ABSTRACT

GU, RUIDONG. Mixed-precision Auto-tuning for Floating-point Applications on CPU and GPU. (Under the direction of Dr. Michela Becchi).

Many scientific applications (e.g., molecular dynamics, climate modeling and astrophysical simulations) rely on floating-point arithmetic. Due to its approximate nature, the use of floating-point arithmetic can lead to inaccuracy and reproducibility issues, which can be particularly significant for long running applications. Indeed, previous work has shown that 64-bit IEEE floating-point arithmetic can be insufficient for many algorithms and applications, such as ill-conditioned linear systems, large summations, long-time or large-scale physical simulations, and experimental mathematics applications. To overcome these issues, existing work has proposed high-precision floating-point libraries (e.g., the GNU multiple precision arithmetic library), but these libraries come at the cost of significant execution time. Over the past decade, GPUs have become a commodity and played an important role in the acceleration of large-scale computations. To extend the accuracy of the computation beyond that allowed by standard IEEE floating-point arithmetic (single, double and long double precision), some high-precision floating-point libraries (e.g., CAMPARY library and QD library) have been made available also on GPU. However, due to the higher resource requirements and data transfer cost of high precision data types, the performance cost of these libraries is even more pronounced on GPU than on CPU. For example, when using 128-bit precision arithmetic, LavaMD, an open-source molecular dynamic application, runs 27x and 192x slower than when using double and single precision arithmetic, respectively. This big performance gap can make it infeasible for some large-scale applications to run with high precision.

In this dissertation, we analyze the performance and accuracy tradeoffs of floating-point applications on CPUs and modern Nvidia GPUs, and we propose two mixed-precision auto-tuners,

one for CPU and one for GPU, for applications requiring high-precision floating-point arithmetic to deliver a prescribed level of accuracy. Our auto-tuners use compiler analysis to discriminate operations and variables that require high-precision from those that can be handled using standard IEEE floating-point arithmetic, and they generate a mixed-precision program that trades off performance and accuracy by selectively using different precisions for different variables and operations. In particular, our auto-tuners leverage loop and data dependencies analysis to quickly identify precision-sensitive variables and operations and provide results that are robust to different input datasets. To better serve the GPU architecture and reduce the search space further, our GPU auto-tuner takes also GPU-specific considerations into account. We test our auto-tuners on a mix of applications with different computational patterns on both CPU and GPU platforms.

Mixed-precision Auto-tuning for Floating-point Applications on CPU and GPU

by
Ruidong Gu

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Computer Engineering

Raleigh, North Carolina
2020

APPROVED BY:

Michela Becchi Committee Chair	Xipen Shen
James Tuck	Huiyang Zhou

DEDICATION

This dissertation is dedicated to the memory of my grandfather Shuangniu Gu. Although he was my inspiration to pursue my doctoral degree, he was unable to see my graduation.

I express my gratitude to my academic adviser Dr. Michela Becchi who has always encouraged me, guided me and kept me on track during my research life.

This dissertation is also dedicated to my family. To my parents Zhong Gu and Meizhen Jia and my elder sisters Ruijuan Gu and Lin He who have always supported and encouraged me in my life. To my wife Zheng Zhao and my daughter Evelyn L. Gu who always love me and have encouraged me to pursue my dream and finish my dissertation.

BIOGRAPHY

Ruidong Gu, was born on October 25, 1990 in ShanXi, China. He attended the Beijing Jiaotong University and graduated in 2012 with a bachelor's degree in communication engineering.

Ruidong participated in an exchange program and performed part of his undergraduate studies at University of Missouri. He was then accepted into University of Missouri's master's program in electrical engineering, and he worked on his M.S. thesis under the supervision of Dr. Michela Becchi. In 2016 he obtained his master's degree from University of Missouri.

In 2017, Ruidong undertook doctoral studies in computer engineering at North Carolina State University under Dr. Michela Becchi's guidance. He earned his doctoral degree in computer engineering in 2020.

During his graduate studies, Ruidong worked as a summer intern at Samsung SARC in Austin, TX, and was offered a position in their GPU software team.

ACKNOWLEDGMENTS

First, I would like to express my gratitude to my advisor Dr. Michela Becchi for guiding and supporting me through my entire research life. As the mentor of my doctoral research, Michela has always shown an excellent competence in various engineering domains and a critical analysis that have guided my research and kept me on track. As a collaborator, Michela has also given out her kind suggestions and advices not only on the challenges I met in my research but also on the difficulties I encountered in my life. I greatly appreciate Michela's energy, rigorous scientific research and ability to solve practical problems with a broader view and a solid research background in the engineering area.

I would like to thank all my committee members – Dr. James Tuck, Dr. Xipeng Shen and Dr. Huiyang Zhou – for their kind and valuable feedback and suggestions. They have really helped me to refine my work and better achieve our research goal. I also would like to express my gratitude to Dr. Michela Taufer and her research group for the kind feedback on my research work.

I would like to express my gratitude to my research collaborators Kittisak Sajjapongse, Sruthikesh Surineni and Paul Beata. I will always remember how they shared their valuable academic experience and encouraged me when I first stepped into the research life. More importantly, as collaborators, they taught me how to communicate and cooperate efficiently.

I have been lucky to perform a summer internship at Samsung SARC, where I had my first industry experience directly connected to my research and worked with great people. Especially, I want to express my gratitude to Raun Krisch and Simon Water who always answered my questions with patience and helped me define clear milestones during my internship. Moreover, I would like to thank Robert Metzger for providing suggestions on my future directions with his industry experience.

My deepest gratitude goes to my family who always supports, encourages and loves me in my whole life. To my wife Zheng Zhao, because she always encourages and supports me both in life and work. She also helps me to get out of the hard times as a friend. To my daughter Evelyn, because she lets me know the responsibility to be a father and I feel lucky to have such a lovely daughter. To my elder sister Ruijuan Gu, because she is the person who first gave me the motivation to work on engineering. To my parent Meizhen Jia and Zhong Gu, because they always show me with their love and patience, and give me endless courage to pursue my dream.

I would like also to thank all my friends Hancheng Wu, Yanan Li, Jiayuan Chen, Fei Zhang, Zheng Cui, Hang Xu, Zhenghao Yang and Jun Yan. They have shown me how important friendship is. They have made me enjoy life and it is good to have a friend to chat anytime.

Finally, I would like to thank all my lab mates Hancheng, Marziyeh, John, Mostafa, Tri and all other collaborators. I am extremely lucky to have had such a great time in the NPS lab.

Thank you!

Ruidong

North Carolina State University

November 2020

This work was supported through National Science Foundation's award CCF-1728850.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

Many high-performance computing applications from various scientific domains, such as molecular dynamics [1], climate modeling [2], astrophysical simulations [3], and experimental mathematics [4], use floating-point arithmetic. With the prevalence of parallel computing on GPUs, more and more scientific applications [5] relying on floating-point arithmetic have been accelerated using modern GPUs. Floating-point arithmetic is by definition approximate since real numbers are not guaranteed to be represented exactly by a finite number of bits. The approximate nature of floating-point numbers can lead to inaccuracy and reproducibility issues, which can be especially problematic for long-running applications. Indeed, previous work [4] has shown that 64-bit IEEE floating-point arithmetic can be insufficient for many computations and applications, such as ill-conditioned linear systems, large summations, long-time or large-scale physical simulations, and experimental mathematics.

Several high-precision libraries encoding real numbers using more than 64 bits are available [6-9]. While the need for high accuracy can be met through the use of these libraries, they can result in a significant cost in execution time. For example, on an Intel Xeon E5 processor the execution time of the Computational Fluid Dynamics (CFD) program from the Rodinia benchmark suite [10] on a medium size dataset increases from 270 seconds when using double precision to 355.3 minutes when using 256-bit GMP (a popular high- precision library), a slowdown of about 80x. A performance drop of this magnitude can make the execution time of large-scale scientific simulations prohibitive. To overcome this issue, previous work has started to explore mixed-precision, i.e., the combined use of different precisions in a floating-point program, as a means to trade off performance and accuracy.

Previous efforts [11-16] on CPU platforms have proposed tuning assistants to either accelerate a floating-point program under a given accuracy constraint [11-13, 16], or reduce the precision of its variables without sacrificing accuracy and performance [15]. In order to do so, these tools have explored the search space resulting from assigning different precisions to the floating-point variables within a given program. For each variables' configuration, these tools first modify the program either by source code transformation or by binary modification, and then they run the modified program with a given input and verify that the results are within a prescribed error bound. While all these methods use heuristics to prune the search space, they still require many runs of the program. This can even result, on a single input, in hours of tuning for programs with execution times on the order of few seconds [11]. Except for [11], which also considers long double variables, these efforts focus only on float and double data types.

This problem is aggravated on GPU, where the performance gap between standard and high precision arithmetic is even more pronounced due to the pressure put by high precision libraries on the hardware resources, including floating-point units, registers, shared memory capacity, and memory bandwidth. For example, when running with 128-bit precision, the Computational Fluid Dynamic (CFD) program from Rodinia Benchmark Suite runs 27x slower than when using double precision on a high-end Nvidia GPU while its time cost for 128-bit high-precision is 15x that of double precision on an Intel Xeon CPU platform. Existing proposals have focused on single, double and long double data types (i.e., 32-, 64- and 80-bit precision, respectively). While these tools have been designed to target CPU, some of them [17-19] have also been showcased on GPU. However, the proposed tuning strategies are generic and not tailored to GPU. Considering GPU-specific issues allows narrowing down the search space, which is essential when targeting high precision arithmetic.

In this dissertation, we aim to design mixed-precision (high-precision and standard precision) autotuning tools for floating-point applications running on CPU and GPU platforms. While doing so, we aim to leverage compiler analysis to provide a feasible search space. First, we design a tuning strategy that leverages the analysis of loops and computational patterns prone to error propagation (e.g., global summation) to provide a reasonable exploration space and tuning results robust to input variability. Second, we incorporate that tuning strategy in two floating-point auto-tuners – one for CPU and one for GPU. Third, we identify GPU-specific considerations that can drive the search space on GPU, and we incorporate them in our GPU auto-tuner implementation. Finally, we show that the combined use of standard and high precision arithmetic is effective in balancing accuracy and performance for applications requiring high precision.

1.1 Contributions

In this dissertation, we propose a data independent mixed-precision auto-tuning design for floating-point programs running on CPU and GPU platforms. We leverage compiler analysis to identify loops and computational patterns that are prone to error propagation and have a pronounced effect on performance and accuracy. Our main contributions are as follows:

First, we study the GPU support available on modern GPUs and provide insights on performance/accuracy tradeoffs originating from the use of different arithmetic precisions in GPU applications (Chapter 2).

Second, we design a tuning strategy to convert floating-point applications into mixed-precision programs and we implement it in a CPU auto-tuner based on the LLVM compiler toolchain. Since we address the use of high-precision libraries with costly operations, our tuning strategy aims to reduce the tuning time and the input sensitivity, and it does so by leveraging compiler analysis to identify basic blocks more likely to affect performance and accuracy (Chapter 3).

Third, we extend our tuning strategy with GPU-specific considerations to better serve the GPU architecture and further reduce the search space. We incorporate this extended tuning strategy into a GPU auto-tuner implemented on top of the Rose compiler infrastructure and using python-based scripts (Chapter 4).

Finally, we summarize our mixed-precision design and the potential benefits of extending our mixed-precision techniques to various platforms and application domains (Chapter 5).

1.2 Methodology

1.2.1 Benchmarking and characterization work

Our first goal is to study the use of floating-point arithmetic on modern GPUs and provide insights into the involved performance/accuracy trade-offs. This helps us find the opportunity to deploy mixed-precision techniques on GPU platform and find out the challenges involved in using mixed-precision on GPU. We consider various floating-point precisions: single and double precision in IEEE 754 standard, GNU Multiple Precision (GMP), and composite precision either based on single or on double precision numbers (float2 and double2, respectively) [1]. Our study covers four generations of Nvidia GPUs: Fermi, Kepler, Maxwell and Pascal devices. Our study is structured as follows. First, we analyze the support for 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, we study the performance/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, with the exception of division, we use microbenchmark applications that perform only one of these operations. In addition, we extend the composite-precision library to support the natural exponential function, and we propose and analyze two implementations of this operation. Third, we analyze the combined use of different arithmetic operations in three benchmark applications characterized by different instruction mixes and arithmetic intensities, namely: Gaussian Elimination, LUD Decomposition and LavaMD (a molecular dynamic simulation application).

1.2.2 Mixed precision auto-tuning on CPU

Our benchmarking and characterization work on GPU platform suggests the potential of using mixed-precision tuning on GPU. Our first step is to design a CPU auto-tuner for applications that require high precision in order to satisfy accuracy constraints.

In our CPU auto-tuner design, we aim to evaluate the feasibility of balancing accuracy and performance of applications that require high-precision arithmetic by providing mixed-precision versions of these applications. To limit the tuning time and the sensitivity of the tuning results to the inputs, we design a tuning method that leverages compiler analysis. In particular, we observe that frequently executed loops tend to affect not only performance, but also accuracy, especially if they implement repeated summations and multiplications and include a large number of data-dependent floating-point operations. Based on this observation, we design a loop-aware tuning strategy, and we use it to analyze the effect of tuning down subsets of operations within basic blocks incrementally based on compiler analysis. Our tuning strategy uses the following criteria: (a) the number of occurrences of a particular variable or operation throughout the code, (b) the presence of the variable or operation in accumulation-like computation patterns (e.g., global summations or sequences of multiplications), and (c) the execution frequency of the basic block containing that operation. We propose a two-step approach. First, we use compiler analysis to generate a tuning plan specific to the program but largely independent of its input data. The tuning plan consists of a sequence of iterations, each specifying a set of operations to be tuned down from higher to lower precision. Second, we execute the tuning plan by modifying the program and running it with a given input and a prescribed error bound. Our proposed tuning method allows a good accuracy-performance tradeoff, providing a good alternative to more traditional, computationally expensive approaches such as GMP to provide high accuracy.

1.2.3 Mixed precision auto-tuning on GPU

The last phase of this dissertation is the design of a mixed-precision auto-tuner for floating-point applications running on GPU. We build on our previous work [20], where we proposed a mixed-precision CPU autotuner for floating-point applications requiring high precision. Specifically, we still focus on computational patterns that are prone to error propagation, such as loops and accumulation patterns. However, here we extend our previous work with GPU-specific considerations to better target CUDA code. From a methodological standpoint, we depart from our previous work and perform variables-based rather than operations-based tuning. This has two advantages: first, it simplifies the problem of limiting data type conversion overhead; second, it allows incorporating considerations on the location of the variables and the memory transfers into the tuning policy. From an implementation point of view, rather than developing a proof-of-concept based on LLVM, we have implemented a fully functional, automated pipeline that leverages the ROSE compiler to simplify code analysis and operate at the level of the source code. Rather than generating binary code, our GPU autotuner generates source code that can be further optimized manually by the programmer.

CHAPTER 2

Performance-accuracy tradeoffs of floating-point arithmetic on GPUs

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 [21], Nvidia lists about 200 GPU-accelerated applications from different domains, including computational chemistry, biology, physics, numerical analytics, weather prediction, computational finance and data mining. Despite the body of work on reproducibility problems of floating-point arithmetic on CPU [11, 16, 22-31], the study of accuracy-performance tradeoffs related to the use of floating-point arithmetic on GPU has received very limited considerations. Several efforts have proposed multiple-precision floating-point libraries for GPU [9, 32, 33]. However, these libraries provide high accuracy at a significant runtime cost. Therefore, if used unnecessarily, they can lead to highly inefficient codes.

Our goal is to study the use of floating-point arithmetic on modern GPUs and provide insights into the involved performance/accuracy tradeoffs. We consider various floating-point precisions: single and double precision in IEEE 754 standard, GNU Multiple Precision (GMP), and composite precision either based on single or on double precision numbers (float2 and double2, respectively) [1]. We focus on four generations of Nvidia GPUs: Fermi, Kepler, Maxwell and Pascal devices. Our study is structured as follows. First, we analyze the support for 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, we study the performance/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, with the exception of division, we use microbenchmark applications that perform only one of these operations. In addition, we extend the composite-precision library to support the natural exponential function, and we propose and analyze two implementations of this operation. Third, we analyze the combined use of different arithmetic operations in three benchmark applications characterized by different instruction mixes and arithmetic intensities, namely: Gaussian Elimination, LUD Decomposition and LavaMD (a molecular dynamic simulation application).

Our study leads to the following findings. First, the use of float2 arithmetic is not beneficial on Fermi and later GPUs, since it underperforms double-precision arithmetic both in terms of accuracy and execution time. Thus, 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 both in terms of accuracy and performance. However, double2 is generally as accurate as double-precision arithmetic on multiplications and exponential functions, and less so 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 additions and subtractions. In the presence of a set of additions followed by a division, it might be beneficial to perform a double2-to-double conversion before the division. Third, on GPU, the performance disadvantage of multiple-precision arithmetic is aggravated by the fact that the high register utilization of its operations 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 Taylor series (as proposed by Thall [34]).

2.1 Background

2.1.1 IEEE 754 Standard for Floating-point Arithmetic

Floating-point numbers are approximations of real numbers. The IEEE 754 standard [35] 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:

$$(-1)^{sign} \times 1.mantissa \times 2^{(exponent\text{-}BIAS)}$$

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 2.1 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).

Table 2.1: IEEE Floating-point standard

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

2.1.2 Arbitrary-Precision Arithmetic

Arbitrary-precision libraries [7, 8, 36] 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 [7] on CPU and the CUDA Multiple Precision Arithmetic (CUMP) [33] library on GPU, both based on the GNU Multiple Precision library (GMP) [37]. GMP is a free open-source library for arbitrary-precision

`[hx, tx] split (float x){`
`   c=float(2t+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 2.1: Dekker's error-free split of floating-point numbers into two comonents (left); Taufer's CMP float2 data structure (right).

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.

2.1.3 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 former while providing substantially better performance than the latter. The foundation of composite-precision arithmetic is in the error-free method proposed by Dekker [38] to split a floating-point number into two half-length floating-point values. This method is illustrated in the pseudo-code in Figure 2.1 (to the left).

```
//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 2.2: Taufer's CMP addition, multiplication and division.

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 [34]. More recently, Taufer et al. [1] 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 1 (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 2.2. As can be seen, CMP addition/subtraction require eight floating-point additions and 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 paper, we extend the CMP library to support the exponential function operation, and we present an analysis of the performance/accuracy tradeoffs involved with the use of standard, composite and arbitrary-precision arithmetic on GPU.

2.1.4 Discussion on Related Work for GPU

In recent years there have been several efforts aimed to provide verification [29], compiler techniques [30, 31] and tuning assistants [11, 16] to verify or improve the accuracy of CPU programs containing floating-point arithmetic. Gappa [29] 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 [30] 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 [31] proposes using a set of code rewrite rules (for example, replacing arithmetic expressions with polinomial 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. Precimonious [11] 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. Precimonious requires the programmer to specify the ranges of the inputs. FPTuner [16] 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 two 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. 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.

2.2 Support of Floating-point Arithmetic on GPU

Table 2.2: Hardware configuration of Nvidia GPUs

GPU	# SM	# SP FU per SM	# DP FU per SM	# SFU per SM	#Register per SM
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

In this section, we discuss the support for floating-point arithmetic on Nvidia GPUs (with a focus on Fermi, Kepler, and Maxwell architectures), and we use a suite of micro-benchmark applications to characterize the latency of floating-point operations on these devices.

Nvidia GPUs consist of multiple parallel processors called Streaming Multiprocessors (SMs), each executing groups of threads (called thread-blocks) in a SIMT fashion. Besides including a shared register file, a shared local memory and an instruction/thread scheduler, every

SM 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 special math operations such as square root, logarithm, exponent, and trigonometric functions. Table 2.2 reports the hardware configurations of different GPU architectures used in our experiments.

2.2.1 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 Henry et al. [39] for the analysis of Fermi GPUs. This

Table 2.3: 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 2.4: 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

suite includes a set of single-threaded programs that perform a sequence of (the same) floating point instruction. 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). Instruction latencies are then measured through the Nvidia `clock()` function, which provides the current value of a thread-level counter that is incremented every clock cycle the thread is executed. 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. We modified and extended this benchmark suite in two ways. First, we observed that, on more recent GPU generations (e.g., Kepler and Maxwell), the compiler could rearrange instructions, possibly leading to incorrect measurements of the instruction latencies. To avoid this rearrangement, we introduced a barrier synchronization (`_syncthreads`) around `clock()` function calls. 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.

Tables 2.3 and 2.4 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, they have been designed to offer efficient 16-bit floating-point operations and larger on-chip memories, rather than fast double-precision floating-point arithmetic.

Fused multiply-add – In order 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.

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 2.3). These approximate operations are not IEEE compliant and they are available only for single-precision arithmetic.

Special math operations – The portion of Tables 2.3 and 2.4 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 2.3 (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

//Without fast math compile option	//With fast math option
FSETP.LT.AND P0,PT,R3.reuse,-126,PT;	
@P0 FMUL R3, R3, 0.5;	
RRO.EX2 R8, R3	RRO.EX2 R8, R3;
MUFU.EX2 R2, R8	MUFU.EX2 R2, R8;
@P0 FMUL R2, R2, R2	

Figure 2.3: Assembly code corresponding to the invocation of single-precision exponential function compiler without (to the left) and with (to the right) –use_fast_math compilation option.

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 2.3), the conditional (predicate) 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.

2.3 Floating-point Instruction Analysis

In this section, 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 2.2. For the sake of space and clarity, in this section we show only the results reported on the Kepler and Maxwell GPUs. As shown in Tables 2.3 and 2.4, 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 Section 2.2, the observations related to the comparison of different arithmetic precisions apply on all GPUs.

2.3.1 Addition

In order to study performance-accuracy tradeoffs of floating-point addition, we use the global summation benchmark proposed by Taufer et al. [1]. 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 value is also added to the input set.

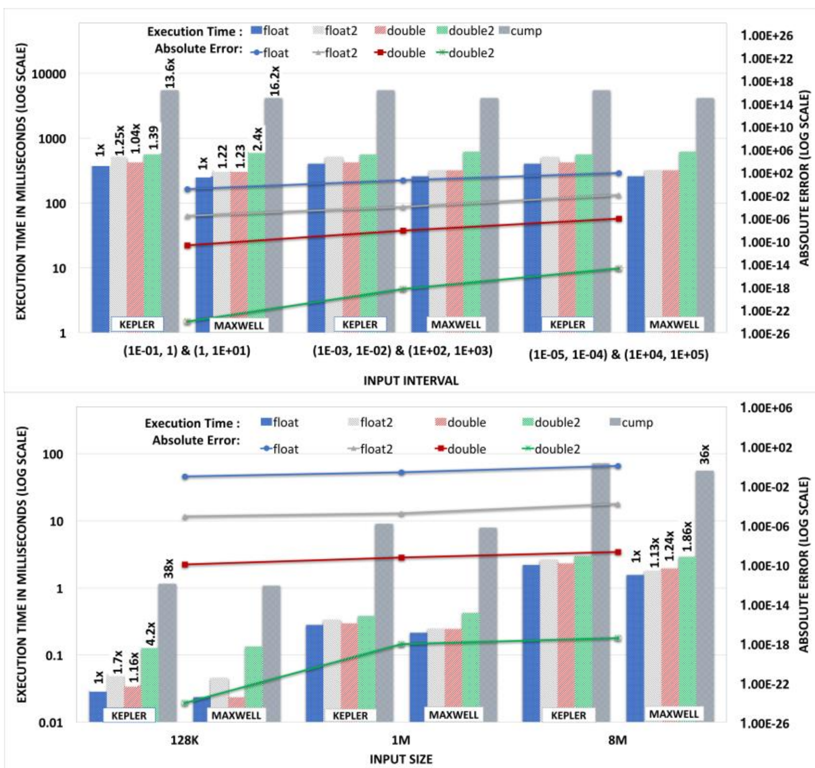


Figure 2.4: Performance and accuracy results for global summation on Kepler and Maxwell GPUs, expressed in terms of execution time and absolute error over the expected zero result (both in log scale), respectively. The numbers on the bars represent the slowdown of float2, double, double2 and CUMP over float. To the top, serial execution on 1 million values drawn from three different input intervals (x-axis). To the bottom, 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}).

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.

Figures 2.4 (top) 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.

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

Table 2.5: Number of registers used by global summation kernel with different arithmetic precision and GPU geneartions

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

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 2.4). Third, float2 is both less accurate and slower than double (or equally fast). This differs from the results in [1], 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.

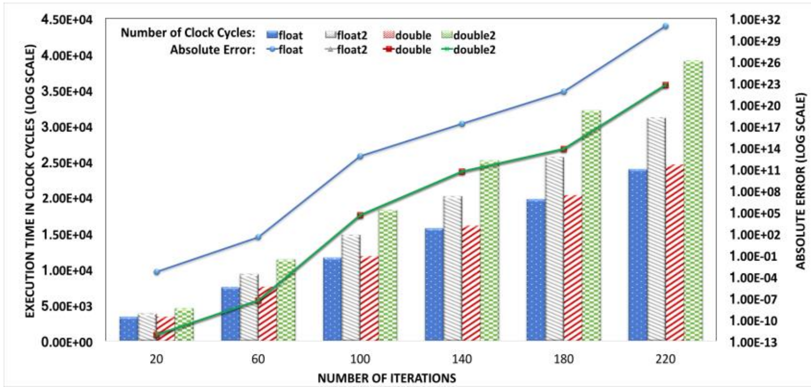
We now consider the relationship between arithmetic precision and allowed kernel concurrency. Table 2.5 shows the number of registers used by the global summation kernel with the considered arithmetic precisions. The register utilization of a kernel limits the number of threads resident on a SM, that 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.

Figure 2.4 (bottom) 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 [40]). 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.

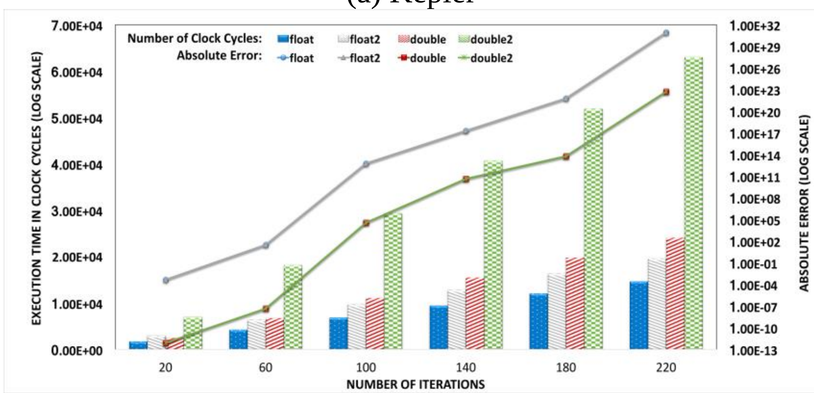
2.3.2 Multiplication

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.

Figure 2.5 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



(a) Kepler



(b) Maxwell

Figure 2.5: Accuracy and performance results of repeated multiplications with inputs randomly drawn from intervals (0.01, 1) and (1, 10). The top and bottom charts show Kepler and Maxwell results, respectively.

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 2.2 (`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 2.1) 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 (Tables 2.3 and 2.4). This motivates the different performance behaviors on the two architectures shown in Figure 2.5.

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.

2.3.3 Division

In order to study the behavior of the division, we use the do-undo benchmark [1], 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 2.6 presents the absolute error of 1 million iterations of the do-undo benchmark using inputs of different orders of magnitude. In Figure 2.6(a-b) we use small values for x and large values for y: specifically, x is randomly drawn from interval (0.0, 10.0) and y is set to 10^6 and 10^8. In Figure 2.6(c-d), we invert the selection, and we consider values 10^6 and 10^8 for x and values randomly drawn from interval (0.0, 10.0) for y. We report the results using float and float2

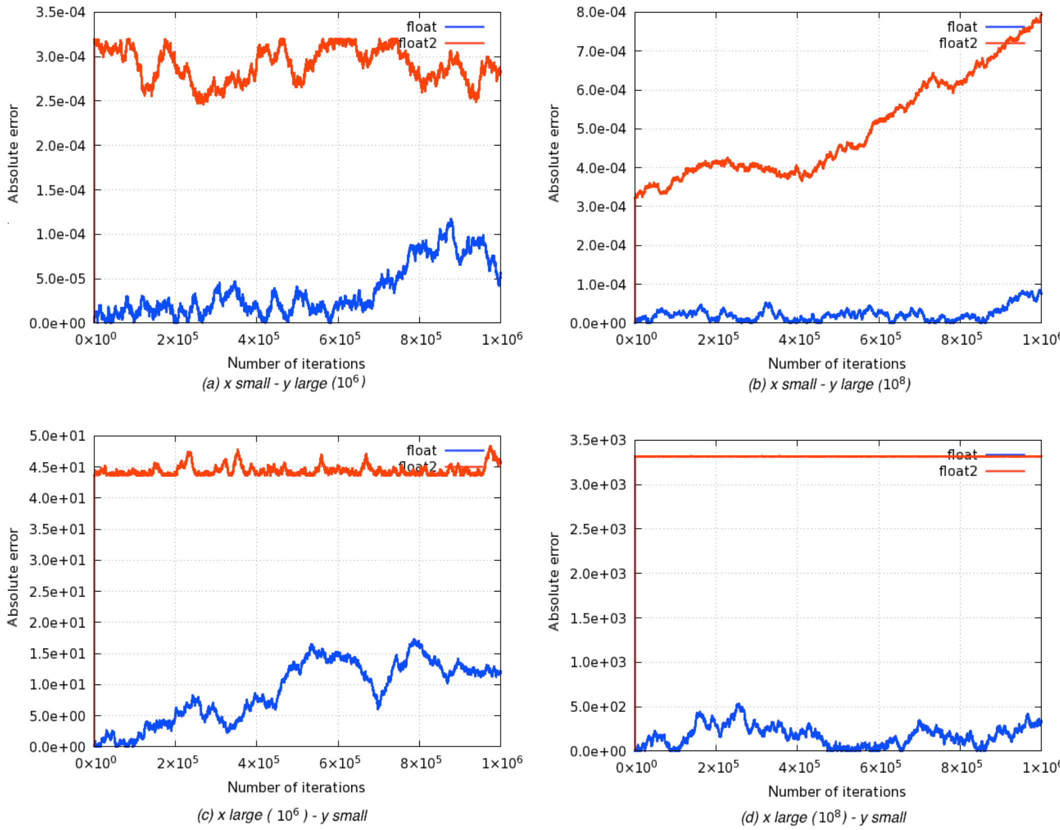


Figure 2.6: Absolute error reported by the do-undo benchmark (1 million iterations) with x and y inputs of different order of magnitude.

(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 might be related to the use of the division.

From Figure 2.2, 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 [1]. This is due to two facts. First, pre-Fermi GPUs used in [1] 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).

2.3.4 Natural Exponential 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 Section 2.4. 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 (*ex2* in Tables 2.3 and 2.4).

Thall [34] proposed approximating the natural exponential function using its Taylor series

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

and the following loop termination condition:

$$latest\ term\left(\frac{x^n}{n!}\right) > 1e - 20 * \exp(x.value) \tag{2}$$

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:

$$\begin{aligned} \exp(x) &= \exp(x.value + x.error) \\ &= \exp(x.value) * \exp(x.error) \end{aligned} \tag{3}$$

where exp(y) = ex2(y*1.4227).

We implemented three kernels that compute the natural exponential function: the first one (*float/double*) invokes the built-in exponential function on single- and double-precision floating-point values, the second one (*float2/double2_BF*) uses the implementation of the composite-

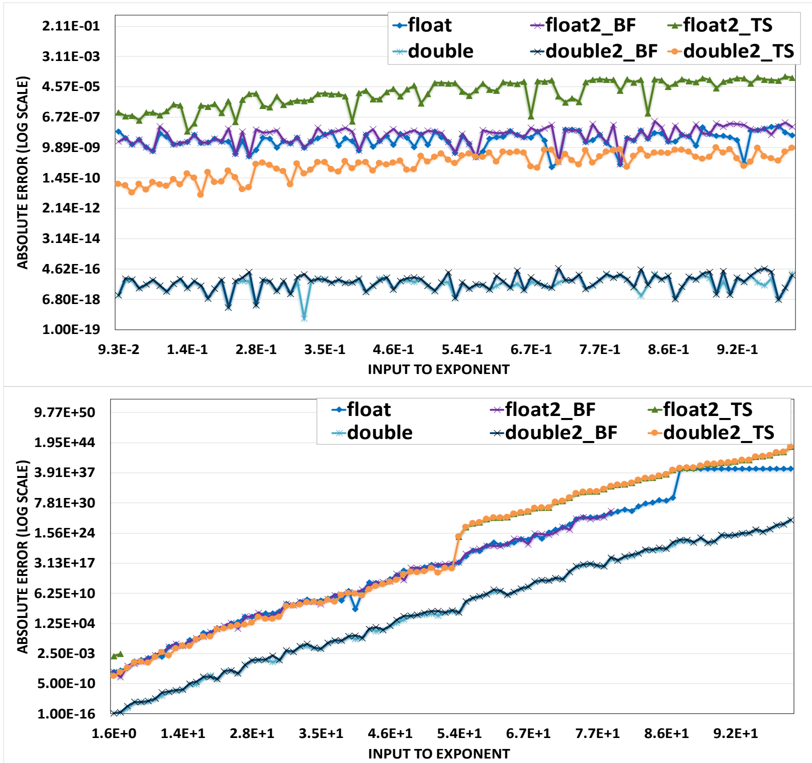


Figure 2.7: Accuracy (in terms of absolute error over GMP result) of the natural exponential function for two input ranges: (0.1,1) in the top chart, and (1, 100) in the bottom chart.

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. We generated the assembly code for these kernels and analyzed its instructions. Table 2.6 shows the assembly instructions generated in case of single, double, and composite precision arithmetic. We recall (Section 2.2.1) 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 2.3). The n parameter in the implementations based on

Table 2.6: Arithmetic instructions generated for the different versions of the natural exponential function

Inst	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	

the Taylor series represents the number of iterations (i.e., the number of terms in the series that are added together).

Although in this paper we focus on the natural exponential function, similar considerations apply for the logarithmic and power functions. As shown in Tables 2.3 and 2.4, 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.

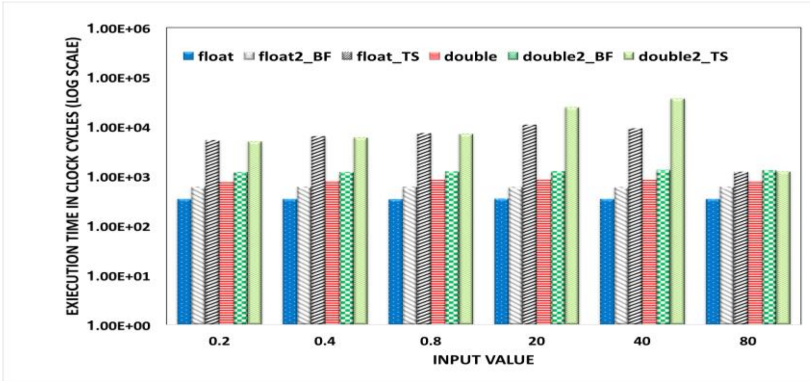
Figure 2.7 shows the accuracy – defined in terms of absolute error over the result of the MPFR library – of the different versions of the natural exponential functions (since CUMP does not support the exponential function, we computed the reference multiple-precision result on CPU). We show two ranges of inputs: (0.1, 1) and (1, 100). We recall that we use the convergence criterium (2) 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 2.7. For inputs less than 0.1, float2_TS converges in very

Table 2.7: Mean and standard deviation of the loop count "n" in the cmp taylor series

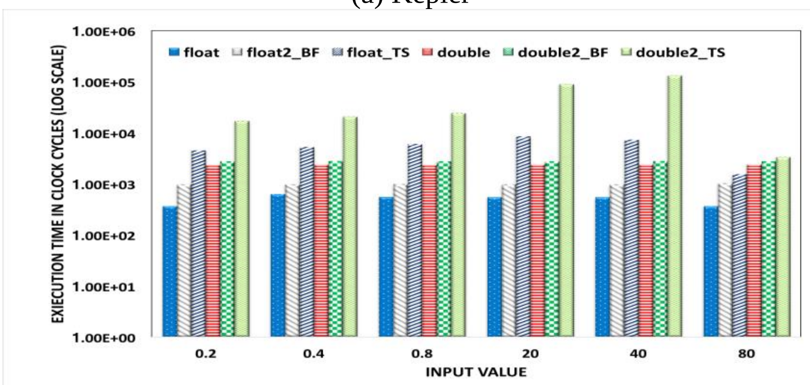
Input Range	Mean		Standard deviation	
	Float2	Double2	Float2	Double2
(0,0.1)	10.6	10.6	1.3	1.3
(0,1)	17.5	17.5	3.1	3.1
(0,100)	14.7	46.8	13.1	50

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). Second, implementations based on the built-in ex2 function exhibit better accuracy than implementations based on the Taylor series. 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. In addition, 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 2.8, missing values in the float2_TS series correspond to NaN results).

Figure 2.8 shows the performance of the natural exponential function on the 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 2.6). 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 (Table 2.7), they result in a larger number



(a) Kepler



(b) Maxwell

Figure 2.8: Execution time of exponential function on input values from ranges (0.1, 1) and (1, 100). The top and bottom charts show Kepler and Maxwell results, respectively.

of assembly instructions. Fourth, 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 2.4), double-precision implementations of the exponential function also perform better on Kepler devices.

In summary, for the exponential function standard floating-point arithmetic (float and double) 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.

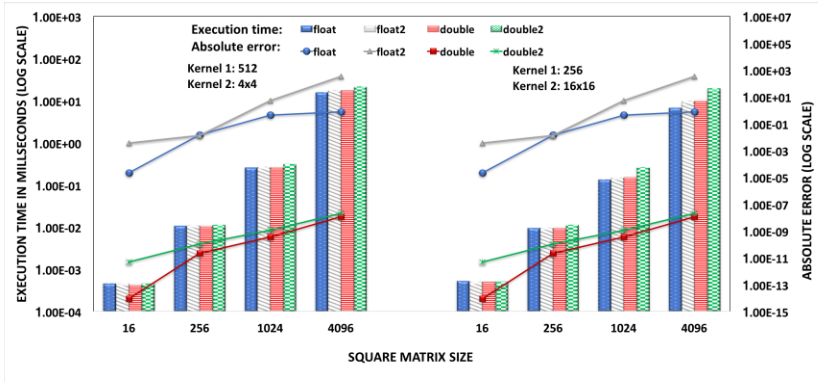
2.4 Benchmark Analysis

In this section, 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 [10]: Gaussian Elimination, LU Decomposition and LavaMD molecular dynamics. As in Section 2.4, 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.

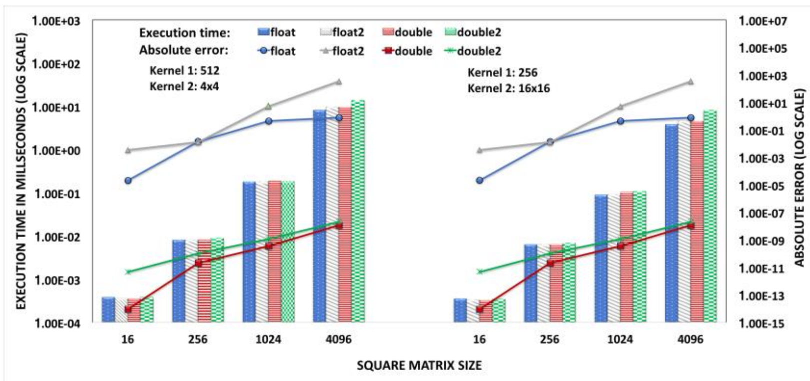
2.4.1 Gaussian Elimination

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 and the back-substitution is performed on CPU. The GPU forward-substitution kernel uses a combination of addition, multiplication, and division floating-point operations (2, 2 and 1, respectively) and is invoked multiple times in a loop. The output of the division is fed to the other operations.

Figure 2.9 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



(a) Kepler

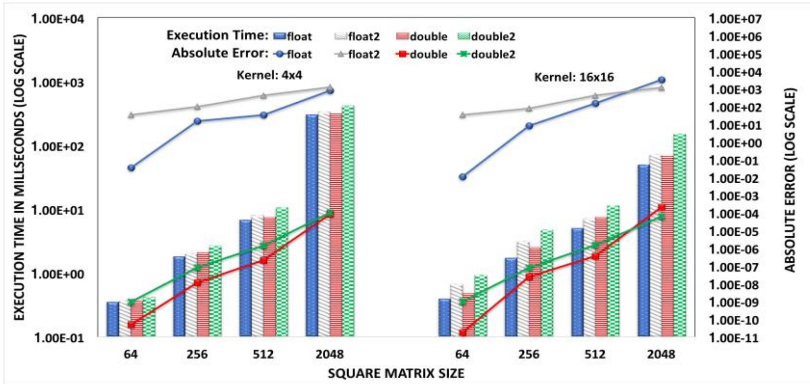


(b) Maxwell

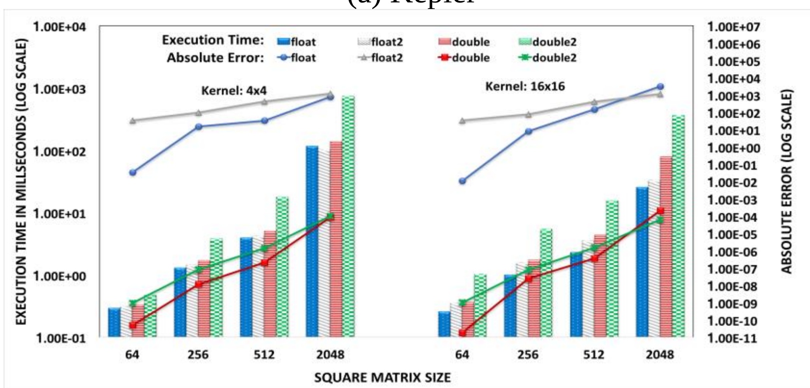
Figure 2.9: Accuracy and performance results of GE with elements of matrix A and vector b randomly drawn from intervals (10-2, 10-1) & (101, 102). The top and bottom charts show Kepler and Maxwell results, respectively.

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 2.9). 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.

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



(a) Kepler



(b) Maxwell

Figure 2.10: 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}). The top and bottom charts show Kepler and Maxwell results, respectively.

the GE forward-substitution kernels (0.08 and 1.12 flops/byte for float/double and CMP, respectively).

2.4.2 LU Decomposition

LU decomposition is a factorization technique to transform a square matrix A into a lower and an upper triangular matrices. Rodinia's LU implementation includes three iterative GPU kernels, which perform a variable numbers 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 2.10 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.

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.

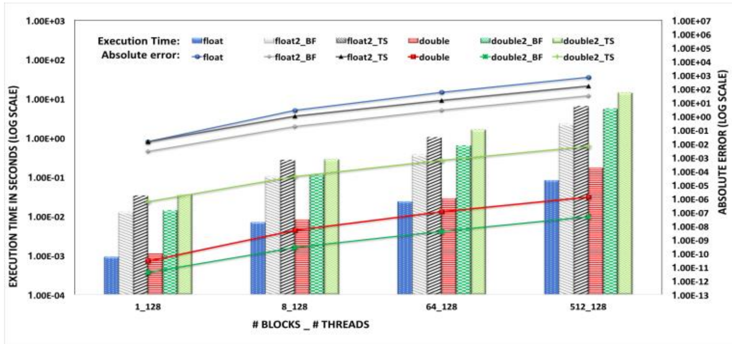
2.4.3 LavaMD Benchmark

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.

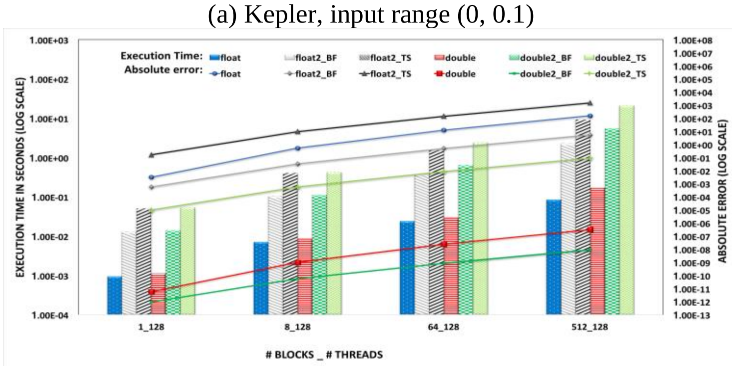
Figure 2.11 shows accuracy and performance results of LavaMD. Accuracy is measured in terms of the sum of the absolute errors on all the particles over a reference result computed using the 256-bit MPFR library. The initial values of the particle forces are randomly drawn from an input interval. We show 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.

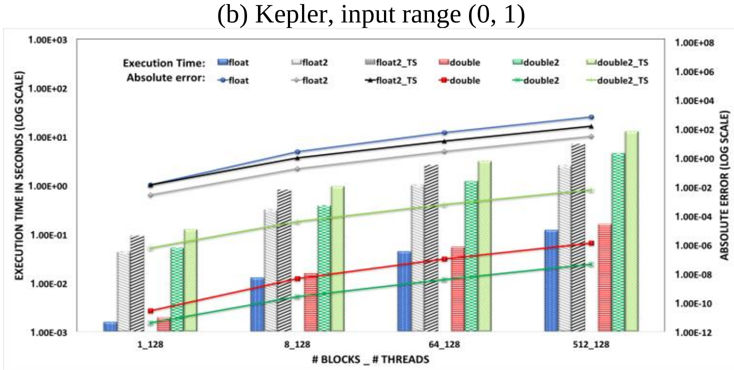
Accuracy: First, implementations based on double precision arithmetic (double, double2_BF, double2_TS) exhibit better accuracy than implementations based on single-precision arithmetic. Second, among these implementations, double2_BF and double2_TS exhibit the best and worst accuracy, respectively. This is coherent with the analysis presented in Section 2.3:



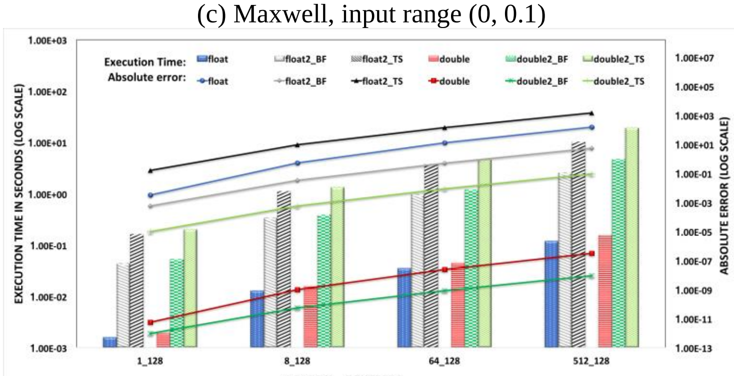
(a) Kepler, input range (0, 0.1)



(b) Kepler, input range (0, 1)



(c) Maxwell, input range (0, 0.1)



(d) Maxwell, input range (0, 1)

Figure 2.11: Accuracy and execution time of LavaMD for input ranges: (0, 0.1) in chart (a), and (0, 1) in chart (b). The top and bottom charts show Kepler and Maxwell results, respectively.

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 2.7); 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.

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 2.3, 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.

2.5 Conclusion

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

CHAPTER 3

Loop-aware Auto-tuner for High-precision Floating-point Applications

Based on the analysis on performance and accuracy of composite-precision on GPU, some arithmetic in composite-precision (i.e. division using double2) provides similar accuracy but involves way higher performance cost compared with double precision, which indicates the mixed-precision tuning opportunity. We decide to design auto-tuning tool on CPU and then extend it on GPU platform since the available CPU auto-tuning tools could not meet our needs. First, we need high-precision support for scientific applications. Second, those mixed-precision tools do not consider the influence of loops in the program. Third, we intend to apply the tuning on complex application that is not fully support by current state-of-art tools since the giant search space of optimal precision. Third, we aim to design an auto-tuning tool to simplify the work for programmers.

Since high-precision operations are typically significantly more costly than their basic floating-point counterparts, the relative performance gain achieved when tuning down from high-precision data types to double-precision is much more substantial than that resulting from tuning down from double to float. For example, on the same hardware platform, a purely float version of the CFD program mentioned above has an execution time of 185 seconds, 32% less than the double precision version of the program (recall that the double precision version is 80x faster than its 256-bit GMP counterpart). Meanwhile, when moving from double to float, the average error (with respect to the 256-bit GMP solution) increases from 2.27x10-16 to 1.26x10-7, a 109 order of magnitude difference. Previous work [14] has shown that, for many benchmark programs, tuning down variables from double to float while keeping the same level of accuracy does not lead to performance improvements. Similar results are reported in [11], which shows little to no benefit from using mixed precision on basic floating-point data types, particularly when the prescribed error bound goes below 10^{-6}.

These observations have several implications when broadening the scope of the analysis to applications that require high- precision arithmetic to guarantee a sufficient level of accuracy. First, tuning the precision of the floating-point operations for applications that require high-precision arithmetic can be more impactful both in terms of accuracy and performance than tuning applications whose accuracy requirements are satisfied by basic floating-point data types (i.e., float and double). Second, because of the high cost of high-precision arithmetic, reducing the number of tuning iterations (i.e., the size of the search space) becomes critical. This holds particularly for applications that have long runtimes even when using double- precision arithmetic alone. Third, having a tuning methodology that leads to results that are consistent across different inputs is equally relevant, since the tuning overhead is justifiable only if the application is run multiple times on different inputs (this holds also when using only basic floating-point arithmetic).

Based on this observation, we design a loop-aware tuning strategy, and we use it to analyze the effect of tuning down subsets of operations within basic blocks incrementally based on compiler analysis. Our tuning strategy uses the following criteria: (a) the number of occurrences of a particular variable or operation throughout the code, (b) the presence of the variable or operation in accumulation-like computation patterns (e.g., global summations or sequences of multiplications), and (c) the execution frequency of the basic block containing that operation. We propose a two-step approach. First, we use compiler analysis to generate a tuning plan specific to the program but largely independent of its input data. The tuning plan consists of a sequence of iterations, each specifying a set of operations to be tuned down from higher to lower precision. Second, we execute the tuning plan by modifying the program and running it with a given input and a prescribed error bound. Our proposed tuning method allows a good accuracy-performance tradeoff, providing a good alternative to using more traditional, computationally expensive approaches such as GMP to provide high accuracy.

3.1 Background

3.1.1 Floating-point Arithmetic

Floating-point arithmetic approximates real numbers by representing them with a finite number of bits. Floating-point encoding consists of a sign bit, a mantissa and an exponent. The IEEE floating-point standard includes single- and double- precision arithmetic, consisting of 32 and 64 bits, respectively. The bits for the mantissa field are 23 for single- and 52 for double-precision numbers, allowing approximately 7-8 and 16 decimal digit accuracy, respectively. To achieve higher accuracy, several high-precision libraries have been proposed. These include multiple and composite precision libraries. The former (e.g., GMP [6] and MPFR [7]) perform arithmetic operations on numbers with arbitrary size, theoretically limited only by the amount of memory available on the hardware. The latter (e.g., QD [8] and CAMPARY [9]) encode a floating- point number with a summation of multiple standard floating- point values. In the evaluation section, we use the GNU MPFR library as the "golden solution" for accuracy. In our auto-tuner, however, we use 128- and 256-bit CAMPARY as initial high- precision data type. This is because, for the considered bench- marks, CAMPARY provides the same accuracy as MPFR with a lower performance cost. Our tuning strategy, however, can also be applied to other high-precision libraries.

3.1.2 Related work

There have been several efforts aimed to design tuning assistants that use mixed precision to balance performance and accuracy of floating-point applications. Precimonious [11] operates by exploring the search space resulting from assigning different candidate precisions to the floating-point variables in a given program. To reduce the tuning time, it performs a delta-debugging search with heuristic pruning. Even so, the tuning time can be significant: up to hours for simple programs with an initial execution time in the other of few seconds. Blame Analysis [14] reduces Precimonius's search space further through compiler techniques aimed to identify variables that are less likely to impact accuracy and are better candidates for reduced precision. Along the same line, HiFP Tuner [15] reduces the search space by static analysis of the code and runtime monitoring of its behavior. Specifically, the tool identifies dependencies among floating-point variables and uses this information to guide a hierarchical search algorithm that progressively lowers the precision of groups of variables in the program. These tuning assistants operate only on float, double and long double and, on many of the benchmarks considered, show little or no performance benefit from using mixed precision on these basic floating-point data types [11], [14]. In addition, they don't focus on input sensitivity. How- ever, some of the proposed techniques can be incorporated in our tuner to allow fine-grained optimizations. FPTuner [16] 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. Unlike Precimonious, FPTuner focuses on tuning operators rather than tuning variables and then inferring the precision of the operators from that of the variables. FPTuner targets small code segments within loops, as opposed to full programs. It might be interesting to investigate combining FPTuner's techniques with ours, as it could allow for more fine-grained and rigorous tuning within basic blocks. Rather than operating on the source code, CRAFT [12], [13]uses binary instrumentation and modification to produce mixed-precision programs, and uses a hierarchical search to efficiently find regions of the double-precision code whose precision can be reduced. This tool has a low tuning overhead (up to 10x) and does not require the program's source code.

Besides auto-tuners, in recent years there have been several efforts aimed to provide verification [29, 41, 42] and compiler techniques [30], [31] for applications that include floating-point arithmetic. Gappa [29] is a proof assistant to facilitate the proof of numerical properties of floating-point programs based on interval arithmetic (for example, ensuring that variables and errors stay contained within specified ranges). It transforms a high-level description of a mathematical proof into a machine checkable version, and focuses on floating-point elementary functions. The Rosa compiler [30], [43] introduces a generic "real" datatype that can be used to write programs, and whenever possible, it generates implementations that ensure 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 [31] provides 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. The FLiT tool [41] is a cross-platform tester for HPC floating- point programs to understand how much variability exists under various compiler flags on different platforms, offering insight on performance and accuracy tradeoffs of floating- point programs on multiple platforms. Finally, ADAPT [17] uses algorithmic differentiation to estimate errors in the output of HPC programs and provide a precision-sensitivity profile that can be used to identify regions of the code that can be converted to lower precisions. These works are complementary to ours.

3.2 Auto-tuner Design

The proposed tuning process consists of two phases. The first phase determines a tuning plan for a given program through compiler analysis. The tuning plan consists of several steps (referred to as "tuning iterations" in the remainder of the paper). Each tuning iteration has an associated list of floating-point operations whose precision can be tuned down to improve performance. Except for one exception discussed later (section 3.3.2), the tuning plan generation does not require running the program. The second phase executes the tuning plan derived in the previous phase. For each iteration, this requires: (a) transforming the code by tuning down the operations indicated in the tuning plan and inserting the required data type definitions and conversions, (b) running the generated mixed precision code and verify that the achieved performance is within the prescribed error bound.

The tuning plan is generated on the double-precision version of the program. This is because the code information required to define the tuning plan (i.e., loop analysis, data dependencies analysis, number of operations in each basic block, etc.) can be derived from the double-precision code. Before executing the tuning plan, all the variables and instructions in the program are transformed to high precision ("iteration 0"). This is necessary to obtain the reference accurate result used to compute the error at the end of every tuning iteration. As mentioned in Section 3.1.2, while we use the CAMPARY library for high precision during the tuning process, in our experiments we compute the reference result using 256-bit GMP. We do so because we have observed that, on the considered benchmark applications, CAMPARY provides the same accuracy as 256-bit GMP but offers better performance (i.e., execution time). Our method, however, can be applied to any high precision library. In addition, it can be extended to multiple levels of precision.

When used to achieve better performance within a pre- scribed error bound, the tuner can terminate the execution of the tuning plan when the result violates the accuracy constraint. At the end of each tuning iteration, the tuner stores the version of the program generated in that iteration and records the error and the execution time of that version of the code. The executable file is required in case the error exceeds the error bound in the current iteration. In addition, storing that information provides the users the option to view or use intermediate tuning results. The tuner, however, can also be used to achieve better accuracy while limiting the performance loss. In this case, the tuning plan can be executed in reverse order, starting from the double precision version of the code (iteration 0 is still required to compute the reference result used to compute the error). The execution of the tuning plan will terminate when the performance constraint is violated.

We note that the tuning plan does not need to be executed sequentially. To accelerate the tuning process, when implementing the tuning plan some iterations can be skipped (or, more precisely, combined into a single iteration). For example, the SP benchmark in our experimental results has 158 tuning iterations and the default tuning procedure described above would step through iteration-by-iteration from the 1st to (potentially) the 158th. However, the user can specify to start the tuning from an intermediate iteration of the tuning plan (possibly backtracking if the resulting accuracy does not satisfy the requirements), thus saving tuning time. In other words, it is possible to execute the tuning plan through a binary search or other heuristics. Similarly, to speed up the tuning, it is possible to execute different tuning iterations in parallel. In this case, the process executing tuning step i must also perform the code transformations associated to the previous tuning iterations (i.e., iterations 1 to i-1).

3.3 Tuning Strategy Design

In this section, we first discuss several aspects that are relevant to performance and accuracy of floating-point programs. Then, we explain how we have used these observations to design our tuning strategy. The discussion in this section assumes that the floating-point operations of a given program must be progressively tuned down from higher to lower precision to achieve performance gains under a prescribed error bound. To this end, we prioritize tuning down operations (and, more generally, portions of code) that are likely to have a smaller effect on accuracy. As mentioned earlier, the same considerations apply (in reverse order) when tuning up a program from lower to higher precision to achieve better accuracy under a given performance constraint.

3.3.1 Aspects affection our tuning strategy

Loops – Loops are an important factor affecting accuracy and performance of floating-point programs. There are two reasons for this. First, especially for large loop sizes (in terms of number of iterations), instructions executed multiple times are likely to contribute more to the execution time of the program. Second, since floating-point operations may incur rounding errors, calculations inside large loops may lead to error propagation as the number of iterations in the problem increases (i.e., drifting may occur). Loops oftentimes implement accumulation patterns, such as global summations and repeated multiplications. Besides overflow, additions can incur rounding errors, especially when the elements summed together are of different orders of magnitude. Multiplications are prone to underflow and overflow. These problems get amplified when these operations are repeated. For example, when summing the elements of an array with positive values together, the partial sum increases progressively. After many iterations, the order of magnitude of the partial sum is likely to become higher than that of the remaining elements of the array.

Global summations and repeated multiplications are com- mon in scientific applications that require progressively updating the value of a system variable. For example, scientific simulations may need to update an energy potential (leading to a global summation), or to progressively update and accumulate some probabilities (leading to repeated multiplications) [1]. In addition, repeated summations and multiplications can originate from the use of Taylor series to approximate functions. For example, [31] discusses using polynomial approximations to reduce rounding errors.

With these considerations in mind, we prioritize tuning down floating-point operations existing outside of all the loops first, and then floating-point operations within loops in order of increasing loop size. Since loop sizes are often related to the input data size, this strategy makes the tuning plan more robust and input insensitive.

Accumulation patterns – Programs can include two kinds of accumulation patterns. The first kind includes the “repeated operation” pattern discussed above. Besides additions and multiplications, this pattern can of course apply to any operations, such as division and exponent, leading to similar considerations. This pattern is typically implemented using a loop with the loop size being the number of repeated operations; thus, the loop size indicates the number of error propagations as well. However, rounding errors can also originate from shorter sequences of accumulations, such as $x = a + b + c + d$. Here, we define accumulated dependency (AD) as the sequential length of the same type of operation. This metric is used to measure the degree of similarity between a series of operations and a repeated operation pattern. For example, computing the sum of four floating-point numbers results in AD = 2 for addition (e.g., $x = a + b + c + d$; in other words, the AD states that there are an additional two operations in sequence beyond the fundamental $a + b$ sum). Like loops, large AD has the potential for causing error propagation.

With this in mind, we prioritize tuning down operations with lower AD first, and repeated operations at last.

Operation types – Different arithmetic operations have a different cost when implemented in a high precision library. The cost of the operations depends on the high precision library in use. For example, in the composite precision library described in [1] composite precision addition requires 8 double precision additions and subtractions, multiplication requires 4 multiplications and 2 additions, and division requires 1 reciprocal, 4 multiplications and 2 additions and subtractions. CAMPARY offers certified and non-certified operations, each requiring a different number of double precision operations. In this work, whenever possible, we use certified operations (the library does not include a certified version of division). All things being the same, we prioritize tuning down those operations whose high precision implementation is more costly. This allows achieving higher performance gains in earlier iterations.

Data dependencies – Data dependencies between operations can affect both performance and accuracy of the code. Besides those in accumulation patterns, data dependencies can broadly involve sequences of instructions with different operations. In terms of accuracy, data dependencies can also cause error propagation. Since data dependencies imply operations to be executed in a particular sequence, if subsequent operations have different precisions, data conversion is also required. The cost of data conversion depends on the high precision library in use. For example, the conversion from double to composite precision can simply be done by assigning the double value to one term of the composite precision variable and zero to the other (as done in CAMPARY). Other libraries (e.g. [1]) use Dekker's split [38] in the conversion function, requiring 1 multiplication and 3 additions. CAMPARY performs the normalization of the terms within each arithmetic operation.

3.3.2 Tuning strategy

The considerations above guide the definition of our tuning strategy. First, the tuner sorts the loops in order of increasing loop size (i.e., number of loop iterations). Then, it tunes down the loops one by one, from the smallest to the largest. For each loop, it uses the information on the accumulation patterns, the dependencies, the kind of operations and their number to define the tuning order of the operations. Function calls require special consideration.

Inter-loop tuning – To sort loops, we need to determine the relative loop size of the various loops in the considered program. The tuner uses two schemes to determine the loop-tuning order: static analysis and dynamic profiling. The default option for the tuner is to use static analysis to determine the loop size while profiling the LLVM IR code. However, the tuner requires dynamic profiling if the relative loop size cannot be determined statically and in the presence of recursive functions. The dynamic profiling option runs the program with a given sample input and records the number of iterations that each loop performs. Note that, in order to sort and prioritize the loops, the tuner requires only the relative loop size. In most cases, while the number of loop iterations is data dependent (i.e., larger data sets can lead to more loop iterations), the relative loop size does not depend on the size of the input. Therefore, the order in which the loops of a program are tuned is mostly data independent. All operations outside of the loops are grouped together as the 0th loop, which has a "size" of one, and this is the scope that is always tuned down first.

Intra-loop tuning – Within each loop (including the outer- most one), the tuning is performed by operation. Operations of the same type (e.g., additions/subtractions, multiplications, divisions, etc.) are tuned down in the same tuning iteration. This allows limiting the number of tuning iterations. To define the tuning order, this strategy considers the presence of accumulation patterns, the number of operations (NBOP) of a particular type, and the cost of the operation. First the tuner sorts the operation types based on the number of accumulations (i.e., global summations and repeated multiplications) inside the considered loop. Operation types that don't appear in any repeated operation are tuned down first, since they are less likely to affect accuracy.

These being the same, the tuner sorts the operation types such that operations with a lower AD are tuned down first. For a given AD, it sorts by decreasing NBOP such that operation types with lower NBOP are tuned down first. Everything being equal, the tuner considers the cost of the high precision versions of the operations, and tunes down the most expensive operations first. Grouping the operations by type allows decreasing the number of tuning iteration.

While tuning the operations of the same type in the same tuning iteration reduces the number of tuning iterations, a more fine-grained tuning strategy can be derived from the above considerations. Specifically, the loop body can be divided in clusters of operations, each containing a set of data dependent instructions. All the operations belonging to the same cluster are tuned down in the same tuning iteration. The tuning order of the clusters is then defined based on the criteria above. First, clusters are sorted based on the presence of repeated operations (i.e., clusters without repeated operations are tuned down first). These being the same, clusters are sorted first by AD, then by NBOP, and finally by operation cost.

Handling of functions – In some smaller applications, the floating-point operations are mostly inside a single function to which we can easily apply our tuning strategies. For example, the whole computation may be performed in a single main function. However, most scientific applications have a large code base and the computation is split into multiple functions, posing a potential challenge to our tuner.

In order to have an efficient tuning result, for functions that are invoked only from a single call site we tune the program function-by-function and sort all functions based on the values of our tuning metrics (loop sizes, AD and NBOP) determined through static analysis and dynamic profiling. If a function is invoked from multiple call sites, before generating the tuning plan the tuner creates multiple copies of that function, one for each call site, and it tunes each of this copy independently along with the function invoking it. In other words, during the static analysis and dynamic profiling phase the tuner conceptually "inlines" the function within the calling function at each call site. This form of inlining does not change the code functionality, but it enables a coherent definition of the tuning plan, since it allows considering the effective contribution of each variable and instruction to the final result.

3.4 Implementation

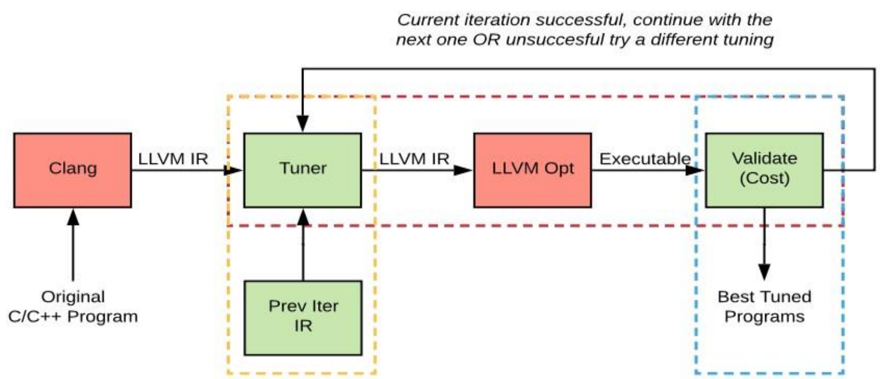


Figure 3.1: Proposed auto-tuner pipeline.

Figure 3.1 shows our auto-tuner pipeline, which is built on top of the LLVM compiler infrastructure. As can be seen, the tuner accepts a C/C++ program input and a desired error bound; we use Clang as the frontend. Our tuner operates on the LLVM IR representation before invoking the LLVM optimizer. We design the LLVM pass to instrument instructions, analyze the program structure, and perform the required program transformations. These transformations involve CAMPARY result array construction, floating-point precision conversion and operation replacement. The subsequent iterations perform the inverse transformation from CAMPARY to double. Figure 3.2 summarizes the operation of the tuner.

Figure 3.3 represents our added LLVM compiler pass. As can be seen, our added LLVM pass contains the following functional components: FP (floating-point) recognition module, FP

```
Input
     File: program.<c/cpp>,
     Value: error_bound (e.g., 10^-10)
Output
     Executables: program-<iter*>,
     Metrics: {error, time, iter}
Algorithm
     compile program.c into LLVM IR code
     // Static code analysis
     for f in functions in LLVM IR
      for b in basic blocks in f
       for i in instructions in b
        if (new loop structure found)
         create new loop container LC
         detect all operation types in LC
         for j in {add, mul, div, sqrt}
          determine acc. dependence ad[j]
          determine num. of operations nbop[j]
          LC_size = loop size for LC
          record LC_size, {ad}, {nbop} with LC
          store LC into LC_list;
     // Dynamic profiling to determine the relative size of the loops
     if (relative loop size input dependent)
       perform dynamic profiling to update LC_size
     // Application of the tuning rules
     sort LC in LC_list by LC_size
     sort operations in LC based on {ad, nbop}
     tuning plan: num of loops * num of op_types/LC
     // Execution of the tuning plan
     for each tuning iteration
      if (iter = 0)
        convert all FP ops and variables to CAMPARY
      Else
        tune FL ops based on tuning plan
      generate updated binary: program.ex
      run program.ex
      if (iter > 0)
       compute error E relative to iteration 0
      record computing time T
      store {E, T, iteration} in the output array
      if (E > error_bound)
        option to terminate the tuning process
```

Figure 3.2: Tuner operation pseudocode.

pro- filer, and FP insert & replace (INSRP) module. Our auto-tuner follows a standard approach [44] for instrumenting a program using LLVM. The tuner iterates through the modules at the program level first, then through the functions of a given module, then through the basic blocks, and finally through the actual instructions. We embed different functional components in LLVM to allow an analysis of the data flows and the identification of loops, AD and NBOP during the LLVM pass.

For the CAMPARY library, we have embedded in our auto-tuner two precision representations: multi_prec<2> (128- bit per variable) and multi_prec<4> (256-bit per variable). We observed that, on all the benchmarks considered, the use of 256-bit precision leads to accuracy

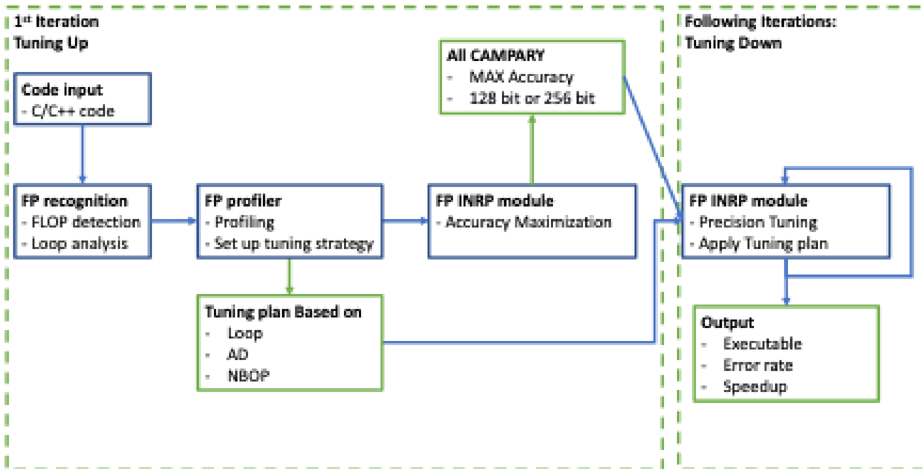


Figure 3.3: Compiler implementation flow chart.

similar to the use of 128- bit precision but with a significant performance disadvantage. Hence, in our experiments we use multi_prec<2> as our default choice. Note that the LLVM pass can be modified to use other high-precision libraries (e.g., GMP), while keeping the same tuning strategy.

FP Recognition: The main duties of the FP recognition module are (a) to recognize floating-point arithmetic operations, (b) to detect loops, and (c) to separate basic blocks within and outside loops. In the end, all the basic blocks in the code are categorized into two types: those that belong to a loop and those that do not. When determining the number of iterations of the inner loops (and, consequently, the number of times each basic block within each inner loop will be executed), we (recursively) multiply the number of iterations of the inner loop by the number of iterations of the outer loop.

FP profiler: Having collected information on the loops in the program and the floating-point operations contained in those loops, we then use the FP profiler to record this information for the auto-tuner to reference during subsequent tuning iterations. The auto-tuner can then set up the tuning plan based on the stored profiling results. The LLVM pass identifies and records the floating-point arithmetic operations while iterating through the instructions. The FP profiler assigns the instruction number from the IR code as the instruction ID. The FP profiler uses the

Pseudo assembly code (LLVM IR)

```
%0 = ld double *   // load a
%1 = ld double *   // load b
%2 = ld double *   // load c
%3 = ld double *   // load d
%4 = add, %0, %1   // AD=0
%5 = mul, %10, %11
%6 = add, %4, %2   // AD=0 -> AD=1
%7 = div, %12, %13
%8 = add, %5, %3   // AD=1 -> AD=2
Store %8, addr
%9 = ld double * , addr
%res = mul, %7, 2.0
```

Figure 3.4: Example of accumulated dependence calculation.

relative loop size information, determined using a combination of static analysis and dynamic profiling, to sort loops for tuning. In the static method, the LLVM IR associates a loop header to each loop, and this header contains the loop size information (i.e., the number of loop iterations). For dynamic profiling, the FP profiler will need to derive this information using a given input provided by the user. The accumulated dependence (AD) is calculated by tracing the usage of floating-point operations at the instruction level, as demonstrated in Figure 3.4. We define the AD for each operation type (addition, multiplication, etc.) as the maximum AD of the operations of that type in a given loop. The operations with lower AD values are tuned down first. The FP profiler counts the number of operations of each type inside and outside loops and stores the value as the NBOP for that operation type.

FP insert & replace module: The FP insert & replace module (INSRP) performs two main tasks: (1) maximize the precision using CAMPARY in the 0th iteration of the tuning process, and (2) apply the tuning strategy determined by the FP profiler previously. To enable precision tuning using the CAMPARY library, the INSRP module must support the conversion of all the floating-point variables and operations from double precision to the CAMPARY equivalents and vice versa. When the INSRP module replaces both operands and arithmetic operations with the appropriate CAMPARY library equivalents (left and right operands in Figure 3.5), the INSRP

```
IR code before the Target FLOP
%61 = getelementptr inbounds double, double* %58, i64 %60
%62 = load double, double* %61
…
%66 = getelementptr inbounds double, double* %63, i64 %65
%67 = load double, double* %66
// The original double version
//%68 = fadd double %62, %67
// The transformed CAMPARY version
%left  = alloca %class.multi_prec
convertDToCMP(%multi_prec* %left, double* %61, i32 1)
%right = alloca %multi_prec
convertDToCMP(%multi_prec* %right, double* %66, i32 1)
%Res   = alloca % multi_prec
certifAddExpans(%multi_prec* %Res,
                %multi_prec* %left,
                %multi_prec* %right)
```

Figure 3.5: Precision conversion demonstration.

module uses utility functions provided by LLVM to replace each following usage with the new CAMPARY floating-point operation. In this stage, we replace a given certified operation from the CAMPARY library with an equivalent function call to our tuner library that automatically performs the necessary precision transformations described above.

Handling of conversion overhead: Recall that our tuning strategy operates by progressively downgrading the precision of the floating-point operations from CAMPARY to double. When an operation is downgraded, the corresponding operands need to be converted from CAMPARY to double precision. This can cause multiple data conversions of the same variable in different points of the program, leading to unnecessary data conversion overhead. These data conversions can be avoided by propagating the type of the operation to the variables involved in it. This type propagation is complicated by the fact that it becomes necessary to handle data type mismatches between a modified variable and subsequent high-precision operations on the same variable. In other words, data conversion elimination requires handling all the data dependencies in the program. To address this problem, during the tuning process we keep both double and high-precision copies of each variable. When an operation (and related variables) are tuned down to double precision, the auto-tuner will check all the subsequent usages of the tuned down variables:

Table 3.1: Execution time of all-gmp, all-campary and all-double code versions and tuning space analysis

Bench-mark	# var-iables	Execution time (sec)			# Tuning iterations			Tuning-time (sec)	LOC
		GMP-256bit	CAMPARY	double	Autotuner	Exhaustive search	Precimonious		
SP	116	4,446	906.6	52.2	157	2^{116}	$2.6\text{x}10^{22}$ - $6.6\text{x}10^{32}$	55,615.8	2,237
CFD	51	10,570	1,654.1	102.9	19	2^{51}	201-1,435	44,805.5	386
dot	2	24	0.7	0.7	2	2^{2}	2-4	1.3	60
ge	52	-	12.3	4.6	6	2^{52}	201-1,435	440.1	316
lud	3	8,143.9	6,484.6	705.1	5	2^{3}	6-8	10,784.6	122
hot	11	568.2	355	277	5	2^{11}	23-61	3,476	206
lava	22	1,428.1	744.1	729.6	2	2^{22}	3-663	16,211.7	276

if any subsequent operation still uses the high-precision copy of the variable, the auto-tuner updates both copies of that variable; otherwise, the auto-tuner will keep and update only the double precision copy.

3.5 Experimental evaluation

3.5.1 Experimental Setup

We performed experiments to validate our auto-tuner and investigate the accuracy-performance tradeoff. We used seven benchmark applications: a kernel computing the dot product of two arrays of floating-point numbers; five applications drawn from the Rodinia Benchmark Suite [10], namely: Gaussian elimination (GE), LU decomposition (LUD), LavaMD, Hotspot, and Computational Fluid Dynamics (CFD); and one application from the NAS Parallel Benchmark Suite [37]: the Scalar Penta-diagonal solver (SP).

The benchmark applications considered contain a combination of additions (and subtractions), multiplications, divisions, exponents and square roots. The number of variables and LOC of these codes are reported in Table 3.1 (columns 2 and 10, respectively). In all cases, we performed the experiments on an Intel Xeon E5 processor (2.4 GHz) running Red Hat Enterprise 7.5 OS. We compiled the code using Clang with optimization level –O3. As a measure of performance, we consider the execution time of the primary computational functions. As a measure of accuracy, we consider the relative error using a reference solution computed using the 256-bit GMP library for all floating-point variables and operations. For those applications where the solution is an array, we take the relative error averaged over the whole array. Table 3.1 (columns 3-5) reports the execution time of the 256-bit GMP version, the 128-bit CAMPARY version and double precision version of the code.

3.5.2 Experimental Results on Complex Codes

SP – Figure 3.6 shows the iteration-by-iteration tuning results for the SP benchmark. In these experiments, we use the class S and class A input datasets, which consist of 1728 and 262144 elements, respectively. We recall that our auto-tuner operates in multiple iterations. First, it "tunes up" the whole program to provide the all-CAMPARY (128-bit) version of the code (iteration 0). Next, in each subsequent iteration (along the x- axis), it tunes down subsets of the floating-point operations and variables to double-precision arithmetic based on the application-specific tuning plan. For comparison, the charts also include the performance and accuracy results reported using the all-float and all-double versions of the program.

SP consists of five main functions, each invoked one time. The resulting tuning plan operates as follows: (1) iterations 1- 40 tune 135 operations inside function #2, (2) iterations 41-78 tune 135 operations inside function #3, (3) iterations 79-116 tune 136 operations inside function #4, (4) iterations 117-157 tune 301 operations inside function #1, and the final iteration tunes 1 global summation operation in function #5. Functions #1-#4 have loops with similar sizes and have similar AD, but differ in terms of NBOP. Function #5 contains a single loop that implements the

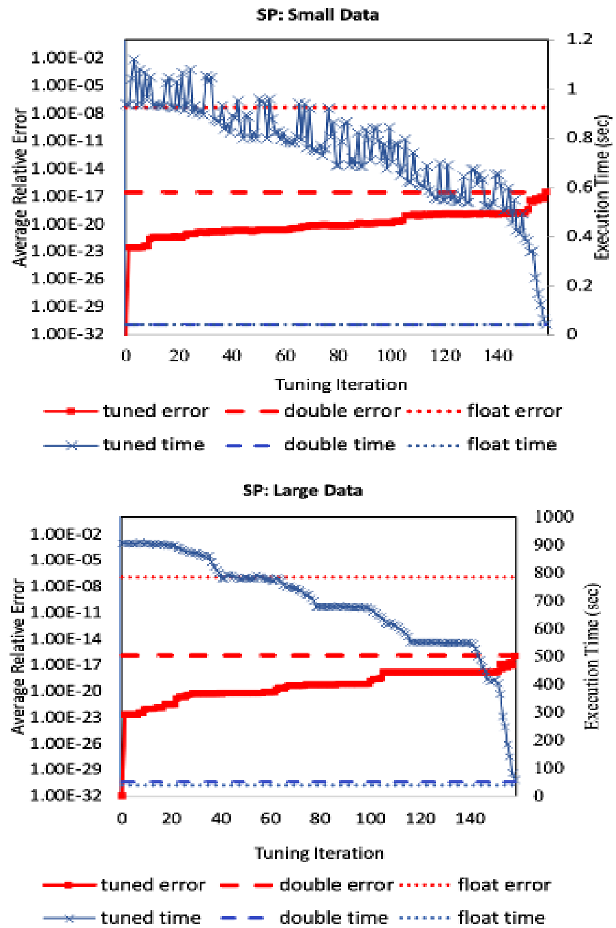


Figure 3.6: Tuning results for SP benchmark. The small dataset (chart on the top) has 1,728 elements. The large one (chart on the bottom) has 262,144 elements.

only accumulation pattern in SP. Due to this global summation pattern, function #5 is tuned last. To validate our tuning strategy, we tested tuning function #5 at the very beginning of the tuning process. This caused the relative error to rise to 10-17 (almost the same error as the double precision version of the code) after the 1st tuning iteration. On the other hand, with the proposed tuning strategy (Figure 3.6), the relative error is on the order of 10-23 after the 1st iteration and does not grow to 10-17 until the last tuning iteration.

Figure 3.6 leads to the following observations. First, for both input sizes, the tuned program shows better accuracy than the double precision version of the code. Progressively tuning down operations and variables leads to increasing performance gains while gradually sacrificing accuracy; as expected, at the last tuning iteration the accuracy and performance of the tuned program match the original double precision version of the code since all variables and operations have been tuned down to double. Second, while the execution time of the program increases with the data set size, the accuracy and performance trends stay the same across datasets. The mixed-precision version of the code (i.e., the tuned one) adds at least 2-3 digits of accuracy up to the 151st tuning iteration. After this tuning iteration we observe an abrupt accuracy loss (and execution time decrease). As can be seen, on the small dataset the execution time of the tuned program has some oscillations (while still following the trend). In this case, the short running time of the program (< 1 second) makes it more sensitive to the data conversions required when using mixed precision. In addition, the effect of natural runtime variations is more obvious when the execution time is limited. The overhead of the required data conversions and the effect of standard runtime variations are less pronounced on larger input datasets that lead to more substantial execution times (e.g., 50-900 seconds on the class A dataset). While tuning a simple program has little practical use, we show this result to demonstrate the validity of the tuning plan across inputs. In addition, we note that previous work focusing on single and double precision tuning alone has been showcased on applications with short running times (on the order of a few seconds). Third, we observe that, especially on the large dataset, the relative error exhibits three plateaus: (1) iterations 28-60, (2) iterations 70-100, and (3) iterations 105-151. Since the execution time decreases progressively, the last iteration of each plateau has a good performance/accuracy tradeoff. For example, at iteration 151 the code reports a 2.5x speedup over the all-CAMPARY version while providing 3 additional digits of accuracy over the double-precision version of the code (and, this holds on both datasets considered). Finally, we observe that the use of single precision arithmetic for SP would be not

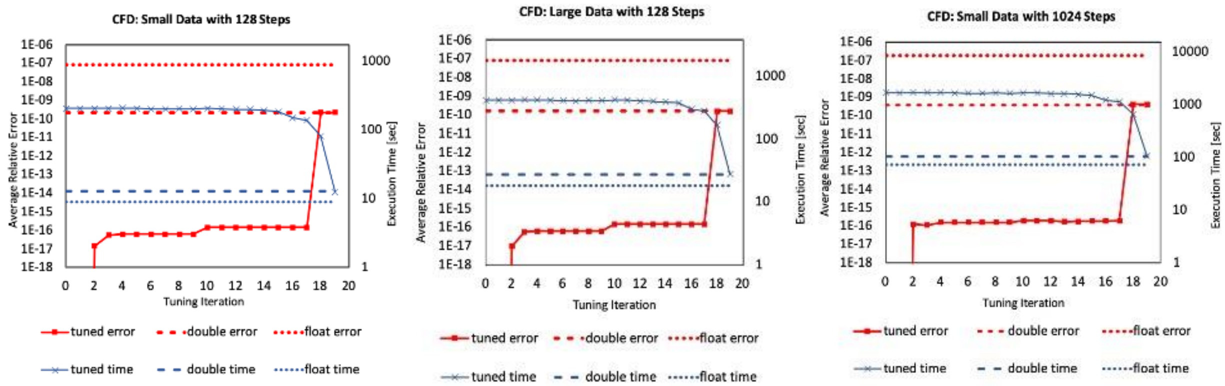


Figure 3.7: Tuning results for CFD benchmark. The small dataset (charts to the left and to the right) has 97k elements. The large one (middle chart) has 193k elements. The leftmost and middle chart show data collected on 128 steps, and the one to the right on 1,024 steps.

advantageous, in that it would lead to performance similar to that of double precision arithmetic with a significance accuracy loss (about 8 decimal digits).

CFD – Figure 3.7 shows the iteration-by-iteration results for the CFD benchmark on two datasets (97k and 193k elements) and different numbers of simulation time-steps (128 and 1024). As for SP, we use the same tuning plan on all considered input configurations, and we report the accuracy and performance results achieved by the all-float and all-double versions of the code. CFD has fairly complex computational patterns, large loops and a large number of FLOPs. We tuned the CFD code in the following order (n is the size of the input data set and m is the number of time steps in the outer temporal dimension): (1) steps 1-4 tune 26 operations outside of all loops, (2) steps 5-8 tune 21 operations of a first loop with size nm, (3) steps 9-11 tune 11 operations inside a second loop of size 3nm, (4) steps 12-15 tune 30 operations inside a third loop of size 3nm, and finally (5) steps 16-19 tune 159 operations of the remaining loop of size 12nm.

We make the following observations. First, as expected, the tuned program has a better accuracy compared with the double and float versions, and its accuracy tends to decrease as more tuning iterations are performed; again, at the final iteration the accuracy matches the double-precision case as all operations and variables have been tuned down. Similarly, the execution time of the tuned program gradually decreases as more operations are tuned down to double precisions. Second, while the execution time of the program increases with the data set size and the number of time steps simulated (the all-CAMPARY version of the code has an execution time of 139.5 seconds, 278.9 seconds, and 1,107.7 seconds for the three dataset/time step combinations considered), the trend of the accuracy and performance data stays the same across datasets. Specifically, the mixed-precision version of the code (i.e., the tuned one) adds at least 6 digits of accuracy in all the tuning space up to the 17th tuning iteration. After this tuning iteration we observe an abrupt accuracy loss (and execution time decrease). The mixed-precision code of iteration 17 exhibits the best performance/accuracy tradeoff, as it reports a 1.5x speedup over the all-CAMPARY version of the code while providing 6 additional digits of accuracy over the double-precision version of the code. The largest speedup occurs during the last tuning region (iteration 16-19), which corresponds to the most computationally intensive portion of the code. Here, 159 floating-point operations were tuned down: 86 additions, 66 multiplications, 4 divisions and 3 square root operations. Finally, as for SP, we observe that the use of single precision arithmetic would lead to a significance accuracy loss compared to double precision without bringing a significant performance advantage.

3.5.3 Experimental Results on Simple Codes

Table 3.2 summarizes the tuning results reported on the remaining five applications using small and large datasets (column 4). These applications have a simple code structure and contain only one or two loops. As a result, their tuning plans are fairly simple (i.e., they include only few iterations). Column 3 shows the number of tuning iterations (after iteration #0). Again, the tuning plan is specific to the program. For the tuned programs, we show the results after iteration 0 (i.e.,

Table 3.2: Summary of Experimental Results

Data Size	Bench-mark	#Tune Iter.	Input Size	Float		Double		Iter #n	Tune with 128-bit CAMPARY				Tune with 256-bit CAMPARY			
				Err.	Time	Err.	Time		Iter #0 acc_gain	Iter #0 time_cost	Iter #n acc_gain	Iter #n time_cost	Iter #0 acc_gain	Iter #0 time_cost	Iter #n acc_gain	Iter #n time_cost
Small	dot	2	2^{17}	$3.4\text{x}10^{-1}$	0.03	$5.4\text{x}10^{-10}$	0.03	1	10^{16}	8.9x	10^{4}	8.0x	exact	111.9x	10^{4}	20.0x
	ge	6	512^{2}	$2.5\text{x}10^{-1}$	0.02	$3.8\text{x}10^{-10}$	0.02	0	10^{4}	1.7x	10^{4}	1.7x	10^{4}	17.3x	10^{4}	17.3x
	lud	5	64^{2}	$6.8\text{x}10^{-3}$	0.08	$7.2\text{x}10^{-12}$	0.09	1	10^{16}	20.8x	10^{1}	19.9x	10^{59}	303x	10^{1}	302x
	hot	5	$2{,}048^{2}$	$2.1\text{x}10^{-2}$	20	$6.3\text{x}10^{-19}$	18	1	10^{16}	1.2x	10^{16}	1.2x	10^{48}	4.1x	10^{48}	4.1x
	lava	2	23	$1.5\text{x}10^{-4}$	4.3	$2.7\text{x}10^{-13}$	4.4	1	10^{3}	1.1x	10^{3}	1.1x	10^{3}	1.7x	10^{3}	1.2x
Large	dot	2	2^{19}	$1.5\text{x}10^{1}$	0.7	$3.4\text{x}10^{-7}$	0.7	1	10^{16}	12.5x	10^{6}	11.5x	exact	156.5x	10^{6}	36.3x
	ge	6	$8{,}192^{2}$	$7.5\text{x}10^{1}$	3.8	$1.1\text{x}10^{-8}$	4.6	0	10^{6}	2.7x	10^{6}	2.7x	10^{6}	28.6x	10^{6}	28.6x
	lud	5	$1{,}024^{2}$	$8.6\text{x}10^{-1}$	691	$3.3\text{x}10^{-7}$	705	1	10^{16}	9.2x	10^{2}	9.2x	10^{50}	135.0x	10^{2}	127.0x
	hot	5	$4{,}096^{2}$	$2.1\text{x}10^{-2}$	277	$6.3\text{x}10^{-19}$	279	0	10^{16}	1.3x	10^{16}	1.3x	10^{48}	4.2x	10^{48}	4.2x
	lava	2	8^{3}	$6.8\text{x}10^{-4}$	722	$1.4\text{x}10^{-12}$	730	1	10^{3}	1.0x	10^{3}	1.0x	10^{3}	1.7x	10^{3}	1.2x

the initial all-CAMPARY iteration) and the last iteration that provides an accuracy gain over double precision ("iteration n", where n is specified in column 9). Past iteration n, the accuracy of the tuned code is the same as that of the double precision version of the code, in some cases despite a performance loss. We show the results achieved using both 128- and 256-bit CAMPARY as high precision representation. In the table, the accuracy gain (acc_gain) metric indicates the order of magnitude increase in accuracy (i.e., decrease in error) over the double-precision version of the code. For example, acc_gain = 106 indicates six extra decimal digits of precision. Additionally, the time cost metric (time cost) indicates the slow-down of the tuned code over the double-precision solution.

We make the following observations. First, using 128-bit CAMPARY is generally preferable to using 256-bit CAM- PARY, since it brings the same accuracy gain with a lower performance loss. On hot and lud 256-bit CAMPARY leads to higher precision, but the more significant performance loss (especially for lud) makes the 128-bit representation still preferable.

Second, despite their simple structure, for hot and lava the tuned version of the code achieves 16 and 3 additional digits of precision over the double precision version of the code with a limited performance loss (20% and 10%, respectively). On ge, it achieves 4 additional digits of precision at a 70% time cost. For dot and lud, the performance loss is more significant, and, when high accuracy is needed, using the all-CAMPARY representation (i.e., code of iteration #0) is preferable over mixed precision since it brings higher accuracy at a small or no performance loss. Third, as for SP and CFD, using double is preferable over float, in that it allows significantly higher accuracy at a negligible performance loss. Last, as for CFD and SP, for all five applications the accuracy gain and performance cost have similar trends on small and large datasets, showing that the tuning plan is robust to different input data.

3.5.4 Tuning Cost

Table 3.1 (column 9) reports the tuning time of our auto-tuner. The tuning time depends on the number of tuning iterations (i.e., program runs) required by a particular tuning strategy. The execution time of each tuning iteration depends on the precision of the variables in the corresponding version of the program, and it falls between the all-double and all-CAMPARY execution time (columns 4 and 5 of Table 3.1). Table 3.1 (columns 6-8) reports the number of tuning iterations required by our auto-tuner, those resulting from the exhaustive search over the program's variables, and (an estimate of) the iterations required by the search heuristic implemented in the Precimonious auto-tuner [11] (which was designed to operate on float, double and long double variables). The largest (in terms of number of variables) application considered in [11] is *ge,* for which [11] reports up to 1,435 tuning iterations. To provide a coarse estimate of Precimonius's search space for those applications that were not evaluated in [11] (*CFD*, *dot*, *lud*, *hot* and *lava*), we indicate the minimum and maximum search space reported in [11] for applications with a comparable number of variables. For SP (the largest application), we computed the minimum and maximum space reduction factors of ge over exhaustive search reported in [11], and applied these factors to the search space of SP. As can be seen, to meet the requirements of complex applications with large search spaces and make high precision tuning practical, our auto-tuner limits the search space to a reasonable size.

3.6 Conclusion

In conclusion, we proposed an auto-tuner that leverages compiler analysis to convert a floating-point source code into a mixed-precision executable while balancing accuracy and performance. Our proposed tuning method allows a good accuracy-performance tradeoff, providing a good alternative to using more traditional, computationally expensive approaches such as GMP to provide high accuracy. For example, the 256- bit GMP version of CFD has a runtime of nearly three hours (75.7 times the one of the tuned version of the program). While designed to address large applications with complex code patterns, our auto-tuner proved effective also on simple applications and algorithms that include only one or a couple of loops. We note that naïve tuning methods use exhaustive searches from testing the candidate precisions on all floating- point variables, leading to large tuning spaces. For complex applications with tens to hundreds of floating-point variables and several loops, the proposed tuning approach is a practical way to improve the program's accuracy without killing performance or suffering from extensive searches.

CHAPTER 4

Mixed-precision Auto-tuning for Floating-point Applications on GPU

GPUs have been widely used to accelerate a wide range of applications from disparate domains: 3D rendering, animation, astronomy, machine learning, bioinformatics, computational mechanics, databases, computational finance, to name a few [5]. Many of these applications, including molecular dynamics [1], astrophysics [3], physics [4] and climate modeling [2], rely on floating-point arithmetic, which is approximate in nature. To address this issue, previous work has proposed high-precision libraries such as GMP/MPFR [7] and QD [45], and some of these libraries have been ported to GPU [9, 32, 33]. However, these libraries offer improved accuracy at a significant performance cost. For example, when using 128-bit composite precision, LavaMD from Rodinia Benchmark [10] run on a high-end Nvidia GPU experiences a 27x and 192x slowdown over double and single precision, respectively. This performance gap affects the practicality of high-precision floating-point libraries on large-scale applications and simulations that can take days to run even when using double precision arithmetic.

In this work, we propose a mixed precision autotuner for floating-point applications running on GPU. Our tuner supports single, double and 128-bit composite precision, and can be easily extended to other arithmetic precisions. To reduce the tuning space while identifying representative tuning points, our autotuner leverages compiler analysis. In particular, our tuning strategy takes into account code patterns prone to error propagation and performance degradation when using high precision data types (e.g., loops, accumulation patterns, floating-point intensive code sections), and GPU-specific considerations (CPU-GPU memory transfers, kernel configurations, shared memory variables and atomic operations).

Our autotuner starts by generating a high precision version of the code, and then tunes down the precision of the variables progressively to achieve performance gains with limited accuracy loss. To achieve this goal, it operates in two phases. First, it uses static analysis and dynamic profiling to generate a tuning plan that specifies the tuning orders of the floating-point variables within the program. Second, it executes the tuning plan iteratively, either to completion (i.e., until all variables are in single precision), or until a prescribed error bound is reached.

We build on our previous work [20], where we proposed a mixed-precision CPU autotuner for floating-point applications requiring high precision. Specifically, we still focus on computational patterns that are prone to error propagation, such as loops and accumulation patterns. However, here we extend our previous work with GPU-specific considerations to better target CUDA code. From a methodological standpoint, we depart from our previous work and perform variables-based rather than operations-based tuning. This has two advantages: first, it simplifies the problem of limiting data type conversion overhead; second, it allows incorporating considerations on the location of the variables and the memory transfers into the tuning policy. From an implementation point of view, rather than developing a proof-of-concept based on LLVM, we have implemented a fully functional, automated pipeline that leverages the ROSE compiler to simplify code analysis and operate at the level of the source code. Rather than generating binary code, our GPU autotuner generates source code that can be further optimized manually by the programmer.

4.1 Background and Related work

4.1.1 High precision library on GPU autotuner

Arbitrary-precision libraries [6, 36, 45] provide increased accuracy by performing arithmetic operations on data types with arbitrary sizes. CUMP [33] is an arbitrary precision library for CUDA based on GMP [7], but it only supports a very limited number of arithmetic operations. Composite precision arithmetic uses a fixed number of floating-point variables to represent a real number. Composite precision is a compromise between arbitrary and standard precision: it provides better accuracy than standard precision and better performance than arbitrary precision. CAMPARY [9] is a composite-precision library for GPU that provides IEEE-compliant floating-point operations. In this work, we use a composite precision library called QD [32]. Differently from CAMPARY, QD offers reasonable compilation time on GPU. In addition, it provides better arithmetic operation support than CAMPARY and easier primitive invocation (overloading operator support). QD supports both 128- bit (*gdd_real* type) and 256-bit (*gqd_real* type) precision. In this work, we use *gdd_real* since the considered benchmark applications do not benefit from 256-bit precision. Finally, *gdd_real* on GPU is based on the double2 CUDA data type and provides coalesced memory accesses.

4.1.2 Impact of precision on performance and resource utilization on GPU

To understand how the use of different precisions affects performance and resource utilization on GPU, we use three micro-benchmarks that perform a sequence of additions, multiplications and divisions on an array of floating-point values. We progressively increase the number of operations performed so to gradually increase the arithmetic intensity of the kernels. To better understand the performance results, we also analyze the CUDA PTX code of basic arithmetic operations (additions, multiplications and divisions) under different precisions (single, double, and 128-bit *gdd_real*). Figure 4.1 shows the performance of the three microbenchmarks and the memory transfer cost for single precision, double precision and *gdd_real*. In all cases, the input array consists of one million elements, and the arithmetic operation performed is repeated from 1 to 16,000 times ("repeated computation factor" along the x-axis). In addition, in all cases

Table 4.1: Number of arithmetic operation and register usage for different precisions

	Repeated addition			Repeated multiplication			Repeated division		
	gdd_real	double	single	gdd_real	double	single	gdd_real	double	single
add	11	1	1	11	0	0	22	0	0
mul	0	0	0	11	1	1	10	0	0
div	0	0	0	0	0	0	2	1	1
total	11	1	1	22	1	1	34	1	1
registers	22	4	4	48	4	4	68	4	4

the thread configuration has been chosen so to fully utilize the GPU resources. Table 4.1 reports the number of addition, multiplication and division instructions in the CUDA PTX code for the three microbenchmarks, along with the register utilization.

We make the following observations. First, for all three arithmetic operations, the performance of *gdd_real* is half that of double precision. Second, the performance gap between *gdd_real* and single precision is higher for the division microbenchmark. There are two reasons

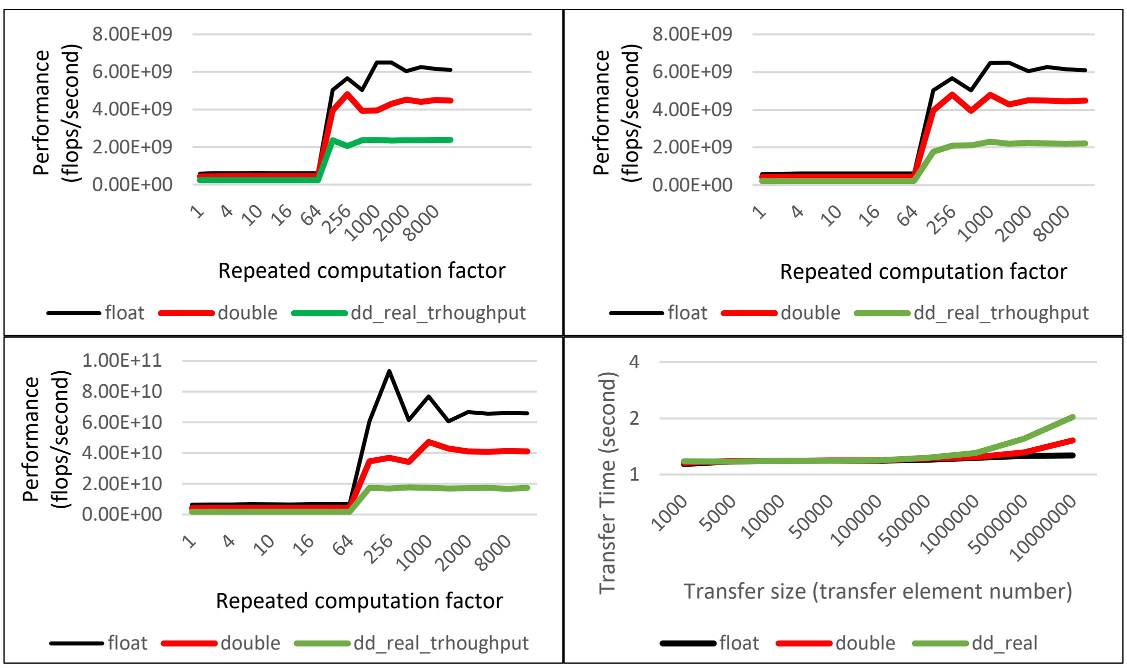


Figure 4.1: Performance (flops/second) trend with increasing arithmetic intensity and memory transfer performance with increasing data size. Top left is the repeated addition, top right is the repeated multiplication, bottom left is the repeated division and bottom right is the memory transfer.

behind this. First, as shown in Table 4.1, the register usage for 128-bit division is 17 times higher than that of *single* precision. Second, the 128-bit division implementation requires 34 basic arithmetic instructions. Not only does the GPU implementation of the *gdd_real* type use two standard *double* precision variables (*value* and *error*), but it also uses some compensation techniques to further improve the accuracy [32, 45]. The higher register utilization of the *dd_real* data type affects the number of resident threads that can interleave on a GPU, also contributing to performance degradation. Finally, the difference in memory transfer cost between different precisions is less pronounced than the performance gap. When the number of input elements is below 1000k, the data transfer costs are almost the same across precisions. GPU data transfers are done in batches.

4.1.3 Related work

Several efforts have focused on mixed-precision tuning on CPU. The Precimonious autotuner [11] leverages delta debugging to reduce the search space for single, double and long double precision. A follow-up work uses blame analysis [14] to further reduce the search space. FPTuner [16] limits the search space by using a SMT-solver to tune groups of operations in the same step, and it provides a rigorous error bound. Similarly, HiFP-Tuner [15] tunes groups of variables instead of single variables, and it does so by applying community structure detection to mixed-precision fields. These works don't consider high precision and GPU optimizations. More recently proposed floating-point tuners [17-19] have been showcased also on GPU. ADAPT [17] uses algorithmic differentiation to estimate the error and reduce the exploration space. Different from our tuner, ADAPT does not generate code but only provides a report with recommended precision configurations. AMPT-GA [18] uses a dependency graph to build a performance model used to reduce the tuning space, and it covers type casting overhead in the analysis. The tool requires some manual tuning. AMPT-GA does not support high precision and does not include GPU-specific considerations in the tuning strategy. The recently proposed GPUMixer [19] is a performance-driven autotuner for GPU kernels. GPUMixer uses static analysis to identify sets of floating-point operations on low precision, called *FISets*, such that the data that enter and exit each set are in high precision. In order to maximize performance, the algorithm proposed to identify *FISets* aims to maximize the number of low-precision arithmetic operations while minimizing the number of type cast operations required on the boundaries of the sets. In addition, GPUMixer uses shadow execution to determine the error and maintain a prescribed error bound. Like other GPU tuners, GPUMixer does not support high precision (beyond double) and does not cover additional GPU-specific considerations (like the ones included in our study to prune the search space).

Nvidia [46] has proposed a method to adaptively tune the precision of deep learning pipelines. Modular precision [47] accelerates data accesses for mixed-precision by decoupling the storage format from the arithmetic format through a modular precision ecosystem. This approach splits the storage of standard IEEE format into different segments (*mantissa* and *exponent*), and lays them out so that the mantissa can be retrieved efficiently. Anzt et al [48] leverage adaptive precision in block-Jacobi preconditioning on GPU. They store some data blocks in lower precision to reduce the data movement cost without affecting the computation and later restore the precision if needed. Grützmacher et al. [49] apply customized precision to accelerate Page-rank on GPU, leading to fast data access for mixed precision. These works are complementary to ours.

4.2 Autotuner Design

The goal of our autotuner is to automatically generate mixed-precision versions of a given GPU program achieving different tradeoffs between accuracy and performance. In particular, the autotuner can be used either to generate code that maximizes performance under a prescribed error bound, or code that maximizes accuracy while limiting the performance penalty over a low-precision version of the code. In the former case (described below), the autotuner starts from a high-precision version of the code and progressively reduces the precision of its variables so as to improve performance while limiting the accuracy loss. In the latter case, the autotuner starts from a low-precision version of the code and progressively increases the precision of select variables. Given a performance/accuracy goal, the exhaustive search over all possible settings of the variables' precision would require exploring a number of configurations that is exponential in the number of variables in the program. Our autotuner aims to reduce the size of the search space by leveraging compiler analysis and characteristics of GPU code. Our implementation supports CUDA code, but it can be easily ported to support OpenCL code as well.

4.2.1 High-level design

Our autotuner operates in two phases: *tuning plan generation* and *tuning plan execution*. In the tuning plan generation phase, the autotuner leverages compiler analysis to generate a *tuning plan* guiding the tuning orders of the functions and variables in the program. The tuning plan consists of a set of iterations. Each iteration has an associated version of the program, where different code versions differ in the settings of the precision of the floating-point variables (both CPU and GPU variables). When the goal is to maximize performance, the tuner starts converting all floating-point variables and operation in a given program into high precision (in our implementation, 128-bit *gdd_real*). In subsequent iterations of the tuning plan, the tuner incrementally reduces the precision of the variables (one variable per iteration) based on the tuning strategy explained below. In the tuning plan generation phase, we use the ROSE compiler [50] to perform static analysis of the input program. In addition, we perform a single run of the low-precision version of the program to collect dynamic profiling information such as the relative loop sizes, the size of the CPU-GPU memory transfers, and the thread configurations of the CUDA kernels.

In the tuning plan execution phase, the tuner "executes" the tuning plan. In other words, iteration-by-iteration, it runs the associated version of the program, and records the relative error and the speedup compared to the full high precision version of the program corresponding iteration 0. In this phase, our autotuner can be used in two modes. In the first mode (prescribed error mode), the tuning terminates when a user-provided error bound is reached. In the second mode (full tuning) the autotuner executes the entire tuning plan (from all high precision to all single precision version of the code). The autotuner can be configured to skip some tuning iterations (effectively tuning multiple variables in a single iteration), to start tuning from an intermediate tuning iteration, or used various strategy to execute the tuning plan (e.g., binary search).

Our current implementation supports three candidate precisions: 128-bit *gdd_real*, 64-bit double precision and 32-bit single precision. However, it can be easily extended to support more precisions. It operates in two stages: in the first stage, the autotuner tunes all the *gdd_real* variables gradually into double precision. In the 2^{nd} stage, it tunes all the double precision variables into single precision. This two-stage tuning has two advantages. First, since each tuning stage has only two candidate precisions (1^{st} tuning stage: *gdd_real* and double precision, 2^{nd} tuning stage: double and single precision), it is easy to implement and extend to additional precisions. Second, not including high precision variables, the second tuning stage has less tuning overhead (i.e., shorter execution time per-iteration).

4.2.2 Tuning strategy design

The tuning strategy determines the tuning order of the floating-point variables in the program under consideration. Our tuning strategy operates in two phases: functions sorting and variables sorting. In the first phase, it sorts the functions within the program (both CPU functions and GPU kernels); in the second, it analyzes the program function-by-function and determines the tuning order of all floating-point variables. The tuning orders for functions and variables are determined based on code characteristics and on GPU-related considerations. In particular, for both CPU and GPU code, the tuning strategy considers the following of code properties: loops, presence of accumulation patterns, number of occurrences of floating-point variables, and kind of floating-point operations performed. For GPU code, it also considers the following aspects: CPU-GPU data transfers, reuse of variables across multiple kernels, shared memory variables, thread configuration of the kernels (i.e., number of thread-blocks and their size), and atomic operations.

Below, we provide more details on the code properties and GPU considerations affecting the tuning strategy, and then we detail the operation of functions' and variables' tuning.

4.2.2.1 General code patterns guiding tuning

Loops – Loops are an important factor affecting performance and accuracy of floating-point programs. This can be easily explained. First, instructions executed multiple times are likely to contribute more to the execution time of the program. Second, rounding errors are an important source of inaccuracy for floating-point operations, and large loops can cause error propagation as the number of iterations increases. In addition, loops often implement accumulation patterns, such as global summations and sequences of multiplications. Besides overflow, additions can incur rounding errors, especially when the elements summed together are of different orders of magnitude. Multiplications can be subject to underflow and overflow. Iterative code patterns can aggravate these issues. For example, when summing the elements of an array with positive values together, the partial sum increases progressively, and eventually its order of magnitude can become higher than that of the remaining elements of the array. These accumulation patterns are found often in scientific applications that require progressively updating the value of a system variable (e.g., energy potential), or computing probabilities, or approximating functions through Taylor series [1, 31]. We this in mind, *we prioritize tuning down floating-point variables updated within loops in order of increasing loop size*.

Accumulation patterns – Programs can include two kinds of accumulation patterns. The first kind includes repeated operations such as global summations (e.g., `x+=a[i]`), and it can apply to any arithmetic operation. In serial code, this pattern is typically implemented using a loop with the loop size being the number of repeated operations, and, as a consequence, of error propagations. In parallel code, global summations (and generic accumulations) can also be cooperatively performed by multiple threads, often leading to the use of atomics. In addition, rounding errors can accumulate in the presence of shorter sequences of operations, such as `x=a+b*c/d`. We define accumulated dependency (AD) as the sequential length of the computation to calculate a variable. For example, variable x has AD=3 in this example; and a global summation implemented using a loop of size N results in a variable with AD=N. The AD of a variable provides a measure of the degree of error propagation that can occur during its computation. *We prioritize tuning down floating-point variables with lower AD first, and high-AD variables computed using repeated operations (such as global summations) at last*.

Operation types – As shown in Table I, the cost of high-precision operations varies from operation to operation. Divisions and additions are the most and least expensive operations, respectively, both in terms of number of double-precision instructions they require and of register utilization. All other aspects being equal, *we prioritize tuning down variables involved in more expensive operations*.

4.2.2.2 GPU-specific considerations

CPU-GPU data transfers – Our tuning strategy differentiates variables involved in CPU-GPU data transfers from other variables, since their precision affects data transfer costs and, ultimately, performance.

Shared memory variables – The number of threads that can be resident on a GPU and run in an interleaved fashion depends on the shared memory and registers utilization of a CUDA kernel, and the shared memory capacity is often a limiting factor. The reduced resource utilization resulting from more variables being tuned down to lower precision affects the number of resident threads, possibly improving performance.

Degree of multithreading – The number of threads run plays a role on the utilization of the floating-point units and the amount of synchronization resulting from atomics. Thus, our tuning strategy takes this factor into consideration.

Atomic operations – As mentioned above, atomic operations within GPU kernels often implement accumulation patterns, such as global summations. In order to determine the size of the accumulation pattern, we consider both the size of the loops enclosing the atomic operation and the number of threads and thread-blocks spawned by the kernel. CUDA provides atomic operations only for standard data types. We extend the *dd* library with atomic operations for the *gdd_real* data type implemented using the *atomicCAS* CUDA function [51].

4.2.2.3 Phase 1: Determining the functions' tuning order

We recall that our tuning strategy starts by sorting the CPU and GPU functions of the program under consideration based on static analysis and dynamic profiling. In this phase, device functions and small helper functions are inlined, and their variables are tuned along with those of the calling function. We aim to tune down first functions that include fewer sources of inaccuracy.

We base the tuning order of functions on the following code properties: accumulation patterns, number of floating-point operations, loop sizes, number of function invocations, and, for GPU kernels, number of threads spawned. Specifically, we proceed as follows.

First, we associate to each function a function-AD and a function-FLOP, which provide a measure of the accumulation patterns and the number of floating-point operations originating from that function. To define the function-AD, we associate a "weighted AD" to each variable updated within the function in consideration. The weighted AD of a variable is computed by multiplying the variable's AD and the number of times the variable is updated. For each variable, the number of updates is equal to the size of the loops enclosing the variable's update, in case of GPU kernels multiplied by the number of threads spawned by the kernel. This allows accounting for accumulation patterns over threads, and handling CPU and GPU functions uniformly. The function-AD is then computed by summing the weighted AD of all its variables and multiplying the result by the number of times the function is invoked. For example, if a GPU kernel is invoked N times, it spawns T threads, and it updates a single variable with AD=2 within a loop of size L, the function-AD will be equal to N*T*L*2. Similarly, we define the function-FLOP as the product of the number of floating-point operations performed by a function and the number of times that function is invoked. Again, in computing the number of floating-point operations performed, we consider the size of the loops within the function and, for GPU kernels, the number of threads spawned. Finally, we sort the functions by decreasing function-AD (i.e., variables updated in functions with lower function-AD will be tuned down first). If two functions have the same function-AD, we prioritize the one with lower function-FLOP.

4.2.2.4 Phase 2: Determining the variables' tuning order

Once the functions in the program under consideration have been sorted, the variables' tuning order can be determined. Local variables are tuned function-by-function following the defined function order; variables updated by multiple functions are tuned globally. The tuning order of variables is determined taking into consideration the following code properties: accumulation patterns, loop sizes, number of occurrences of the variables, operations cost, CPU-GPU data transfers, shared memory use, and kernels' configuration.

General strategy – First, we sort the variables based on the product of their AD and number of updates. This way, variables less likely to incur error propagation are tuned down first, while variables storing the result of large accumulation patterns (e.g., global summations) are tuned down later and, consequently, kept at high precision for more tuning iterations. If two variables have the same weighted AD, they are sorted based on their number of occurrences in the code, so that variables used less frequently and tuned down first. If two variables have also the same number of occurrences, then the tuner aims to maximize performance by reducing the cost of computation. To this end, it prioritizes tuning down variables involved in more costly operations (as reported in Table 4.1 and Figure 4.1).

Handling global variables – Variables used by multiple kernels and involved in CPU-GPU data transfers deserve more considerations. We allow users to annotate variables transferred between CPU and GPU that contain input and output data. We break down the following cases. (i) *Read-only variables transferred from CPU to GPU* – Adding precision to input variables does not improve accuracy. However, maintaining variables at low precision can reduce the data transfer cost. Therefore, input variables are the first to be tuned down, and other read/only variables transferred from CPU to GPU are tuned down next. (ii) *Result variables transferred from GPU to CPU* – To preserve accuracy, the result of a program should be kept in high precision. Therefore, variables in this category are the last to be tuned down. (iii) *Variables used in multiple kernels and remaining read/write variables involved in CPU-GPU data transfers*– We delay the tuning for these variables just before the result variables for two reasons. First, these variables often participate in more computation and keeping them in high precision helps preserving accuracy. Second, as shown in Figure 4.1, the overhead of memory transfer for high precision is not significant for data transfer under 1000k elements.

Handling shared memory variables – As explained above, reducing shared memory utilization can increase the number of threads resident on a streaming multiprocessor, thus providing better memory latency hiding. When evaluating variables defined inside kernels, shared memory variables storing input data can be tuned down first. This can allow better performance without accuracy loss (since input data are in standard precision in any case). The autotuner can be configured to handle the remaining shared memory variables in two ways: according to the general policy, or before the other local variables. The second policy, which we call *shared-memory-first* policy, trades off performance with accuracy.

Summary – In summary, we tune down categories of variables in the following order: (1) input and read-only variables transferred from CPU to GPU, (2) local variables defined inside kernels and CPU functions, (3) variables used by multiple kernels, (4) read/write variables transferred between CPU and GPU, (5) output variables transferred from GPU to CPU. Within each category, we use the general tuning strategy summarized above.

Example – Figure 4.2 shows a sample code with four variables: `a_d`, `s_d`, `cst`, and `mtp` and a single GPU function (`kernel`). The tuning strategy operates as follows. First, `cst` is a read-only variable transferred from CPU to GPU. Thus, it is tuned down first. Second, `mtp` is the only local variable (and has `AD=1` and two occurrences). Thus, it is tuned down next. Finally,

<pre>int main () {
 cudaMemcpy(a_d);
 cudaMemcpy(s_d);
 cudaMemToSymbol(cst);
 Kernel <<<…, …>>>
  (cst, a_d, s_d);
 	
 cudaMemcpy(s_d);
 	
}</pre>
	<pre>__global__ kernel (…){
	 double mtp;
	 mtp = cst * 2.0;
	 for(i=0; i<iter; i++) {
	  a_d[tid][i] = mtp *
	   sqrt(5.2);
	  s_d[tid] +=
	   (a_d[tid][i] *2.2
	    +mtp) / 5;
	 }
	}</pre>

Figure 4.2: Demonstration of tuning plan for a CUDA sample code.

variables `s_d` and `a_d` are involved in CPU-GPU data transfers and updated by the kernel. In particular, `s_d` is the result of a global summation and has AD=`iter` (i.e., the size of the enclosing loop), while `a_d` has AD=1. Thus, `a_d` is tuned down before `s_d`. In summary, the tuning plan is as follows: function order=(`kernel`) and variable order=(`cst`, `mtp`, `a_d`, `s_d`).

4.2.2.5 Hybrid tuning optimization

Many GPU kernels have customizable kernel configurations. In other words, they are written so to generate correct results independent of the number of thread-blocks and threads spawned. In this case, the optimal thread configuration for the kernel depends on its register and shared memory utilization, which, in turns, depends on the precision of the variables used in the code. We introduce a “hybrid tuning” optimization, which allows the autotuner to dynamically select the kernel configurations, possibly leading to different settings for different code variants. In other words, hybrid tuning allows the thread configuration of the GPU kernels to be tuned along with the precision of the variables.

One way to allow dynamic thread-block setting is through the *cudaOccupancyMaxPotentialBlockSize* function available in the CUDA runtime API since version 6.5 [51]. This function provides a recommendation for setting the thread-block size of a given kernel. Our tuner inserts a call to this function in the tranformed code, and uses the recommended block size in the kernel invocation. The *cudaOccupancyMaxPotentialBlockSize* function returns a one-dimensional thread-block size; when needed, our autotuner adjusts the configuration to support two- and three-dimensional thread-blocks. However, our experiments show that the recommendation provided is not always optimal. To address this problem, our autotuner also offers an alternative adaptive thread-block selection method. When using this method, our tuner evaluates a set of candidate power-of-2 thread-block sizes. In particular, it starts from the largest power-of-2 value that is smaller than the configuration recommended by the *cudaOccupancyMaxPotentialBlockSize* function, and progressively increases the block size until it finds a better configuration or reaches 1024 threads. This adaptive selection is performed only on the high and double precision versions of the program. If a better configuration is found, the tuner uses the configuration found on the high precision code on all code variants that include high precision variables, and the one found on the double precision code on the remaining code variants. For kernels with customizable grid size, to allow full device utilization, the autotuner sets the grid size to twice the number of streaming multiprocessor available on the GPU.

4.3 Implementation

4.3.1 Autotuner pipeline

Figure 4.3 shows our autotuner pipeline. As can be seen, the input CUDA code is fed to the ROSE compiler infrastructure, which performs static analysis, invokes online profiling and generates the tuning plan. All remaining modules (including the pipeline coordination module) are implemented through python scripts. The tuning plan execution module has two jobs: transform the input code according to the tuning plan and run the transformed code. At the end of each tuning iteration, the error check modules records execution time and relative error, stores the transformed code, and, when configured to do so, it terminates the tuning process when the relative error of the

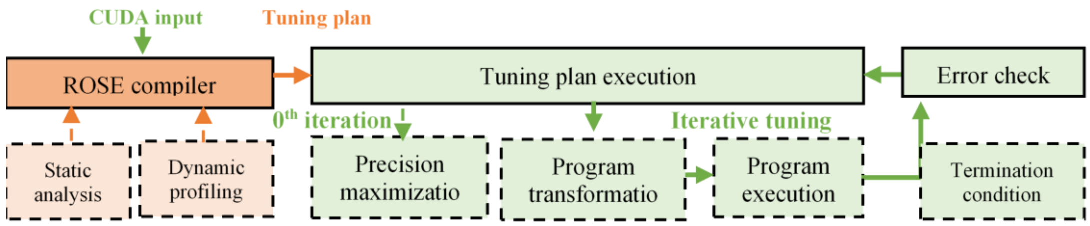


Figure 4.3: Proposed auto-tuner pipeline.

tuned program meets the prescribed error bound. Figure 4.4 summarizes the implementation of the auto tuner.

4.3.2 Implementation design

ROSE compiler module – ROSE [50] is an open source compiler infrastructure for source-to-source program transformation, and it supports C, C++ and CUDA (among other programming languages). One advantage of ROSE over LLVM is that it provides easier and higher-level tools to conduct source-level code analysis. For example, ROSE provides tools to detect the size of floating-point arrays and loops easily. However, at present handling mixed precision with ROSE is problematic, and its CUDA support is out of date (for example, it does not support the *cudaOccupancyMaxPotentialBlockSize* function). Therefore, while we use ROSE for static analysis and tuning plan generation, we perform code transformation using our own python script.

Program analysis is conducted automatically by this module and includes the following tasks: (a) identifying floating-point variables, (b) profiling to determine relative loop sizes, CPU-GPU memory transfer sizes and CUDA kernel configurations, (c) static analysis to determine general code properties (such as static loop sizes, accumulated dependencies, number of occurrence of variables) and GPU-specific tuning metrics (CPU-GPU memory transfer, kernel configuration, thread-block setting, shared memory variables and atomic operations). The module generates the tuning plan as described in Section 4.2. The overhead of program analysis and tuning plan generation is quite small; dynamic profiling is done on a single program run using low precision.

Tuning plan execution module – This module is implemented in a python script and performs the following tasks. First, it converts standard floating-point operations into *gdd_real* precision. This step generates the "golden solution" used to compute the relative error. Second, it supports hybrid tuning (it implements the required code transformations and launches the test runs

```
Input
    File: program.<cu>,
    Value: error_bound (e.g., 10^-10)
    Option: hybrid – shared mem,
            hybrid – thread-block
Output

    Executables: program-<iter*>.cu,
    program-exe,
    Metrics: {error, time, iter}
Algorithm
Run ROSE compiler analysis

// Static code analysis

for f in functions in program

 if floating point variable in f

  Insert f into func_list, creat
f_var_list container
  Mark if f is kernel or CPU function
  for (floating point variable
defined or in the function
parameters) v in f
   if (v is not device var and f is
not kernel)
    get loop size for acc.
calculation

    determine acc. dependence ad[v]
    determine num. of operations for
var flop[v]
    Record ad[v], flop[v] with
f_var_list
   else if (v is global var and f is
not kernel)
    Insert var into glb_var list and
check read/write
   else if (v is device var and f is
kernel)
    get loop size for acc.
calculation
    determine acc. dependence ad[v]
    determine num. of operations
flop[v] for var, num. of occurrences
and flop cost
    if v is in dev_var list
      Mark is_trans = 1
    if v is shared memory var
      Mark is_shared = 1
    Record ad[v], flop[v] with
f_var_list

// Dynamic profiling to determine the relative size of the loops
if (relative loop size input dependent)

  perform dynamic profiling to update LC_size,
kernel_configuratoin
// Application of the tuning rules

for f in functions in program
  sort var in f_var_list by ad And occurrences
  (execlude variable marked is_trans = 1)
  update function-AD. and function-FLOP for f
in func_list
  sort f in func_list based on {function-AD,
function-FLOP}
  sort vars marked is_trans =1 and put sorted
    list in the end of current tuning plan
  tuning plan: <func name> (variable list);…;
    <CPU-GPU transfer> (variable list)
  if hybrid – shared-memory-first:
    Move vars with is_shared =1 at the top  of
      tuning plan

  if hybrid – dynamic thread-block enabled:

    insert cudaOccupancyMaxPotentialBlockSize()
      and related codes to enable dynamic
      thread-block setting
// Execution of the tuning plan
for each tuning iteration

  if (iter = 0)

    convert all FP ops and variables to 128 bit
      gdd_real
  else

    tune single var in var_list in a function

  generate updated source code: program.cu

  run program.exe
  if (iter > 0)

    compute error E relative to iteration 0
  record computing time T
  store {E, T, iteration} in the output array
  if (E > error_bound)
    option to terminate the tuning process
```

Figure 4.4: Auto tuner implementation pseudocode.

needed by the adaptive thread-block selection method). Third, for each tuning iteration, it performs the code transformations required to tune down variables according to the tuning plan. Finally, it launches the tuned program and invokes the error check module. The tuning plan execution module dominates the tuning time since it performs multiple program runs.

Our code transformation script operates as follows. First, it locates the target variable declaration and its usages in the program. Second, it tunes the precision of the variable by modifying its declaration. Third, it implements a dependency check feature to ensure that all the usages of the modified variable are valid. Specifically, the script introduces in the code the required data conversion calls, and, if the modified variable is used as a function parameter, it updates the function parameters accordingly.

Data dependency handling – While our autotuner typically tunes a variable in each tuning iteration, there are cases where program correctness requires the tuner to handle multiple variables in a single iteration. This, for example, occurs when multiple pointer variables point to the same memory location.

Precision propagation – In mixed precision code, it is common to have situations where the operands of an arithmetic operation have different precisions, or an assignment involves variables with different precisions. In the presence of single and double precision arithmetic, most compilers support type casting without intervention. In case of an arithmetic operation in mixed precision, the compiler converts the single precision operand into double precision and the operation is performed in high precision arithmetic. In case of mixed precision assignments, the compiler converts a single precision operand into double precision before assigning it to a double precision variable, and it truncates a double precision operand before assigning it to a single precision variable. When using the DD library, precision propagation is handled as follows. For

Table 4.2: Benchmark applications characteristics

Benchmark	Domain	# of floating-point variables		Original precision	# of kernels	LOC
		total	transferred btw CPU & GPU			
SP	Linear Algebra	198	12	double	11	2193
CFD	Fluid Dynamics	51	6	double	4	673
GE	Linear Algebra	4	3	single	2	254
LUD	Linear Algebra	8	1	single	3	291
HOTSPOT	Physics Simulation	21	3	single	1	313
LAVA	Molecular Dynamics	30	10	double	1	464

arithmetic operations in mixed precision, the DD library supports type casting through overloading. For mixed precision assignments, the DD library provides primitives (*to_double* and *dd_add*) to convert *dd_real* variables into double, and vice versa. Our code transformation module invokes the proper conversion primitive as required.

4.4 Experimental Analysis

In this section we present experiments performed to validate our auto-tuner and investigate accuracy-performance tradeoffs when using mixed precision in GPU applications.

4.4.1 Experimental setup

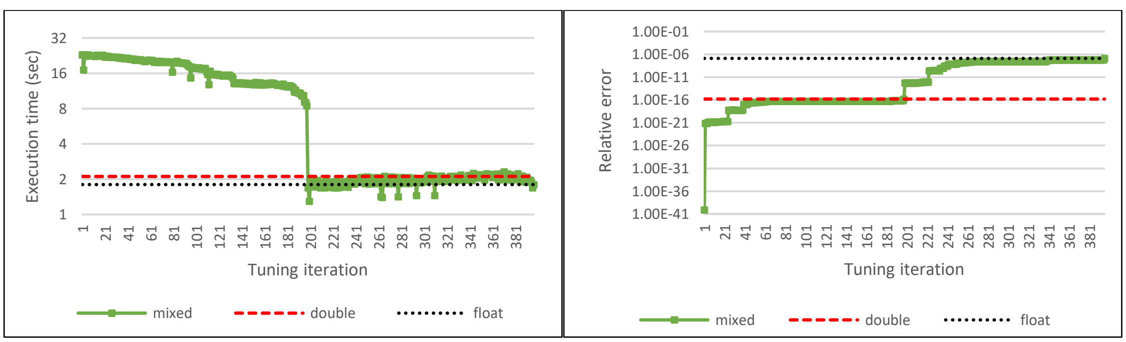
Benchmark applications – We performed experiments on six applications: five of them drawn from the Rodinia Benchmark Suite [10], namely: Gaussian elimination (GE), LU decomposition (LUD), LavaMD (LAVA), Hotspot (HOT), and Computational Fluid Dynamics (CFD); and one of them from the NAS Parallel Benchmark Suite [37]: the Scalar Penta-diagonal solver (SP). Table 4.2 summarizes the core characteristics of the benchmarks considered. These applications contain a combination of additions (and subtractions), multiplications, divisions, exponents and square roots. All the codes are implemented using CUDA. HOT, LUD and LAVA use shared memory. The thread-block size of the kernels can be dynamically configured for all applications except SP and CFD, which require a fixed kernel configuration.

Target platform and metrics – We performed all experiments on an Intel Xeon E5 processor (2.4 GHz) running Red Hat Enterprise 7.5 OS and equipped with a high-end Nvidia GPU. We compiled the code using *nvcc* with optimization level –O3. As a measure of performance, we consider the execution time of the whole application. As a measure of accuracy, we consider the relative error over a reference solution computed using the 128-bit *dd* library for all floating-point variables and operations. For those applications where the solution is an array, we report the relative error averaged over the whole array.

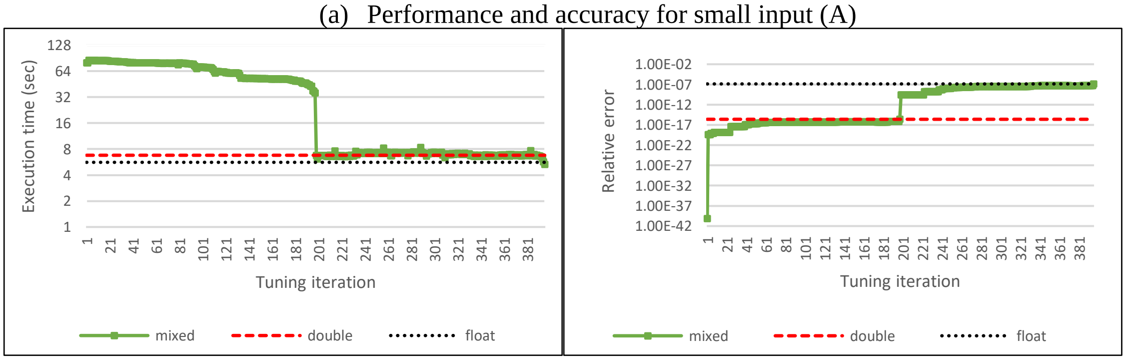
4.4.2 Autotuning results

Figure 4.5-4.10 show the performance and accuracy results reported by our auto-tuner on the applications considered. We recall that our auto-tuner generates an application-specific tuning plan and operates in multiple phases and iterations. First, it "tunes up" the whole program to provide the all-*gdd_real* (128-bit) version of the code (iteration 0). Next, in each subsequent tuning iteration (along the *x*-axis of the charts), it tunes down one of the floating-point variables of the program. Specifically, the tuner gradually tunes down the floating-point variables from *gdd_real* to double-precision arithmetic in phase 1, and from double to single precision in phase 2. This results in a number of tuning iterations equal to twice the number of floating-point variables in the application. For comparison, the charts also include the performance and accuracy results reported using the all-float and all-double versions of the program. The results of the all-*gdd_real* version of the code are those of iteration 0. For all applications, we show the results reported on two input datasets with different sizes. This allows us to validate the generated tuning plan across inputs.

The following high-level observations apply across applications. First, as expected, the mixed-precision code resulting from autotuning offers better accuracy compared with the double and float versions, and its accuracy tends to decrease as more tuning iterations are performed; in the final iteration, the accuracy matches the single-precision case as all variables have been tuned down to single-precision. Similarly, the execution time of the mixed-precision code gradually decreases as more variables are tuned down to double and single precision. Second, while the execution time of the programs increases with the dataset size, the accuracy and performance trends stay the same across datasets. Third, mixed-precision tuning allows balancing performance and accuracy. In particular, areas of plateau in the performance or accuracy trend lines allow identifying code versions that provide good performance/accuracy tradeoffs. For example, in the presence of a plateau in the accuracy trend line, the code corresponding to the rightmost iteration of the plateau allows achieving the same accuracy as the code of the previous tuning iterations



(a) Performance and accuracy for small input (A)



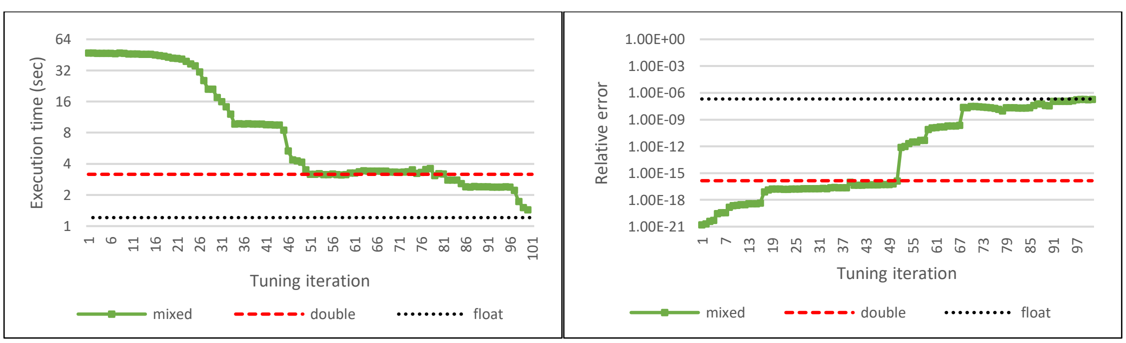
(b) Performance and accuracy for large input (B)

Figure 4.5: SP mixed-precision tuning results.

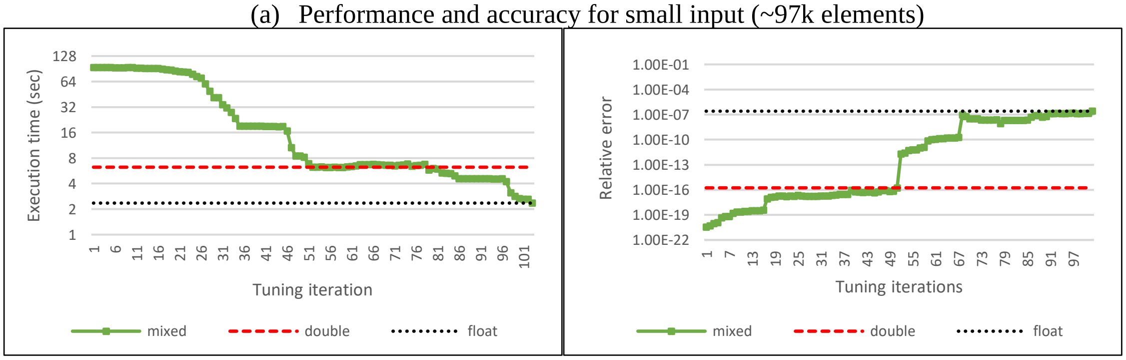
with a lower execution time. Below, we provide more details on the application-specific results and trends.

SP – Figure 4.5 shows the iteration-by-iteration tuning results for the SP benchmark. In these experiments, we use the class A and class B input datasets, which consist of 262,144 and 1,061,208 elements, respectively. SP consists of five main functions, each invoked one time, and it includes 198 floating variables and 11 CUDA kernels. The resulting tuning plan consists of 396 iterations. The last kernel tuned contains a single loop that implements a global summation. The result of this summation (transferred from GPU to CPU) is the last variable to be tuned.

We make the following observations. First, double precision provides about 8 additional decimal digits of accuracy over single precision at a negligible performance cost (about 15% increase in execution time); hence, there is little advantage in using hybrid float/double versions of the code. Second, while using *gdd_real* variables can provide up to 25 additional decimal digits of accuracy over double precision, after the first few tuning iterations the accuracy of the mixed-



(a) Performance and accuracy for small input (~97k elements)

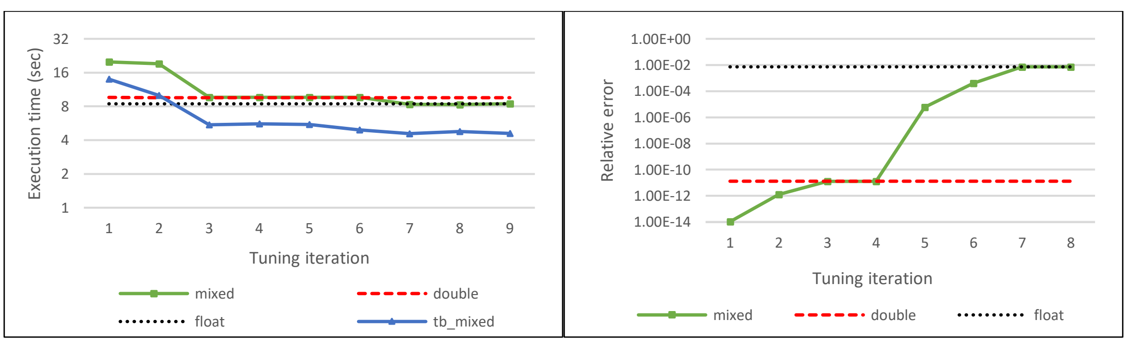


(b) Performance and accuracy for large input (~193k elements)

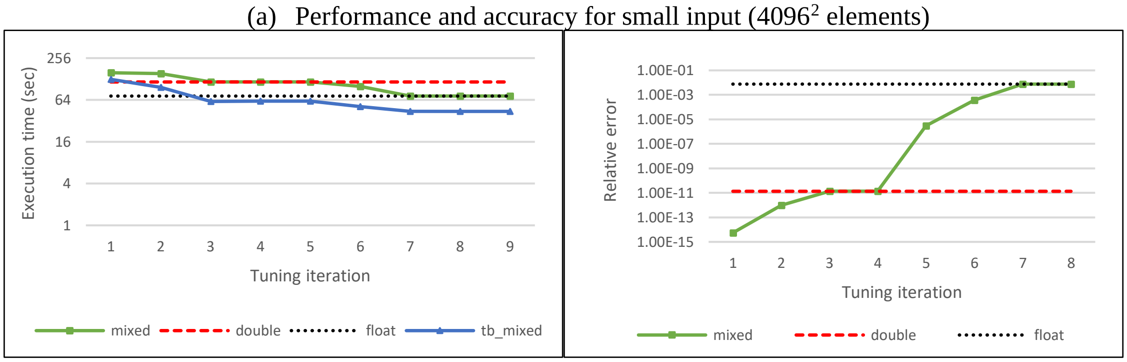
Figure 4.6: CFD mixed-precision tuning results.

precision code decreases fast while its performance increases at a significantly lower rate. After the 40th tuning iteration the accuracy difference between mixed and double precision becomes negligible, while the performance has improved only by 20%. Hence, when accuracy is a concern, the all-*gdd_real* code provides a better accuracy/performance tradeoff than the hybrid *gdd_real*/double code variants. Finally, on the small dataset the execution time of the tuned program has some oscillations (while still following the trend). In this case, the short running time of the program (1-2 seconds) makes it more sensitive to the data conversion overhead resulting from the use of mixed precision.

CFD - Figure 4.6 shows the iteration-by-iteration results for the CFD benchmark on two datasets consisting of about 97k and 193k elements. CFD has fairly complex computational patterns, large loops and a large number of floating-point operations. We make the following observations. First, using *gdd_real* in the code adds some accuracy (up to 5 decimal digits of precision) at a significance performance cost. Along the plateau between iterations 17 and 50 the



(a) Performance and accuracy for small input (4096^2 elements)

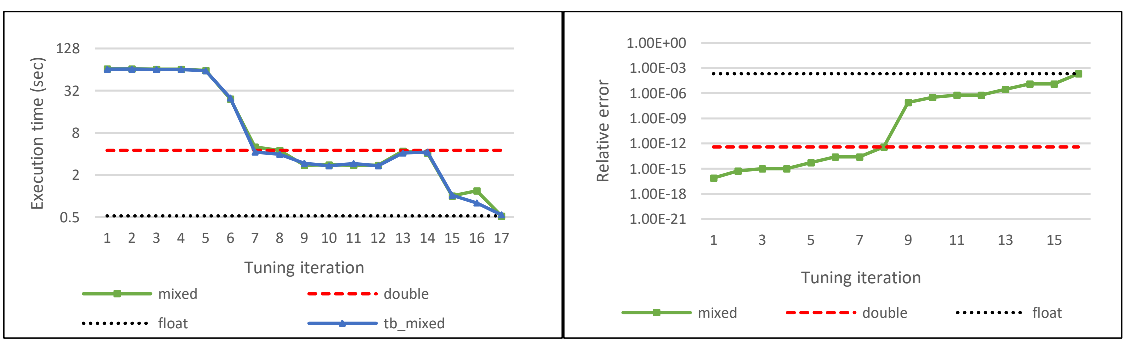


(b) Performance and accuracy for large input (8192^2 elements)

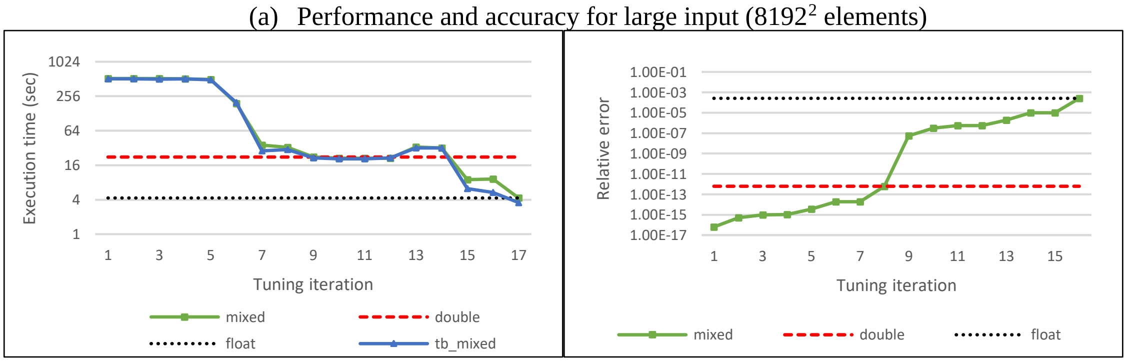
Figure 4.7: GE mixed-precision tuning results.

mixed-precision version of the code provides 1 additional decimal digit of accuracy over the double precision code and up to 10x speedup over all-*gdd_real* code. Finally, the use of mixed precision in the 2nd tuning phase (between double and single precision arithmetic) allows adding 1-6 decimal digits of precision compared to single precision while degrading performance by a factor 2x-3x.

GE and LUD – Figure 4.7 and 4.8 report the iteration-by-iteration tuning results for the GE and LUD benchmarks. As shown in Table 4.2, these are small programs with very few floating-point variables, resulting in only 8 and 16 tuning iterations. GE and LUD kernels have configurable thread-block sizes: in the charts, we also report the results from hybrid tuning, where the thread-block size is set dynamically (blue line with triangular markers). Recall that hybrid tuning does not affect accuracy. We make the following observations. First, hybrid tuning benefits the programs to different degrees. Specifically, dynamically setting the thread-block size in the tuned code allows a 2x speedup for GE, and a 20% performance gain for LUD. Second, for GE hybrid



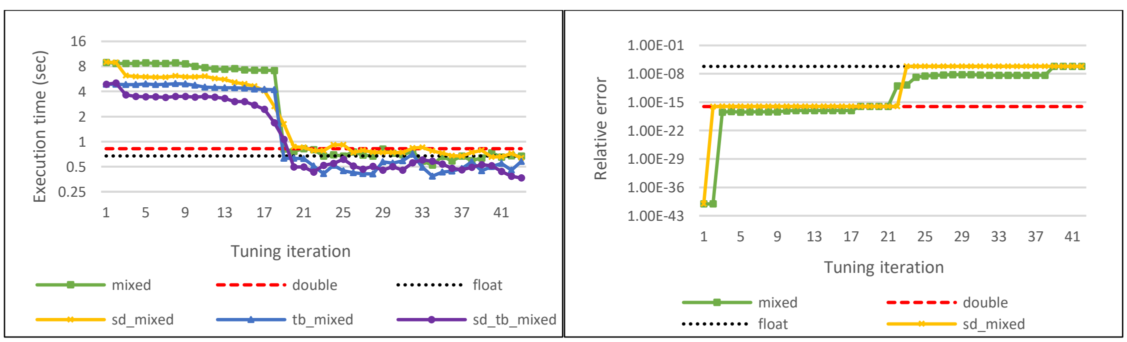
(a) Performance and accuracy for large input (8192^2 elements)



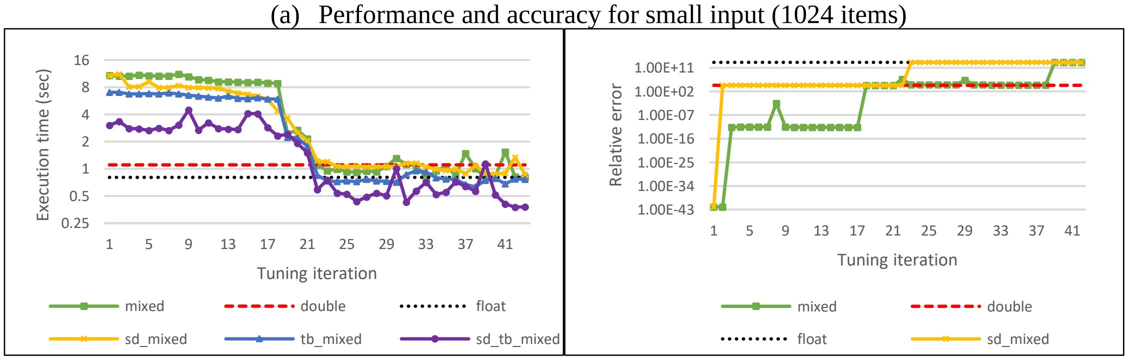
(b) Performance and accuracy for large input (16384^2 elements)

Figure 4.8: LUD mixed-precision tuning results.

tuning offers the best tradeoff between accuracy and performance, allowing 1-2 additional decimal digits of accuracy over the all-double code without performance loss. For LUD, the mixed precision program at iterations 6 and 7 (a plateau) provides 1 additional decimal digit of accuracy over the all-double version of the code without performance degradation, and 13x speedup over all-*gdd_real* version of the code. Third, for GE double precision offers performance comparable to single precision while adding about 8 decimal digits of accuracy. On the other hand, for LUD using double precision leads to a significant slowdown over single precision (by a factor ~9x and ~5x on small and large datasets, respectively). This is because LUD uses shared memory, and increased precision limits the degree of multithreading achievable by the code. Finally, we observe that the use of mixed precision in both GE and LUD in the 2nd tuning phase (between double and single precision arithmetic) leads to an accuracy gain (around 1-3 decimal digits for both GE and LUD) over the single precision code without incurring a noticeable performance cost.



(a) Performance and accuracy for small input (1024 items)

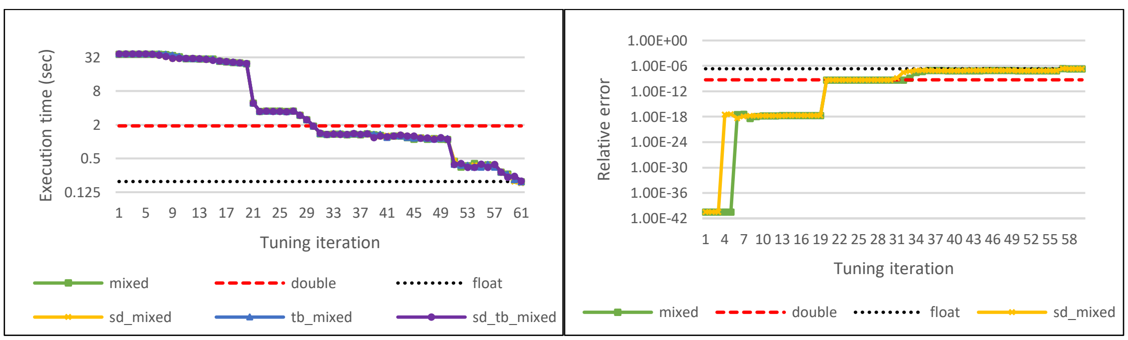


(b) Performance and accuracy for large input (8192 items)

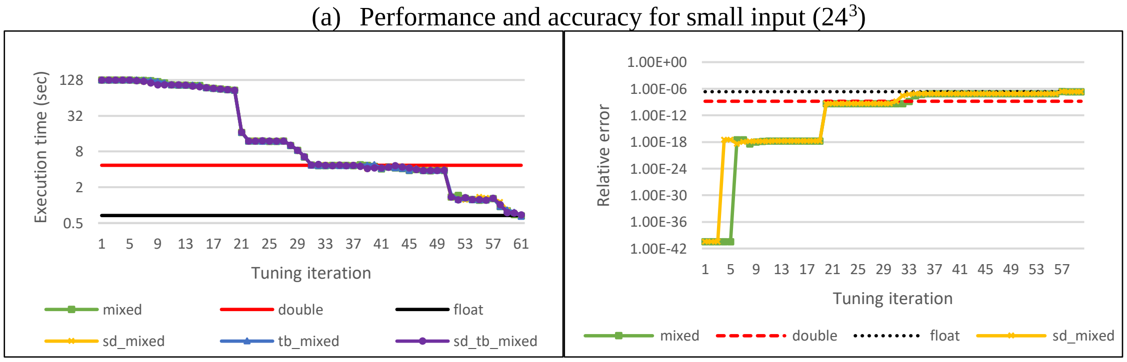
Figure 4.9: HOTSPOT mixed precision tuning results.

HOT and LAVA – Figure 4.9 and 4.10 show the tuning results for the HOT and LAVA benchmarks. In both cases, we show the results from hybrid tuning (blue line with triangular markers), from shared-memory-first policy (yellow line), and from the combined use of these two strategies (purple line). Recall that, unlike hybrid tuning, the shared-memory-first policy affects the tuning order and, as such, the resulting accuracy.

We make the following observations. First, the use of hybrid tuning, shared-memory-first policy and their combination leads to a 2x, 35% and 3.5x performance improvement, respectively. However, since HOT updates its shared memory variables often and is sensitive to their accuracy, tuning them down first has a negative impact on accuracy, especially for the large dataset. Hence, for HOT using dynamic thread-block setting alone leads to the best results. In the case of LAVA, hybrid tuning and shared-memory-first policy have a much less noticeable effect on performance (only up to 10% runtime reduction). Second, the mixed-precision version of HOT with dynamic thread-block setting between iteration 3 and iteration 18 (a plateau) provides 1 and 8 additional



(a) Performance and accuracy for small input (24^3)



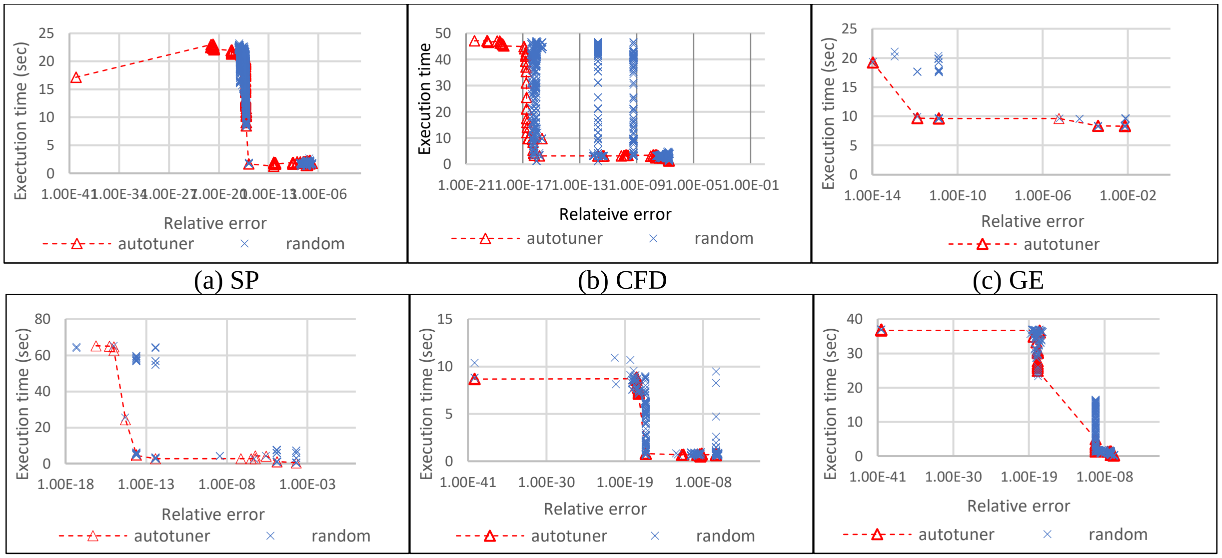
(b) Performance and accuracy for small input (36^3)

Figure 4.10: LAVA mixed precision tuning results.

decimal digits of accuracy over double precision on the small and large dataset, respectively, and about 2x performance improvement over the all-*gdd_real* version of code. LAVA results exhibit two plateaus. In iterations 6-19 hybrid tuning provides 8-9 additional decimal digit of accuracy over all-double and a 30% speedup over the all-*gdd_real* version of the code. On the large dataset, between iteration 21 and iteration 32 hybrid tuning provides 1 additional decimal digit of accuracy over double precision and up to 27x speedup over the all-*gdd_real* version of the code. Finally, we observe that the use of mixed precision in LAVA during the 2nd tuning phase (from double to single precision arithmetic) leads to an accuracy gain (additional 1 decimal digit) compared to single precision with a 2-7x performance cost. For HOT, double precision provides 9 additional digits of accuracy over single precision with a small performance cost (up to 25% runtime increase), and mixed precision with dynamic thread-block setting allows double accuracy while eliminating that performance cost.

4.5 Validation of tuning strategy

Given a program with N floating-point variables and 3 candidate precisions (single, double and *gdd_real* in our study), an exhaustive search over all possible precision settings for the N variables would require evaluating 3^N configurations. Our tuner uses compiler analysis to limit the size of the tuning space to $2N$ configurations. Here, we want to evaluate the effectiveness of our tuning strategy in finding good configurations. In other words, we want to find out if our strategy misses configurations that would provide better performance/accuracy results. Since performing an exhaustive search is unfeasible in a reasonable time, we compare our strategy with a randomized approach. In particular, we randomly generate a set of configurations ~10-20 times larger than the one originated by our strategy. Specifically, we explore 4000, 2000, 80, 160, 840 and 1200 configurations for SP, CFD, GE, LUD, HOTSPOT and LAVA, respectively.

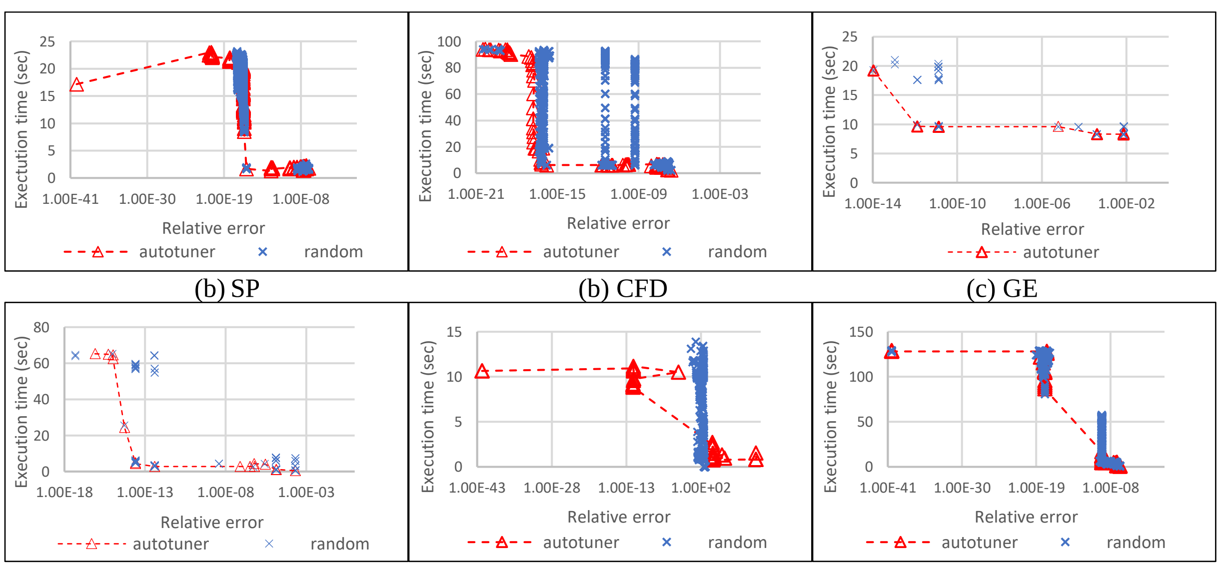


(a) SP (b) CFD (c) GE

(d) LUD (e) HOTSPOT (f) LAVA

Figure 4.11: Execution time and relative error: mixed-precision autotuning vs randomized tuning (small datasets).

Figure 4.11 and 4.12 plots the results on the small and large datasets, respectivly: we observed similar trends on the large ones. As can be seen, almost all the randomly generated configurations sit either on the trend line of our autotuner, or above it, or to the right of it. This means that they offer either the same performance and accuracy, or worse performance with the



(b) SP (b) CFD (c) GE

(d) LUD (e) HOTSPOT (f) LAVA

Figure 4.12: Execution time and relative error: mixed-precision autotuning vs randomized tuning (large datasets).

same accuracy, or worse accuracy with the same performance. There are a couple of exceptions (although not noticeably better than our tuning configurations): these are configurations benefitting from limited data conversion and type casting overheads.

4.6 Conclusions

In conclusion, we have proposed a mixed-precision autotuner that balances performance and accuracy of GPU applications. Our tuning strategy leverages code properties and GPU-specific considerations to narrow down the search space. Our tuner operates at the level of the source code, allowing users to further optimize the code manually. Our experimental results collected on benchmark applications with various code complexities show performance-accuracy tradeoffs for these applications and the effectiveness of our tool in identifying representative tuning points. Our work can be extended to other arithmetic precisions, application domains (e.g., deep learning) and platforms (e.g., FPGA).

CHAPTER 5

Conclusions and future directions

In summary, the goal of this dissertation has been to provide loop-aware and architecture-aware auto-tuners for floating-point applications running on CPU and GPU. Our proposed auto-tuners use mixed-precision to trade off performance and accuracy, and provide tuning results robust to input variability. While performing mixed-precision autotuning, we have leveraged static analysis and dynamic profiling to explore the full search space potentially consisting of all possible precision configurations for the variables in the program under consideration. The proposed tuning strategy has proven to effectively reduce the search space and offer good performance and accuracy tradeoffs on CPU and GPU with a mix of different benchmark applications. Different from related work, we have focused our study on computational patterns that affect performance and accuracy, and that are prone to error propagation. In addition, we have proposed a tuning methodology that reduces the exploration space on GPU by taking into consideration GPU-specific aspects, such as CPU-GPU data transfers, degree of multithreading, shared memory and atomic operation handling.

Our work can be extended to other platforms, such as FPGA and mobile devices. In addition, our mixed-precision work is not restricted to the scientific computing domain, but can be extended to other domains that can benefit from trading off accuracy and performance. For example, deep learning on mobile system requires low power consumption, and using mixed-precision along with half or lower precision can allow achieving good performance/accuracy tradeoffs, leading to a better energy consumption.

REFERENCES

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

[2] D. Bailey and D. Broadhurst, "Parallel integer relation detection: Techniques and applications," *Mathematics of Computation,* vol. 70, pp. 1719-1736, 2001.

[3] P. H. Hauschildt and E. Baron, "Numerical solution of the expanding stellar atmosphere problem," *Journal of Computational and Applied Mathematics,* vol. 109, pp. 41-63, 1999.

[4] D. H. Bailey, "High-precision computation: Mathematical physics and dynamics," *Applied Mathematics and Computation,* vol. 2018, pp. 10106-10121, June 2012 2012.

[5] *Nvidia GPU application catalog*. Available: https://www.nvidia.com/en-us/gpu-accelerated-applications/

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

[7] 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.

[8] Y. Hida, X. S. Li, and D. H. Bailey, "Algorithms for quad-double precision floating point arithmetic," presented at the Proceedings 15th IEEE Symposium on Computer Arithmetic. ARITH-15 2001, 2001.

[9] 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.

[10] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," presented at the Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC), 2009.

[11] C. Rubio-González, N. Cuong, N. Hong Diep, J. Demmel, W. Kahan, K. Sen, *et al.*, "Precimonious: Tuning assistant for floating-point precision," presented at the 2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2013.

[12] 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.

[13] M. O. Lam and J. K. Hollingsworth, "Fine-grained floating-point precision analysis," *The International Journal of High Performance Computing Applications,* vol. 32, pp. 231-245, 2018.

[14] C. Rubio-González, C. Nguyen, B. Mehne, K. Sen, J. Demmel, W. Kahan, *et al.*, "Floating-point precision tuning using blame analysis," presented at the Proceedings of the 38th International Conference on Software Engineering, 2016.

[15] H. Guo and C. Rubio-González, "Exploiting community structure for floating-point precision tuning," presented at the Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2018.

[16] W.-F. Chiang, M. Baranowski, I. Briggs, A. Solovyev, G. Gopalakrishnan, and Z. Rakamarić, "Rigorous floating-point mixed-precision tuning," presented at the

Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, 2017.
[17] H. Menon, M. O. Lam, D. Osei-Kuffuor, M. Schordan, S. Lloyd, K. Mohror, *et al.*, "ADAPT: algorithmic differentiation applied to floating-point precision tuning," presented at the SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, 2018.
[18] P. V. Kotipalli, R. Singh, P. Wood, I. Laguna, and S. Bagchi, "AMPT-GA: automatic mixed precision floating point tuning for GPU applications," presented at the Proceedings of the ACM International Conference on Supercomputing, 2019.
[19] I. Laguna, P. C. Wood, R. Singh, and S. Bagchi, "GPUMixer: Performance-driven floating-point tuning for GPU scientific applications," presented at the International Conference on High Performance Computing, 2019.
[20] R. Gu, P. Beata, and M. Becchi, "A Loop-aware Autotuner for High-Precision Floating-point Applications," presented at the IEEE International Symposium on Performance Analysis of Systems and Software, Boston, 2020.
[21] *Popular GPU-Accelerated Applications Catalog*. Available: http://www.nvidia.com/docs/IO/123576/nv-applications-catalog-lowres.pdf
[22] D. H. Bailey, "High-precision Floating-point Arithmetic in Scientific Computing," presented at the IEEE Computing in Science and Engineering, 2005.
[23] D. M. Priest, "Algorithms for arbitrary precision floating point arithmetic," presented at the [1991] Proceedings 10th IEEE Symposium on Computer Arithmetic, 1991.
[24] J. Demmel and Y. Hida, "Accurate and efficient floating point summation," *SIAM Journal on Scientific Computing,* vol. 25, pp. 1214-1248, 2004.
[25] 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, ed, February 2009, pp. 560–564.
[26] J. Demmel and H. D. Nguyen, "Fast Reproducible Floating-Point Summation," presented at the 2013 IEEE 21st Symposium on Computer Arithmetic, 2013.
[27] M. M. P. Langlois, and L. Thevenoux, "Accuracy Versus´ Time: A Case Study with Summation Algorithms," presented at the Pro-ceedings of the 4th International Workshop on Parallel and Symbolic Computation (PASCO ’10), 2010.
[28] P. Ahrens, H. D. Nguyen, and J. Demmel, "Efficient Reproducible Floating Point Summation and BLAS," EECS Department, UC Berkeley Technical Report No. UCB/EECS-2016-121, 2016.
[29] F. De Dinechin, C. Lauter, and G. Melquiond, "Certifying the floating-point implementation of an elementary function using Gappa," *IEEE Transactions on Computers,* vol. 60, pp. 242-253, 2011.
[30] E. Darulova and V. Kuncak, "Sound compilation of reals," presented at the Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2014.
[31] 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.

[32] 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, 2010.
[33] T. Nakayama and D. Takahashi, "Implementation of multiple-precision floating-point arithmetic library for GPU computing," presented at the Proceedings of the 23rd iasted international conference on parallel and distributed computing and systems, 2011.
[34] A. Thall, "Extended-precision floating-point numbers for GPU computation," presented at the ACM SIGGRAPH 2006 Research posters, Boston, Massachusetts, 2006.
[35] I. C. Society, "IEEE Standard for Floating-Point Arithmetic," in *IEEE Standard 754-2008*, ed, Aug 2008.
[36] Y. H. David H. Bailey, Xiaoye S. Li and Brandon Thompson, "ARPREC: An Arbitrary Precision Computation Package," Lawrence Berkeley National Lab 2002.
[37] D. Bailey, T. Harris, W. Saphir, R. Van Der Wijngaart, A. Woo, and M. Yarrow, "The NAS parallel benchmarks 2.0," Technical Report NAS-95-020, NASA Ames Research Center1995.
[38] T. J. Dekker, "A floating-point technique for extending the available precision," *Numerische Mathematik,* vol. 18, pp. 224-242, 1971.
[39] H. Wong, M. M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU microarchitecture through microbenchmarking," presented at the 2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS), 2010.
[40] *Floating Point and IEEE 754 Compliance for NVIDIA GPUs*. Available: http://docs.nvidia.com/cuda/floating-point/index.html
[41] G. Sawaya, M. Bentley, I. Briggs, G. Gopalakrishnan, and D. H. Ahn, "FLiT: Cross-platform floating-point result-consistency tester and workload," presented at the Workload Characterization (IISWC), 2017 IEEE International Symposium on, 2017.
[42] 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.
[43] E. Darulova and V. Kuncak, "Towards a Compiler for Reals," *ACM Trans. Program. Lang. Syst.,* vol. 39, pp. 1-28, 2017.
[44] C. Lattner and V. Adve, "The LLVM compiler framework and infrastructure tutorial," presented at the International Workshop on Languages and Compilers for Parallel Computing, 2004.
[45] Y. Hida, X. S. Li, and D. H. Bailey, "Quad-double arithmetic: Algorithms, implementation, and application," presented at the 15th IEEE Symposium on Computer Arithmetic, 2000.
[46] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia*, et al.*, "Mixed precision training," *arXiv preprint arXiv:1710.03740,* 2017.
[47] T. Grützmacher and H. Anzt, "A Modular Precision Format for Decoupling Arithmetic Format and Storage Format," presented at the European Conference on Parallel Processing, Springer, Cham, 2018.
[48] H. Anzt, J. Dongarra, G. Flegar, N. J. Higham, and E. S. Quintana‐Ortí, "Adaptive precision in block‐Jacobi preconditioning for iterative sparse linear system solvers," *Concurrency and Computation: Practice and Experience,* vol. 31, p. e4460, 2019.

[49] T. Grützmacher, T. Cojean, G. Flegar, H. Anzt, and E. S. Quintana-Ortí, "Acceleration of PageRank with Customized Precision Based on Mantissa Segmentation," *ACM Trans. Parallel Comput.*, vol. 7, p. Article 4, 2020.
[50] D. Quinlan, C. Liao, J. Too, R. P. Matzke, M. Schordan, and P. Lin, "ROSE compiler infrastructure," ed, 2012.
[51] *CUDA Toolkit Documentation*. Available: https://docs.nvidia.com/cuda/