PERFORMANCE/ACCURACY TRADE-OFFS OF FLOATING-POINT ARITHMETIC ON NVIDIA GPUS: FROM A CHARACTERIZATION TO AN AUTO-TUNER

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

SRUTHIKESH SURINENI

Dr. Michela Becchi, Thesis Supervisor

DECEMBER 2017

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

PERFORMANCE/ACCURACY TRADE-OFFS OF

FLOATING-POINT ARITHMETIC ON NVIDIA GPUS:

FROM A CHARACTERIZATION TO AN AUTO-TUNER

presented by Sruthikesh Surineni,

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 WILLIAM HARRISON

_____

Professor XIAOQIN ZOU

ACKNOWLEDGEMENTS

First, my most humble thanks go to my advisor, Michela Becchi. This work would not have been possible without her invaluable expertise, guidance, and generous patience. Dr. Becchi encouraged and supported me throughout my time at Mizzou. I graduate as a better computer engineer and researcher because of her supervision.

I would like to thank Professor William Harrison and Professor Xiaoqin Zou for serving as my master's thesis defense committee. I would like to thank my lab-mates Ruidong Gu and Huyen Nguyen for their work and time on this research. I would like to thank all my lab-mates at NPS lab-NCSU for providing me their invaluable feedback on my research. I would like to thank my colleagues Satyaki Koneru, Ke Yin, and Val Cook at ThinCI for sharing your expertise and opportunity you have provided.

My time at Mizzou has been very enjoyable because of my friends at Mizzou. I would like to thank all of you for giving me these great memories.

Finally, a special thank you for my family, as my accomplishments throughout my career can be directly attributed to their support and encouragement.

Thank you!

Sruthikesh Surineni

University of Missouri-Columbia

December 2017

CONTENTS

Page

LIST OF ILLUSTRATIONS

Figure **Page**

LIST OF TABLES

Table **Page**

PERFORMANCE/ACCURACY TRADE-OFFS OF
FLOATING-POINT ARITHMETIC ON NVIDIA GPUS:
FROM A CHARACTERIZATION TO AN AUTO-TUNER

Sruthikesh Surineni

Dr. Michela Becchi, Thesis Supervisor

ABSTRACT

Floating-point computations produce approximate results, possibly leading to inaccuracy and reproducibility problems. Existing work addresses two issues: first, the design of high precision floating-point representations, and second, the study of methods to support a trade-off between accuracy and performance of central processing unit (CPU) applications. However, a comprehensive study of trade-offs between accuracy and performance on modern graphic processing units (GPUs) is missing. This thesis covers the use of different floating-point precisions (i.e., single and double floating-point precision) in the IEEE 754 standard, the GNU Multiple Precision Arithmetic Library (GMP), and composite floating-point precision on a GPU using a variety of synthetic and real-world benchmark applications. First, we analyze the support for a single and double precision floating-point arithmetic on the considered GPU architectures, and we characterize the latencies of all floating-point instructions on GPU. Second, a study is presented on the performance/accuracy tradeoffs related to the use of different arithmetic precisions on addition, multiplication, division, and natural exponential function. Third, an analysis is given on the combined use of different arithmetic operations on three benchmark

applications characterized by different instruction mixes and arithmetic intensities. As a result of this analysis, a novel auto tuner was designed in order to select the arithmetic precision of a GPU program leading to a better performance and accuracy tradeoff depending on the arithmetic operations and math functions used in the program and the degree of multithreading of the code.

Chapter 1: INTRODUCTION

Floating-point operations are at the core of many scientific applications, including experimental mathematics and physics [1], astrophysics [2], climate modeling [3], quantum chemistry and molecular dynamics [4] simulations. Because floating-point numbers are an approximation of real numbers and floating-point addition and multiplication are not associative [5], applications relying on floating-point arithmetic suffer from inaccuracies and result in reproducibility issues. One way to mitigate this problem has been the use of high-precision arithmetic operations [6, 7]. However, this comes at a significant performance cost. To avoid this problem, several efforts have aimed at providing reproducible and efficient implementations of commonly used mathematical functions, such as global summation [8-11] and linear algebra [12].

1.1 An Overview

Over the last decade, many-core graphics processing units (GPUs) have been widely used to accelerate a wide variety of applications. In its online catalog [13], NVidia lists > 200 GPU-accelerated applications from different domains, including computational chemistry, biology, physics, numerical analytics, machine learning, weather prediction, computational finance and data mining. Despite the body of work on reproducibility problems of floating-point arithmetic on CPU [6-12, 14-22], the study of accuracy-performance tradeoffs related to the use of floating-point arithmetic on GPUs has received very limited consideration. Several efforts have proposed multiple-precision floating-point libraries for GPU [23-25].

However, these libraries provide high accuracy at a significant runtime cost. Therefore, if used unnecessarily, they can lead to highly inefficient codes.

Taufer et al. [4] proposed using composite precision arithmetic, which encodes a real variable into two single-precision floating-point numbers, as a compromise between single- and double-precision arithmetic both in terms of accuracy and performance. Their study was done on NVidia pre-Fermi GPUs and was motivated by the observation that, on these GPUs, the cost of double-precision arithmetic was 10 times that of single-precision arithmetic. In more recent GPU architectures, however, the performance of single and double precision arithmetic has greatly increased and the difference in performance between the two has significantly decreased. For example, the peak single-precision floating-point performance, measured in giga floating point operations per second (GFLOPS), increased from approximately 933 GFLOPS on Tesla C1060 (pre-Fermi) GPUs to approximately 1,030 GFLOPs on Tesla C207x (Fermi) [26] GPUs, to approximately 4,000 GFLOPS on K20/K40 (Kepler) GPUs to approximately 4,981 GFLOPS on GTX980 (Maxwell) and 8,873 GFLOPS on GTX1080 (Pascal) GPUs [27]. At the same time, the peak double precision performance increased from approximately 77.6 GFLOPs on C1060s to approximately 515 GFLOPs on C207x to approximately 1,400 GFLOPS on K20/K40 GPUs. But on recent GPU architectures the peak double precision performance decreased to approximately 206 GFLOPS on GTX980 (Maxwell) and 380 GFLOPS on GTX1080 (Pascal) GPUs [28]. Therefore, Taufer's proposal needs to be revisited in light of these recent GPU developments.

Even though scientific applications are continually requiring better accuracy and reproducibility, many modern applications are tolerant to minor accuracy variations. For example, machine learning or deep neural networks require a vast amount of computing power for training the models. To reduce the time required to train the models, performance architects train the model at a lower precision. Working at lower precision improves the performance because of higher low precision throughputs on modern GPUs. To support these kinds of applications Nvidia modern GPU architectures support half-precision floating-point operations with double throughput (18.7 TFLOPS) compared to single-precision floating-point operations. However, reducing the precision of a program randomly to achieve better performance may result in an incorrect outcome from the model. To achieve the same results at higher precision, a programmer should have expert knowledge about the algorithm and the hardware architecture. An existing solution [14, 17-19, 29] for this problem is to automate this process of searching for lower precision computing on CPU. GPU receives little to no work in addressing this problem. To address this problem with a GPU, we analyze the mixed-precision throughputs and introduces a novel auto-tuner for searching lower precision on highly multi-threaded GPU architectures.

1.2 Contributions

Our first goal was to characterize the use of floating-point arithmetic precisions on modern GPUs and provide new insights into the involved performance and accuracy trade-offs. Keeping in mind these insights provided from the study, we designed a novel auto-tuner for GPU programs. For our analysis we considered various floating-point precisions: single and double precision in the IEEE 754 standard, the GNU Multiple Precision Arithmetic

Library (GMP), and composite precision based on either single or on double precision numbers (float2 and double2, respectively) [4]. We focus on four generations of NVidia GPUs: Fermi, Kepler, Maxwell, and Pascal devices. Our study is structured as follows:

1) We analyze the support for single and double precision floating-point arithmetic on the considered GPU architectures. This include characterization of the latencies of all floating-point instructions on GPU.
2) We study the performance and accuracy tradeoffs related to the use of the floating-point arithmetic precisions listed above on addition, multiplication, division, and natural exponential function. To this end, except for division, we used micro benchmark applications that performed only one of these operations. In addition, we extended the composite-precision library to support the natural exponential function, and analyzed two implementations of this operation.
3) We analyze the combined use of different arithmetic operations in three benchmark applications characterized by different instruction mixes and arithmetic intensities, namely: Gaussian Elimination (GE), LU Decomposition (LUD) and LavaMD (LMD). The latter is a molecular dynamic simulation application.

Our study led to the following findings. First, the use of float2 arithmetic is not beneficial on Fermi and later GPUs, since it underperforms a double-precision arithmetic both in terms of accuracy and execution time. Thus, a float2 arithmetic should be considered only in combination with double-precision arithmetic for applications that would otherwise

exhaust all the double-precision execution units available on GPU. Second, the use of double2 arithmetic for addition operations is a good compromise between double- and multiple-precision arithmetic implementation in terms of both accuracy and performance. However, double2 is generally as accurate as double-precision arithmetic on multiplications and exponential functions, but not as much on divisions. Therefore, double2 should be used only when the accuracy provided by standard double-precision arithmetic is not sufficient for the application, and on variables involved prevalently in addition and subtraction problems. In the presence of a set of addition problems followed by a division problem, it can be beneficial to perform a double2-to-double conversion before the division. Third, on a GPU, the performance disadvantage of multiple-precision arithmetic units is aggravated by the fact that the high register utilization of its operation limits the amount of multithreading that can be exploited to perform latency hiding. Fourth, for the composite-precision natural exponential function, an implementation based on the built-in single- and double-precision `exp`() functions is preferable to one based on the Taylor series. as proposed by Thall [30].

Along with this study we designed and architected a novel auto tuner to improve hardware utilization. GPU programs with mixed-precision operations occupy all the available functional units which improves hardware utilization and on modern GPUs, lower precision arithmetic has higher throughput. The main goal when building this auto tuner was to provide an automatic search technique to reduce the precision of a program for better hardware utilization. Applications like machine learning are tolerant to slight variations in accuracy and can benefit from this auto tuner by reducing the precision. Lower

precision computing is important because most modern GPUs like Maxwell and Pascal are targeted towards AI. The throughput of lower precision single- or half-precision is significantly higher.

From the floating-point addition micro benchmark study, it can be concluded that mixed precision computing improves hardware utilization and performance. The novel auto tuner resulting from our research does not need any annotations or hints from the source code for tuning the program.

1.3 Thesis outline

The remainder of the work is organized into seven chapters. Chapter 2 includes the background and motivation necessary for the floating-point performance and accuracy tradeoff analysis on GPUs. Chapter 3 describes the characterization of floating-point instruction latencies on Fermi, Kepler, Maxwell, and Pascal GPU architectures. Chapter 4 provides a systematic performance/accuracy trade-offs analysis of floating point arithmetic addition, multiplication, division and natural exponent function. In chapter 5, we extend our performance/accuracy tradeoff analysis to real applications chosen from the Rodinia test suite, which consists of a mix of the floating-point arithmetic. Chapter 6 describes the necessity of an auto-tuner on modern GPUs to improve hardware utilization by analyzing a micro-benchmark. Chapter 7 describes our novel dataflow auto-tuner architecture and its implementation details. Finally, chapter 8 concludes this thesis and discuss future goals.

Chapter 2: FLOATING-POINT BACKGROUND AND MOTIVATION FOR THIS ANALYSIS

This section consists of the floating-point information necessary to adequately understand the remainder of this thesis. Section 2.1 describes IEEE 754-2008 floating-point representation and its error on modern machines. Section 2.2 briefly describes some of the arbitrary-precision arithmetic libraries and their disadvantages. Section 2.3 describes the Composite precision representation and arithmetic. Finally, Section 2.4 discusses studies related to floating-point accuracy on CPU and techniques to improve the accuracy.

2.1 IEEE 754 floating-point Standard

2.1.1 Representation

Floating-point numbers are approximations of real numbers and floating-point makes it possible to represent very large and very small values in a single format. Gaps in floating-point representation on a real scale are shown in Figure 1. These gaps in the floating representation increase with the exponent which lead to a higher approximation. The IEEE 754 standard [31] governs the floating-point arithmetic and defines its formats, numeric conversions, rounding rules, operations and exception handling. Per IEEE 754 standard, a floating-point number is encoded by three components – a sign, an exponent and a mantissa – according to the following formula shown in (1).

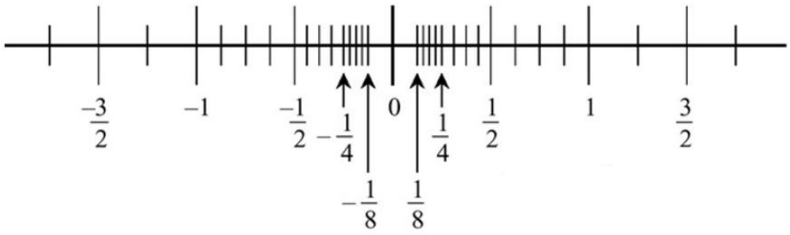

Figure 1. Gaps in floating-point representation shown on a real scale

While the exponent can be both negative and positive, it is always encoded as an unsigned integer. Thus, when encoding a value into floating-point format, the IEEE standard adds a positive bias (127 for single and 1023 for double precision) to the exponent. Table I reports the bit-size of sign, exponent and mantissa for single and double precision arithmetic, along with the numerical range covered. The floating-point notation has a specific representation for the following special numbers: zero (exponent and mantissa equal to 0), infinity (exponent=255 and mantissa=0), and NaN (not-a-number: exponent=255 and non-zero mantissa).

$$(-1)^{sign} \times 1.mintissa \times 2^{exponent-BIAS} \quad (1)$$

Table I. IEEE 754 floating-point storage formats

Format	**# of bits**			**Representation range**
	Sign	**Exponent**	**Mantissa**	
Single prec.	1	8	23	~[-3.4*10^{38}, 3.4*10^{38}]
Double prec.	1	11	52	~[-1.7*10^{308}, 1.7*10^{308}]

2.1.2 Rounding Error

The IEEE 754-2008 standard also documents many rules that describe how each arithmetic instruction must be performed to ensure a deterministic output across several machine architectures. The accuracy of instructions can be bounded using ULP (Units in Last Place). IEEE 754 defines several rounding methods including rounding to nearest, round to zero, round upward and round downward. Round to nearest is the default in most of the modern compilers.

Floating point arithmetic like addition, multiplication is not associative because the order of the rounding results in different output value. Shown below in the equation (2), RHS and LHS may not produce the same result. This is due to changing the order of rounding the arithmetic.

$$a * (b * c) \neq (a * b) * c \tag{2}$$

2.2 Arbitrary-Precision Arithmetic

Arbitrary-precision libraries [32-34] provide high accuracy by performing arithmetic operations on numbers with arbitrary size, virtually limited only by the amount of memory available on the host computer. In this work we use the GNU MPFR library [32] on CPU and the CUDA Multiple Precision Arithmetic (CUMP) [24] library on GPU, both based on the GNU Multiple Precision library (GMP) [35]. GMP is a free open-source library for arbitrary-precision arithmetic, and it operates on integers, rational, and floating-point numbers. In principle, arbitrary-precision arithmetic libraries should allocate additional

space dynamically whenever the accurate representation of a variable requires it. However, the current version of the GMP library (version 6.0) supports the automatic expansion of the precision only for integer and rational numbers, but requires the precision of floating-point numbers to be chosen statically and does not change the size of these variables after initialization. Yet, since GMP allocates memory space (in units called limbs) to represent numbers, GMP operations are substantially slower than standard floating-point operations. In our analysis, we use 256-bit GMP floating-point precision and consider this GMP output as reference compared to all other representations.

2.3 Composite-Precision Arithmetic

Composite-precision arithmetic represents real numbers through a fixed number of floating-point variables (typically two of them). Composite-precision arithmetic is a compromise between standard and arbitrary-precision arithmetic: it allows better accuracy than the IEEE 754 standard while providing substantially better performance than the arbitrary-precision. The foundation of composite-precision arithmetic is in the error-free method proposed by Dekker [36] to split a floating-point number into two half-length floating-point values. This method is illustrated in the pseudo-code in Figure 2 (to the left).

Dekker's splitting method has been used in several studies, including Thall's work on the use of extended-precision floating-point arithmetic on GPU [30]. More recently, Taufer et al. [4] have redefined Thall's extended-precision arithmetic and introduced the composite-precision (CMP) floating-point library used in this paper. CMP decomposes a floating-point number (float2) into two single-precision floating-point variables: a value and an error component, as shown in Figure 2 (to the right). Similarly, the use of a double-

precision floating-point value and error results in a double2 variable. The CMP library then defines the addition, subtraction, multiplication, and division operations on float2 (and double2) as combinations of multiple single (and double) precision instructions, as shown in Figure 3. As can be seen in Figure 3 and Table II, CMP addition/subtraction require eight floating-point additions/subtractions; CMP multiplication requires four floating-point multiplications and two additions; while CMP division needs a floating-point reciprocal, four multiplications, one addition, and one subtraction. In this thesis, we extend the CMP library to support the exponent and logarithm operations, and we present an analysis of the performance/accuracy tradeoffs involved with the use of standard, composite and arbitrary-precision arithmetic on GPU.

```
[hx, tx] split (float x){
  c=float(2^t+1);
  p = float (x * c);
  hx = float(p – (p-x));
  tx = float(x-hx);
}
t=12 for single-precision, t=24 for double precision.
```

```
struct float2{
   float value;
   float error;
}x2;
```

Figure 2. Dekker's error-free split of floating-point numbers into two component (left); Taufer's CMP float2 data structure (right).

```
//Addition
float2 x, y, z;
float t ;
z.value = x.value + y.value ;
```

```
t       = z.value – x.value ;
z.error = x.value – (z.value - t) +
          (y.value - t)+ x.error + y.error;
```

```
//Multiplication
float2 x, y, z ;
z.value = x.value * y.value ;
z.error = x.value * y.error +
          x.error * y.value +
          x.error * y.error ;
```

```
//Division
float2 x, y, z;
float t, s, diff;
t = (1 / y.value);
s = t * x.value;
diff = x.value – (s * y.value);
z.value = s + t * diff;
z.error = t * diff;
```

Figure 3. Taufer's CMP addition, multiplication and division.

Table II. Number of floating-point arithmetic required for composite precision arithmetic

Op	Add/Sub	Mul	Reciprocal
CMP Add	8	-	-
CMP Mul	2	4	-
CMP Div	2	4	1

2.4 Discussion on Related Work for CPU

In recent years there have been several efforts aimed to provide verification [15], compiler techniques [16, 17] and tuning assistants [14, 18] to verify or improve the accuracy of CPU programs containing floating-point arithmetic. Gappa [15] is a proof assistant based on interval arithmetic aimed to facilitate the proof of numerical properties of floating-point programs (for example, ensuring that variables and errors stay contained within specified ranges). The Rosa compiler [16] allows developers to write programs using a generic "real" datatype, and, whenever possible, it generates implementations that achieve a target accuracy using the least possible precision. Rosa requires the programmer to specify desired error bounds, input data ranges, postconditions and sources of uncertainty. Herbie [17] proposes using a set of code rewrite rules (for example, replacing arithmetic expressions with polynomial approximations) to improve the accuracy of a program for specific input intervals.

Rather than focusing on accuracy alone, tuning assistants for CPU code consider the tradeoff between accuracy and performance. Precimonius [14] explores the search space corresponding to the assignment of different precisions to the floating-point variables within a program, with the goal of setting the precision of the variables so as not to violate given accuracy and performance constraints. Precimonius requires the programmer to specify the ranges of the inputs. FPTuner [18] proposes a method to tune the performance of a program while maintaining rigorous error bounds by selectively reducing the precision of groups of variables, and it analyzes the search space using an SMT-solver.

Our work focuses on GPU and has three goals: first, improving the programmer's understanding of the floating-point support offered by modern GPUs and the performance/accuracy tradeoffs related to the use of different floating-point precisions on these devices; second, providing insights and criteria that can help the design of tuning assistants for GPU codes; third we design a novel GPU program auto-tuner which improves the hardware utilization and performance. A characterization of the behavior of different floating-point precisions on different arithmetic operations and an analysis of how the arithmetic precision can affect the degree of multithreading (and, thus, the performance) can help narrowing down the space that must be explored by tuning assistants for GPU code. In addition, accuracy/performance considerations that apply to distinct arithmetic operations but are independent of the values/ranges of the program inputs can help designing auto-tuning techniques that require less programmer intervention.

Chapter 3: CHARACTERIZATION OF NVidia GPU INSTRUCTION LATENCY

In this chapter, we discuss the support for floating-point arithmetic on NVidia GPUs (with a focus on Fermi, Kepler, Maxwell and Pascal architectures), and we use a suite of micro-benchmark applications to characterize the latency of floating-point operations on these devices. Section 3.1 describes briefly about the Nvidia GPU architecture and programmability. In Section 3.2, we characterize and analyze the latency of floating-point operations on modern Nvidia GPU architecture using extended cudabmk test suite.

3.1 Support of Floating-Point Arithmetic on GPU

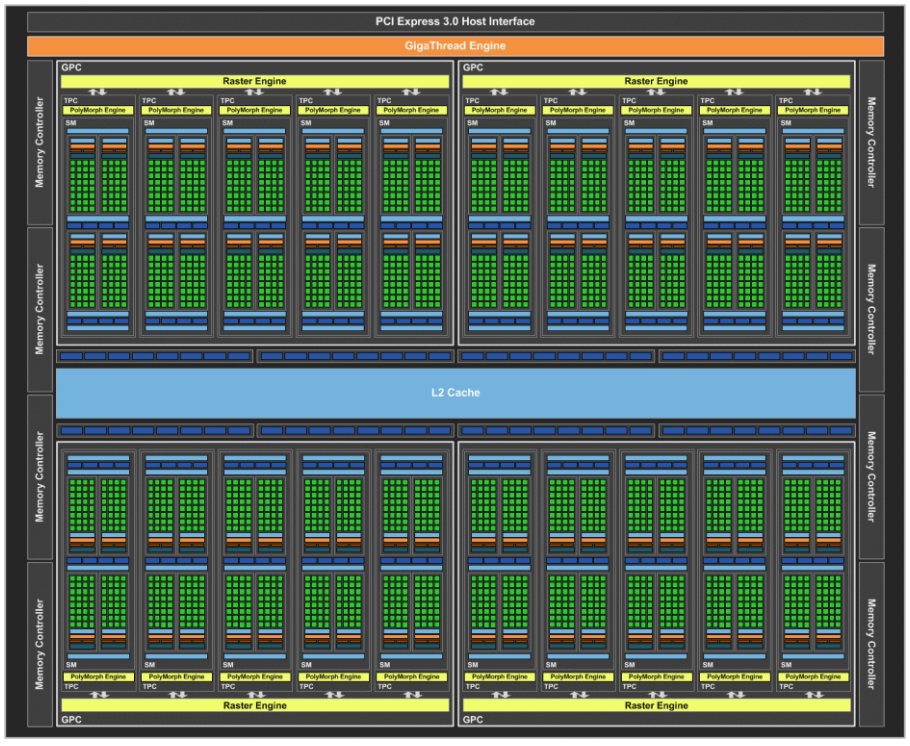

Figure 4. NVidia Pascal GP104 architecture

Figure 5. NVidia Pascal streaming multiprocessor (SM) architecture

NVidia GPUs as consist of multiple parallel processors called streaming multiprocessors (SMs) as shown in Figure 4, each executing groups of threads (called thread-blocks) in a grid. Block of threads are given to each SM and these threads share some resources on SM for communication and synchronization. The SM decodes the instructions and schedules warps of 32 threads executing in a SIMT lockstep fashion with a program pointer per warp. Besides including a shared register file, a shared local memory and an instruction/thread scheduler, every SM as illustrated in Figure 5 comprises multiple single (SP) and double precision (DP) functional units, which are responsible for the execution of single and double precision instructions, respectively. In addition, SMs have multiple special functional units (SFU) responsible for the execution of single precision special math operations such as square root, logarithm, exponent, and trigonometric functions. Table

III reports the hardware configurations of different GPU architectures used in our experiments. We use NVidia CUDA as our programming language to program the GPU kernels.

Table III. Hardware configuration of NVidia GPUs

GPU	Per SM				
	# SM	**# SP cores**	**# DP cores**	**# SFU**	**#Registers**
Tesla C2070	14	32	16	4	32768
Kepler K40C	15	192	64	32	65536
Maxwell TitanX	24	128	4	32	65536
Pascal Titan Xp	30	64	32	16	65536

3.2 Characterization of arithmetic instruction latencies

In order to characterize the floating-point instruction latencies on GPUs, we used and extended the benchmark suite proposed by Wong et al. [37] for the analysis of Fermi GPUs. This suite includes a set of single-threaded programs that perform a sequence of (the same) floating point instructions. These programs contain a data dependency between every two back-to-back instructions to avoid multiple instructions to be issued to different functional units at the same time (NVidia GPUs have multiple instruction schedulers/dispatchers per SM, allowing for some degree of ILP). The single-threaded program is illustrated in the pseudo code below. Instruction latencies are then measured through the `clock()` function, which provides the current value of a thread-level counter that is incremented every clock. To ensure consistency of the results, we repeated each experiment twice: once with a sequence of 128 instructions and once with one of 256 instructions.

Pseudo code of the original suite where the `clock()` positions are reordered resulting in wrong latencies:

```
start_time = clock();

repeat {

fop a a b

fop b a b

}

stop_time = clock();
```

We modified and extended this benchmark suite in two ways. First, we observed that, on more recent GPU generations (e.g., Kepler, Maxwell, and Pascal), the compiler could rearrange `clock()` function instructions, possibly leading to incorrect measurements of the instruction latencies. To avoid this rearrangement, we introduced a barrier synchronization (`__syncthreads()`) around `clock()` function calls as shown in pseudo code below. This barrier synchronization ensures the instructions are not rearranged across these barrier boundaries. Second, we added support for the double-precision special functions (reciprocal, square root, inverse square root, trigonometric, exponent, logarithm and power functions) considered in our analysis.

Pseudo code after adding barrier synchronization to ensure the compiler does not reorder:

```
start_time = clock();

__syncthreads();

repeat {
```

```
fop a a b
fop b a b
}
__syncthreads();
stop_time = clock();
```

Table IV and Table V show the measured latencies for single- and double-precision floating-point instructions. As can be seen, the latencies of single-precision instructions have improved on every new GPU generation. On the other hand, the latencies of double-precision operations have increased on the Maxwell and Pascal architecture. Maxwell and Pascal GPUs have been designed to target machine-learning applications, which can tolerate reduced arithmetic accuracy. To this end, the micro architects have been designed to offer efficient 16-bit floating-point operations and larger on-chip memories, rather than fast double-precision floating-point arithmetic.

Table IV. Single-precision floating-point instruction latencies

Instruction	Fermi		Kepler		Maxwell		Pascal	
	Std	**Fast**	**Std**	**Fast**	**Std**	**Fast**	**Std**	**Fast**
add, sub	18		9		6		6	
mul	18		9		6		6	
max, min	18		9		6		12	
fma	20		9		6		6	
div	996	40	520	18	371	15	338	16
__fdividef	89	40	42	18	34	15	34	15
rcp	204	40	110	18	87	15	80	13
sqrt	184	44	101	18	84	13	80	13
sin, cos	40	40	18	18	15	15	15	15
ex2	88	40	49	18	40	15	40	16
lg2	70	22	40	9	34	13	40	13
pow	110	58	64	27	58	21	50	20

Table V. Double precision floating-point instruction latencies

Instruction	Fermi	Kepler	Maxwell	Pascal
add, sub	22	10	48	48
mul	22	10	48	48
max, min	22	10	48	96
fma	22	10	52	53
div	1152	688	968	1114
rcp	189	126	331	338
sqrt	241	193	400	397
sin, cos	573	356	850	835
ex2	784	426	881	997
lg2	1151	700	1327	1325
pow	2488	1714	3675	3892

3.2.1 Fused multiply-add

To limit the accumulation of round-off errors when performing a multiply followed by an add operation, the IEEE 754 standard includes a *fused multiply-add* (FMA) instruction, which performs these two operations using a single rounding step rather than two (one for the multiplication and one for the addition). As can be seen, besides providing better accuracy, the FMA provides performance advantages. Specifically, its latency is lower than the sum of the latencies of multiply and add instructions.

```
//Without fast math compile option

FSETP.LT.AND P0,PT,R3.reuse,-126,PT;
@P0 FMUL R3, R3, 0.5;
RRO.EX2 R8, R3
MUFU.EX2 R2, R8
@P0 FMUL R2, R2, R2
```

```
//With fast math option



RRO.EX2 R8, R3;
MUFU.EX2 R2, R8;
```

Figure 6. Assembly code corresponding to the invocation of single-precision exponential function compiled without (to the left) and with (to the right) –use_fast_math compilation option.

3.2.2 Fast-math option

Addition, subtraction, multiplication, division, fused multiply-add and square root are IEEE compliant. In addition, the `__fdividef` operator provides a faster, but non-IEEE compliant single-precision division. Furthermore, the `--use_fast_math` compilation options allows an approximate and fast version of the division and of most special math functions (column “fast” in Table IV). These approximate operations are not IEEE compliant and they are available only for single-precision arithmetic. Their loss in accuracy is both operation and input dependent.

3.2.3 Special math operations

The portion of Table IV and Table V related to special math functions (reciprocal, square root, trigonometric, base-2 exponential, logarithmic and power functions) is highlighted in grey. The single-precision versions of these math functions are implemented in hardware by dedicated SFU. The SFU operate by looking up the value of these functions for a predefined set of input values (acceptable input set). Figure 6 (to the left) shows the assembly code corresponding to the invocation of a single-precision special function (exponent). The opcodes of the hardware instructions are prefixed with the MUFU keyword (MUFU.EX2 in the exponent example). Beside the invocation of the specific hardware instruction used, the assembly code contains a few additional instructions, whose goal is to reduce the input value and check that it is within the acceptable input set. Otherwise, the element in the acceptable input set that is nearest to the given input is fed to the SFU, adding approximation to the result. When the `--use_fast_math` compilation option is enabled (right side of Figure 6), the conditional instructions that are

part of the input reduction are removed from the assembly code, leading to faster but less accurate execution.

Double precision special math functions are implemented in software via inline subroutines. In order for the double precision instructions to have consistent latency independent on the inputs, the corresponding inline subroutines are not iterative. Because of their software implementation, the double precision math functions have longer latencies compared to their single-precision counterparts.

Chapter 4: PERFORMANCE AND ACCURACY TRADE-OFFS OF FLOATING-POINT ARITHMETIC

In this chapter, we analyze the accuracy and performance of the floating-point addition, multiplication, division, and natural exponential function operations using different arithmetic precisions. We performed our experiments on all four GPU devices of Table III. For the sake of space and clarity, we show only the results reported on the two GPU architectures that are the most dissimilar in terms of floating-point latencies, namely, Kepler and Maxwell GPUs. As shown in Table IV and Table V, the Maxwell architecture has the lowest single-precision and about the highest double-precision latencies (very similar to those of Pascal devices), while on Kepler GPUs single and double precision instructions have very similar latencies. While the absolute performance changes slightly from GPU to GPU due to the differences in instruction latencies discussed in chapter 3, the observations related to the comparison of different arithmetic precisions apply on all GPUs.

4.1 Addition

In order to study performance-accuracy tradeoffs of floating-point addition, we use the global summation benchmark proposed by Taufer et al. [4] shown below. This benchmark application computes the sum of a set of floating-point values with expected zero result in a parallel fashion. Each thread is responsible for performing the summation of a subset of the inputs, and a final reduction step accumulates the partial results computed by the different threads. Input values are randomly drawn with equal probability from two intervals with different orders of magnitude. In order to have an expected accurate result

equal to zero, for each input number, its opposite (negative) value is also added to the input set.

Pseudo code of Global Summation -

```
procedure global_summation(x[])
  sum = 0
  for i in range 1..n
    sum += x[i]
  end for
  return sum
end procedure
```

We modified the global summation benchmark so as to use CMP double2 and multiple-precision (CUMP based on 256-bit GMP) in addition to single, double, and CMP float2 precision. We performed two sets of experiments – serial and multithreaded execution – using different input intervals and sizes. Here, we show results reported on Kepler and Maxwell GPUs. Each data point represents the average over 10 runs.

4.1.1 Serial execution

Figure 7 shows the execution time and accuracy of single-threaded execution on 1 million element inputs drawn from intervals with different orders of magnitude. The accuracy is expressed in terms of absolute error (difference between the result of the summation and

the expected zero result): the larger the error, the smaller the accuracy. The absolute error of CUMP (not shown) is zero in all cases.

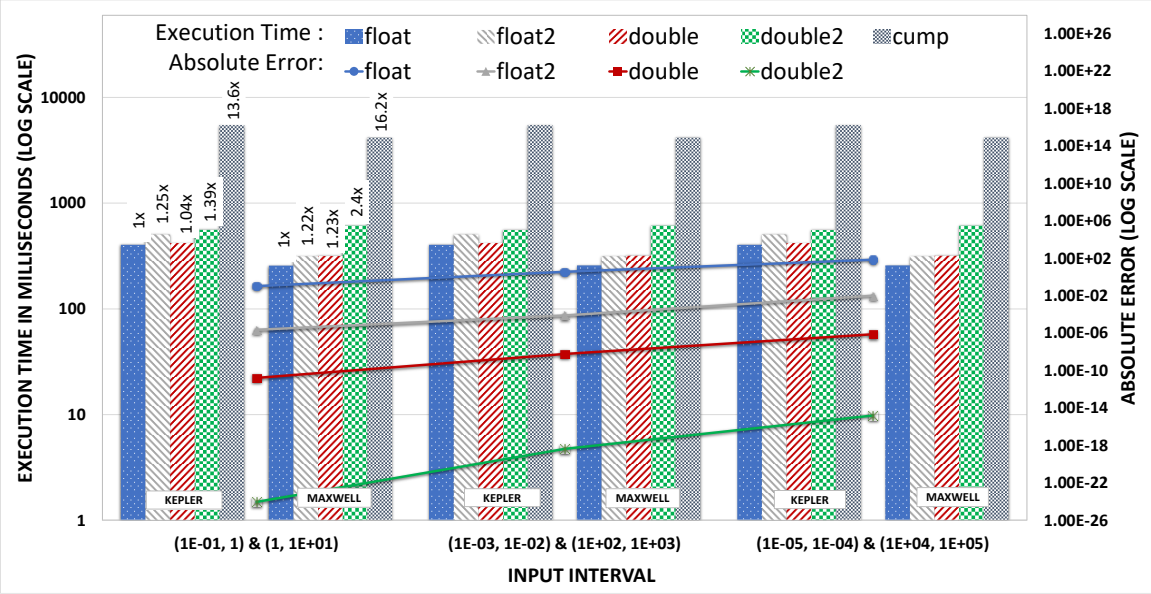

Figure 7. Global Summation serial execution on 1 million values drawn from three different input intervals (x-axis)

We make the following observations. First, the absolute error increases with the difference in order of magnitude of the input intervals. This is because, when two numbers are added together, their representation is adjusted to the same exponent before the two mantissas are summed, possibly leading to the truncation of the smallest number. Second, CMP leads to better accuracy than standard precision but at the cost of increased execution time (i.e., float2 is more accurate but slower than float, and double2 is more accurate but slower than double). However, this difference in accuracy is more significant than the difference in execution time. For example, the difference in absolute error between float and float2 is of 4-5 orders of magnitude, but the execution time of float2 is longer than that of float only

by 22-25%. Similarly, the difference in absolute error between double and double2 is of 8-14 orders of magnitude, but double2 is slower than double only by 33% on Kepler and a factor ~2x on Maxwell GPU. The limited reduction in performance of CMP over standard arithmetic is due to the fact that global summation has a low arithmetic intensity, and that Kepler and Maxwell GPUs have some support for ILP and can issue instructions to different functional units in parallel. The more pronounced performance difference between double and double2 on Maxwell (over Kepler) is due to the higher latency of double precision arithmetic on this device (Table V). Third, float2 is both less accurate and slower than double (or equally fast). This differs from the results in [4], where float2 was proposed as a compromise between float and double both in terms of accuracy and performance. This is because on pre-Fermi GPUs double precision arithmetic was significantly slower than single precision. On Maxwell, where the double precision latency is about 5 times larger than the single precision one, float2 and double perform similarly. While on current GPUs float2 does not bring any accuracy/performance advantage over double, double2 provides a good compromise between double and multiple-precision floating-point arithmetic both in terms of accuracy and performance. Double2 outperforms CUMP by a factor ~9.8x on Kepler and ~6.8x on Maxwell GPUs, while offering fairly good accuracy.

4.1.2 Parallel execution

We now consider the relationship between arithmetic precision and allowed kernel concurrency. Table VI shows the number of registers used by the global summation kernel with the considered arithmetic precisions. The register utilization of a kernel limits the

number of threads resident on a SM, that is, the number of threads that can run concurrently on a SM and, by interleaving execution, perform latency hiding. As can be seen, the register utilization of double2 leads to a register utilization that is ~3 times lower than CUMP, and the register utilization of float2 is higher than both that of float and double. Therefore, also from the point of view of the allowed concurrency and opportunities for latency hiding, float2 does not bring any advantages over standard float and double, and double2 represents a compromise between double and CUMP.

Table VI. Number of registers used by global summation kernel with different arithmetic precisions and GPU generations.

	float	**double**	**float2**	**double2**	**CUMP**
Kepler	10	12	15	28	71
Maxwell	15	18	16	24	82

Figure 8 shows the execution and accuracy of multithreaded execution using the optimal kernel configuration of 64 blocks with 32 threads each. The observations on the accuracy are the same as in the single-threaded case (multithreading slightly improves accuracy of global summation by allowing more balanced reduction trees [38]). From the performance standpoint, we observe that the slow-down of CUMP over standard and composite precision is much more pronounced for multi-threaded than for single-threaded execution. For example, float is faster than CUMP by 13.6-16.3x for serial execution, and by 32.2-36.5x for multithreaded execution (on the same dataset). This is due to CUMP's higher register utilization, which limits concurrency and reduces latency hiding opportunities.

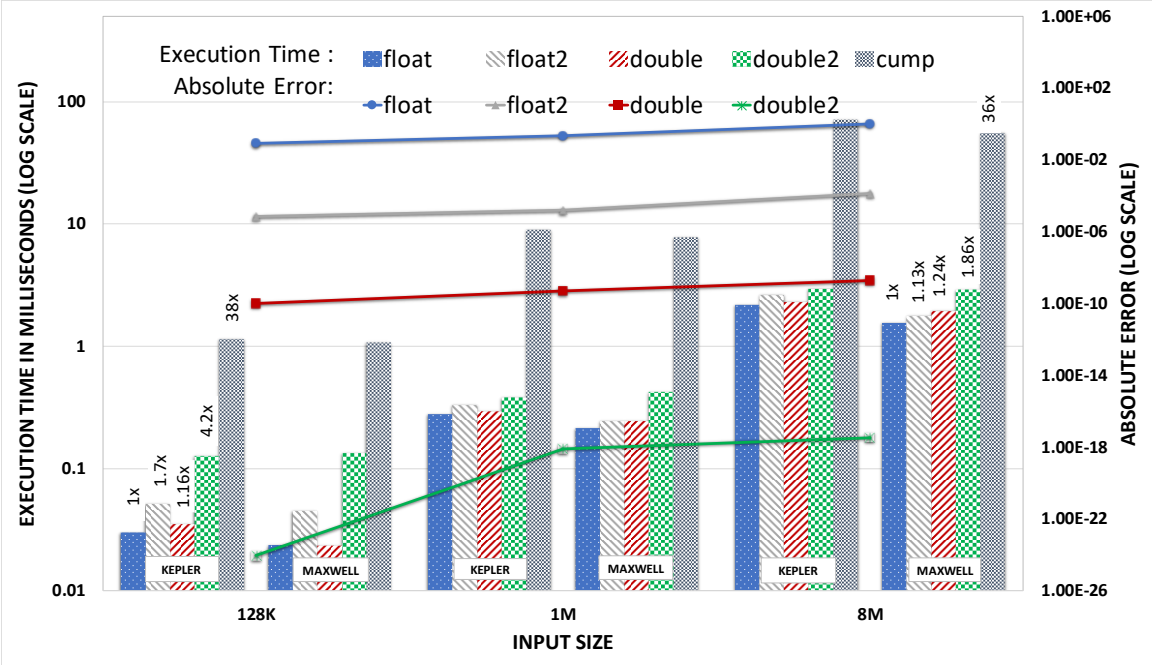

Figure 8. Global summation parallel execution with 64 blocks and 32 threads/block on inputs of various sizes (x-axis) drawn from intervals $(10^{-3}, 10^{-2})$ *&* $(10^{2}, 10^{3})$

4.2 Multiplication

To evaluate the performance and accuracy of floating- point multiplication using different arithmetic precisions, we used a micro-benchmark application that performs a sequence of multiplication instructions. At every iteration, the result of the previous iteration is multiplied by an input value. In order to avoid underflow and overflow, the input set consists of numbers randomly drawn from two intervals, one including values less than 1 and the other including values greater than 1 (we alternate between the two intervals over iterations). We compare the results obtained using float, double, float2 and double2 arithmetic with CUMP reference results.

Pseudo code for Multiplication micro benchmark -

```
procedure multiplication(x[])
result = 1
for i in range 1..n
  result *= x[i]
end for
return result
end procedure
```

Figure 9 and Figure 10 shows the accuracy and performance results reported on Kepler and Maxwell GPUs (to the top and to the bottom, respectively). The execution time is measured in clock cycles, and it includes only the sequence of multiplications (and not the initial load and final store instructions). We performed a number of multiplications varying from 20 to 220. We make the following observations. The multiplication error is generally propagated from iteration to iteration. From the accuracy angle, float2 and double2 exhibit an error comparable to that of float and double, respectively. This can be explained as follows. The last term of the error component of the CMP multiplication formula in Figure 3 (x.error * y.error) tends to be much smaller than the first two terms, leading to truncation when added to the other terms. Dekker's split (Figure 2) was designed to increase the accuracy of composite precision addition while not compromising the multiplication accuracy (when compared with standard floating-point arithmetic). In terms of execution time, float2 and double2 are 1.2-1.5 times slower than float and double (respectively) on Kepler GPUs, and 3-4 times slower than float and double on Maxwell GPUs. This is

because composite precision arithmetic requires four standard floating-point multiplications and two standard floating-point additions (the latency of some of these operations, however, is hidden by ILP). CUMP (not shown) experiences an 80x slow-down over float. Finally, on the considered GPUs, the multiplication micro-benchmark requires 22, 26, 28, 32, and 99 registers for float, double, float2, double2 and CUMP, respectively. Therefore, as for addition, CUMP leads to less kernel concurrency than standard and CMP arithmetic, resulting in more pronounced performance disadvantages when the degree of multithreading is high. Finally, we recall that on Kepler single- and double-precision multiplications have similar latencies, while on Maxwell GPUs the latency of double-precision operations is significantly higher (Table IV and Table V). This motivates the different performance behaviors on the two architectures shown in Figure 9 and Figure 10.

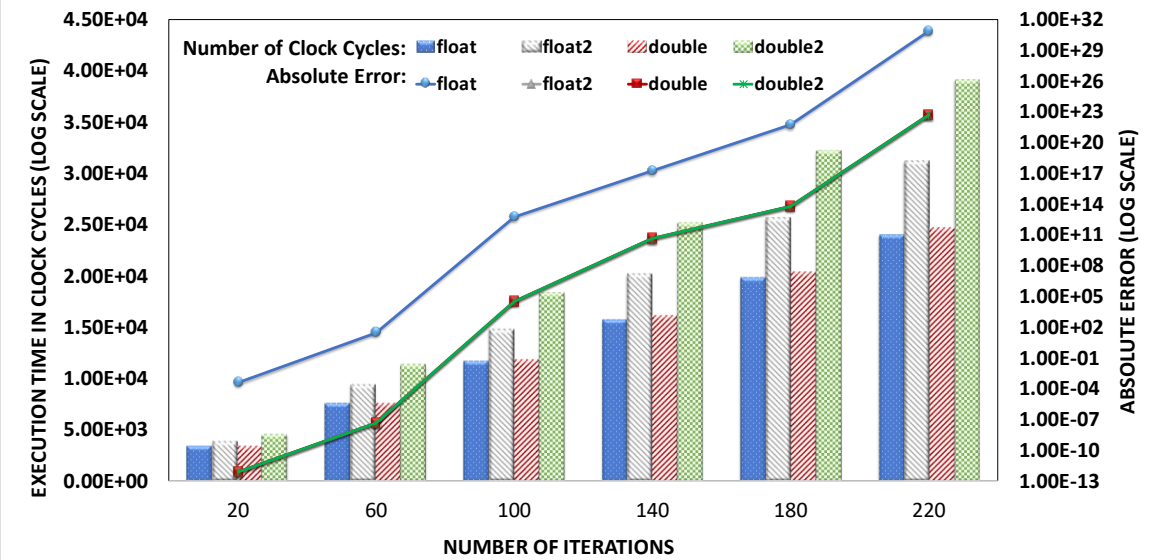

Figure 9. Accuracy and performance results of repeated multiplications on Kepler GPU with inputs randomly drawn from intervals (0.01, 1) and (1, 10).

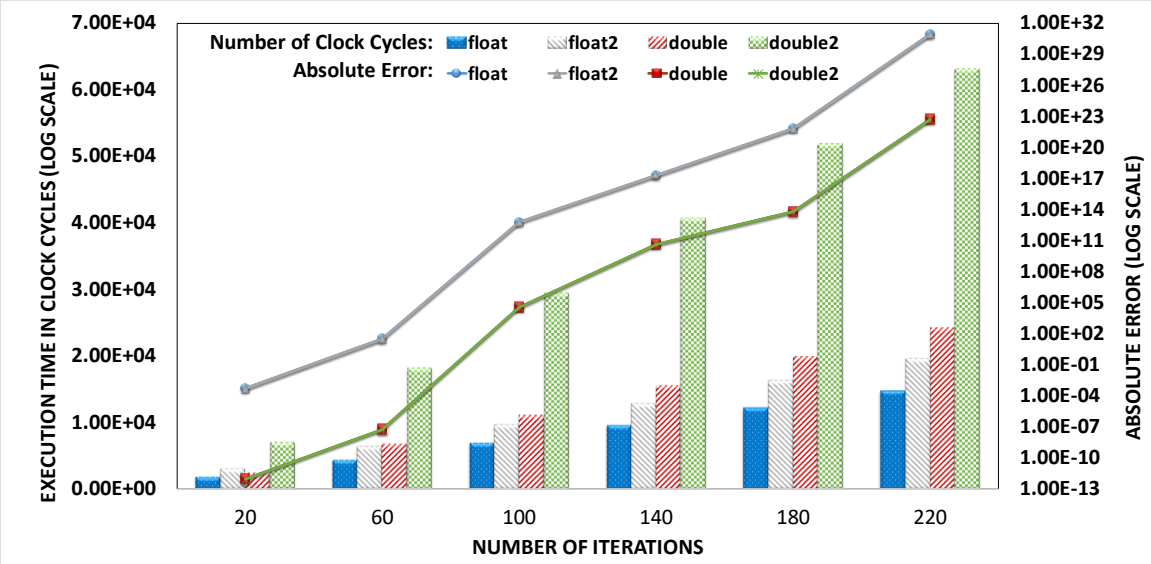

Figure 10. Accuracy and performance results of repeated multiplications on Maxwell GPU with inputs randomly drawn from intervals (0.01, 1) and (1, 10).

In summary, float2 and double2 are not advantageous over standard floating-point precision for multiplication: they provide similar accuracy but exhibit longer execution time than float and double, respectively.

4.3 Division

In order to study the behavior of the division, we use the do-undo benchmark [4], which performs a sequence of interleaved multiplications and divisions such that the expected result of the computation is equal to the original input value. Specifically, given an input value *x* and an input array *y*, the do-undo benchmark performs a sequence of $(x * y_i)/y_i$ operations. In our experiments, we use the IEEE compliant division operator (`__fdiv_rn()` intrinsic function).

Figure 11, Figure 12, Figure 13, and Figure 14 presents the absolute error of 1 million iterations of the do-undo benchmark using inputs of different orders of magnitude. In Figure 11 and Figure 12 we use small values for x, small and large values for y respectively: specifically, x is randomly drawn from interval (0.0, 10.0) and y is randomly drawn from interval (0.0, 10.0) and (0.0, 10^6) for small and large value respectively. In Figure 13 and Figure 14, we invert the selection, and we consider large values drawn from interval (0.0, 10^6) for x and values randomly drawn from interval (0.0, 10.0) and (0.0, 10^6) for y respectively. We report the results using float and float2 (we observed the same trends with double and double2). As can be seen, the results obtained using float2 are consistently less accurate than those obtained using float. From the multiplication micro-benchmark, we know that the accuracies of float and float2 multiplications are comparable. We infer that the inaccuracy of the float2 results must be related to the use of the division.

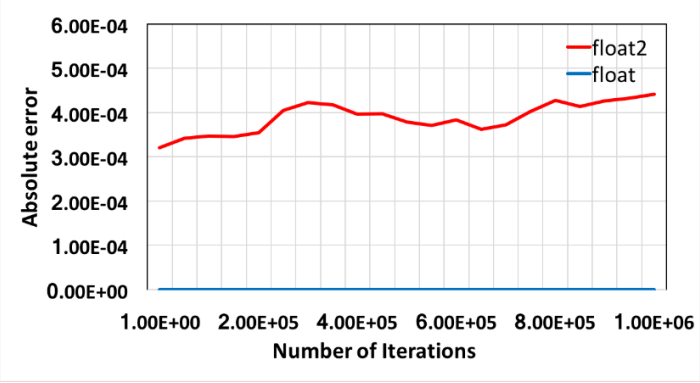

Figure 11. Absolute error reported by the do-undo benchmark (1 million iterations) with input orders ***x*** *(0, 10) and* ***y*** *(0,* 10^6*)*

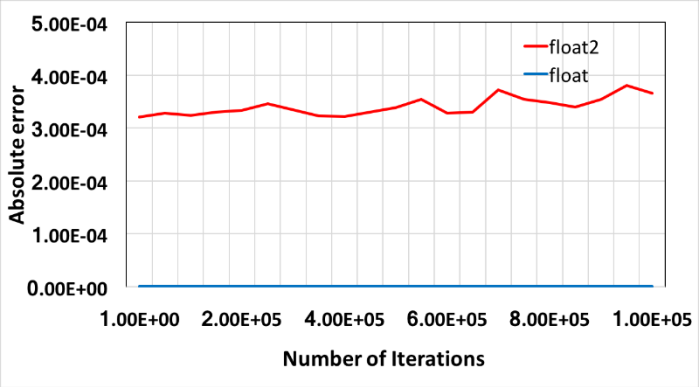

Figure 12. Absolute error reported by the do-undo benchmark (1 million iterations) with input orders ***x*** *(0, 10) and* ***y*** *(0, 10)*

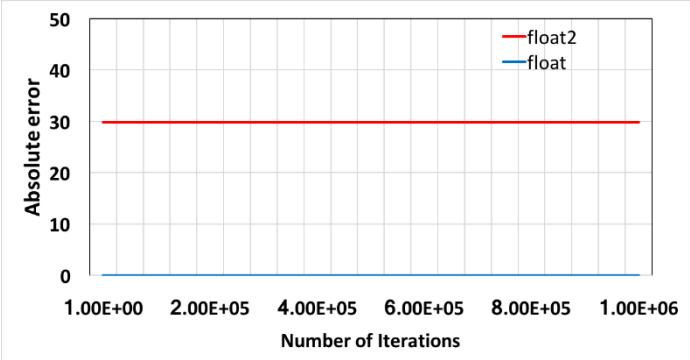

Figure 13. Absolute error reported by the do-undo benchmark (1 million iterations) with input orders ***x*** *(0,* 10^6*) and* ***y*** *(0, 10)*

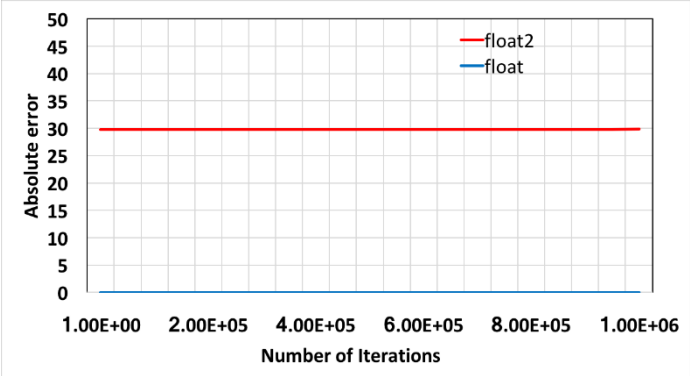

Figure 14. Absolute error reported by the do-undo benchmark (1 million iterations) with input orders ***x*** *(0,* 10^6*) and* ***y*** *(0,* 10^6*)*

From Figure 3, we observe that the equations defining the CMP division algorithm ignore the error component of the operands. This leads to inaccuracies that depend on the magnitude of this component. Our results differ from those reported in [4]. This is due to two facts. First, pre-Fermi GPUs used in [4] do not have support for IEEE compliant division. Second, those results were obtained using two distinct division operators in the float and float2 experiments.

We conclude that CMP arithmetic is not beneficial for the division operation: it leads to less accurate results than standard floating-point precision while also causing loss in performance (due to the additional standard floating-point precision operations it requires).

4.4 Natural Exponent Function

In this section, we study the natural exponential function, which is used in several scientific applications, including the molecular dynamic simulation benchmark considered in chapter 5. We extend the CMP library to support this function. To this end, we consider two implementations: one based on the Taylor series approximation and one based on the built-in base-2 exponential function as shown below (ex2 in Table IV and Table V).

Thall [30] proposed approximating the natural exponential function using its Taylor series.

$$\exp(x) = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!} \tag{3}$$

This equation is implemented as an iterative algorithm, in each iteration we compute the new term ($\frac{x^n}{n!}$). The termination condition for the series is shown below:

$$latest\ term\left(\frac{x^n}{n!}\right) > 1e-20 * \exp(x.value) \tag{4}$$

Since the termination condition depends on the input value, the execution time of this implementation of the natural exponential function is also input-dependent.

Using the built-in ex2 function, we can implement the composite precision natural exponential function using the following formula:

$$\begin{aligned} \exp(x) &= \exp(x.value + x.error) \\ &= \exp(x.value) * \exp(x.error) \end{aligned} \quad (5)$$

where $\exp(y) = ex2(y * 1.4227)$, this is how the exponent is implemented in using power of 2 (ex2) hardware instruction.

We implemented three kernels that compute the natural exponential function: the first one (float/double) invokes the built-in exponential function (BF) on single- and double-precision floating-point values, the second one (float2/double2_BF) uses the implementation of the composite-precision exponential function based on the ex2 built-in function, and the last one (float2/double2_TS) uses the implementation of the composite-precision exponential function based on the Taylor series (TS). We generated the assembly code for these kernels and analyzed its instructions. Table VII shows the assembly instructions generated in case of single, double, and composite precision arithmetic. We recall (Chapter 3) that the ex2 function is implemented in hardware for single-precision and in software for double-precision arithmetic. Thus, the versions of the natural exponential function based on the double-precision built-in ex2 function have significantly more instructions than their float counterparts. To understand the instructions in the float case, we recall that the compiler adds some assembly instructions to check that the input is

within a set of acceptable values (Figure 6). The n parameter in the implementations based on the Taylor series represents the number of iterations (i.e., the number of terms in the series that are added together).

Table VII. Arithmetic instructions generated for the different versions of the natural exponential function

Instruction	**Single-precision**			**Double Precision**		
	float	***float2_BF***	***float2_TS***	***double***	***double2_BF***	***double2_ TS***
add		13	16*n	6	24	13*n
mul	3	9	6*n	3	13	6*n
fma				13	26	
div			1*n			1*n
rro	1	2				
ex2	1	2		1	2	

Although in this thesis we focus on the natural exponential function, similar considerations apply for the logarithmic and power functions. As shown in Table IV and Table V, the base-2 logarithmic function (lg2) is available and implemented in hardware for single- and in software for double-precision arithmetic. The float power function (pow) is implemented using a combination of ex2 and lg2 instructions.

Figure 15 and Figure 16 shows the accuracy – defined in terms of absolute error over the result of the MPFR library – of the different versions of the natural exponential functions (since CUMP does not support the exponential function, we computed the reference multiple-precision result on CPU). We show two ranges of inputs: (0.1, 1) and (1, 100). We recall that we use the convergence criterium (4) to set the value of n (the loop count of the Taylor series for the TF implementations); this leads to the values reported in Table VIII. For inputs less than 0.1, float2_TS converges in very few iterations, and float, float2_BF and float2_TS have similar accuracy. We make the following observations. First, for implementations based on the ex2 built-in function, the accuracy of composite-

precision is similar to the one of standard precision arithmetic (that is, float and float2 have comparable accuracy, and so do double and double2) this is due to multiplications involved in composite precision exponent. Second, implementations based on the built-in ex2 function exhibit better accuracy than implementations based on the Taylor series because the later algorithm is an approximation and we limit the number of iterations which causes the error to increase. Third, as expected, double/double2 have better accuracy than float/float2. Fourth, the absolute error is constrained when the input is less than 1 and the result of the exponential function itself is constrained, and it diverges when the input is greater than 1, this happens due to the range reduction operation used to approximate the values supported by ex2 hardware instruction. In addition to high error in Taylor series, because both the numerator and the denominator of the terms of the Taylor series diverge for large n when x is greater than 1, implementations based on the Taylor series cannot support large inputs (on the bottom chart of Figure 16, missing values in the float2_TS series correspond to NaN results).

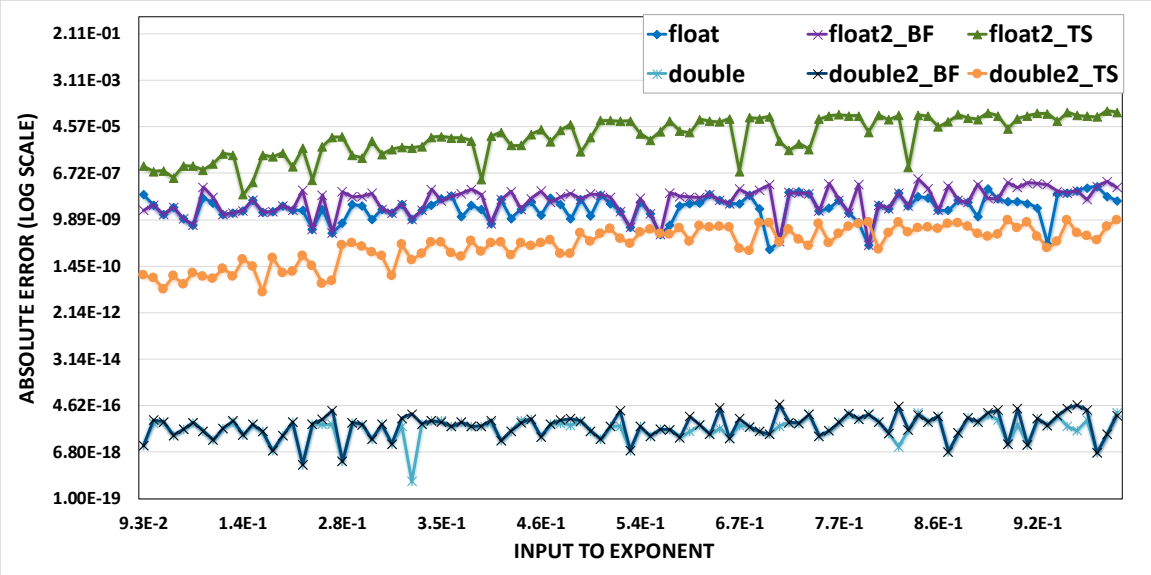

Figure 15. Accuracy (in terms of absolute error over GMP result) of the natural exponential function for two input ranges: (0.1,1)

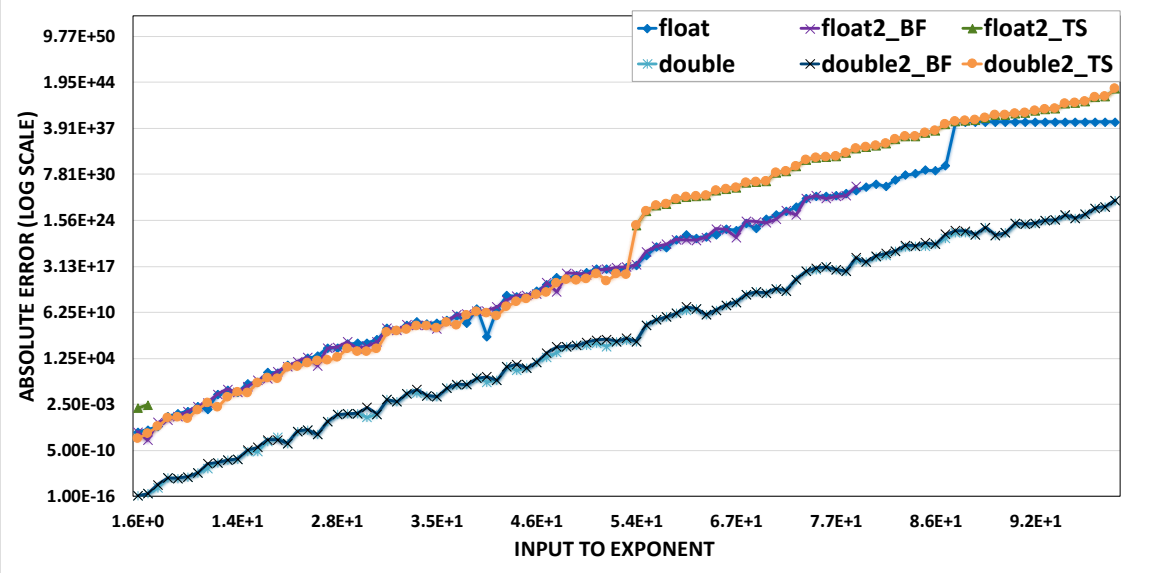

Figure 16. Accuracy (in terms of absolute error over GMP result) of the natural exponential function for two input ranges: (1, 100)

Table VIII. Mean and Standard deviation of the loop count "n" in the cmp taylor series

Input Range	Mean		Standard deviation	
	Float2	Double2	Float2	Double2
(0,0.1)	10.6	10.6	1.3	1.3
(0,1)	17.5	17.5	3.1	3.1
(0,100)	14.7	46.8	13.1	50

Figure 17 and Figure 18 shows the performance of the natural exponential function on the Kepler and Maxwell architecture. We make the following observations. First, for implementations based on the built-in function, composite precision performs slightly worse than standard precision arithmetic (the execution time of float2 and double2 are 30% and 62% longer than the execution times of float and double, respectively). This is due to the larger number of instructions executed by the composite precision functions. Second, in this case the execution times of double and double2 are significantly longer than those of float (by a factor 4x) and float2 (by a factor 5x), respectively. This is because the double-precision ex2 function is implemented in software rather than in hardware, and the double2 exponential function executes significantly more instructions than the float2 counterpart (see Table VII). Third, the composite precision implementations based on the Taylor series are significantly slower than the ones based on the ex2 built-in function (by a factor 2-15x). This is because, by executing multiple iterations (on average 10-15 for the considered inputs), they result in a larger number of assembly instructions. Finally, as expected, the execution time of implementations based on the Taylor series is data dependent (due to the nature of the termination condition). Finally, since the latency of the double-precision exp2

instruction is lower on Kepler than on Maxwell GPUs (Table V), double-precision implementations of the exponential function also perform better on Kepler devices.

In summary, for the exponential function standard floating-point arithmetic provides the same accuracy as composite precision arithmetic but offers slightly better performance. Composite precision implementations of the exponential function based on Taylor series offer poor accuracy and performance. Double precision provides significantly better accuracy than single precision arithmetic, but at a noticeable performance cost.

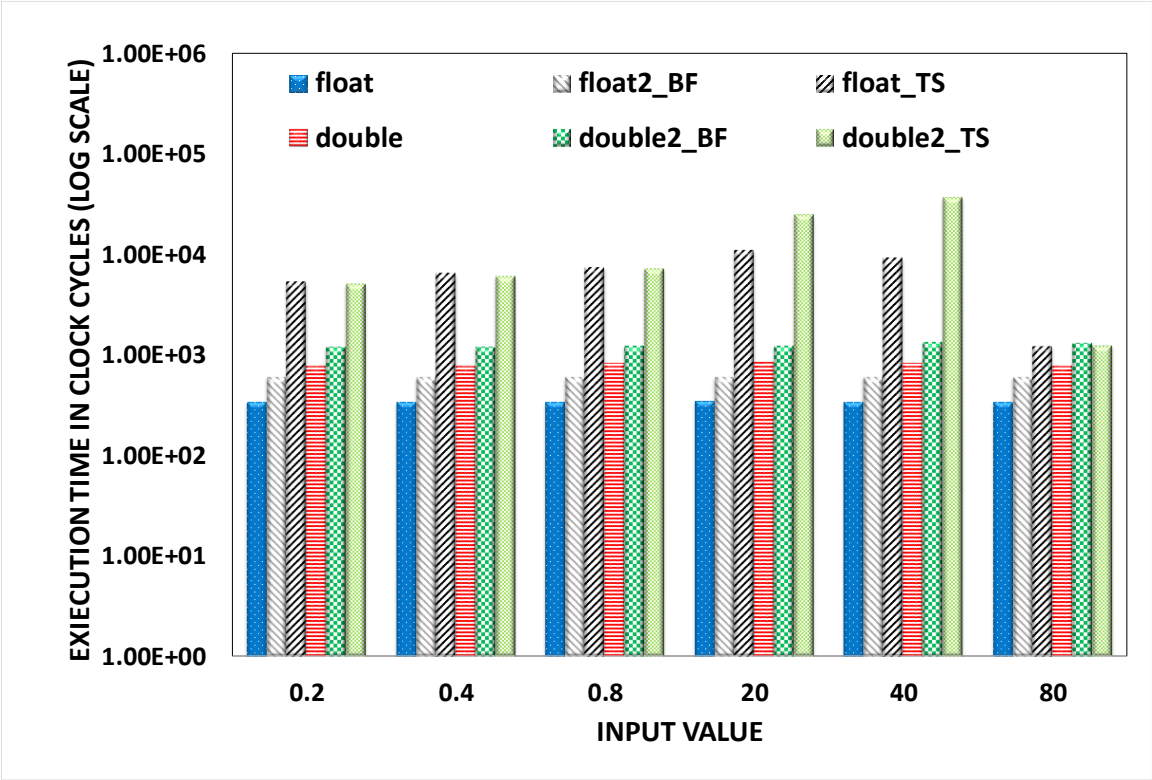

Figure 17. Execution time of exponential function on Kepler GPU for input values drawn from ranges (0.1, 1) and (1, 100).

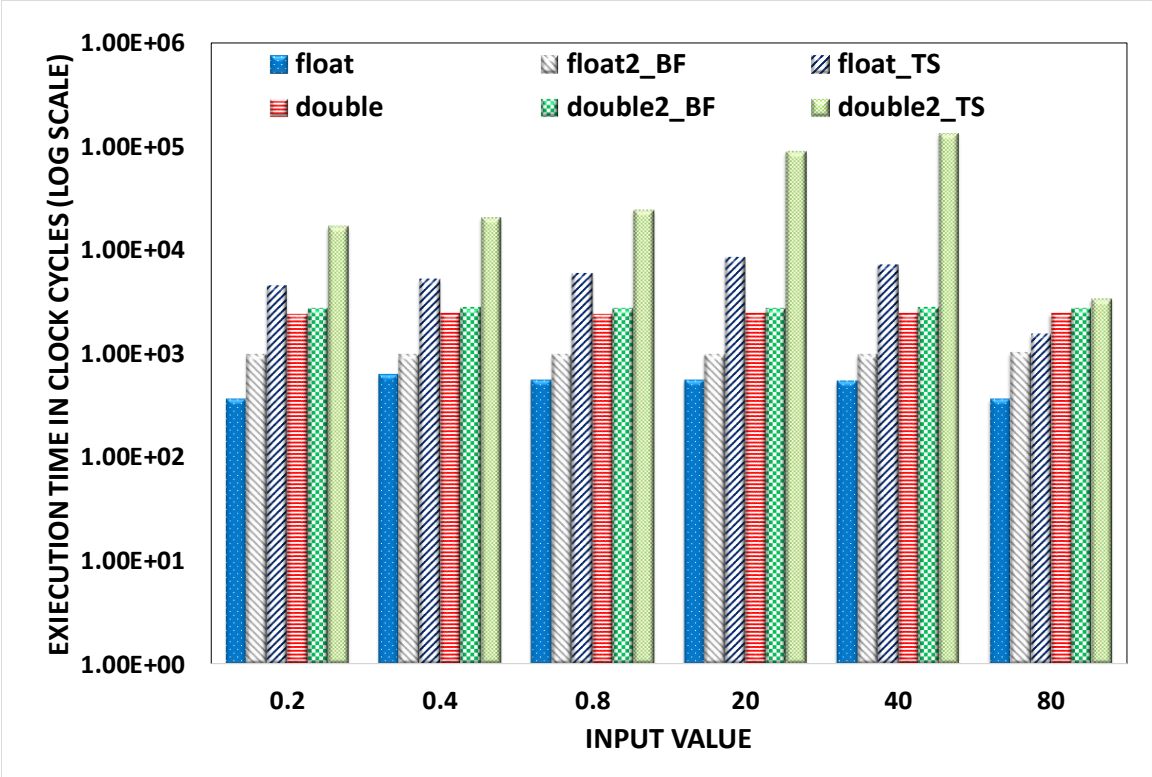

Figure 18. Execution time of exponential function on Maxwell GPU for input values drawn from ranges (0.1, 1) and (1, 100).

Chapter 5: PERFORMACE AND ACCURACY TRADE-OFFS OF REAL APPLICATIONS

In this chapter, we study the use of different floating-point precisions on real-world applications that perform different arithmetic operations. These applications are selected because they have different instruction mixes and different arithmetic intensities (low, medium and high). In particular, we consider three applications from Rodinia Benchmark suite [39]: Gaussian Elimination, LU Decomposition and LavaMD molecular dynamics. As in Chapter 4, we show the results reported on Kepler and Maxwell GPUs. We compute the arithmetic intensity of the kernels based on the number of floating-point instructions executed and the number of 128-byte global memory transactions performed as reported by NVidia CUDA profiler. In addition, we derive register and shared memory utilization from NVidia NVCC compiler.

5.1 Gaussian Elimination

Gaussian Elimination (GE) is a method to solve a system of linear equations Ax=b where A is an n×n coefficient matrix, and b is an n×l vector. GE first reduces the system into an upper triangular form (forward-substitution), and then solves the linear equations by applying back-substitution. In Rodinia's GE the forward-substitution is implemented on GPU using two kernels and the back-substitution is performed on CPU. The forward-substitution kernels use a combination of floating-point additions, multiplications, and divisions (2, 2 and 1, respectively) and are invoked multiple times in a loop. The output of the division is fed to the other operations.

Figure 19 and Figure 20 shows the accuracy and performance results of GE on square matrices A of size *n* varying from 16 to 4096. The input values of matrix A and vector b are randomly drawn from intervals $(10^{-2}, 10^{-1})$ & $(10^{1}, 10^{2})$. We tested several block sizes: from 128 to 1024 threads/block for kernel 1 and from 16x16 to 32x32 threads/block for kernel 2. The grid size is set according to input and block size. We show the results of the kernel configurations leading to the worst (to the left) and best (to the right) performance (the block sizes are reported in Figure 19 and Figure 20). Accuracy is measured in terms of the sum of the absolute errors of the elements of the result matrix over a reference result computed using the 256-bit MPFR library.

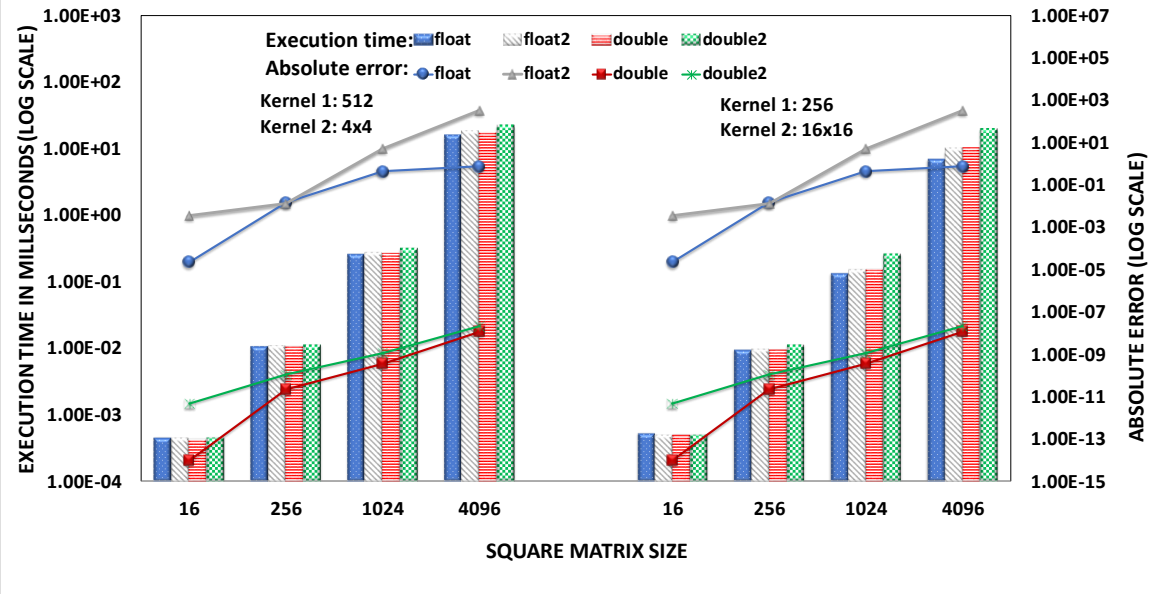

Figure 19. Accuracy and performance results of GE with elements of matrix A and vector b randomly drawn from intervals $(10^{-2}, 10^{-1})$ *&* $(10^{1}, 10^{2})$ *on Kepler GPU.*

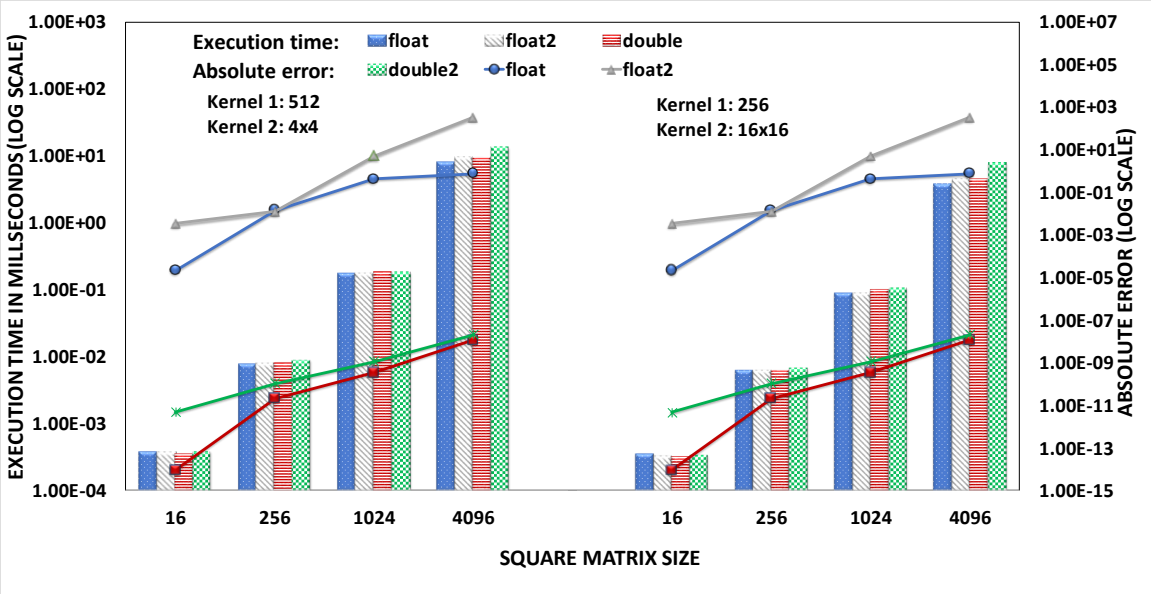

Figure 20. Accuracy and performance results of GE with elements of matrix A and vector b randomly drawn from intervals $(10^{-2}, 10^{-1})$ *&* $(10^{1}, 10^{2})$ *on Maxwell GPU.*

We observe that the accuracy of CMP is comparable to that of standard floating-point precision arithmetic. This is because the accuracy advantage of CMP addition is nullified by the less accurate CMP division operation. Double-precision arithmetic brings a substantial accuracy advantage (by ~3-4 orders of magnitude) over single-precision arithmetic. Execution times of CMP and standard floating-point arithmetic are very similar, due to the low arithmetic intensity of the GE forward-substitution kernels (0.08 and 1.12 flops/byte for float/double and CMP respectively).

5.2 LU Decomposition

LU decomposition is a factorization technique to transform a square matrix A into a lower and an upper triangular matrix. Rodinia's LU implementation includes three iterative GPU kernels, which perform a variable number of divisions, multiplications and additions (or

subtractions). The computation of the result matrices is tiled, and the tiles are loaded into shared memory. The maximum block size allowed by the use of shared memory is 32x32.

In Figure 21 and Figure 22 we show the accuracy and performance results of LU on square matrices A of size n varying from 64 to 2048. The input values of matrix A are again randomly drawn from intervals $(10^{-2}, 10^{-1})$ & $(10^{1}, 10^{2})$. We tested block sizes varying from 4x4 to 32x32 and in all cases set the grid size to (n/block size). Here, we show the results reported by the worst (left) and best (right) kernel configurations (block size equal to 4x4 and 16x16, respectively). We measure accuracy as for the GE benchmark. We make the following observations. First, CMP exhibits worse accuracy than standard arithmetic (float2 is less accurate than float, and double2 is less accurate than double). This is due to the high number of multiplications and divisions, which do not benefit from the use of composite precision. Second, the execution time of double precision is higher than that of single precision kernels. This is because double precision leads to higher register and shared memory utilization than single precision (20, 44, 44 and 57 registers per thread and 192, 384, 384 and 786 bytes of shared memory per 16x16 block for float, float2, double and double2, respectively), in turn limiting the number of resident blocks on a SM and the opportunities for memory latency hiding. Finally, the execution time of CMP is longer than that of standard floating-point arithmetic. This is due to the higher instruction count, register and shared memory requirements of the CMP kernels.

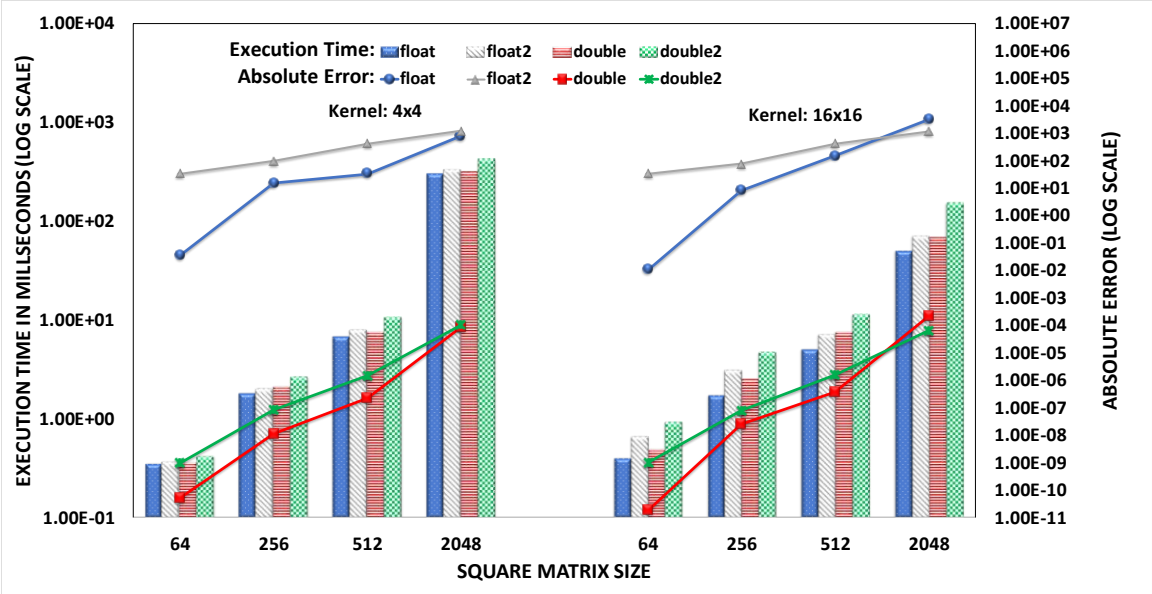

Figure 21. Accuracy and performance results of LU with elements of matrix A and vector b randomly drawn from intervals (10^{-2}, 10^{-1}) & (10^{1}, 10^{2}) on Kepler GPU.

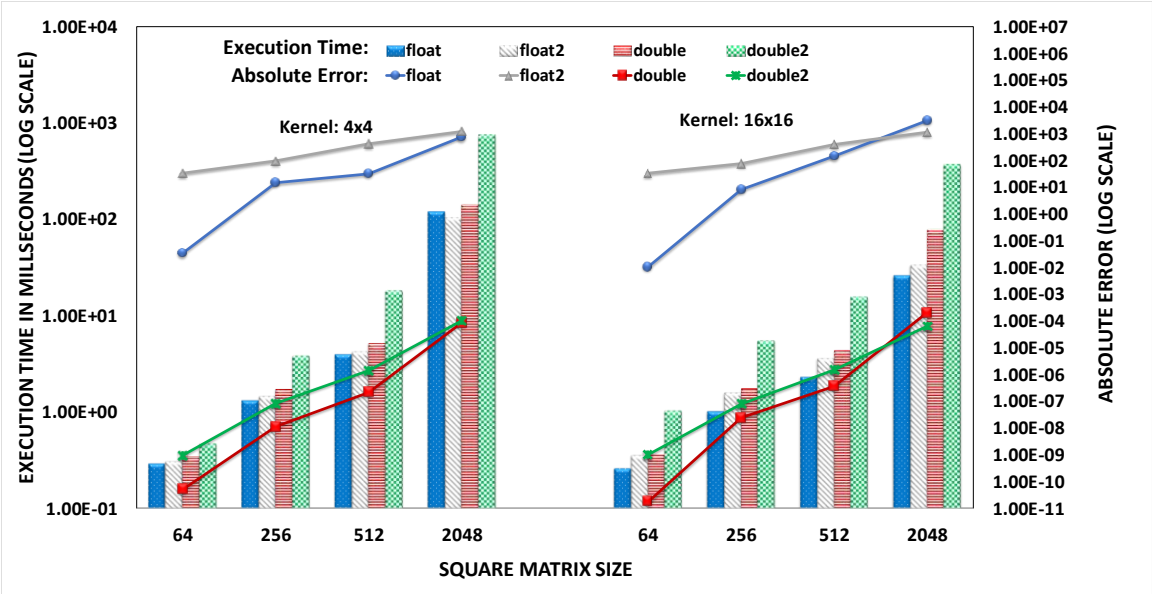

Figure 22. Accuracy and performance results of LU with elements of matrix A and vector b randomly drawn from intervals (10^{-2}, 10^{-1}) & (10^{1}, 10^{2}) on Maxwell GPU.

In summary, we confirm that composite precision arithmetic is not beneficial in the presence of a substantial number of multiplications and divisions. In addition, for kernels using shared memory, the higher memory footprint of CMP arithmetic can limit parallelism and penalize the performance over standard precision arithmetic. Furthermore, since LU has higher arithmetic intensity than GE (4.73 and 5.84 flops/byte for float and double, 22.45 and 24.46 flops/byte for float2 and double2, respectively), floating-point operations weigh more on performance.

5.3 LavaMD benchmark

LavaMD calculates the particle potential and relocation due to mutual forces between particles within a large 3D space. Particles are spatially divided in boxes. In LavaMD simulations, particles interact with other particles that are within a certain cutoff radius since those at larger distances exert negligible forces. LavaMD's kernel has medium arithmetic intensity (1.8 and 1.21 flops/byte for float and double, 2.97 and 3.42 flops/byte for float2 and double2, respectively) and is iterative; each iteration of the loop within the kernel includes 16 multiplications, 11 additions and 2 exponential functions. In addition, this kernel uses shared memory to store the initial forces of the neighboring particles.

Figure 23, Figure 24, Figure 25, and Figure 26 shows accuracy and performance results of LavaMD. Accuracy is measured in terms of the sum of the absolute errors on all the particles over a reference result computed using the 256-bit MPFR library. The initial values of the particle forces are randomly drawn from an input interval. We show our analysis on two input intervals: (0, 0.1), and (0, 1). The kernels are configured such that the number of particles within a box is equal to the number of threads per block and the

number of boxes is equal to the number of blocks. In the experiments, we use a fixed block size of 128 threads and vary the number of blocks from 1 to 512. We consider two composite precision implementations: one where the exponential function is based on the ex2 built-in function and one where it is based on the Taylor series.

We make the following observations.

5.3.1 Accuracy

First, implementations based on double precision arithmetic (double, double2_BF, double2_TS) exhibit better accuracy than implementations based on single-precision arithmetic. Second, among these implementations, double2_BF and double2_TS exhibit the best and worst accuracy, respectively. This is coherent with the analysis presented in section 4.4: double2_BF is more accurate than double on additions and the two data types exhibit similar accuracy on multiplications and exponential functions; meanwhile, on exponential functions double2_TS is the least accurate among the three. Third, the accuracy results are slightly different for implementations based on single precision arithmetic. In this case, the difference in accuracy between the float, float2_BF and float2_TS implementations of the exponential function are very input dependent. For example, for very small inputs, the float2_TS exponential function converges fast and exhibits similar or better accuracy than the float and float2_BF implementations (Figure 15); its accuracy, however, decreases as the input increases. The behavior of the exponential function affects the accuracy of LavaMD. While for single-precision based implementations composite precision is generally more accurate than standard precision, the relative accuracies of float, float2_BF and float2_TS are input dependent.

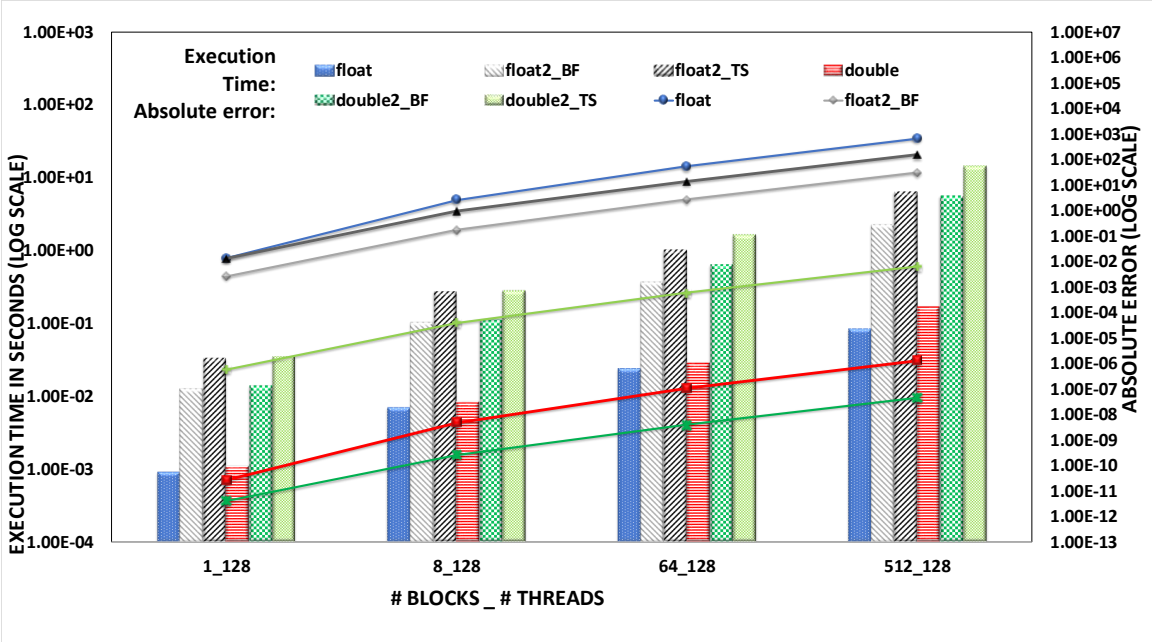

Figure 23. Accuracy and execution time of LavaMD for input range (0, 0.1) on Kepler GPU

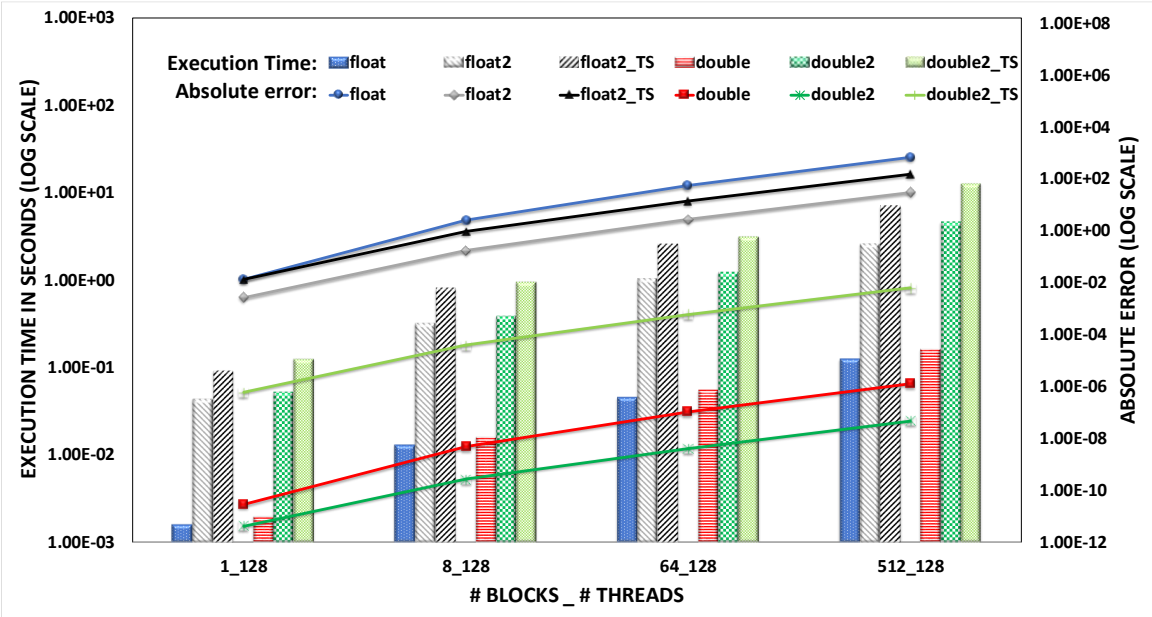

Figure 24. Accuracy and execution time of LavaMD for input range (0, 0.1) on Maxwell GPU

5.3.2 Performance

First, due to their higher floating-point instruction count and shared memory requirement (9216 and 18432 bytes of shared memory per block for float2 and double2, 4680 and 8216 bytes shared memory per block for float and double), composite precision implementations are significantly slower than their standard precision counterparts. Second, as expected from the analysis in section 4.4, the implementations based on the Taylor series have the worst performance. Third, double-precision is only slightly slower than single-precision arithmetic. To conclude, for applications with LavaMD profile, standard double precision provides the best tradeoff between accuracy and performance, while double2_BF is preferable when accuracy is a primary concern.

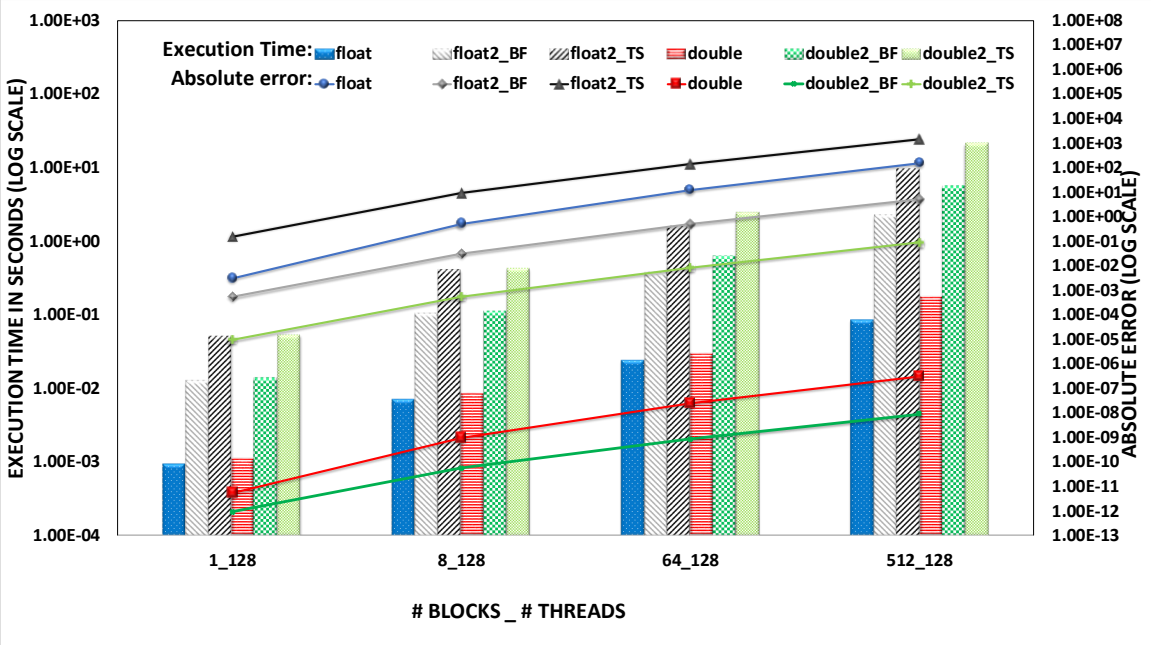

Figure 25. Accuracy and execution time of LavaMD for input range (0, 1) on Kepler GPU.

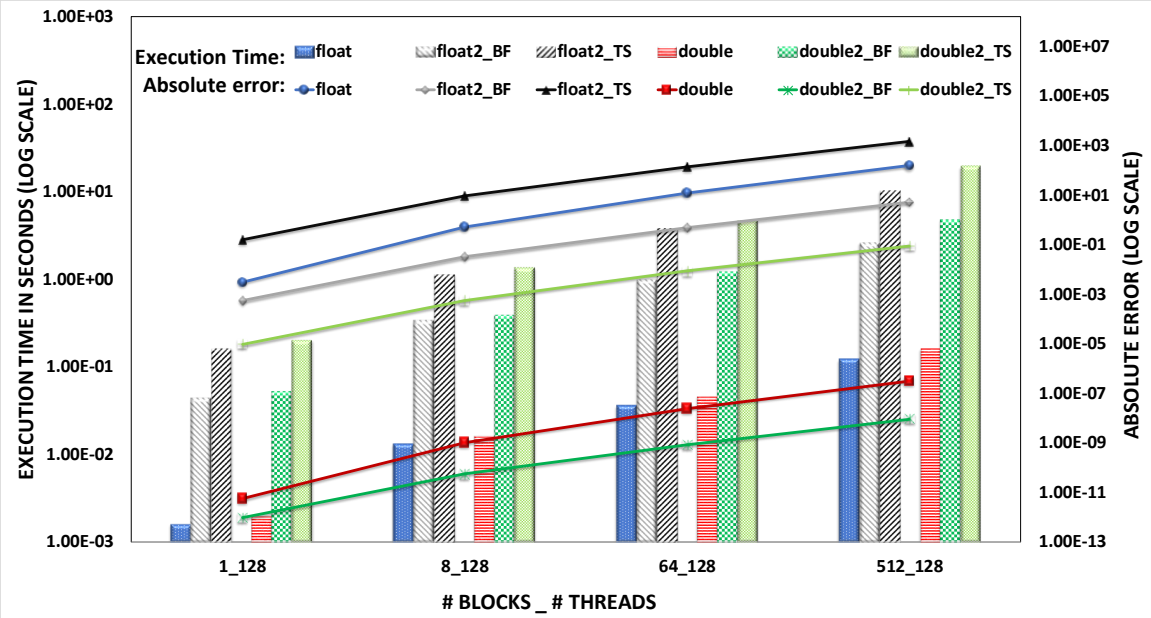

Figure 26. Accuracy and execution time of LavaMD for input range (0, 1) on Maxwell GPU.

Chapter 6: GPU PROGRAM AUTO-TUNER

In this chapter, we discuss the motivation and necessity for building an auto tuner for GPU programs. Section 6.1 describes the impact of hardware utilization on the performance of an application. In Section 6.2, we show how to build a micro benchmark specifically to study the throughputs of a modern GPU. In Section 6.3 we propose an auto tuner for the GPU program and in Section 6.4 we study some existing auto tuners. Finally, in section 6.5 we consider the challenges and features of the proposed auto tuner.

6.1 Hardware Utilization

Often programmers with minimal knowledge about floating point operation prefer to safely build an application in a higher precision supported program, even though it is not necessary. Further, not all variables in a program need a higher precision to provide an accurate result. It may be convenient to write a program in one precision. However, if the operations and inputs are involved in only part of the program and can be set at a higher precision than the rest of the program, writing such a mixed-precision program requires expert knowledge about the hardware architecture and floating-point. Also, the challenge in mixed-precision programming is determining which operations should be performed at a lower precision. For example, most deep neural network frameworks for training do the multiplication at half-precision and the accumulation at single-precision.

Nvidia's streaming multiprocessors as described in chapter 3 consist of single precision and double precision specific cores which can run in parallel. Moreover, on modern GPU

architectures, double precision instructions have higher latency than single precision instructions (Table V) on modern GPU architectures (Maxwell and Pascal). Another important performance trait pertains to the throughput (instructions per cycle) of double-precision being significantly lower compared to single-precision performance. This also applies to integer precision, since smaller datatypes have better throughput. Moreover, the modern GPU architectures (Pascal) support half-precision floating point implementation, which has 2 $\times$ the throughput of single precision. Considering these characteristics and architectural shifts, a double precision program runs slower and only occupies the double-precision cores leaving the single-precision cores idle. This results in underutilization of hardware.

One solution for solving this performance problem and underutilization is mixed-precision. GPUs are highly multithreaded; thus, running mixed-precision kernels occupy both the single precision and double precision cores on the streaming multiprocessor (SM). This results in better hardware utilization and improved performance by utilizing the faster single-precision cores. Also, lower precision execution units toggle fewer gates compared to higher precision. So, a mixed-precision program reduces the dynamic power as well.

6.2 Micro benchmark to verify the mixed-precision hardware utilization

To verify the above theory, we considered a microbenchmark which measures the throughputs of single-, double-, and mixed-precision. This microbenchmark consists of back-to-back data dependent repeated floating-point additions in single precision and double precision. For the mixed precision we interleaved both double and single precision floating-point additions together as shown in Figure 27.

Pseudo code for the micro benchmark:

```
//Single-/double-precision

load a
load b
repeat {
fadd/dadd a a b
fadd/dadd b a b
}
```

```
//Mixed precision
load a
load b
load double d
load double c
repeat {
fadd a a b
dadd c c d
fadd b a b
dadd d c d
}
```

Figure 27. Micro benchmark to measure the throughput of single-, double- and mixed-precision

We measured the throughputs of the above kernels on a Maxwell TitanX GPU with add latencies - six cycles for single-precision and 48 cycles for double-precision. Table IX shows the results of throughputs for three kernels.

Table IX. Floating point addition micro benchmark throughputs for single-, double-, and mixed-precision

Theoretical Single Precision	*6.6 TFlops*
Theoretical Double Precision	*206 GFlops*
Single Precision	*2.4 TFlops*
Double Precision	*93 GFlops*
Mixed Precision	*186 GFlops*

Single- and double-precision throughputs are significantly lower than theoretical numbers. This is because theoretical peak measurements are taken considering every execution unit is supplied with instructions at every cycle.

The mixed-precision program with 50:50 interleaves of single-precision and double-precision is two times faster than the double-precision kernel. Thus, mixed-precision improves the performance of the program by utilizing all available functional units. Consider if the double-precision program can be transformed to a mixed-precision consisting of 90% single-precision and 10% double-precision.

6.3 Automatic tuning of a GPU Program

As mentioned earlier it is difficult even for an expert to program a mixed-precision program. With high performance computing being the mainstream programming model for cryptocurrencies, machine learning and other scientific applications, performance of these applications is highly essential. To address this problem, we propose an LLVM (low-level virtual machine) [40] based compiler tool which auto tunes the precision of a GPU program. The goal of this auto-tuner when provided with a higher precision program is to make sure the output is an optimized mixed-precision kernel with both higher precision and lower precision instructions. Using this auto tuner, a programmer does not need to worry about which operations in the kernel can be transformed to a lower precision. This auto-tuner is a profile-guided optimization which ensures the accuracy and performance of the tuned program. For accuracy measurements, we consider the original program as ground truth and compare the tune results to ensure that the error is within the threshold.

GPU thread configuration (grid dimensions and block dimensions) is also a crucial factor impacting the hardware utilization and parallelism of a kernel. So, the performance of a program is highly dependent on the thread configuration. Deciding a thread configuration before executing a kernel requires knowledge about the resources utilized in the kernel. To address the problem of choosing a thread configuration, this auto tuner, being a profile-guided tool, also suggests thread configurations for the kernel. We provide a detailed description of the auto tuner in chapter 7.

6.4 Related work

A vast amount of prior work [14, 17-22, 41] already exists addressing this problem of performing an automated search to tune a program. We will highlight some of the interesting work. Precimonious [14] is a low-level LLVM-based tuning assistant tool that tries to reduce the precision of variables in a program. The input to this tool is a JSON (JavaScript open notation) file which consists of a search space for each variable in the program. JSON is easy to use and it is easy for machines to parse and generate. An extension to precimonious is the tuning that uses blame analysis [41] and shadow value analysis to speed up the tuning by reducing the search space. The profile-driven automated mixed precision (AMP) [19] takes a single precision application as input and profiles the application at operation level. Using this profile, AMP detects ill-behaved operations and promotes them to double precision. Herbie [17] is a tool that performs random sampling alongside arbitrary precision to identify code modifications that will improve accuracy. STOKE [21] is a stochastic optimization tool for × 86 − 64 binaries, and it is capable of producing binaries, which outperform classic compilers. This was done using an MCMC

(Markov chain Monte Carlo) sampling technique. OpenTuner [22], an auto tuning framework represents the tunable parameters in a unique way. Using an ensemble of machine learning search techniques, OpenTuner searches the tunable parameters to provide an optimal combination.

All these works so far have focused towards tuning serial CPU programs, but there has not been any significant development towards highly parallel GPUs. The auto-tuner developed as part of the research dedicated to this thesis can tune a GPU program by doing a target independent trace-based tuning on low-level virtual machine-intermediate representation code (LLVM IR).

6.5 Challenges and features of the proposed auto tuner

It is difficult to tune a GPU program considering the parallelism and multi-threading of GPUs. As mentioned earlier, we proposed (and built) a novel dynamic analysis auto-tuner designed to tune the precision of floating-point GPU programs. Our auto tuner will ensure a better hardware utilization with mixed-precision keeping in mind the performance/accuracy trade-offs studied in previous chapters. Our auto-tuner works at a much finer instruction level rather than tuning the source code variables. Also, this tool does not require any other annotations or inputs except for the target hardware and the error threshold. This makes it easy to tune old legacy programs. With just the error threshold and target hardware, we tune the program to get a better performance and resource utilization. Since our auto-tuner is a profile-guided optimizer, along with tuning, we also have a validation stage. This validation stage is responsible for profiling the tuned program to examine the error and performance of the program.

Following are some of the challenges of an auto-tuner for a GPU Program:

6.5.1 Search space

Unlike many other tools, which do a random search to tune a program, this auto tuner provides a unique data flow based on the program's tuning needs. Data flow based tuning reduces the search space enormously because as we go through the data flow, the risk in accuracy loss also increases. Tuning is achieved with a series of versatile peephole instructions, which allow assessment of possible accuracy loss. If we find a higher risk instruction, we can skip that instruction and go ahead with other instructions in the dataflow. In other words, the data flow inaccuracy risks, which are a part of the process, can now be identified and dealt with naturally and quickly.

6.5.2 Thread configuration

Programmers need not worry about the resources utilized to figure out good thread configuration for a program. The auto-tuner suggests thread configurations for a program by looking at sample-profiled performance counters.

6.5.3 Profiling

Since this auto-tuner is a profile-guided optimization, we profile the tuned program to ensure the transformed program output does not vary more than the threshold. For accuracy measurements, we execute the program with sample inputs, which test the floating-point arithmetic in the program. And, we vary the testing inputs for each program considering the computation involved in the program.

Chapter 7: ARCHITECTURE AND IMPLEMENTATION OF AUTO TUNER

This chapter describes the auto-tuner details. Section 7.1 briefly describes the LLVM compiler infrastructure and Section 7.2 describes the NVCC CUDA compiler. In section 7.3, we provide an example tuning and set the expectations from the auto-tuner. Finally, in section 7.4 we describe the architecture and the implementation details of the auto-tuner.

7.1 LLVM compiler infrastructure

For implementing our auto-tuner we chose the Clang-LLVM based compiler model. Before looking further into the auto-tuner, a brief description of the LLVM compiler infrastructure is necessary [40]. LLVM defines a common, low-level intermediate code representation in static single assignment (SSA) form with several novel features. LLVM IR uses an abstract with the help of the reduced instruction set computer (RISC)-like instruction set for effective analysis. Most of the compiler optimizations were done using LLVM's intermediate representation (IR) code, which is target independent. An example of the LLVM IR is shown in Figure 28 (on the right). Also, LLVM IR uses the infinite register SSA technique which simplifies several optimizations passes.

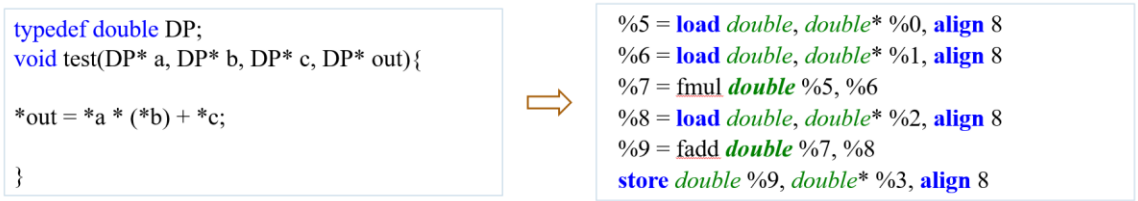

Figure 28. Source code (left) and its equivalent LLVM IR generated (right)

Figure 29 presents the Clang-LLVM based compiler pipeline. Clang is the front-end and the driver for the compiler. Clang supports many low-level languages and transforms all of them to unique LLVM IR representation. Generated LLVM IR now goes through multiple target independent optimization passes (LLVM optimizer) and outputs an optimized IR. This optimized IR is now translated and assembled into machine binary using the backend. Since the pipeline is highly modularized it is easy to develop or support multiple source languages and various hardware architectures.

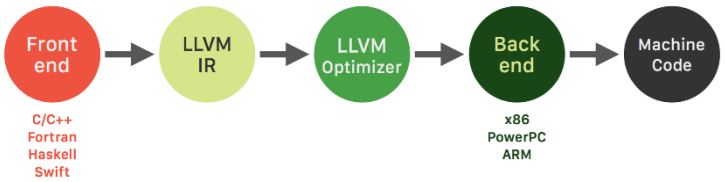

Figure 29. Clang-LLVM Compiler pipeline

7.2 LLVM in NVCC flow

Nvdia's CUDA Compiler (NVCC) [42] is based on the LLVM compiler infrastructure. NVCC compiles the CUDA code into PTX intermediate representation. The graphics driver is now responsible for translating the PTX into a binary code, which can be executed on a GPU. Nvidia open sourced their PTX code generation backend. This PTX code generation is part of the LLVM compiler flow named NVPTX. But the graphics driver is not open sourced. So, the driver is not part of the Clang-LLVM infrastructure. To generate the executable from this PTX we need to use the Nvidia provided tools and the driver.

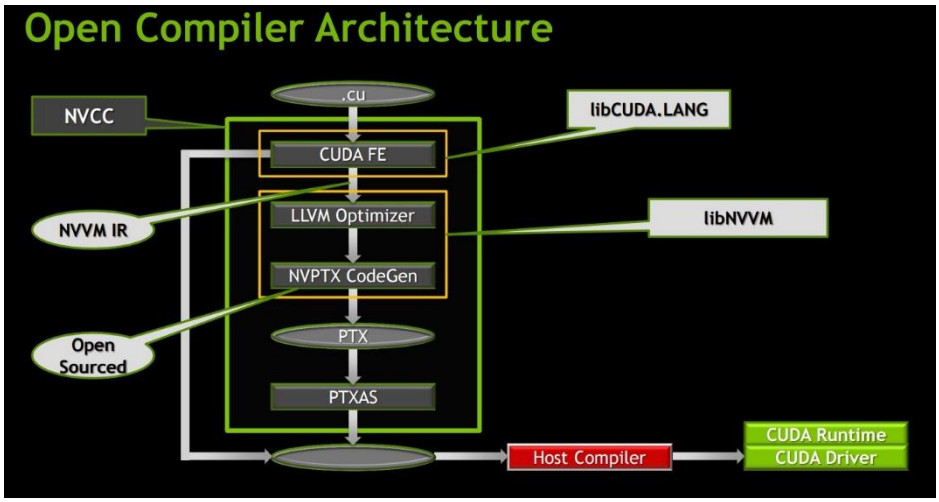

Figure 30. Open compiler architecture for NVidia GPUs

Figure 30 is the open compiler architecture of NVCC which is like the Clang-LLVM architecture. A CUDA file with host functions (CPU functions) and device functions (GPU kernels) are translated into intermediate representation using a CUDA FE at the front-end. The IR now undergoes a target independent optimization. This optimized IR is now transformed to PTX using NVPTX hardware at the backend. PTXAS and the CUDA driver are used to generate a binary which is ready to be executed.

Our auto tuner is an extension to this Clang-LLVM framework with NVPTX backend. In addition to typical pipeline stages we have tuner and validation stages to the pipeline.

7.3 Example tuned program

Consider the program shown in Figure 28: The programmer chose to write this program in double-precision. The operation involved in the program is just a multiply and add ($(a * b) + c$) with the corresponding IR generated as shown on the right. Considering the inputs

to the program, the accuracy still holds even if the multiplication is at a lower precision. This generated IR along with the error threshold is the input to the tuner and output after tuning is shown in Figure 31. For tuning (as mentioned earlier), we searched the computation dataflow to reduce the precision, which resulted in multiplication at single-precision and typecast instructions.

```
%5 = load double, double* %0, align 8
%6 = fptrunc double %5 to float
%7 = load double, double* %1, align 8
%8 = fptrunc double %7 to float
%9 = fmul float %6, %8
%10 = fpext float %9 to double
%11 = load double, double* %2, align 8
%12 = fadd double %11, %10
store double %12, double* %3, align 8
```

Corresponding source code

```
typedef double DP;
void test(DP* a, DP* b, DP* c, DP* out){

*out = float(*a) * float(*b) + *c;

}
```

Figure 31. Expected output from the tuner (left) and corresponding source code (right)

7.4 Auto tuner pipeline and architecture

As mentioned earlier we extended the Clang-LLVM compiler framework to include our auto-tuner. Our auto-tuner consists of two prominent stages: Tuner and the Validation as shown in Figure 32. The tuner stage transforms the instructions considering the data flow, accuracy, and rules or lessons learned from our floating-point analysis. After each iteration, we went through the pipeline to generate the executable. This executable is now profiled for different combinations of sample inputs and thread configurations. From the profiled runs we measure the cost (accuracy and performance) and decide if the current iteration of tuning is successful or not.

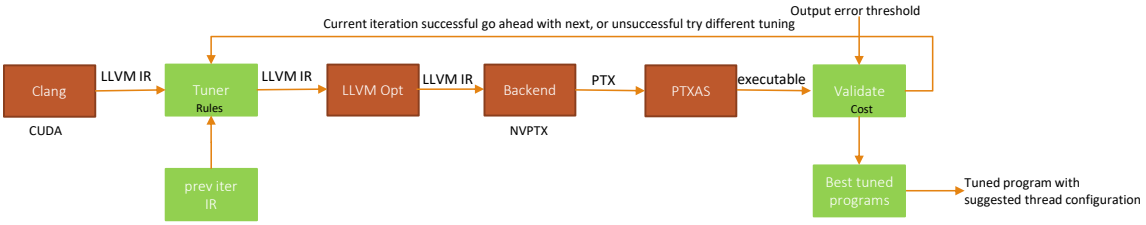

Figure 32. Auto tuner pipeline

The following describes the steps involved in tuning a GPU program:

7.4.1 Original program profiling

Input to Clang is the source code along with inputs required for the auto-tuner, that is, target hardware architecture and error threshold. Transformed IR generated by Clang is the input into the tuner stage. For the first iteration, we profile and measure the outputs from the original program. This is because the original program is the ground truth reference for accuracy and performance measurements for tuning. In the first iteration, the program is not modified, and the code just goes through the LLVM pipeline. In addition to generating the binary, we collect some stats from the program, which are essential for the validation stage.

Usually the programmer uses a CUDA occupancy calculator to estimate the hardware utilization of a program using the resource utilization (number of registers and total shared memory) and the thread configuration. Using the same formulae used in the CUDA occupancy calculator we can figure out thread configurations, which result in a better hardware utilization. We collect a few thread configurations which are optimal for the original program.

The stats collected from the original program in the first iteration are histograms of instruction types in the program. These histograms are used to determine the test inputs to the program. As noted in Chapter 4, every floating-point arithmetic requires different input values to test the limits of the operations. All of these profiling methods and test inputs are based on worst-case scenarios.

For all the combinations of thread configurations and test input values, we measure the performance and save the outputs. These profiled results are now the reference for tuning.

7.4.2 Tuner

Once all the required results for a program have been gathered, the profile is complete, which means we can now start tuning the program from 2nd iteration. The tuner is built as an independent tool. Like the LLVM optimizer tool, the tuner consists of certain passes which already exist in the LLVM Optimizer. These redundant passes included in the tuner are dead code elimination, dead store elimination, dead instruction elimination and combined redundant instructions. These passes are required for the tuner since we opted to run the tuner before the LLVM Optimizer in the pipeline. This is because it is more efficient for the tuner if we can remove unnecessary instructions in the program. We have three passes for tuning a program: 1) the output prune pass, which separates stores and associated codes into separate functions, 2) the precision analysis pass which tunes the program using the data flow, and 3) the merge pass (merge functions) which combines all the separated stores into one function.

7.4.2.1 Output Prune pass

This is a simple pass. We separate the stores and its associated code into separate functions. The implementation of this pass is quite simple and is achieved by keeping a single store and passing it through dead code elimination pass which removes the unnecessary code from the function. We then do the same for all the stores and thereby generate separate functions for each store.

The separation of stores is not an essential pass, but it makes the analysis easier. Figure 33 is a program with multiple loads (nodes in the last level) and single store (root node). There are two approaches to data flow—we either start backward tracing from store to load (the bottom-up approach) or progress forward from load to store (the top-down approach). The bottom-up approach does not require the stores to be separated, but it is intuitively better for a tuner to progress forward rather than tracing backwards. This is because the computation (expression) of a program starts from loads and these initial computations could be at lower precision. But as we progress deeper into the data flow, the risk of accuracy loss increases. For this reason, we implement the top-down data flow approach. Thus, separating the stores and making them independent facilitates the analysis making it easier to complete. There could be scenarios where multiple stores share the same computations (subset of a dataflow) when this is true the functions per store will have the redundant computations. Independent tuning of these store functions may result in different precision tuning. After tuning when we merge the functions with separately tuned subset. Since they are of different precision some of compute will stay separate, a possible optimization here to ensure only one of the subset (higher precision) is in the final program.

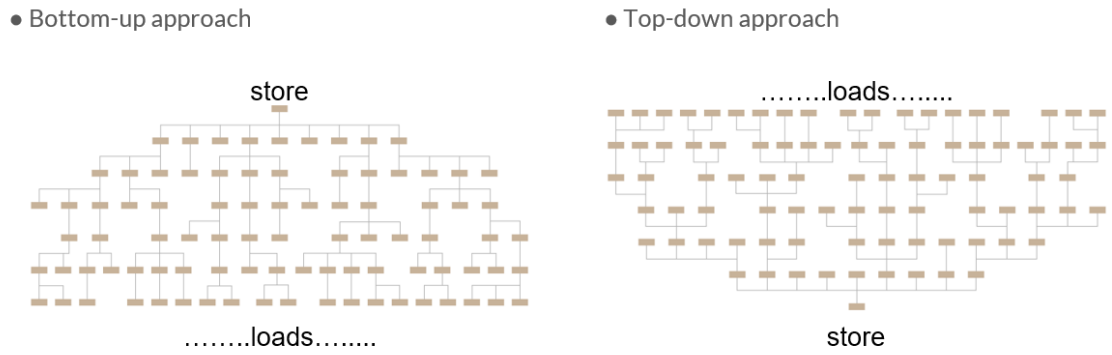

Figure 33. Two approaches for dataflow analysis

7.4.2.2 Precision analysis pass

As mentioned in the previous section, the data flow top-down analysis starts with the loads and ends at the store. To implement the top-down approach, use-def chains are needed. A simple example of use-def chain is shown in Figure 34. A complete data flow can be traversed by recursively making references to the uses and definitions; thus, the `use_iterator()` was chosen to implement the use-def chains.

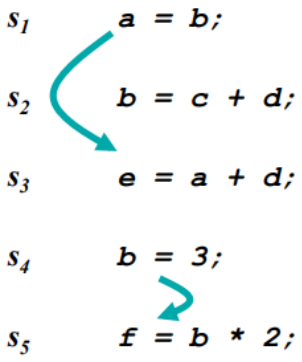

Figure 34. Example of a use-def chain

Figure 35 is a computation data flow graph with the top node being the load with the store at the bottom of the graph. All computations are in double precision. The goal of the auto-tuner is to reduce the double-precision computations to single precision. Assume this kernel has a single load from memory and the loaded data goes through the steps until finally, the output from the computation is stored to memory. The dotted lines in the graph represent the level at which we tune the program.

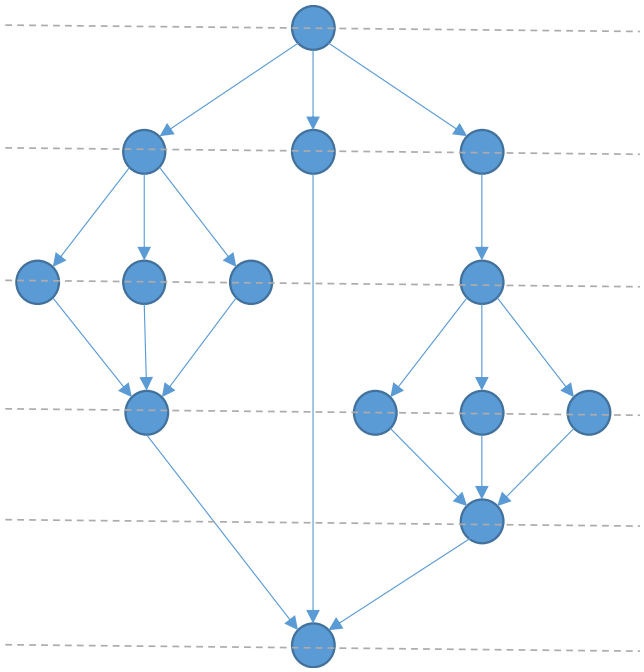

Figure 35. Data flow of a double precision kernel from load to store

Figure 36 is the data flow graph after the first iteration of tuning. Since we do not modify the memory operations, we insert the typecast instruction (double- to single- precision) after the first load. The new registers introduced due to tuning are at an offset of 512 registers so they do not overlap with the existing registers. For example, the first typecast

instruction has a destination register number of 512. After the typecast, the loaded data is now in single precision and all its uses need to be single-precision. For this we transform all the uses of this register from double-precision to single precision instruction. In case of binary operations, we must typecast the other source operands as well. Finally, we typecast these single-precision outputs back to double-precision as shown in Figure 37.

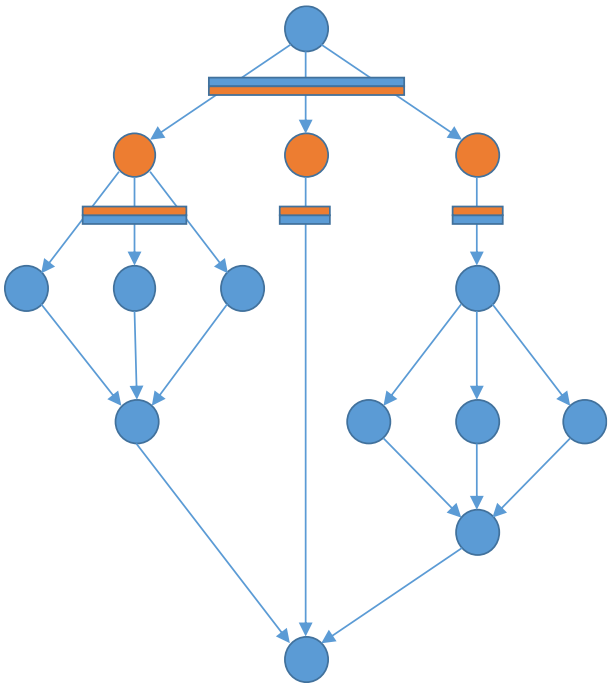

Figure 36. Data flow of the program after first iteration of tuning (blue represents the double precision ops and orange is the single precision op)

Figure 37 is the output of the second iteration of tuning. After a successful first iteration tuning, the validation stage goes back to the tuner stage with the profile status. Along with the status, the validation stage also provides the maximum output error to tuner for all the test inputs. Using the error and the lessons (rules) learned in our floating-point analysis, we determine if the next arithmetic instruction in the data flow is safe to tune. For some

operations (instructions) a small error in input could result in a higher output variation. For example, instructions like exponent and logarithm result in an exponential error if input has high error. Considering all these rules, we tune the program again as shown in Figure 37. At this stage we can either choose to introduce new typecast instructions or just move the typecast as shown here. Both approaches will result in the same output because the LLVM optimizer will remove the redundant instructions.

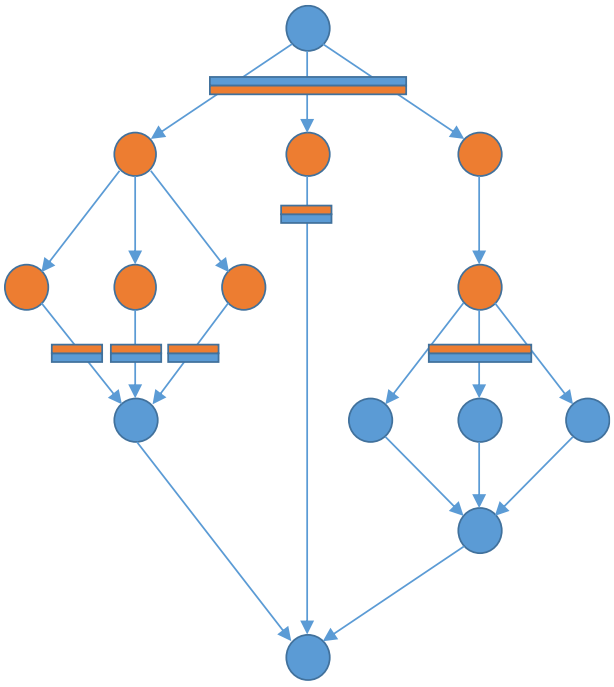

Figure 37. Dataflow of the tuned program after second iteration

7.4.2.3 Techniques to tune faster

As described in the previous section, at every iteration we tune a level (depth) of computation in the graph. This might be a slower approach even though we tune all the

instructions (breadth) in one level. Following are some techniques to speed up the tuning and reduce the number of iterations.

7.4.2.3.1 Step Size

Step size iteration is inspired from the stochastic gradient descent optimization algorithm in machine learning. Instead of doing one level at each iteration, we designed a step size where we group tuned multiple levels together. For example, initially we set this step size to 8, which means in each iteration we tuned eight computation levels of graphs together. Using this approach, we could reduce the total number of iterations significantly. But this is possible only if the tuned program passes validation. In the case of an unsuccessful tuning, we correct the problem by loading the previous successful IR and then reducing the step size to either 4, 2, or 1. Using the reduced step size we can then fine tune the program and validate the program.

7.4.2.3.2 Cut points

Figure 38, is the basic code structure of LLVM IR. Module represents a source file and everything else is contained in module. Modules contain functions, functions contain basic blocks and basic blocks contain instructions. A basic block keeps track of inputs (live-in) and outputs (live-out) registers out of the basic block. We use this concept to speed up the tuning across basic blocks in parallel. In a basic block we introduce cut points forking a single basic block into multiple basic blocks. Using these live-in and live-out of each basic block as load and store we tune these basic blocks in the same iteration. This effectively reduces the number of iterations required to tune the program. In case of an unsuccessful tuning we fallback to the previous IR and do one basic block in each iteration.

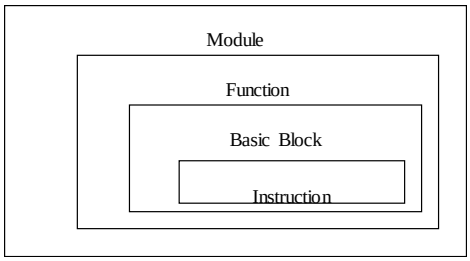

Figure 38. LLVM IR code structure

7.4.3 Validation Stage

Until now, we have discussed how to tune a program to a lower precision. Now we look at the other important stage in the auto-tuner, validation. Briefly, the tuned executable is validated against the original program. The outcome of the validation is the input to the tuner for further iterations. As previously determined, the original program is run for all the possible combinations of thread configurations and different input values.

A program with these parameters (inputs and thread configurations) incorporated into the program would need to be recompiled to generate the executable for a different combination. To avoid recompiling the program for each test set combination, we consider a program with thread configuration as command line arguments to the program. Furthermore, input variables are loaded from files which are again command line arguments.

To evaluate the tuned program, we define a cost function which consists of determining characteristics of a program. In this case we have the correctness term (accuracy) and performance term (execution time). Correctness is how accurate the tuned program is when compared to the original program result. We measure the accuracy by taking the maximum

of absolute error between output of the tuned program and the original program. Performance is the inverse of speed gained when compared to the original program. Cost of the original program would be 1 because correctness is zero and performance is 1.

$$cost = correctness + performance$$

Along with the executable, we also provide the error threshold in ULP (units in last place). For a program to be successful the maximum absolute error of a tuned program must be within the defined threshold. At a certain iteration if the error crosses the threshold, then the current iteration is unsuccessful, and we provide this to the tuner. The tuner now takes the previous successfully tuned IR and goes ahead with the tuning by skipping the instructions which caused a higher error.

7.4.3.1 Best tuned programs

After every successful tuning iteration, considering the cost, we stored the executable and the configuration used for the program. We had a limited number of executables that could be stored. In case of an overflow, we replace the highest cost program (slower or high error) if we the current cost is smaller. Maintaining many such tuned programs is important to finally choose the best performing program in terms of both accuracy and execution time. After tuning the entire program, we chose the best from the stored programs as the final executable along with its thread configuration.

7.5 Function calls

A program may consist of library functions like exponent, sin, and cos used in the computation or it could be user defined functions. LLVM IR represents both the library

functions and user defined functions as the same. User defined functions, which are defined in the program, need to be tuned when encountered. However, for library functions, since we do not have their definitions, we cannot tune them. Instead we replace them with respective single-precision library functions. To differentiate between defined functions and declared functions, we used the LLVM `isDeclaration()` function. This returns true if the program is defined in the current module—otherwise, it is false. For the functions defined in the module we traverse the functions recursively for tuning.

Chapter 8: CONCLUSIONS AND FUTURE WORK

In this work, we have explored the use of different floating-point arithmetic precisions on GPU architectures from both the accuracy and performance angles. Our study has shown that the use of float2 arithmetic operations is not beneficial on Fermi and later GPUs, since it underperforms double-precision arithmetic implementation, both in terms of accuracy and execution time. For additions, double2 arithmetic is a good compromise between double- and multiple-precision arithmetic operations, both in terms of accuracy and performance. However, double2 is generally as accurate as double-precision arithmetic on multiplications and exponential functions. The performance disadvantage of multiple-precision arithmetic operations is accentuated when the degree of multithreading is high. For division, double2 arithmetic is less accurate compared to double-precision arithmetic implementation, due to incomplete double2 division algorithm.

Using these insights, we designed an auto tuner that helps in the automatic selection of the precision of each floating-point variable present in a GPU program by analyzing not only the operations performed on that variable, but also the data flows of the application, its arithmetic intensity, and its use of the available functional units for a given degree of multithreading. The data flow approach of searching the instructions to lower the precision reduces the search space significantly. In addition to the vanilla data flow approach, we also propose multiple techniques to speed up the search.

With this profile-guided auto tuner, a GPU program is tuned to improve hardware utilization and performance. In addition to tuning a program, the auto tuner can validate

the tuned program to ensure that the error is within the threshold. The auto tuner can also provide better functional unit utilization, and suggest thread configuration for the program.

8.1 Future work

The novel designed auto tuner can efficiently tune a directed acyclic data flow program. In the future, we plan to extend this to support cyclic computations (loops). To support this, the accuracy and performance measurements must be profiled at runtime. For static loops, the compiler can unroll all the loops to convert a loop into an acyclic data flow. But dynamic loops, which are input dependent, need runtime tuning. For dynamic loops, we plan to build a runtime system (a just in-time compilation model) which can analyze the data in the loops and tune accordingly at run time.

In addition to supporting dynamic loops, we plan to add integer precision along with floating-point precision to the search space. Integer precision arithmetic requires fewer gates than floating-point arithmetic; moreover, lower precision integer (8 bit-int) can save registers.

References

[1] D. H. Bailey and J. M. Borwein, "High-Precision Computation and Mathematical Physics," *XII Advanced Computing and Analysis Techniques in Physics Research*, 2008.

[2] P. H. Hauschildt and E. Baron, "Numerical solution of the expanding stellar atmosphere problem," *Journal of Computational and Applied Mathematics*, vol. 109, no. 1, pp. 41-63, 1999/09/30/ 1999.

[3] D. H. Bailey and D. J. Broadhurst, "Parallel integer relation detection: techniques and applications," *Mathematics of Computation*, vol. 70, pp. 1719-1736, n/a 1, 2001 2001.

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

[5] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," in "ACM Computing Surveys," 1991, vol. 23.

[6] D. H. Bailey, "High-precision Floating-point Arithmetic in Scientific Computing," presented at the IEEE Computing in Science and Engineering, 2005.

[7] D. M. Priest, "Algorithms for arbitrary precision floating point arithmetic," in *[1991] Proceedings 10th IEEE Symposium on Computer Arithmetic*, 1991, pp. 132-143.

[8] J. D. a. Y. Hida, "Accurate and efficient floating point summation," in *SIAM J. Sci. Comput*, 2003, vol. 25, pp. 1214-1248.

[9] J. Demmel and H. D. Nguyen, "Fast Reproducible Floating-Point Summation," in *2013 IEEE 21st Symposium on Computer Arithmetic*, 2013, pp. 163-172.

[10] M. M. P. Langlois, 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.

[11] B. M. V.Y. Pan, G. Qian, and R.E. Rosholt, "A new error-free floating-point summation algorithm," in *Computers & Mathematics with Applications*, vol. 57, February 2009, pp. 560–564.

[12] J. Demmel, P. Ahrens, and H. D. Nguyen, "Efficient Reproducible Floating Point Summation and BLAS," EECS Department, University of California, BerkeleyUCB/EECS-2016-121, June 18 2016, Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-121.html.

[13] NVidia. *GPU-Accelerated Applications for HPC Industries*. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-121.html

[14] C. N. C. Rubio-Gonzalez, H. D. Nguyen, J. Demmel,´W. Kahan, K. Sen, D.H. Bailey, C. Iancu, and D. Hough, "Precimonious: Tuning Assistant for Floating-point Precision," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*, 2013.

[15] F. d. Dinechin, C. Lauter, and G. Melquiond, "Certifying the Floating-Point Implementation of an Elementary Function Using Gappa," *IEEE Transactions on Computers*, vol. 60, no. 2, pp. 242-253, 2011.

[16] E. Darulova and V. Kuncak, "On Sound Compilation of Reals," *ArXiv e-prints*, vol. 1309, Accessed on: September 1, 2013Available: http://adsabs.harvard.edu/abs/2013arXiv1309.2511D

[17] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock, "Automatically improving accuracy for floating point expressions," presented at the Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, 2015.

[18] W.-F. Chiang *et al.*, "Rigorous floating-point mixed-precision tuning," *SIGPLAN Not.*, vol. 52, no. 1, pp. 300-315, 2017.

[19] R. Nathan, H. Naeimi, D. J. Sorin, and X. Sun, "Profile-Driven Automated Mixed Precision," *ArXiv e-prints*, vol. 1606, Accessed on: June 1, 2016Available: http://adsabs.harvard.edu/abs/2016arXiv160600251N

[20] M. O. Lam and B. L. Rountree, "Floating-point shadow value analysis," presented at the Proceedings of the 5th Workshop on Extreme-Scale Programming Tools, Salt Lake City, Utah, 2016.

[21] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic optimization of floating-point programs with tunable precision," presented at the Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, Edinburgh, United Kingdom, 2014.

[22] J. Ansel *et al.*, "OpenTuner: an extensible framework for program autotuning," presented at the Proceedings of the 23rd international conference on Parallel architectures and compilation, Edmonton, AB, Canada, 2014.

[23] M. Lu, B. He, and Q. Luo, "Supporting extended precision on graphics processors," presented at the Proceedings of the Sixth International Workshop on Data Management on New Hardware, Indianapolis, Indiana, 2010.

[24] T. N. a. D. Takahashi, "Implementation of Multiple-Precision Floating-Point Arithmetic Library for GPU Computing," presented at the IASTED International Conference on Parallel and Distributed Computing and Systems, 2011.

[25] J.-M. M. Mioara Joldes, Valentina Popescu, and Warwick Tucker, "CAMPARY: Cuda Multiple Precision Arithmetic Library and Applications," presented at the 5th International Congress on Mathematical Software (ICMS), Berlin, Germany, Jul 2016.

[26] NVidia. *TESLA™ C2050 / C2070 GPU Computing Processor*. Available: http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C2050_C2070_jul10_lores.pdf

[27] NVidia. *Pascal GPU Compute Architecture*. Available: https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf

[28] Microway. *Comparison of NVIDIA Tesla/Quadro and NVIDIA GeForce GPUs*. Available: https://www.microway.com/knowledge-center-articles/comparison-of-nvidia-geforce-gpus-and-nvidia-tesla-gpus/

[29] M. O. Lam, J. K. Hollingsworth, B. R. d. Supinski, and M. P. Legendre, "Automatically adapting programs for mixed-precision floating-point computation," presented at the Proceedings of the 27th international ACM conference on International conference on supercomputing, Eugene, Oregon, USA, 2013.

[30] A. Thall, "Extended-Precision Floating-Point Numbers of GPU Computation," presented at the ACM SIGGRAPH 2006 Research posters, August 2006.

[31] *IEEE Standard for Floating-Point Arithmetic*, Aug 2008.

[32] G. H. L. Fousse, V. Lef`evre, P. P´elissier and P. Zimmermann, "MPFR: A multiple-precision binary floating-point library with correct rounding," *ACM Transactions on Mathematical Software 33*, 2007, Art. no. 13.

[33] Y. Hida, X. S. Li, and D. H. Bailey, "Algorithms for quad-double precision floating point arithmetic," in *Proceedings 15th IEEE Symposium on Computer Arithmetic. ARITH-15 2001*, 2001, pp. 155-162.

[34] Y. H. David H. Bailey, Xiaoye S. Li and Brandon Thompson, "ARPREC: An Arbitrary Precision Computation Package," ed, 2002.

[35] *The GNU Multiple Precision Arithmetic Library*. Available: https://gmplib.org/

[36] T. J. Dekker, "A Floating-point technique for extending the available precision," in *Numerische Mathematik 18*, 1971, pp. 224–242.

[37] H. Wong, M. M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU microarchitecture through microbenchmarking," in *2010*

IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS), 2010, pp. 235-246.

[38] NVidia. *Floating Point and IEEE 754 Compliance for NVIDIA GPUs*. Available: http://docs.nvidia.com/cuda/floating-point/index.html

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

[40] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," presented at the Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, Palo Alto, California, 2004.

[41] C. Rubio-González *et al.*, "Floating-Point Precision Tuning Using Blame Analysis," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 1074-1085.

[42] "Compiling CUDA and other languages for GPUs," 2012.