A study on linear algebra operations using extended precision floating-point arithmetic on GPUs

Graduate School of Systems and Information Engineering
University of Tsukuba

November 2013

Daichi Mukunoki

Contents

1 Introduction **1**

1.1 Precision of Floating-Point Arithmetic 2

1.1.1 IEEE Standard for Floating-Point Arithmetic 2

1.1.2 Extended Precision Floating-Point Arithmetic 3

1.2 GPU Computing 4

1.2.1 Performance Characteristics 5

1.2.2 CUDA 5

1.3 Overview of the Thesis 8

1.3.1 Scope of the Study 8

1.3.2 Contributions 9

1.3.3 Organization of the Thesis 9

2 Triple- and Quadruple-Precision BLAS Subroutines on GPUs **12**

2.1 Introduction 12

2.2 Quadruple-Precision Floating-Point Operations 13

2.2.1 DD-type Quadruple-Precision Floating-Point Format 14

2.2.2 DD-type Quadruple-Precision Floating-Point Arithmetic 15

2.3 Triple-Precision Floating-Point Operations 18

2.3.1 D+S-type Triple-Precision Floating-Point Format 19

2.3.2 D+S-type Triple-Precision Floating-Point Arithmetic 19

2.3.3 Computation of D+S-type Values using DD Arithmetic 21

2.3.4 D+I-type Triple-Precision Floating-Point Format 23

2.4 Implementation of Triple- and Quadruple-Precision BLAS Subroutines on GPUs 25

2.4.1 DD Arithmetic Functions 26

2.4.2 Implementation of BLAS Kernels 27

2.4.3 Data Structures of Triple- and Quadruple-Precision Value Arrays in Global Memory 28

2.5 Performance Prediction 30
2.5.1 Theoretical Peak Performance on DD Arithmetic 30
2.5.2 Performance Prediction using the Bytes/Flop and Bytes/DDFlop ratios 32
2.6 Performance Evaluation 33
2.6.1 Evaluation Methods 34
2.6.2 Performance Comparison of Double-, Triple- and Quadruple-Precision BLAS Subroutines 34
2.6.3 AoS Layout vs. SoA Layout 39
2.6.4 D+S Arithmetic v.s. DD Arithmetic 39
2.6.5 Accuracy Evaluation 40
2.7 Related Work 41
2.8 Conclusion 42

3 Optimization of Sparse Matrix-vector Multiplication on NVIDIA Kepler Architecture GPUs 44
3.1 Introduction 44
3.2 Related Work 46
3.3 Kepler architecture GPUs 47
3.4 Implementation 48
3.4.1 48KB Read-only Data Cache 48
3.4.2 Avoid Outermost Loop 49
3.4.3 Shuffle Instruction 52
3.5 Performance Evaluation 52
3.5.1 Evaluation Methods 52
3.5.2 Result 54
3.6 Conclusion 59

4 Krylov Subspace Methods using Quadruple-Precision Arithmetic on GPUs 61
4.1 Introduction 61
4.2 Related Work 64
4.3 Implementation 65
4.3.1 Overview of Quadruple-Precision Versions 65
4.3.2 Implementation of CG and BiCGStab Methods on GPUs . . . 66
4.4 Experimental Results 67
4.4.1 Unpreconditioned Methods 67
4.4.2 Cases with Preconditioning 73

4.5 Conclusion 77

5 Conclusion 79
5.1 Summary 79
5.2 Future Work 82

Bibliography 84

A List of Publications 92

Listings

2.1 Conversion from DD-type to D+I-type 25
2.2 Conversion from D+I-type to DD-type 26
2.3 Implementation of QuadMul 27
2.4 Triple-precision AXPY using D+I-type format 28
3.1 Host code of SpMV 49
3.2 Kernel code of SpMV for the Fermi architecture 50
3.3 Kernel code of SpMV for the Kepler architecture 51

List of Figures

1.1 GPU architecture (NVIDIA Fermi architecture) 6
1.2 CUDA device and thread hierarchy 7

2.1 Quadruple-precision floating-point format on DD arithmetic 14
2.2 Concept of DD arithmetic 15
2.3 D+S-type triple-precision floating-point format 19
2.4 Triple-precision operations using DD arithmetic on GPUs 23
2.5 D+I-type triple-precision floating-point format 24
2.6 GEMV kernel 29
2.7 GEMM kernel 30
2.8 Array of Structures (AoS) layout and Structure of Arrays (SoA) layout on D+S-type values 31
2.9 Performance of triple- and quadruple-precision AXPY (DDFlops: DD-type floating-point operations per second) 36
2.10 Relative execution time of AXPY (value is the multiple of the execution time of the double-precision subroutine) 36
2.11 Performance of triple- and quadruple-precision GEMV (DDFlops: DD-type floating-point operations per second) 37
2.12 Relative execution time of GEMV (value is the multiple of the execution time of the double-precision subroutine) 37
2.13 Performance of triple- and quadruple-precision GEMM (DDFlops: DD-type floating-point operations per second) 38
2.14 Relative execution time of GEMM (value is the multiple of the execution time of the double-precision subroutine) 38
2.15 Performance comparison of AoS (Array-of-Structures) and SoA (Structure-of-Arrays) layouts on D+S-type triple-precision AXPY 39
2.16 Performance comparison of D+S and DD arithmetic on GEMM . . . 40

3.1 CRS format 45

3.2 Thread mapping for the cases of NT=1, 2 and 4 on the CRS-vector method . . . 47
3.3 Flops performance for the cases with the 4 largest and 4 smallest speedup of Ver. A (only using 48KB read-only data cache) to Ver. Fermi . . . 54
3.4 Flops performance for the cases with the 4 largest and 4 smallest speedup of Ver. B (only avoiding outermost loop) to Ver. Fermi . . . 55
3.5 Flops performance for the cases with the 4 largest and 4 smallest speedup of Ver. C (only using shuffle instruction) to Ver. Fermi . . . 56
3.6 Flops performance for the cases with the 4 largest and 4 smallest speedup of Ver. Kepler to Ver. Fermi . . . 57
3.7 Flops performance for the cases with the 4 largest and 4 smallest speedup of Ver. Kepler to cuSPARSE's subroutine . . . 58

4.1 Preconditioned CG method . . . 62
4.2 Preconditioned BiCGStab method . . . 63
4.3 Accelerating iterative methods using high precision arithmetic . . . 64

List of Tables

1.1 Binary floating-point basic formats defined in IEEE 754-2008 3

2.1 Number of double-precision floating-point instructions and Flops for quadruple-precision multiply-add 18

2.2 Floating-point instruction and cycle counts for D+S and DD arithmetic on Tesla M2050 22

2.3 Bytes/Flop and Bytes/DDFlop for BLAS subroutine 34

2.4 Evaluation environment 34

2.5 Relative error to octuple-precision result (input: uniform random numbers in the range of 0 to 1 in double-precision) 41

3.1 Evaluation Environment 53

3.2 Properties of the matrices shown in Figure 3.3 54

3.3 Properties of the matrices shown in Figure 3.4 55

3.4 Properties of the matrices shown in Figure 3.5 56

3.5 Properties of the matrices shown in Figure 3.6 57

3.6 Properties of the matrices shown in Figure 3.7 58

4.1 Evaluation environment 67

4.2 The number of problems which could be solved using quadruple-precision (QP) versions with stopping criterion: $||r||_2/||r_0||_2 \leq \epsilon$ and satisfying the true relative residual: $||b - Ax||_2/||b||_2 \leq \epsilon$ 68

4.3 Cases which can be solved using both quadruple-precision (QP) and double-precision (DP) versions with stopping criterion: $||r||_2/||r_0||_2 \leq 10^{-8}$ and satisfying the true relative residual (TRR): $||b-Ax||_2/||b||_2 \leq 10^{-8}$ but QP versions are faster 69

4.4 Cases which can be solved using quadruple-precision (QP) versions with stopping criterion: $||r||_2/||r_0||_2 \leq 10^{-8}$ and satisfying the true relative residual (TRR): $||b - Ax||_2/||b||_2 \leq 10^{-8}$ but which do not converge when using double-precision (DP) versions with the stopping criterion within the maximum number of iterations of 30,000 70
4.5 Cases which can be solved using quadruple-precision (QP) versions and converged by double-precision (DP) versions with stopping criterion: $||r||_2/||r_0||_2 \leq 10^{-8}$ but not satisfying the true relative residual (TRR): $||b - Ax||_2/||b||_2 \leq 10^{-8}$ when using DP versions 71
4.6 The 2 largest and 2 smallest relative execution times per iteration (QP/DP ratio) 72
4.7 The number of problems which can be solved using preconditioned quadruple-precision (QP) versions with stopping criterion: $||r||_2/||r_0||_2 \leq \epsilon$ and satisfying the true relative residual (TRR): $||b - Ax||_2/||b||_2 \leq \epsilon$ 74
4.8 Cases with preconditioning which can be solved using both quadruple-precision (QP) and double-precision (DP) versions with stopping criterion: $||r||_2/||r_0||_2 \leq 10^{-8}$ and satisfying the true relative residual (TRR): $||b - Ax||_2/||b||_2 \leq 10^{-8}$ but QP versions are faster 75
4.9 Cases with preconditioning which can be solved using quadruple-precision (QP) versions and converged by double-precision (DP) versions with stopping criterion: $||r||_2/||r_0||_2 \leq 10^{-8}$ but not satisfying the true relative residual (TRR): $||b - Ax||_2/||b||_2 \leq 10^{-8}$ when using DP versions 76

Abstract

Most computational tasks in scientific and engineering calculations are linear algebraic operations, which compute vectors and matrices, such as the solving of systems of linear equations. On computers, real numbers are represented using floating-point numbers and are computed with floating-point arithmetic operations. Floating-point operations have rounding errors and these errors may become a critical issue for some applications. With the advances realized in computational science, there is a need for more accurate computation, especially in large-scale and long-term simulations. In such applications the accumulation of numerical errors may lead to even more serious problems in the future. Therefore we need to improve the accuracy and precision in floating-point operations.

In this thesis, I will describe the implementation, performance, and effectiveness of linear algebraic operations using extended precision floating-point arithmetic on Graphics Processing Units (GPUs). Although GPUs are specialized hardware accelerators that are designed to perform graphics processing, in recent years GPUs have become capable of performing general purpose computational operations that were traditionally handled by CPUs. As a result, General Purpose computing on GPUs (GPGPUs) has been a major topic of research in the HPC area.

I will firstly describe the implementation and performance of triple- and quadruple-precision Basic Linear Algebra Subprograms (BLAS) subroutines: AXPY, GEMV and GEMM on GPUs. Quadruple-precision operations are performed using Double-Double (DD) arithmetic which is a method for performing quadruple-precision floating-point arithmetic in software. On the other hand, I am proposing two new triple-precision floating-point formats, Double+Single (D+S)-type and Double+Int (D+I)-type formats, and a method of computing these values on GPUs. I will show the performance comparison of double-, triple-, and quadruple-precision subroutines on an NVIDIA Tesla M2050 Fermi architecture GPU. Since the GPU has relatively high floating-point performance compared to the memory bandwidth, the performance of triple- and quadruple-precision AXPY and GEMV are memory bound. Therefore, their execution times of triple- and quadruple-precision subroutines are close to 1.5

and 2 times more than that of double-precision subroutines, respectively.

Next I will describe the application of using extended precision arithmetic for sparse linear algebra on GPUs. To implement fast sparse matrix operations on GPUs, I will present techniques for optimizing Sparse Matrix-Vector multiplication (SpMV) for the CRS format on NVIDIA Kepler architecture GPUs. The proposed implementation is based on an existing method proposed for Fermi architecture, an earlier generation of the Kepler architecture. My proposed implementation takes advantage of three new features introduced in Kepler: a 48KB read-only data cache, shuffle instructions and expansion of the MaxGridDimX. On the Tesla K20 Kepler architecture GPU, my proposed implementation achieved better double-precision performance than implementations optimized for the previous generation of GPU.

Finally, I will describe the implementation and performance evaluation of Krylov subspace methods using quadruple-precision floating-point arithmetic on GPUs. The convergence of the Krylov subspace methods, which are iterative methods for solving linear systems, is significantly affected by rounding errors and there are cases where reducing rounding errors with extended precision arithmetic causes the algorithm to converge more quickly. I implemented the CG and BiCGStab methods, which are Krylov subspace methods, using quadruple-precision floating-point arithmetic, and compares the performance to the standard double-precision implementations. On unpreconditioned methods, the use of quadruple-precision arithmetic required approximately 1.11–2.20 times more execution time than that of the double-precision versions for one iteration. On the other hand, the quadruple-precision iteration time for methods with double-precision incomplete LU preconditioning is only slightly more than that of double-precision. I will show cases where the quadruple-precision version can reach a solution faster than the double-precision version.

Chapter 1

Introduction

Today, numerical analysis techniques are widely used in scientific and engineering fields, such as astronomy, weather forecasting, molecular modeling, and industrial design. Computational tasks in these fields typically require a large number of computations and a large amount of memory; supercomputers are used to perform such computations. Therefore, computer science, especially the high performance computing (HPC) area, plays a key role in developing hardware and software for supercomputer computation.

Most computational tasks in scientific and engineering calculations are linear algebraic operations, which compute vectors and matrices such as the solving of systems of linear equations. On computers, real numbers are represented using floating-point numbers and are computed with floating-point arithmetic operations. In fact, the LINPACK benchmark [1], which is used to rank the world's top supercomputers in the TOP500 list [2], performs dense linear algebraic operations using floating-point operations, and its performance is evaluated using the "Flops" value, which represents the number of floating-point operations performed per second.

Floating-point operations are required not only for scientific and engineering computations, but also for multimedia processing; most processors currently provide floating-point support in hardware. Floating-point numbers are stored with a fixed number of significand and exponent digits, and the numbers are computed with finite-precision floating-point arithmetic operations. Thus, floating-point numbers cannot always precisely represent real numbers, and the result of floating-point operations may include numerical errors, such as loss of significance and rounding errors.

These errors may become a critical issue for some applications, and in some cases, the usual hardware floating-point operation precision is insufficient; therefore, improved accuracy and precision in floating-point operations is required. For

example, it is known that the convergence of Krylov subspace methods, which are iterative methods for solving linear systems, is significantly affected by rounding errors; there have also been cases where the reduction of rounding errors with higher precision floating-point arithmetic caused the algorithm to converge more quickly [3]. Moreover, with the advances realized in computational science, there is a need for more accurate computation, especially in large-scale and long-term simulations, and the accumulation of numerical errors may lead to even more serious problems in the future [4].

Currently, most modern processors support 64-bit floating-point operations with a 52-bit significand. I will focus on the demand for extending significand precision in linear algebraic operations by examining the implementation, performance, and effectiveness of linear algebraic operations using such extended precision floating-point arithmetic on Graphics Processing Units (GPUs). Although GPUs are specialized hardware accelerators for CPUs, which are designed to perform graphics processing, they have recently enabled the performance of general purpose computations that were traditionally handled by CPUs. As a result, General Purpose computing on GPUs (GPGPUs) has been a major topic of research in the HPC area. It is believed that the study of linear algebraic operations using extended precision floating-point arithmetic on GPUs is important for future use.

The remainder of this chapter gives the background knowledge required to understand the following chapters and an overview of this thesis.

1.1 Precision of Floating-Point Arithmetic

1.1.1 IEEE Standard for Floating-Point Arithmetic

Floating-point formats and rounding are standardized by the Institute of Electrical and Electronics Engineers (IEEE) as the IEEE Standard for Floating-Point Arithmetic (IEEE 754). The first version, which was adopted in 1985 (IEEE 754-1985), defined the well-known "single-" and "double-" precision binary floating-point formats. The latest version, IEEE 754-2008 [5], published in 2008, defines four types of binary floating-point formats as shown in Table 1.1. Currently, most modern processors such as the x86 and GPUs have hardware support for the IEEE's binary floating-point formats of single-precision or both single- and double-precision, and its arithmetic operations. The "binary16" half-precision and "binary128" quadruple-precision formats were officially added into IEEE 754-2008. With the exception of binary16, the remaining formats are basic and can be used for arithmetic operations, whereas binary16 is not a basic format and is intended only for storage. The for-

Table 1.1: Binary floating-point basic formats defined in IEEE 754-2008

Name	Common name	Exponent	Significand	Total
binary16	Half-precision	5 bits	10+1 bits	16 bits
binary32	Single-precision	8 bits	23+1 bits	32 bits
binary64	Double-precision	11 bits	52+1 bits	64 bits
binary128	Quadruple-precision	15 bits	112+1 bits	128 bits

mats have one sign bit, the exponent bits, and the significand bits with the extra bit shown as "+1" in Table 1.1 of the explicitly stored width, because a floating point number is normalized such that the most significant bit is one. In addition to the binary floating-point formats, IEEE 754-2008 now supports decimal floating-point formats intended for applications that are required to emulate decimal rounding.

Moreover, IEEE 754-2008 specifies extended- and extendable-precision formats. The IEEE's extended precision format is recommended for extending the precisions used for arithmetic beyond the binary floating-point basic formats with both wider significand precision and exponent range, and both precision and range are defined under user control in the extendable-precision format. The x86 processor supports the 80-bit extended precision floating-point format, which meets the requirements of the IEEE's extended precision format, with a 15-bit exponent and 64-bit significand.

Note that the IEEE's extended precision format requires the extension of both significand precision and exponent range. However, in this thesis, "extended precision" denotes also the extension of the significand precision alone.

1.1.2 Extended Precision Floating-Point Arithmetic

Besides the 80-bit extended precision on the x86, most common processors, including GPUs, do not support any precision higher than double-precision; therefore, operations with higher precision than a precision supported by hardware must be executed by software implementation. For example, IEEE's binary128 quadruple-precision floating-point operations are available in the Intel Fortran Compiler [6], the GNU Fortran compiler and the GNU C Compiler [7] on the x86, but they are computed by software.

There are two principal ways of computing extended precision floating-point numbers using software. The first approach is the most straightforward: computing the extended precision floating-point numbers using integer arithmetic operations. This method is mainly used on arbitrary precision libraries such as the GNU Multiple

Precision Arithmetic Library (GMP) [8]. The binary128 emulations on the x86 are also implemented by this approach.

The second approach computes extended precision floating-point numbers using hardware implemented floating-point arithmetic and stores the numbers in a few pairs of floating-point values. The representative example is quadruple-precision floating-point operations using double-double (DD) arithmetic [9][10][11]. The method is used on the well-known quadruple- and octuple-precision floating-point arithmetic library – the QD library [12]. For DD arithmetic, a quadruple-precision floating-point number is represented using a pair of double-precision floating-point values, which have a higher and a lower part of the significand of the quadruple-precision floating-point numbers. DD arithmetic computes quadruple-precision floating-point arithmetic using only double-precision floating-point arithmetic operations. This approach can utilize the sign, the exponent, and the significand of existing floating-point formats; therefore, compared to the former approach, it has a speed advantage when only quadruple- or octuple-precision are required [4]. However, this approach cannot extend the exponent bit of the existing floating-point format.

Software processing of extended precision floating-point arithmetic operations involves highly computational intensive operations, and generally requires a greater computation time compared to the single- and double-precision floating-point operations by hardware on most modern processors.

1.2 GPU Computing

GPUs are specialized hardware accelerators for CPUs, designed to perform graphics processing tasks such as 3D shading and were originally designed to only perform graphics processing. In recent years, however, they have enabled us to perform general purpose computations that were traditionally handled by CPUs by a process called GPGPUs, which has been a major topic of research in the HPC area. In fact, since November 2009, GPU-equipped supercomputers have appeared in the top 10 of the TOP500 list [2].

The computational performance of GPUs has increased more rapidly than that of CPUs due to the rapid increase in the video games market. As a result, GPUs have provided a better cost performance than traditional CPUs. The early generation of GPUs supported only single-precision floating-point operations; GPU vendors, however, have released products that have targeted the HPC market, supporting double-precision floating-point operations, and error correcting code memory (ECC memory). Furthermore, GPU programming has become easier with the development

of GPU programming tools such as AMD Accelerated Parallel Processing (APP) [13], NVIDIA compute unified device architecture (CUDA) [14], and OpenCL [15]. In addition, GPUs have recently received attention as energy efficient processors.

1.2.1 Performance Characteristics

GPUs have high floating-point performance and high memory bandwidth. For instance, the Cray XK7 supercomputer [16], which is used as the base system of the Titan supercomputer at Oak Ridge National Laboratory, has an AMD Opteron 6274 CPU (16 cores) and an NVIDIA Tesla K20X GPU per node. The theoretical peak performance of the CPU is 281.6 GFlops of double-precision floating-point performance and 51.2 GB/s of the bandwidth of the main DRAM memory, whereas for the GPU, the theoretical peak performance is 1.31 TFlops and the bandwidth of the DRAM memory is 250 GB/s, respectively.

However, the architecture of GPUs is quite different from that of traditional CPUs. The most noticeable feature of the GPU is that it has a "many core" and "multithreaded" architecture. GPUs have a number of simple cores, which generally range from a few hundred to a few thousand, and a program is performed with a massive number of threads on the cores. Because of this architecture, GPUs achieve high performance only for highly parallel computations, such as vectors or matrices.

On the other hand, GPUs need to be controlled from CPUs and operating systems do not run on GPUs. GPUs can act only as accelerators for CPUs, which are connected via a peripheral bus such as the PCI Express (PCIe) bus. In addition, a general GPU connected bus, PCIe 2.0 $\times 16$, has a theoretical maximum bandwidth of 8 GB/s each way, which is narrower than that of the DRAM memory. Thus, only computationally intensive applications are capable of effectively utilizing GPUs.

1.2.2 CUDA

Compute Unified Device Architecture (CUDA) [14] is a parallel computing platform and programming model developed by NVIDIA. CUDA provides programming languages of extensions of C/C++ and Fortran to develop programs for GPUs. Although various languages and frameworks for developing these have been proposed, CUDA is one of the most widely used GPGPU platforms. I use CUDA and here describes its corresponding technical terms and knowledge to understand this thesis with reference to the NVIDIA CUDA C Programming Guide [17].

In CUDA, a GPU is called a "CUDA device" or simply a "device," whereas the CPU is called a "host." CUDA devices are classified by several generations according

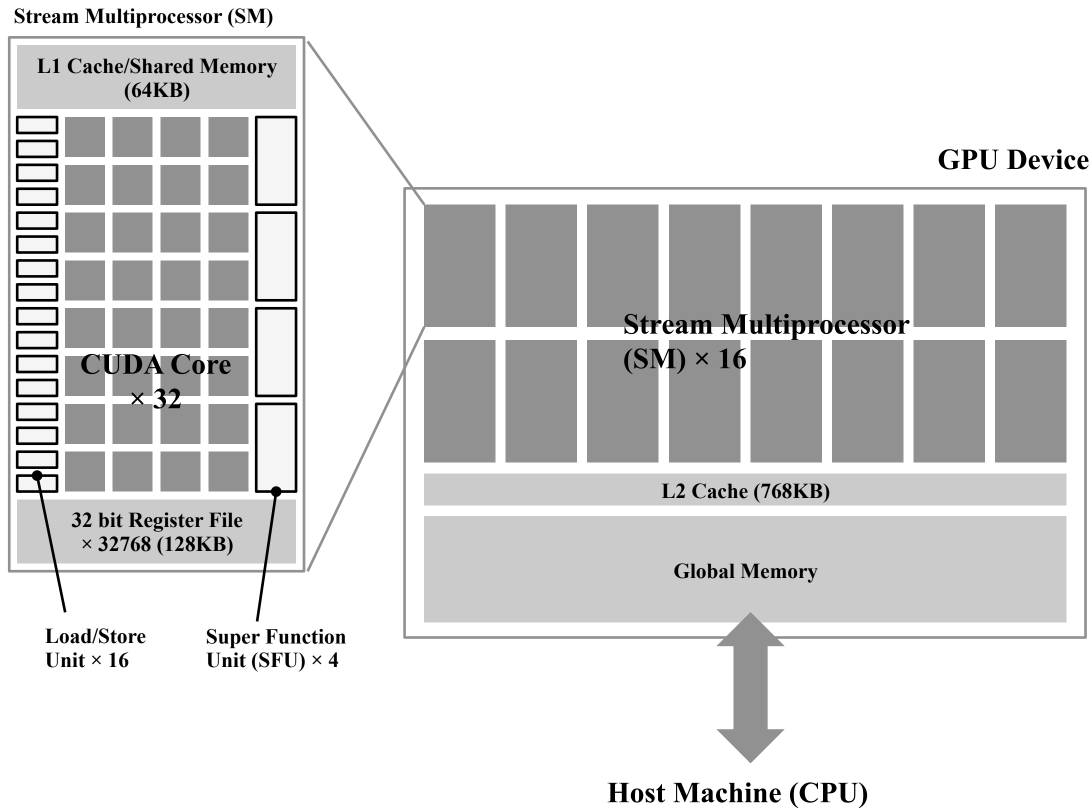


Figure 1.1: GPU architecture (NVIDIA Fermi architecture)

to the Compute Capability (CC). As an example of the CUDA devices, Figure 1.1 shows NVIDIA Fermi architecture GPUs [18], which was the CC 2.0 version released in 2009.

Devices having a large number of simple cores are called "CUDA cores." Each CUDA core has an Arithmetic Logic Unit (ALU) and a Floating-Point Unit (FPU). The FPU supports the IEEE 754-2008 compliant single- and double-precision floating-point operations, and the Fused Multiply-Add (FMA) instruction that performs $a \times b + c$ with one rounding step. Several CUDA cores are contained in the Streaming Multiprocessor (SM) and a device has several SMs. Each SM has register files, Special Function Units (SFUs), load/store units, and 64 KB of fast on-chip memory. The on-chip memory can be configured as 48 KB of shared memory, which can be used as scratch-pad memory with 16 KB of L1 cache or as 16 KB of shared memory with 48 KB of L1 cache. Each device also has a global memory, which is an off-chip DRAM memory and an L2 cache.

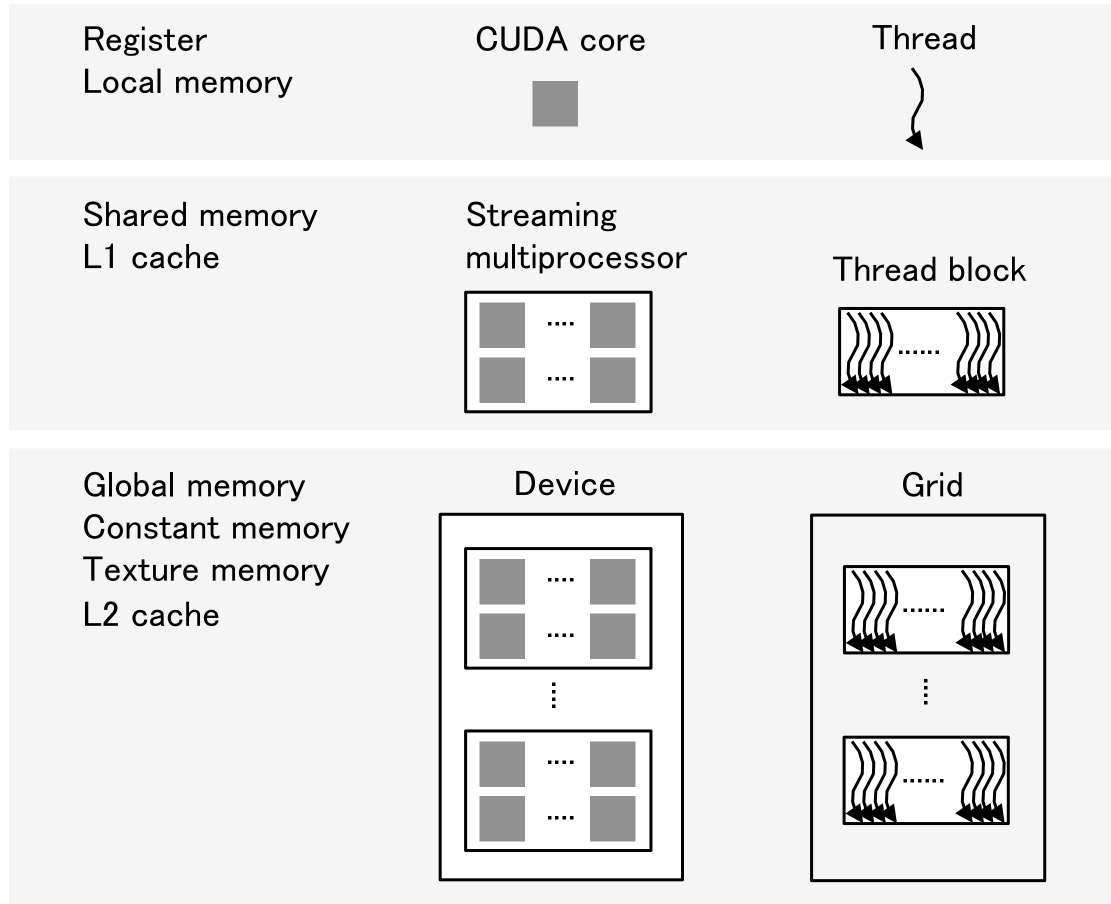


Figure 1.2: CUDA device and thread hierarchy

In CUDA, programs that are performed on GPUs are called "kernel grids" or "grids," which are invoked by the host. The grid is performed using a large number of threads on many CUDA cores. The threads are distributed to SMs as a group of threads called a "thread block." Each thread is performed on a CUDA core, and each thread block is performed on an SM. The number of threads in a thread block and the number of thread blocks are configurable, but the threads on a thread block are executed with every 32 threads on an SM. The unit of 32 threads, which is called a "warp," executes the same instruction at the same time. This architecture is called single instruction multiple threads (SIMT) architecture by NVIDIA. In CUDA, GPUs achieve high performance by hiding memory access latency with computations by switching the large number of threads.

The hierarchy of CUDA cores and threads is related to the memory hierarchy. Figure 1.2 shows the hierarchy and relation among the CUDA core, memory, and

thread. Each level of the hierarchy corresponds to the data accessibility of the threads. For example, a thread cannot access the data on the register files of the other threads; the data on shared memory, however, is accessible among the threads in the same thread block. This means that shared memory can be used not only as scratch-pad memory, but also for data communication between threads in the same thread block.

1.3 Overview of the Thesis

The thesis intends to show the implementation and performance of linear algebraic operations using extended precision floating-point arithmetic on GPUs. This chapter describes the scope of this study, main contributions, and the organization of this thesis.

1.3.1 Scope of the Study

As extended precision operations, this paper will focus on triple- and quadruple-precision operations where they are defined as a computation for floating-point data, which approximately has the triple and quadruple length of IEEE 754 single-precision's significand of 23-bit, respectively. As described previously, GPUs do not support higher than double-precision operations by hardware and therefore, such higher precision operations are executed by software implementation.

For implementation and evaluation, I will target NVIDIA GPUs and the CUDA environment. Specifically, it uses GPUs with the Fermi architecture [18] released in 2009 and the Kepler architecture [19] released in 2012. I will focus on linear algebra operations on a single GPU; the operations are computation for the data on the global memory of a single GPU. Thus, this thesis does not discuss the performance, which includes the time required to transfer data between the CPU and GPU via the PCIe bus.

As linear algebra operations, the first half of the thesis focuses on basic linear algebra operations for vector and dense matrix, and the latter half on sparse operations. Specifically, in the first half of the thesis, I will implement three Basic Linear Algebra Subprograms (BLAS) [20] subroutines: AXPY ($y = \alpha x + y$), GEMV ($y = \alpha Ax + \beta y$) and GEMM ($C = \alpha AB + \beta C$) as examples of vector and dense matrix operations. In the latter half, I will focus on sparse matrix operations. The thesis firstly shows the implementation and optimization of double-precision Sparse Matrix-Vector multiplication (SpMV) on GPUs. I then implement the Conjugate Gradient (CG) and Bi-Conjugate Gradient Stabilized (BiCGStab) methods, which

are Krylov subspace methods, hence, iterative methods for solving linear systems: $y = Ax$, using quadruple-precision arithmetic. The CG and BiCGStab methods mainly consist of some SpMV subroutines and some vector-vector operations such as AXPY.

1.3.2 Contributions

The main contributions of this thesis are as follows.

First, this paper reveals that the performance of some linear algebra operations using quadruple-precision floating-point arithmetic becomes memory bound on GPUs, and the execution time is approximately two times that of the double-precision operations due to the low Bytes/Flop ratio of GPUs.

Second, triple-precision operations for linear algebra operations on GPUs as a new extended precision choice between double- and quadruple-precision are proposed. This paper realizes the triple-precision linear algebra operations which are faster than the quadruple-precision operations where the performance of the operation is memory bound on both triple- and quadruple-precision.

Third, this paper presents optimization techniques of a SpMV subroutine for the CRS format on NVIDIA Kepler architecture GPUs. By utilizing new features of the Kepler architecture GPUs, speedups are achieved over the implementation for the earlier generation of Kepler architecture GPUs.

Finally, the performance of the CG and BiCGStab methods, which are Krylov subspace methods, are shown, using quadruple-precision floating-point arithmetic on GPUs. This paper suggests potential speedups by using quadruple-precision arithmetic instead of double-precision on GPUs.

1.3.3 Organization of the Thesis

The remainder of this thesis is organized as follows.

Chapter 2 will show the implementation and performance of triple- and quadruple-precision BLAS subroutines: AXPY, GEMV and GEMM on the Fermi architecture GPU. This section will firstly introduce double-double (DD) arithmetic, an existing method used to perform quadruple-precision floating-point arithmetic in software. On the other hand, I am proposing two new triple-precision floating-point formats, Double+Single (D+S)-type and Double+Int (D+I)-type formats, and a method of computing these values on GPUs. The evaluation will show the performance comparison of double-, triple-, and quadruple-precision subroutines on an NVIDIA Tesla M2050 GPU and discuss the performance using the Bytes/Flop ratio of the GPU

and the operations. As a result, this chapter will show that although the performance of the triple- and quadruple-precision GEMM is computationally bound, triple- and quadruple-precision AXPY and GEMV are memory bound, and their execution times are therefore close to 1.5 and 2 times more than that of double-precision subroutines, respectively.

Chapter 3 will show optimization techniques for double-precision SpMV for the CRS format on NVIDIA Kepler architecture GPUs using CUDA. This research is a step toward implementing and evaluating iterative methods for solving sparse linear systems using quadruple-precision operations in Chapter 4. This chapter will show the optimizations utilizing three new features of the Kepler architecture: a 48KB read-only data cache, shuffle instructions and expanding MaxGridDimX. The performance evaluation will show the effects of three optimizations on the 200 matrices that are randomly selected from the University of Florida Sparse Matrix Collection [21] on the Tesla K20 Kepler architecture GPU. As a result, the implementation optimized for the Kepler architecture is approximately 1.04 to 1.78 times faster than the original implementation for the Fermi architecture. Moreover, the implementation optimized for the Kepler architecture outperforms the NVIDIA cuSPARSE library's [22] implementation of the SpMV routine for the CRS format for 174 of the 200 matrices.

Chapter 4 will show the application of quadruple-precision operations on GPUs for iterative methods for solving sparse linear systems. The convergence of the Krylov subspace methods, which are iterative methods for solving linear systems: $y = Ax$, is significantly affected by rounding errors. Therefore, there are cases where reduction in rounding errors with quadruple-precision arithmetic causes the algorithm to converge more quickly. Even if the use of quadruple-precision arithmetic increases the execution time of one iteration, the total time until convergence may be reduced if an increased precision can sufficiently compensate for this by reducing the number of iterations. I implemented the CG and BiCGStab methods, which are Krylov subspace methods, using quadruple-precision floating-point arithmetic on GPUs. Then, I will compare the performance to the standard double-precision implementations on the Tesla K20 GPU. Note that this research aims to accelerate the methods using quadruple-precision arithmetic instead of double-precision arithmetic; therefore, the input data, the coefficient matrix, and the right hand vector are given in double-precision. The performance evaluation will show the execution time of one iteration of the unpreconditioned methods using quadruple-precision arithmetic is only approximately 1.11 to 2.20 times more than the double-precision versions. Moreover, I will show that there are cases where the time until convergence can be reduced using quadruple-precision arithmetic instead of double-precision,

even when quadruple-precision arithmetic is not necessary.

Finally, Chapter 5 will conclude the thesis and discuss future work.

Chapter 2

Triple- and Quadruple-Precision BLAS Subroutines on GPUs

2.1 Introduction

Most computational tasks in scientific and engineering computations are linear algebraic operations that use floating-point operations; however, floating-point operations have rounding errors, which may become a critical issue for some applications. With progress in computational science, there is a need for more accurate computation, especially in large-scale computing where the accumulation of rounding errors may lead to even more serious problems. The usual precision of hardware floating-point operations is insufficient in some cases, and greater accuracy and precision are required.

Although GPUs are specialized hardware accelerators for CPUs and are designed to perform graphics processing, they have recently enhanced general purpose computations and performances that were traditionally handled by CPUs. As a result, GPU computing has become a major focus of research in the HPC area. However, GPUs do not support higher than double-precision floating-point arithmetic through hardware. Therefore, it must be executed via software implementation.

Quadruple-precision operations have often been used as extended precision operations. For example, the double-double (DD) arithmetic [9] [10] [11] has been proposed and is often used to compute quadruple-precision floating-point arithmetic by software.

This chapter aims to discuss the performance of triple- and quadruple-precision linear algebra operations on GPUs. In order to evaluate the performance of triple- and quadruple-precision operations on basic linear algebra operations, I will imple-

ment representative Basic Linear Algebra Subprograms (BLAS) [20] subroutines: AXPY ($y = \alpha x + y$), GEMV ($y = \alpha Ax + \beta y$), and GEMM ($C = \alpha AB + \beta C$) on an NVIDIA Tesla M2050, which is a Fermi architecture GPU.

The remainder of this chapter is organized as follows. Section 2.2 introduces DD arithmetic for quadruple-precision operations on GPUs. Section 2.3 proposes triple-precision floating-point formats and their operations on GPUs. The implementation of triple- and quadruple-precision BLAS subroutines on GPUs is shown in Section 2.4. Section 2.5 provides the performance prediction of triple-precision BLAS subroutines on GPUs. Section 2.6 shows the performance evaluation results, which include performance comparisons of the triple- and quadruple-precision subroutines with the double-precision subroutines. Finally, Section 2.8 concludes this chapter.

2.2 Quadruple-Precision Floating-Point Operations

The quadruple-precision floating-point format was standardized in IEEE 754-2008 [5] as binary128. Some compilers supported the fotmat and operations by software for the x86 [6] [7]. However, the support is not available on GPUs.

However, double-double (DD) arithmetic [9] [10] [11] has been proposed and is often used to compute quadruple-precision floating-point arithmetic by software. This represents one quadruple-precision number by combining two double-precision numbers, and quadruple-precision floating-point arithmetic operations are performed using only double-precision floating-point arithmetic operations. Although this approach cannot extend the exponent bit of the existing floating-point format, it can utilize the sign, the exponent, and the significand of existing floating-point arithmetic operations; therefore, it has a speed advantage compared to methods implemented using integer arithmetic operations [4]. Binary128 emulation requires integer arithmetic operations, and therefore, I used DD arithmetic.

The concept of DD arithmetic was proposed in 1971 by Dekker [9], who called this approach “double-length” arithmetic, and showed his ALGOL 60 procedures in his paper. After the late 1990s, Bailey [10] and Briggs [11] developed a software library to perform quadruple-precision arithmetic operations using two IEEE 754 double-precision arithmetic operations, calling it “double-double” arithmetic. In addition, Hida et al. have developed the QD library [12], written in C++/Fortran-90, for quadruple-precision arithmetic by using DD arithmetic, and octuple-precision arithmetic by using quad-double (QD) arithmetic, which represents an octuple-precision number by combining four double-precision numbers. DD and QD arithmetics are

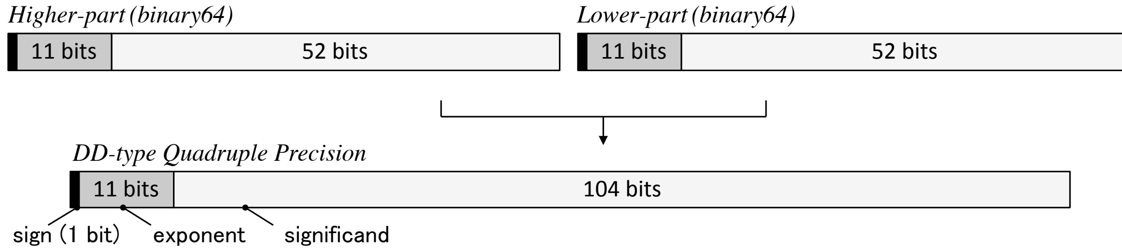


Figure 2.1: Quadruple-precision floating-point format on DD arithmetic

also implemented for NVIDIA's GPUs by Lu et al. [23] and Field-Programmable Gate Array (FPGA) devices by Dou et al. [24]. Graça [25] implemented double-precision floating-point arithmetic for GPUs that are not supported by hardware using double-float arithmetic, which is an approach that is similar to DD arithmetic. Thall [26] also implemented double-float and quad-float arithmetic on GPUs.

This section introduces algorithms of DD arithmetic for addition and multiplication that are required to implement the three BLAS subroutines. I used the same algorithms as DD arithmetic in the QD library. Throughout this thesis, normal arithmetic operations are represented using $\{+, -, \times, \div\}$, and the IEEE754-2008 floating-point operations with rounding to the nearest-even are represented using $\{\oplus, \ominus, \otimes, \oslash\}$.

2.2.1 DD-type Quadruple-Precision Floating-Point Format

DD arithmetic represents a quadruple-precision floating-point number a using a pair of two double-precision floating-point numbers, a_{hi} and a_{lo}: $a = a_{hi} + a_{lo}$, as shown in Figure 2.1. a_{hi} and a_{lo} represent a higher and a lower part of a significand of a quadruple-precision number, respectively. This assumes that $|a_{lo}| \leq 0.5\text{ulp}(a_{hi})$. "Ulp" denotes "unit in the last place," i.e., the gap between a floating-point number and the next largest number.

In IEEE 754-2008, the double-precision floating-point number format is defined as "binary64" with a significand part having 52 bits (actually, the total precision is 53 bits including the implicit bit) as shown in Table 1.1. Therefore, the total significand precision of the DD number is $52 + 52 = 104$ bits (actually, 106 bits including the implicit bits), and approximately 32 decimal digits. Note that the exponent and the significand of the DD number are less than the "binary128" IEEE 754-2008 quadruple-precision floating-point value format, which has a 112 bit significand with a 15 bit exponent.

QuadAdd: $c = a + b$

$$\begin{array}{rcc} & a_{hi} & a_{lo} \\ + & b_{hi} & b_{lo} \\ \hline & a_{hi}+b_{hi} & a_{lo}+b_{lo} \\ + & & e(a_{hi}+b_{hi}) \\ \hline & c_{hi} & c_{lo} \end{array}$$

Normalization: $|c_{lo}| <= 0.5\mathrm{ulp}(c_{hi})$

$$\begin{array}{cc} c_{hi} & c_{lo} \end{array}$$

QuadMul: $c = a \times b$

$$\begin{array}{rccc} & a_{hi} & a_{lo} & \\ \times & b_{hi} & b_{lo} & \\ \hline & a_{hi}b_{hi} & a_{hi}b_{lo} & \\ & & b_{hi}a_{lo} & \text{Not used} \\ + & & e(a_{hi}b_{hi}) & \cancel{a_{lo}b_{lo}} \\ \hline & c_{hi} & c_{lo} & \end{array}$$

Normalization: $|c_{lo}| <= 0.5\mathrm{ulp}(c_{hi})$

$$\begin{array}{cc} c_{hi} & c_{lo} \end{array}$$

$$(a = a_{hi}+a_{lo},\ b = b_{hi}+b_{lo},\ c = c_{hi}+c_{lo})$$

Figure 2.2: Concept of DD arithmetic

2.2.2 DD-type Quadruple-Precision Floating-Point Arithmetic

This section introduces algorithms of quadruple-precision addition and multiplication in DD arithmetic. Figure 2.2 shows the concept of the quadruple-precision addition and multiplication in DD arithmetic. The methods are similar to those used by humans to compute two-digit numbers on paper.

DD arithmetic is based on the following error-free floating-point operations.

Algorithm 1 Error-free addition: TwoSum (Knuth [27])

1: **function** $[s, e] \leftarrow \textsc{TwoSum}(a, b)$
2: $\quad s \leftarrow a \bigoplus b$
3: $\quad v \leftarrow s \bigominus a$
4: $\quad e \leftarrow (a \bigominus (s \bigominus v)) \bigoplus (b \bigominus v)$
5: **end function**

TwoSum by Knuth [27] is an error-free floating-point addition algorithm. It produces an expansion $a + b$ such that $s + e = a + b$, where s is an approximation of $a + b$, and e represents the rounding error in the calculation of s.

QuickTwoSum by Dekker [9] is another error-free floating-point addition algorithm. It produces an expansion $a + b$ such that $s + e = a + b$, provided that $|a| \geq |b|$. The TwoSum algorithm requires six double-precision floating-point op-

Algorithm 2 Error-free addition: QuickTwoSum (Dekker [9])

Require: $|a| \geq |b|$
1: **function** $[s, e] \leftarrow$QuickTwoSum(a, b)
2: $s \leftarrow a \bigoplus b$
3: $e \leftarrow b \bigominus (s \bigominus a)$
4: **end function**

erations (Flops), but the QuickTwoSum algorithm only requires three Flops. This algorithm is used to normalize the DD number to satisfy $|a_{lo}| \leq 0.5\text{ulp}(a_{hi})$.

Algorithm 3 Error-free multiplication: TwoProdFMA (Karp et al. [28])

1: **function** $[p, e] \leftarrow$TwoProdFMA(a, b)
2: $p \leftarrow a \bigotimes b$
3: $e \leftarrow$`fma` $(a \times b - p)$
4: **end function**

TwoProdFMA is an error-free floating-point multiplication algorithm. This algorithm produces $p+e = a\times b$, where p is an approximation of $a\times b$, and e represents the rounding error in the calculation of p. The first error-free multiplication algorithm was proposed by Veltkamp and Dekker [9]. Afterward, Karp and Markstein [28] modified the algorithm to that described above by using a double-precision Fused Multiply-Add (FMA) instruction, which calculates $a \times b + c$ with an intermediate result of 106 bits, with one rounding step (see also Nievergelt [29] and Ogita et al. [30]). In this algorithm, `fma`$(a \times b - p)$ represents the calculation of $a \times b - p$ using the FMA instruction.

The FMA instruction is available on some processors such as Power architecture processors, Intel Itanium, and recent NVIDIA, and AMD GPUs. Most processors that support the FMA instruction perform $a \times b + c$ (2 Flops) by one instruction in the same number of cycles as either a multiply $(a \times b)$ or an add $(a + b)$ instruction that performs one Flop. Therefore, the TwoProdFMA algorithm performs three Flops with two instructions.

Quadruple-precision addition algorithms use the aforementioned error-free floating-point operation algorithms. There are two kinds of quadruple-precision addition algorithms in the QD library: QuadAdd and QuadAddSloppy.

The QuadAdd algorithm calculates a quadruple-precision addition: $c = a + b$ $(c = c_{hi} + c_{lo}$, also for a and $b)$.

The QuadAddSloppy algorithm permits a few bits of error in the lower part of the DD number. Although this algorithm does not prevent the loss of digits when

Algorithm 4 Quadruple-precision addition: QuadAdd (Bailey [12])

```
1: function [c_hi, c_lo] ←QuadAdd(a_hi, a_lo, b_hi, b_lo)
2:     [s_1, s_2] ← TwoSum(a_hi, b_hi)
3:     [t_1, t_2] ← TwoSum(a_lo, b_lo)
4:     s_2 ← s_2 ⊕ t_1
5:     [s_1, s_2] ← QuickTwoSum(s_1, s_2)
6:     s_2 ← s_2 ⊕ t_2
7:     [c_hi, c_lo] ← QuickTwoSum(s_1, s_2)
8: end function
```

Algorithm 5 Quadruple-precision addition: QuadAddSloppy (Bailey [12])

```
1: function [c_hi, c_lo] ←QuadAddSloppy(a_hi, a_lo, b_hi, b_lo)
2:     [s, e] ← TwoSum(a_hi, b_hi)
3:     e ← e ⊕ (a_lo ⊕ b_lo)
4:     [c_hi, c_lo] ← QuickTwoSum(s, e)
5: end function
```

a carry occurs in the calculation of the lower part, it is simpler than the former QuadAdd algorithm. The QuadAddSloppy algorithm requires 11 Flops, whereas the QuadAdd algorithm requires 20 Flops and is considered effective when users do not want to sacrifice speed in exchange for a small increase in accuracy. In fact, the QuadAddSloppy algorithm is used by the default addition algorithm in the QD library. Therefore, I used the QuadAddSloppy algorithm.

The quadruple-precision multiplication algorithm is also based on error-free floating-point algorithms.

Algorithm 6 Quadruple-precision multiplication: QuadMul (Bailey [12])

```
1: function [c_hi, c_lo] ←QuadMul(a_hi, a_lo, b_hi, b_lo)
2:     [p_1, p_2] ← TwoProdFMA(a_hi, b_hi)
3:     p_2 ← p_2 ⊕ (a_hi ⊗ b_lo) ⊕ (a_lo ⊗ b_hi)
4:     [c_hi, c_lo] ← QuickTwoSum(p_1, p_2)
5: end function
```

The QuadMul algorithm calculates a quadruple-precision multiplication: $c = a \times b$ ($c = c_{hi} + c_{lo}$, also for a and b). This algorithm guarantees 106-bit accuracy and IEEE-style rounding, which is the same as the QuadAdd algorithm.

As previously shown, the quadruple-precision addition and multiplication algorithms are performed using only double-precision floating-point operations without

any branch-statement. Table 2.1 shows the number of double-precision floating-point instructions and Flops for quadruple-precision multiply-add operations in DD arithmetic. Three BLAS subroutines: AXPY, GEMV, and GEMM mainly consist of multiply-add operations. Note that in the QuadMul algorithm, there is a difference between the total number of instructions and the total number of Flop counts because of the use of the FMA instruction that performs $a \times b + c$ (two Flops) with one instruction.

Table 2.1: Number of double-precision floating-point instructions and Flops for quadruple-precision multiply-add

	# of double-precision floating-point instructions				# of
	Add/Sub	Mul	FMA	Total	Flops
QuadAddSloppy	11	0	0	11	11
QuadMul	5	3	1	9	10
QuadMul+QuadAdd	16	3	1	20	21

2.3 Triple-Precision Floating-Point Operations

I will propose new methods for triple-precision operations for linear algebra operations on GPUs. The performance of triple-precision linear algebra operations has been ignored in recent years. Triple-precision operations may be effective in cases where double-precision is insufficient and quadruple-precision is not necessary, but triple-precision is sufficient. In such cases, triple-precision operations, which are faster than quadruple-precision operations, are desired.

Triple-precision floating-point operations were implemented by Ikebe [31] in the 1960s. His method represents a triple length floating-point value by combining three floating-point values and performs triple-precision arithmetic using fixed-point arithmetic operations. In 2012, Ozawa [32] also implemented triple-precision arithmetic on x86 CPUs. He represents a triple-precision floating-point value using three double-precision values, and performs triple-precision arithmetic using double-precision floating-point arithmetic. Such methods using three floating-point values require more computation cost compared to DD arithmetic; therefore, I will propose new methods to store triple-precision floating-point values and compute these values on GPUs, which are based on DD arithmetic.

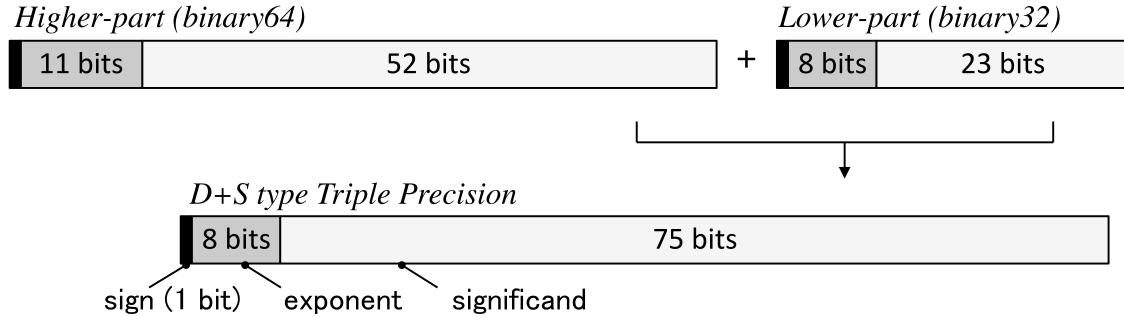


Figure 2.3: D+S-type triple-precision floating-point format

2.3.1 D+S-type Triple-Precision Floating-Point Format

I proposed the double+single (D+S)-type triple-precision floating-point format shown in Figure 2.3. This format uses a single-precision value to represent the lower part instead of a double-precision value as in the DD-type format: a triple-precision floating-point value $a^{(t)}$ is represented as $a^{(t)} = a_{hi}^{(d)} + a_{lo}^{(s)}$ using a double-precision floating-point value $a_{hi}^{(d)}$ and a single-precision floating-point value $a_{lo}^{(s)}$ ($|a_{lo}^{(s)}| \leq 0.5\text{ulp}(a_{hi}^{(d)})$). Here, superscripts of values, '(s)', '(d)', '(t)' and '(q)' denote single-, double-, triple- and quadruple-precision floating-point values, respectively.

The total significand precision of the D+S-type format is $52 + 23 = 75$ bits (77 bits, including the implicit bits) and approximately 24 decimal digits. The exponent is 8 bits, which is less than 11 bits of IEEE's double-precision and the DD-type, because the lower part is stored into the "binary32" single-precision, which has an 8-bit exponent.

2.3.2 D+S-type Triple-Precision Floating-Point Arithmetic

This thesis attempts to realize D+S arithmetic that uses the same algorithms as DD arithmetic. DD arithmetic consists entirely of double-precision floating-point operations, while D+S arithmetic can use single-precision floating-point operations when calculating the single-precision portion of the lower part.

This thesis includes some symbols to show the precision of a value and an operation (single or double) for the original algorithms of DD arithmetic. Superscripts of values, '(s)' and '(d)' denote single- and double-precision floating-point values, respectively. $(x^{(d)})^{(s)}$ indicates typecasting a value from double-precision $x^{(d)}$ to single-precision $x^{(s)}$. Operations where both values have the same precision are performed using that precision.

TwoSumTriple is an error-free floating-point addition algorithm for D+S arithmetic, which is based on the TwoSum algorithm. The TwoSumTriple algorithm produces an expansion $a^{(d)} + b^{(d)}$ such that $s^{(d)} + e^{(s)} = a^{(d)} + b^{(d)}$, where $s^{(d)}$ is an approximation of $a^{(d)} + b^{(d)}$ and $e^{(s)}$ is the rounding error in the calculation of $s^{(d)}$.

Algorithm 7 Error-free addition: TwoSumTriple

1: **function** $[s^{(d)}, e^{(s)}] \leftarrow$TwoSumTriple$(a^{(d)}, b^{(d)})$
2: $\quad s^{(d)} \leftarrow a^{(d)} \oplus b^{(d)}$
3: $\quad v^{(d)} \leftarrow s^{(d)} \ominus a^{(d)}$
4: $\quad e^{(s)} \leftarrow (a^{(d)} \ominus (s^{(d)} \ominus v^{(d)}))^{(s)} \oplus (b^{(d)} \ominus v^{(d)})^{(s)}$
5: **end function**

QuickTwoSumTriple is also an error-free floating-point addition algorithm to normalize the D+S number, which is based on the QuickTwoSum algorithm. The QuickTwoSumTriple algorithm produces an expansion $a^{(d)} + b^{(s)}$ such that $s^{(d)} + e^{(s)} = a^{(d)} + b^{(s)}$, provided $|a^{(d)}| \geq |b^{(s)}|$.

Algorithm 8 Error-free addition: QuickTwoSumTriple

Require: $|a^{(d)}| \geq |b^{(s)}|$
1: **function** $[s^{(d)}, e^{(s)}] \leftarrow$QuickTwoSumTriple$(a^{(d)}, b^{(s)})$
2: $\quad s^{(d)} \leftarrow a^{(d)} \oplus (b^{(s)})^{(d)}$
3: $\quad e^{(s)} \leftarrow b^{(s)} \ominus (s^{(d)} \ominus a^{(d)})^{(s)}$
4: **end function**

TwoProdFMATriple is an error-free floating-point multiplication algorithm for D+S arithmetic, which is based on the TwoProdFMA algorithm. It produces an expansion $a^{(d)} \times b^{(d)}$ such that $p^{(d)} + e^{(s)} = a^{(d)} \times b^{(d)}$, where $p^{(d)}$ is an approximation of $a^{(d)} \times b^{(d)}$ and $e^{(s)}$ is the rounding error in the calculation of $p^{(d)}$.

Algorithm 9 Error-free multiplication: TwoProdFMATriple

1: **function** $[p^{(d)}, e^{(s)}] \leftarrow$TwoProdFMATriple$(a^{(d)}, b^{(d)})$
2: $\quad p^{(d)} \leftarrow a^{(d)} \otimes b^{(d)}$
3: $\quad e^{(s)} \leftarrow (\texttt{fma}\ (a^{(d)} \times b^{(d)} - p^{(d)}))^{(s)}$
4: **end function**

The triple-precision addition and multiplication in D+S arithmetic uses the aforementioned algorithms.

TripleAddSloppy is the triple-precision addition algorithm based on the QuadAddSloppy algorithm: $c^{(t)} = a^{(t)} + b^{(t)}$ ($c^{(t)} = c_{hi}^{(d)} + c_{lo}^{(s)}$, also for a and b).

Algorithm 10 Triple-precision addition: TripleAddSloppy

1: **function** $[c_{hi}^{(d)}, c_{lo}^{(s)}] \leftarrow$ TRIPLEADDSLOPPY$(a_{hi}^{(d)}, a_{lo}^{(s)}, b_{hi}^{(d)}, b_{lo}^{(s)})$
2: $\quad (c_{hi}^{(d)}, c_{lo}^{(s)}) \leftarrow$ TwoSumTriple$(a_{hi}^{(d)}, b_{hi}^{(d)})$
3: $\quad c_{lo}^{(s)} \leftarrow c_{lo}^{(s)} \oplus (a_{lo}^{(s)} \oplus b_{lo}^{(s)})$
4: $\quad (c_{hi}^{(d)}, c_{lo}^{(s)}) \leftarrow$ QuickTwoSumTriple$(c_{hi}^{(d)}, c_{lo}^{(s)})$
5: **end function**

The TripleMul algorithm based on the QuadMul algorithm calculates the triple-precision multiplication: $c^{(t)} = a^{(t)} \times b^{(t)}$.

Algorithm 11 Triple-precision multiplication: TripleMul

1: **function** $[c_{hi}^{(d)}, c_{lo}^{(s)}] \leftarrow$ TRIPLEMUL$(a_{hi}^{(d)}, a_{lo}^{(s)}, b_{hi}^{(d)}, b_{lo}^{(s)})$
2: $\quad (c_{hi}^{(d)}, c_{lo}^{(s)}) \leftarrow$ TwoProdFMATriple$(a_{hi}^{(d)}, b_{hi}^{(d)})$
3: $\quad c_{lo}^{(s)} \leftarrow c_{lo}^{(s)} \oplus ((a_{hi}^{(d)})^{(s)} \otimes b_{lo}^{(s)}) \oplus (a_{lo}^{(s)} \otimes (b_{hi}^{(d)})^{(s)})$
4: $\quad (c_{hi}^{(d)}, c_{lo}^{(s)}) \leftarrow$ QuickTwoSumTriple$(c_{hi}^{(d)}, c_{lo}^{(s)})$
5: **end function**

Table 2.2 shows the number of floating-point instructions and cycles for D+S and DD addition and multiplication functions on the Tesla M2050 GPU. For the "add", "mul", and "fma" instructions, "rn" denotes round-to-the-nearest-even, and "f32" and "f64" indicate 32-bit (single-precision) and 64-bit (double-precision) floating-point operations, respectively. "cvt" indicates the typecasting instruction. On the GPU, single- and double-precision floating-point instructions are performed in one and two cycles, respectively. Therefore, D+S arithmetic is initially expected to attain better performance than DD arithmetic. However, many typecastings between single- and double-precisions, which also require two cycles, are required. For this reason, D+S arithmetic requires more cycles than DD arithmetic.

2.3.3 Computation of D+S-type Values using DD Arithmetic

Theoretically D+S arithmetic requires more instructions than DD arithmetic on the Tesla M2050 GPU due to many typecastings, as previously mentioned. Hence, it is better to compute triple-precision values using DD arithmetic than to compute them using D+S arithmetic. In other words, the input and output data of a linear algebra operation use the D+S-type triple-precision floating-point formats; the arithmetic operations are, however, performed on quadruple-precision using DD arithmetic; the temporary values on register files for storing intermediate results are also stored

Table 2.2: Floating-point instruction and cycle counts for D+S and DD arithmetic on Tesla M2050

Instruction	D+S arithmetic		DD arithmetic	
	TripleAddSloppy	TripleMul	QuadAddSloppy	QuadMul
add.rn.f32	1 cycle × 4	1 cycle × 3	1 cycle × 0	1 cycle × 0
mul.rn.f32	1 cycle × 0	1 cycle × 2	1 cycle × 0	1 cycle × 0
add.rn.f64	2 cycles × 7	2 cycles × 2	2 cycles × 11	2 cycles × 5
mul.rn.f64	2 cycles × 0	2 cycles × 1	2 cycles × 0	2 cycles × 3
fma.rn.f64	2 cycles × 0	2 cycles × 1	2 cycles × 0	2 cycles × 1
cvt.f64.f32	2 cycles × 1	2 cycles × 1	2 cycles × 0	2 cycles × 0
cvt.f32.f64	2 cycles × 3	2 cycles × 4	2 cycles × 0	2 cycles × 0
Total cycles	26 cycles	23 cycles	22 cycles	18 cycles

using the DD-type format. Although this method has no speed advantage compared to quadruple-precision operations for arithmetic operations, it is able to reduce the data access time for global memory and it may then have a speed advantage on linear algebra operations.

Figure 2.4 shows the concept of this method. The D+S-type data on global memory is loaded to register files, and is then these converted to DD-type data. The DD-type data is computed by using DD arithmetic. The result stored in DD-type is converted to D+S-type on the register files, and finally, the D+S-type data is stored in global memory.

On GPUs, the main bottleneck in data access from an FPU to data in global memory is the access time for global memory. Although the bandwidth of a Tesla M2050's register is unknown, it can be estimated in a way similar to that presented by Tan et al. [33], i.e., the bandwidth of shared memory, which is fast on-chip memory on GPUs, is 1030.4 GB/s. It can be presumed that the bandwidth of the register is at least greater than this value. However, the nominal peak bandwidth of global memory, which is off-chip memory, is 148 GB/s.

Therefore, this method may still have a speed advantage compared to quadruple-precision operations when the data access time for global memory is greater than the time for arithmetic operations. This method is also effective for reducing data transfer time between CPU and GPU via the PCIe bus and inter-node communication time on GPU clusters. The theoretical peak bandwidth of the PCIe 2.0 x16 is 8GB/s for each direction, which is much slower than that of global memory, and therefore, the bandwidth may become an application performance bottleneck.

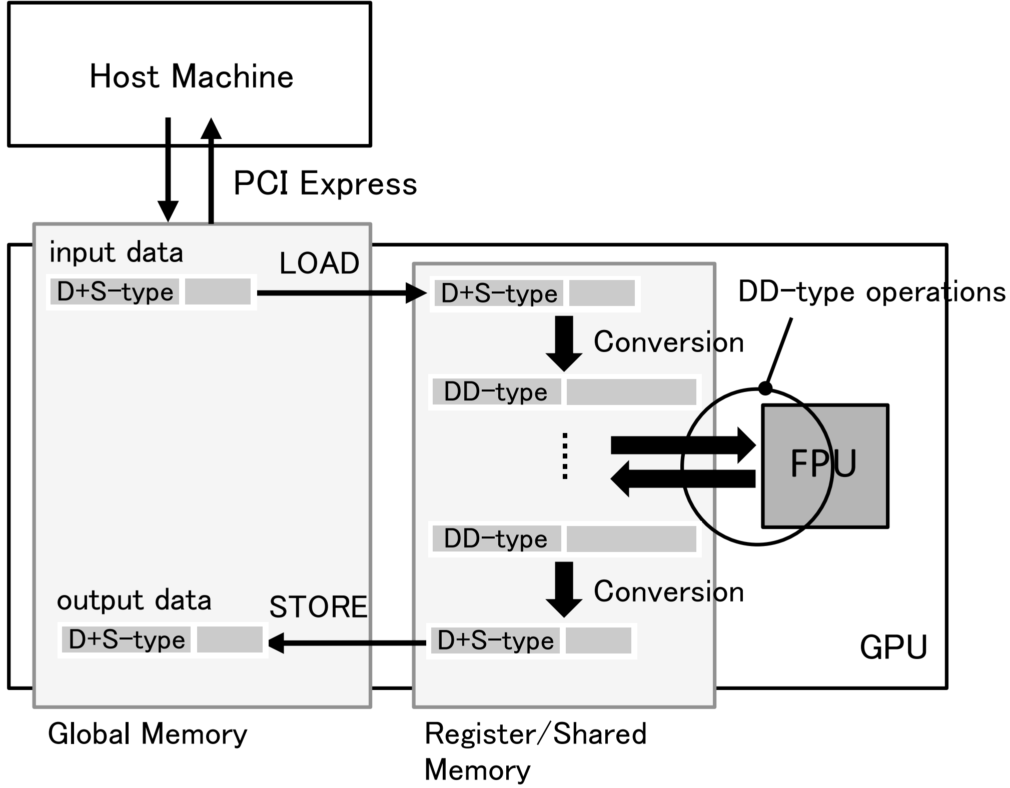


Figure 2.4: Triple-precision operations using DD arithmetic on GPUs

However, this method does not offer an advantage in computationally bound operations, except that it saves global memory space. In such a case, the peak performance is equal to that of the quadruple-precision operation using DD arithmetic. In addition, this method wastes register space compared to the triple-precision arithmetic using D+S arithmetic.

2.3.4 D+I-type Triple-Precision Floating-Point Format

The D+S-type triple-precision floating-point format has an 8-bit exponent, which is less than the 11-bit exponent of the IEEE's double-precision format and the DD-type format. This may become a problem resulting in over- or underflow when the data is converted to the D+S-type format from the double- and quadruple-precision formats.

The Double+Int (D+I)-type triple-precision floating-point format shown in Figure 2.5 is proposed to address this problem. The arithmetic operations are assumed to be computed using DD arithmetic as in the case of the D+S-type values. The

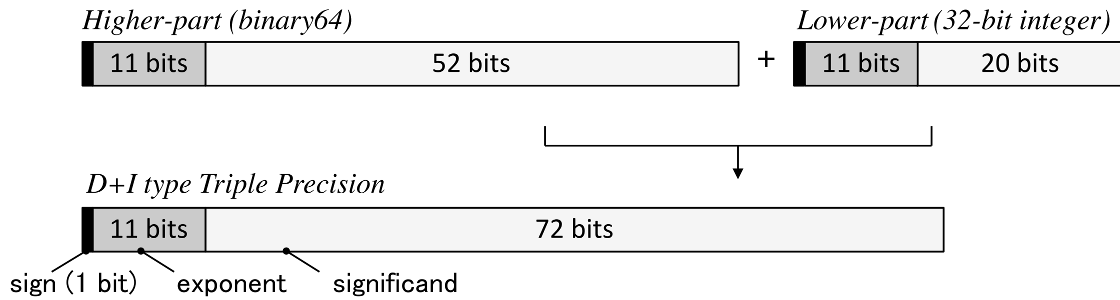


Figure 2.5: D+I-type triple-precision floating-point format

D+I-type format uses a 32-bit integer value to store the lower part of the DD-type format rather than the single-precision value of the D+S-type format. The top 32 bits of the lower part of the DD-type format, which is stored in a double-precision value, are stored in the 32-bit integer value, i.e., the top 32 bits include the sign bit (1 bit), the exponent bits (11 bits), and the significand of the top 20 bits of the double-precision value. The total significand precision of the D+I-type format is $52 + 20 = 72$ bits (73 bits, including the implicit bits) and the exponent is 11 bits: it is increased to the same width as that of the double-precision and the DD-type format. However, the significand is decreased by 3 bits compared to the D+S-type.

Typecastings between D+I- and DD-type values can be implemented using the union definition of the C language and logical shift operations. Listing 2.1 shows the typecasting function from the DD- to D+I-type format. The function refers to the lower double-precision floating-point value of the DD-type value as a 64-bit integer value, and cuts down the lower 32 bits by a logical right-shift operation, while the top 32 bits are stored in a 32-bit integer value. Lines 14–23 in Listing 2.1 perform rounding to the nearest even, which is the most accurate and is the default rounding mode in the IEEE standard. When the rounding operation is skipped it rounds to zero. Rounding to the nearest even allows error within 0.5 ulp and rounding to zero allows error within 1 ulp. The typecasting function from D+I- to the DD-type format shown in Listing 2.2 refers to the lower 32-bit integer value of the D+I-type value as a 64-bit integer value, left-shifts the bit data, and stores the bit data in a double-precision floating-point value using the union definition.

Listing 2.1: Conversion from DD-type to D+I-type

```
1  union double_int64 {
2          int64_t int64_;
3          double double_;
4  };
5
6  __host__ __device__ __forceinline__ void dd_to_di
7          (double2 dd, double &d, int32_t &i) {
8          union double_int64 u;
9          int64_t odd, border;
10
11         u.double_ = dd.y;
12         l = (int32_t)(u.int64_ >> 32);
13
14         odd = u.int64_ & 0xFFFFFFFF;
15         border = 0x80000000;
16
17         if (odd < border) {
18         }
19         else if (odd > border) {
20                 i++;
21         } else {
22                 if (i&1 == 1) i++;
23         }
24         d = dd.x;
25 }
```

2.4 Implementation of Triple- and Quadruple-Precision BLAS Subroutines on GPUs

Several frequently used low-level kernel operations on basic linear algebra operations, such as matrix multiplication, are defined as Basic Linear Algebra Subprograms (BLAS) [20], which is an application programming interface (API) for basic linear algebra operations. The BLAS is a de facto standard that has various implementations on different architectures, not only according to computer vendors [34], but also based on open source projects [35]. The BLAS subroutines are classified

Listing 2.2: Conversion from D+I-type to DD-type

```
1  __host__ __device__ __forceinline__ void di_to_dd
2          (double d, int32_t i, double2 &dd) {
3          union double_int64 u;
4          int64_t l;
5
6          l = (int64_t)i;
7          u.int64_ = l << 32;
8
9          dd.x = d;
10         dd.y = u.double_;
11 }
```

into three level operations: 1, 2, and 3. Level-1 contains vector-vector operations such as AXPY ($y = \alpha x + y$). Level-2 contains matrix-vector operations such as GEMV ($y = \alpha Ax + \beta y$). Level-3 contains matrix-matrix operations such as GEMM ($C = \alpha AB + \beta C$).

To evaluate the performance of linear algebra operations on triple- and quadruple-precision, I will implement representative BLAS subroutines of three levels: AXPY, GEMV, and GEMM and evaluates their performance on GPUs.

2.4.1 DD Arithmetic Functions

The DD arithmetic functions that perform the quadruple-precision addition and multiplication are implemented as CUDA device functions. These functions are defined as an inline function using the "__forceinline__" keyword to avoid a function call overhead. The DD value is stored using a double2-type value, a vector-type defined in CUDA. The double2-type value consists of two double-type values, and it is handled as one 16 byte value by the CUDA compiler [36].

Listing 2.3 shows an example of the implementation of the DD multiplication. The CUDA compiler automatically replaces multiply-add operations with the FMA instruction. To prevent the FMA instruction from altering the result of the quadruple-precision multiplication algorithm, the built-in functions, `__dmul_rn` and `__dadd_rn` are used for a separate multiplication and addition, respectively.

Listing 2.3: Implementation of QuadMul

```
1 __device__ __forceinline__ void QuadMul
2                 (double2 a, double2 b, double2 &c) {
3         double2 t;
4         TwoProdFMA (a.x, b.x, t.x, t.y);
5         t.y = __dadd_rn(t.y, __dadd_rn
6                 (__dmul_rn(a.x, b.y), __dmul_rn(a.y, b.x)));
7         QuickTwoSum (t.x, t.y, c.x, c.y);
8 }
```

2.4.2 Implementation of BLAS Kernels

Implementation techniques for kernels for the BLAS subroutines are the same as the general approach employed for single- or double-precision subroutines. Triple- and quadruple-precision subroutines perform quadruple-precision floating-point arithmetic using DD arithmetic instead of double-precision floating-point arithmetic by calling the aforementioned DD arithmetic device functions.

On the AXPY and the GEMV, each thread computes an element of the vector of y in parallel. On the GEMM, these threads are arranged as a two-dimensional structure. I implemented only the general case where the input matrices are not transposed and the input vectors are incx = incy = 1 (the increment interval for the elements of vector x and y).

For triple-precision subroutines, the interface of triple-precision BLAS subroutines uses D+S- or D+I-type triple-precision formats. However, the operations are performed using DD arithmetic, and the temporary values for storing the intermediate results are of DD-type. In other words, the BLAS kernel is quadruple-precision, but the interface is triple-precision. Listing 2.4 shows an example of the implementation of a triple-precision AXPY subroutine using the D+I-type format. Since the implementation internally uses DD arithmetic, triple-precision subroutines require typecasting between triple- and quadruple-precision formats for loading and storing data. The "di_to_dd" and "dd_to_di" functions perform typecasting between the DI- and DD-type formats shown in Listing 2.2 and Listing 2.1, respectively.

The GEMV and the GEMM use a blocking algorithm to access the shared memory for data reuse using an example from Nath et al.'s GEMM implementation for the Fermi architecture [37]. The shared memory is a fast on-chip memory that can be shared among threads in the multiprocessor and can be used as a user-managed scratch-pad memory. Figures 2.6 and 2.7 show the implementation of the GEMV

Listing 2.4: Triple-precision AXPY using D+I-type format

```
1  __global__ void TAXPY
2          (int n, double a1, int a2, double *x1, int *x2,
3           double *y1, int *y2) {
4          int tid = blockDim.x * blockIdx.x + threadIdx.x;
5          double2 ar, xr, yr;
6
7          di_to_dd (a1, a2, ar);
8          while (tid < n) {
9                  di_to_dd (x1[tid], x2[tid], xr);
10                 di_to_dd (y1[tid], y2[tid], yr);
11                 yr = QuadAdd (QuadMul (ar, xr), yr);
12                 dd_to_di (yr, y1[tid], y2[tid]);
13                 tid += gridDim.x * blockDim.x;
14         }
15 }
```

and the GEMM, respectively. In these figures, the areas shown in black are stored to the shared memory. NT means the number of threads in a thread block and BLK means the GEMM blocking size. Each thread block performs the inner product calculations in the direction of the black arrow. The optimal values of NT and BLK are experimentally determined, based on the thread group that accessed the GPU's memory. For GEMV the optimal NT is 128, for GEMM the optimal BLK is 16, and NT is 8 on both triple- and quadruple-precision subroutines.

2.4.3 Data Structures of Triple- and Quadruple-Precision Value Arrays in Global Memory

This section discusses the optimal way to layout D+I-, D+S-, and DD-type arrays in global memory, which in turn defines the BLAS interface.

On dense linear algebra operations, such as BLAS subroutines, the memory access pattern is sequential. In CUDA, several memory access transactions are coalesced into a single transaction when consecutive threads access consecutive memory addresses. In particular, one memory transaction can transfer a maximum of 128 bytes at a time, provided the data is aligned to an address that is a multiple of the memory transaction size.

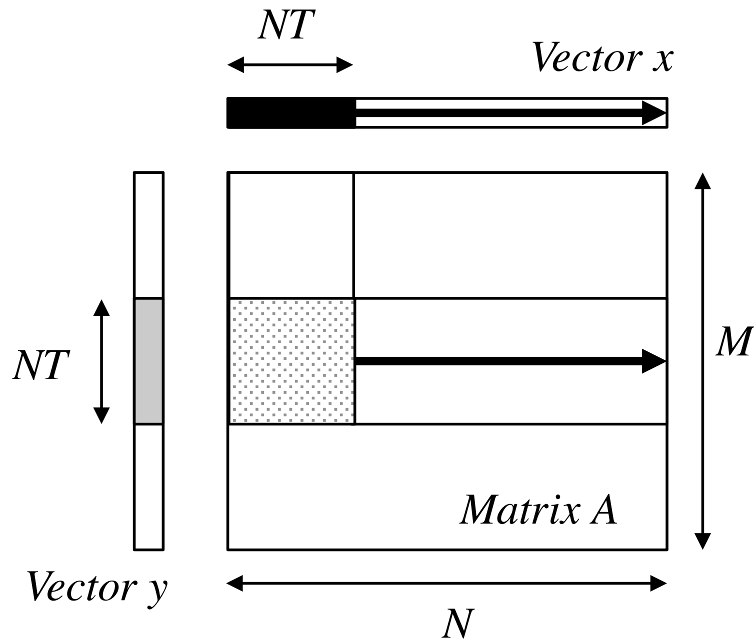


Figure 2.6: GEMV kernel

One triple-precision value is represented by a pair of 8-byte (a double-precision floating-point value) and 4-byte value (a single-precision floating-point value on the D+S-type or an integer value on the D+I-type): the structure, which defines one triple-precision value, is 12 bytes. Therefore, when using an array of the structures, which is called the array-of-structures (AoS) layout, satisfying this alignment condition is difficult. Instead, a triple-precision value array can be stored independently using two arrays, an 8-byte and a 4-byte value array. This method is called the structure-of-arrays (SoA) layout and can satisfy the 128-byte alignment condition. Figure 2.8 shows the layout of D+S-type value arrays in global memory for both AoS and SoA layouts. I used the latter SoA layout for storing triple-precision data in global memory. This chapter presents a comparison of both layouts in the performance evaluation and shows that the SoA layout performs better than the AoS layout.

For quadruple-precision, one DD-type value is represented by two 8-byte values, such problems do not occur. I used the AoS layout which stores one quadruple-precision array to an array of DD-type structures.

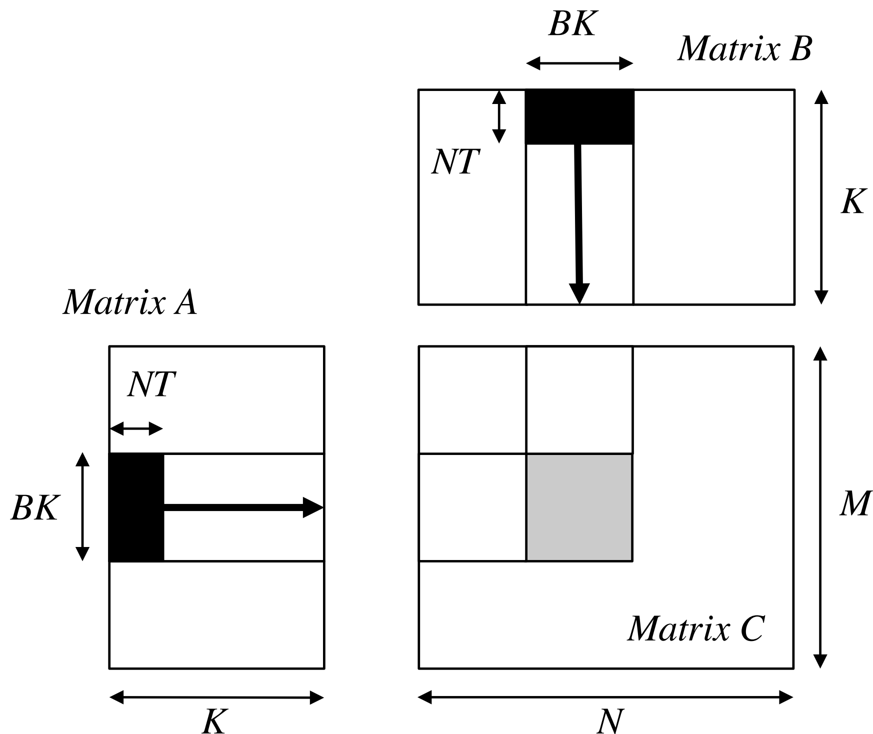


Figure 2.7: GEMM kernel

2.5 Performance Prediction

This section estimates the performance of the three BLAS subroutines: AXPY ($y = \alpha x + y$), GEMV ($y = \alpha Ax + \beta y$), and GEMM ($C = \alpha AB + \beta C$) on the Tesla M2050 GPU. This research assumes that test problems are square matrices of size $N \times N$ and vectors of length N.

2.5.1 Theoretical Peak Performance on DD Arithmetic

The three BLAS subroutines consist mainly of multiply-add operations that perform $a \times b + c$. All of the floating-point operations on AXPY are multiply-add operations. For GEMV, the largest $O(N^2)$ term, a matrix-vector multiplication of Ax, all consists of multiply-add operations, as does that for GEMM, the largest $O(N^3)$ term, which is a matrix-matrix multiplication of AB. The performance of multiply-add operations on the GPU is discussed here.

Firstly, the theoretical peak floating-point performance of the Tesla M2050 GPU is shown. An Floating-Point Unit (FPU) of the GPU performs double-precision

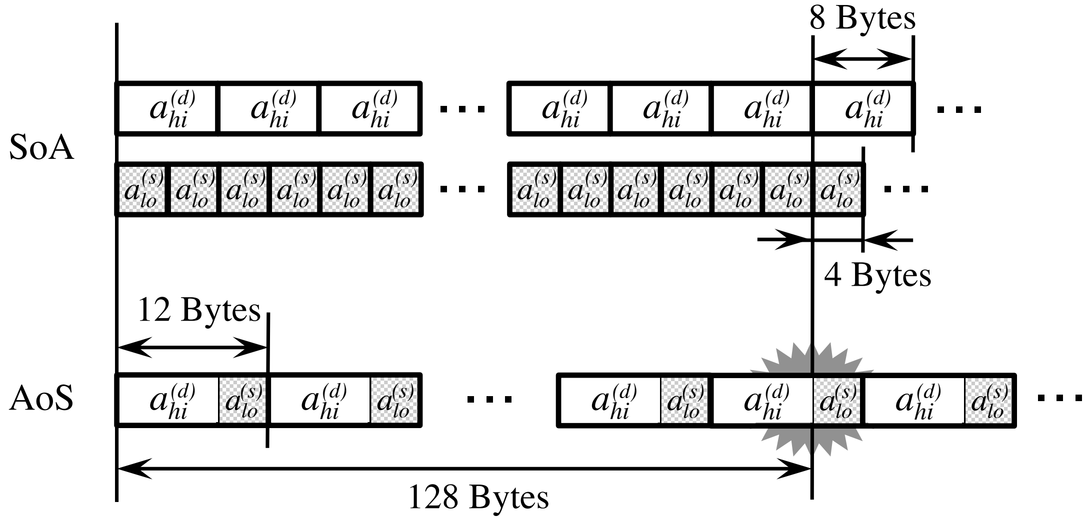


Figure 2.8: Array of Structures (AoS) layout and Structure of Arrays (SoA) layout on D+S-type values

multiply-add operations ($a \times b + c$) using one FMA instruction in two cycles. Therefore, the theoretical peak performance of double-precision multiply-add operations is 1.15[GHz] $\times$ 14[SM] $\times$ 32[CUDACore] $\times$ (2[Flop]/2[cycle]) = 515.2[GFlops].

Quadruple-precision multiply-add operations are equivalent to 11+10=21 [Flop] of double-precision floating-point operations using DDAddSloppy and DDMul, as shown in Table 2.1 and require 40 cycles. Therefore, DD arithmetic requires 20 times the cycles (i.e., execution time) of double-precision arithmetic. The theoretical peak performance of quadruple-precision multiply-add operations is 1.15[GHz] $\times$ 14[SM] $\times$ 32[CUDACore] $\times$ (21[Flop]/40[cycle]) = 270.48[GFlops] of double-precision floating-point operations. This performance is approximately half of the theoretical peak performance of the GPU. This is because on quadruple-precision multiply-add operations, only one of the 21 instructions is the FMA instruction that performs two Flops computations, while the other instructions are addition or multiplication instructions that perform only one Flop operation.

Here, I introduce DDFlops that refer to the number of DD-type quadruple-precision floating-point operations per second, instead of general Flops for double-precision. Thus, quadruple-precision multiply-add operations can be defined as 2 DDFlops. Therefore, the theoretical peak performance of quadruple-precision multiply-add operations is 1.15[GHz]$\times$14[SM]$\times$32[CUDACore]$\times$(2[DDFlop]/40[cycle]) = 25.76[GDDFlops]. This is just 1/20 of the value of double-precision theoretical peak Flops.

2.5.2 Performance Prediction using the Bytes/Flop and Bytes/DDFlop ratios

The Bytes/Flop ratio indicates the amount of data (bytes) required per floating-point operation (one Flop) for a certain operation. For a processor, the ratio indicates the amount of data that can be processed by a system in one Flop. When the Bytes/Flop ratio of a BLAS subroutine is larger than that of a GPU, the performance of the subroutine on the GPU is memory bound: limited by the global memory bandwidth of the GPU. This section provides the performance prediction of double-, triple-, and quadruple-precision BLAS subroutines on the GPU using the ratio. The thesis uses Bytes/Flop for double-precision operations. For triple- and quadruple-precision subroutines with DD arithmetic on GPUs, Bytes/DDFlop, i.e., the amount of data (bytes) required or can be processed per DD-type floating-point operation (one DDFlop) is used.

Bytes/Flop and Bytes/DDFlop for GPU

On the Tesla M2050 GPU, the theoretical peak floating-point performances of double- and DD arithmetic are 515.2 GFlops and 25.76 GDDFlops, respectively. The theoretical peak memory bandwidth of global memory is 148 GB/s. Therefore, for double-precision arithmetic operations, the Bytes/Flop ratio is 148[GB/s]/515.2[GFlops] $\approx$ 0.29[Bytes/Flop], and for DD arithmetic operations, the Bytes/DDFlop ratio is 148[GB/s]/25.76[GDDFlops] $\approx$ 5.75[Bytes/DDFlop].

Bytes/Flop and Bytes/DDFlop for AXPY

On AXPY, $2N$ Flops computations are performed on $2N$ and N floating-point values for storing and loading, respectively. For the double-precision subroutine, a floating-point value is 8 Bytes, and therefore, the Bytes/Flop ratio is 12 Bytes/Flop. The Bytes/Flop ratio is bigger than GPU's Bytes/Flop ratio for double-precision, approximately 0.29 Bytes/Flop. Therefore, it can be predicted that the performance of double-precision AXPY will be memory bound on the GPU. For the triple- and quadruple-precision subroutines, the floating-point values are 12 and 16 Bytes, respectively, and their Bytes/DDFlop ratios are 18 and 24 Bytes/DDFlop, respectively. The ratios are larger than GPU's ratio for DD arithmetic, approximately 5.75 Bytes/DDFlop, and therefore, it can be presumed that the performances of triple- and quadruple-precision AXPY will also be memory bound on the GPU.

Bytes/Flop and Bytes/DDFlop for GEMV

On GEMV, for a matrix of size $N \times N$, $2N^2+3N$ Flops computations are performed on N^2+2N and N for floating-point value loading and storing, respectively. For simplicity, when the $O(N)$ term is omitted, the Bytes/Flop ratio for the double-precision subroutine is 4 Bytes/Flop. For the triple- and quadruple-precision subroutines, the Bytes/DDFlop ratios are 6 and 8 Bytes/DDFlop, respectively. For all precision subroutines, the ratios are larger than the corresponding GPU's ratios. Therefore, it can be predicted that the performances of all precision GEMV will be memory bound on the GPU as well as AXPY.

Bytes/Flop and Bytes/DDFlop for GEMM

On GEMM, for matrices of size $N \times N$, $2N^3 + 3N^2$ Flops computations are performed on $4N^2$ floating-point value load/store operations. However, in the estimation, the cache is assumed to be unlimited. For simplicity, when $O(N^2)$ is omitted, the Byte/Flop ratio for double-precision subroutine is $16/N$ Bytes/Flop and the Bytes/DDFlop ratios for triple- and quadruple-precision subroutines are $24/N$ and $32/N$ Bytes/DDFlop, respectively. For all precision subroutines, the ratios are smaller than the corresponding GPU's ratios when $N > 55$ and therefore, it can be predicted that the performances of all precision GEMM will be computationally bound on the GPU.

In conclusion, it can be predicted that AXPY and GEMV are memory bound and GEMM is computationally bound on all precision in theory. Table 2.3 shows the Bytes/Flop and Bytes/DDFlop ratios for three BLAS subroutines. It is expected that when the performance is memory bound, the execution times of triple- and quadruple-precision subroutines are approximately 1.5 and 2.0 times that of the double-precision subroutine, respectively in theory. When the performance is computationally bound, the performance of triple-precision subroutines is approximately equal to that of quadruple-precision subroutines because both subroutines perform the same DD arithmetic and the execution time of triple- and quadruple-precision subroutines are approximately 20 times that of the double-precision subroutine.

2.6 Performance Evaluation

This section evaluates the performance of triple- and quadruple-precision BLAS subroutines and comparison with the double-precision subroutines on the Tesla M2050.

Table 2.3: Bytes/Flop and Bytes/DDFlop for BLAS subroutine

	AXPY	GEMV	GEMM
Double-precision [Bytes/Flop]	12	4	16/N
Triple-precision [Bytes/DDFlop]	18	6	24/N
Quadruple-precision [Bytes/DDFlop]	24	8	32/N

Table 2.4: Evaluation environment

CPU	Intel Xeon E5630 (2.53 GHz, Quad-Core) × 2 sockets
RAM	24 GB (DDR3)
GPU	NVIDIA Tesla M2050
Video RAM	3 GB (GDDR5, ECC-enabled)
GPU Bus	PCI-Express 2.0 x16
OS	CentOS 6.3 (x86-64) kernel 2.6.32-279.11.1.el6.x86_64
CUDA	CUDA 5.0
Compiler	gcc 4.4.6 (-O3), nvcc 5.0 (-O3)

2.6.1 Evaluation Methods

Table 2.4 shows the evaluation environment. I will compare the execution times of double-, triple-, and quadruple-precision BLAS subroutines. For double-precision subroutines, NVIDIA CUBLAS 5.0 [34] is used. Test problems are square matrices of size $N \times N$ with column-major order and vectors of length N. All input data and the α and β parameters comprise random numbers which are generated using the DD-type random number generator function (`ddrand`) in the QD library. For double- and triple-precision, the DD-type random numbers are converted to each precision.

To accurately evaluate the performance, a subroutine is repeatedly executed for at least one second and at least three times and then the average execution time is calculated. The performances of triple- and quadruple-precision subroutines are represented using DDFlops.

2.6.2 Performance Comparison of Double-, Triple- and Quadruple-Precision BLAS Subroutines

This section shows the DDFlops performance of triple- and quadruple-precision subroutines. In addition, the relative execution times for triple- and quadruple-precision

subroutines are presented as a multiple of the execution time of the double-precision subroutines of CUBLAS.

AXPY

Figures 2.9 and 2.10 show the DDFlops performance and relative execution times, respectively. Although the triple- and quadruple-precision operations require 20 times the double-precision floating-point instructions of the double-precision operation, the actual execution times for triple- and quadruple-precision subroutines are close to 1.5 and 2.0 times, respectively, as expected in Section 2.5. However, when $N < 10,000$, no performance gap exists due to the kernel generation overhead because the execution time of an empty kernel is approximately equal to that of AXPY when $N < 10,000$. The performance of triple-precision subroutines using the D+S- and D+I-type formats is also approximately the same.

GEMV

Figures 2.11 and 2.12 show the DDFlops performance and relative execution times, respectively. As was the case for AXPY, it can be predicted that the performance of GEMV is memory bound on all precisions on the GPU in Section 2.5. As a result, the execution times for triple- and quadruple-precision subroutines are close to 1.5 and 2.0 times that of the double-precision subroutine, respectively. The relative execution times for triple- and quadruple-precision subroutines decrease as N increases because the throughput of the triple- and quadruple-precision GEMV kernel is less than that of the double-precision subroutine of CUBLAS for small vectors and matrices.

GEMM

Figures 2.13 and 2.14 show the DDFlops performance and relative execution times, respectively. It can be predicted that the performance of GEMM is computationally bound for all precisions on the GPU as expected in Section 2.5. Therefore, the performance of triple-precision subroutines is approximately equal to that of quadruple-precision subroutines because both subroutines perform the same DD arithmetic. When $N = 2,048$, quadruple-precision GEMM attains approximately 22.4 GDDFlops and reaches approximately 87% of 25.76 GDDFlops which is the theoretical peak performance. On the other hand, the performance of CUBLAS's double-precision GEMM is approximately 313.4 GFlops and only reaches approximately 61% of the theoretical peak performance. Tan et al. [33] reported the dif-

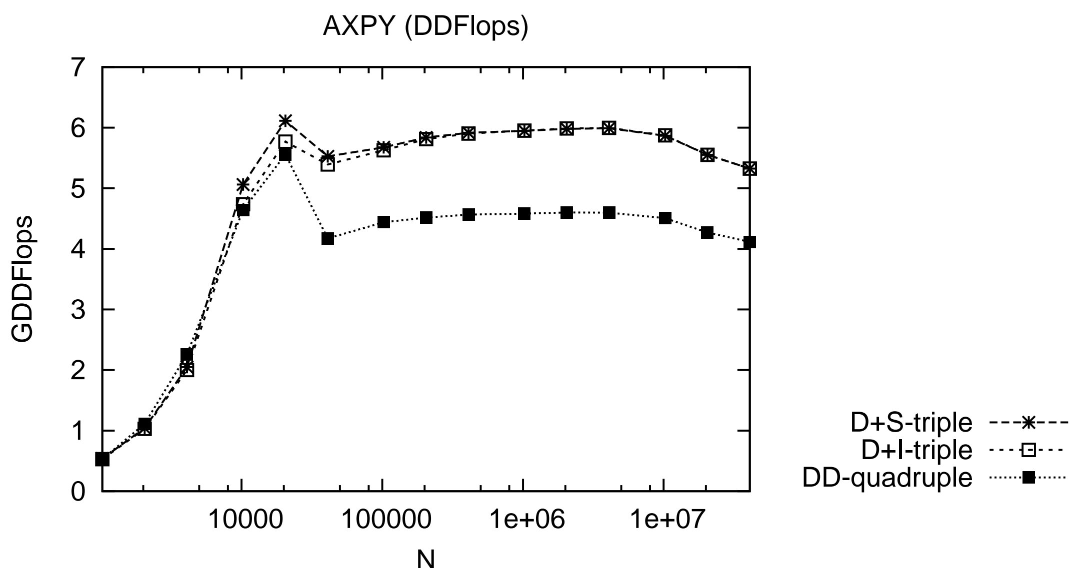


Figure 2.9: Performance of triple- and quadruple-precision AXPY (DDFlops: DD-type floating-point operations per second)

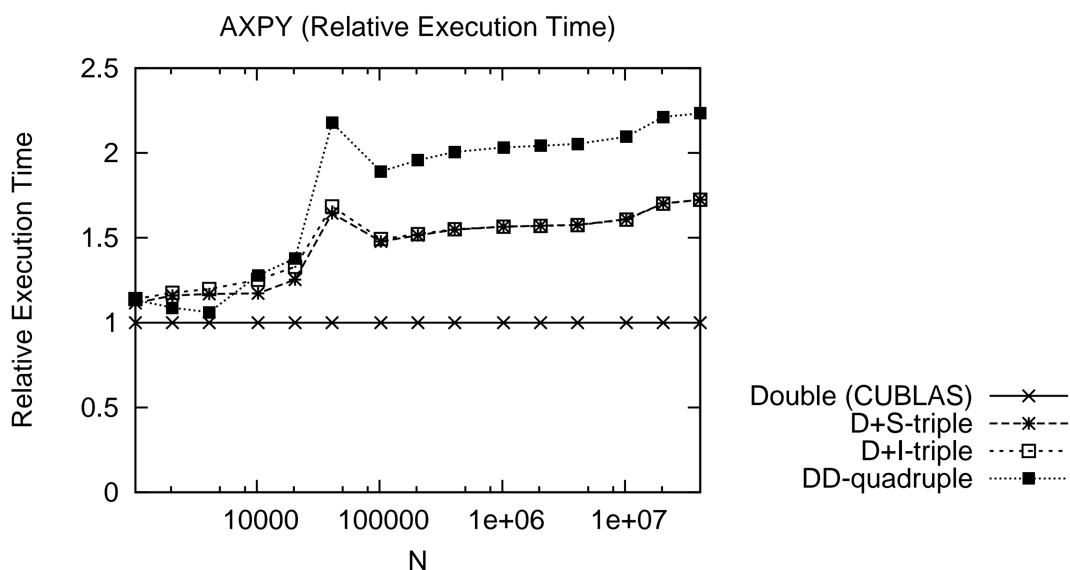


Figure 2.10: Relative execution time of AXPY (value is the multiple of the execution time of the double-precision subroutine)

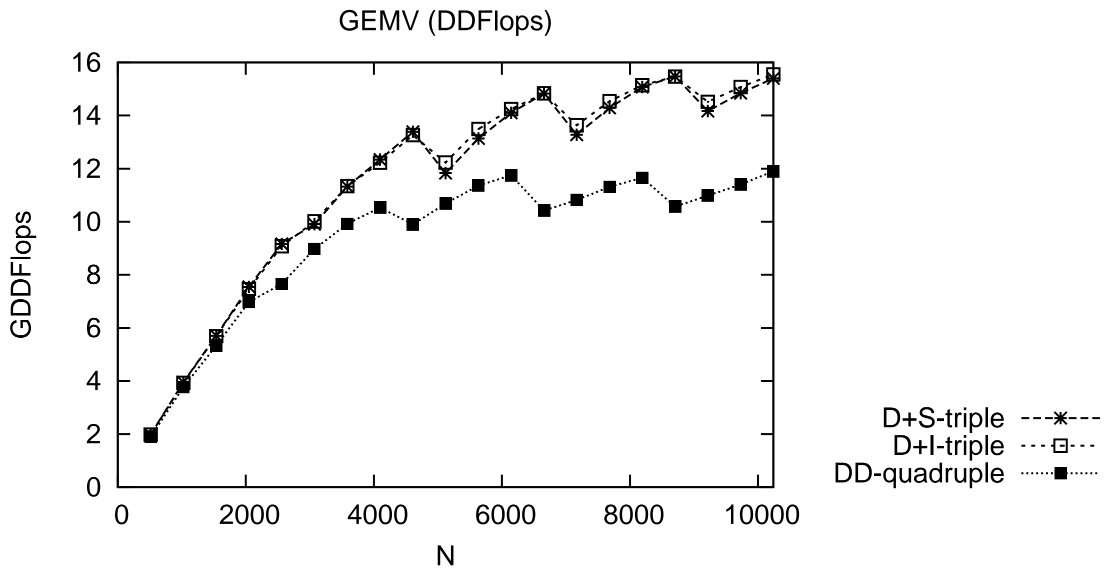


Figure 2.11: Performance of triple- and quadruple-precision GEMV (DDFlops: DD-type floating-point operations per second)

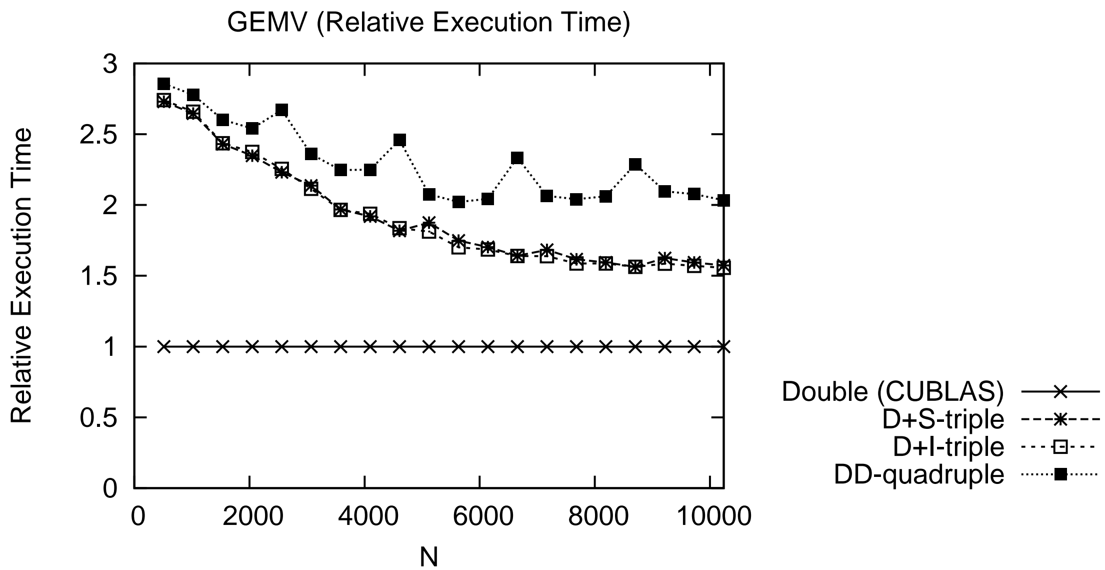


Figure 2.12: Relative execution time of GEMV (value is the multiple of the execution time of the double-precision subroutine)

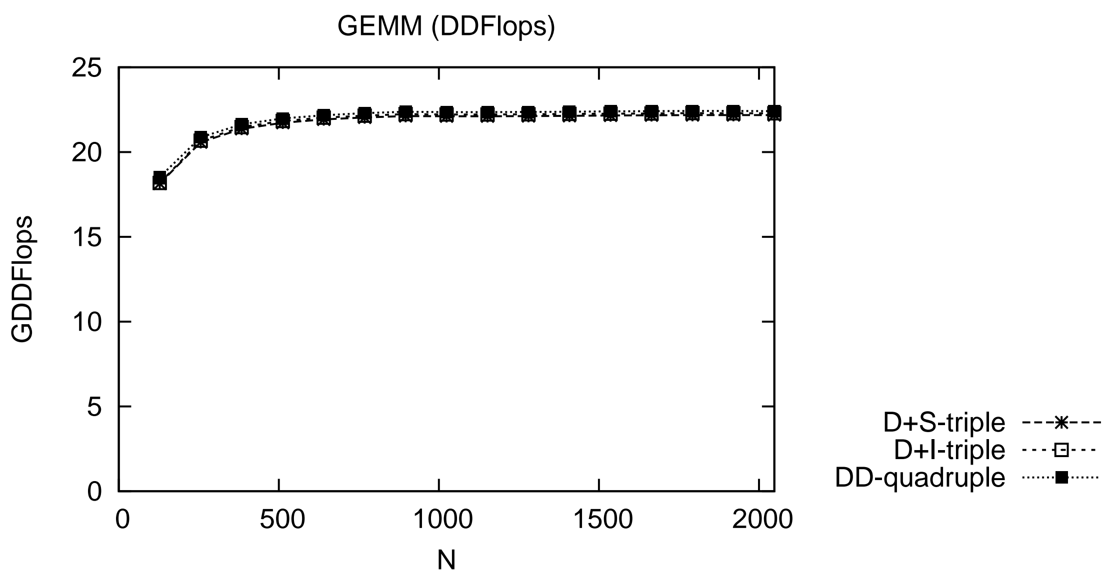


Figure 2.13: Performance of triple- and quadruple-precision GEMM (DDFlops: DD-type floating-point operations per second)

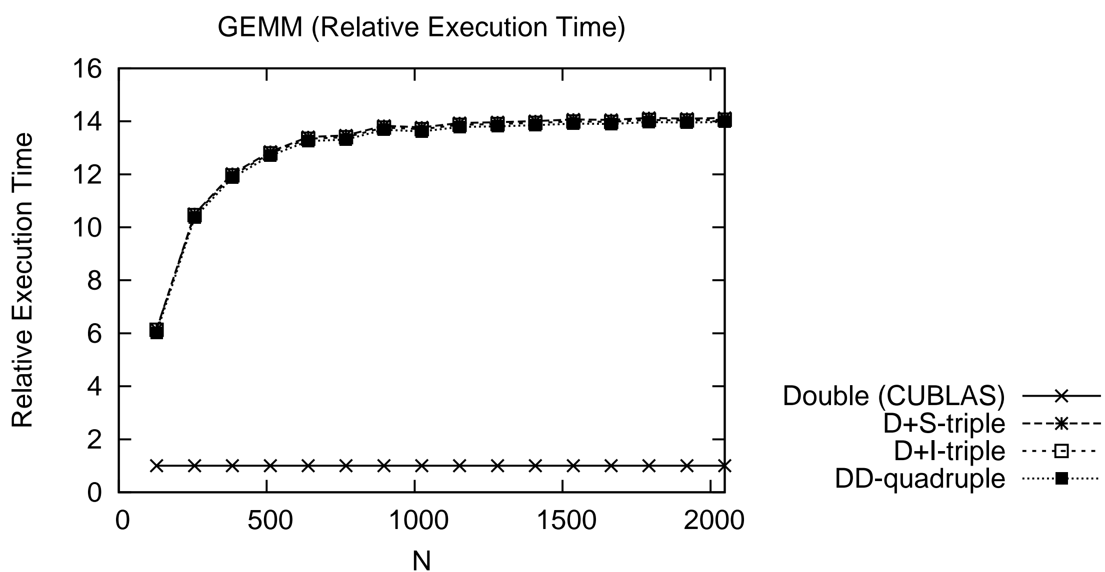


Figure 2.14: Relative execution time of GEMM (value is the multiple of the execution time of the double-precision subroutine)

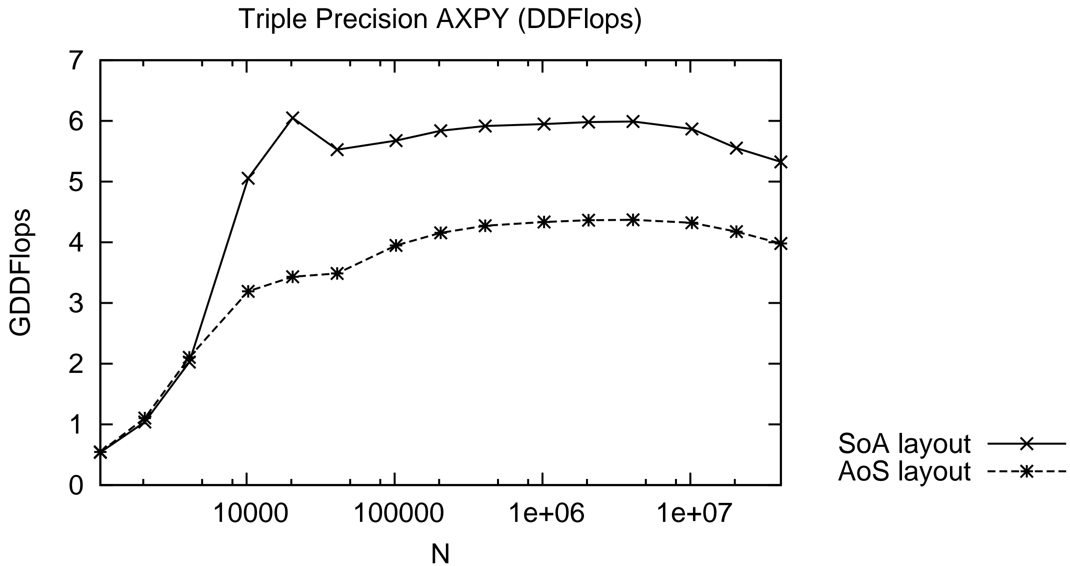


Figure 2.15: Performance comparison of AoS (Array-of-Structures) and SoA (Structure-of-Arrays) layouts on D+S-type triple-precision AXPY

ficulty with the optimization for double-precision GEMM on Fermi architecture GPUs. However, the use of DD arithmetic increases the density of arithmetic instructions per memory access and, as a result, higher execution efficiency was achieved. Hence, the execution times for triple- and quadruple-precision subroutines are approximately 14 times that of the double-precision subroutine, despite the fact that DD arithmetic requires 20 times the execution time of double-precision arithmetic in theory.

2.6.3 AoS Layout vs. SoA Layout

Figure 2.15 shows the performance comparison of the D+S-type triple-precision AXPY using the AoS and SoA layouts. The performance of AXPY is memory bound; therefore, it is greatly affected by memory access performance. This result indicates that the time taken by the SoA layout is approximately 1.3 times less than that of the AoS layout.

2.6.4 D+S Arithmetic v.s. DD Arithmetic

Here the performances of triple-precision GEMM using D+S and DD arithmetic are compared.

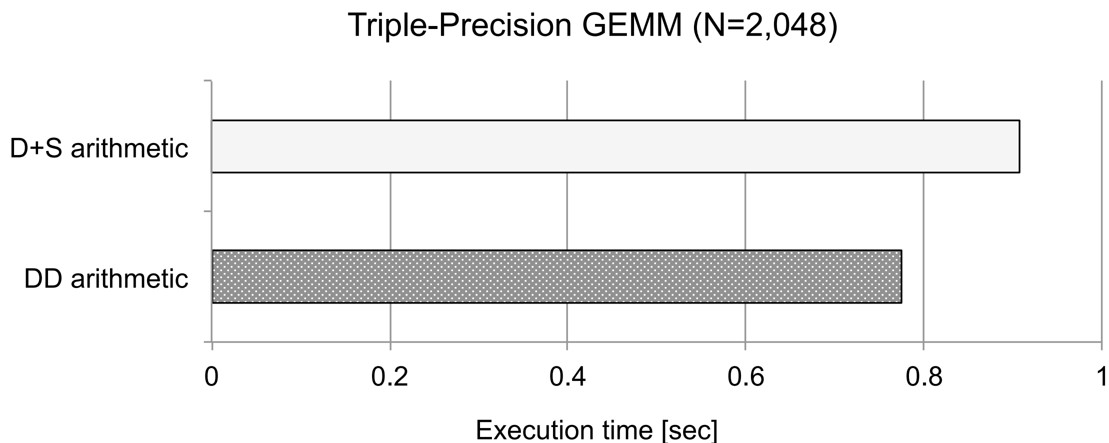


Figure 2.16: Performance comparison of D+S and DD arithmetic on GEMM

One D+S-type floating-point operation is defined as "1 DSFlop" and the 1 DSFlop per second is defined as "1 DSFlops". For D+S arithmetic, triple-precision multiply-add operations (2 DSFlops) are performed in $26 + 23 = 49$ [cycle] on the Tesla M2050 GPU, as shown in Table 2.2. Therefore, the theoretical peak performance of D+S arithmetic on the GPU is 1.15[GHz] $\times$ 14[SM] $\times$ 32[CUDACore] $\times$ (2[DSFlop]/49[cycle]) $\approx$ 21.03[GDSFlops]. This value is approximately 1/1.22 times that of the theoretical peak performance of DD arithmetic.

Figure 2.16 shows the execution times for triple-precision GEMM using D+S and DD arithmetic on the GPU when $N = 2,048$. The implementation using D+S arithmetic is approximately 1.28 times slower than that using DD arithmetic. This performance gap is equivalent to the theoretical peak performance gap of the D+S and DD arithmetic of approximately 1.22 times. The performance efficiencies of triple-precision GEMM with D+S and DD arithmetic are approximately 90% and 86% of the theoretical peak, respectively. On AXPY and GEMV, the performance with D+S and DD arithmetic is approximately equal because the performance of the two subroutines is memory bound.

2.6.5 Accuracy Evaluation

The results of the accuracy comparison among double-, triple-, and quadruple-precision subroutines on GEMM are shown in Table 2.5. This table shows the error relative to the octuple-precision result by the equivalent subroutine of MBLAS on CPUs. In the evaluation, the input is uniform random double-precision numbers from 0 to 1. Note that the triple-precision subroutines use DD arithmetic.

Table 2.5: Relative error to octuple-precision result (input: uniform random numbers in the range of 0 to 1 in double-precision)

	GEMV		GEMM	
	$N = 100$	$N = 1,000$	$N = 100$	$N = 1,000$
Double-precision	2.77E-16	4.60E-16	2.70E-16	7.83E-16
D+S-type triple-precision	8.20E-25	1.36E-24	8.75E-25	1.34E-24
D+I-type triple-precision	6.36E-24	1.16E-23	6.93E-24	1.07E-23
DD-type quadruple-precision	1.92E-32	6.57E-32	2.14E-32	6.45E-32

The error of the D+I-type triple-precision is approximately eight to nine times that of the D+S-type triple-precision because the significand bits of the D+I-type format is 3 bits smaller than that of the D+S-type format.

2.7 Related Work

A quadruple-precision GEMM subroutine has been implemented on GPUs. Nakasato [38] implemented this using DD arithmetic on ATI's GPUs. Nakata et al. also implemented such GEMM on an NVIDIA Tesla C2050 GPU [39]. The two papers focused on the optimization of GEMM and achieved higher execution efficiency than the quadruple-precision GEMM implemented in this chapter. However, there is no implementation of the other subroutines and discussion of the performance. I have presented the performance of all levels of BLAS subroutines on GPUs and shown that the performance of quadruple-precision AXPY and GEMV using DD arithmetic is memory bound.

There are BLAS implementations using DD arithmetic for CPUs. MPACK [40] is a multiple-precision LAPACK that includes a multiple-precision BLAS, MBLAS. MPACK uses two existing high-precision arithmetic libraries: the GNU multiple precision library (GMP) [8] and MPFR [41] for arbitrary-precision operations, and the QD library for octuple- and quadruple-precision operations. XBLAS [42] is an extended precision BLAS implementation using DD arithmetic internally where both the input and output are double-precision.

2.8 Conclusion

This chapter showed the implementation and performance of triple- and quadruple-precision BLAS subroutines: AXPY, GEMV and GEMM on the NVIDIA Tesla M2050 Fermi architecture GPU. For quadruple-precision subroutines, DD arithmetic was used. For triple-precision subroutines, I have proposed new methods to store triple-precision floating-point values: D+S-type and D+I-type triple-precision formats. The D+S-type has a 75-bit significand and 11-bit exponent, and the D+I-type has a 72-bit significand and 8-bit exponent. I have also proposed D+S arithmetic, but it is slower than DD arithmetic. Therefore, DD arithmetic was also used for the computation of triple-precision values.

This chapter revealed the relative time cost of the triple- and quadruple-precision BLAS subroutines on the GPU. On the BLAS subroutines, DD arithmetic requires 20 times the execution time of double-precision arithmetic in theory. However, since the GPU has relatively high floating-point performance compared to the memory bandwidth, the performance of triple- and quadruple-precision AXPY and GEMV is limited by the bandwidth of the global memory, and the computation time of DD arithmetic is hidden by the memory access time. As a result, the execution times of triple- and quadruple-precision subroutines are close to 1.5 and 2 times more than that of the double-precision subroutines, respectively. In such cases, the triple- and quadruple-precision operations by software achieve sufficient performance without hardware support of the triple- and quadruple-precision arithmetic. On GEMM, the performance is computationally bound and the execution times for the triple- and quadruple-precision subroutines are approximately 14 times that of the double-precision subroutine because of the low execution efficiency of CUBLAS's double-precision subroutine. The use of DD arithmetic increases the density of arithmetic instructions per memory access, so as a result, higher execution efficiency was achieved.

I have proposed new methods of triple-precision operations for linear algebra operations on GPUs as a new choice of extended precision between double- and quadruple-precision. Triple-precision operations may be effective in cases where double-precision is insufficient and quadruple-precision is not necessary, but triple-precision is sufficient. The triple-precision floating-point formats that are proposed in this paper realized a triple-precision linear algebra operation which is faster than the quadruple-precision operation in cases where the performance of the operation is memory bound on both triple- and quadruple-precision. In such cases, the use of the triple-precision formats reduces the data size and the execution time to 3/4 that of quadruple-precision formats. It is predicted that memory bandwidth bottleneck

will be tight in exascale computing [43]. In other words, the Bytes/Flop ratio of processors and systems is becoming smaller. In such environments, many operations are becoming memory bound rather than computationally bound. For GPU clusters, the bandwidth of the PCI Express bus (PCIe) is 8 GB/s; therefore, the performance of the internode communication between GPUs may be limited by the bandwidth of PCIe. Moreover, in large-scale computing, an accumulation of rounding errors can become a more serious problem. Therefore, triple-precision floating-point operations are predicted necessary for the emerging exascale computing era.

For future work, the performance evaluation and the utilization of triple- and quadruple-precision operations in actual applications is expected.

Chapter 3

Optimization of Sparse Matrix-vector Multiplication on NVIDIA Kepler Architecture GPUs

3.1 Introduction

Sparse Matrix-Vector multiplication (SpMV), that performs $y = Ax$ (where x and y are vectors and A is a sparse matrix) is one of the most important operations in scientific and engineering computing. In general, in order to save memory, a sparse matrix is stored in two kinds of arrays: a data array which only stores the non-zero elements of the matrix and index arrays which store the addresses of the non-zero elements. Thus, sparse matrix operations require indirect memory access which is complex compared to the dense operations. Moreover, various kinds of distributions of the non-zero elements are considered. Therefore, efficient implementation of SpMV requires a large number of optimization techniques.

This chapter presents optimization techniques for SpMV for the Compressed Row Storage (CRS) format on NVIDIA Kepler architecture GPUs using CUDA. The CRS format is one of the most widely used storage formats for sparse matrices. In the CRS format, a sparse matrix is stored into the data array by scanning the matrix in the row direction and two index arrays: an index array, which represents the column number of the non-zero elements in the data array, and a pointer array, which points to the first non-zero element of each row (see Figure 3.1).

Various storage schemes have been proposed for GPUs. For example, Bell and

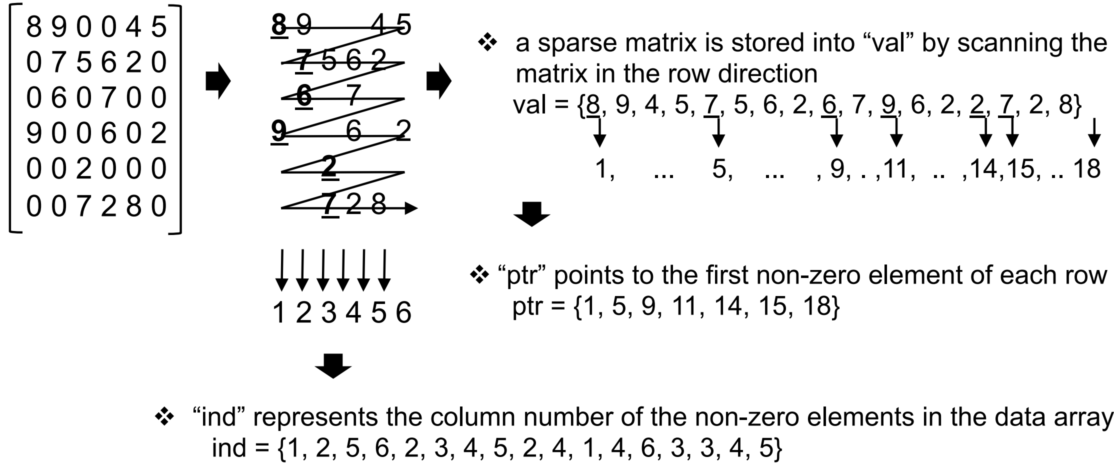


Figure 3.1: CRS format

Garland [44] proposed a new storage format, HYB (Hybrid), which combines the existing ELL (Ellpack) and COO (Coordinate) formats and was implemented on GPUs. The HYB format outperforms the CRS format. Many other storage schemes have been implemented on GPUs so far [45] [46] [47]. On the other hand, an auto-tuning method is effective for SpMV since the optimal storage scheme is dependent upon the distribution of the non-zero elements of the matrix. For instance, Kubota and Takahashi [48] presented an auto-tuning method which selects the optimal storage format using a percentage and variability of non-zero elements.

However, an SpMV routine for the CRS format that can perform well for a wide variety of matrices is still necessary, especially in numerical libraries. It may be necessary to convert the storage format from CRS to other formats in some cases. On auto-tuning methods, it may be necessary to scan the matrix to determine the optimal storage format in advance. In fact, the SpMV routine for the CRS format is still provided in various numerical libraries such as the NVIDIA cuSPARSE library [22] for sparse matrix operations on GPUs for CUDA environments.

In this chapter, I will implement a fast SpMV routine for the CRS format for Kepler architecture GPUs. The implementation is based on an existing method proposed for the Fermi architecture, which is an earlier generation of the GPU, and takes advantage of some of the new features of the Kepler architecture. This chapter will be organized as follows: Section 3.2 will describe related work. Section 3.3 will briefly introduce the Kepler architecture. Section 3.4 will describe the implementation. Section 3.5 will show the effects of the optimization techniques and compare the performance of the SpMV routine to the NVIDIA cuSPARSE library. Finally, Section 3.6 will conclude this chapter.

3.2 Related Work

Two methods to implement SpMV for the CRS format on GPUs have been presented by Bell and Garland [44]. The first one, the CRS-scalar method, performs the calculation of each row of a matrix (calculation of one element of the vector y in $y = Ax$) using one thread per row. The second method, the CRS-vector method, assigns multiple threads to calculate a single row. The CRS-scalar method can be easily implemented with minimum changes from the CPU code, but it may be not suitable for efficient memory access on GPUs. On GPUs, several memory access transactions are coalesced into a single transaction when consecutive threads access consecutive memory addresses. The CRS-vector method can offer more efficient memory access patterns than the CRS-scalar method. Bell and Garland allocated 32 threads for the calculation of one row.

In the CRS-vector method, if the number of non-zero elements per row is less than 32, reducing the number of calculation threads per row may improve the performance. Baskaran and Bordawekar [49] used 16, instead of 32, threads to compute a row with CRS-vector. Guo and Wang [50] proposed a method that switches the number of threads to either 16 or 32 based on the characteristics of the input matrix. El Zein and Rendell [51] switched between the CRS-scalar and CRS-vector methods based on the number of non-zero elements per row. Reguly and Giles [52] improved the performance of the CRS-vector method by selecting the optimal number of threads from among 1, 2, 4, 8, 16 and 32 in proportion to the average number of non-zero elements per row. Furthermore, Yoshizawa and Takahashi [53] selected the optimal number of threads from among 1, 2, 4, 8, 16 and 32 based on the maximum number of non-zero elements per row. The strategy of varying the number of threads from 1–32 based on the number of non-zero elements per row is effective. The average-based approach is preferred because the average number of non-zero elements per row can be calculated without pre-scanning the input matrix. Therefore, the implementation is based on Reguly and Giles's method.

The Kepler architecture was launched by NVIDIA in 2012. Davis and Chung [54] compared the performance of the Kepler and the Fermi, which is an earlier generation of GPU, using the GeForce series of GPUs. Their report shows that the Kepler is slower than the Fermi, but they used the same program on the both GPUs. Most existing reports have focused on an earlier generation of GPU. There is no research regarding the implementation and evaluation of optimization techniques for the Kepler architecture of GPUs. I will target a Tesla K20 GPU which is based on the Kepler architecture.

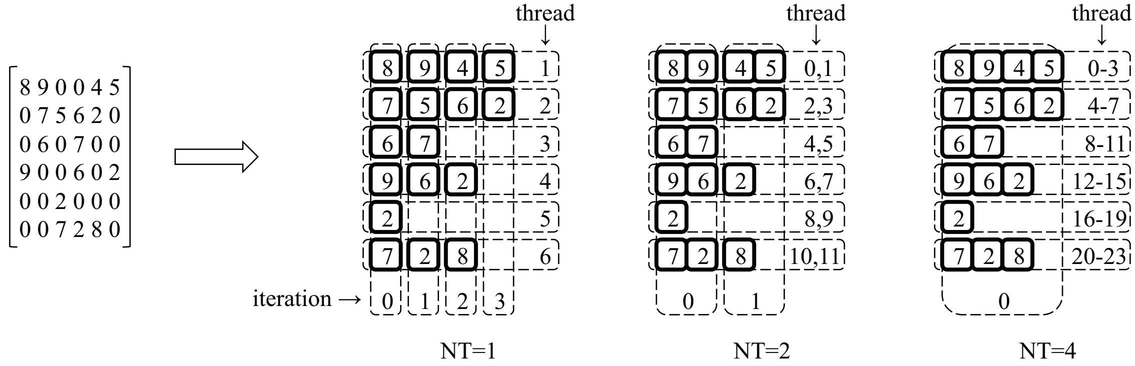


Figure 3.2: Thread mapping for the cases of NT=1, 2 and 4 on the CRS-vector method

3.3 Kepler architecture GPUs

An overview of the Kepler architecture can be found in the White Paper [19] by NVIDIA. The most major change from the previous generation of Fermi architecture GPUs is that the streaming multiprocessor, called SM on the Fermi architecture, has been replaced with an updated version called SMX. The SM has 32 CUDA cores, but that number has been increased to 192 on the SMX. As a result, the maximum number of warps, threads, and thread blocks per multiprocessor have also increased. MaxGridDimX (the number of thread blocks in the x-direction that can be defined in a grid) has also increased from 65,535 to 2,147,483,647. In addition, the total number of registers has doubled to 65,536 and the total number of registers available to a thread has also increased from 63 to 255. Moreover, the execution efficiency of double-precision operations has been improved from the Fermi architecture by improving the warp scheduler.

On the other hand, the Kepler architecture supports some new features. Among them, a new 48KB read-only data cache and new shuffle instructions will be expected to improve the performance of SpMV. The 48KB read-only data cache can only be used to load data that does not change the value during the kernel execution. This cache was accessible by using the texture unit on earlier generations of GPUs, but has seen major improvements on the Kepler architecture. The shuffle instructions are new instructions used to access a value different threads in the same warp. I will utilize the 48KB read-only data cache, shuffle instructions and expansion of the MaxGridDimX to optimize SpMV on the Kepler architecture. The next section will explain the details of the three optimization techniques.

3.4 Implementation

This section describes the implementation. The SpMV routine, implemented using double-precision, computes $y = \alpha Ax + \beta y$, which is compatible with the SpMV routine of the cuSPARSE library. The implementation is based on Reguly and Giles's method which is based on the CRS-vector method and selects the number of threads for the calculation of a single row (NT) from among NT = 1, 2, 4, 8, 16 and 32 in proportion to the average number of non-zero elements per row. The average number of non-zero elements per row is available in advance without pre-scanning the input matrix.

Figure 3.2 shows a conceptual diagram of the thread mapping for the cases of NT=1, 2 and 4 on the CRS-vector method. When NT=1, it is equivalent to the CRS-scalar method. The CRS-vector method computes an inner product in the row direction using multiple threads. In Figure 3.2, "iteration" means a loop in the row direction. The NT can be up to 32 because thread synchronization is not required within a warp (=32 threads). Listing 3.1 shows the host code. In the host code, the NT is determined and the kernel codes for each NT are called.

Listing 3.2 shows the kernel code for the Fermi architecture. I further optimized the implementation for the Kepler architecture by (1) using the 48KB read-only data cache, (2) avoiding the outermost loop, (3) using shuffle instructions. Listing 3.3 shows the kernel code optimized for the Kepler architecture. Note that, the second for-loop in the kernel codes (Listings 3.2 and 3.3) is unrolled since the number of iterations which is the same as NT is determined in advance of the kernel launch. The following subsections will explain the details of the three optimization techniques for the Kepler architecture.

3.4.1 48KB Read-only Data Cache

The 48KB read-only data cache can be applied only to data that does not change the value during the execution of a kernel. The cache can be accessed via a texture unit by mapping data in global memory to texture memory, which can also be done on the earlier Fermi generation architecture as shown in Listing 3.2, but before Kepler using the cache required complex programs and had many limitations. However, starting with the Kepler architecture, the cache can be accessed directly from the SM with general load operations. Reading through the read-only data cache is performed using an independent path of the L1 cache path. The read-only data cache is automatically managed by the CUDA compiler by using "`const`" and "`__restrict__`" qualifiers to direct the compiler to pass data in this read-only cache

Listing 3.1: Host code of SpMV

```
1  int SpMV (char trans, int m, int n, int nnz, double alpha,
2          double *a_val, int *a_ptr, int *a_idx, double *x,
3          double beta, double *y) {
4      int NT, ntx, nbx;
5      float nnzrow = (float)nnz/(float)m;
6      NT = max(1, min(32, (int)pow(2., ceil(log2(nnzrow)))));
7      ntx = 128;
8      nbx = m / (ntx / NT) + ((m % (ntx / NT)) != 0);
9      dim3 threads (ntx);
10     dim3 grid (nbx);
11     if (trans == 'N') {
12         if (NT == 32) {
13             cudaFuncSetCacheConfig(SpMV_kernel32,
14                 cudaFuncCachePreferL1);
15             SpMV_kernel32 <<< grid, threads >>>
16                 (m, alpha, a_val, a_ptr, a_idx, x, beta, y);
17         } else if (NT == 16) {
18             ....
19         } else if (NT == 2) {
20             ....
21         } else {
22             ....
23         }
24     }
25 }
```

as arguments to kernel functions. The implementation utilized the 48KB read-only data cache to read the vector x from the global memory in the implementation.

3.4.2 Avoid Outermost Loop

On the Kepler architecture, MaxGridDimX (the number of thread blocks that can be defined in the direction of dimension x in a grid) was extended from 65,535 to 2,147,483,647. As a result, the implementation can avoid the outermost loop in the CRS-vector method to calculate an index of a vector by using a thread ID and a thread block ID.

Listing 3.2: Kernel code of SpMV for the Fermi architecture

```
1  texture <int2, cudaTextureType1D,
2      cudaReadModeElementType> tex_x;
3  static __inline__ __device__ double fetch_x (const int &i) {
4      register int2  v = tex1Dfetch(tex_x, i);
5      return __hiloint2double(v.y, v.x);
6  }
7
8  __global__ void SpMV_kernelNT_Fermi (int m, double alpha,
9                      double *a_val, int *a_ptr, int *a_idx,
10                     double *x, double beta, double *y) {
11     int i;
12     int tx = threadIdx.x;
13     int tid = blockDim.x * blockIdx.x + tx;
14     int rowid = tid / NT;
15     int lane  = tid % NT;
16     __shared__ double vals[128];
17     while (rowid < m) {
18         vals[tx] = 0.0;
19         for (i = a_ptr[rowid] + lane;
20             i < a_ptr[rowid + 1]; i += NT)
21             vals[tx] += a_value[i] * fetch_x(a_index[i]);
22         for (i = NT / 2; i > 0; i >>= 1)
23             vals[tx] += vals[tx + i];
24         if (lane == 0)
25             y[rowid] = alpha * vals[tx] + beta * y[rowid];
26         __syncthreads ();
27         rowid += gridDim.x * blockDim.x / NT;
28     }
29 }
```

In the CRS-vector method, RowMax, the maximum dimension of a matrix that can be calculated, is obtained by setting RowMax = MaxGridDimX × BlockDim.x / NT. "BlockDim.x" means the number of threads for the x dimension in a thread block. In the implementation, the optimal size of the BlockDim.x is 128. RowMax becomes minimum when NT = 32. Thus on the Fermi architecture, RowMax

Listing 3.3: Kernel code of SpMV for the Kepler architecture

```
1  __global__ void SpMV_kernelNT_Kepler (int m, double alpha,
2          double *a_val, int *a_ptr, int *a_idx,
3          const double * __restrict__ x,
4          double beta, double *y) {
5      int i, int val_hi, val_lo;
6      int tx = threadIdx.x;
7      long long tid = blockDim.x * blockIdx.x + tx;
8      long long rowid = tid / NT;
9      int lane  = tid % NT;
10     double val;
11     if (rowid < m) {
12         val = 0.0;
13         for (i = a_ptr[rowid] + lane;
14             i < a_ptr[rowid + 1]; i += NT)
15             val += a_val[i] * x[a_idx[i]];
16         for (i = NT / 2; i > 0; i >>= 1) {
17             val_hi = __double2hiint(val);
18             val_lo = __double2loint(val);
19             val += __hiloint2double(
20                     __shfl_xor(val_hi, i, 32),
21                     __shfl_xor(val_lo, i, 32));
22         }
23         if (lane == 0)
24             y[rowid] = alpha * val + beta * y[rowid];
25     }
26 }
```

$= 65,535 \times 128/32 = 262,140$. In order to compute a vector longer than 262,140, it is necessary to recalculate the vector's address using a loop: recalculation of rowid and the outermost while-loop are required as shown in Listing 3.2 instead of the outermost if-statement in Listing 3.3. In addition, a thread synchronization instruction was required when using shared memory for reduction (the implementation can avoid this thread synchronization by declaring the shared memory with the volatile suffix, but this method performs worse than using thread synchronization).

On the other hand on the Kepler architecture, RowMax has increased: RowMax

$= 2,147,483,647 \times 128/32 = 8,589,934,588$. The capacity of global memory on current GPUs is less than 10GB. RowMax is equivalent to a 32GB single-precision vector. Therefore, MaxGridDimX on the Kepler architecture is sufficient to point to the index of any vector that can be loaded with current GPUs and the outermost loop is not required.

3.4.3 Shuffle Instruction

The CRS-vector method computes a reduction at the second for loop shown in Listing 3.3. When NT $\geq$ 2 on the CRS-vector method, the NT threads perform the reduction within a single warp. On the earlier architectures, the operation must be performed using shared memory to exchange values among the threads in a warp. The Kepler architecture can access a value on any other thread within the warp without shared memory by using shuffle instructions. There are 4 types of shuffle instructions that are supported starting with Kepler: indexed any-to-any (`__shfl`), shift right to N-th neighbour (`__shfl_up`), shift left to N-th neighbour (`__shfl_down`) and butterfly (XOR) exchange (`__shfl_xor`). Whereas shared memory requires separate load and store steps, shuffle instructions reduce this to a single step, and thus it can be expected that the shuffle instructions will outperform the equivalent shared memory instructions.

The implementation used the butterfly exchange shuffle instruction for the reduction. Because the shuffle instructions support only a 32-bit value, moving 64-bit data requires two 32-bit movements. A 64-bit double value is converted into two 32-bit integer values and exchanged the two 32-bit values with the shuffle instruction, and then the result, which consists of the two 32-bit integer values, is converted to a single 64-bit double value.

3.5 Performance Evaluation

3.5.1 Evaluation Methods

I used an NVIDIA Tesla K20 Kepler architecture GPU. The evaluation environment is shown in Table 3.1. The "-arch sm_35" compiler flag for nvcc is required in order to use the features of the Kepler architecture. This section evaluated the GPU kernel execution time. To measure the performance accurately, a routine is repeatedly executed for at least one second at least 3 times, then computed the average execution time.

Table 3.1: Evaluation Environment

CPU	Intel Xeon E3-1230 3.20GHz
RAM	16 GB (DDR3)
OS	CentOS 6.3 (kernel: 2.6.32-279.14.1.el6.x86_64)
GPU	Tesla K20 (5GB, GDDR5, ECC-enabled)
CUDA	CUDA 5.0 (Driver version: 304.54)
Compiler	gcc 4.4.6 (-O3), nvcc 5.0 (-O3 -arch sm_35)

All input values other than the matrix A are composed of uniform random numbers. For the input sparse matrices, 200 matrices are randomly selected from the University of Florida Sparse Matrix Collection [21]. The selected matrices are all real square matrices that have a different number of non-zero elements or a different number of rows. The number of rows varies between 1,813–5,558,326, and the number of non-zero elements varies between 4,257–117,406,044.

In order to investigate the effect of each optimization technique for the Kepler architecture used in this research, I implemented and evaluated the following five implementations:

- Ver. Fermi: Optimized for the Fermi architecture (does not use Vers. A–C optimizations)
- Ver. A: Only using 48KB read-only data cache
- Ver. B: Only avoiding outermost loop
- Ver. C: Only using shuffle instruction
- Ver. Kepler: Optimized for the Kepler architecture (uses Vers. A–C optimizations)

Ver. Fermi is optimized for the Fermi architecture and is the same implementation shown in Listing 3.2. Vers. A–C is applied each optimization one by one. Note that Ver. Fermi, B, and C take advantage of the texture cache by mapping the data on global memory to texture memory for loading the vector x, like on the Fermi architecture. Ver. A and Ver. Kepler use the read-only data cache instead of the texture cache. Ver. Kepler is the final version and is optimized for the Kepler architecture and is the same implementation shown in Listing 3.3.

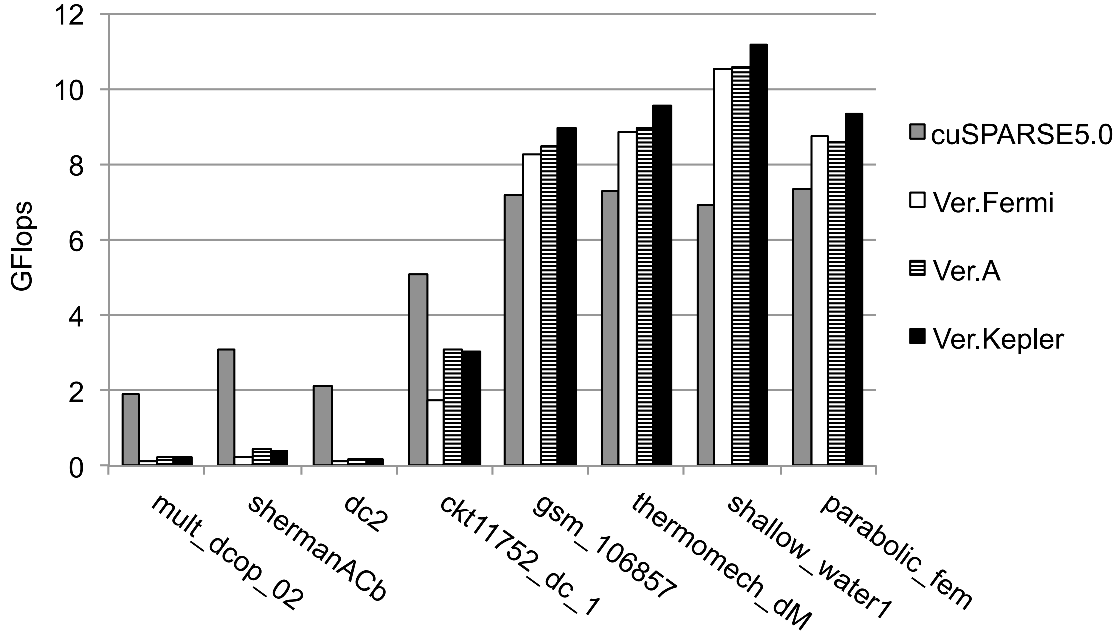


Figure 3.3: Flops performance for the cases with the 4 largest and 4 smallest speedup of Ver. A (only using 48KB read-only data cache) to Ver. Fermi

Table 3.2: Properties of the matrices shown in Figure 3.3

Matrix	Rows	Nonzeros	% of Nonzeros	Nonzeros/Row	NT
mult_dcop_02	25187	193276	0.03050	7.67	8
shermanACb	18510	145149	0.04240	7.84	8
dc2	116835	766396	0.00561	6.56	8
ckt11752_dc_1	49702	333029	0.01350	6.70	8
...	...	...	...	...	...
gsm_106857	589446	21758924	0.00626	36.91	32
thermomech_dM	204316	1423116	0.00341	6.97	8
shallow_water1	81920	327680	0.00488	4.00	4
parabolic_fem	525825	3674625	0.00133	6.99	8

3.5.2 Result

Ver. A: only using 48KB read-only data cache

Figure 3.3 shows the Flops performance for the cases with the 4 largest and 4 smallest speedup of Ver. A to Ver. Fermi on the 200 matrices. Table 3.2 shows the properties of the 8 matrices. The maximum speedup of Ver. A to Ver. Fermi is

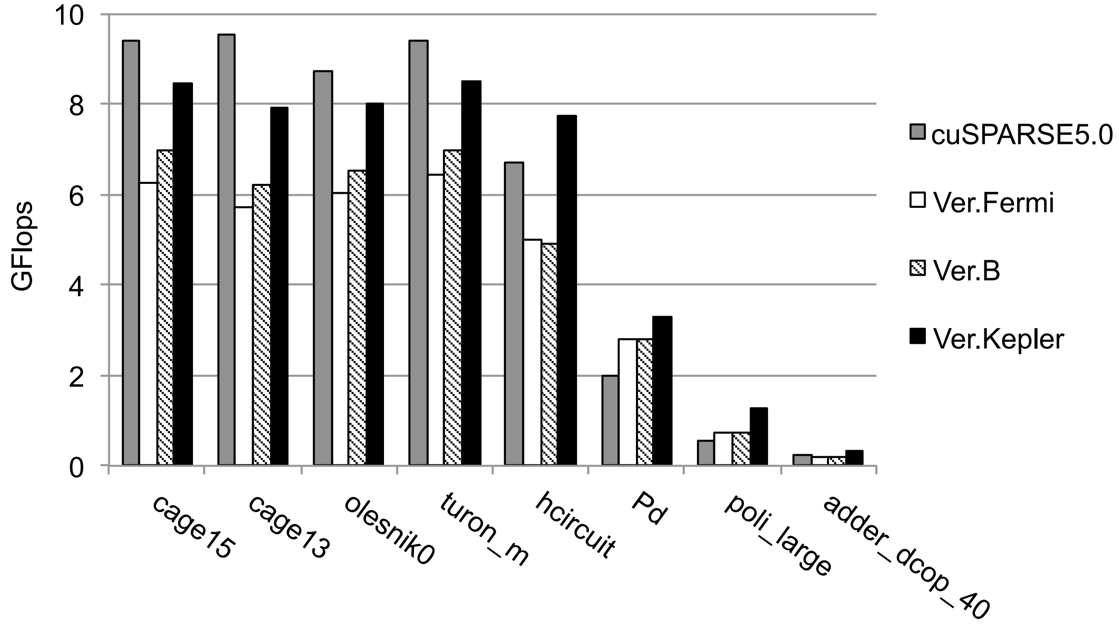


Figure 3.4: Flops performance for the cases with the 4 largest and 4 smallest speedup of Ver. B (only avoiding outermost loop) to Ver. Fermi

Table 3.3: Properties of the matrices shown in Figure 3.4

Matrix	Rows	Nonzeros	% of Nonzeros	Nonzeros/Row	NT
cage15	5154859	99199551	0.00037	19.24	32
cage13	445315	7479343	0.00377	16.80	32
olesnik0	88263	744216	0.00955	8.43	16
turon_m	189924	1690876	0.00469	8.90	16
...	...	...	...	...	...
hcircuit	105676	513072	0.00459	4.86	8
Pd	8081	13036	0.01996	1.61	2
poli_large	15575	33074	0.01363	2.12	4
adder_dcop_40	1813	11246	0.34214	6.20	8

approximately 1.78 times on “mult_dcop_02” and on the others on the top 4 cases also attain approximately 1.77 – 1.78 times speedups. Except the worst case of 0.98 times speed down on “parabolic_fem”, all the other cases on the 200 matrices attain speedup. On the Kepler architecture using the read-only cache improves the performance when compared to the implementation using the texture cache by accessing via a texture unit by mapping data in global memory to texture memory,

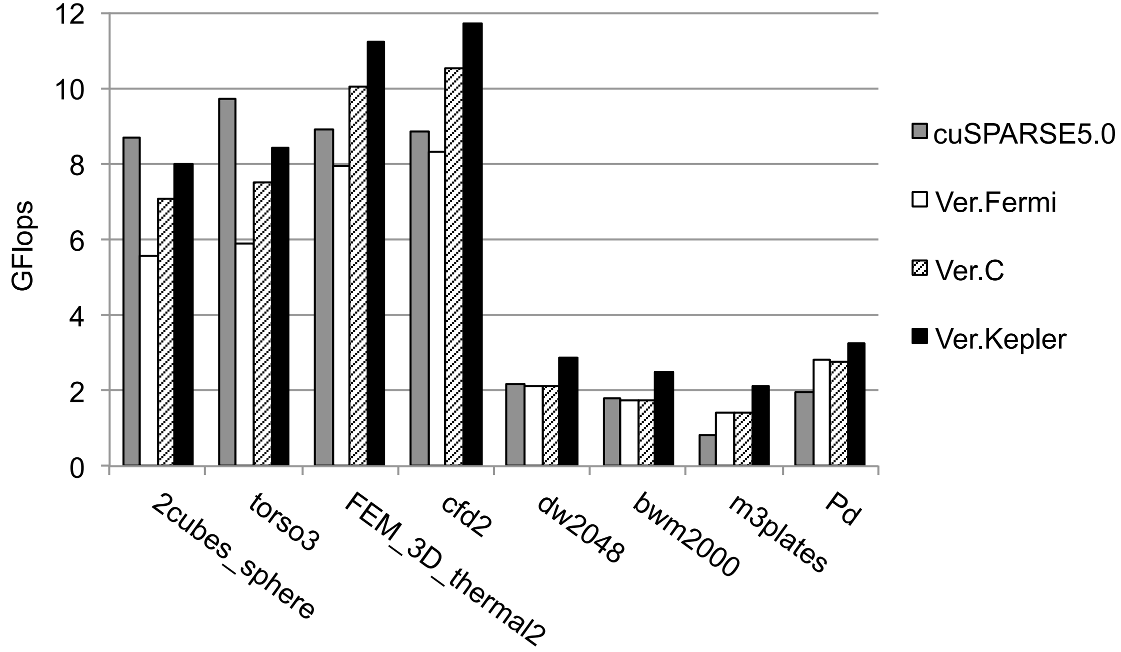


Figure 3.5: Flops performance for the cases with the 4 largest and 4 smallest speedup of Ver. C (only using shuffle instruction) to Ver. Fermi

Table 3.4: Properties of the matrices shown in Figure 3.5

Matrix	Rows	Nonzeros	% of Nonzeros	Nonzeros/Row	NT
2cubes_sphere	101492	1647264	0.01599	16.23	32
torso3	259156	4429042	0.00659	17.09	32
FEM_3D_thermal2	147900	3489300	0.01595	23.59	32
cfd2	123440	3087898	0.02027	25.02	32
...	...	...	...	...	...
dw2048	2048	10114	0.24114	4.94	8
bwm2000	2000	7996	0.19990	4.00	4
m3plates	11107	6639	0.00538	0.60	1
Pd	8081	13036	0.01996	1.61	2

like on the Fermi architecture.

Ver. B: only avoiding outermost loop

Figure 3.4 shows the Flops performance for the cases with the 4 largest and 4 smallest speedup of Ver. B to Ver. Fermi on the 200 matrices. Table 3.3 shows the

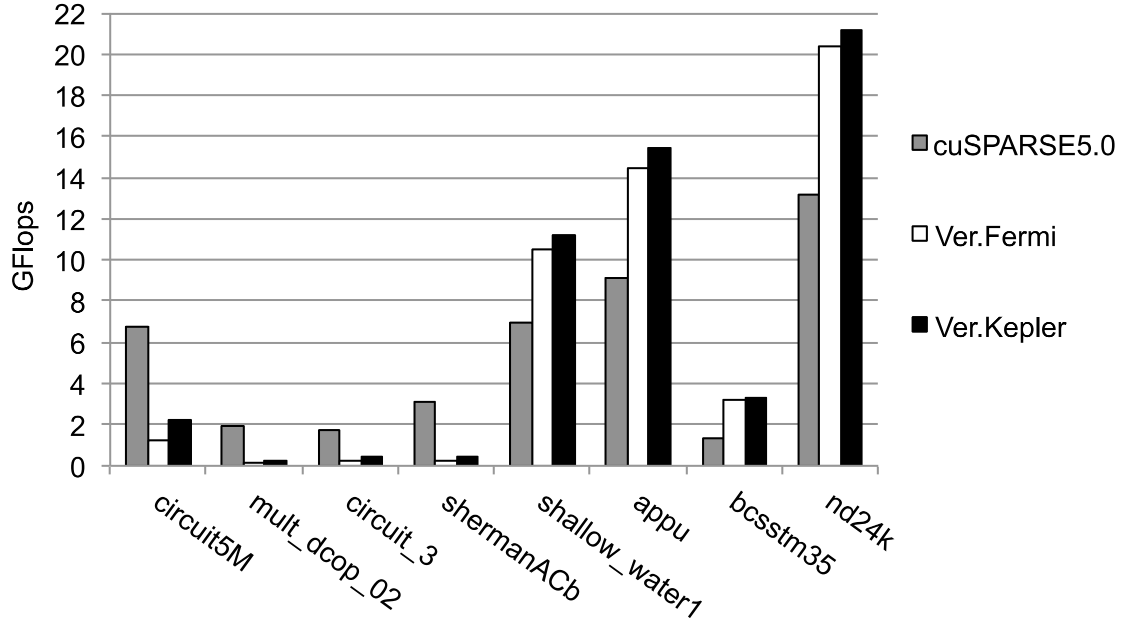


Figure 3.6: Flops performance for the cases with the 4 largest and 4 smallest speedup of Ver. Kepler to Ver. Fermi

Table 3.5: Properties of the matrices shown in Figure 3.6

Matrix	Rows	Nonzeros	% of Nonzeros	Nonzeros/Row	NT
circuit5M	5558326	59524291	0.00019	10.71	16
mult_dcop_02	25187	193276	0.03047	7.67	8
circuit_3	12127	48137	0.03273	3.97	4
shermanACb	18510	145149	0.04236	7.84	8
...	...	...	...	...	...
shallow_water1	81920	327680	0.00488	4.00	4
appu	14000	1853104	0.94546	132.36	32
bcsstm35	30237	20619	0.00226	0.68	1
nd24k	72000	28715634	0.55393	398.83	32

properties of the 8 matrices. The maximum and minimum speedups of Ver. B to Ver. Fermi are approximately 1.22 times on “cage15” and 0.99 times on “adder_dcop_40”, respectively. It is expected that avoiding the outermost loop is effective especially when Rows $> 262,140$, however, the clear relation between Rows and the effect was not found.

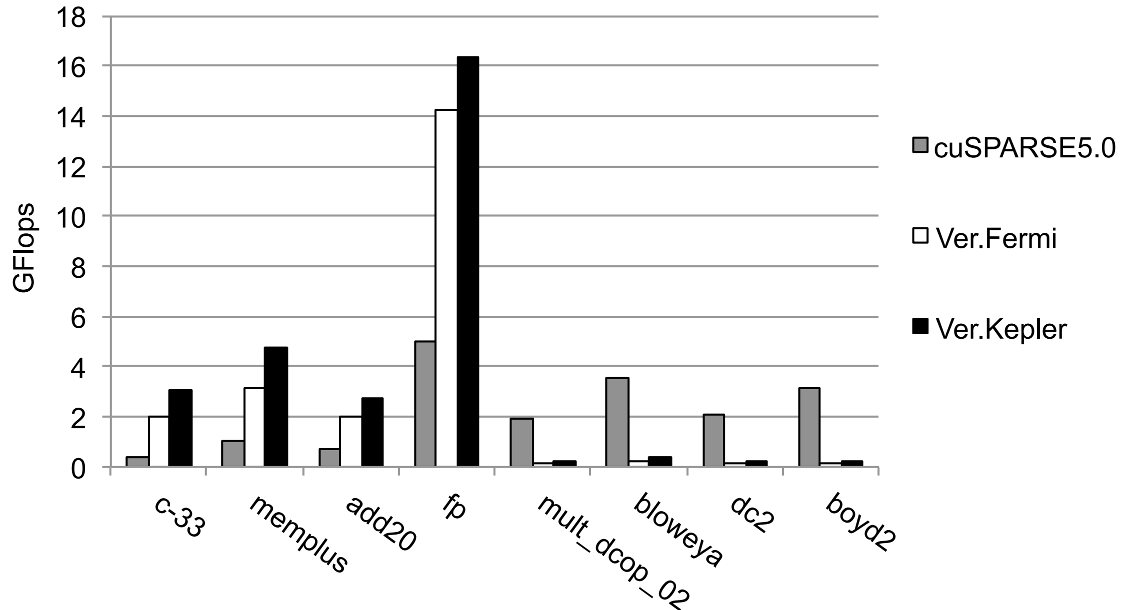


Figure 3.7: Flops performance for the cases with the 4 largest and 4 smallest speedup of Ver. Kepler to cuSPARSE's subroutine

Table 3.6: Properties of the matrices shown in Figure 3.7

Matrix	Rows	Nonzeros	% of Nonzeros	Nonzeros/Row	NT
c-33	6317	56123	0.14064	8.88	16
memplus	17758	126150	0.04000	7.10	8
add20	2395	17319	0.30193	7.23	8
fp	7548	848553	1.48941	112.42	32
...	...	...	...	...	...
mult_dcop_02	25187	193276	0.03047	7.67	8
bloweya	30004	150009	0.01666	5.00	8
dc2	116835	766396	0.00561	6.56	8
boyd2	466316	1500397	0.00069	3.22	4

Ver. C: only using shuffle instruction

Figure 3.5 shows the Flops performance for the cases with the 4 largest and 4 smallest speedup of Ver. C to Ver. Fermi on the 200 matrices. Table 3.4 shows the properties of the 8 matrices. The maximum and minimum speedups of Ver. C to Ver. Fermi are approximately 1.27 times on "2cubes_sphere" and approximately 0.99

times on “Pd”, respectively. On the top 4 matrices, the NT, which is the number of threads for the calculation of a single row, is 32 and larger than that on the 4 worst cases. The implementation selects the NT from among 1–32 in proportion to the Nonzeros/Row and the use of the shuffle instruction increases in proportion to the NT. Therefore, the effect of the shuffle instruction is relatively high on the matrices which have relatively large Nonzeros/Row.

Ver. Kepler: final version for Kepler architecture (uses three optimizations)

Figure 3.6 shows the Flops performance for the cases with the 4 largest and 4 smallest speedup of Ver. Kepler to Ver. Fermi on the 200 matrices. Table 3.5 shows the properties of the 8 matrices. The maximum and minimum speedups of Ver. Kepler to Ver. Fermi are approximately 1.78 times on “circuit5M” and 1.04 times on “nd24k”, respectively.

On the other hand, Ver. Kepler outperforms cuSPARSE’s subroutine for 174 of the 200 matrices. Figure 3.7 shows the Flops performance for the cases with the 4 largest and 4 smallest speedup of Ver. Kepler to cuSPARSE’s subroutine on the 200 matrices. Table 3.6 shows the properties of the 8 matrices. The maximum and minimum speedups of Ver. Kepler to cuSPARSE’s subroutine are approximately 7.60 times on “c-33” and 0.07 times on “boyd2”, respectively.

3.6 Conclusion

This chapter presented optimization techniques for SpMV for the CRS format on NVIDIA Kepler architecture GPUs using CUDA. The implementation is based on the existing method proposed for the Fermi architecture, an earlier generation of GPUs, and takes advantage of three new features of the Kepler architecture: a 48KB read-only data cache, shuffle instructions and expanding the MaxGridDimX. On the Tesla K20 Kepler architecture GPU on double-precision operations, the implementation optimized for the Kepler architecture is approximately 1.04 – 1.78 times faster than the implementation optimized for the Fermi architecture for the 200 matrices which are randomly selected from the University of Florida Sparse Matrix Collection. Especially, the use of read-only cache obtained the biggest speedup among the tree optimizations with minor code change. On programs using the texture cache implemented for the Fermi architecture, the read-only cache should be used instead of the texture cache. Furthermore, I has showed the implementation outperforms the SpMV routine for the CRS format of the cuSPARSE 5.0 for 174 of

the 200 matrices, and it is up to approximately 1.45 times faster than the SpMV routine. It can be concluded that the techniques shown in this chapter are effective for implementing a fast SpMV routine for the CRS format on Kepler architecture GPUs.

Chapter 4

Krylov Subspace Methods using Quadruple-Precision Arithmetic on GPUs

4.1 Introduction

The convergence of the Krylov subspace methods, which are iterative methods for solving linear systems, is significantly affected by rounding errors. Thus, there are cases where reducing rounding errors with quadruple-precision floating-point arithmetic causes the algorithm to converge more quickly when compared to double-precision arithmetic [3]. The Krylov subspace methods are an example of an application that requires extended precision arithmetic. This chapter will describe the implementation and performance evaluation of two Krylov subspace methods: the Conjugate Gradient (CG) and Bi-Conjugate Gradient Stabilized (BiCGStab) when using quadruple-precision arithmetic on an NVIDIA Tesla K20 GPU.

Krylov subspace methods are often used to solve large sparse linear systems $Ax = b$. Figures 4.1 and 4.2 [55] show the algorithms for the CG and BiCGStab methods, respectively. The CG method is applied when the coefficient matrix A is a symmetric positive definite matrix, and the BiCGStab method can be used when the coefficient matrix A is asymmetric. The convergence of the Krylov subspace methods depends on the spectral properties of the coefficient matrix. To improve the convergence, preconditioners which approximate the coefficient matrix are often used. In the algorithms shown in Figures 4.1 and 4.2 the preconditioning matrix M is used. By setting $M = I$, the algorithms become the same as their unpreconditioned counterparts. This chapter will first describe the implementation of

$\boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$
for : $k = 1, 2, ...$ **do**
  solve $\boldsymbol{M}\boldsymbol{z}_{k-1} = \boldsymbol{r}_{k-1}$
  $\rho_{k-1} = \langle \boldsymbol{r}_{k-1}, \boldsymbol{z}_{k-1} \rangle$
  if $k = 1$ **then**
    $\boldsymbol{p}_1 = \boldsymbol{z}_0$
  else
    $\beta_{k-1} = \rho_{k-1}/\rho_{k-2}$
    $\boldsymbol{p}_k = \boldsymbol{z}_{k-1} + \beta_{k-1}\boldsymbol{p}_{k-1}$
  end if
  $\boldsymbol{q}_k = \boldsymbol{A}\boldsymbol{p}_k$
  $\alpha_k = \rho_{k-1}/\langle \boldsymbol{p}_k, \boldsymbol{q}_k \rangle$
  $\boldsymbol{x}_k = \boldsymbol{x}_{k-1} + \alpha_k \boldsymbol{p}_k$
  $\boldsymbol{r}_k = \boldsymbol{r}_{k-1} - \alpha_k \boldsymbol{q}_k$
  if $||\boldsymbol{r}_k||_2/||\boldsymbol{r}_0||_2 \leq \epsilon$ **break**
end for

Figure 4.1: Preconditioned CG method

the unpreconditioned methods and then will describe the implementation of the preconditioned methods.

Using quadruple-precision arithmetic is also effective for improving convergence. Even if the use of quadruple-precision arithmetic increases the execution time of one iteration, the time until convergence may be reduced if the number of iterations is reduced to the point that it compensates for the increased execution time as shown in Figure 4.3. Although the mixed precision approach [56] generally utilizes lower precision arithmetic to accelerate computation, however I decided to use higher precision arithmetic to reduce the computation time by reducing the number of required iterations.

Chapter 2 showed that the performance of quadruple-precision dense matrix-vector multiplication is memory bound, and the execution time is only about twice that of the double-precision operation on GPUs. Since these methods mainly consist of Sparse Matrix-Vector multiplication (SpMV) and some vector-vector operations, they are generally regarded as being memory-intensive. Thus, the use of quadruple-precision arithmetic at most doubles the execution time of one iteration of the double-precision version on GPUs.

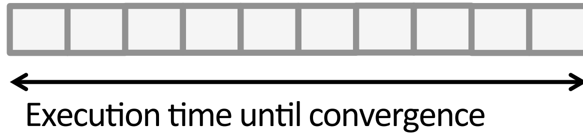
In this chapter, I will evaluate the performance of these methods using quadruple-precision arithmetic. Next, I will discuss how to use quadruple-precision arithmetic

$\boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$
$\tilde{\boldsymbol{r}} = \boldsymbol{r}_0$
for : $k = 1, 2, ...$ **do**
  $\rho_{k-1} = \langle \tilde{\boldsymbol{r}}, \boldsymbol{r}_{k-1} \rangle$
  if $\rho_{k-1} = 0$ **method fails**
  if $k = 1$ **then**
    $\boldsymbol{p}_k = \boldsymbol{r}_{k-1}$
  else
    $\beta_{k-1} = (\rho_{k-1}/\rho_{k-2})(\alpha_{k-1}/\omega_{k-1})$
    $\boldsymbol{p}_k = \boldsymbol{r}_{k-1} + \beta_{k-1}(\boldsymbol{p}_{k-1} - \omega_{k-1}\boldsymbol{v}_{k-1})$
  end if
  solve $\boldsymbol{p}_k = \boldsymbol{M}\hat{\boldsymbol{p}}$
  $\boldsymbol{v}_k = \boldsymbol{A}\hat{\boldsymbol{p}}$
  $\alpha_k = \rho_{k-1}/\langle \tilde{\boldsymbol{r}}, \boldsymbol{v}_k \rangle$
  $\boldsymbol{s} = \boldsymbol{r}_{k-1} - \alpha_k \boldsymbol{v}_k$
  if $||\boldsymbol{s}||_2/||\boldsymbol{r}_0||_2 \leq \epsilon$ **then**
    $\boldsymbol{x}_k = \boldsymbol{x}_{k-1} + \alpha_k \hat{\boldsymbol{p}}$
    break
  end if
  solve $\boldsymbol{s} = \boldsymbol{M}\hat{\boldsymbol{s}}$
  $\boldsymbol{t} = \boldsymbol{A}\hat{\boldsymbol{s}}$
  $\omega = \langle \boldsymbol{t}, \boldsymbol{s} \rangle / \langle \boldsymbol{t}, \boldsymbol{t} \rangle$
  $\boldsymbol{x}_k = \boldsymbol{x}_{k-1} + \alpha_k \hat{\boldsymbol{p}} + \omega_k \hat{\boldsymbol{s}}$
  $\boldsymbol{r}_k = \boldsymbol{s} - \omega_k \boldsymbol{t}$
  if $||\boldsymbol{r}_k||_2/||\boldsymbol{r}_0||_2 \leq \epsilon$ **break**
  if $\omega = 0$ **break**
end for

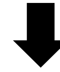
Figure 4.2: Preconditioned BiCGStab method

to potentially speed-up these methods. This chapter is organized as follows: Section 4.2 will introduce related work. Section 4.3 will show the implementation of the CG and BiCGStab methods using quadruple-precision floating-point arithmetic on GPUs. Section 4.4 will compare the performance of the quadruple-precision implementations with the double-precision versions and discuss the results. Finally, Section 4.5 will conclude the chapter.

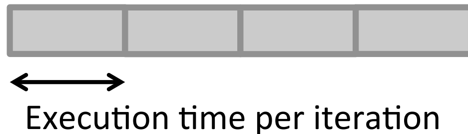
Low precision:



Time per iteration: 1
Number of iterations: 10
Total time: 1×10 = 10



High precision:



Time per iteration: 2
Number of iterations: 4
Total time: 2×4 = 8 **-> Fast**

Figure 4.3: Accelerating iterative methods using high precision arithmetic

4.2 Related Work

Hasegawa [3] compared the performance of an unpreconditioned BiCG method using quadruple-precision arithmetic to the preconditioned method using only double-precision arithmetic on various CPU architectures. He did not show cases where implementations using quadruple-precision arithmetic outperformed those using only double-precision arithmetic, but he expected that the use of quadruple-precision arithmetic may be an effective alternative to preconditioning which has low parallelism on parallel architectures.

Some linear algebra libraries for CPUs support quadruple-precision arithmetic for sparse iterative methods [57] [58] and there are some research into utilizing quadruple-precision arithmetic for Krylov subspace methods. Kotakemori et al. [59] described the implementation and performance of the BiCG methods using quadruple-precision arithmetic for lis [58], a sparse iterative solver library for CPUs. Their paper shows that on CPUs the use of quadruple-precision arithmetic requires approximately 2.99–4.56 times more execution time than that of the double-precision version per iteration. They also showed the DQ-SWITCH method which is a mixed precision method using both double- and quadruple-precision arithmetic on CPUs.

Furuichi et al. [60] implemented the Generalized Conjugate Residual (GCR) methods using quadruple-precision arithmetic on the NEC SX-9 supercomputer. They used double-precision operations only for the preconditioning operations, all other operations were implemented using quadruple-precision. As a result, they improved the convergence without significantly increasing the execution time. Saito et al. [61] also showed convergence improvement of the GCR methods on the Scilab

toolbox they developed for CPUs by using quadruple-precision arithmetic for certain parts of the algorithm.

Such research show that the use of quadruple-precision arithmetic improves the convergence and is useful for solving problems which cannot be solved using double-precision solvers. However, it can be hypothesized that the use of quadruple-precision arithmetic can also be used to accelerate double-precision solvers even when quadruple-precision arithmetic is not necessary. In addition, although Krylov subspace methods have been implemented on GPUs [62] [63], there is no research on the implementation and performance of methods using quadruple-precision arithmetic on GPUs.

4.3 Implementation

This section shows the implementation of the CG and BiCGStab methods using quadruple-precision arithmetic on GPUs. I implemented the unpreconditioned and preconditioned versions using both the double and quadruple-precision to compare the performance. The target environment is the NVIDIA Kepler architecture GPUs [19] of compute capability 3.5.

4.3.1 Overview of Quadruple-Precision Versions

I aimed to improve the convergence of the methods by using quadruple-precision arithmetic instead of double-precision arithmetic. Thus, for the implementations using quadruple-precision arithmetic, on sparse linear systems $Ax = b$, where the input, the coefficient matrix A and the vector b are given in the double-precision format. The vector x and all other floating-point data are stored in the quadruple-precision format. Quadruple-precision floating-point arithmetic is used instead of double-precision arithmetic everywhere except for the norm computation for checking convergence. For the preconditioned methods, quadruple-precision arithmetic is used everywhere except for the norm computation and the preconditioning process.

Quadruple-precision floating-point arithmetic is performed using DD arithmetic. The details are shown in Chapter 2 and the implementation of the DD arithmetic and subroutines using DD arithmetic are the same as in Chapter 2. One DD value is stored using a “double2” type value which is a vector type consisting of two double-precision values defined in CUDA. DD scalar value computations on the CPU side are performed using the QD library [12].

4.3.2 Implementation of CG and BiCGStab Methods on GPUs

Vector operations are performed on GPUs, and scalar operations are performed on CPUs. I implemented SpMV ($y = Ax$), DOT ($r = \langle x, y\rangle$), AXPY ($y = \alpha x + y$) and XPAY ($y = x + \alpha y$). The BiCGStab method additionally requires AXPYZ ($z = \alpha x + y$). I used the Compressed Row Storage (CRS) format for storing sparse matrices. Among the kernels implemented, SpMV is generally the most time-consuming operation. I used the SpMV algorithm and optimization techniques described in Chapter 3. This implementation however does not utilize the symmetric properties of matrices for storing symmetric matrices.

For some double-precision subroutines, vendor provided libraries such as CUBLAS [34] and cuSPARSE [22] are available. However, in order to measure the performance impact of the different precisions accurately, I implemented from scratch all vector operation subroutines that require both double and quadruple-precision versions. Doing so ensures that the algorithms for both the quadruple and double-precision versions are completely the same, including the number of threads used for GPU kernel functions. The norm computation for checking convergence is performed using double-precision for both versions, and thus was implemented using the DNRM2 subroutine of CUBLAS.

The preconditioned methods use an incomplete-LU preconditioner as known as ILU(0), one of the most popular preconditioners for Krylov subspace methods. The ILU(0) preconditioning performs incomplete-LU factorization which approximates $A \approx M = LU$, where L and U are the lower and upper triangular matrices, respectively. On the incomplete-LU factorization, the preconditioning matrix M keeps the non-zero pattern of the original coefficient matrix A. Therefore, $Ax = b$ can be solved as $(M^{-1}A)x = M^{-1}b$ using sparse triangular solvers. The double-precision subroutines for the ILU(0) preconditioning are provided by the cuSPARSE library. My implementation uses them for both double- and quadruple-precision versions and the preconditioning matrix is stored in the double-precision format. In other words, on the preconditioned methods, quadruple-precision arithmetic is used everywhere except for the norm computation and the preconditioning process. In the iterative portion, the cuSPARSE subroutine `cusparseDcsrsv_solve()` is executed two and four times on the CG and BiCGStab methods, respectively. The subroutine solves a sparse lower or upper triangular system with either forward or backward substitutions using double-precision.

Table 4.1: Evaluation environment

CPU	Intel Xeon E5-2609 (2.40GHz, 4 cores)
RAM	16 GB (DDR3)
GPU	NVIDIA Tesla K20
VRAM	5GB (GDDR5, ECC-enabled)
OS	CentOS 6.3
CUDA	CUDA 5.0
Compiler	gcc 4.4.6 (-O3), nvcc 5.0 (-O3 -arch sm_35)

4.4 Experimental Results

This section compares the performance of the quadruple-precision versions with the double-precision versions.

The input matrices are from the University of Florida Sparse Matrix Collection [21]. In order to make the matrices used suitable for execution on GPUs, I used relatively large real square matrices where the number of non-zero elements (Nonzeros) is greater than 1,000,000. There are a total of 154 symmetric matrices and 102 asymmetric matrices in the collection that meet these criteria.

Table 4.1 shows the evaluation environment. I evaluated the execution time of only the iterative portion of the CG and BiCGStab methods. The CG and BiCGStab methods are used for symmetric and asymmetric matrices, respectively. The following conditions are the same for both the double- and quadruple-precision versions: $b = (1, 1, ..., 1)^T$ and the initial vector of x is $x_0 = 0$. The stopping criterion is $\epsilon = 10^{-8}$ or $\epsilon = 10^{-12}$ on $||r||_2/||r_0||_2 \leq \epsilon$ and the maximum number of iterations is 30,000.

4.4.1 Unpreconditioned Methods

Table 4.2 shows the number of problems which were solved with the quadruple-precision versions with the stopping criterion ($||r||_2/||r_0||_2 \leq \epsilon$) and satisfying the true relative residual (TRR): $||b - Ax||_2/||b||_2 \leq \epsilon$. Among these, the cases were QP is preferable to DP are: "solved by both QP & DP and QP is faster" and "solved by QP but not solved by DP". A smaller ϵ occurs in cases satisfying the stopping criterion but not satisfying the accuracy criterion, TRR. Here I will focus on the cases where $\epsilon = 10^{-8}$.

Table 4.2: The number of problems which could be solved using quadruple-precision (QP) versions with stopping criterion: $||r||_2/||r_0||_2 \leq \epsilon$ and satisfying the true relative residual: $||b - Ax||_2/||b||_2 \leq \epsilon$

(a) CG for symmetric matrices (154 matrices)

	$\epsilon = 10^{-12}$	$\epsilon = 10^{-8}$
solved by both QP & DP	8	47
solved by both QP & DP and QP is faster	0	2
solved by QP but not solved by DP	15	12

(b) BiCGStab for asymmetric matrices (102 matrices)

	$\epsilon = 10^{-12}$	$\epsilon = 10^{-8}$
solved by both QP & DP	9	18
solved by both QP & DP and QP is faster	0	2
solved by QP but not solved by DP	7	7

Solved by both QP & DP and QP is faster

Using quadruple-precision arithmetic reduces the execution time for 4 (2 symmetric and 2 asymmetric) of the 65 matrices (47 symmetric and 18 asymmetric) which were solvable by both double- and quadruple-precision versions where $\epsilon = 10^{-8}$. Table 4.3 shows these 4 cases. (a) shows the matrix properties, (b) and (c) show the results for the double- and quadruple-precision versions respectively, and (d) shows the ratios of the number of iterations, time per iteration, and the total execution time for quadruple-precision versions compared to the double-precision versions. "# iter" means the number of iterations until convergence. "1 iter time" represents the execution time of one iteration. "Total time" means the total execution time until convergence. For example, "rajat31", the use of quadruple-precision arithmetic requires approximately 1.84 times more execution time than that of the double-precision version for one iteration. However, the number of iterations decreases to approximately 38 % of the double-precision version by using quadruple-precision arithmetic. As a result, the use of quadruple-precision arithmetic reduces the total time until convergence to approximately 69 % of the double-precision version.

Solved by QP but not solved by DP

19 matrices can be solved using quadruple-precision, but not with double-precision when $\epsilon = 10^{-8}$. 3 of these 19 matrices do not converge within the maximum itera-

Table 4.3: Cases which can be solved using both quadruple-precision (QP) and double-precision (DP) versions with stopping criterion: $||r||_2/||r_0||_2 \leq 10^{-8}$ and satisfying the true relative residual (TRR): $||b - Ax||_2/||b||_2 \leq 10^{-8}$ but QP versions are faster

(a) Matrix properties

Matrix	Rows	Nonzeros	Structure	Application
rajat31	4690002	20316253	asym	circuit simulation problem
venkat01	62424	1717792	asym	computational fluid dynamics problem sequence
crankseg_2	63838	14148858	sym	structural problem
c-73b	169422	1279274	sym	subsequent optimization problem

(b) Double-precision (DP)

Matrix	# iter	1 iter time [sec]	Total time [sec]	TRR
rajat31	11618	0.0182	211	9.94E-09
venkat01	20761	1.33E-03	27.7	9.61E-09
crankseg_2	4915	1.59E-03	7.83	9.44E-09
c-73b	19857	3.04E-03	60.4	9.95E-09

(c) Quadruple-precision (QP)

Matrix	# iter	1 iter time [sec]	Total time [sec]	TRR
rajat31	4380	0.0334	146	8.68E-09
venkat01	15858	1.72E-03	27.2	9.85E-09
crankseg_2	3961	1.95E-03	7.71	9.81E-09
c-73b	8916	6.70E-03	59.7	6.61E-09

(d) Ratio of QP/DP

Matrix	# iter	1 iter time	Total time
rajat31	0.38	1.84	0.69
venkat01	0.76	1.29	0.98
crankseg_2	0.81	1.22	0.98
c-73b	0.45	2.20	0.99

tions, 30,000, when using double-precision. Table 4.4 shows these 3 matrices. These cases could potentially be solved using double-precision by increasing the maximum number of iterations, however, using quadruple-precision arithmetic may be still faster. For instance, in “gyro” the quadruple-precision version is already faster after

Table 4.4: Cases which can be solved using quadruple-precision (QP) versions with stopping criterion: $||r||_2/||r_0||_2 \leq 10^{-8}$ and satisfying the true relative residual (TRR): $||b - Ax||_2/||b||_2 \leq 10^{-8}$ but which do not converge when using double-precision (DP) versions with the stopping criterion within the maximum number of iterations of 30,000

(a) Matrix properties

Matrix	Rows	Nonzeros	Structure	Application
gyro	17361	1021159	sym	model reduction problem
human_gene2	14340	18068388	sym	undirected weighted graph
CoupCons3D	416800	17277420	asym	structural problem

(b) Double-precision (DP)

Matrix	# iter	1 iter time [sec]	Total time [sec]	TRR
gyro	30000	3.25E-04	9.75	2.54E-08
human_gene2	30000	1.95E-03	58.6	4.96E-01
CoupCons3D	30000	6.82E-03	205	3.32E-06

(c) Quadruple-precision (QP)

Matrix	# iter	1 iter time [sec]	Total time [sec]	TRR
gyro	17381	4.44E-03	7.72	9.96E-09
human_gene2	29804	2.66E-03	79.1	9.88E-09
CoupCons3D	26931	0.0125	337	9.44E-09

(d) Ratio of QP/DP

Matrix	# iter	1 iter time	Total time
gyro	0.58	1.37	0.79
human_gene2	0.99	1.36	1.35
CoupCons3D	0.90	1.83	1.65

30,000 iterations.

On the other hand, 12 of the 19 matrices also converged when using double-precision with the stopping criterion ($||r||_2/||r_0||_2 \leq 10^{-8}$) but did not satisfy the TRR: $||b - Ax||_2/||b||_2 \leq 10^{-8}$. Table 4.5 shows 4 of these 12 cases, which include the 2 best and the 2 worst QP/DP total execution time ratio. In these cases, using quadruple-precision arithmetic improves the accuracy of the TRR with a small increase in the execution time: quadruple-precision versions require approximately 1.02 to 1.70 times more execution time than that of the double-precision versions

Table 4.5: Cases which can be solved using quadruple-precision (QP) versions and converged by double-precision (DP) versions with stopping criterion: $||r||_2/||r_0||_2 \leq 10^{-8}$ but not satisfying the true relative residual (TRR): $||b - Ax||_2/||b||_2 \leq 10^{-8}$ when using DP versions

(a) Matrix properties

Matrix	Rows	Nonzeros	Structure	Application
bone010	986703	47851783	sym	model reduction problem
nd6k	18000	6897316	sym	2D/3D problem
...	...	...	...	...
thermal2	1228045	8580313	sym	thermal problem
apache2	715176	4817870	sym	structural problem

(b) Double-precision (DP)

Matrix	# iter	1 iter time [sec]	Total time [sec]	TRR
bone010	16684	9.03E-03	151	7.07E-08
nd6k	8053	8.54E-04	6.87	2.19E-08
...	...	...	...	...
thermal2	5509	2.79E-03	15.4	1.27E-08
apache2	4787	1.61E-03	7.71	1.07E-08

(c) Quadruple-precision (QP)

Matrix	# iter	1 iter time [sec]	Total time [sec]	TRR
bone010	10805	0.0143	154	9.79E-09
nd6k	7775	9.47E-04	7.37	9.16E-09
...	...	...	...	...
thermal2	5494	4.71E-03	25.9	9.96E-09
apache2	4787	2.73E-03	13.1	9.94E-09

(d) Ratio of QP/DP

Matrix	# iter	1 iter time	Total time
bone010	0.65	1.58	1.02
nd6k	0.97	1.11	1.07
...	...	...	...
thermal2	1.00	1.69	1.68
apache2	1.00	1.70	1.70

Table 4.6: The 2 largest and 2 smallest relative execution times per iteration (QP/DP ratio)

(a) Matrix properties and execution time of one iteration (QP/DP ratio)

Matrix	Properties			1 iter time
	Rows	Nonzeros	Structure	[QP/DP]
nd6k	18000	6897316	sym	1.11
crankseg_2	63838	14148858	sym	1.22
rajat31	4690002	20316253	asym	1.84
c-73b	169422	1279274	sym	2.20

(b) Performance of SpMV, percent of a single iteration spent calculating SpMV, and relative execution time of SpMV (QP/DP ratio)

Matrix	DP		QP		Exec time
	GFlops	[%]	GDDFlops	[%]	[QP/DP]
nd6k	20.99	77.0	18.90	77.0	1.11
crankseg_2	20.73	85.7	17.24	84.3	1.20
rajat31	8.04	55.7	4.87	50.0	1.65
c-73b	0.94	89.5	0.42	92.0	2.26

for one iteration.

Cost of quadruple-precision arithmetic

For the unpreconditioned methods, in the 11 cases shown in Tables 4.3 – 4.5, the execution time of one iteration of the methods using quadruple-precision arithmetic is approximately 1.11 – 2.20 times more than the double-precision versions. Table 4.6 shows the performance of SpMV, percent of a single iteration spent calculating SpMV, and the relative execution time of SpMV (QP/DP ratio) for cases with the 2 largest and 2 smallest relative execution times per iteration (QP/DP).

For "nd6k" and "crankseg_2", the relative execution time (QP/DP ratio) of one iteration is close to 1.0 and SpMV occupies more than 77 % of the execution time per iteration of both the double- and quadruple-precision versions. Thus, the relative execution time (QP/DP ratio) of one iteration strongly depends on the QP/DP ratio of SpMV. I theorize that the performance of the SpMV is memory bound on quadruple-precision versions on the GPU as evidenced by the Bytes/Flop and Bytes/DDFlop as well, as shown in Chapter 2. When the number of nonzero elements is NNZ

and the number of rows is N, the SpMV is approximately $(8 + 4) \times$NNZ [Bytes] / (2$\times$NNZ) [DDFlop] = 6.0 [Bytes/DDFlop], where NNZ $\gg$ N. Note that for the quadruple-precision versions, the input matrix on the SpMV using DD arithmetic is given in the double-precision format. On the other hand, the Tesla K20 GPU has 64 double-precision FPUs which can perform double-precision multiply-add operations in one cycle with a theoretical peak bandwidth is 208 GB/s. DD arithmetic requires 20 times as many cycles (i.e. execution time) as double-precision arithmetic as shown in Chapter 2. Therefore, the theoretical peak performance of the DD arithmetic is 706[MHz] $\times$ 13[SMX] $\times$ 64[CUDACore(DP)] $\times$ (2[DDFlop]/20[cycle]) $\approx=$ 58.7[GDDFlops] and the GPU has a Bytes/DDFlop ratio of: 208/58.7 $\approx$ 3.5. Thus, it can be predicted that the performance of SpMV using DD arithmetic is memory bound on this GPU. In addition, SpMV has precision-independent costs such as indirect memory accesses using index arrays. As a result, we can conjecture that the execution times of SpMV using the double-precision and that of DD arithmetic are close to the same because the input matrix is given in the double-precision format.

On the other hand for "rajat31" and "c-73b", the relative execution time (QP/DP ratio) of one iteration and SpMV is close to 2.0 and the Flops and DDFlops values are relatively small compared to that of the top 2 cases. One of the reasons for this might be that the memory access is not well coalesced and therefore the performance may not be limited by memory-bandwidth, but instead by memory access latency. Since quadruple-precision operations may access memory almost twice as often as double-precision, this limitation can significantly impact the performance of the quadruple-precision version. In addition for "rajat31", the percentage of one iteration spent inside SpMV relatively small compared to the other cases, and all the other vector operations except SpMV perform DD arithmetic in the quadruple-precision format. Therefore, we can conjecture that in such cases the execution time of one iteration of the quadruple-precision versions is close to twice that of the double-precision versions.

4.4.2 Cases with Preconditioning

Table 4.7 shows the number of problems which can be solved using preconditioned quadruple-precision versions with stopping criterion ($||r||_2/||r_0||_2 \leq \epsilon$) and satisfying the TRR: $||b - Ax||_2/||b||_2 \leq \epsilon$. When ϵ = 1E-08, using quadruple-precision arithmetic reduces the execution time for 13 (10 symmetric and 3 asymmetric) of the 71 matrices (48 symmetric and 23 asymmetric) which were solvable by both double- and quadruple-precision versions.

Table 4.7: The number of problems which can be solved using preconditioned quadruple-precision (QP) versions with stopping criterion: $||r||_2/||r_0||_2 \le \epsilon$ and satisfying the true relative residual (TRR): $||b - Ax||_2/||b||_2 \le \epsilon$

(a) preconditioned CG for symmetric matrices (154 matrices)

	$\epsilon = 10^{-12}$	$\epsilon = 10^{-8}$
solved by both QP & DP	10	48
solved by both QP & DP and QP is faster	2	10
solved by QP but not solved by DP	11	14

(b) preconditioned BiCGStab for asymmetric matrices (102 matrices)

	$\epsilon = 10^{-12}$	$\epsilon = 10^{-8}$
solved by both QP & DP	15	23
solved by both QP & DP and QP is faster	2	3
solved by QP but not solved by DP	6	1

Solved by both QP & DP and QP is faster

Table 4.8 shows 5 of the 13 cases which can be solved using both double- and quadruple-precision versions with the stopping criterion ($||r||_2/||r_0||_2 \le 10^{-8}$) and satisfying the TRR: $||b - Ax||_2/||b||_2 \le 10^{-8}$ but quadruple-precision versions are faster. On the table, "Precond [%]" means the percentage of the execution time of one iteration spent preconditioning. These 5 cases have the highest speedup when using quadruple-precision arithmetic.

On the other hand, 15 matrices satisfy the stopping criterion but do not satisfy the criterion of accuracy, TRR, when using the double-precision versions. Table 4.9 shows 4 of these 15 cases, which include the highest and the lowest 2 cases of the value of the QP/DP ratio of the total time. The cases shown in the table are for the CG method as the cases with the best and worst speedup were all symmetric. All the asymmetric matrices, which use the BiCGStab method, fell between these two extremes.

Cost of quadruple-precision arithmetic

For the preconditioned methods, the execution time of one iteration for the preconditioning process takes an extremely long time on the GPU as "Precond [%]" shown in Tables 4.8 and 4.9. The preconditioning process is performed using double-precision for both the double- and quadruple-precision versions. Therefore, the execution

Table 4.8: Cases with preconditioning which can be solved using both quadruple-precision (QP) and double-precision (DP) versions with stopping criterion: $||r||_2/||r_0||_2 \leq 10^{-8}$ and satisfying the true relative residual (TRR): $||b - Ax||_2/||b||_2 \leq 10^{-8}$ but QP versions are faster

(a) Matrix properties

Matrix	Rows	Nonzeros	Structure	Application
CO	221119	7666057	sym	theoretical/quantum chemistry problem
GaAsH6	61349	3381809	sym	theoretical/quantum chemistry problem
Ga3As3H12	61349	5970947	sym	theoretical/quantum chemistry problem
raefsky3	21200	1488768	asym	computational fluid dynamics problem
Lin	256000	1766400	sym	structural problem

(b) Double-precision (DP)

Matrix	# iter	1 iter time [sec]	Precond [%]	Total time [sec]	TRR
CO	461	1.65	99.9	762	8.11E-09
GaAsH6	364	0.494	99.9	180	8.78E-09
Ga3As3H12	730	0.578	99.8	422	8.63E-09
raefsky3	146	0.0384	98.0	5.61	1.33E-09
Lin	488	6.51E-03	89.3	3.18	9.81E-09

(c) Quadruple-precision (QP)

Matrix	# iter	1 iter time [sec]	Precond [%]	Total time [sec]	TRR
CO	388	1.66	99.8	643	7.97E-09
GaAsH6	318	0.495	99.8	157	7.44E-09
Ga3As3H12	646	0.578	99.8	373	8.81E-09
raefsky3	135	0.0386	97.3	5.22	3.31E-09
Lin	425	6.97E-03	83.7	2.96	9.59E-09

(d) Ratio of QP/DP

Matrix	# iter	1 iter time	Total time
CO	0.84	1.00	0.84
GaAsH6	0.87	1.00	0.88
Ga3As3H12	0.88	1.00	0.89
raefsky3	0.92	1.01	0.93
Lin	0.87	1.07	0.93

Table 4.9: Cases with preconditioning which can be solved using quadruple-precision (QP) versions and converged by double-precision (DP) versions with stopping criterion: $||r||_2/||r_0||_2 \leq 10^{-8}$ but not satisfying the true relative residual (TRR): $||b - Ax||_2/||b||_2 \leq 10^{-8}$ when using DP versions

(a) Matrix properties

Matrix	Rows	Nonzeros	Structure	Application
msc10848	10848	1229776	sym	structural problem
bmwcra_1	148770	10641602	sym	structural problem
...	...	...	...	...
thermal2	1228045	8580313	sym	thermal problem
c-73	169422	1279274	sym	optimization problem sequence

(b) Double-precision (DP)

Matrix	# iter	1 iter time [sec]	Precond [%]	Total time [sec]	TRR
msc10848	3994	0.0121	97.4	48.3	1.18E-08
bmwcra_1	1379	0.0199	92.5	27.4	2.09E-08
...	...	...	...	...	...
thermal2	2287	0.0348	91.5	74.7	1.12E-08
c-73	9452	6.23E-03	51.2	58.8	1.02E-07

(c) Quadruple-precision (QP)

Matrix	# iter	1 iter time [sec]	Precond [%]	Total time [sec]	TRR
msc10848	3508	0.0122	96.9	42.8	9.54E-09
bmwcra_1	1162	0.0209	88.6	24.3	9.64E-09
...	...	...	...	...	...
thermal2	2289	0.0348	86.5	79.6	9.90E-09
c-73	9143	9.87E-03	32.1	90.3	9.11E-09

(d) Ratio of QP/DP

Matrix	# iter	1 iter time	Total time
msc10848	0.88	1.01	0.89
bmwcra_1	0.84	1.05	0.89
...	...	...	...
thermal2	1.00	1.06	1.07
c-73	0.97	1.59	1.53

time of one iteration is almost the same for both versions. In such cases, using quadruple-precision arithmetic to decrease the number of iterations by even a small amount leads to a speedup.

4.5 Conclusion

In this chapter, I have shown implementations of the CG and BiCGStab methods using quadruple-precision floating-point arithmetic on GPUs and compared the performance to standard double-precision versions on a Tesla K20 Kepler architecture GPU. My goal was to improve the convergence of the methods to solve sparse linear systems $Ax = b$. Quadruple-precision arithmetic was used everywhere in place of double-precision arithmetic everywhere except for the norm computations and the preconditioning process. For the implementations using quadruple-precision arithmetic, the input data: the coefficient matrix A and the vector b are given in the double-precision format.

I have shown the relative time cost of the use of quadruple-precision arithmetic in the CG and BiCGStab methods. On unpreconditioned methods, the use of quadruple-precision arithmetic required approximately 1.11–2.20 times more execution time than that of the double-precision versions for one iteration. On the other hand, the quadruple-precision iteration time for methods with double-precision incomplete LU preconditioning is only slightly more than that of double-precision.

Quadruple-precision versions can solve the problem faster than the double-precision versions in cases where the use of quadruple-precision arithmetic reduces the number of required iterations enough to compensate for the increased time required for one iteration. I showed cases where the quadruple-precision version can reach a solution faster than the double-precision version even when the double-precision version converged within the maximum number of iterations, although such cases were rare. I have shown that the use of quadruple-precision arithmetic is not costly in the CG and BiCGStab methods on the GPU. Therefore, the use of quadruple-precision arithmetic, along with the use of preconditioning may be an effective method for accelerating the computation of Krylov subspace methods on GPUs. In general, the preconditioning process is difficult to parallelize, and it is not effective on massively parallel architectures such as GPU clusters. Thus, the use of quadruple-precision arithmetic may be an effective alternative to preconditioning on such environments.

Obviously the effectiveness of the use of quadruple-precision arithmetic needs to be further investigated. The performance of Krylov subspace methods is affected by many factors such as the solver algorithm, the preconditioning methods, and

the sparse matrix storage format. The optimal choice for those factors depends on the problem characteristics and the computation environment including the architecture and parallelism. In addition, the matrices of the University of Florida Sparse Matrix Collection are not necessarily distributed evenly over such properties as size, nonzero pattern, application, and mathematical properties. Therefore, the results in this chapter do not necessarily show a statistical significance of the use of quadruple-precision arithmetic. Further research is necessary to determine under which conditions the number of iterations decreases when using quadruple-precision arithmetic.

Chapter 5

Conclusion

I have shown the implementation and the performance of linear algebraic operations using extended precision floating-point arithmetic on GPUs. This chapter summarizes this thesis and describes areas for future research.

5.1 Summary

Floating-point operations have rounding errors and these errors may become a critical issue for some applications, therefore, the usual precision of hardware floating-point operations is insufficient in some cases and extension of accuracy and precision is required. I mainly focused on triple- and quadruple-precision operations by software and has shown the implementation and evaluation of linear algebraic operations using that on GPUs.

In Chapter 2, I have shown the implementation and performance of triple- and quadruple-precision BLAS subroutines on the NVIDIA Tesla M2050 Fermi architecture GPU. For quadruple-precision subroutines, DD arithmetic was used. I have implemented three BLAS subroutines: AXPY, GEMV and GEMM as the representatives of vector-vector, matrix-vector and matrix-matrix operations, respectively. For the BLAS subroutines on the GPU, DD arithmetic requires 20 times the execution time of double-precision arithmetic in theory. However, the performances of double- and quadruple-precision AXPY and GEMV are memory-bound on the GPU; therefore, the execution time of the quadruple-precision subroutines is approximately twice that of the double-precision subroutines. On GEMM, the performance is computationally bound and the execution time for the quadruple-precision subroutine is close to 14 times that of the double-precision subroutine because of the low execution efficiency of the double-precision subroutine. The use of DD arith-

metic increases the density of arithmetic instructions per memory access, so as a result, higher execution efficiency was achieved. On the other hand, I have proposed two methods to store triple-precision floating-point values and a method to compute triple-precision floating-point arithmetic operations that are based on DD arithmetic. Although I was not able to realize the faster triple-precision arithmetic operations rather than DD arithmetic, the proposed triple-precision floating-point formats are effective for linear algebra operations when the performance of which is memory-bound. In comparison with a quadruple-precision format, triple-precision formats can save the limited memory space on GPUs and reduce data translation time to 3/4 times. On AXPY and GEMV, the performance of the triple-precision subroutines is memory-bound, and therefore, the execution times are close to 3/4 times that of the quadruple-precision subroutines. Therefore, triple-precision operations are effective for memory-bound operations in cases where double-precision is insufficient and quadruple-precision is not required, but triple-precision is sufficient.

Furthermore, I have shown the application of using quadruple-precision arithmetic for sparse linear algebra on GPUs. To implement fast sparse matrix operations on GPUs, in Chapter 3, I have presented optimization techniques of sparse matrix-vector multiplication (SpMV) for the CRS format on NVIDIA Kepler architecture GPUs. The proposal implementation is based on the existing method proposed for Fermi architecture, an earlier generation of Kepler architecture, and takes advantage of three new features of the latter: a 48KB read-only data cache, shuffle instructions and expansion of the MaxGridDimX. On the Tesla K20 Kepler architecture GPU on double-precision operations, the proposal implementation achieved speedups over the implementation for the earlier generation of Kepler architecture GPUs. Furthermore, the thesis has shown that the implementation outperforms the SpMV routine for the CRS format of the cuSPARSE 5.0 for 174 of the 200 matrices.

In Chapter 4, I showed the application of quadruple-precision floating-point arithmetic on GPUs for sparse iterative methods. The convergence of the Krylov subspace methods, which are iterative methods for solving linear systems, is significantly affected by rounding errors, and there are cases wherein reduction in rounding errors with quadruple-precision arithmetic causes the algorithm to converge more quickly. I implemented the CG and BiCGStab methods, which are Krylov subspace methods, using quadruple-precision floating-point arithmetic and compared the performance to the standard double-precision implementations on the Tesla K20 GPU. On unpreconditioned methods, the use of quadruple-precision arithmetic required approximately 1.11–2.20 times more execution time than that of the double-precision versions for one iteration. On the other hand, the quadruple-precision iteration time for methods with double-precision incomplete LU preconditioning is

only slightly more than that of double-precision. I have shown cases in which a quadruple-precision versions can solve the problem faster than the double-precision versions in cases where the use of quadruple-precision arithmetic reduces the number of required iterations enough to compensate for the increased time required for one iteration.

Throughout this research, it is postulated that the use of quadruple-precision arithmetic by software using DD arithmetic is not costly for some memory-intensive operations on GPUs. This is because recent GPUs have tremendous floating-point performance compared to memory performance: the Bytes/Flop ratio of the GPU is relatively low compared to that of the quadruple-precision subroutines. Nowadays, floating-point performance of processors has rapidly increased compared to the performance increase of data access time of, for example, a memory, a bus, and a network. In other words, the Bytes/Flop ratio of processors and systems is becoming smaller, and it is predicted that memory bandwidth bottlenecks will be tight in exascale computing [43]. In fact, particularly on GPU-equipped supercomputer systems, the PCIe and network bandwidth are insufficient compared to the memory bandwidth and its floating-point performance. In such environments, many operations are becoming memory-bound rather than compute-bound. As a result, extended precision floating-point operations by software may be getting cost effective. In some cases, triple-precision operations may be becoming the cost-effective alternative for quadruple-precision operations. Triple- and quadruple-precision can also be utilized in a mixed precision approach.

Furthermore, I showed the application of quadruple-precision floating-point arithmetic to Krylov subspace methods on GPUs in Chapter 4 as for a case study. To improve the convergence of Krylov subspace methods, preconditioning is often used and using quadruple-precision arithmetic is also effective. I showed cases where the implementation using quadruple-precision arithmetic can solve the problem faster than the double-precision versions. In general, the preconditioning process is difficult to parallelize, and it is not effective on massively parallel architectures such as GPU clusters. Therefore, the use of quadruple-precision arithmetic, along with the use of preconditioning may be an effective method for accelerating the computation of Krylov subspace methods on GPUs. As well as the cases in the CG and BiCGStab methods on the GPU, on massively parallel architectures, existing algorithms are not always effective and the use of extended precision arithmetic may become an effective alternative for the existing method. Utilizing extended precision arithmetic operations on other actual applications can be expected.

5.2 Future Work

Here describes areas for future research.

I showed cases on a single GPU, but recent HPC applications are performed on multiple GPUs or multiple-node systems. Therefore, performance evaluation on such practical environments is required. There is a difference in the performance model between a single GPU and multiple-GPU-equipped systems, such as GPU clusters. One of the biggest differences is the existence of data translation via PCIe bus. Therefore, application performance tends to be limited by the bandwidth of the bus. In addition, although, I have shown only the performance on basic linear algebra operations, the performance on actual applications that require extended precision operations should be shown in the future. Furthermore, performance, implementation and effectiveness of linear algebra operations using extended precision arithmetic should also be discussed on other processors. Quadruple-precision arithmetic is implemented also on Field-Programmable Gate Array (FPGA) devices [24] [64]. For example, the performance evaluation on Intel Many Integrated Core Architecture (MIC) is also desired.

Although I showed the application of quadruple-precision arithmetic on GPUs for iterative methods for solving sparse linear systems, obviously the effectiveness of the use of quadruple-precision arithmetic needs to be further investigated. The performance of sparse iterative solvers is affected by many factors such as the solver algorithm, the preconditioning methods, and the sparse matrix storage format and the optimal choice for those factors depends on the problem characteristics and the computation environment including the architecture and parallelism. Further research is necessary to determine under which conditions the number of iterations decreases when using quadruple-precision arithmetic. The application of triple-precision operations is also desired.

In this research, I used the same algorithms of DD arithmetic in the QD library [12], but some other algorithms for DD arithmetic are also proposed. For example, Nagai et al. noted two types of DD multiplication algorithms in their papers [65] [66]. They investigated HITACHI Optimizing C Compiler's quadruple-precision multiplication software processing on a SR11000/J2 and then slightly modified the HITACHI's algorithm to decrease the number of instructions for multiply-add operations by utilizing FMA operations [67]. Using the algorithm, the relative cost of quadruple-precision operations on computationally bound operations will decrease. I proposed the triple-precision formats for memory-bound operations by focusing on the effects of reducing the data size compared to the quadruple-precision format. Similarly, half-precision may be effective compared to single-precision in some cases.

In addition, a double-precision floating-point value can be stored in a 32-bit integer value with the same exponent bits of the double-precision value by using the same method as the D+I-type format for example. The performance of memory-bound operations may be improved by focusing on the precision of the data and the storage format.

For widely practical use of extended precision on linear algebra operations on GPUs, well-optimized implementation of linear algebra libraries such as BLAS, for example GotoBLAS [68] for single- and double-precision operations, are desired. However, most current linear algebra libraries support only single- and double-precision floating-point arithmetic. There are few linear algebra libraries that support extended precision floating-point arithmetic. MPACK [40] is the only LAPACK implementation that supports multiple-precision arithmetic including DD arithmetic, but MBLAS is implemented as a reference implementation and is not well optimized. Autotuning technology is one of the most important for developing linear algebra libraries that support multiple precisions. Automatically Tuned Linear Algebra Software (ATLAS) [69] is one of the most well-known implementations of BLAS using autotuning technology. It is unrealistic to implement and optimize various subroutines by hand, especially for various precisions: for example, single-, double-, triple- and quadruple- for real and complex operations. For instance, Kurzak et al. showed the performance improvement in the GEMM subroutines on GPUs by utilizing a heuristic autotuning [70]. For future work, all set of BLAS subroutines should be implemented by using such auto-tuning technique.

Acknowledgements

First of all, I would like to express the deepest appreciation to my principal supervisor, Processor Daisuke Takahashi, who has provided me with generous support, advice and comments. Without his guidance and persistent support this work would not have been possible. I am deeply indebted to him also for providing many opportunities of working in the excellent research environment.

I would also like to express my gratitude to Professor Hidehiko Hasegawa, who was my previous supervisor during my undergraduate in the School of Library and Information Science, University of Tsukuba, for directing me toward attractive research areas, such as high performance computing. I received helpful advice and comments also during the course of this study.

I would like to express my special thanks to my thesis committee members, Professor Daisuke Takahashi, Professor Hidehiko Hasegawa, Professor Koichi Wada, Professor Taisuke Boku and Professor Tetsuya Sakurai for reading the thesis and giving me fruitful advice and comments.

I sincerely thank all members of the High Performance Computing Systems Laboratory (HPCS Lab.) in University of Tsukuba for many discussions and encouragement. Especially I thank the Algorithm team and the GPU team for discussing a variety of interesting topics such as GPU computing and helping me. I am indebted to Joel Tucci for English proofreading on English works including this thesis.

A part of this work was supported by JST, CREST "An evolutionary approach to construction of a software development environment for massively-parallel heterogeneous systems" and JSPS Grant-in-Aid for JSPS Fellows No. 251290.

Finally, I would also like to express my gratitude to my family for their moral support and warm encouragement.

Bibliography

[1] Dongarra, J., Bunch, J., Moler, C. and Stewart, G.: *LINPACK Users' Guide*, SIAM (1979).

[2] Meuer, H., Strohmaier, E., Dongarra, J. and Simon, H.: TOP500 Supercomputing Sites, http://www.top500.org/.

[3] Hasegawa, H.: Utilizing the quadruple-precision floating-point arithmetic operation for the Krylov Subspace Methods, *Proc. 8th SIAM Conference on Applied Linear Algebra (LA03)* (2003).

[4] Bailey, D., Barrio, R. and Borwein, J.: High-precision computation: Mathematical physics and dynamics, *Applied Mathematics and Computation*, Vol. 218, No. 20, pp. 10106–10121 (2012).

[5] IEEE Computer Society: IEEE Standard for Floating-Point Arithmetic, *IEEE Std 754-2008*, pp. 1–58 (2008).

[6] Intel Corporation: Intel Fortran Compilers, http://software.intel.com/en-us/fortran-compilers.

[7] GNU Project: GCC, the GNU Compiler Collection, http://gcc.gnu.org/fortran/.

[8] Granlund, T. and the GMP development team: GMP: GNU Multiple Precision Arithmetic Library, http://gmplib.org/.

[9] Dekker, T.: A Floating-Point Technique for Extending the Available Precision, *Numerische Mathematik*, Vol. 18, pp. 224–242 (1971).

[10] Bailey, D.: DDFUN90 (Fortran-90 double-double package), http://crd.lbl.gov/~dhbailey/mpdist/.

[11] Briggs, K.: doubledouble, http://keithbriggs.info/doubledouble.html (1998).

[12] Bailey, D.: QD (C++/Fortran-90 double-double and quad-double package), http://crd.lbl.gov/~dhbailey/mpdist/.

[13] Advanced Micro Devices, Inc.: Accelerated Parallel Processing (APP) SDK, http://developer.amd.com/tools-and-sdks/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/.

[14] NVIDIA Corporation: CUDA Zone, https://developer.nvidia.com/category/zone/cuda-zone.

[15] Khronos Group.: OpenCL - The open standard for parallel programming of heterogeneous systems, http://www.khronos.org/opencl/.

[16] Cray Inc.: Cray XK7 online product brochure, http://www.cray.com/Assets/PDF/products/xk/CrayXK7Brochure.pdf.

[17] NVIDIA Corporation: CUDA C PROGRAMMING GUIDE, http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf (2013).

[18] NVIDIA Corporation: Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Fermi, http://www.nvidia.com/object/IO_86775.html (2009).

[19] NVIDIA Corporation: Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110, http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf (2012).

[20] Lawson, C., Hanson, R., Kincaid, D. and Krogh, F.: Basic Linear Algebra Subprograms for Fortran Usage, *ACM Trans. Math. Softw.*, Vol. 5, No. 3, pp. 308–323 (1979).

[21] Davis, T. and Hu, Y.: The University of Florida Sparse Matrix Collection, http://www.cise.ufl.edu/research/sparse/matrices/.

[22] NVIDIA Corporation: cuSPARSE Library (included in CUDA Toolkit), https://developer.nvidia.com/cusparse.

[23] Lu, M., He, B. and Luo, Q.: Supporting Extended Precision on Graphics Processors, *Proc. 6th International Workshop on Data Management on New Hardware (DaMoN 2010)*, pp. 19–26 (2010).

[24] Dou, Y., Lei, Y., Wu, G., Guo, S., Zhou, J. and Shen, L.: FPGA Accelerating Double/Quad-Double High Precision Floating-Point Applications for ExaScale Computing, *Proc. 24th ACM International Conference on Supercomputing (ICS '10)*, pp. 325–336.

[25] Graça, G. D. and Defour, D.: Implementation of float-float operators on graphics hardware, *Proc. 7th Conference on Real Numbers and Computers (RNC7)*, pp. 23–32.

[26] Thall, A.: Extended-Precision Floating-Point Numbers for GPU Computation, *ACM SIGGRAPH 2006 Research Posters* (2006).

[27] Knuth, D.: *The Art of Computer Programming Vol.2 Seminumerical Algorithms*, Addison-Wesley, 3rd edition (1998).

[28] Karp, A. and Markstein, P.: High-Precision Division and Square Root, *ACM Trans. Math. Softw.*, Vol. 23, pp. 561–589 (1997).

[29] Nievergelt, Y.: Scalar Fused Multiply-Add Instructions Produce Floating-Point Matrix Arithmetic Provably Accurate to the Penultimate Digit, *ACM Trans. Math. Softw.*, Vol. 29, pp. 27–48 (2003).

[30] Ogita, T., Rump, S. and Oishi, S.: Accurate Sum and Dot Product, *SIAM Journal on Scientific Computing*, Vol. 26, pp. 1955–1988 (2005).

[31] Ikebe, Y.: Note on Triple-Precision Floating-Point Arithmetic with 132-bit Numbers, *Communications of the ACM*, Vol. 8, pp. 175–177 (1965).

[32] Ozawa, K.: A Control Method of Errors in Long-Term Integration, *IPSJ SIG Techinical Report*, Vol. 2012-HPC-134, No. 4, pp. 1–8 (2012). (in Japanese).

[33] Tan, G., Li, L., Triechle, S., Phillips, E., Bao, Y. and Sun, N.: Fast Implementation of DGEMM on Fermi GPU, *Proc. 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, No. 35, pp. 1–11 (2011).

[34] NVIDIA Corporation: CUBLAS Library (included in CUDA Toolkit), https://developer.nvidia.com/cublas.

[35] Xianyi, Z., Qian, W. and Yunquan, Z.: Model-driven Level 3 BLAS Performance Optimization on Loongson 3A Processor, *Proc. IEEE 18th International Conference on Parallel and Distributed Systems (ICPADS 2012)*, pp. 684–691 (2012).

[36] NVIDIA Corporation: CUDA COMPILER DRIVER NVCC, http://docs.nvidia.com/cuda/pdf/CUDA_Compiler_Driver_NVCC.pdf (2013).

[37] Nath, R., Tomov, S. and Dongarra, J.: An Improved MAGMA GEMM for Fermi GPUs, *University of Tennessee Computer Science Technical Report*, No. UT-CS-10-655 (2010).

[38] Nakasato, N.: A Fast GEMM Implementation On the Cypress GPU, *ACM SIGMETRICS Performance Evaluation Review - Special issue on the 1st international workshop on performance modeling, benchmarking and simulation of high performance computing systems (PMBS 10)*, Vol. 38, pp. 50–55 (2011).

[39] Nakata, M., Takao, Y., Noda, S. and Himeno, R.: A Fast Implementation of Matrix-matrix Product in Double-double Precision on NVIDIA C2050 and Application to Semidefinite Programming, *Proc. 3rd International Conference on Networking and Computing (ICNC 2012)*, pp. 68–75 (2012).

[40] Nakata, M.: The MPACK; Multiple precision arithmetic BLAS (MBLAS) and LAPACK (MLAPACK), http://mplapack.sourceforge.net/ (2010).

[41] Hanrot, G., Lefèvre, V., Pélissier, P., Théveny, P. and Zimmermann, P.: MPFR : GNU MPFR Library, http://www.mpfr.org/.

[42] Li, X., Demmel, J., Bailey, D., Hida, Y., Iskandar, J., Kapur, A., Martin, M., Thompson, B., Tung, T. and Yoo, D.: XBLAS – Extra Precise Basic Linear Algebra Subroutines, http://www.netlib.org/xblas/.

[43] Dongarra, J., Beckman, P., Aerts, P., Cappello, F., Lippert, T., Matsuoka, S., Messina, P., Moore, T., Stevens, R., Trefethen, A. and Valero, M.: The International Exascale Software Project: a Call To Cooperative Action By the Global High-Performance Community, *International Journal of High Performance Computing Applications*, Vol. 23, pp. 309–322 (2009).

[44] Bell, N. and Garland, M.: Efficient Sparse Matrix-Vector Multiplication on CUDA, *NVIDIA Technical Report*, No. NVR-2008-004 (2008).

[45] Xu, W., Zhang, H., Jiao, S., Wang, D., Song, F. and Liu, Z.: Optimizing Sparse Matrix Vector Multiplication Using Cache Blocking Method on Fermi GPU, *Proc. 13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD 2012)*, pp. 231–235 (2012).

[46] Feng, X., Jin, H., Zheng, R., Hu, K., Zeng, J. and Shao, Z.: Optimization of Sparse Matrix-Vector Multiplication with Variant CSR on GPUs, *Proc. IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS 2011)*, pp. 165–172 (2011).

[47] Matam, K. and Kothapalli, K.: Accelerating Sparse Matrix Vector Multiplication in Iterative Methods Using GPU, *Proc. International Conference on Parallel Processing (ICPP 2011)*, pp. 612–621 (2011).

[48] Kubota, Y. and Takahashi, D.: Optimization of Sparse Matrix-Vector Multiplication by Auto Selecting Storage Schemes on GPU, *Proc. 11th International Conference on Computational Science and Its Applications (ICCSA 2011)*, Lecture Notes in Computer Science, Vol. 2, No. 6783, pp. 547–561 (2011).

[49] Baskaran, M. and Bordawekar, R.: Optimizing Sparse Matrix-Vector Multiplication on GPUs, *IBM Research Report*, No. RC24704 (2009).

[50] Guo, P. and Wang, L.: Auto-Tuning CUDA Parameters for Sparse Matrix-Vector Multiplication on GPUs, *Proc. International Conference on Computational and Information Sciences (ICCIS 2010)*, pp. 1154–1157 (2010).

[51] Zein, A. E. and Rendell, A.: Generating Optimal CUDA Sparse Matrix Vector Product Implementations for Evolving GPU Hardware, *Proc. Concurrency and Computation: Practice and Experience*, Vol. 24, pp. 3–13 (2012).

[52] Reguly, I. and Giles, M.: Efficient sparse matrix-vector multiplication on cache-based GPUs, *Proc. Innovative Parallel Computing: Foundations and Applications of GPU, Manycore, and Heterogeneous Systems (InPar 2012)*, pp. 1–12 (2012).

[53] Yoshizawa, H. and Takahashi, D.: Automatic Tuning of Sparse Matrix-Vector Multiplication for CRS format on GPUs, *Proc. 15th IEEE International Conference on Computational Science and Engineering (CSE 2012)*, pp. 130–136 (2012).

[54] Davis, J. and Chung, E.: SpMV: A Memory-Bound Application on the GPU Stuck Between a Rock and a Hard Place, *Microsoft Technical Report*, Vol. MSR-TR-2012-95 (2012).

[55] Barrett, R., Berry, M., Chan, T., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C. and der Vorst, H. V.: *Templates for the So-*

lution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition, SIAM (1994).

[56] Baboulin, M., Buttari, A., Dongarra, J., Kurzak, J., Langou, J., Langou, J., Luszczek, P. and Tomov, S.: Accelerating scientific computations with mixed precision algorithms, *Computer Physics Communications*, Vol. 180, No. 12, pp. 2526–2533 (2009).

[57] Renard, Y.: Gmm++ user documentation Release 4.2, http://download.gna.org/getfem/doc/gmm_userdoc.pdf (2013).

[58] The Scalable Software Infrastructure Project: Lis User Guide Version 1.4.15, http://www.ssisc.org/lis/lis-ug-en.pdf (2013).

[59] Kotakemori, H., Fujii, A., Hasegawa, H. and Nishida, A.: Implementation of Fast Quad Precision Operation and Acceleration with SSE2 for Iterative Solver Library, *IPSJ Transactions on Advanced Computing Systems*, Vol. 1, No. 1, pp. 73–84 (2008). (in Japanese).

[60] Furuichi, M., May, D. and Tackley, P.: Development of a Stokes flow solver robust to large viscosity jumps using a Schur complement approach with mixed precision arithmetic, *Journal of Computational Physics*, Vol. 230, No. 24, pp. 8835–8851 (2011).

[61] Saito, T., Ishiwata, E. and Hasegawa, H.: Analysis of the GCR method with mixed precision arithmetic using QuPAT, *Journal of Computational Science*, Vol. 3, No. 3, pp. 87–91 (2012).

[62] Gravvanis, G., Filelis-Papadopoulos, C. and Giannoutakis, K.: Solving finite difference linear systems on GPUs: CUDA based Parallel Explicit Preconditioned Biconjugate Conjugate Gradient type Methods, *The Journal of Supercomputing*, Vol. 61, No. 3, pp. 590–604 (2012).

[63] Naumov, M.: Incomplete-LU and Cholesky Preconditioned Iterative Methods Using CUSPARSE and CUBLAS, https://developer.nvidia.com/content/incomplete-lu-and-cholesky-preconditioned-iterative-methods-using-cusparse-and-cublas (2011).

[64] Lei, Y., Dou, Y., Shen, L., Zhou, J. and Guo, S.: Special-purposed VLIW architecture for IEEE-754 quadruple precision elementary functions on FPGA, *Proc. IEEE 29th International Conference on Computer Design (ICCD 2011)*, pp. 219–225 (2011).

[65] Nagai, T., Yoshida, H., Kuroda, H. and Kanada, Y.: Fast Quadruple Precision Arithmetic for Multiply/Add Operations on SR11000/J2, *Proc. International Conference on Scientific Computing (CSC 2007)*, pp. 151–157 (2007).

[66] Nagai, T., Yoshida, H., Kuroda, H. and Kanada, Y.: Fast Quadruple Precision Arithmetic Library on Parallel Computer SR11000/J2, *Proc. 8th International Conference on Computational Science, Part I*, ICCS '08, pp. 446–455 (2008).

[67] Tsutsui, N., Yoshida, H., Kuroda, H. and Kanada, Y.: Implementation and Evaluation of FFT Using Improved Quadruple Precision Arithmetic on SR11000/J2, *IPSJ SIG Techinical Report*, Vol. 2008–HPC–116, No. 11, pp. 61–66 (2008). (in Japanese).

[68] Goto, K.: GotoBLAS2, http://www.tacc.utexas.edu/tacc-projects/gotoblas2/.

[69] Whaley, R. C. and Petitet, A.: Minimizing development and maintenance costs in supporting persistently optimized BLAS, *Software: Practice and Experience*, Vol. 35, No. 2, pp. 101–121 (2005).

[70] Kurzak, J., Tomov, S. and Dongarra, J.: Autotuning GEMM Kernels for the Fermi GPU, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 23, No. 11, pp. 2045–2057 (2012).

Appendix A

List of Publications

Journal Papers (in Japanese)

1. 椋木大地，高橋大介：GPU における 3 倍・4 倍精度浮動小数点演算の実現と性能評価，情報処理学会論文誌 コンピューティングシステム（ACS41），Vol. 6，No. 1，pp. 66-77，2013 年 1 月.

Conference Papers

1. Daichi Mukunoki and Daisuke Takahashi: Implementation and Evaluation of Quadruple Precision BLAS Functions on GPUs, Proc. 10th International Conference on Applied Parallel and Scientific Computing (PARA 2010), Part I, Lecture Notes in Computer Science, No. 7133, pp. 249-259, March 2012.

2. Daichi Mukunoki and Daisuke Takahashi: Implementation and Evaluation of Triple Precision BLAS Subroutines on GPUs, Proc. 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW 2012), The 13th Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC-12), pp. 1378-1386, May 2012.

3. Daichi Mukunoki and Daisuke Takahashi: Optimization of Sparse Matrix-vector Multiplication for CRS Format on NVIDIA Kepler Architecture GPUs, Proc. The 13th International Conference on Computational Science and Its Applications (ICCSA 2013), Lecture Notes in Computer Science, No. 7975, pp. 211-223, Jun 2013.

Other Publications (in Japanese)

1. 椋木大地，高橋大介：GPU における 4 倍精度浮動小数点演算を用いたクリロフ部分空間法の高速化，情報処理学会研究報告，2013-HPC-140，No. 25，2013 年 8 月.

2. 椋木大地，高橋大介：GPU における高速な CRS 形式疎行列ベクトル積の実装，情報処理学会研究報告，2013-HPC-138，No. 5，2013 年 2 月.

3. 椋木大地，高橋大介：GPU における 4 倍精度演算を用いた疎行列反復解法の実装と評価，情報処理学会研究報告，2012-ARC-202，2012-HPC-137，No. 37，2012 年 12 月.

4. 椋木大地，高橋大介：GPU による 3 倍精度浮動小数点演算の検討，情報処理学会研究報告，2011-ARC-197，2011-HPC-132，No. 23，2011 年 11 月.

5. 椋木大地，高橋大介：GPU による 4 倍・8 倍精度 BLAS の実装と評価，2011 年ハイパフォーマンスコンピューティングと計算科学シンポジウム HPCS2011 論文集，pp. 148-156，2011 年 1 月.

6. 椋木大地，高橋大介：GPU による 4 倍精度 BLAS の実装と評価，計算工学講演会論文集，Vol. 15，No. 2，pp. 891-894，2010 年 5 月.

7. 椋木大地，高橋大介：GPU による 4 倍精度 BLAS の実装と評価，情報処理学会研究報告，2009-ARC-186，2009-HPC-123，No. 13，2009 年 11 月.