analyze the effect of different types of division on accuracy.

After modifying the benchmark applications above to use the considered floating-point precisions and datasets, we analyze their register and memory requirements and, with the aid of the CUDA occupancy calculator, we determine degrees of multithreading that should be considered in the analysis. In addition, we profile the applications and study their PTX assembly code to determine the arithmetic intensity of each application and the number and kind of instructions it performs.

For applications for which the accurate results are not known a priori, we use the result of the GMP library as a reference.

The benchmark applications above are only the starting point of our analysis. To be able to generalize our analysis and findings, and to better understand our observations, we constructed two micro-benchmarks:

- Micro-benchmark for analyzing the affect of arithmetic intensity on the performance/accuracy tradeoff on GPU.
- Micro-benchmark for explaining the behavior of composite precision multiplication and division.

Chapter 4 Related work

We first explored the single floating-point composite precision library [1]. As explained in Chapter 2, a single floating-point composite number is composed of two single precision floating-point numbers, the *value* and the *error*. The composite number value, n, is the sum of two single floating-point numbers as in formula (1).

$$n = n_{value} + n_{error} \text{ (1)}$$

The approximation error of the arithmetic operation such as the sum or the product of two numbers is stored in n_{error}. Our thought was that we could monitor the change of n_{error} and decided to increase the precision when the error reached a threshold level. The idea was more stable when we implemented a floating-point program using GNU multiple precision library (GMP). This arbitrary precision arithmetic library provided us an unlimited precision bits to represent a real number, and so gave the accurate results. In theory, GMP library should automatically change the precision when needed, but the truth was that GMP only supported the dynamic change precision bits for integer data-types. For floating-point numbers, we had to declare the number of necessary precision bits. Because of that reason we thought about a library that could change the precision of floating-point numbers dynamically (the precision we chose included single floating-point (*float*), double floating-point (*double*), composite number (*float2*, *double2*)). We then began to study the necessary materials and tools to support our ideas. During this period of the research, we have learned and practiced the function of extracting floating-point numbers into three components: *sign*, *exponent* and *mantissa*; we have also learned the techniques to split a floating-point number into two floating-point numbers [3][4][5].

Beside of those papers mentioned in this section, the dynamic tool – PRECIMONIOUS [8] – is one of my motivations to think about the idea of dynamic library. This tool has four main components to assist users for tuning the precision of floating-point numbers. In the first component, the search file is created based on the input program in LLVM bitcode format and the number of floating-point types, and this search file contains all variables needed to be tuned. In the second component, all valid type configurations are found using the authors' LCCSearch algorithm. In the third component, all program versions corresponding to all valid configurations are generated in the LLVM bitcode format. In the last component, results produced by all new program versions are compared with the result of original version, and the running times of all versions are computed and compared with the running time of original version. After checking and comparing, the PRECIMONIOUS give proposed configuration that can give the correct answer with appropriated performance. However, this tool is only used on CPU, and has many limitations including modifying at source code level, lack of communication between variable etc. The idea about dynamically changing precision that can be used on GPUs is formed.

Unfortunately, from the basic functions we started to monitor the error and built our dynamic library, but we received unexpected results. Figure 4 show that the n_{error} in composite library was unpredictable because it could be both negative and positive number, thus the error could not increase gradually. After a period of time trying many ways to have better control on n_{error} and getting disappointed results, we had to change our direction and start our new idea that is to make a study considering accuracy and runtime for multithreaded applications with different size of dataset. This is the basic idea for this thesis.

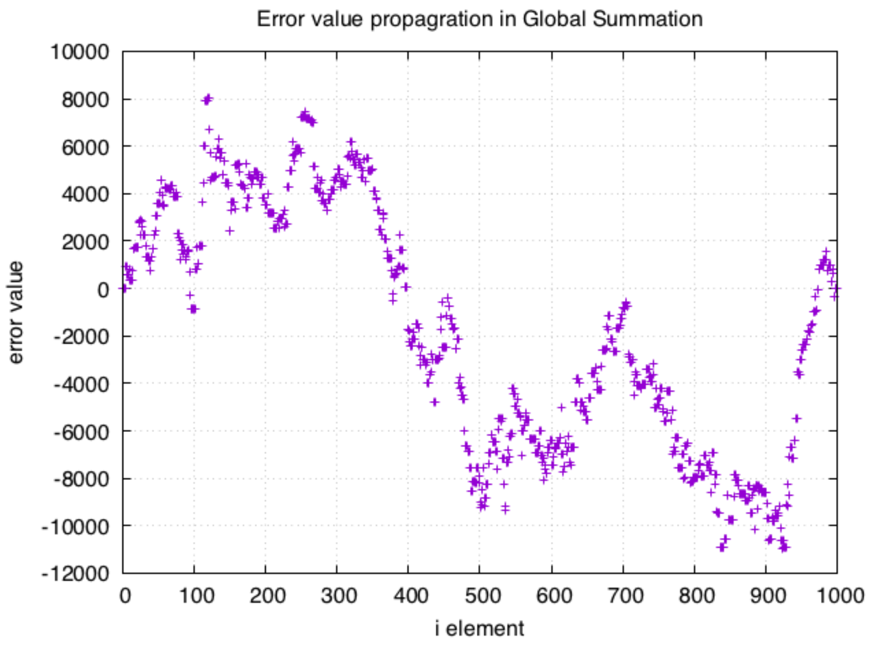


Figure 4: Error value propagation in Global summation

Chapter 5 Global Summation

This chapter presents an analysis of numerical accuracy and performance issues found in the global summation benchmark described by Taufer *et al*. [1] due to the use of floating point arithmetic. We analyze the global summation benchmark not only using single precision floating-point (*float*) and single composite precision (*float2*) arithmetic as in [1], but also using multiple precision (GMP *mpf_t*), double precision floating-point (*double*), and double composite precision (*double2*) arithmetic. First, we modify the global summation program to use multiple-precision arithmetic: namely, the GMP library on CPU, and the CUMP library on GPU. On GPU, we extend the single thread-block GPU kernel proposed in [1] so as to allow also experiments with multiple thread-blocks. Then, we extend the global summation kernel to use *double* and *double2* (beside *float* and *float2*) arithmetic. Our goal is to study the performance-accuracy tradeoff of global summation at different degrees of multithreading.

5.1 Global summation with GMP and CUMP

The global summation program was introduced to evaluate the composite precision library described in [1]. This program calculates the summation of an array of floating point values on CPU and GPU. The input array can be configured to contain numbers of various orders of magnitude; the content of the array is automatically generated so as to have a known accurate sum of zero. When using floating-point arithmetic, the sum of a very large and a very small number may lead to the small term to be neglected (cancellation). This, in turn, may lead to inaccuracy problems in the global summation of sets of numbers with different orders of magnitude. The resulting inaccuracy depends on the arithmetic precision and the order of magnitude of the elements in the array.

Because of the two properties: (1) the array contains a subset of very small numbers and a subset of very large numbers, and the magnitude of small and large numbers can be configured; (2) the correct summation of the array is zero, we choose this program to start our study.

To learn how multiple precision numbers can improve the accuracy and study their effect on the performance, we first analyze the GMP – CUMP version of the global summation when increasing the number of precision bits, and for different gaps between the small and large numbers. Specifically, we observe the program on both CPU and GPU with four GMP configurations: *32-bit* (*2x64-bit limbs*), *64-bit* (*2x64-bit limbs*), *128-bit* (*3x64-bit limbs*), and *256-bit* (*5x64-bit limbs*). In addition an, we consider five ranges of input intervals as below:

- Range 1: Small number $(10^{-01}, 10^{+00})$ and Large number $(10^{+00}, 10^{+01})$
- Range 2: Small number $(10^{-04}, 10^{-03})$ and Large number $(10^{+03}, 10^{+04})$
- Range 3: Small number $(10^{-13}, 10^{-12})$ and Large number $(10^{+12}, 10^{+13})$
- Range 4: Small number $(10^{-19}, 10^{-18})$ and Large number $(10^{+18}, 10^{+19})$
- Range 5: Small number $(10^{-37}, 10^{-36})$ and Large number $(10^{+36}, 10^{+37})$

Table 7: Accuracy of global summation using GMP and CUMP. Device = Tesla C2075

Array Size = 1024 element				
Sequential (CPU)				
Format:	*mpf_t 64*	*mpf_t 32*	*mpf_t 128*	*mpf_t 256*
Range 1: (10^{-01}, 10^{+00}) & (10^{+00}, 10^{+01})	0.00E+00	0.00E+00	0.00E+00	0.00E+00
Range 2: (10^{-04}, 10^{-03}) & (10^{+03}, 10^{+04})	4.07E-19	0.00E+00	0.00E+00	0.00E+00
Range 3: (10^{-13}, 10^{-12}) & (10^{+12}, 10^{+13})	6.44E-18	2.01E-18	0.00E+00	0.00E+00
Range 4: (10^{-19}, 10^{-18}) & (10^{+18}, 10^{+19})	9.60E+01	9.60E+01	6.51E-18	0.00E+00
Range 5: (10^{-37}, 10^{-36}) & (10^{+36}, 10^{+37})	0.00E+00	0.00E+00	5.34E-18	0.00E+00
Parallel (GPU)				
Format:	*cump 64*	*cump_32*	*cump_128*	*cump_256*
Range 1: (10^{-01}, 10^{+00}) & (10^{+00}, 10^{+01})	0.00E+00	0.00E+00	0.00E+00	0.00E+00
Range 2: (10^{-04}, 10^{-03}) & (10^{+03}, 10^{+04})	2.67E-19	0.00E+00	0.00E+00	0.00E+00
Range 3: (10^{-13}, 10^{-12}) & (10^{+12}, 10^{+13})	3.18E-18	1.06E-18	0.00E+00	0.00E+00
Range 4: (10^{-19}, 10^{-18}) & (10^{+18}, 10^{+19})	5.98E+01	5.98E+01	3.77E-18	0.00E+00
Range 5: (10^{-37}, 10^{-36}) & (10^{+36}, 10^{+37})	0.00E+00	0.00E+00	2.89E-18	0.00E+00
Array Size = 1,048,576 elements				
Sequential (CPU)				
Format:	*mpf_t 64*	*mpf_t 32*	*mpf_t 128*	*mpf_t 256*
Range 1: (10^{-01}, 10^{+00}) & (10^{+00}, 10^{+01})	0.00E+00	0.00E+00	0.00E+00	0.00E+00
Range 2: (10^{-04}, 10^{-03}) & (10^{+03}, 10^{+04})	1.58E-16	0.00E+00	0.00E+00	0.00E+00
Range 3: (10^{-13}, 10^{-12}) & (10^{+12}, 10^{+13})	2.62E-15	6.77E-16	0.00E+00	0.00E+00
Range 4: (10^{-19}, 10^{-18}) & (10^{+18}, 10^{+19})	3.85E+04	3.85E+04	2.61E-15	0.00E+00
Range 5: (10^{-37}, 10^{-36}) & (10^{+36}, 10^{+37})	0.00E+00	0.00E+00	7.95E+01	1.48E-34
Parallel (GPU)				
Format:	*cump 64*	*cump_32*	*cump_128*	*cump_256*
Range 1: (10^{-01}, 10^{+00}) & (10^{+00}, 10^{+01})	0.00E+00	0.00E+00	0.00E+00	0.00E+00
Range 2: (10^{-04}, 10^{-03}) & (10^{+03}, 10^{+04})	3.53E-17	0.00E+00	0.00E+00	0.00E+00
Range 3: (10^{-13}, 10^{-12}) & (10^{+12}, 10^{+13})	5.61E-16	1.45E-16	0.00E+00	0.00E+00
Range 4: (10^{-19}, 10^{-18}) & (10^{+18}, 10^{+19})	8.20E+03	8.20E+03	5.64E-16	0.00E+00
Range 5: (10^{-37}, 10^{-36}) & (10^{+36}, 10^{+37})	0.00E+00	0.00E+00	6.94E+01	3.89E-35

From Table 7 we can observe that, on the 1024-element array in *Range 1*, the program gives accurate results in all cases. The global summation program starts to provide inaccurate results when the input sequence is created in *Range 2*. While the *32-bit* and *64-bit GMP* precisions have the same number of *limbs* (2 *limbs*), we find that *64-bit* results are less accurate than *32-bit* results. This is explained as follow: in our program, we generate *32-bit GMP* arrays from *float* arrays, and *64-bit GMP* arrays from *double* arrays. A single precision number has fewer significant digits after the radix point than a double number; for example: compare 1.12340808090000000 *float* with 1.12340808080870080078865200 1 *double*. The non-zero digits after the radix point in the *64-bit GMP* array contribute to the inaccuracy of the result.

Next, we progressively increase the (positive and negative) order of magnitude of the intervals. Our data show that *32-bit GMP* precision leads to inaccurate results when the input sequence is in *Range 3*, and *128-bit GMP* precision causes error when the input sequence is in *Range 4*.

In the case of 1024-element arrays, when increasing the gap between the intervals from minimum value of *float* to maximum value of *float*, *256-bit GMP* precision still produces the correct sum. Therefore, to illustrate the inaccuracy problem when using *256-bit GMP,* we have to use larger inputs. For example, *256-bit GMP* produces inaccuracy when the array size is 1,048,576 elements (10^{20} elements), and the values of elements are in *Range 5*.

All of the above results demonstrate that we can use GMP and CUMP libraries with appropriate precision bits to increase accuracy for applications involving floating-point numbers. However, we do not choose to use these libraries for all applications because of the trade-off between accuracy and performance. To learn more about this issue, we monitor the execution time of the program and report the results in Table 8.

Table 8: Execution time of GMP and CUMP vs. double precision.

Array size = 1024									
Sequential (CPU)									
	mpf32	mpf64	mpf128	mpf256	double	Execution time of gmp / double			
						32 bit	64 bit	128 bit	256 bit
	0.0562	0.0563	0.0583	0.0623	0.0049	11.4	11.4	11.8	12.7
Parallel (GPU)									
# of threads	cump32	cump64	cump128	cump256	double	Execution time of cump / double			
						32 bit	64 bit	128 bit	256 bit
1	4.7144	4.7408	5.3914	5.7841	0.1499	31.5	31.6	36.0	38.6
32	0.7185	0.7357	0.8867	0.9298	0.0740	9.7	9.9	12.0	12.6
512	0.0183	0.0305	0.0211	0.0226	0.0410	0.4	0.7	0.5	0.6
Array size = 1048576									
Sequential (CPU)									
	mpft32	mpft64	mpft128	mpft256	double	Execution time of gmp / double			
						32 bit	64 bit	128 bit	256 bit
	69.3969	69.8896	69.1415	70.2967	3.9748	17.5	17.6	17.4	17.7
Parallel (GPU)									
# of threads	cump32	cump64	cump128	cump256	double	Execution time of cump / double			
						32 bit	64 bit	128 bit	256 bit
1	4925.7116	4947.8438	5522.3788	6067.5971	130.1221	37.9	38.0	42.4	46.6
32	912.3816	915.9377	1054.0357	1150.5203	58.8911	15.5	15.6	17.9	19.5
512	99.7857	100.5909	117.0254	130.4637	7.4749	13.3	13.5	15.7	17.5

Table 8 shows that, for sequential summation, the performance of *GMP* is lower than that of *double* precision arithmetic by a factor *12-18x*. It also shows that, for parallel summation, the performance of *CUMP* is lower than that of double precision arithmetic by a factor *47x*. This proves that the use of *GMP* and *CUMP* is suitable only when an application really needs very high accuracy, and significant execution time degradations are not an important issue.

In addition, during the study of the *CUMP* library, we observed that *CUMP* kernels use many registers (Table 9). The maximum number of registers per block is 65,536 on the Kepler K40C GPU and 32,768 on the Fermi C2075 GPU used in this study. The register requirement limits the size of the thread-block that can be configured for the summation kernel to 65,536/71 ~ 923 for Kepler K40C and 32,768/63 ~ 520 for Fermi C2075.

Table 9: Number of registers using by kernels

	CUMP Kernel	*Float2 Kernel*	*Double2 Kernel*	*Float kernel*	*Double kernel*
Sm_20	63	14	21	9	12
Sm-35	71	15	28	10	12

Since the considered global summation kernel uses only a single thread-block, this block-size limitation does not allow fully utilizing the GPU hardware, leading to performance limitations. To solve this problem, we modify the global summation program to allow a multi-block kernel configuration. As shown in Table 10, the multi-block program can improve our performance by hiding latency, and we can get the best performance at *grid-size = 64* and *block-size = 32*.

Table 10: Execution time (in seconds) of global summation using CUMP with various kernel configurations. Array size = 1,048,576. Device = Tesla C2075

# of blocks	*# of threads/block*	*cump_32*	*cump_64*	*cump_128*	*cump_256*
1	1	5583.4224	5571.0618	5898.5952	5900.0278
1	32	1093.5682	1000.4324	1316.6489	1315.535
1	512	115.7171	112.3185	131.2939	131.3426
32	64	24.0186	22.9559	28.7283	28.7056
64	32	23.5261	22.6891	28.1209	28.0694
128	16	32.6578	30.9646	38.4514	38.3379
256	8	34.9698	33.3845	41.5386	41.2134
512	4	37.9684	36.5593	43.0963	43.0193

5.2 Global summation with single/double floating point and composite precision numbers

In this section, we aim to evaluate global summation on GPU with floating-point representations that use lower number of precision bits than multiple-precision arithmetic, namely *float*, *double*, *float2*, and *double2*. As done for multiple-precision arithmetic, we study the effect of the arithmetic precision on accuracy and performance.

First, we observe the change in accuracy on 8M-element arrays when varying the absolute order of magnitude of the input intervals (Table 11). We recall that the expected result of the global summation is zero. However, all precisions but *256-bit CUMP* produce non-zero results. The results of the summation correspond to the cumulative error of the program, and these errors become larger when increasing the range of intervals. We can arrange precisions in ascending order of accuracy as follows: *float* (average error: 10^{+01}), *float2* (average error: 10^{-03}), *double* (average error: 10^{-08}), *double2* (average error: 10^{-16}), and *256-bit CUMP* (average error: 0).

Table 11: Accuracy of global summation using various precisions and five input ranges.

Interval range	*Float*	*Double*	*Float2*	*Double2*	*256 bit CUMP*
Range 1: $(10^{-2}, 10^{-1})$ & $(10^{+1}, 10^{+2})$	1.38E-01	3.18E-10	1.17E-05	0.00E+00	0.00E+00
Range 2: $(10^{-3}, 10^{-2})$ & $(10^{+2}, 10^{+3})$	1.51E+00	2.75E-09	9.84E-05	3.78E-18	0.00E+00
Range 3: $(10^{-4}, 10^{-3})$ & $(10^{+3}, 10^{+4})$	1.43E+01	2.57E-08	1.38E-03	1.44E-16	0.00E+00
Range 4: $(10^{-5}, 10^{-4})$ & $(10^{+4}, 10^{+5})$	1.10E+02	2.51E-07	1.53E-02	2.01E-15	0.00E+00
Range 5: $(10^{-6}, 10^{-5})$ & $(10^{+5}, 10^{+6})$	1.21E+03	3.64E-06	6.06E-03	1.08E-14	0.00E+00

Figure 5 provides a graphical illustration of the error in logarithmic scale. In this and the other figures in this thesis, for the purpose of drawing the chart in logarithmic scale, the accurate sum (zero) is represented by 10^{-25} rather than by $10^{-\infty}$. Each group of bars in Figure 5 represents the errors generated using the five considered arithmetic precisions in the same range of intervals. From the left to the right of Figure 5, the gap between the small and large input intervals increases. As can be seen, the accuracy of the results increases moving from *float*, to *float2*, to *double*, to *double2*, to *CUMP*. In addition, a large gap between the input intervals leads to worse accuracy for all precisions but *CUMP*, which always provides accurate results.

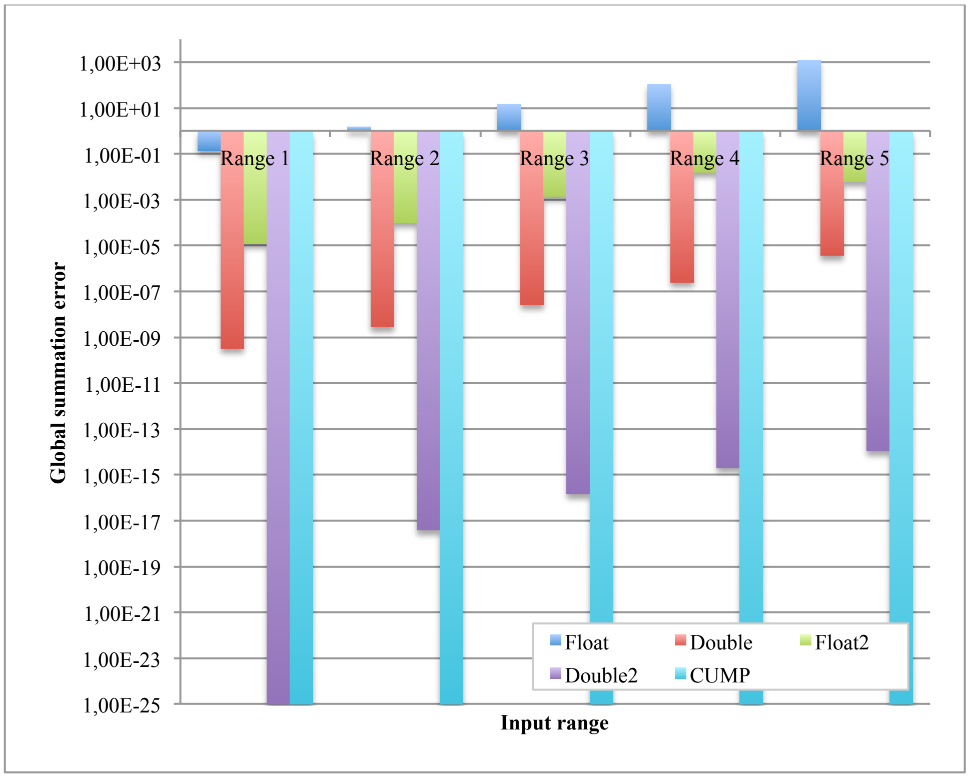


Figure 5: Accuracy of global summation

In order to evaluate performance, we start with the *one-block* kernel and increase the number of threads so as to determine the optimal *block-size*. Then, we fix the *block-size*, and increase the number of blocks (*grid-size*) so as to fully populate our GPUs. Table 12 shows the execution times reported on an 8M-element array (the order of magnitude of the elements in the array is irrelevant to performance). As can be seen, a *block-size* of 32 (equal to the *warp-size*) is in all cases optimal for performance. In addition, increasing the number of blocks up to 64 leads to performance improvements. However, since 64 blocks allow good GPU utilization and memory latency hiding, no performance improvements are observed when further increasing the number of blocks.

Table 12: Execution time of 8M-array global summation using various precisions and different kernel configuration, intervals: (10^{-6}, 10^{-5}) & (10^{+5}, 10^{+6}), device = Tesla C2070.

# of blocks	*# of threads /block*	*float*	*double*	*float2*	*double2*	*(D-F)/F*	*(F2-F)/F*	*(D2-F)/F*
1	1	1518.6588	1721.518	2479.5372	2946.7413	0.13	0.63	0.94
1	32	160.0303	165.8942	189.0708	200.9911	0.04	0.18	0.26
1	64	79.5287	82.9776	94.8857	101.3879	0.04	0.19	0.27
1	128	39.927	41.7331	47.7054	51.8449	0.05	0.19	0.3
1	1024	5.3105	5.8793	6.8778	10.0757	0.11	0.3	0.9
16	32	10.1154	10.5467	12.08	13.0258	0.04	0.19	0.29
32	32	5.1069	5.3986	6.1285	6.8715	0.06	0.2	0.35
64	32	2.6152	2.9384	3.2512	3.9833	0.12	0.24	0.52
128	32	2.6544	3.0005	3.326	4.1112	0.13	0.25	0.55

Figure 6 summarizes the execution time (in seconds) of the different implementations when increasing the number of 32-thread-blocks from 1 to 128. We make the following observations. First, multithreading improves performance for all arithmetic precisions. Second, *256-bit CUMP* reports far worse performance than all other precisions (note that the y-axis is in logarithmic scale). Third, *double* reports only a slight performance degradation compared to *float*, and composite precision reports only a slight performance degradation compared to *float* and *double*. Finally, as explained above, the last two groups of bars show that no performance improvements are observe beyond 128 thread-blocks.

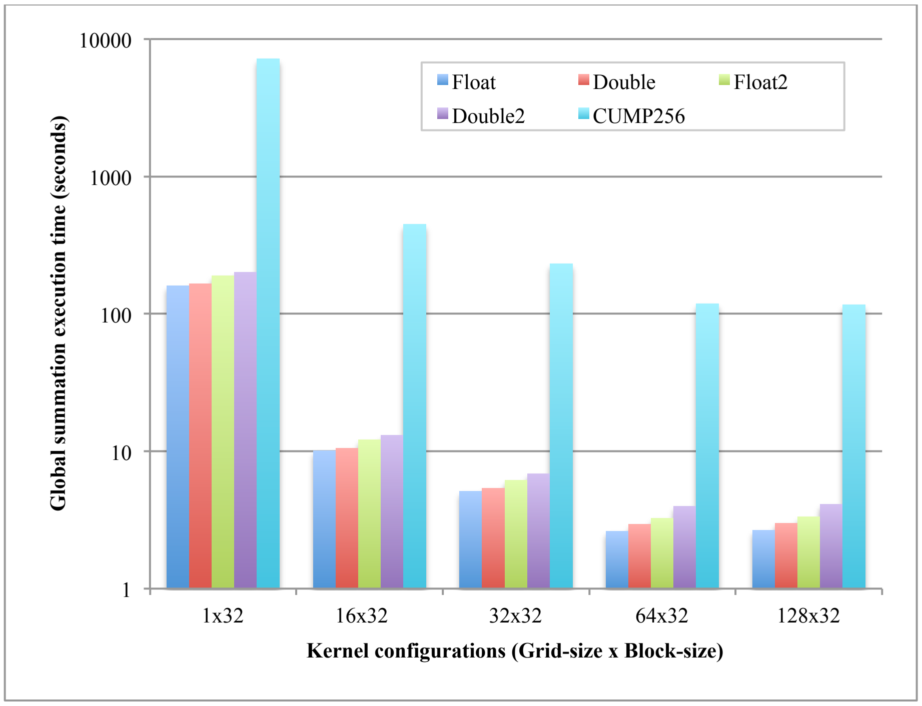


Figure 6: Execution time (seconds) of 8M-array global summation using various precisions and different kernel configurations.

Let us now analyze the results reported using composite arithmetic precision in more depth. Composite precision addition requires eight single/double precision additions/subtractions, and therefore the expected running time of composite precision addition should be eight times larger than the running time of regular floating point arithmetic. However, in the worst case (*block-size* = 1 and *grid-size* = 1) that reflects sequential computation, the running time of single composite precision is 1.63 times that of single floating point; the running time of double composite precision is around 1.94 times that of single floating point; and the running time of single and double floating point precisions are almost the same. In order to understand these observations, we should answer two questions:

(1) Why is standard single precision only slightly slower than standard double precision arithmetic?

(2) Why is the running time of composite precisions (*float2*/*double2*) not eight times larger than that of standard floating point (*float*/*double*)?

To answer these questions, we generate the PTX assembly codes of four global summation kernels as below:

Single floating-point kernel

```
BB39_2:
        mov.u32         %r1, %r9;
        cvt.s64.s32     %rd8, %r1;
        add.s64         %rd9, %rd8, %rd2;
        shl.b64         %rd10, %rd9, 2;
        add.s64         %rd11, %rd1, %rd10;
        ld.global.f32   %f4, [%rd11];
        add.f32         %f5, %f4, %f5;
        st.global.f32   [%rd3], %f5;
        add.s32         %r2, %r1, %r4;
        setp.lt.s32     %p2, %r2, %r3;
        mov.u32         %r9, %r2;
        @%p2 bra        BB39_2;
```

Double floating-point kernel

```
BB37_2:
        mov.u32         %r1, %r9;
        cvt.s64.s32     %rd8, %r1;
        add.s64         %rd9, %rd8, %rd2;
        shl.b64         %rd10, %rd9, 3;
        add.s64         %rd11, %rd1, %rd10;
        ld.global.f64   %fd4, [%rd11];
        add.f64          %fd5, %fd4, %fd5;
        st.global.f64    [%rd3], %fd5;
        add.s32          %r2, %r1, %r4;
        setp.lt.s32      %p2, %r2, %r3;
        mov.u32          %r9, %r2;
        @%p2 bra         BB37_2;
```

Single composite number kernel

```
BB41_2:
        mov.u32          %r1, %r9;
        mov.f32          %f3, %f19;
        cvt.s64.s32      %rd8, %r1;
        add.s64          %rd9, %rd8, %rd2;
        shl.b64          %rd10, %rd9, 3;
        add.s64          %rd11, %rd1, %rd10;
        ld.global.v2.f32        {%f9, %f10}, [%rd11];
        add.f32          %f19, %f3, %f9;
        sub.f32          %f13, %f19, %f3;
        sub.f32          %f14, %f19, %f13;
        sub.f32          %f15, %f3, %f14;
        sub.f32          %f16, %f9, %f13;
        add.f32          %f17, %f16, %f15;
        add.f32          %f18, %f20, %f17;
        add.f32          %f20, %f10, %f18;
        st.global.v2.f32        [%rd3], {%f19, %f20};
        add.s32          %r2, %r1, %r4;
        setp.lt.s32      %p2, %r2, %r3;
        mov.u32          %r9, %r2;
        @%p2 bra         BB41_2;
```

<u>Double composite number kernel</u>

```
BB43_2:
        mov.u32          %r1, %r9;
        mov.f64          %fd3, %fd19;
        cvt.s64.s32      %rd8, %r1;
        add.s64          %rd9, %rd8, %rd2;
        shl.b64          %rd10, %rd9, 4;
        add.s64          %rd11, %rd1, %rd10;
        ld.global.v2.f64         {%fd9, %fd10}, [%rd11];
        add.f64          %fd19, %fd3, %fd9;
        sub.f64          %fd13, %fd19, %fd3;
        sub.f64          %fd14, %fd19, %fd13;
        sub.f64          %fd15, %fd3, %fd14;
        sub.f64          %fd16, %fd9, %fd13;
        add.f64          %fd17, %fd16, %fd15;
        add.f64          %fd18, %fd20, %fd17;
        add.f64          %fd20, %fd10, %fd18;
        st.global.v2.f64         [%rd3], {%fd19, %fd20};
        add.s32          %r2, %r1, %r4;
        setp.lt.s32      %p2, %r2, %r3;
        mov.u32          %r9, %r2;
        @%p2 bra         BB43_2;
```

Each iteration needs to access global memory by using one LOAD instruction at beginning and one STORE instruction at the end in all four kernels. LOAD/STORE instructions usually cost from *400* to *800* clock cycles while arithmetic latency is only *16-22* clock cycles. Thus, the execution time of the global summation kernel is dominated by the latency of the memory operations (in other words, the kernel has low arithmetic intensity). Additionally, in modern devices, double floating-point arithmetic can be as fast as single floating-point arithmetic. These facts answer the first question.

As confirmed by the PTX assembly codes above, single and double composite precision kernels have seven additional floating-point operations between the LOAD and STORE instructions. However, some of these operations can be issued in parallel to different functional units. Moreover, the considered GPUs have a 128-byte cache line. As a consequence, in *float*, *double* and *float2* kernels, LOAD instructions can save memory access time by hitting L1/L2 caches. These facts answer the second question.

Finally, in Figure 7, 8, 9, 10, and 11 we study the performance-accuracy tradeoff. These Figures show that curves of accuracy vs. running time tend to go upward because the accuracy of the program worsened when we increase gaps between small and large elements in input array. For example, in Figure 7, accuracy vs. running time curve of *double* is a parallel line to the horizontal axis at *x*-coordinate *(-25)* pointing out *double2* precision provides accurate results; Then, we increase the gaps, this curves is going up to *x*-coordinate *(-18)* in Figure 8, *x*-coordinate *(-16)* in Figure 9, *x*-coordinate *(-15)* in Figure 10, *x*-coordinate *(-14)* in Figure 11. On each curve, there are six points that represent six sizes of inputs: 2^{10} (1K), 2^{17} (100K), 2^{18} (200K), 2^{20} (1M), 2^{21} (2M), and 2^{23} (8M) elements. We can observe points in the same position on curves to learn the running time of each precision for the same input; such as in Figure 7, last points of four curves show that for an 8M-element input, the increasing order in running time is arranged as follow: *float (*the fastest), *float2, double, and double2 (the slowest)*.

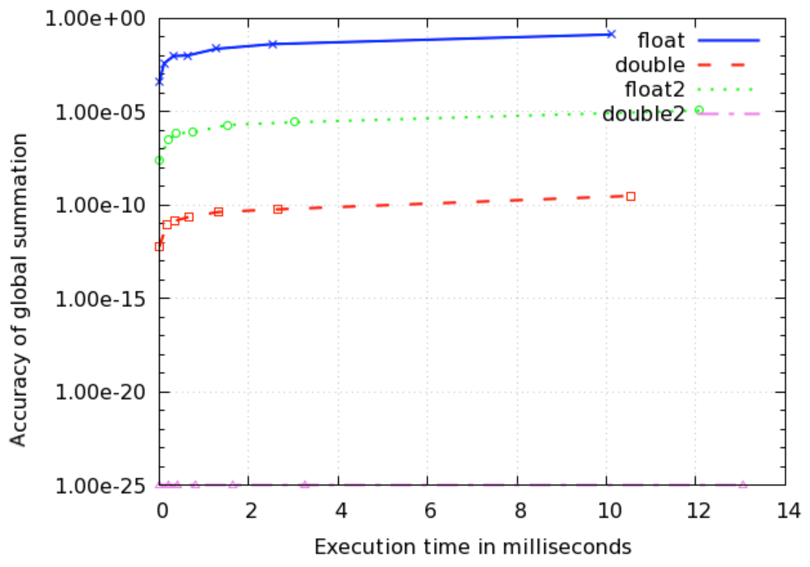


Figure 7: Accuracy vs. Execution time with intervals: (10^{-2}, 10^{-1}) & (10^{+1}, 10^{+2})

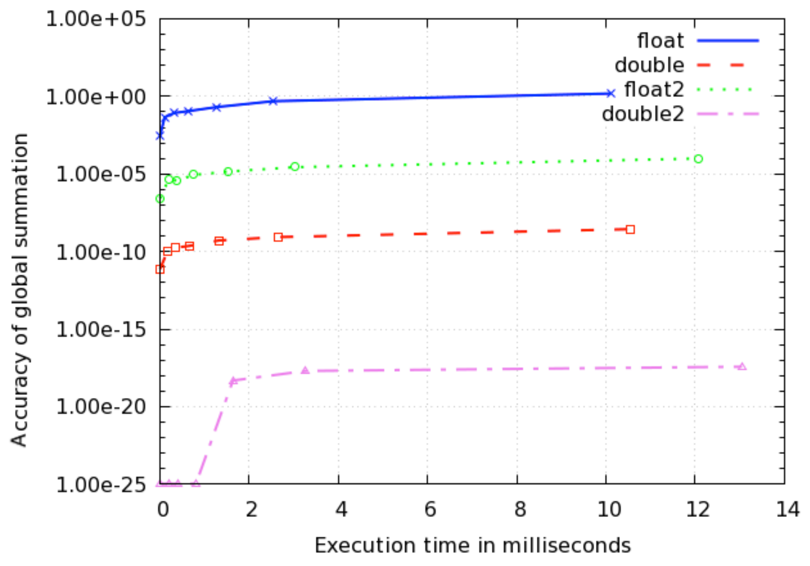


Figure 8: Accuracy vs. Execution time with intervals: (10^{-3}, 10^{-2}) & (10^{+2}, 10^{+3})

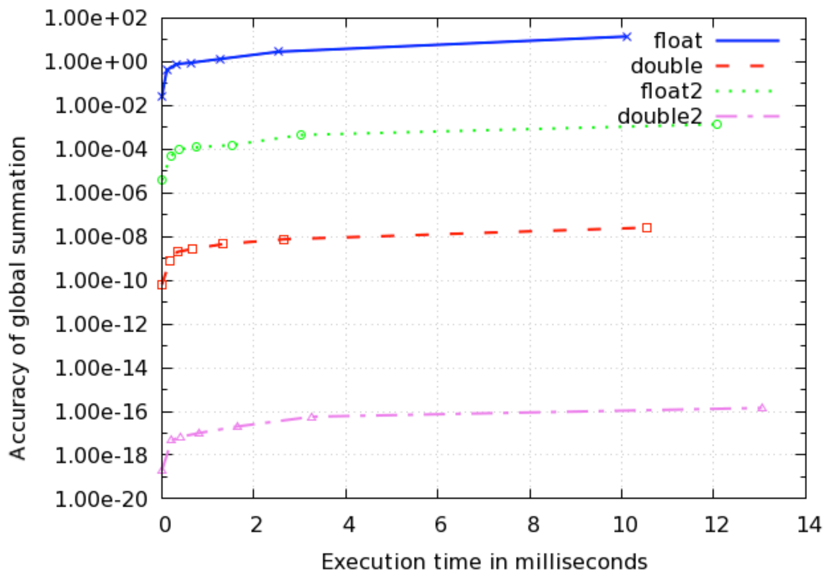


Figure 9: Accuracy vs. Execution time with intervals: (10^{-4}, 10^{-3}) & (10^{+3}, 10^{+4})

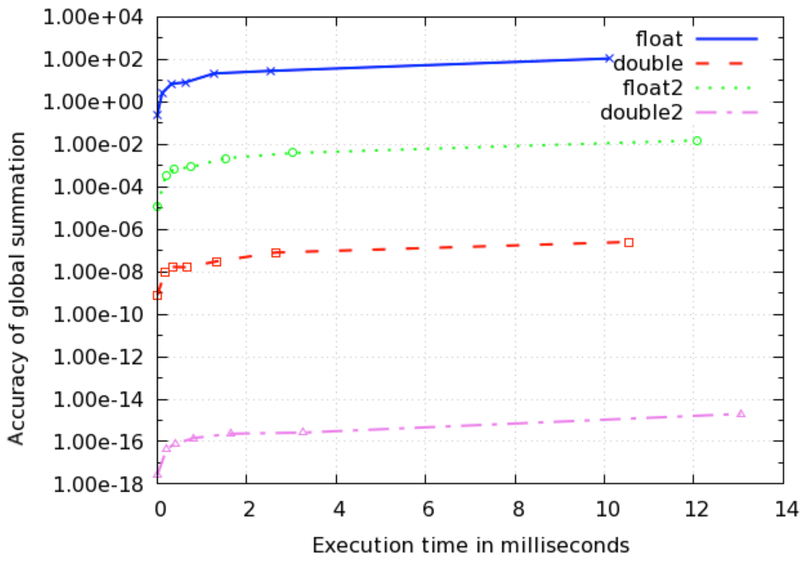


Figure 10: Accuracy vs. Execution time with intervals: (10^{-5}, 10^{-4}) & (10^{+4}, 10^{+5})

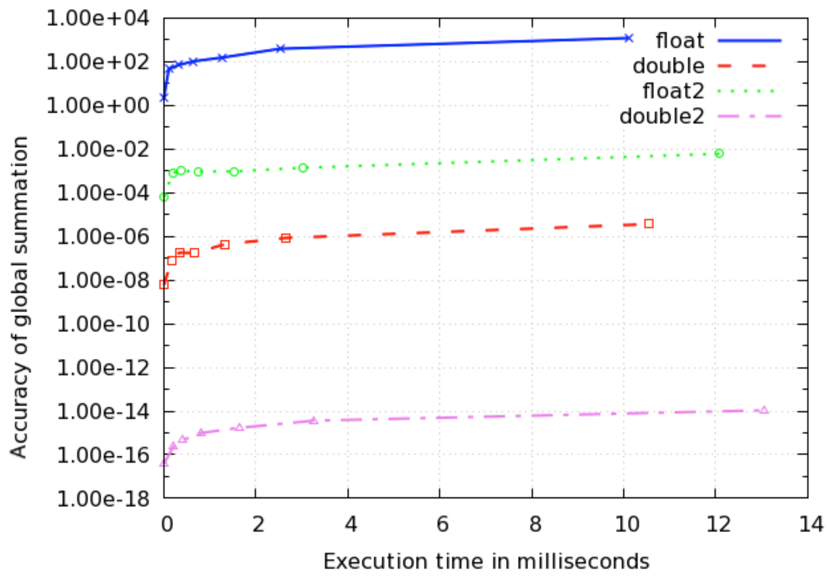


Figure 11: Accuracy vs. Execution time with intervals: (10^{-6}, 10^{-5}) & (10^{+5}, 10^{+6})

After the analysis in Chapter 5, we can conclude some important points:

- Higher precision arithmetic generally leads to higher accuracy (as expected).
- Composite precision arithmetic can improve the accuracy of programs containing only additions/subtractions arithmetic operations.
- Multithreading can help hiding arithmetic latency. In the global summation case, the best launch configuration is *block-size* = 32 (*warp-size*) and *grid-size* = 64.
- In global summation, it is never beneficial to use single composite precision arithmetic because it provides less accuracy than standard double precision, and its execution time is higher than double precision's execution time. Double composite precision arithmetic can be beneficial if the application requires high accuracy and can tolerate a slight increase of execution time.

- Compared to single precision, double precision arithmetic can lead to significant improvement in accuracy at a very low runtime overhead.
- Global summation is a *low compute-intensive* program. This leads to the following question: is multithreading beneficial for *high compute-intensive* applications? This question leads us to build the micro-benchmark discussed in Chapter 6.

Chapter 6: Micro-benchmark for analyzing the effect of arithmetic intensity on the performance/accuracy tradeoff on GPU

Arithmetic (or compute) intensity of a program is the ratio between the number of floating-point operations and the number of memory operations performed by the program. In this chapter, we propose a micro-benchmark to study the effect of arithmetic intensity on the performance/accuracy tradeoff on GPU. Specifically, we propose a workload generator that produces GPU kernels with a variable number of arithmetic operations and a fixed number of memory operations, leading to GPU kernels with variable arithmetic intensity. We then invoke the automatically generated kernels from a test program using different kernel launch configurations (leading to different degrees of multithreading), so as to study how the arithmetic intensity affects the performance/accuracy tradeoff at different degrees of multithreading.

In order to write a synthetic kernel generator, we should note the following points. *(i)* The *nvcc* compiler will automatically remove from kernels the instructions that do not have side effects. To avoid these simplifications, we need to insert global memory accesses both at the beginning and at the end of the generated kernel. *(ii)* We want to limit the global memory accesses to the ones strictly necessary and make sure that they are coalesced. *(iii)* We want to make sure that all variables have side effects and limit the data dependencies to keep functional units busy.

Our workload generator produces kernels that contain sequences of addition or multiplication operations and do not contain any branch operations. The number of arithmetic operations in each kernel is controlled through the command line *"Number of iterations"* parameter. We generate kernels that use different arithmetic precisions (*float*, *double*, *float2* and *double2*).

The pseudo code of the CUDA kernel generator based on addition operations and *float* arithmetic is below.

```
void sum_f(int iters){
      print to file (file name of float addition kernel);
      print to file (load GMEM to the first $x_0$ variable as a float);
      for i=1 to i=15
            print to file (declare $x_i$  as a float);
            print to file ($x_i$ = $x_{i-1}$ + 1.0);//these $x_i$ are different.
      while (count < iters){
            for i=0 to i=15
                  if (count <16)}
                        print to file (declare $y_i$ as a float);
                        print to file (initialize $y_i$ =0.0);
                  }
                  print to file(perform an addition of two float:
                              $y_i$ = $y_i$ + $x_i$);
                  count++;
      }
      print to file(store sum of $y_i$ to GMEM);
}
```

The pseudo-code for *double/float2/double2* addition and multiplication kernels is similar to that of the *float* addition kernel. The variable declarations and operations performed depend on the arithmetic operation used. To observations *(i), (ii), (iii)* above are reflected in the following aspects of the program.

- The number of y_i (and x_i) variables should be large enough to allow data independence between the addition/multiplication operations in the sequence. The use of a single y variable would lead to two subsequent update operations to be data dependent.
- The initial value of variable x_0 is read from global memory. The other variables are initialized as $x_i = x_{i-1} + 1.0$ for addition and $x_i = x_{i-1} * 2.0$ for multiplication. These initializations are performed so to prevent the *nvcc* compiler from simplifying away variables. While these initializations introduce data dependencies, they are done only once at the beginning of the kernel.
- The final result of variable y_i should be stored back to global memory, again to prevent the *nvcc* compiler from simplifying away operations on these variables.

The main program invokes addition and multiplication kernels using different arithmetic precisions and a given setting for the number of compute iterations. The program is structured as follows:

```
For the given number of compute iterations n:
      Call sum_f(n); //create float addition kernels
      Call sum_d(n); //create double addition kernels
      Call sum_f2(n); //create float2 addition kernels
      Call sum_d2(n); //create double2 addition kernels
      Call mul_f(n); //create float multiplication kernels
      Call mul_d(n); //create double multiplication kernels
      Call mul_f2(n); //create float2 multiplication kernels
      Call mul_d2(n); //create double2 multiplication kernels
```

After generating all kernels, we build a test program to launch various *compute-intensive* kernels, and collect execution times in two steps:

- Step 1: Use *block-size* = 1 and *grid-size* = 1 as kernel launch configuration (sequential execution); launch addition/multiplication kernels with a different number of compute iterations. The results are shown in Table 14.
- Step 2: Keep *grid-size* = 1 and increase *block-size*, so to progressively populate the floating-point functional units. The results are shown in Table 15 and Table 16.

In all cases, we show both the running time of the kernels and the increase/decrease in running time over using single precision *float* arithmetic. To explain the results in Table 14, Table 15 and Table 16, we look into the PTX assembly codes of the 1000-iteration kernels and analyze the instructions generated. First, for addition operations, the PTX code of the *float* kernel contains more than 900 lines of 32-bit additions; the PTX code of the *double* kernel contains more than 900 lines of 64-bit additions; the PTX code of the *float2* kernel contains more than 8,000 lines of 32-bit addition/subtraction; and the PTX code of the *double2* kernel contains more than 8,000 lines of 64-bit addition/subtraction. Second, for multiplication operations, the PTX code of the *float* kernel contains 16 lines of 32-bit additions, 847 lines of 32-bit multiplications, and 14 lines of 32-bit FMA (fused-multiply-add) operations; the PTX code of the *double* kernel contains 16 lines of 64-bit additions, 847 lines of 64-bit multiplications, and 14 lines of 64-bit FMA (fused-multiply-add) operations; the PTX code of the *float2* kernel contains 152 lines of 32-bit additions/subtractions, 1,968 lines of 32-bit multiplications, and 2,000 lines of 32-bit FMA operations; the PTX code of *double2* kernel contains contains 152 lines of 64-bit additions/subtractions, 1,968 lines of 64-bit multiplications, and 2000 lines of 64-bit FMA operations.

Table 13: Execution time (in milliseconds) of compute-intensive benchmark with a varying number of computational iterations. Kernel configuration: block-size=1, grid-size=1, device = Tesla C2070

# blocks	*# iters*	*Float*	*Double*	*Float2*	*Double2*	*(D-F)/F*	*(F2-F)/F*	*(D2-F)/F*
Addition								
1	1	0.0073	0.0073	0.0074	0.0073	-0.01	0.0027	-0.0053
1	20	0.0074	0.0075	0.0097	0.0104	0.0192	0.3132	0.4074
1	50	0.0075	0.0076	0.0115	0.0121	0.0084	0.5273	0.6091
1	100	0.0077	0.0078	0.0146	0.0168	0.0152	0.8906	1.1784
1	500	0.0098	0.0103	0.0427	0.0437	0.0502	3.3484	3.4498
1	1000	0.0124	0.0134	0.0761	0.0782	0.0776	5.1418	5.3045
1	5000	0.0446	0.045	0.3442	0.3499	0.01	6.7209	6.8496
1	10000	0.0815	0.0826	0.679	0.712	0.0135	7.3317	7.7358
Multiplication								
1	m1	0.0073	0.0071	0.0074	0.0072	-0.0249	0.0161	-0.0108
1	m20	0.0077	0.0074	0.0087	0.0085	-0.0413	0.131	0.1119
1	m50	0.0076	0.0074	0.0095	0.0104	-0.0161	0.2545	0.376
1	m100	0.0078	0.0077	0.0114	0.013	-0.0131	0.4576	0.6617
1	m500	0.0098	0.0107	0.0413	0.1272	0.0877	3.2098	11.9776
1	m1000	0.0122	0.0142	0.078	0.2477	0.1564	5.3722	19.2349
1	m5000	0.0445	0.0473	0.3707	1.4152	0.0634	7.3295	30.8009
1	m10000	0.0808	0.087	1.0078	2.409	0.0766	11.4734	28.8156

In general, the running time includes global memory access time and computation time. Let us first consider Table 14 (*grid-size=1* and *block-size=1*). At 1 iteration, the global memory access time dominates and the computation time is almost negligible. The results at 1 iteration reflect our observation in Chapter 5: for low arithmetic intensity, the *float, double, float2, and double2* kernels have almost the same performance. As we increase the number of iterations, the computation time becomes increasingly significant. The observed results for the addition kernels at 10,000 iterations show that the relative increase in running time over *float* is 7.3 for *float2* and 7.7 for *double2*; this is consistent with the ratio between *float2*/*double2* PTX code lines and *float*/*double* PTX code lines (ratio = 8000/900 ~ 8.89). For multiplication kernels, at 500 iterations, the relative increase in running time of *double2* over *float* is significantly higher than the relative increase observed in case of addition operations. This is due to limited hardware resources. Each SM in Fermi GPUs has 32 single-precision units (SPUs) and 16 double-precision units (DFUs). Moreover, the performance of double-precision FMA operations is half of the performance of single-precision FMA operations. Double precision composite kernels run a set of 64-bit multiplications and double-precision FMA operations simultaneously. Therefore, if the number of concurrent operations exceeds the available hardware resources, these operations are enqueued, causing an execution slow-down.

Figure 12 shows a graphical illustration of the percentage increase in execution time over *float*. As can be seen, the bars that represent the percentage change in execution time between the *double* and *float* kernels are nearly at zero level proving that the performance of *float* and *double* addition is almost the same; the bars that represent the percentage change in execution time between the *float2* and *float* kernel (or *double2* and *float* kernel) heighten with the increase in compute-intensity. The figure also shows that *float2* computation is as slow as *double2* computation. At 10,000 iterations, the global memory access time becomes negligible, so *float2/double2* computation is 8 times slower than *float/double.*

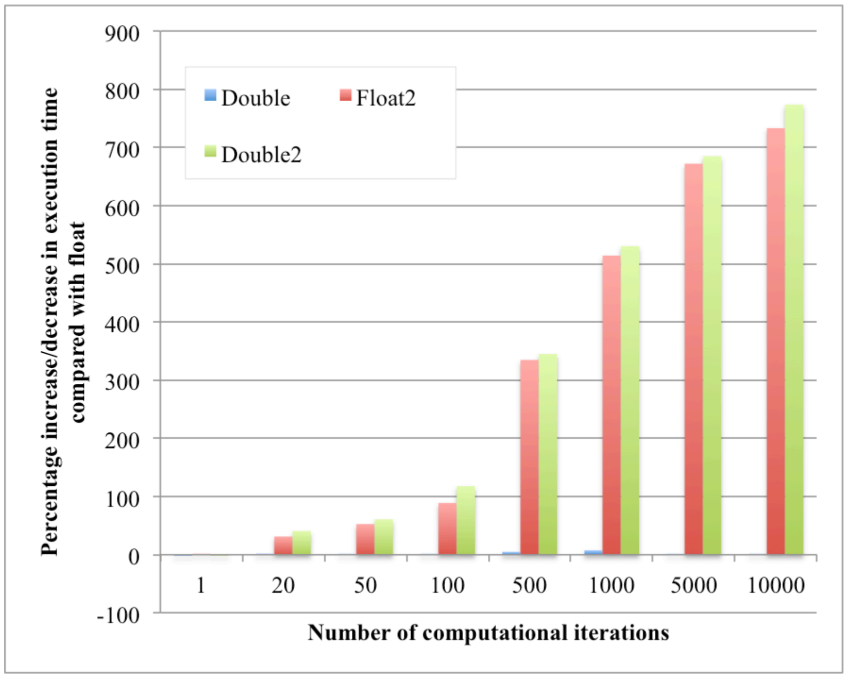


Figure 12: Percentage increase/decrease in running time for addition kernels of compute-intensive micro-benchmark with block-size = 1, grid-size = 1.

Table 14 and 15 show that, as we increase the *block-size*, the functional units become quickly populated. For example, in Table 15, the running time of the *double2* addition kernel increases significantly at 1,000 iterations since the DFUs are fully occupied, and it is twice the running time of the *float2* addition kernel. For multiplication kernels, the DFUs and FMA operations are filled faster, so the relative performance of the *double2* multiplication kernel worsens already at 500 iterations.

Figure 13 provides a graphical illustration of the percentage change of running time over the *float* kernel for *block-size=256* and *grid-size = 1*. In this figure, the gaps between the percentage change of the *float2* and *double2* kernels increase from left to right. At high arithmetic intensity, the performance of the *double2* addition kernel is worse than that of the *float2* addition kernel because of the limit in the number of DFUs.

Table 14: Execution time (in milliseconds) of compute-intensive benchmark with a varying number of computational iterations: grid-size=1, block-size=256, device = Tesla C2070.

# blocks	*# iters*	*Float*	*Double*	*Float2*	*Double2*	*(D-F)/F*	*(F2-F)/F*	*(D2-F)/F*
Addition								
256	1	0.0074	0.0075	0.0075	0.0076	0.0185	0.0158	0.029
256	20	0.0075	0.008	0.0104	0.0129	0.0716	0.3913	0.7253
256	50	0.0076	0.0084	0.0124	0.0179	0.1148	0.637	1.3611
256	100	0.008	0.0089	0.0158	0.0236	0.1138	0.975	1.9422
256	500	0.0105	0.0139	0.0438	0.0686	0.3254	3.1881	5.5626
256	1000	0.0138	0.02	0.078	0.125	0.4488	4.6431	8.0459
256	5000	0.0452	0.0693	0.352	0.5767	0.531	6.7799	11.7473
256	10000	0.0829	0.1308	0.6945	1.1644	0.5774	7.3785	13.0464
Multiplication								
256	m1	0.0073	0.0074	0.0076	0.0076	0.006	0.0379	0.0386
256	m20	0.0079	0.0075	0.0091	0.0108	-0.0504	0.1464	0.3647
256	m50	0.0076	0.008	0.0105	0.0173	0.0547	0.3844	1.2788
256	m100	0.008	0.0086	0.0129	0.0236	0.0676	0.6066	1.944
256	m500	0.0105	0.0133	0.0391	0.3013	0.2622	2.7118	27.6357
256	m1000	0.0138	0.0194	0.0714	0.6515	0.4023	4.1635	46.1339
256	m5000	0.0452	0.0687	0.3298	3.3636	0.5188	6.2924	73.3839
256	m10000	0.0825	0.13	0.6594	6.5767	0.5752	6.9915	78.7037

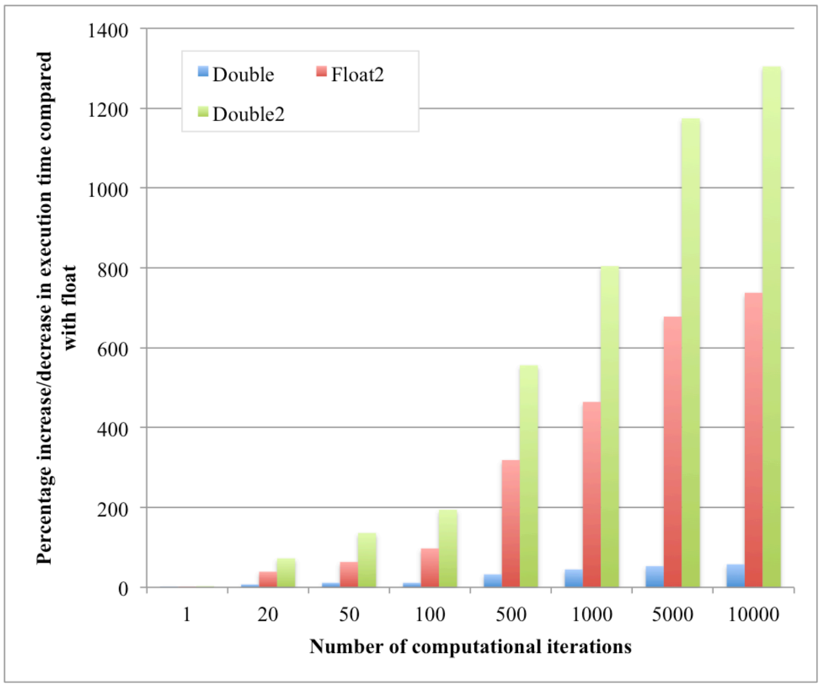


Figure 13: Percentage increase/decrease in running time for the addition kernel in the compute-intensive micro-benchmark with grid-size = 1, block-size = 256.

Table 15 shows that *float2* multiplication kernels start to occupy all SPUs at 5,000 iterations while *double2* multiplication and addition kernels fill the DFUs already at 500 iterations.

Table 15: Execution time of compute-intensive benchmark with different computational iterations. Kernel configuration: grid-size=1, block-size=512, device = Tesla C2070

# blocks	# iters	Float	Double	Float2	Double2	(D-F)/F	(F2-F)/F	(D2-F)/F
Addition								
512	1	0.0076	0.0076	0.0079	0.0082	-0.0006	0.0366	0.0816
512	20	0.0081	0.0086	0.0129	0.0185	0.0727	0.6055	1.2964
512	50	0.0084	0.0096	0.0163	0.0282	0.1457	0.9429	2.3631
512	100	0.009	0.0104	0.0222	0.0394	0.1514	1.4628	3.3738
512	500	0.0138	0.0203	0.067	0.1289	0.47	3.8419	8.3116
512	1000	0.0201	0.0328	0.123	0.2419	0.6314	5.1238	11.046
512	5000	0.0692	0.1307	0.57	1.1425	0.8893	7.2423	15.5214
512	10000	0.1303	0.2527	1.129	2.2929	0.9389	7.6633	16.5939
Multiplication								
512	m1	0.0074	0.0074	0.0077	0.008	0.002	0.0483	0.0807
512	m20	0.008	0.0081	0.0109	0.0143	0.0153	0.3651	0.7857
512	m50	0.008	0.0086	0.0134	0.0254	0.0725	0.6711	2.1687
512	m100	0.0087	0.0095	0.0176	0.0366	0.0993	1.0333	3.2256
512	m500	0.0136	0.0193	0.0607	0.3757	0.4238	3.4782	26.7058
512	m1000	0.0198	0.0318	0.1252	0.7888	0.6111	5.3338	38.9165
512	m5000	0.0688	0.1297	0.6772	3.994	0.8838	8.8376	57.0207
512	m10000	0.1299	0.2517	1.461	7.9606	0.9377	10.2485	60.2908

The data of Figure 14 (block-size= 512) are similar to those of Figure 13 since the functional units are already fully utilized with 512 threads.

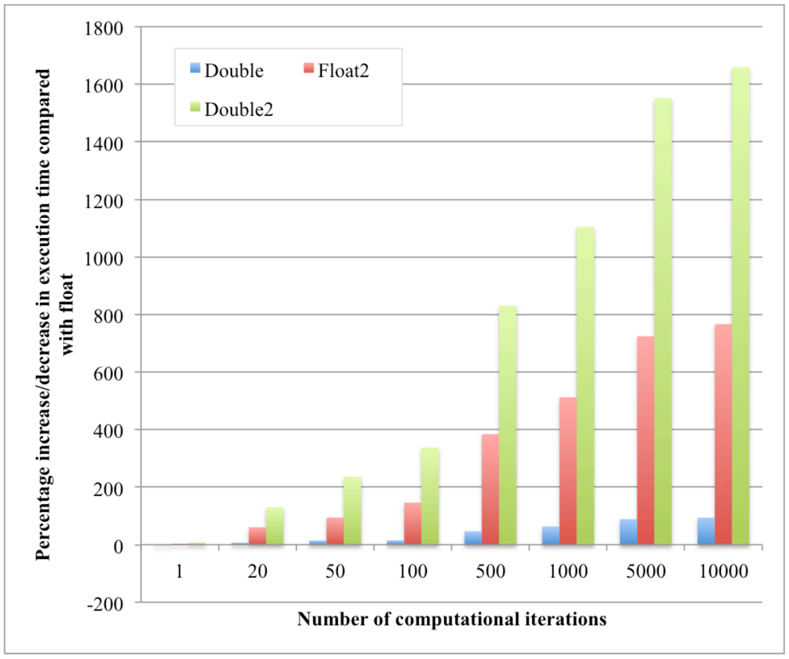


Figure 14: Percentage increase/decrease in running time for addition kernel of compute-intensive micro-benchmark with grid-size = 1, block-size = 512.

The main lessons learned from the study presented in this chapter are the following.

- We have confirmed some observations made in Chapter 5: on modern GPUs (compute capability 2 and above) and for kernels with low arithmetic intensity, *float* and *double* precision experience the same or similar performance; *float2* precision reports worse performance and accuracy than *double*, and should therefore not be used.
- Multi-threading can conceal the latency of the arithmetic operations up to the point where the functional units are fully utilized.
- The arithmetic intensity influences the runtime/accuracy tradeoff. For low arithmetic intensity, the use of higher precision (*double2*) can provide better accuracy without significantly sacrificing the execution time. However, as the arithmetic intensity grows and the available functional units become fully utilized,

lower precision (e.g. *double*) can become preferable to composite precision arithmetic (e.g. *double2*).

In the remainder of this thesis, we will focus on multiplication and division.

Chapter 7 Gaussian Elimination and LU Decomposition (LUD) Benchmarks

In this chapter, we analyze two benchmark applications that use floating-point division and multiplication: Gaussian Elimination and LU Decomposition. We first describe the Gaussian Elimination and the LU Decomposition algorithms. We then study the Gaussian Elimination and LU Decomposition benchmarks proposed in [2] and analyze how the use of floating-point multiplication/division with different arithmetic precisions affects the performance/accuracy tradeoff.

7.1 Introduction to Gaussian Elimination and LU Decomposition

7.1.1 Gaussian Elimination

Gaussian Elimination (GE) is a method of solving a system of linear equations *$Ax = b$*. This method first reduces the system into an upper triangular form, and then solves it by applying back substitution.

The algorithm of *Gaussian Elimination* is as below:

```
Given a coefficient matrix A size nxn
for j = 1 to (n - 1)
    for i = j + 1 to n      // n: size of coefficient matrix A
        $m_{ij} = a_{ij}/a_{jj}$        // $a_{jj} \neq 0$, find multiplier $m_{ij}$
        for k = j + 1 to n
            $a_{ik} = a_{ik} - m_{ij}a_{jk}$   //eliminating lower elements of A
        $b_i = b_i - m_{ij}b_j$          //updating vector b
```

This forward elimination algorithm requires $\frac{2}{3}n^3$ arithmetic operations [18].

After the upper triangular form of *A* is computed, the system is solved by the back-substitution algorithm below, which requires $O(n^2)$ arithmetic operations:

```
for i = n to 1
    $x_i = b_i$
    for j = (i + 1) to n
        $x_i = x_i - a_{ij}x_j$ //$a_{ij}$: element (i,j) of upper triangular matrix
    $x_i = x_i/a_{ii}$
```

7.1.2 LU Decomposition

LU Decomposition (LUD) is a factorization technique for transforming a square matrix *A* into a product of a lower triangular matrix *L* and an upper triangular matrix *U*. A lower triangular matrix is a matrix whose elements above the diagonal equal to zero; an upper triangular matrix is a matrix whose elements below the diagonal equal to zero.

LU Decomposition can be considered as a matrix form of *Gaussian Elimination* used to solve a system of linear algebraic equations $Ax=b$ where *A* is a *n*x*n* coefficient matrix, *b* is a given *n*-vector, and *x* is a unknown solution *n*-vector that needs to be computed. We first decompose *A* into *L* and *U,* so a system $Ax = b$ becomes $LUx = b$. Next, we solve a lower triangular system $Ly = b$ using forward substitution to obtain *y*. Finally we solve an upper triangular system $Ux = y$ using backward substitution to obtain solution *x*.

The upper triangular matrix *U* is the updated triangular form of *A* after the process of Gaussian Elimination. The lower triangular matrix *L* is given by:

$$L = \begin{bmatrix} 1 & 0 & \dots & 0 \\ m_{21} & \dots\dots & \dots & 0 \\ \dots & \dots\dots & \dots & \dots \\ m_{n1} & m_{n2} & \dots & 1 \end{bmatrix}$$

Where m_{ij} are the recorded multipliers mentioned in section 7.1.1.

We can store upper and lower triangular matrices *L* and *U* in the original matrix *A*. *Gaussian Elimination* algorithm in 7.1.1 is rewritten to implement *LU Decomposition* as below:

```
Given a coefficient matrix A size nxn
for j = 1 to (n - 1)
      for i = j + 1 to n      // n: size of coefficient matrix A
            $a_{ij} = a_{ij}/a_{jj}$        // updating lower triangular matrix L
      for i = j + 1 to n      // split loops for parallel computation
            for k = j + 1 to n
                  $a_{ik} = a_{ik} - a_{ij}a_{jk}$ //updating upper triangular matrix U
```

The complexity of factorizing the matrix *A* into *LU* is also $O(n^3)$. Once we have *LU*, we can solve $Ly=b$ in $O(n^2)$, then solve $Ux = y$ also in $O(n^2)$.

Considering the case that vector *b* of the system can be changed, we have *r* vectors *b*. Now we can take the advantage of *LU* decomposition because we do not need to recomputed *LU* matrix. Therefore, we can solve $(r-1)$ cases of vector *b* using $O(n^2)$ operations while if we do standard *GE* separately for each vector *b*, the total cost scales to $O(rn^3)$. This is the reason scientists refer to use *LU* decomposition instead of solving systems using *GE* independently.

7.2 Gaussian Elimination Benchmark

The *Gaussian Elimination (GE) Benchmark* proposed in [2] performs GE in parallel. In this implementation, the forward elimination phase includes two parallel kernels: a *Fan1* 1D kernel that performs division operations to find multiplier factors (matrix *M*), and a *Fan2* 2D kernel that performs the multiplications and subtractions required to transform coefficient matrix *A* into upper triangular matrix and update vector *b*.

Inputs of GE benchmark: The original version of the benchmark contains a tool to generate the GE inputs, including matrix *A*, vector *b*, and a pre-computed solution vector *x*, and save them into file for later use. However, these inputs are not suitable for analyzing the accuracy/performance using different floating-point precisions. We modified the benchmark so to generate a random matrix *A* and a random vector *b* with elements belonging to specified intervals. This allows us to evaluate the accuracy of GE when the input coefficients have different magnitudes.

Outputs of GE benchmark: We computed the solution vector *x* using the *float*, *double, float2, double2* and *GMP* precisions. We measured the accuracy of each solution in terms of the average absolute error between the reference 256-bit *GMP* solution *x* and the solution obtained using the other considered arithmetic precisions.

Evaluation: We run GE with seven input sizes and configuring the *Fan2* kernel with two block sizes. The elements of matrix *A* and vector *b* were randomly drawn from intervals $(10^{-2}, 10^{-1})$ & $(10^{+1}, 10^{+2})$. Figure 15 shows the accuracy results; the size of the matrix A (*n*) is increased along the *x*-axis. As can be seen, the *double* kernel is the most accurate, followed by the *double2*, the *float*, and the *float2* kernel. These results suggest that, in the presence of multiplication and division operations, composite precision arithmetic may not be advantageous even in terms of accuracy.

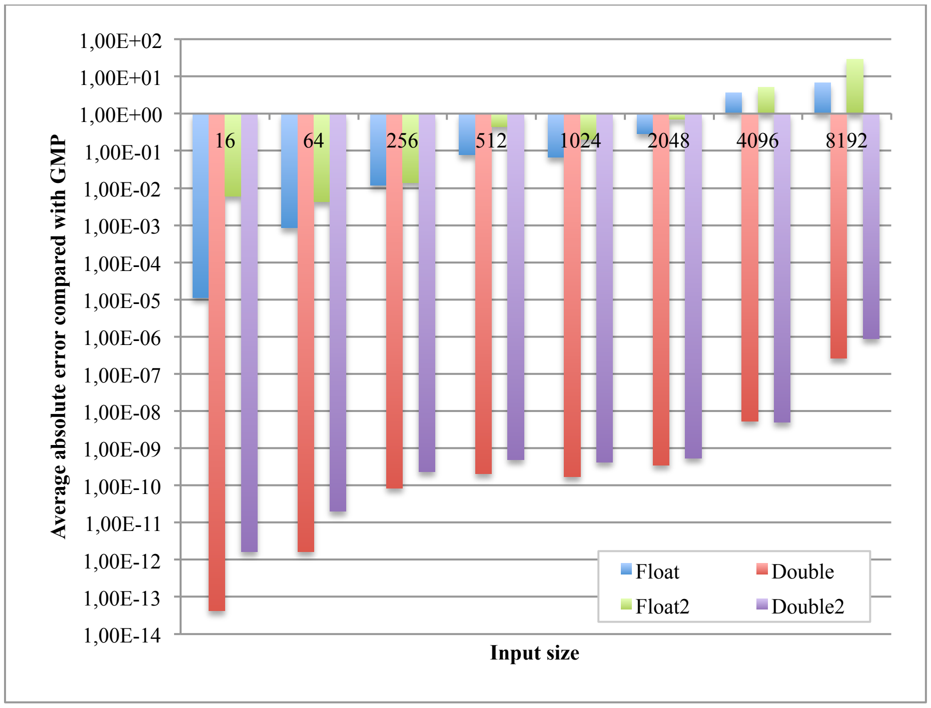


Figure 15: Average absolute error of GE benchmark.

Table 16 shows the performance results of GE reported by configuring the *Fan2* kernel with two block sizes: 4x4 and 8x8. When the block size is 4x4, the number of threads in one block is not high enough to form a warp, leading to low performance. In Figure 16, we show the speedup of the 8x8 block configuration over the 4x4 block configuration. As can be seen, the speedup is up to ~2.88x for *float* precision, ~2.2x for *double* and *float2* precision, and ~1.67x for *double2* precision.

Table 16: Running time (in seconds) of GE benchmark when the size of matrix A varies from 16x16 to 8,192x8,192.

Fan1 block-size = 512, Fan2 block-size= 4 X 4				
Size	*float*	*Double*	*float2*	*double2*
16x16	0.00040	0.00038	0.00038	0.00038
64x64	0.0017	0.0016	0.0017	0.0017
256x256	0.0119	0.0120	0.0124	0.0137
512x512	0.0628	0.0659	0.0682	0.0775
1024x1024	0.4312	0.4456	0.4634	0.5379
2048x2048	3.2387	3.4272	3.5761	4.2160
4096x4096	26.4912	28.6380	29.7799	36.1070
8192x8192	226.4626	250.3975	257.9343	290.4082
Fan1 block-size = 512, Fan2 block-size= 8 X 8				
Size	*float*	*double*	*float2*	*double2*
16x16	0.00036	0.00036	0.00034	0.00035
64x64	0.0017	0.0016	0.0016	0.0017
256x256	0.0083	0.0088	0.0089	0.0111
512x512	0.0284	0.0341	0.0395	0.0510
1024x1024	0.1607	0.2017	0.2032	0.3209
2048x2048	1.1214	1.5074	1.5323	2.6307
4096x4096	9.1784	13.6071	13.8189	24.5510
8192x8192	82.1154	131.7608	132.7769	192.8968

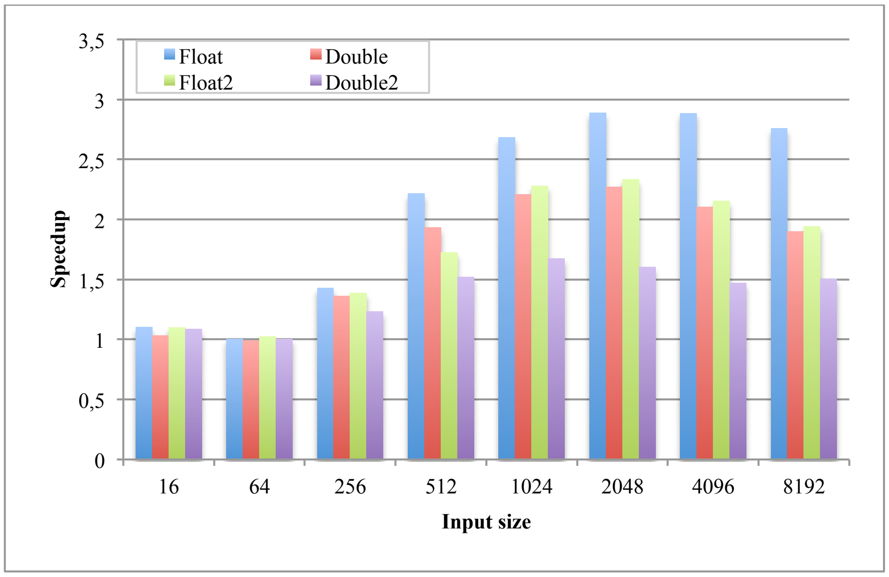


Figure 16: Speedup of GE benchmark as we increase the block-size of the Fan2 kernel from 4x4 to 8x8

In Figure 17 we correlate the performance and accuracy results presented above using an 8x8 block-size configuration for the Fan2 kernel. Note that the x- and y-axes are in logarithmic scale. On each curve, the data points from left to right are obtained by varying the size of matrix A from 16x16 to 8,192 x 8.192. As can be seen, the *float* and *float2* curves have similar trends and partially overlap: *float* and *float2* provide almost equivalent accuracy and performance (the execution time of *float2* is slightly higher than the execution time of *float*). The same observation can be made for *double* and *double2*. In summary, for this benchmark application composite precision is not helpful. This behavior may be due to the composite precision implementation of multiplication and division operations. To confirm this speculation, we continue our study on LU Decomposition, another benchmark including multiplication and division operations.

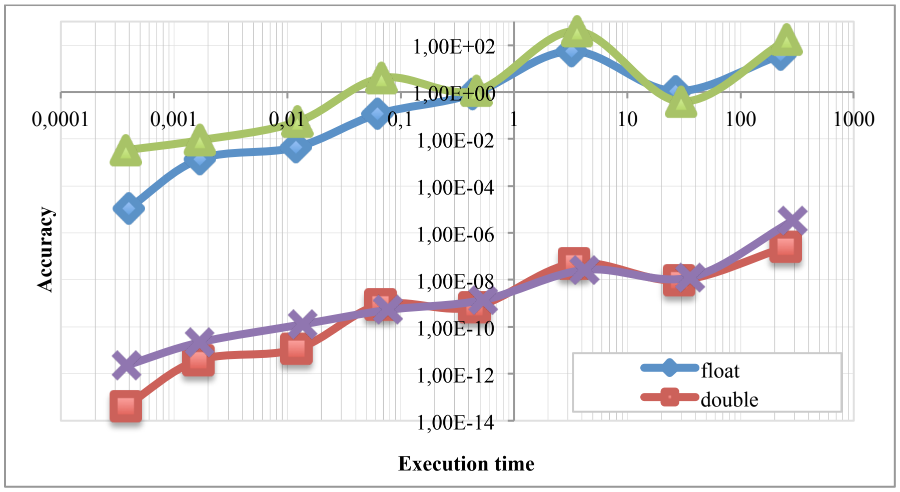


Figure 17: Accuracy vs. Execution time of GE benchmark

7.3 LU Decomposition Benchmark

The *LU Decomposition* program that is part of the Rodinia Benchmark Suite [2] performs the *LU* factorization of an input matrix *A* in both sequential and parallel fashion. In particular, the parallel version splits the computation into three kernels: a *lud_diagonal* kernel working on sub-matrices on the diagonal line, a *lud_perimeter* kernel working on sub-matrices along the edges (rows and columns), and a *lud_internal* kernel working on sub-matrices inside matrix *A* but not including diagonal elements.

<u>Inputs of LUD benchmark:</u> Although Rodinia Benchmark Suite includes some sample matrices, these matrices could not meet our requirements to analyze the accuracy of the *LUD* benchmark with different magnitudes of elements in the coefficient matrix *A*. Therefore, we generated a set of random matrices with elements in three intervals, as shown in Table 18. For each interval, we created matrices with 4 dimensions: *64x64, 256x256, 512x512, and 2048x2048*.

<u>Outputs of LUD benchmark:</u> We generate *LU* matrices using *float, double, float2,* and *double2* precisions on GPU, and *GMP* precision on CPU.

<u>Evaluation:</u>

We use two methods to evaluate the accuracy of the *LUD* benchmark. In the first method, we first split the solution matrix *LU* into matrices *L* and *U*, we then compute the product of *L* and *U*, and we finally compare this product with input matrix *A*. We report the number of mismatches between the elements of the product of *L*U* and the input matrix A using tolerance threshold = 0.0001. This number was always equal to zero when using *GMP* precision, confirming the high accuracy of this representation. Table 17 reports the number and percentage of mismatches when using *float*, *float2*, *double* and *double2* precision. As can be seen, *float2* precision provides the worst results and reports a huge number of miss-matches even in the case of small gaps between large and small input elements. In general, composite precision does not help accuracy even when based on double precision arithmetic.

*Table 17: Number and percentage of mismatches between original matrix A and the product of L*U for float/float2/double2 precisions*

Matrix Dim	*Float*		*Double*		*Float2*		*Double2*	
Interval: $(10^{-1}, 10^{+0})$ & $(10^{+0}, 10^{+1})$								
	# of miss	*% of miss*	*# of miss*	*% of miss*	*# of miss*	*% of miss*	*# of miss*	*% of miss*
64	204	4.9805	0	0	2571	62.7686	0	0
256	39419	60.1486	0	0	54433	83.0582	0	0
512	205141	78.2551	0	0	230510	87.9326	0	0
2048	4002671	95.4311	0	0	4097534	97.6928	0	0
Interval: $(10^{-2}, 10^{-1})$ & $(10^{+1}, 10^{+2})$								
64	3116	76.0742	0	0	3625	88.5010	0	0.0000
256	60669	92.5735	0	0	63227	96.4767	0	0.0000
512	254578	97.1138	0	0	258498	98.6092	85	0.0324
2048	4154682	99.0553	0	0	4176226	99.5690	120	0.0029
Interval: $(10^{-3}, 10^{-2})$ & $(10^{+2}, 10^{+3})$								
64	3534	86.2793	0	0	3824	93.3594	1	0.0244
256	63112	96.3013	0	0	64512	98.4375	381	0.5814
512	256954	98.0202	0	0	259804	99.1074	3615	1.3790
2048	4172933	99.4905	0	0	4185760	99.7963	72324	1.7243
Interval: $(10^{-4}, 10^{-3})$ & $(10^{+3}, 10^{+4})$								
64	3630	88.6230	0	0.0000	3855	94.1162	977	23.8525
256	63408	96.7529	0	0.0000	64625	98.6099	15075	23.0026
512	257943	98.3974	0	0.0000	259996	99.1806	87048	33.2062
2048	4175069	99.5414	3	0.0001	4186261	99.8082	1422031	33.9039
Interval: $(10^{-5}, 10^{-4})$ & $(10^{+4}, 10^{+5})$								
64	3467	84.6436	2098	51.2207	3751	91.5771	2689	65.6494
256	62519	95.3964	15961	24.3546	64212	97.9797	36051	55.0095
512	257566	98.2536	161445	61.5864	259222	98.8853	206318	78.7041
2048	4166277	99.3318	973696	23.2147	4175825	99.5594	2582436	61.5701

In our second method, we define accuracy in terms of average absolute error between the *GMP* results and the results obtained using other arithmetic precisions. Figure 18, 19, and 20 show the results reported using this method. As can be seen, the average absolute error increases significantly with the gap between small and large input elements, showing that

LUD is very sensitive to the difference in magnitude among the elements of the input matrix. These figures also confirm that *double2* precision is not better than *double* precision, and *float2* precision is not better than *float* precision.

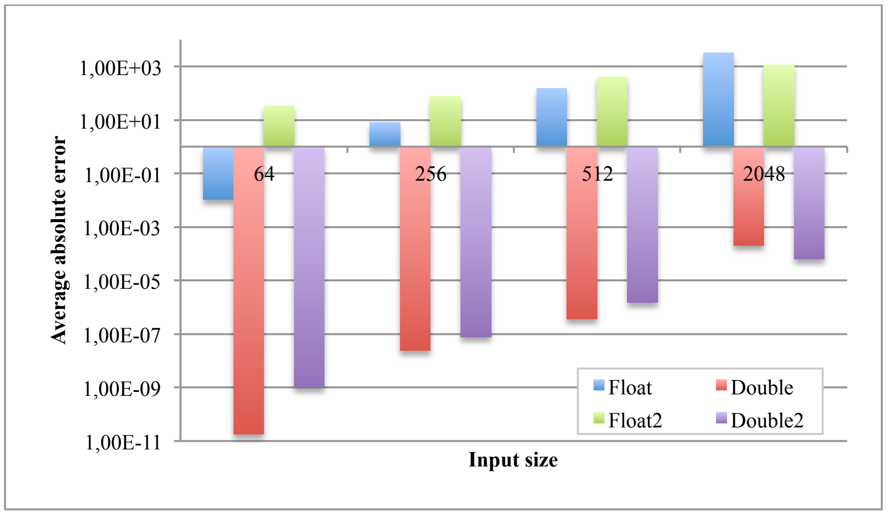


Figure 18: Average absolute error of LUD benchmark when the elements in the input matrix are randomly drawn from intervals (10^{-1}, 10^{+0}) & (10^{+0}, 10^{+1})

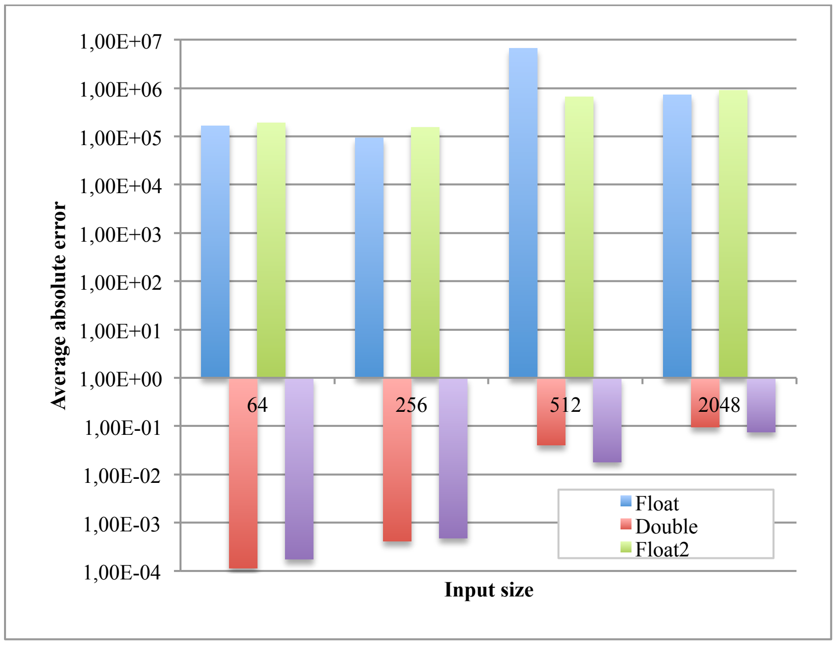


Figure 19: Average absolute error of LUD benchmark when the elements in the input matrix are randomly drawn from intervals (10^{-3}, 10^{-2}) & (10^{+2}, 10^{+3})

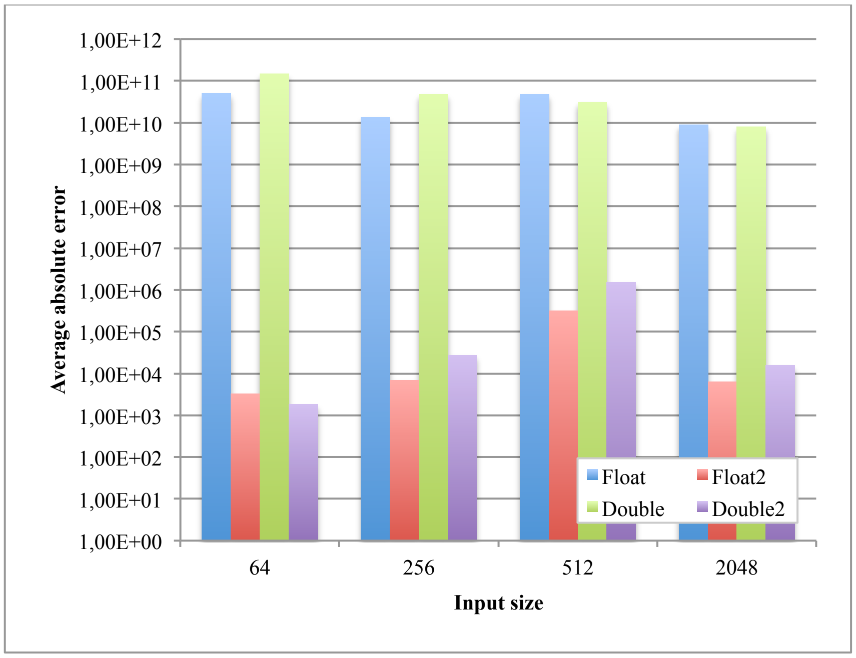


Figure 20: Average absolute error of LUD benchmark when the elements in the input matrix are randomly drawn from intervals (10^{-5}, 10^{-4}) & (10^{+4}, 10^{+5})

In Table 18 and 19 we show the running time (in milliseconds) reported by *LUD* on CPU and GPU, respectively. As can be seen, the GPU version reports a speedup up to ~300x for *float/double,* and up to ~1500x for *float2/double2* over the CPU version.

Table 18: LUD running time (milliseconds) on CPU

Matrix dim	*Float*	*Double*	*Float2*	*Double2*	*GMP*
64	0.98	0.59	6.73	5.04	19.34
256	37.76	44.87	348.54	291.96	1133.34
512	310.58	326.31	2757.17	2344.05	11266.33
2048	21410.89	42018.54	208913.45	304839.82	905661.23

From Table 19 we can also see that the best performance is achieved when the block size is equal to 16x16. Due to the shared memory requirements of the GPU kernels, we could not use block configurations beyond 32x32.

Table 19: LUD running time (milliseconds) in GPU

BLOCK-SIZE	*Matrix size*	*Float*	*Double*	*Float2*	*Double2*
4x4	64	0.32	0.37	0.40	0.48
	256	2.15	2.84	2.97	4.24
	512	11.07	14.57	15.71	21.68
	2048	619.62	720.36	801.51	1077.20
8x8	64	0.30	0.41	0.49	0.65
	256	1.37	2.04	2.43	4.09
	512	4.28	6.42	7.49	12.45
	2048	114.02	150.06	188.21	354.68
16x16	64	0.54	0.89	1.12	1.14
	256	2.38	4.33	5.32	6.34
	512	5.89	10.97	13.3	17.01
	2048	66.71	133.21	180.62	330.51
32x32	64	1.86	1.45	1.75	1.80
	256	10.40	7.71	9.23	9.78
	512	22.95	17.44	21.05	24.79
	2048	113.80	174.52	230.97	387.47
64x64	64	2.57	out of resources	out of resources	out of resources
	256	22.00	out of resources	out of resources	out of resources
	512	48.64	out of resources	out of resources	out of resources
	2048	295.02	out of resources	out of resources	out of resources

Figure 21 correlates the accuracy and performance time of LUD. Again, the x- and y-axes are in logarithmic scale. On each curve, the data points from left to right correspond to experiments performed varying the block size from 4x4 to 32x32. As in the GE case, we can observe that the use of composite precision arithmetic does not improve the accuracy. We believe that this may be due to the presence of multiplication and division operations in this application. This speculation will be confirmed in the following chapter.

We note that the accuracy can change with the block size because it is affected by the order of arithmetic operators. However, in this study, this change of accuracy is not important as when we change the magnitudes of elements in co-efficient matrices. Thus, we focus on observing only the accuracy with different magnitudes and sizes of inputs.

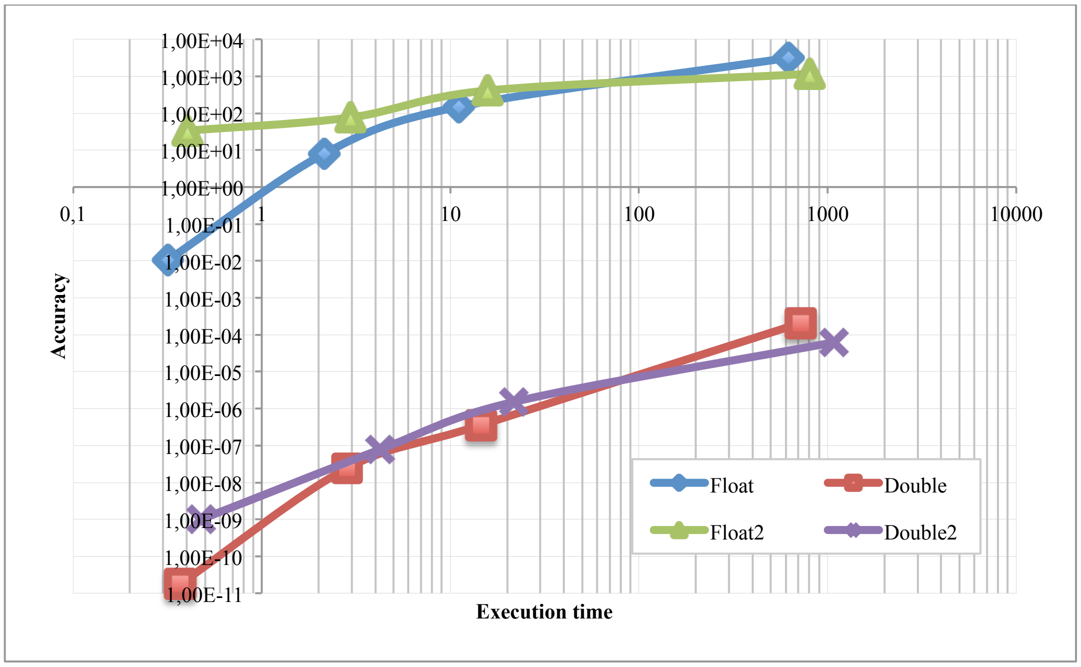


Figure 21: Accuracy vs. Execution time of LUD benchmark

Chapter 8 Micro-benchmarks for analyzing the behavior of composite precision multiplication and division.

In this chapter we use two micro-benchmarks to study the accuracy of the multiplication and division operations in the composite precision library: the first benchmark uses only multiplications, while the latter (do-undo) combines multiplications and divisions.

8.1 Multiplication micro-benchmark

The multiplication micro-benchmark is a simple program that implements a sequence of multiplications using *float, double, float2,* and *double2* precisions, and then computes the difference between these results and *GMP* reference results.

Inputs of this micro-benchmark are randomly generated arrays with elements of different magnitudes. Figure 22 and 23 report the results of the multiplication micro-benchmark using two sets of input intervals. Absolute errors in the figures represent the absolute difference between the results of *float/double/float2/double2* precision and the result of *GMP* precision after *n* multiplications. In both cases, *float2/double2* results are worse than *float/double* results, respectively. In order to explain this result, we consider the pseudo-code of composite precision multiplication. The *error* component is computed using the formula: *z2.error= x2.value * y2.error + x2.error * y2.value + x2.error * y2.error*. This formula is prone to error propagation. In particular, after a few multiplications the last term of the error component will tend to underflow.

To verify this speculation, we performed another set of experiments. Specifically, before executing each multiplication, we normalize the composite precision representation of the intermediate result. In other words, we convert the *float2/double2* result to *float/double* and then convert the value back to composite precision *float2/double2*. This allows intermediate results to maintain the error-free representation of Dekker's splitting method. We observed that the introduction of this intermediate conversion step leads to more accurate results. We conclude that the composite arithmetic library is not effective for multiplications.

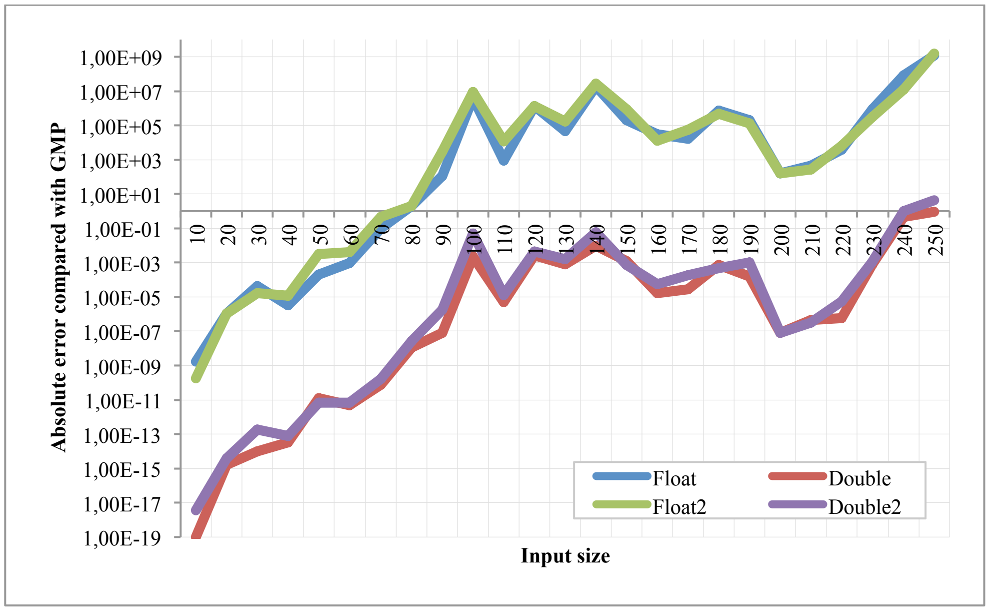


Figure 22: Absolute error of multiplication micro-benchmark when the input elements are drawn from intervals (0.1; 1.0)&(1.0; 6.0)

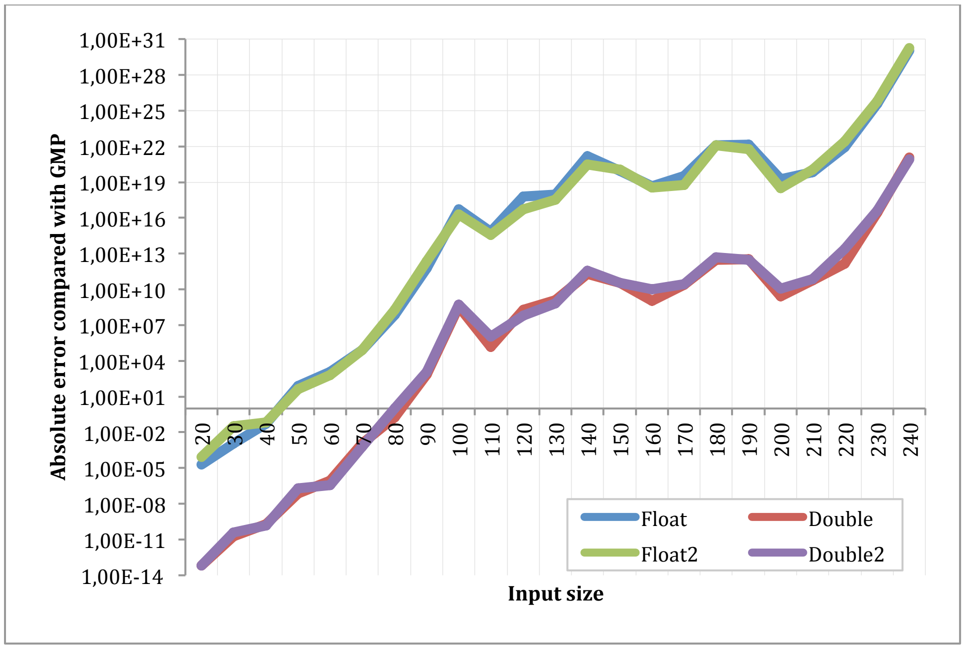


Figure 23: Absolute error of multiplication micro-benchmark when the input elements are drawn from intervals (0.1; 1.0)&(1.0; 10.0)

8.2 Do-undo micro-benchmark

8.2.1 Background related to divisions on GPU

Before discussing the do-undo micro-benchmark, we provide some background on the division operation on GPU.

8.2.1.1 Single precision division

On devices with compute capability 2.x and higher, the division operator '/' maps to IEEE round-to-nearest-even division when the code is compiled with *–pre-div=true* option (which is the *nvcc* default). If the option *--pre-div=false* is used (which also happens as a result of using compiler option *–use_fast_math*), the division operator '/' is mapped to approximate division.

On devices with compute capability 1.x the division operator '/' always maps to approximate division.

In both cases, the *__fdiv_rn(x,y)* intrinsic function allows IEEE round-to-nearest-even division, but it is a slow function.

CUDA Math API introduces intrinsic functions that can be only used in device code. Among these intrinsic functions are the less accurate, but faster versions of some standard functions. The use of the *--use_fast_math* compiler option forces some functions to compile to their intrinsic functions. For example, it causes the division operator 'x/y' to compile into the *__fdividef(x,y)* intrinsic – the fast approximate division function.

8.2.1.2 Double precision division

Double precision floating-point support has been added to CUDA starting from devices with compute capability 1.3. For all devices that support double precision, the division operator '/' always maps to IEEE round-to-nearest-even division.

8.2.1.3 PTX codes related to division

In order to better understand the use of the division on GPU, we analyzed the PTX assembly codes to crosscheck the division operator used. The following division instructions can be found in PTX assembly code:

- *div.approx.f32* is the fast approximate division with restricted range using reciprocal, and corresponds to the use of the __fdividef() intrinsic.
- *div.full.f32*: is a fast full range approximate division that scales values to achieve better accuracy, but is not fully IEEE 754 compliant.
- *div.rn.f32* is the IEEE compliant division

8.2.2 Do-undo benchmark

Given a random number *x* and a random array *y*, the do-undo benchmark that we developed based on do/undo program in [1] performs a sequence of operations based on the formula $(x*y_i)/y_i$ using *float* and *float2* on CPU and GPU. The accurate result of every iteration would be the value of *x*. As a consequence, when using this benchmark we can measure accuracy in terms of the absolute difference between *x* and the value $(x*y_i)/y_i$. If we use the same floating-point standard, the CPU and GPU implementations provide exactly the same results. In this thesis, we focus on the GPU implementation and study the impact of different kinds of divisions provided by CUDA on accuracy and performance.

The GPU version of the do-undo benchmark includes two user-defined options for single floating-point division. The first option uses the division operator '/' for *float* division; as explained above, this operator will be mapped to approximate division if the –*use_fast_math* compilation option is enabled. The second option uses the intrinsic function *__fdiv_rn()*for *float* division, leading to the use of the IEEE round-to-nearest-even standard division. The PTX assembly code generated in all cases is reported below.

Case 1: PTX code when using the first user-defined option and –use_fast_math option (float division is approximate division)

```
 BB23_1:
        mov.u64      %rd8, %rd20;
        mov.u64      %rd7, %rd19;
        mov.u64      %rd6, %rd18;
        mov.u64      %rd5, %rd17;
        add.s64      %rd17, %rd5, 4;
        ld.global.f32      %f16, [%rd5+4];
        ld.global.f32      %f17, [%rd7];
        mul.rn.ftz.f32     %f18, %f17, %f16;
        div.approx.ftz.f32      %f19, %f18, %f16;// user-defined option 1
        add.s64      %rd19, %rd7, 4;
        st.global.f32      [%rd7+4], %f19;
        add.s64      %rd18, %rd6, 8;
        ld.global.f32      %f20, [%rd6+8];
        ld.global.f32      %f21, [%rd8];
        mul.rn.ftz.f32     %f22, %f21, %f20;
        div.rn.ftz.f32     %f24, %f8, %f20; //my_lib.cu using __fdiv_rn()
        mul.rn.ftz.f32     %f25, %f22, %f24;
        mul.rn.ftz.f32     %f26, %f20, %f25;
        neg.ftz.f32        %f27, %f26;
        add.rn.ftz.f32     %f28, %f22, %f27;
        add.s64      %rd20, %rd8, 8;
        mul.rn.ftz.f32     %f29, %f24, %f28;
        add.rn.ftz.f32     %f30, %f25, %f29;
        st.global.v2.f32  [%rd8+8], {%f30, %f29};
        add.s32      %r5, %r5, 1;
        setp.lt.u32 %p2, %r5, %r3;
        @%p2 bra     BB23_1;
```

<u>Case 2: PTX code when using the second user-defined option and –use_fast_math option (float division is IEEE-compliant division).</u>

```
BB23_1:
          mov.u64          %rd8, %rd20;
          mov.u64          %rd7, %rd19;
          mov.u64          %rd6, %rd18;
          mov.u64          %rd5, %rd17;
```

```
        add.s64          %rd17, %rd5, 4;

        ld.global.f32    %f16, [%rd5+4];

        ld.global.f32    %f17, [%rd7];

        mul.rn.ftz.f32   %f18, %f17, %f16;

        div.rn.ftz.f32   %f19, %f18, %f16;// user-defined option 2

        add.s64          %rd19, %rd7, 4;

        st.global.f32    [%rd7+4], %f19;

        add.s64          %rd18, %rd6, 8;

        ld.global.f32    %f20, [%rd6+8];

        ld.global.f32    %f21, [%rd8];

        mul.rn.ftz.f32   %f22, %f21, %f20;

        div.rn.ftz.f32   %f24, %f8, %f20; //my_lib.cu using __fdiv_rn()

        mul.rn.ftz.f32   %f25, %f22, %f24;

        mul.rn.ftz.f32   %f26, %f20, %f25;

        neg.ftz.f32      %f27, %f26;

        add.rn.ftz.f32   %f28, %f22, %f27;

        add.s64          %rd20, %rd8, 8;

        mul.rn.ftz.f32   %f29, %f24, %f28;

        add.rn.ftz.f32   %f30, %f25, %f29;

        st.global.v2.f32        [%rd8+8], {%f30, %f29};

        add.s32          %r5, %r5, 1;

        setp.lt.u32      %p2, %r5, %r3;

        @%p2 bra         BB23_1;
```

Case 3: PTX code when using the first user-defined option and compiling without –use_fast_math option (float division maps to IEEE-compliant division)

```
BB23_1:

        mov.u64          %rd8, %rd20;

        mov.u64          %rd7, %rd19;
        mov.u64         %rd6, %rd18;
        mov.u64         %rd5, %rd17;
        add.s64         %rd17, %rd5, 4;
        ld.global.f32   %f16, [%rd5+4];
        ld.global.f32   %f17, [%rd7];
        mul.rn.f32      %f18, %f17, %f16;
        div.rn.f32        %f19, %f18, %f16; // '/' maps to IEEE-div by
default
        add.s64         %rd19, %rd7, 4;
        st.global.f32   [%rd7+4], %f19;
        add.s64         %rd18, %rd6, 8;
        ld.global.f32   %f20, [%rd6+8];
        ld.global.f32   %f21, [%rd8];
        mul.rn.f32      %f22, %f21, %f20;
        div.rn.f32      %f24, %f8, %f20;
        mul.rn.f32      %f25, %f22, %f24;
        mul.rn.f32      %f26, %f20, %f25;
        neg.f32         %f27, %f26;
        add.rn.f32      %f28, %f22, %f27;
        add.s64         %rd20, %rd8, 8;
        mul.rn.f32      %f29, %f24, %f28;
        add.rn.f32      %f30, %f25, %f29;
        st.global.v2.f32        [%rd8+8], {%f30, %f29};
        add.s32         %r5, %r5, 1;
        setp.lt.u32     %p2, %r5, %r3;
        @%p2 bra        BB23_1;
```

In these three cases, case 1 maps '/' to *__fdividef()* approximate division, so it is the fastest but least accurate option. This is suitable for applications that are not sensitive to the accuracy. Case 2 maps to the IEEE round-to-nearest-even function *__fdiv_rn()*; Case 3 is the default option that will map '/' into IEEE round-to-nearest-even on architecture 2.x and higher. Thus, case 2 and case 3 provide the same results. *__fdividef()* division needs 20 clock cycles while *__fdiv_* functions take 36 clock cycles. This may lead to the performance of case 1 better than case 2 and 3.

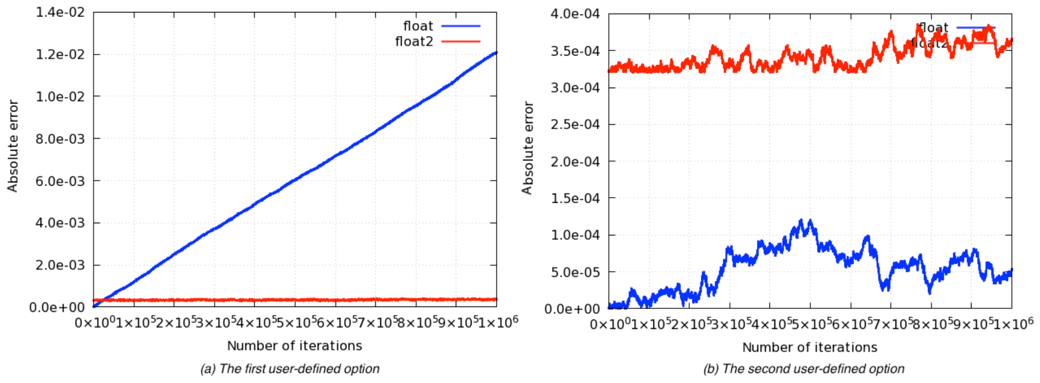


Figure 24: float vs. float2 absolute error of do-undo micro-benchmark with --use-fast-math option enabled

Figure 24 (a) shows that with 10^6 iterations and *x, y* in range *(0.0, 10.0),* the results using single floating point with approximate division is less accurate than using composite floating point.

However, Figure 24 (b) shows that with 10^6 iterations and *x, y* in range *(0.0, 10.0)*, the results using single floating point with IEEE-compliant division is more accurate than using composite floating point.

Next we will explore the accuracy of the do-undo micro-benchmark when varying the magnitude of the inputs and the size of the *y* array.

With 10^7 iterations, if both x and y are in interval *(0.0, 10.0)*, the absolute errors of the results are shown in Figure 25; if x and y are in interval *(10^{+5}, 10^{+6})*, absolute errors of the results are shown in Figure 26; if x and y have different magnitudes, absolute errors of the results are shown in Figure 27.

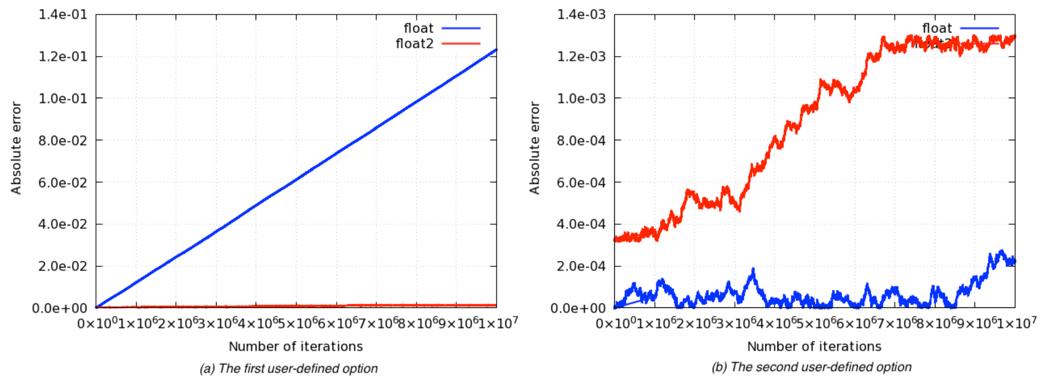


Figure 25: the absolute error of do-undo micro-benchmark with –use-fast-math option for 10M iterations with x small and y small

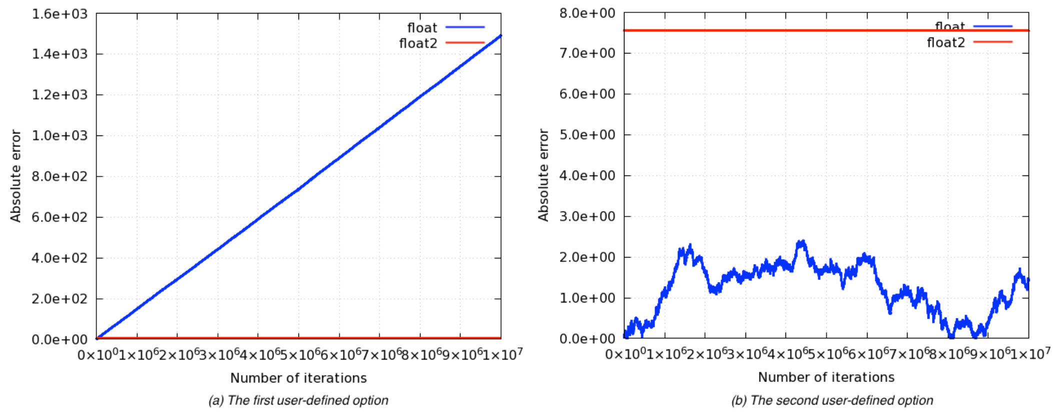
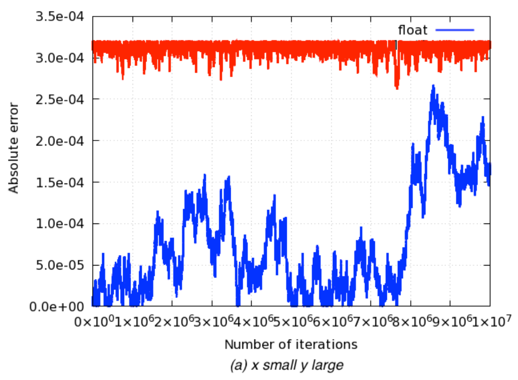


Figure 26: the absolute error of do-undo micro-benchmark with –use-fast-math option for 10M iterations with x large and y large



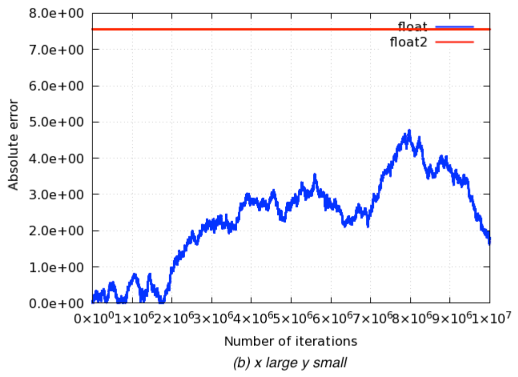


Figure 27: the absolute error of do-undo micro-benchmark with –use-fast-math option for 10M iterations with different magnitudes of x and y

In Figure 25(b), Figure 26(b) and Figure 27 (a)(b), division used for *float* is IEEE-compliant division. These four cases show that the error of *float* is lower than the error of *float2* numbers. We recognize that the absolute error of *float2* precision seems to stay around a horizontal line. Figure 28 below illustrates some more results.

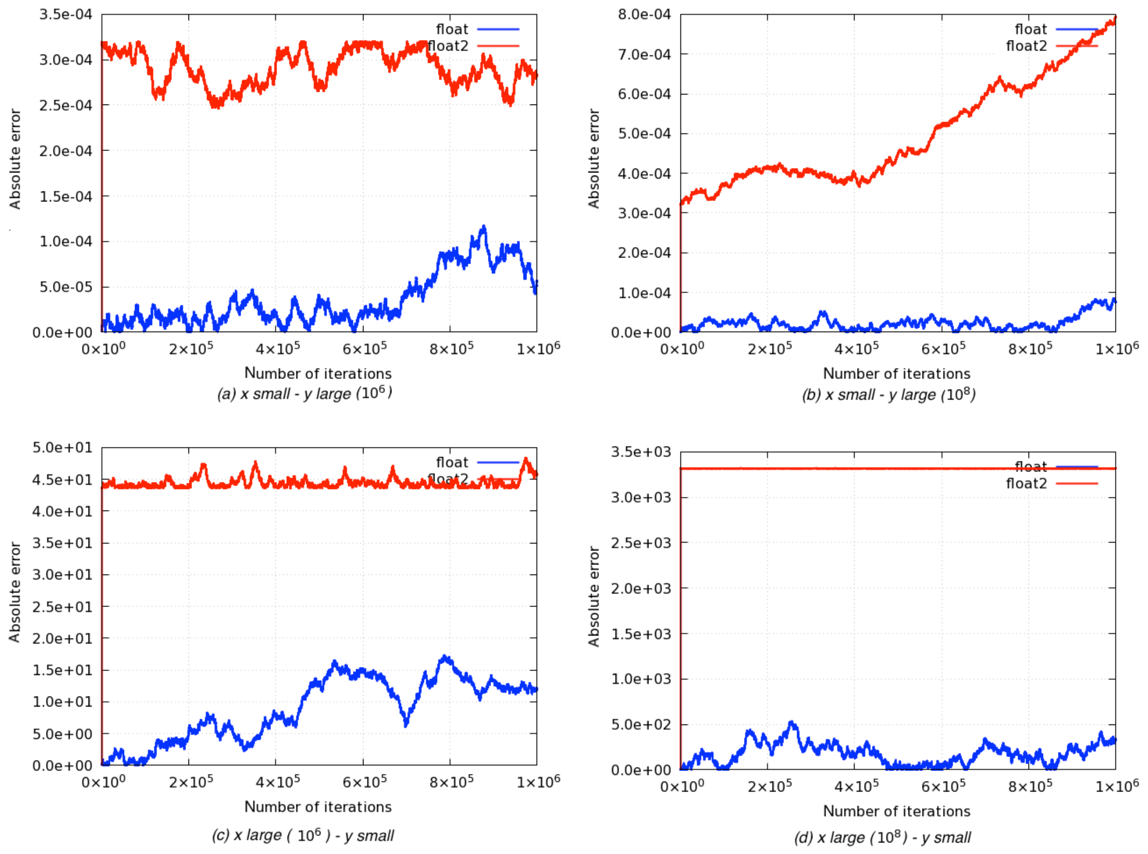


Figure 28: the absolute error of do-undo micro-benchmark with –use-fast-math option for 1M iterations with different magnitudes of x and y

Figure 28(a)(b) and 27(a) show the different trend of error rate when x small, y large. Figure 28(c)(d) and 27(b) show that when x is large and y is small, the accuracy of *do-undo* micro-benchmark gets worse.

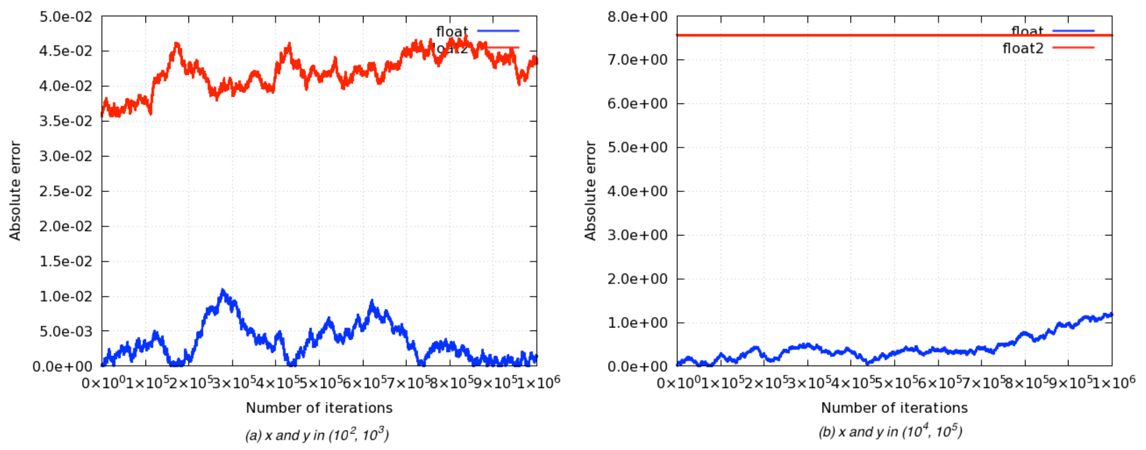


Figure 29: the absolute error of do-undo micro-benchmark with –use-fast-math option for 1M iterations with the same magnitude of x and y

Figure 29 show that when x and y are large, the accuracy of *do-undo* micro-benchmark also give large inaccurate results.

In the last part of this chapter, we try to explain why composite multiplication/division provides inaccurate result.

*Table 20: Gap between (x*y) and y in (x*y)/y the small value is in (0.0; 10.0); and the large value is bigger than 10^4.*

x value	*y value*	*Gap between (x*y) and y in (x*y)/y*
Small	Large	Small
Small	Small	Small
Large	Small	Large
Large	Large	Large

The inaccuracy of (x*y)/y consists of the errors of multiplication and the errors of division. First, for multiplication, if *x* is large and y is small, composite multiplication produces errors as explained in section 8.1. Second, for division, on one hand, the pseudo-code shows that error components of both *x* and *y* do not contribute the result ("x large and y large" case has largest lost of error component); on the other hand, the large gap between (*x*y*) and *y* produces a larger error component that would be neglected in the next step. Therefore, as *x* large, we get the worst absolute errors.

After exploring do-undo micro-benchmark we can confirm that:

- Composite division/multiplication is not better than standard floating-point division/multiplication as expected.
- Composite division/multiplication is very sensitive to the magnitude of the inputs.
- CUDA supports various types of division. Depending on users' desire, they can choose the less accurate, but fast division, or the high accurate, but slow division.

Chapter 9 Conclusion

Our analysis on the tradeoff between the performance and accuracy on GPU has brought some advantages for GPU programmers. By understanding the differences of floating-point precisions, and their impact on applications' performance, programmers can ensure the accuracy of their applications while maintaining acceptable performance. In general, the performance can be improved using multi-threading. From our study, we can advise users that *double* precision is the best choice in general if there is no specific requirement for accuracy and performance; if the program requires very high accuracy, they can use *CUMP* with paying high cost of performance; if the program contains only subtraction/addition, *double2* precision can be used to improve performance with acceptable performance.

Compute-intensive micro-benchmark provides CUDA developers statistical data on execution time of addition/subtraction or multiplication kernels corresponding to different level of compute-intensive. Accuracy of addition/subtraction kernels keeps the same, but for high compute intensive kernels, the performance of *float2/double2* becomes worse. Therefore, if the accuracy of *double* is acceptable, then *double* is the best choice for kernels using addition/subtraction operations with high compute-intensive level. Collected running time while studying this micro-benchmark also give CUDA developers a guidance to select optimized CUDA kernel configuration.

The observation of multiplication micro-benchmark shows that *float2/double2* multiplications are worse than *float/double* multiplications respectively. Combined with the performance of multiplication kernels provided by compute-intensive micro-benchmark, we can conclude that it is not worth to use *float2/double2* precisions for multiplication operations.

The study of different types of divisions on GPUs in do-undo micro-benchmark shows that users can modify their codes to make sure that it uses only IEEE-compliant operations to improve accuracy. Although the requirement of clock cycles for IEEE-compliant instructions is almost 2x times approximate instructions, this slow-down in performance can be easily hidden by multi-threading. The do-undo micro-benchmark confirms that *float2/double2* divisions are slower and less accurate than *float/double* divisions with IEEE-compliant.

Finally, through this thesis, we also learn extra knowledge related to some factors that affect the performance of applications on GPUs such as registers, shared memory, number of floating-point function units.

Reference

[1] M. Taufer, O. Padron, P. Saponaro, and S. Patel, “Improving numerical reproducibility and stability in large-scale numerical simulations on GPUs,” in IEEE International Symposium on Parallel Distributed Processing (IPDPS), April 2010, pp. 1–9.

[2] Rodinia: https://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php

[3] T.J. Dekker, A floating-point technique for extending the available precision, Numer. Math. 18, 224-242 (1971).

[4] A. Thall. Extended Precision Floating-point Number for GPU computation. ACM SIGGRAPH, 2006

[5] V.Y Pan, B. Murphy, G. Qian, R.E Rosholt, A new error-free floating-point summation algorithm.

[6] https://en.wikipedia.org/wiki/Scientific_notation

[7] The GNU Multiple Precision Arithmetic Library https://gmplib.org/

[8] C. Rubio-Gonzalez, C. Nguyen, H. D. Nguyen, J. Demmel, ´W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, “Precimonious: Tuning Assistant for Floating-point Preci-sion,” in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC ’13), 2013.

[9] P. Langlois, M. Martel, and L. Thevenoux, “Accuracy Versus ´ Time: A Case Study with Summation Algorithms,” in Pro-ceedings of the 4th International Workshop on Parallel and Symbolic Computation (PASCO ’10), 2010, pp. 121–130.

[10] http://www.hpcs.cs.tsukuba.ac.jp/~nakayama/cump/

[11] http://www.nvidia.com/docs/IO/43395/
NV_DS_Tesla_C2050_C2070_jul10_lores.pdf

[12] http://www.nvidia.com/content/tesla/pdf/nvidia-tesla-kepler-family-datasheet.pdf

[13] http://www.nvidia.com/content/pdf/fermi_white_papers/
nvidia_fermi_compute_architecture_whitepaper.pdf

[14] https://www.nvidia.com/content/PDF/kepler/ NVIDIA-Kepler-GK110-
Architecture-Whitepaper.pdf

[15] https://en.wikipedia.org/wiki/Floating_point

[16] http://developer.download.nvidia.com/assets/cuda/files/NVIDIA-CUDA-
Floating-Point.pdf

[17] http://www.aes.tu-berlin.de/fileadmin/fg196/publication/
poster_andresch_acaces2014.pdf

[18] http://www.math.ryerson.ca/~danziger/professor/MTH108/Handouts/gauss-
complexity.pdf