Modeling and Optimization of Parallel Matrix-based Computations on GPU

by

Andrew White

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
May 5, 2013

Keywords: GPU, matrix-based computations, parallelization procedure, execution metrics

Approved by

Soo-Young Lee, Chair, Professor of Electrical and Computer Engineering
Victor Nelson, Professor of Electrical and Computer Engineering
Chwan-Hwa Wu, Professor of Electrical and Computer Engineering

Abstract

As *graphics processing units* (GPUs) are continually being utilized as coprocessors, the demand for optimally utilizing them for various applications continues to grow. This work narrows the gap between programmers and minimum execution time for matrix-based computations on a GPU. To minimize execution time, computation and communication time[1] must be considered. For computation, the placement of data in GPU memory significantly affects computation time and therefore is considered. Various matrix-based *computation patterns* are examined with respect to the layout of GPU memory. A computation pattern refers to the order in which each GPU thread computes a result. From examination of computation patterns, an *optimized computation pattern*, a pattern which reduces computation time, is derived. After the optimized computation pattern is derived, the *access pattern* to GPU memory, or order in which data is accessed, is considered. From the *optimized access pattern*, fine-tuning is performed to the GPU code such as minimization of index calculations and loop unrolling to further reduce computation time and resource allocation. After fine-tuning, *input parameters* which yield the minimum computation time for a matrix-based computation on the GPU are derived. Input parameters are defined as the dimensions of the grid and blocks assigned for execution on the GPU. *Execution metrics* are formulated, as functions of the input parameters, to represent the executional behavior of the GPU. The execution metrics are utilized to derive the *optimal input parameters* which are input parameters that yield the minimum computation time.

[1]In this work, computation time refers to the amount of time required for the GPU to execute a given computation. Communication time refers to the amount of time required for a CPU to GPU or a GPU to CPU data transfer.

The matrix-based computations considered are *matrix-vector multiplication* (Mv), *matrix-matrix multiplication* (MM), *convolution*, and the *conjugate gradient* method. A parallelization procedure is developed to minimize execution time by deriving the optimal placement of data, optimized computation pattern, optimized access pattern, and optimal input parameters. Each step of the procedure is developed through analysis of Mv and MM. The procedure includes an accurate communication model and estimates for CPU and GPU data transfers. With accurate estimates for communication and computation times, partitioning computation for CPU and GPU execution is performed to minimize execution time. Convolution and conjugate gradient are utilized to verify the validity of the procedure. Therefore, the overall goal of this work is to develop a parallelization procedure which minimizes execution time of matrix-based computations executing on the GPU.

Acknowledgments

First, I would like to thank God for blessing me with the opportunity to complete this work. Countless prayers from family, friends and myself enabled me to finish this dissertation. I would like to thank my wife, Abigail, who has always been supportive through the many years. Her dedication is truly inspiring. My family, particularly my parents, Bob and Ann, provided financial assistance and emotional support which none of this would be possible without. The lengthy phone calls about this work with my father also provided an outside perspective for improvement and I am sincerely grateful.

I would like to thank the entire Department of Electrical Engineering at Auburn University. Specifically, I would like to acknowledge my advisor, Dr. Soo-Young Lee, for always being supportive. This work would not have been possible without his constant help and guidance. Through countless meetings, emails, and phone calls, Dr. Lee always believed we could finish this work and went above and beyond expectations of an advisor.

I would also like to acknowledge Dr. Victor Nelson and Dr. Chwan-Hwa Wu for their thorough review of this work as well as availability to discuss any problems encountered along the way. Lastly, Dr. Amnon Meir, the outside reader of this work, provided quick and helpful revisions for completion. It was always a pleasure meeting with these professors and I truly appreciate the help.

Lastly, I would like to acknowledge the graduate students with whom I have shared an office for many years at Auburn. It was always enjoyable and relaxing to discuss life, and of course work, with Dr. Chris Wilson over lunch. I also appreciate the conversations with Siwei Wang and Praveen Venkataramani which provided me a wider understanding of the rest of the world and some social interaction in an otherwise non-social environment.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Definition

As the market for massively multithreaded architectures continues to grow, so does the development of general-purpose graphics processing unit (GPGPU) applications. Many programming models such as NVIDIA's *compute unified device architecture* (CUDA) try to ease the gap between programmers and GPUs. While much research has been done in optimizing written GPU applications, little has been done to bridge the gap between programmers and minimum execution time for matrix-based computations on a GPU. Therefore, a parallelization procedure is necessary to provide programmers a guide to achieve minimum execution time for matrix-based computations on a GPU.

To develop a parallelization procedure, it is necessary to consider computation and communication time. Accurate estimates for computation and communication time are necessary to minimize execution time. To assist in modeling GPU computation time, the layout of GPU memory must be examined for various matrix-based computation patterns. Therefore, it is necessary to determine in which type of GPU memory data is placed. After the placement of data is determined, the order in which threads executing on the GPU compute results, the computation pattern, is considered. Therefore, a computation pattern which reduces the computation time with respect to the layout of GPU memory for matrix-based computations must be derived. After deriving the optimized computation pattern, it is necessary to derive an optimized access pattern to GPU memory since the order in which threads access memory affects computation time. Utilizing the optimized access pattern, the code must be fine-tuned to reduce computation and resource allocation. From the fine-tuned code, it is necessary to derive the input parameters for the GPU computation which yield

the minimum computation time. To derive optimal input parameters, an accurate model of the computational behavior of the GPU is required. Since modeling GPU computation using input parameters is insufficient, execution metrics must be formulated as functions of the input parameters to model the behavior. From the execution metrics, optimal input parameters must then be derived which yield the minimum computation time for matrix-based computations.

Communication estimates between the CPU and GPU are also necessary to develop a parallelization procedure which minimizes execution time. Accurate communication estimates, in addition to accurate computation estimates, are necessary to determine which computations are performed by the CPU and GPU to minimize execution time.

Therefore, a parallelization procedure which minimizes execution time of matrix-based computations on the GPU must consider the placement of data, computation patterns, access patterns, input parameters, communication time, and computation partitioning.

1.2 Review

In early GPU research, Fatahalian and others performed a study of matrix-matrix multiplication on GPU architectures [1]. It was shown at the time that GPUs suffer from memory bandwidth limitations which has later been expounded upon by many research groups. At that time, work showed that CPUs were better candidates for applications that feature data reuse such as matrix-matrix multiplication. Since then, GPU architecture has evolved creating an increasing atmosphere for GPUs being utilized as coprocessors.

Shortly after 2004, work began on automatically tuning matrix-matrix multiplication for GPUs [2]. However, results proved that automatic tuning of GPU programs severely reduced performance. After that, a Microsoft research group proposed a system to program GPUs [3]. Results proved that CPUs were several times faster than GPUs at executing computations such as Mv. After that, research began to focus on modeling GPU performance [4]. At the

time, results showed accurate estimations of GPU computation time. However, the advent of CUDA and CUDA-enabled devices obsoleted this work.

Researchers at the Georgia Institute of Technology attempted to predict GPU performance in very fine detail by examining the PTX code (NVIDIA assembly code equivalent) to determine the number of computation and memory access cycles and using these to determine the GPU computation time [5, 6]. This work was utilized to predict the optimal number of active cores for the GPU and to disable some of the cores to reduce energy consumption [7]. Although the work provides a detailed algorithm to count the number of instructions from PTX code, this can be a tedious job to calculate the cycles and predict the result. Analysis at a higher level allows a more general approach and can reduce the time required to yield a minimum execution time. In addition, while the predictions for older model GPUs such as the FX5600 are mostly accurate, the behavior of the predicted time on newer GPUs, such as 200-series architectures, is not the same as the behavior of the measured time. Moreover, the work provided a static approach to predicting the computation time but left optimizations up to the programmer. Most importantly, the layout of the global memory in the GPU was not considered which largely affects the computation time for matrix-based computations on GPUs.

Researchers at the International Institute of Information Technology in Hyderabad, India focused on explaining the behavior of current NVIDIA GPUs and worked on developing a model to illustrate the performance of an NVIDIA GPU [8]. Their work focused on creating a simulator for the GPU in the future which would aid in writing optimized applications. However, their research, as stated in publications, does not aim to provide a procedure to programmers which minimizes execution time for GPU applications.

Researchers at the Center for Reliable and High-Performance Computing at the University of Illinois at Urbana-Champaign have also spent time researching GPUs and their performance [9–13]. The work is a discussion of balancing shared resource usage to achieve high performance on the GeForce 8800 [9]. They focused on generating enough threads to

hide memory latency, choosing applications with a high percentage of floating point operations such as MM, and reducing the number of global memory accesses. They use this example to show that shared memory is useful for reducing redundant loads and thus reducing the pressure on memory bandwidth. In addition, some of their research focuses on resource utilization by the kernel such as shared memory, registers and occupancy [10, 11]. Similar to previous work, PTX code is examined for modeling computation time. The research focused on minimizing the number of trials necessary to find optimal input parameters rather than deriving them. In addition, the research did not consider the layout of global memory or access pattern and thus does not work well for memory-bound problems such as many matrix-based computations.

In addition to the previous work mentioned, the group at the Center for Reliable and High-Performance Computing at the University of Illinois at Urbana-Champaign developed a program called CUDA-lite [14]. This was developed to be an enhancement to CUDA that automates the task of selecting the appropriate memory to use and the coding of data transfers between memories. The tool was written to coalesce memory accesses which was a roadblock to GPU performance at the time. However, NVIDIA continues to relax the requirements to achieve coalesced read/writes with every SDK released so there is a dwindling need for a programmer to use an automated compiler to handle such a task. In addition, requiring programmers to learn other tools in addition to CUDA creates an additional burden. Regardless, it should be assumed that programmers understand the basic concepts of the memory hierarchy and thus do not need additional tools to optimize the access patterns if some basic guidelines are followed when writing applications. Lastly, this work does not consider the global memory layout and can produce poor results in such cases as utilizing shared memory to coalesce accesses.

The most recent research by this group includes an adaptive performance modeling tool [12] and a CUDA application survey [13]. In [13], a survey is performed of different

applications and the suitability of CUDA-enabled GPUs. While this provides a useful summary of the architecture and how the GPU operates, along with analysis as to why certain applications are optimal, it does not satisfy the need for a procedure to minimize computation time. In [12], a compiler-based approach is utilized to estimate GPU computation time. They, too, analyzed the effects on computation time from utilizing various GPU memories. While this was a major step in automatic selection of optimal applications, it does not provide a procedure which yields optimal GPU code and derivation of input parameters that yield the minimum computation time.

With the release of CUDA, NVIDIA released CUBLAS [15], an optimized implementation of the BLAS [16] routines for GPUs. However, source for CUBLAS is closed and little work is published on how each routine is implemented. Alternatively, MAGMA [17] released an optimized implementation of the BLAS routines similar to LAPACK [18] for GPUs. Several works [19–34] provide insights into optimized GPU applications. However, the research aims to provide users optimized BLAS routines rather than provide programmers a procedure for optimizing GPU computations and minimizing execution time.

Other work [35–48] also focuses on optimizing individual computations, such as Mv or MM, but is limited to specific implementations or ignores certain aspects of the GPU such as memory layout.

Lastly, work has been performed that highlights the programming experiences with CUDA as well as the architectures of GPUs [49–52]. [49] provides a summary of research done with different applications on GPUs utilizing CUDA. The work highlights the benefits of CUDA as well as what is necessary to make applications suitable for a GPU, such as exposing sufficient amounts of fine-grained parallelism to occupy the device, ability to block computation, efficiency of data-parallel programs without thread divergence and finally, usage of shared memory. [51] focuses on the architecture of Tesla GPUs and how it has evolved over the last decade from fixed-function graphics pipelines to programmable processors with computing power greater than multi-core CPUs. One of the newest GPU architectures,

Fermi, is outlined in [52], which also discusses the demand for increasing GPU capabilities and the future of GPUs. The research shows the benefit of co-processing architectures (such as CUDA) and explains the reason the market for these architectures and their usability will continue to grow. Because of this expanding market, research is needed to provide users with a procedure for achieving the minimum execution time.

1.3 Motivation

With the development of environments such as CUDA, GPUs are increasingly becoming a viable option for parallel processing. With the increased interest in utilizing the GPU as a coprocessor, it is necessary that programmers be able to write applications for this platform without the need to fully understand the underlying architecture of the GPUs. Often, GPU manufacturers do not disclose certain aspects of the hardware, or sometimes even the software, to maintain a competitive advantage. Therefore, researchers are constantly testing and analyzing the new generations of GPUs to understand how to fully utilize them. It is necessary to provide programmers a procedure for minimizing the execution time while also providing a logical reasoning.

Previous research does not provide programmers a parallelization procedure for matrix-based computations which yields minimum execution time. More specifically, previous research does not account for the layout of GPU global memory or the access pattern to that memory, which is determined by the code, and thus, optimized access patterns are not derived to minimize computation time. In addition, previous research does not consider the computation pattern by the executing threads, which affects computation time. Therefore, optimized computation patterns are not derived to achieve the minimum time. In previous research, input parameters are determined through testing the application, and thus, optimal input parameters are not derived for various computations to yield the minimum computation time. Therefore, it is necessary to develop a parallelization procedure for matrix-based

computations executing on a GPU which minimizes execution time by considering the placement of data, computation patterns, access patterns, input parameters, communication time, and computation partitioning.

1.4 Objectives

Since the primary goal of this work is to develop a procedure which yields the minimum execution time for matrix-based computations on a GPU, the objectives are to

- examine the layout of GPU memory,
- formulate execution metrics to accurately represent the computational behavior of the GPU,
- model and estimate CPU to GPU and GPU to CPU communication time,
- model and estimate CPU and GPU computation time,
- determine the optimal placement of data in GPU memory,
- analyze computation patterns for GPUs and derive the optimized computation pattern,
- derive the optimized access pattern for GPU memory,
- fine-tune code to minimize computation and resource allocation,
- from the execution metrics, derive the optimal input parameters for the GPU,
- determine the optimal partitioning of computation between the CPU and GPU,
- verify these objectives by comparing the procedure with several matrix-based computations.

This is accomplished with consideration to grid and block partitioning, data arrangement and partitioning, code arrangement and data transfers. Therefore, the intellectual contribution

of this work is a parallelization procedure which minimizes execution time for matrix-based computations on a GPU.

1.5 Organization

Chapter 2 is an introduction to GPUs, CUDA and terms utilized in this work. An examination of the layout of GPU memory and the formulation of execution metrics to accurately represent the computational behavior of the GPU are presented in Chapter 3. In addition, modeling communication and computation times is presented in Chapter 3. From the layout of GPU memory and formulation of execution metrics, the parallelization procedure to minimize execution time for matrix-based computations executing on a GPU is developed in Chapter 4. The parallelization procedure considers the placement of data, computation patterns, access patterns, fine-tuning, input parameters, communication, and computation partitioning. The procedure is developed through and applied to Mv and MM in Chapter 4. Results are included to illustrate the impact on time for each step of the procedure. In Chapter 5, the procedure is applied to convolution and the conjugate gradient method. The application of the procedure to convolution is shown. Results illustrate the performance of the procedure applied to convolution and the conjugate gradient method. Chapter 6 is a summary and conclusion of the presented work.

Chapter 2

GPU, CUDA and Terms

This chapter includes a brief history of GPUs, a comparison of CPU and GPU architectures, and the performance of GPUs. Following is a summary of CUDA, the NVIDIA GPU programming environment, and lastly, terms are introduced.

2.1 GPU History

As GPUs continue to become low-cost, parallel processing architectures, additional programmers will turn to these architectures to minimize execution time. Therefore, there is an increasing need for optimizing applications written for GPUs. While much work has been done by NVIDIA and other researchers using the CUDA environment to explain the impact of different application parameters, much information is not public knowledge about how GPUs function. Therefore, it is necessary to expand upon the previous research into optimizing applications, and develop a procedure for minimizing execution time to ease future programmers into writing efficient CUDA applications for GPUs.

Over the last 30 years, GPU architecture has evolved to a massively parallel design. In the 1980s, GPUs were large expensive systems that typically cost in the range of $50,000 and were capable of processing 50 million pixels per second. By the 1990s, GPU costs decreased significantly, causing them to be more widespread as they were deployed in small workstations as PC accelerators. During this time, graphics APIs such as DirectX became popular and programmers began utilizing the fixed-functions to perform other tasks. In the 2000s, GPUs dropped in price to the $100 range and became available on every computer. Those GPUs were capable of processing 1 billion pixels per second. More importantly, those GPUs became programmable for general purpose computing using such languages as CUDA and Open CL.

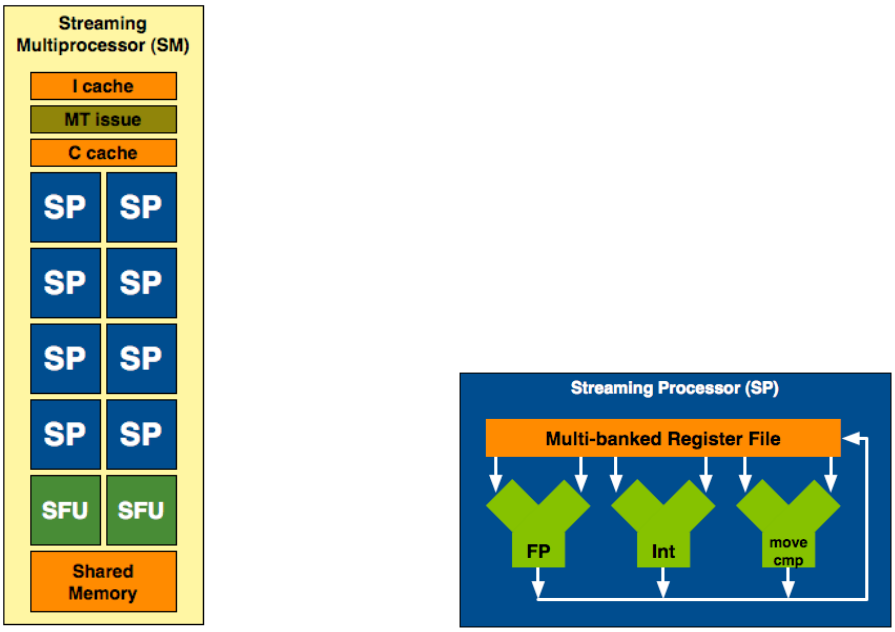
This advancement in graphics performance has been driven by the market demand for high-quality, real-time graphics in computer applications, namely the gaming industry. The result is that, over time, graphics architecture has changed from being a simple pipeline for drawing wire-frame diagrams to a highly parallel computing chip [53] [52] [51].

In order to fully utilize today's chips and maximize performance, GPU algorithms need several key components: access to data with minimal bank conflicts, SIMD parallelism and many arithmetic computations. GPU algorithms are best suited for computationally intensive applications that require little inter-process communication and include but are not limited to physical modeling, computational engineering, matrix algebra, convolution, correlation and sorting. Therefore, matrix-based computations are ideally suited for GPUs.

2.2 GPU Architecture

GPUs, as opposed to CPUs, are not well-suited for all algorithms due to their architecture. CPUs are designed for optimal sequential performance and application performance increases with increasing clock frequencies. However, the rate at which clock speeds increase is beginning to slow. Because of the importance of sequential performance, CPU cores include the full x86 instruction set with complex branching. In addition CPUs have large cache memories to decrease instruction and data latencies for applications. The larger cache memories and the need for complex control logic forces designers to use less of the available silicon for ALUs. On the other hand, GPUs are designed for optimal parallel performance. While GPU clock speeds are slower than CPU speeds, the application performance increases with the number of cores. The lack of large cache memories and complex control logic decreases the sequential performance in comparison to CPUs but allows designers to implement more ALUs on the available silicon thus increasing the numeric computing capability. Lastly, since GPUs are designed with a focus on parallelism, their structure includes many smaller cores than CPUs.

The NVIDIA GPU[1] architecture is a collection of *streaming multiprocessors* (SMs) with each SM having a number of cores or *streaming processors* (SPs). The control logic and instruction cache for all SPs within an SM are shared as illustrated in Figure 2.1a. This work utilizes the *Tesla T10* (T10) GPU which is built on the NVIDIA 200-series architecture. The T10 architecture has a total of 30 SMs with each SM having 8 SPs. A newer architecture, Fermi, includes only 16 SMs but each SM has 32 SPs. In both architectures, each SP consists of an ALU and FPU as illustrated in Figure 2.1b. The 200-series architecture has a 24-bit ALU while the Fermi architecture includes a 32-bit ALU. However, the 200-series architecture is capable of 32-bit precision arithmetic through the use of multiple arithmetic instructions.



(a) Streaming Multiprocessor. (b) Streaming Processor.

Figure 2.1: Architecture of the T10 GPU.
Source: `http://www.anandtech.com/show/2549/2`

In current GPU systems, the GPU and CPU have a separate memory space with a GPU consisting of three separate types of memory: global, constant and shared. Figure 2.2 depicts the organization of GPU memory. Global and constant memory are accessible by all

[1]From this point forward, GPU refers to an NVIDIA GPU.

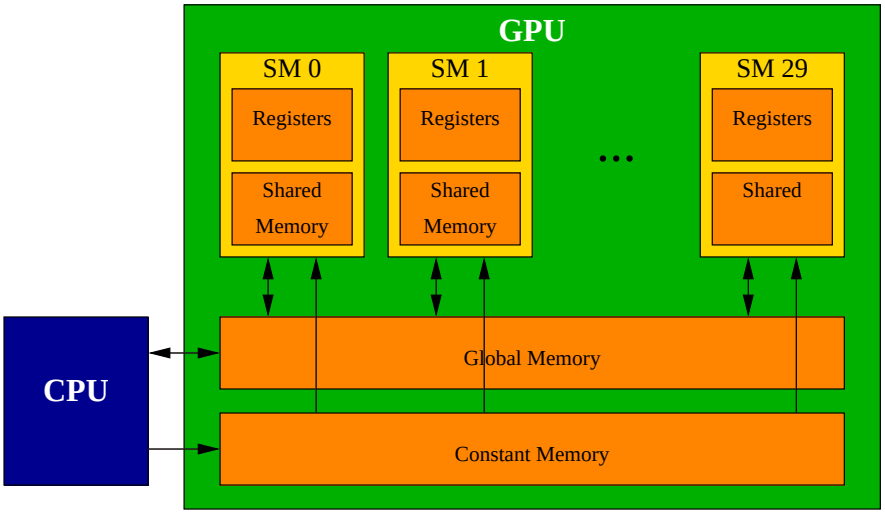


Figure 2.2: Memory organization of the T10 GPU.

SPs while shared memory is partitioned for each SM. Therefore, each SP in the same SM can share data through the shared memory. However, SPs from one SM cannot share data with SPs from a different SM. Global memory is the slowest memory available on the GPU and is 4GB in size for the Tesla T10 GPU. It is utilized as a transfer medium for the host (CPU) to the device (GPU) or for the device to the host. The GPU can read and write to the global memory while the CPU can copy memory to or from the global memory. Global memory is persistent through kernel calls while constant and shared are not. A CUDA application can transfer data from the system memory at $4GB/s$ and at the same time upload data to the system memory at $4GB/s$. Constant memory is much faster and smaller than the global memory and is only $64KB$ in size on the Tesla T10. However, the GPU cannot write to constant memory therefore it is simply used to transfer constant data from the CPU to the GPU. Shared memory is also faster and smaller than global memory and can be written to and read from by the GPU. There are $16KB$ of shared memory per SM for the Tesla T10. However, shared memory cannot be accessed by the CPU therefore it is typically used as a scratch-pad memory for each SP. All computational results generated by the GPU must be stored in global memory to be accessible by the CPU. In some applications, shared memory can also be used for sharing data amongst SPs in the same SM. In addition to

the three types of memory, each thread has access to registers and local memory[2] both of which are read/write. The shared memory can be read/written by all threads in the same SM. However, threads from different SMs cannot access the same shared memory while the constant memory can be read by any thread. The global memory can be read/written by any thread.

2.3 GPU Performance

Because of differences in architecture, GPUs have continually outperformed CPUs in terms of floating-point operations per second. "As of 2009, the ratio between many-core GPUs and multicore CPUs for peak floating-point calculation throughput is about 10 to 1" [53]. The fundamental difference in the design of CPUs versus GPUs has created this large increase in performance in terms of FLOPS.

While the CPU is designed to optimize sequential performance, GPUs are designed to optimize parallel instructions throughput. Therefore, GPUs need thousands of threads executing in parallel to achieve full efficiency where a CPU may only need a few. However, each thread of a GPU is lightweight in comparison and requires little overhead to create. The NVIDIA 200-series architecture supports 1024 threads per SM for a total of 30720 threads running in parallel for the entire chip. In addition, new generations of hardware, such as the Fermi, support even more threads running concurrently. Today's Tesla C1060, currently sold by NVIDIA, includes one Tesla T10 processor built on the 200-series architecture. The T10 includes 240 cores, each with a clock speed of $1.33GHz$ and is connected to $4GB$ of DRAM. The C1060 fits in a standard PCIe dual slot and consumes around 160 watts of power. Because of these low-cost, low-power GPUs, massively parallel applications are quickly being ported for GPGPU use.

To date, more than 200 million GPUs have been deployed, providing economical parallel processing around the globe. GPGPU hardware is currently being used in a variety of

[2]Local memory is a subsection of global memory used to store variables when the maximum amount of registers is exceeded.

applications, including MRI products. In the past, typical parallel processing research was focused on using large clusters but actual clinical MRI machines needed to be much smaller than that. Because of this, groups such as the National Institutes of Health (NIH) would not fund parallel processing research since it was considered to be limited to large cluster-based machines. However, current MRI machine manufacturers ship MRI products with GPUs and the NIH funds research using GPUs as coprocessors [53].

Previously, a large drawback of GPGPU applications was the necessary programming. In the past, programmers had to learn to program the GPUs using drivers and assembly language. These languages also did not include many useful general computing instructions like integer or bit operations since they were not implemented at the time. In addition, a lack of communication between any of the processors in the GPU was limited making any data sharing difficult. While this is still somewhat of a limitation, it has been greatly improved, since the original design, through CUDA.

2.4 CUDA Environment

Since GPUs were starting to be used as massively parallel processors, NVIDIA developed a programming interface to utilize them known as CUDA. CUDA stands for Compute Unified Device Architecture and is based on the C/C++ language. It enables programmers to write applications in either C or C++ to utilize the GPU as a massively parallel co-processor. In addition, since MATLAB version 2010b, there is support for m-files to use the CUDA environment allowing access for computation on the GPU. CUDA provides constructs for memory transfers between the CPU and GPU for sharing data. It is based on an SPMD programming style which executes the same program on multiple parts of the data. This is different than SIMD since the processing units do not have to be executing the same instruction at the same time. In the CUDA programming environment, there is a host which is the CPU and a device which is the GPU. The NVIDIA CUDA programming language is a large reason more and more industries are looking at GPUs as portable parallel processing

systems. The CUDA extension of the C programming language has enabled all programmers to create parallel applications without any special hardware or software knowledge beyond C. However, knowledge of the underlying architecture to achieve high speedups has inhibited typical programmers from using GPUs as coprocessors.

2.5 Terms

Using CUDA, a C/C++ program consists of *host* code, code written to execute on the CPU, and *device* code, code written to execute on the GPU. Device code is divided into individual functions known as *kernels*. To compile kernels, NVIDIA provides a C compiler known as NVCC. It separates the host and device code and only performs compilation on the device code while using the gcc for the host code. In addition, NVIDIA provides a low-level programming language, parallel thread execution, *PTX*, similar to assembly language for writing kernels without exposing any of the underlying instruction set. Kernels can be written for any computational operation involving the GPU and are similar to functions in C. They are executed using a specified number of *threads*. Since the GPU is an SPMD structure, all threads execute the same code on different portions of data and each thread has an identifier known as the thread index. Thread identifiers are used for control logic and data access. A collection of threads is a *block*, as shown in Figure 2.3. A block can be partitioned in 1, 2, or 3-dimensions. The number of threads in the x- and y-dimension of each block is defined by $dBlk.x$ and $dBlk.y$, respectively. Threads in the same block

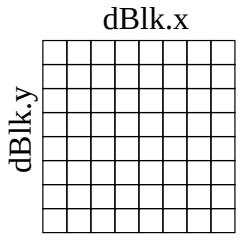


Figure 2.3: A square block of size 64.

can cooperate through atomic operations and synchronization. However, threads in separate

blocks cannot since there is no guarantee to which SM a block is assigned. Each block also has an identifier similar to a thread, known as the block index. The collection of blocks is considered the *grid* and can be partitioned in 1 or 2-dimensions. The number of blocks in the x- and y-dimension of the grid is defined by $dGrd.x$ and $dGrd.y$, respectively. Each kernel call can define different grid and block sizes and dimensions but these cannot be changed dynamically during kernel execution.

In the T10 processor, each grid can have a maximum of 65,536 blocks in each dimension and each block can have a maximum of 512 threads yielding a maximum total of 2^{41} threads. The dimensions of a grid and block determine the total number of threads operating in the GPU, so each need to be chosen to ensure the GPU is fully occupied. Since blocks can execute in any order with respect to other blocks, CUDA applications are very scalable which allows applications to adapt to new hardware without changing any of the existing code. However, architecture changes that often accompany new hardware can cause issues with scalability. Once the partitioning for the grid and blocks is determined, the scheduler organizes threads into groups of 32 known as *warps* as shown in Figure 2.4. Since warps

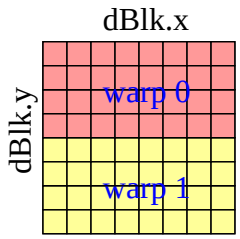


Figure 2.4: Partitioning of a square block of size 64 into warps.

waiting for long-latency operations such as intensive arithmetic operations or memory access are not selected for execution, this provides a type of latency hiding. While those warps are waiting for their operations to finish executing, other warps that are not waiting are scheduled to execute. This type of zero-overhead thread scheduling ensures that the maximum instruction throughput is realized. This is one major reason GPUs do not dedicate as much chip area to cache memories and branch prediction mechanisms as CPUs, thus allowing for many more ALUs to fit in the same size of silicon.

A warp is divided into *half-warps* (HWs), a group of 16 threads in row-major order as shown in Figure 2.5. Memory accesses are issued at the HW level. The number of HWs

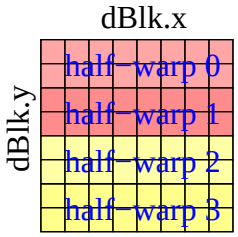


Figure 2.5: Partitioning of a square block of size 64 into half-warps.

is defined by the dimensions of the grid and blocks, *input parameters*, which are specified by the programmer. The four input parameters are defined as $dGrd.x$, $dGrd.y$, $dBlk.x$, and $dBlk.y$. $dGrd.x$ and $dGrd.y$ are the dimensions of the grid in the x and y-dimensions, respectively. $dBlk.x$ and $dBlk.y$ are the dimensions of each block in the x and y-dimensions, respectively. Lastly, the focus of this work is on matrix-based computations. All matrices are assumed to be square and n is used to denote the width or height of a square matrix.

Chapter 3

Modeling

Developing a parallelization procedure to minimize execution time requires an accurate model of communication and computation times. Several modeling techniques, including fine-grain, vector-based, and table-based modeling, were initially applied to model GPU computational behavior. Fine-grain modeling was performed by measuring computation times for arithmetic operations and extrapolating for larger computations. Vector-based modeling was performed by measuring computation times for vector-based arithmetic operations such as a dot product and extrapolating for larger computations. Lastly, table-based modeling was performed by measuring computation time for various numbers of blocks and threads and extrapolating for other input parameters. However, none of these techniques provided reasonable models of the GPU computational behavior, partly due to the layout of the GPU's memory.

Therefore, this chapter includes an examination of the layout of GPU memory and the formulation of execution metrics to accurately represent the computational behavior of the GPU. Section 3.1 is an examination of the layout of global memory on the GPU. Included in this section are the effects on computation time from the layout. Section 3.2 is an examination of the layout of shared memory and the effects on computation time. The beginning of Section 3.3 provides reasoning for utilizing execution metrics to model GPU computational behavior. Instead of utilizing input parameters to model behavior, execution metrics are formulated as functions of the input parameters for modeling. Section 3.4 is a model and estimation of CPU to GPU and GPU to CPU communication time. Results are included to prove the validity of the estimation. CPU computation time is estimated in Section 3.5 through curve-fitting measured data and results are included in the section.

3.1 Global Memory Layout

One reason the aforementioned modeling techniques for GPU computational behavior yield inaccurate models is the layout of global memory on the GPU. Due to the layout of global memory, a problem referred to as *partition camping* exists. Partition camping has been defined by NVIDIA [54], although its effects on various applications have not been studied until recently [55] [56] [57].

3.1.1 Partition Camping

A description of the GPU architecture is necessary to understand partition camping. The global memory on an 8-series and 200-series GPU is divided into 6 and 8 partitions, respectively. On both GPU architectures, each partition is 256-bytes wide. All partitions are connected to a memory controller allowing access by all SMs as depicted in Figure 3.1. Data

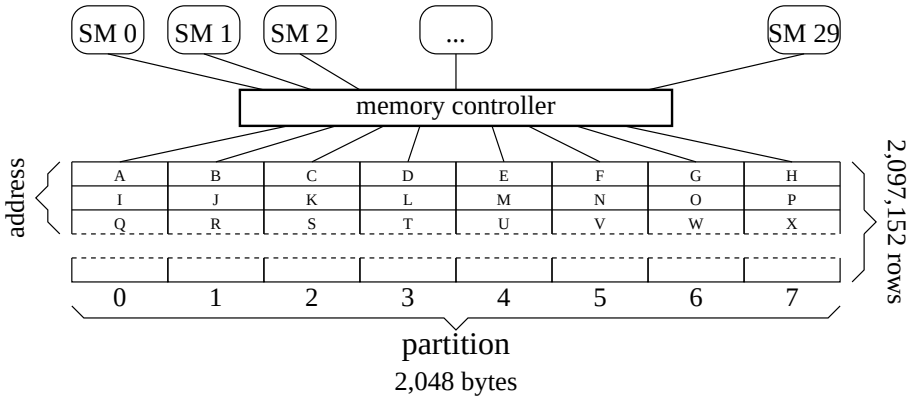


Figure 3.1: Global memory connection of the T10 GPU. Memory is divided into 8 equal-sized partitions. Each address represents 256 bytes.

is stored in row-major order and each address in the figure occupies one row of a partition.

Accesses to global memory partitions are performed at the HW level. Therefore, partition camping occurs when multiple HWs attempt to access the same partition, but not necessarily the same address, at a given time. Figure 3.2 illustrates partition camping occurring when two HWs access two consecutive partitions where Figure 3.1 depicts in which partition each address resides. Since addresses A and I are in the same partition, HW 1

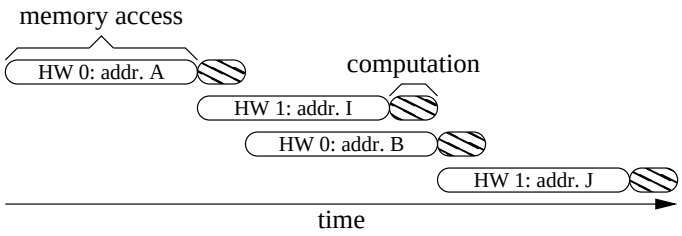


Figure 3.2: An example of partition camping occurring: global memory accesses by 2 HWs to 2 partitions.

must wait, until HW 0 finishes accessing address A, to access address I. The same occurs for the second accesses to addresses B and J. Therefore, partition camping occurs and the memory accesses are serialized.

Figure 3.3 illustrates the access of two partitions by two HWs which eliminates partition camping. For the first access, HW 0 accesses address A which resides in partition 0 and HW

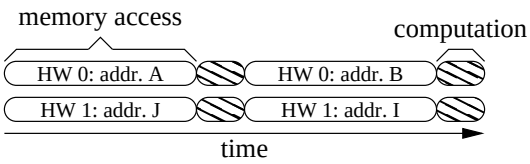


Figure 3.3: An example of partition camping not occurring: global memory accesses by 2 HWs to 2 partitions.

1 accesses address J in partition 1. Partition camping does not occur since the accesses by each HW occur in different partitions. Similarly, for the second access, each HW accesses different partitions and partition camping does not occur. Figures 3.2 and 3.3 demonstrate the theoretical effects on computation time due to partition camping on a GPU such that minimizing partition camping should be performed to minimize computation time.

Effects

For most GPU computations, thousands of HWs are executing at a given time. To minimize partition camping, and thus the computation time, it is necessary that HWs utilize all partitions of global memory. In addition, it is necessary that the HWs be evenly distributed to the partitions.

The code in Listing 3.1 is a portion of a kernel tested to measure the effects of utilizing a varying number of partitions.

```
1 int offset = floor(BlkIdx / BlksPerPart) * 64;
2 for(j = 0; j < v; j++)
3   temp += A[j * 512 + offset + threadIdx.y];
```

Listing 3.1: Code to test the effects of partition camping.

Line 1 calculates which partition of global memory each block accesses by utilizing a linear block index, $BlkIdx$, and the number of assigned blocks per partition, $BlksPerPart$. The fraction of the two is multiplied by 64 since each value is stored as a float and a partition is 256 bytes wide on a 200-series GPU. The length of a vector in the matrix that is summed is represented by v. Adding the thread index in the y-dimension, *threadIdx.y*, minimizes the possible effect of memory request merging [58] [59].

The computation time and effective bandwidth, from executing the code in Listing 3.1 on the T10 GPU, are given in Table 3.1[1]. Each time was measured using a square block of 256 threads and 120 blocks to yield a maximum occupancy for each SM. Since threads are assigned to warps in row-major order, each thread in a HW sums identical values. However, each HW in a block computes different sums. The bandwidth decreases linearly as the

	number of partitions			
v	8	4	2	1
512	0.4	0.9	1.7	3.4
1024	0.9	1.7	3.4	6.7
2048	1.7	3.4	6.8	13.5
4096	3.3	6.8	13.5	26.9
8192	6.8	13.4	26.5	53.8
16384	13.7	26.5	52.3	107.5

(a) Measured computation time (ms).

	number of partitions			
v	8	4	2	1
512	69.4	34.4	17.2	8.7
1024	68.4	34.6	17.2	8.7
2048	69.9	34.5	17.2	8.7
4096	71.7	34.4	17.4	8.7
8192	69.6	35.1	17.7	8.7
16384	68.5	35.4	17.9	8.7

(b) Effective bandwidth (GB/s).

Table 3.1: Measured results of executing Listing 3.1 which demonstrate the effect of partition camping on the T10 GPU. v is the number of values each thread summed.

number of partitions utilized decreases, due to partition camping. Since the number of HWs remains constant in the tests, using less partitions increases partition camping which

[1]These results suggest memory request merging mentioned in [58] and [59] does not occur.

increases the computation time. For MM, all partitions are utilized at some point when $n^2 \geq 512$, where n is the height or width of a square matrix. However, the order in which HWs access each partition varies depending on the computation pattern, which is determined by how the code is written. Therefore, partition camping and the effects are considered in the parallelization procedure.

3.2 Shared Memory Layout

Shared memory is divided into equal-sized banks for GPUs. For the T10 GPU, there are 16 banks as depicted in Figure 3.4. Each bank consists of 256 rows, 4 bytes wide, and all banks can be accessed simultaneously. Bank conflicts occur at the thread level. If all

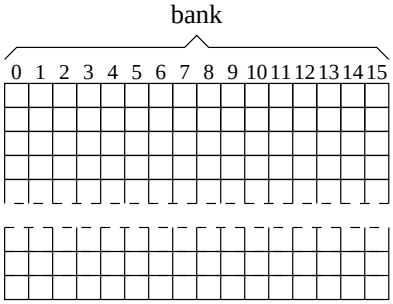


Figure 3.4: Shared memory layout in the T10 GPU. Memory is divided into 16 equal-sized partitions. Each column is 4 bytes wide.

threads within a HW access different banks, as illustrated in Figure 3.5, no bank conflicts occur and only one access is necessary.

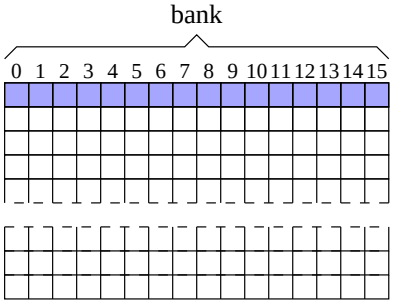


Figure 3.5: Threads within a HW accessing 16 shared memory banks. No bank conflicts occur.

3.2.1 Bank Conflicts

If two or more addresses of a memory access to shared memory fall into the same memory bank, a bank conflict occurs. Since memory accesses are issued at the HW level, threads within a HW accessing differing addresses in the same bank cause bank conflicts and the accesses are serialized. In Figure 3.6, all threads within a HW access different rows of shared memory but the same column. Since the column resides in the same shared memory bank,

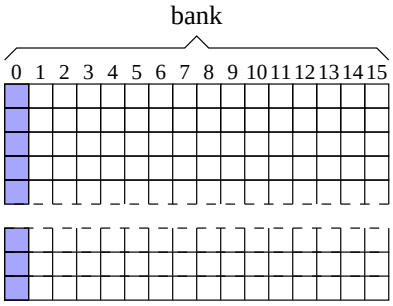


Figure 3.6: Threads within a HW accessing one shared memory bank. 16 bank conflicts occur.

16 bank conflicts occur. Therefore, 16 accesses to shared memory are issued to service the HW. However, the T10 GPU supports shared memory broadcasting. Therefore, if threads within a HW access the same address in shared memory, no bank conflicts occur.

Effects

The portion of the kernel utilized to test the effects of bank conflicts is depicted in Listing 3.2.

```
1  __shared__ float As[dBlkx][dBlky];

2  int offset  = blockIdx.x % 8 * 64;
3  int soffset = threadIdx.x % Banks;

4  for(j = 0; j < v; j++) {
5    As[threadIdx.y][threadIdx.x] = d_A[j * 512 + offset + threadIdx.y];
6    temp += As[threadIdx.x][soffset];
7  }
```

Listing 3.2: Code to test the effects of bank conflicts.

The number of banks each HW accesses is defined by $Banks$. In Line 3, $soffset$ is calculated to specify which column of shared memory is accessed and therefore, the number of bank conflicts is fixed. *blockIdx.x* and *threadIdx.x* are the indices of each block and thread in the x-dimension, respectively.

Table 3.2 shows the effect on computation time due to bank conflicts. Similar to the

	number of banks				
v	16	8	4	2	1
512	0.5	0.5	0.6	0.9	1.5
1024	0.9	1.0	1.3	1.8	2.9
2048	1.7	1.9	2.5	3.6	5.8
4096	3.4	3.8	5.0	7.2	11.5
8192	6.6	7.6	9.9	14.3	23.0
16384	12.9	15.1	19.6	28.5	46.0

(a) Computation time (ms).

	number of banks				
v	16	8	4	2	1
512	65.1	58.6	45.8	31.8	19.9
1024	66.6	60.4	46.5	32.2	20.1
2048	67.7	61.4	47.1	32.6	20.2
4096	69.1	61.7	47.3	32.7	20.3
8192	71.2	61.9	47.6	32.8	20.4
16384	72.4	62.1	47.7	32.9	20.4

(b) Effective bandwidth (GB/s).

Table 3.2: Measured results of executing Listing 3.2 which demonstrates the effect of bank conflicts on the T10 GPU. v is the number of values summed by each thread.

effects of partition camping in global memory, shared memory bank conflicts significantly affect computation time. As the number of banks increases, the number of bank conflicts decreases as does the computation time. Although the change in computation time is not linearly proportional to the number of banks utilized, the effects of bank conflicts are clearly illustrated. Therefore, bank conflicts and the effects are considered in the parallelization procedure.

3.3 Execution Metrics

In this section, execution metrics are formulated to represent computational behavior of the GPU to assist in modeling. Figure 3.7 shows the necessity for execution metrics, as input parameters, the dimensions of the grid and blocks, do not accurately model computational behavior. Initially, all measured computation times of MM utilizing varying input parameters are grouped by n. The average percent difference between the maximum and minimum time within a group is shown. Grouping by n yields an average difference of

163% between the maximum and minimum time of a group. Next, all measured computation times utilizing varying input parameters are grouped by n and $dBlk.x$. The average percentage difference is 109 between the maximum and minimum of a group. The order of the groupings is determined by the input parameter which yields the smallest average percent difference between the maximum and minimum time of a group. Continuing until 3 of the 4 input parameters are exhausted for grouping yields an average percent difference of 28. Therefore, modeling the computational behavior of the GPU is inaccurate utilizing input parameters. Because of this, execution metrics are formulated as functions of input parameters to accurately represent computational behavior.

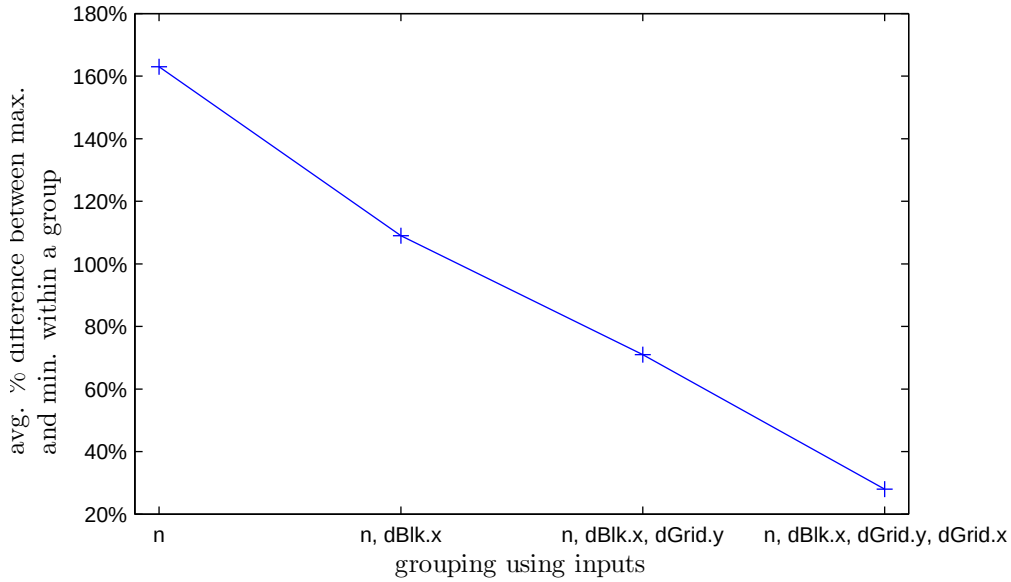


Figure 3.7: Modeling MM with input parameters: average percent difference between the maximum and minimum computation time (ms) of a group defined by n and a subset of the input parameters.

Four input parameters, $dGrd.x$, $dGrd.y$, $dBlk.x$ and $dBlk.y$, and the size of each matrix, n, are used to formulate the execution metrics. Input parameters and n are assumed to be powers of two in this work. The product of the four input parameters yields the total number

of threads assigned for execution on the GPU[2] as

$$\text{Thds}_{GPU}^{total} = dGrd.x \times dGrd.y \times dBlk.x \times dBlk.y. \tag{3.1}$$

The execution metrics can be formulated for any matrix-based computation. However, in this work, the focus of the execution metrics pertains to Mv, MM, and convolution.

3.3.1 Global Memory Accesses

$\text{Thds}_{GPU}^{total}$ is used to formulate an execution metric for representing the number of global memory accesses. The number of memory accesses significantly affects the computation time for matrix-based computations since the time is largely dependent on memory access latency.

In general, for matrix-based computations, all matrices reside in global memory and portions of matrices may reside in shared memory. For MM, there are $2n^3$ values read from global memory, although consecutive reads from threads in the same block can be combined. The number of writes to global memory for MM is n^2 and consecutive writes from threads in the same block can be combined. Therefore, the amount of time for writing to global memory for MM is significantly less than the amount of time for reading from global memory. This is similar for other matrix-based computations such as Mv and 2D convolution. Therefore, the number of writes to global memory is ignored in this work.

Memory accesses are issued at the HW level and the GPU combines consecutive thread accesses within a HW into one access, known as coalescing. The memory access is issued as either a 32-, 64- or 128-byte memory transaction. Therefore, the number of reads is the sum of $32B$, $64B$ and $128B$ transactions.

$$\text{Gld} = \text{Gld}_{32B} + \text{Gld}_{64B} + \text{Gld}_{128B}.$$

[2]The notation used in Equation (3.1) is used throughout this work. The unit being defined, threads (Thds) or blocks (Blks), is specified first. The superscript represents the type of unit being defined, total (total), active (active) or fragmented (frag.). The subscript represents the execution unit being defined, GPU (GPU) or SM (SM).

Initially, each transaction is considered a $128B$ access by the GPU's memory controller and is reduced if possible. If the threads in a HW access between 9 and 16 consecutive and aligned memory locations of float or integer value, only one $64B$ memory transaction is issued. If the HW accesses less than 9 consecutive memory locations of float or integer value, one $32B$ memory transaction is issued. If the HW accesses memory locations which are not consecutive and $128B$ aligned, multiple $32B$ and/or $64B$ memory transactions are issued until all threads in the HW have been serviced. Since each HW is a group of 16 threads, the number of HWs that execute on the GPU is defined as

$$\text{HWs}^{\#} = \frac{\text{Thds}_{GPU}^{total}}{16}.$$

The equation assumes there are at least 16 threads in a block.

From the input parameters, and considering coalescing, the number of $32B$ and $64B$ reads for matrix-based computations can be formulated[3].

For a naïve implementation of Mv utilizing only global memory, $thread_j$, to compute c_j where $\mathbf{c}=\mathbf{A}\times\mathbf{b}$, computes $row_j \cdot \mathbf{b}$ where row_j is one row of $\mathbf{A}$. Therefore, the neighboring thread, $thread_{j+1}$, computes $row_{j+1} \cdot \mathbf{b}$. If $dBlk.x \geq 16$, each HW accesses 16 values of $\mathbf{A}$ and 1 value of $\mathbf{b}$. Since 1 value of $\mathbf{b}$ is accessed, a $32B$ transaction is issued. The 16 values of $\mathbf{A}$ reside in differing rows of $\mathbf{A}$, and therefore the accesses are uncoalesced and require 16 $32B$ transactions. This is repeated for each HW n times. Therefore, the number of $32B$ transactions for this implementation of Mv is

$$\text{Gld}_{32B} = \begin{cases} \dfrac{(dBlk.x+1)n^2}{dBlk.x} & \text{if } dBlk.x \leq 8 \\ \dfrac{17n^2}{16} & \text{if } dBlk.x \geq 16. \end{cases} \tag{3.2}$$

This equation assumes only global memory is utilized for Mv. The equation is modified for varying memories as shown in Section 4.1.

[3] $128B$ reads do not exist in matrix-based computations when using values of type int or float.

For a naïve implementation of MM utilizing only global memory, to compute C_{ij} where $\mathbf{C}$=$\mathbf{A}$×$\mathbf{B}$, $thread_{ij}$ accesses row_i and col_j. If $dBlk.x > 1$, the neighboring thread, $thread_{i(j+1)}$, accesses row_i and col_{j+1}. Since both threads are in the same HW, then if $dBlk.x \geq 16$, n $32B$ transactions are issued to read row_i and n $64B$ transactions to read $col_j ... col_{j+15}$. However, if $dBlk.x \leq 8$, then only $32B$ transactions are issued. Therefore, the total number of reads performed to global memory is defined as

$$\text{Gld}_{32B} = \begin{cases} \dfrac{n^3}{dBlk.x} + \dfrac{n^3}{16} & \text{if } dBlk.x \leq 8 \\ \dfrac{n^3}{16} & \text{if } dBlk.x \geq 16 \end{cases} \tag{3.3}$$

$$\text{Gld}_{64B} = \frac{n^3}{16} \quad \text{if } dBlk.x \geq 16. \tag{3.4}$$

These equations assume only global memory is utilized for MM. The equations are modified for varying memories as shown in Section 4.1. Assuming $dBlk.x \geq 16$, half of the reads are $32B$ transactions and half are $64B$. Since memory access latency to constant and shared memory is much shorter than global, the number of accesses performed to constant and shared memory is ignored in this work as it has an insignificant impact on computation time.

For a naïve implementation of 2D convolution[4], $thread_{ij}$ computes C_{ij} where $\mathbf{C} = \mathbf{A} * \mathbf{B}$. In image processing, $\mathbf{A}$ is an image and $\mathbf{B}$ is a *filter*. Therefore, $\mathbf{C}$ is the result of applying a filter to an image and C_{ij} is one pixel of the resulting image. In this work, all filters, $\mathbf{B}$, are assumed to be square. Therefore, the size, height or width, of a filter is represented by FS. For a naïve implementation of convolution utilizing only global memory, to compute C_{ij}, $thread_{ij}$ accesses FS^2 values of $\mathbf{A}$ and $\mathbf{B}$. Since each thread within a HW accesses the same value of $\mathbf{B}$ in each iteration of computation, the number of $32B$ transactions for accessing $\mathbf{B}$ is defined only by FS. If $dBlk.x \geq 16$, neighboring threads within a HW access neighboring values of $\mathbf{A}$ and therefore the accesses are coalesced into one $64B$ transaction. However, if

[4]From this point forward, convolution refers to 2D convolution of two matrices where $\mathbf{C} = \mathbf{A} * \mathbf{B}$.

$dBlk.x \leq 8$, no $64B$ transactions occur and the number of $32B$ transactions is dependent on FS and $dBlk.x$. Therefore, the number of global memory accesses is approximately[5]

$$\text{Gld}_{32B} = \begin{cases} \frac{n^2FS^2}{16}\left(1 + \frac{16}{dBlk.x}\right) & \text{if } dBlk.x \leq 8 \\ \frac{n^2FS^2}{16} & \text{if } dBlk.x \geq 16 \end{cases} \tag{3.5}$$

$$\text{Gld}_{64B} = \frac{n^2FS^2}{16} \quad \text{if } dBlk.x \geq 16. \tag{3.6}$$

These equations assume only global memory is utilized for convolution. The equations are modified for varying memories as shown in Section 5.1.1. The number of accesses performed to shared memory is ignored in this work as it has an insignificant impact on computation time. In addition to formulating the amount of reads to global memory, it is necessary to formulate the amount of threads executing in parallel on the GPU.

3.3.2 Active Threads

The number of threads executing at a given time on a GPU, or the number of active threads, is defined by $\text{Threads}_{GPU}^{active}$. To formulate the number of active threads, it is necessary to formulate the number of active blocks per SM, $\text{Blks}_{SM}^{active}$. $\text{Blks}_{SM}^{active}$ is dependent on the number of registers used per block (RegsPerBlk), amount of shared memory used per block (SMemPerBlk), maximum amount of blocks allowable, and maximum amount of threads allowable. The number of registers used per thread is dependent on the compiler and determined after compilation. The amount of shared memory used per block is defined by the kernel and determined after compilation.

The maximum number of blocks and threads that can be active varies depending on the GPU. The T10 GPU consists of 30 SMs and each SM can have a maximum of 8 blocks or 1024 threads. Assuming there are more blocks assigned for execution than SMs, $dGrd.x \times dGrd.y >$

[5]Due to boundary checking, not all reads are issued.

30, $\text{Blks}_{SM}^{active}$ is formulated as

$$\text{Blks}_{SM}^{active} = \min\left(\left\lfloor \frac{16384}{\text{RegsPerBlk}} \right\rfloor, \left\lfloor \frac{16384}{\text{SMemPerBlk}} \right\rfloor, 8, \frac{1024}{dBlk.x \times dBlk.y}\right). \tag{3.7}$$

Therefore, from Equation (3.7), the number of active blocks on the GPU at a given time is

$$\text{Blks}_{GPU}^{active} = \min\left(30 \times \text{Blks}_{SM}^{active}, dGrd.x \times dGrd.y\right). \tag{3.8}$$

The number of active threads on the GPU is formulated as the product of Equation (3.8) and the number of threads per block. Therefore,

$$\text{Thds}_{GPU}^{active} = \text{Blks}_{GPU}^{active} \times dBlk.x \times dBlk.y. \tag{3.9}$$

3.3.3 Fragmented Threads

If the number of active threads on a GPU is not evenly divisible by the number of total threads ($\text{Thds}_{GPU}^{active} \bmod \text{Thds}_{GPU}^{total} \neq 0$)[6], fragmentation occurs. Fragmented threads are threads which are not executed with the same amount of parallelization as active threads. From (3.8) and (3.9), two execution metrics are formulated to represent the number of fragmented blocks and threads: $\text{Blks}_{GPU}^{frag.}$ and $\text{Thds}_{GPU}^{frag.}$, respectively. Therefore, the number of fragmented blocks on the GPU is

$$\text{Blks}_{GPU}^{frag.} = (dGrd.x \times dGrd.y) \bmod \text{Blks}_{GPU}^{active} \tag{3.10}$$

and the number of fragmented threads on the GPU is

$$\text{Thds}_{GPU}^{frag.} = \text{Blks}_{GPU}^{frag.} \times dBlk.x \times dBlk.y. \tag{3.11}$$

[6] $x \bmod y$ denotes the remainder of $\frac{x}{y}$.

3.3.4 Global Memory Partitions

In addition to formulating execution metrics to represent the number of global memory accesses, active and fragmented threads, two execution metrics are formulated to represent the layout of global memory. The global memory on a 200-series GPU architecture is divided into 8 partitions where each partition is 256B wide. To represent the number of global memory partitions accessed per block, an execution metric, PartsPerBlk, is formulated as

$$\text{PartsPerBlk} = \frac{dBlk.x}{64} \tag{3.12}$$

Since HWs are formed in row-major order, the x-dimension of the block is used. Since values of type float are 4 bytes and each global memory partition is 256 bytes wide, 64 values are stored per row for each partition. Therefore, Equation (3.12) represents the number of global memory partitions accessed per block.

From PartsPerBlk, the number of global memory partitions accessed by the active blocks on the GPU, PartsPerGPU, is formulated as

$$\text{PartsPerGPU} = \begin{cases} \min\left(\text{Blks}_{GPU}^{active} \times \text{PartsPerBlk}, 8\right) & \text{if } \text{Blks}_{GPU}^{active} < dGrd.x \\ \min\left(dGrd.x \times \text{PartsPerBlk}, 8\right) & \text{if } \text{Blks}_{GPU}^{active} \geq dGrd.x. \end{cases} \tag{3.13}$$

PartsPerGPU is dependent on the number of global memory partitions accessed per block and $\text{Blks}_{GPU}^{active}$. The equation defines a maximum of 8 partitions being accessed since there are 8 partitions of global memory in the T10 GPU.

3.4 Communication Time

Estimating the communication time between the CPU and GPU is necessary for a parallelization procedure to minimize execution time. For small matrix-based computations, it is possible the communication time is higher than the CPU computation time. Therefore, it is necessary to accurately model the communication time between the CPU and GPU.

In addition, communication and computation time estimates are necessary to determine the partitioning of computation between the CPU and GPU.

Communication time, $T_{comm.}$, is modeled linearly with a setup time, T_{su}, and a maximum transfer rate, T_x. Therefore,

$$T_{comm.} = \frac{b}{T_x} + T_{su} \tag{3.14}$$

where b is the number of bytes being transferred. Research shows T_x is dependent on the type of transfer (CPU to GPU or GPU to CPU). In addition, analysis of measured data suggests T_{su} is a function of b. Therefore,

$$T_{su} = c_0 b + c_1 \tag{3.15}$$

where c_0 is the time per byte to set up a transfer and c_1 is a constant. Both are determined through curve-fitting measured data. For Mv, MM, and convolution, there are two CPU-GPU transfers and one GPU-CPU.

3.4.1 CPU-GPU

From NVIDIA, the advertised transfer rate between the CPU and GPU is $4GB/s$. However, the maximum transfer rate, T_x, measured for CPU to GPU communication is $3.5GB/s$. For transfers smaller than $256KB$, measured data shows c_0 is approximately $525ms/GB$ and c_1 is approximately $0.014ms$. For transfers between $256KB$ and $1MB$, data shows c_0 is approximately $3.2ms/GB$ and c_1 is approximately $0.061ms$. Lastly, for transfers of size $1MB$ and larger, data shows c_0 is approximately $3.2ms/GB$ and c_1 is approximately $0.19ms$. The varying values for c_0 and c_1 are presumably due to CPU cache size and memory layout as well as the layout of GPU memory. In this work, $n \geq 512$, as discussed in Section 4.1, and therefore, all CPU to GPU matrix transfers are larger than $1MB$. Figure 3.8 depicts the measured and estimated time for two CPU to GPU transfers. Two transfers are measured and estimated since the GPU requires a matrix and vector for Mv and two matrices for MM

and convolution. The nonuniform behavior illustrated in the figure is because c_0 and c_1 vary with the size of data.

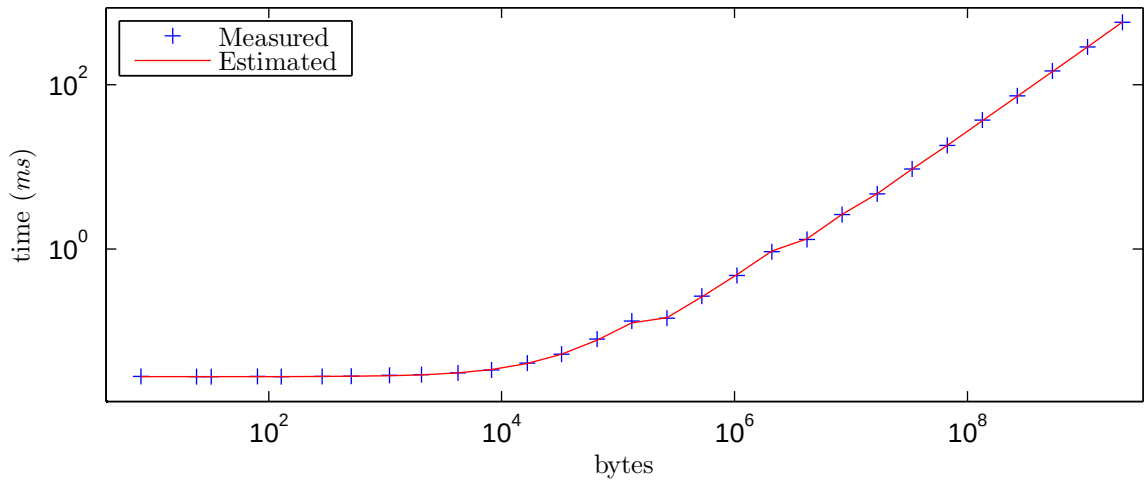


Figure 3.8: Comparison of measured and estimated CPU to GPU communication time (ms) on the T10 GPU.

For Mv, $4n^2 + 4n$ bytes are transferred for the matrix, **A**, and vector, **b**, since values of type float are utilized. For MM, $8n^2$ bytes are transferred for the two matrices, **A** and **B**. For convolution, $4n^2 + 4FS^2$ bytes are transferred for the two matrices, **A** and **B**. The maximum and average percent error between the measured and estimated time are 5.0% and 1.0%, respectively.

3.4.2 GPU-CPU

The maximum transfer rate measured for GPU to CPU communication is $3.1GB/s$. For transfers smaller than $1MB$, measured data shows c_0 is approximately $210ms/GB$ and c_1 is approximately $0.02ms$. For transfers larger than $1MB$, data shows c_0 is approximately $6.9ms/GB$ and c_1 is approximately $0.21ms$. The varying values for c_0 and c_1 are presumably due to CPU cache size and memory layout as well as the layout of GPU memory. Figure 3.9 depicts the measured and estimated time for one GPU to CPU transfer.

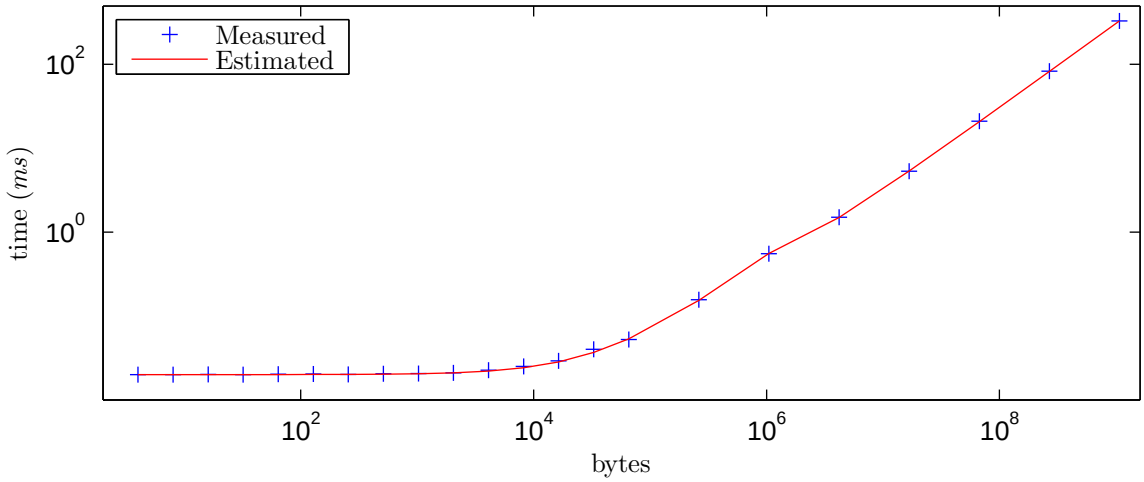


Figure 3.9: Comparison of measured and estimated GPU to CPU communication time (ms) on the T10 GPU.

For Mv, one GPU to CPU transfer of $4n$ bytes is necessary for the vector, $\mathbf{c}$. For MM and convolution, one transfer of $4n^2$ bytes is necessary for the matrix, $\mathbf{C}$. The maximum and average percent error between the measured and estimated time are 8.9% and 1.4%, respectively.

3.5 Computation Time

For a parallelization procedure to minimize execution time of matrix-based computations on a GPU, it is necessary to model CPU and GPU computation time. Computation time in this work refers to time for memory accesses and computation. Determining which computations are performed by the CPU and GPU require accurate computation and communication time estimates.

The execution time for matrix-based computations is defined as the sum of the communication and computation times and therefore,

$$T_{exec.} = T_{comm.} + T_{comp.}$$

The computation time, $T_{comp.}$, is the sum of the CPU and GPU computation times, $T_{comp.}^{CPU}$ and $T_{comp.}^{GPU}$, respectively, as shown in Equation (3.16):

$$T_{comp.} = T_{comp.}^{CPU} + T_{comp.}^{GPU}. \tag{3.16}$$

3.5.1 CPU

Due to CPU cache memories, branch prediction mechanisms, hyper-threading and other instruction and data latency-hiding optimizations, there is significant research in modeling CPU computation time. Since the goal of this work is to minimize execution time of matrix-based computations on a GPU, an estimate of CPU computation time is obtained through analysis of measured data. In this work, the ATLAS [60] package is utilized for matrix-based computations executing on the CPU. ATLAS is an ongoing research project which provides C interfaces to an optimized BLAS implementation as well as some routines from LAPACK [18]. ATLAS is currently used in MAPLE, MATLAB, Octave and planned for use in Mathematica.

Computation time for Mv and MM is measured utilizing the ATLAS interface to the sgemv (Mv) and sgemm (MM) BLAS functions, respectively. Bandwidth is utilized to estimate CPU computation time and is calculated as the number of bytes accessed for each computation divided by the computation time.

For Mv, $2n^2$ float values are accessed and therefore, effective bandwidth is $\frac{8n^2 bytes}{T_{comp.}^{CPU}}$. From measured data, the maximum bandwidth is dependent on n and measured between $2GB/s$ and $8GB/s$. Therefore, for Mv, the bandwidth is estimated as $\min\left(\max\left(\frac{16384}{n}, 2\right), 8\right)$. Therefore,

$$T_{comp.}^{CPU} = \frac{\frac{8n^2}{1024^3}}{\min\left(\max\left(\frac{16384}{n}, 2\right), 8\right)} seconds$$

for the optimized BLAS implementation of Mv. Figure 3.10 depicts the measured and estimated time for the optimized BLAS implementation of Mv on the CPU.

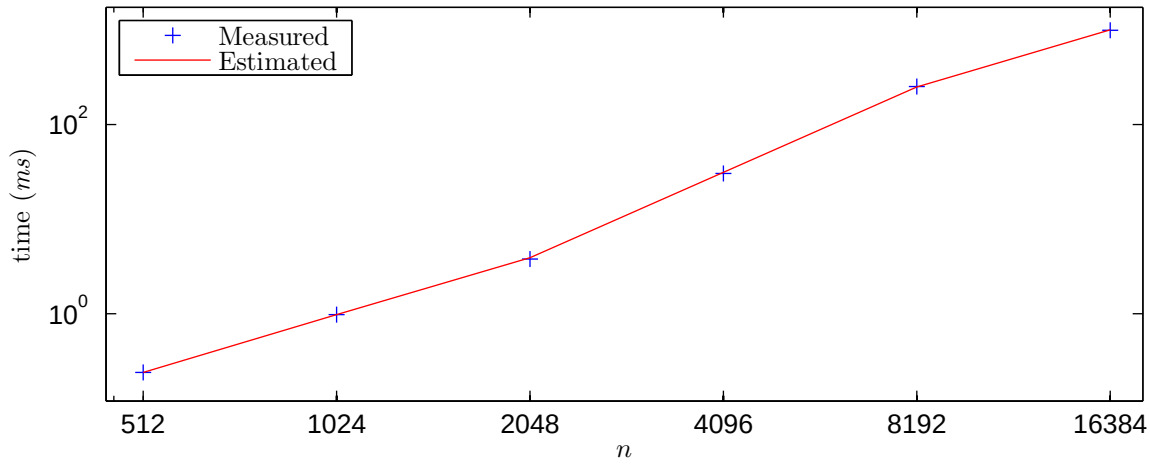


Figure 3.10: Comparison of measured and estimated computation time (ms) for the optimized BLAS implementation of Mv on the CPU.

The maximum and average percent error between the measured and estimated time are 3.0% and 1.4%, respectively.

For MM, $2n^3$ float values are accessed and therefore, effective bandwidth is $\frac{8n^3 bytes}{T^{CPU}_{comp.}}$. From measured data, the maximum bandwidth varies between $50GB/s$ and $54GB/s$. From the median bandwidth $(51GB/s)$ measured,

$$T^{CPU}_{comp.} = \frac{\frac{8n^3}{1024^3}}{51} seconds$$

for the optimized BLAS implementation of MM. Figure 3.11 depicts the measured and estimated time for the optimized BLAS implementation of MM on the CPU.

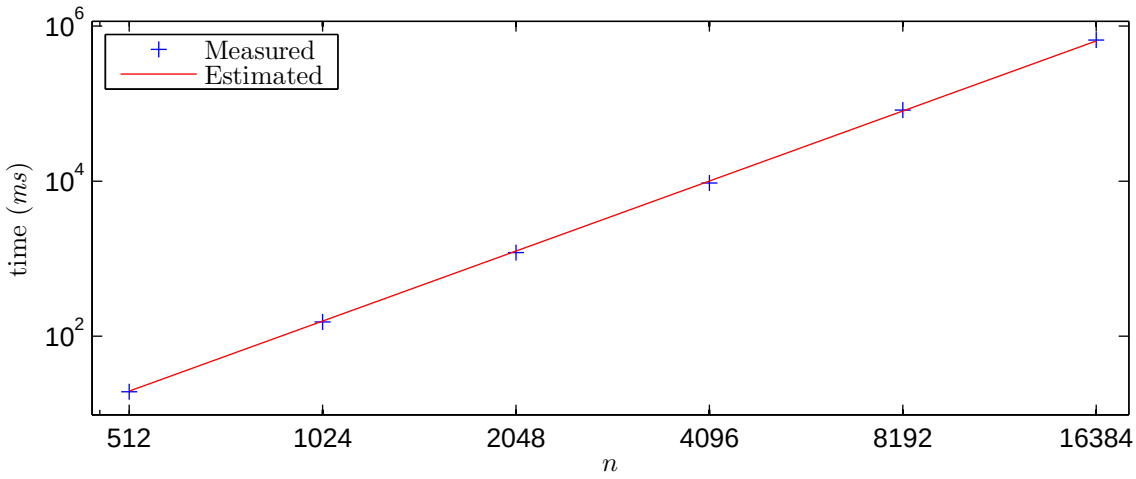


Figure 3.11: Comparison of measured and estimated computation time (ms) for the optimized BLAS implementation of MM on the CPU.

The maximum and average percent error between the measured and estimated time are 5.6% and 3.2%, respectively.

Since BLAS routines do not include convolution, time for a non-optimized C implementation is measured. For convolution, $2n^2FS^2$ float values are accessed. Therefore, effective bandwidth is calculated as $\frac{8n^2FS^2bytes}{T^{CPU}_{comp.}}$. From measured data, maximum bandwidth is estimated as $(\frac{3}{\sqrt{FS}} - \frac{n}{20000} + 2.7)GB/s$ and therefore,

$$T^{CPU}_{comp.} = \frac{\frac{8n^2FS^2}{1024^3}}{\left(\frac{3}{\sqrt{FS}} - \frac{n}{20000} + 2.7\right)} seconds$$

for the non-optimized C implementation of convolution. Figure 3.12 depicts the measured and estimated time for the non-optimized C implementation of convolution with a filter size of 3 on the CPU.

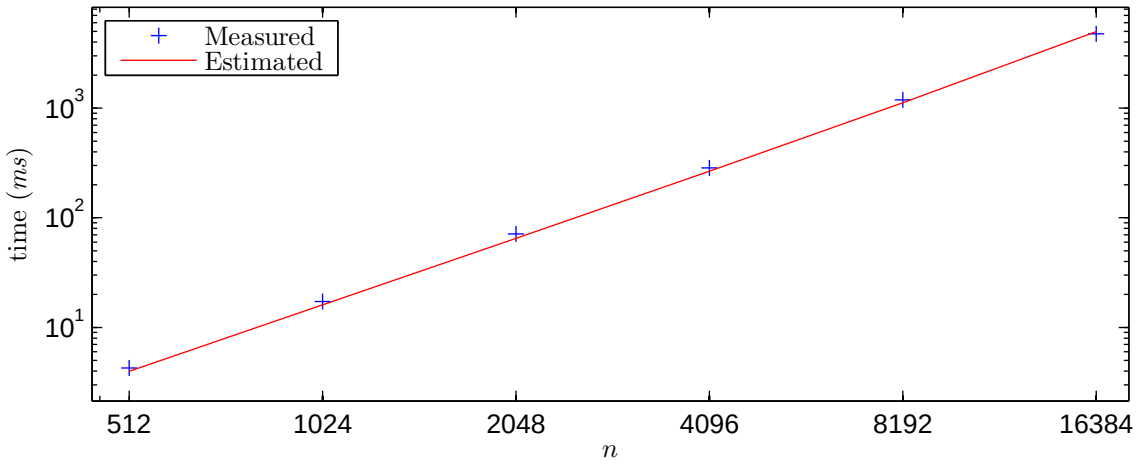


Figure 3.12: Comparison of measured and estimated computation time (ms) for the non-optimized C implementation of convolution on the CPU. FS=3.

The maximum and average percent error between the measured and estimated time are 9.6% and 7.0%, respectively. Figure 3.13 depicts the time for convolution with a filter size of 63.

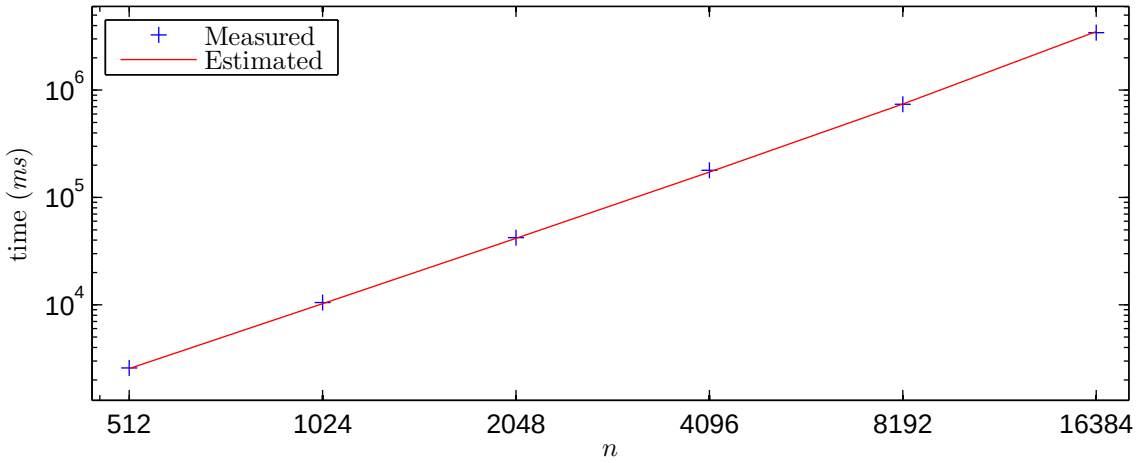


Figure 3.13: Comparison of measured and estimated computation time (ms) for the non-optimized C implementation of convolution on the CPU. FS=63.

The maximum and average percent error between the measured and estimated time are 4.0% and 2.1%, respectively. Figure 3.14 depicts the time for convolution with a filter size of 513.

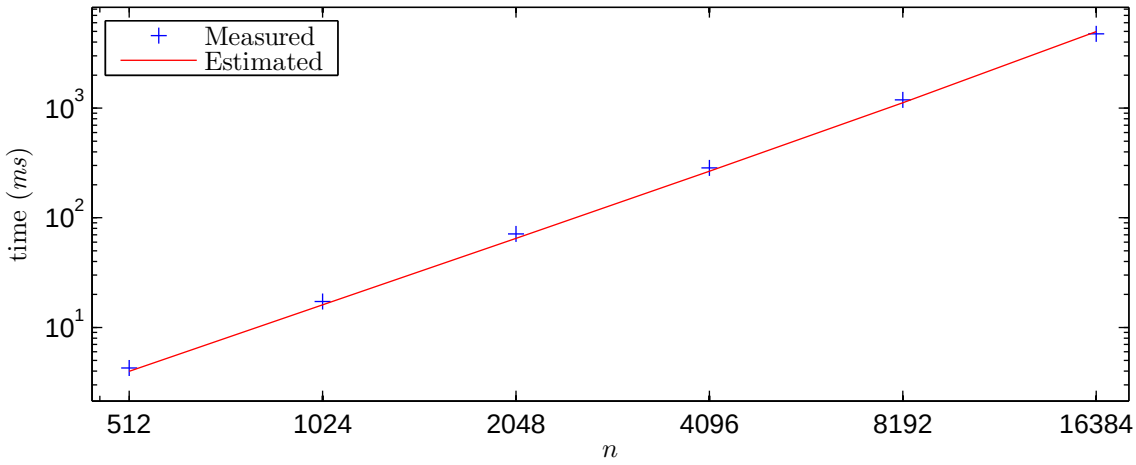


Figure 3.14: Comparison of measured and estimated computation time (ms) for the non-optimized C implementation of convolution on the CPU. FS=513.

The maximum and average percent error between the measured and estimated time are 15.1% and 8.6%, respectively.

Therefore, for the implementations of Mv, MM, and convolution, the worst-case percent error between the measured and estimated CPU computation time is 15.1%.

3.5.2 GPU

Similar to CPU computation time, GPU computation time, $T_{comp.}^{GPU}$, is dependent on the implementation of the matrix-based computation. Since the parallelization procedure minimizes execution time of matrix-based computations executing on a GPU, it is assumed the procedure yields the minimum computation time. Therefore, the time is estimated assuming the theoretical bandwidth to global memory.

The theoretical bandwidth to global memory is calculated as the product of the memory clock rate and data bus width. The T10 GPU has a $792MHz$ memory clock rate and a 512-bit data bus. Since the memory is double-data rate, the theoretical bandwidth to global memory is approximately $94GB/s$. According to NVIDIA [61], 70-80% of theoretical bandwidth "is very good" for memory-bound computations.

Since GPU computation time is estimated assuming theoretical bandwidth, it is necessary to determine the number of bytes accessed from global memory for each matrix-based computation. The number of bytes accessed is dependent on the memory utilized and the size of each memory transaction. Therefore, the GPU computation time is estimated in the last step of the parallelization procedure after the type of memory is determined and the number and size of memory transactions are calculated.

Chapter 4

Parallelization Procedure

This chapter describes the development of the parallelization procedure and its impact on minimizing execution time for matrix-based computations on the GPU. The procedure is developed to be general in that it applies to any matrix-based computation. Each step of the procedure is developed through research, testing, and analysis of measured data for Mv, MM, or both. The procedure is applied to convolution and the conjugate gradient method in Chapter 5 for verification. The procedure considers the placement of data in GPU memory, computation patterns of the GPU, access patterns to GPU memory, fine-tuning GPU code, input parameters to the GPU, and computation partitioning between the CPU and GPU. Each aforementioned item is discussed individually in Sections 4.1 - 4.6.

In each section of this chapter, one step of the parallelization procedure is developed through analysis of Mv, MM, or both. Results are included at the end of each section to demonstrate the effectiveness of each step. Each section includes the kernel for each matrix-based computation to illustrate the application of each step of the procedure. All results in this chapter were gathered using the NVIDIA Tesla S1070, which includes a pair of T10 GPUs connected to a quad-core $2.26GHz$ Intel Xeon E5520 CPU at Alabama Supercomputer Authority [62]. The height or width of a square matrix, n, is varied in the results from 512 to 16384 as defined by Equation (4.1) in Section 4.1. $dBlk.x$ is varied in the results from 8 to 512, the maximum allowable value for the T10.

The first step is to determine the optimal placement of data in GPU memory. Therefore, Section 4.1 is a discussion of the three types of memory in the GPU and the appropriateness of each memory for various computations. Utilizing various memories on the GPU other than global can reduce the number of accesses to global memory and therefore, reduce GPU

computation time. This is the first step of the procedure, as it is the most common optimization to GPU computations to minimize time. In addition, other steps of the procedure such as deriving the optimized computation and access patterns and optimal input parameters are dependent on which memories are utilized and therefore, this is the first step. An algorithm is presented to determine the optimal placement of data in GPU memory to minimize GPU computation time.

Step 2 in Section 4.2 is a discussion of various computation patterns determined by the kernel. The kernel defines which result(s) each thread computes and therefore, an examination is performed of various computation patterns of Mv and MM. From measured data, it is shown that for Mv, if $thread_j$ computes one value of $\mathbf{c}$, c_j, where $\mathbf{c}$=$\mathbf{Ab}$, the computation time is bound by the number of threads and data size. Likewise, for MM, if $thread_{ij}$ computes one value of $\mathbf{C}$, C_{ij}, where $\mathbf{C}$=$\mathbf{AB}$, the computation time is bound by computation and memory accesses. Increasing the overlap of computation and memory accesses reduces the computation time. Therefore, the optimized computation pattern which reduces GPU computation time is derived. Since the access pattern is dependent on the computation pattern, this step of the procedure is included before deriving the optimized access pattern.

The optimized access pattern is dependent on the placement of data in GPU memory and the computation pattern. Therefore, Step 3 is included to derive the optimized access pattern. Access patterns to GPU memory are discussed in Section 4.3 and the optimized access pattern is derived to minimize partition camping and eliminate bank conflicts. This step is developed through analysis of Mv and MM. For Mv, bank conflicts occur when accessing shared memory and affect computation time as shown in Section 3.2. For Mv and MM, all HWs begin accessing matrix $\mathbf{A}$ from the same partition of global memory; thus partition camping occurs. From Section 3.1, partition camping affects computation time. Therefore, the optimized access pattern is derived to minimize partition camping and eliminate bank conflicts.

After the optimal placement of data is determined and the optimized computation and access patterns are derived, fine-tuning of the kernels is performed to reduce the amount of computation and resource allocation, as shown in Section 4.4. This step, Step 4, is developed from analysis of MM, as the optimized computation pattern for MM utilizes many registers. Due to the register usage, this step is included in the procedure to reduce the register requirement of the kernels such that more threads execute in parallel, and thus reduce the execution time. Included in this step is the minimization of index calculations. Developed from analysis of measured data for MM, reducing index calculations, particularly in inner loops, reduces the time necessary to issue global memory transactions thus reducing the computation time.

After fine-tuning the kernels, analysis of Mv and MM yields a necessity of optimal input parameters. Therefore, Step 5 of the procedure is included to derive the optimal input parameters, which are input parameters that yield the minimum GPU computation time. From measured data, computation time varies significantly depending on the number of blocks and size of each block. Therefore, from the formulation of execution metrics in Section 3.3, optimal input parameters are derived in Section 4.5. This step of the procedure is performed after Steps 1-4 since each of the aforementioned steps affects the execution metrics. Execution metrics are affected by each previous step since they are dependent on such things as the allocation of registers and shared memory.

The last step, Step 6, of the parallelization procedure to minimize execution time is to determine the partitioning of computation between the CPU and GPU as shown in Section 4.6. From measured data for Mv, it is shown that the communication time may exceed the CPU computation time. Therefore, optimally determining which computations are performed by the CPU and GPU is included in the procedure. Optimally determining the partitioning of computation requires accurate estimates of communication and computation

times as discussed in Sections 3.4 and 3.5, respectively. An algorithm utilizing the communication and computation times is included for determining the optimal partitioning of computation.

Therefore, the parallelization procedure to minimize execution time of matrix-based computations utilizing the GPU consists of

1. determining the optimal placement of data in GPU memory,
2. deriving the optimized computation pattern which reduces computation time,
3. deriving the optimized access pattern to data in GPU memory,
4. fine-tuning GPU code to reduce computation time,
5. deriving optimal input parameters which yield the minimum computation time for the GPU, and
6. optimally partitioning computation between the CPU and GPU.

4.1 Placement of Data

To minimize GPU computation time, and thus the execution time, it is necessary to consider which GPU memories are utilized. Three types of memory exist in the T10 GPU: global, constant and shared.

Global memory is the only memory available for both CPU to GPU and GPU to CPU transfers. It is the main memory of the device due to its read/write capability, large size, accessibility by both the CPU and GPU and its accessibility by all executing threads. It is the largest of the available memories, at $4GB$ in the T10. However, it is the slowest memory with a peak theoretical bandwidth of $94GB/s$. Since some global memory is reserved by the system, not all $4GB$ are accessible. Therefore, using powers of two for n, the height or width of a square matrix, the maximum value of n is 16384. The minimum value of n, in

this work, is determined such that one row of a matrix spans at least one global memory partition. Therefore,

$$512 \leq n \leq 16384. \tag{4.1}$$

Constant memory is available for CPU to GPU transfers and is used for storing constant data; it is a read-only memory from the GPU's perspective. Similar to shared memory, it is much faster than global memory and can reduce computation time in applications by reducing the number of global memory accesses. The T10 GPU includes $64KB$ of constant memory that is accessible by all threads. However, since it is read-only by the GPU, it can only be used for transferring data from the CPU to GPU. In computations such as MM, when $n \geq 512$, utilizing constant memory is not possible as matrices exceed the size. However, for Mv, constant memory can be utilized to store the vector, and thus reduce the number of global memory reads as well as the partition camping effect.

Shared memory is read/write for the GPU but not accessible to the CPU. Therefore, to utilize shared memory, the CPU must transfer data to the GPU's global memory and the GPU must transfer the data from global memory to shared. Shared memory is much faster than global and is widely used as temporary storage, or scratchpad space, for threads. Utilizing shared memory for computations such as MM can greatly reduce the computation time as many of the accesses to global memory can be reduced. Data is transferred from the CPU to the GPU's global memory and then the GPU copies portions of the data that are reused to shared memory thus greatly reducing the number of global memory accesses. The T10 GPU includes $16KB$ of shared memory per SM, for a total of $480KB$ of shared memory. However, shared memory is allocated per SM, so threads executing on one SM cannot read or write to shared memory of another SM. Even if it were possible, if $n \geq 512$, the limited data size would not accommodate one matrix. However, significant reductions in computation time can be realized in computations which reuse data by partitioning the reused data into smaller blocks and then utilizing shared memory. For MM, data is reused and utilizing shared memory reduces the demand on the global memory bus thus reducing

computation time. Shared memory can also be utilized to coalesce accesses to global memory for data which are not reused such as the matrix in Mv, and therefore, reduce computation time.

From the summaries of each memory, Listing 4.1 is developed to determine the placement of data.

```
1  if(data reused)
2       if(sizeof(data reused) ≤ sizeof(cmem))
3            utilize cmem for data reused
4       if(sizeof(data reused) > sizeof(cmem))
5            partition data reused into blocks ≤ sizeof(smem)
6            utilize smem for data reused
7  else
8       if(uncoalesced memory accesses)
9            utilize smem to coalesce accesses
10      utilize gmem for data
```

Listing 4.1: Algorithm for determining the placement of data in GPU memory.

From Lines 7-10, data which is not reused for computation is stored in global memory (**gmem**). Since constant memory (**cmem**) and shared memory (**smem**) have similar access times, data, which is reused (*data reused*) and smaller than the maximum amount of constant memory, is stored in constant memory as shown in Lines 2 and 3. Data, which is reused but larger than constant memory, is partitioned into blocks no greater than the size of shared memory, as shown in Lines 4-6. However, the CPU does not have access to shared memory so data which is reused must be transferred to global memory. The GPU kernel partitions data which is reused into blocks and transfers the data from global to shared memory. Lines 8-9 refer to the access pattern to data which is not reused. If there is no data reuse, such as **A** for Mv, shared memory is utilized to coalesce the uncoalesced memory accesses and thus reduce computation time.

4.1.1 Mv

Listing 4.2 is the kernel for the global memory implementation of Mv on a GPU, where each thread computes one c_j. As mentioned in Section 4.2, computation time increases as each thread computes additional values of **c**. Each thread computes the index of the c_j to

```
1  __global__ void Mv(float* A, float* b, float* c, int n) {
2    int index = blockIdx.x * blockDim.x + threadIdx.x;
3    float temp = 0.0;
4    for(int j = 0; j < n; j++) {
5      temp += A[index * n + j] * b[j];
6    }
7    c[index] = temp;
8  }
```

Listing 4.2: The global memory implementation of Mv on a GPU.

compute in Line 2. Partial c_js are computed by each thread in Line 5 and the final c_j is stored in global memory in Line 7. The number of reads to **A** and **b** for this kernel is defined by Equation (3.2). Assuming $dBlk.x \geq 16$, the number of reads is $\frac{17 \times n^2}{16}$.

The CPU code for this implementation of Mv is shown in Listing 4.3.

```
1  ...
2  float* d_A;
3  cudaMalloc((void**) &d_A, sizeof(float) * n * n);
4  float* d_b;
5  cudaMalloc((void**) &d_b, sizeof(float) * n);
6  float* d_c;
7  cudaMalloc((void**) &d_c, sizeof(float) * n);
8  cudaMemcpy(d_A, h_A, sizeof(float) * n * n, cudaMemcpyHostToDevice);
9  cudaMemcpy(d_b, h_b, sizeof(float) * n, cudaMemcpyHostToDevice);
10 Mv<<< dGrd.x, dBlk.x >>>(n, d_A, d_b, d_c);
11 cudaMemcpy(h_c, d_c, sizeof(float) * n, cudaMemcpyDeviceToHost);
12 ...
```

Listing 4.3: CPU code for the global memory implementation of Mv.

The notation h_ denotes CPU (host) memory and d_ denotes GPU (device) memory. Lines 2-7 allocate the matrix and vectors in global memory of the GPU. Lines 8 and 9 transfer **A**, h_A, and **b**, h_b, from the CPU to global memory, d_A and d_b. Line 10 launches the kernel for execution using $dGrd.x$ blocks and $dBlk.x$ threads per block. The size of data, n, and pointers to **A**, **b** and **c** are passed to the kernel. Line 11 transfers **c** from the GPU global memory, d_c, to the CPU, h_c.

Lines 2-3 of Listing 4.1, which determines the placement of data, specifies that reused data is placed in constant memory if the size of the reused data is less than the size of

constant memory. **b**, which is reused, is of size n. The upper limit of n is defined by Equation (4.1). Therefore, since $n \leq 16384$, **b** is stored in constant memory. Listing 4.4 shows the modification of the kernel in Listing 4.2 necessary to utilize constant memory.

```
1 __constant__ float b[n];
2 __global__ void Mv(float* A, float* c, int n) {
3 ...
```

Listing 4.4: The constant memory implementation of Mv on a GPU.

Line 1 allocates constant memory of size n for **b**. Constant memory cannot be dynamically allocated and therefore, n must be defined for execution. Pointers to **A** and **c**, which reside in global memory, and the height or width of **A**, n, are passed to the kernel as shown in Line 2 of the kernel definition. To transfer **b** from the CPU to GPU constant memory, Line 9 is modified from Listing 4.3 as shown in Listing 4.5.

```
8  ...
9  cudaMemcpyToSymbol(b, h_b, sizeof(float) * n);

10 Mv<<< dGrd.x, dBlk.x >>>(n, d_A, d_c);
11 ...
```

Listing 4.5: CPU code for the constant memory implementation of Mv.

Line 10 of Listing 4.3 is modified such that the kernel is no longer passed a pointer to **b**. No other modifications to the kernel or CPU code are necessary. Utilizing constant memory for **b** eliminates global memory accesses to **b**. Therefore, the number of global memory reads for the constant memory implementation of Mv is

$$\text{Gld}_{32B} = n^2.$$

This is a 6.25% reduction of global memory reads compared to the global memory implementation of Mv. In addition, with fewer accesses to global memory, partition camping is reduced.

For the constant memory implementation of Mv, $thread_j$, to compute c_j, computes $row_j \cdot \mathbf{b}$ where row_j is one row of **A**. Therefore, the neighboring thread, $thread_{j+1}$, computes $row_{j+1} \cdot \mathbf{b}$. An example of a HW accessing **A** is depicted in Figure 4.1. In this example,

the HW is only 4 threads and each memory access is 4 bytes. Since the values of **A** needed in each iteration reside in different rows, the accesses to **A** are uncoalesced and the HW performs 4 reads in each iteration.

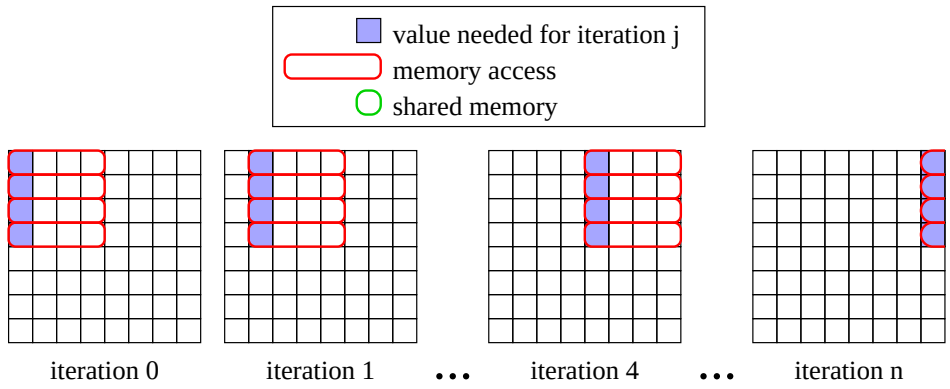


Figure 4.1: Example of a HW, consisting of 4 threads, accessing **A** in each iteration of Mv. Each memory transaction is 4 bytes. The accesses to **A** are uncoalesced and 4 accesses are required for each iteration.

Assuming $dBlk.x \geq 16$, each HW accesses 16 values of **A** from global memory. The 16 values of **A** reside in different rows of **A**, and therefore the accesses are uncoalesced and require 16 $32B$ transactions[1]. This is repeated for each HW n times. Coalescing the accesses to **A** significantly reduces the number of transactions and thus the computation time. Reducing the number of transactions also reduces the effects of partition camping.

Lines 8-9 of Listing 4.1, which defines the placement of data, specifies that shared memory is utilized to coalesce uncoalesced accesses for data which is not reused. Since accesses to **A** of the global and constant memory implementations of Mv are uncoalesced, shared memory is utilized to coalesce accesses as shown in Figure 4.2. In the figure, the HW is only 4 threads.

[1] $32B$ is the minimum size of a memory transaction on the T10 GPU.

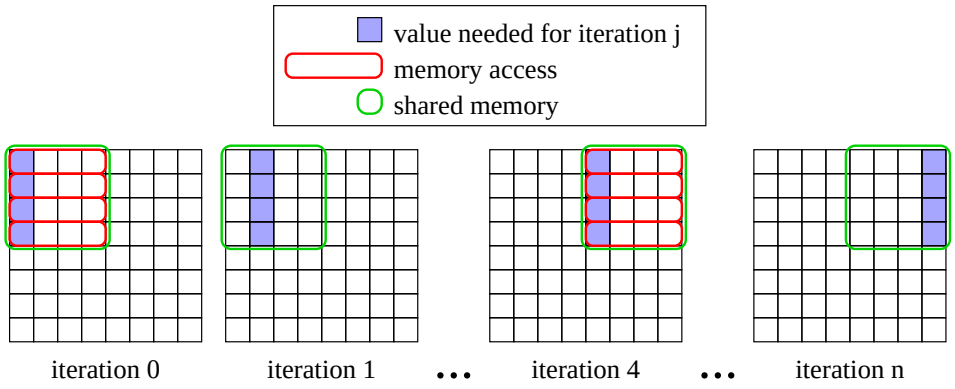


Figure 4.2: Example of a HW, consisting of 4 threads, accessing **A** in each iteration of Mv utilizing shared memory. Each memory transaction is 4 bytes. The accesses to **A** are coalesced and 4 accesses are required every 4^{th} iteration.

In the first iteration, the HW performs 4 reads of 4 bytes each and stores the data in shared memory. Therefore, in the next iteration, the next value needed for computation is read from shared memory instead of global. This reduces the number of global memory accesses by a factor of 4. Listing 4.6 is the kernel for the shared memory implementation of Mv.

```
1  __constant__ float b[n];
2  __global__ void Mv(int n, float* A, float* c) {
3    __shared__ float As[blockDim.x][blockDim.x];
4    int index = blockIdx.x * blockDim.x;
5    float temp = 0.0;
6    for(int j = 0; j < n; j += blockDim.x) {
7      for(int k = 0; k < blockDim.x; k++) {
8        As[k][threadIdx.x] = A[(index + k) * n + threadIdx.x + j];
9      }
10     __syncthreads();
11     for(k = 0; k < blockDim.x; k++) {
12       temp += As[threadIdx.x][k] * b[j + k];
13     }
14     __syncthreads();
15   }
16   c[index + threadIdx.x] = temp;
17 }
```

Listing 4.6: The shared memory implementation of Mv on a GPU. Constant memory is utilized for **b**.

In Line 3, shared memory is allocated per block of size $dBlk.x^2$. Therefore, for this kernel, the amount of shared memory per block is

$$\text{SMemPerBlk} = (dBlk.x^2) \times 4. \tag{4.2}$$

The first row each block accesses from **A** is calculated in Line 4. In Lines 7-9, HWs load the values of **A** into shared memory in a coalesced manner.

Line 10 is a synchronization instruction. HWs in a block stop executing at this instruction until all HWs within the block reach this instruction. This is to ensure that shared memory is loaded before the following computation. In Lines 11-13, threads utilized shared memory values to compute a portion of c_j. Line 14 ensures each HW completes the current iteration of computation before reloading shared memory. Similar to constant memory, shared memory cannot be allocated dynamically and therefore, $dBlk.x$ must be defined.

As mentioned, the shared memory implementation of Mv, Listing 4.6, significantly reduces the number of global memory accesses by coalescing the accesses to **A**. The number of global memory accesses for the shared memory implementation of Mv is

$$\text{Gld}_{32B} = \frac{n^2}{dBlk.x} \quad \text{if } dBlk.x \leq 8$$

and

$$\text{Gld}_{64B} = \frac{n^2}{16} \quad \text{if } dBlk.x \geq 16.$$

Assuming $dBlk.x \geq 16$, the total number of reads for this implementation is $\frac{n^2}{16}$ which is 16 times less than the number of reads for the constant memory implementation. Therefore, the shared memory implementation of Mv significantly reduces the computation time compared to global and constant implementations.

Figure 4.3 depicts maximum effective bandwidth of the global memory implementation of Mv (Listing 4.2) and the shared and constant memory implementation (Listing 4.6). Effective bandwidth is calculated as $\frac{8n^2 bytes}{T^{GPU}_{comp.}}$ as discussed in Section 3.5.

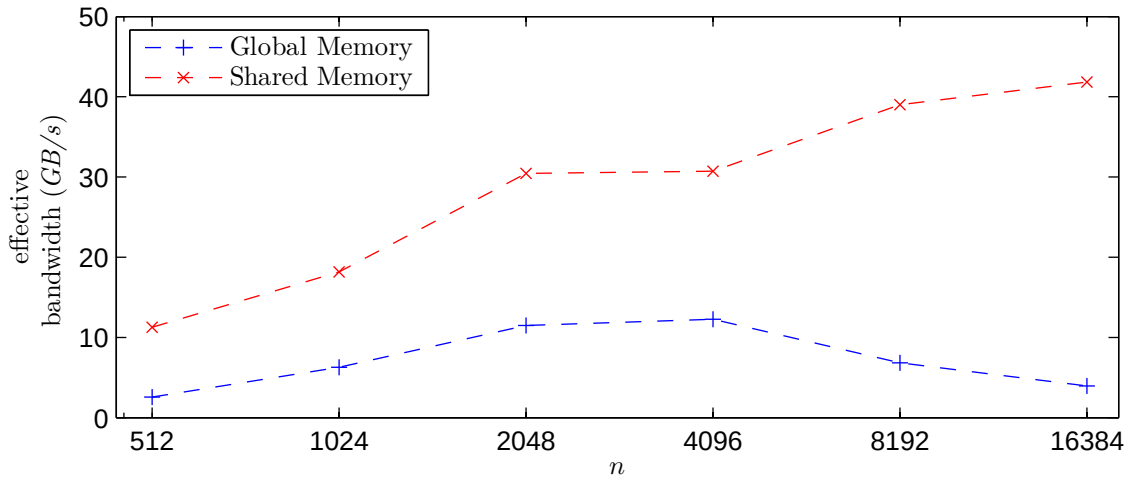


Figure 4.3: Comparison of GPU memories: Maximum effective bandwidth (GB/s) for Mv on the T10 GPU.

As illustrated in the figure, the shared and constant memory implementation yields a significant speedup compared to the global memory implementation. As mentioned, utilizing shared memory for Mv coalesces memory accesses to global memory. Utilizing constant memory reduces the number of global memory transactions, and therefore, reduces partition camping. The shared and constant memory implementation yields a speedup, on average, of 4.79 compared to the global memory implementation. The minimum (n = 4096) and maximum (n = 16384) speedups are 2.51 and 10.58, respectively. Therefore, for Mv, this step of the procedure significantly reduces GPU computation time.

4.1.2 MM

Listing 4.7 is the kernel for the global memory implementation of MM if $thread_{ij}$ computes C_{ij}. Pointers to **A**, **B** and **C** and the height or width of each matrix, n, are passed to the kernel in Line 1. Lines 2 and 3 compute the row and column indexes of the C_{ij} being computed for each thread, respectively. The x- and y-indices of each block are specified by

```
1 __global__ void MM(float* A, float* B, float* C, int n) {
2   int Row = blockIdx.y * blockDim.y + threadIdx.y;
3   int Col = blockIdx.x * blockDim.x + threadIdx.x;
4   float temp = 0.0;
5   for(int j = 0; j < n; j++) {
6     temp += A[Row * n + j] * B[j * n + Col];
7   }
8   C[Row * n + Col] = temp;
9 }
```

Listing 4.7: The global memory implementation of MM on a GPU where threads compute one C_{ij}.

blockIdx.x and blockIdx.y, respectively. Block indices in the x- and y-dimensions have a range of 0 to $dGrd.x - 1$ and 0 to $dGrd.y - 1$, respectively. The x- and y-indices of each thread are specified by *threadIdx.x* and threadIdx.y, respectively. Thread indices in the x- and y-dimensions have a range of 0 to $dBlk.x - 1$ and 0 to $dBlk.y - 1$, respectively. The x- and y-dimensions of each block are specified by blockDim.x ($dBlk.x$) and blockDim.y ($dBlk.y$), respectively. In Line 6, each thread computes the partial C_{ij} and the final C_{ij} is stored in global memory in Line 8.

The number of global memory reads for the global memory implementation of MM is defined by Equations (3.3) and (3.4). Assuming $dBlk.x \geq 16$, all accesses to **A** and **B** are coalesced and $\frac{2n^3}{16}$ accesses are performed.

Lines 4-6 of Listing 4.1, which determines the placement of data, specifies that shared memory is utilized for reused data which is larger than the size of constant memory. Line 5 specifies that data which is reused is partitioned into blocks smaller than the size of shared memory. Therefore, **A** and **B** are partitioned into blocks of shared memory to reduce the number of global memory accesses as shown in Listing 4.8. Lines 3 and 4 allocate shared memory for blocks of **A** and **B**. Therefore, for this kernel,

$$\text{SMemPerBlk} = (dBlk.x \times dBlk.y + dBlk.x^2) \times 4. \tag{4.3}$$

```
1  __global__ void MM(float* A, float* B, float* C, int n, ...
2  int nele_x, int nele_y) {

3    __shared__ float As[blockDim.y][blockDim.x];
4    __shared__ float Bs[blockDim.x][blockDim.x];

5    int Row = blockIdx.y * blockDim.y + threadIdx.y;
6    int Col = blockIdx.x * blockDim.x + threadIdx.x;

7    float temp = 0.0;
8    for(int j = 0; j < n; j += blockDim.x) {
9      As[threadIdx.y][threadIdx.x] = A[Row * n + (j + threadIdx.x)];
10     Bs[threadIdx.y][threadIdx.x] = B[(j + threadIdx.y) * n + Col];
11     __syncthreads();

12     for(int k = 0; k < blockDim.x; k++) {
13       temp += As[threadIdx.y][k] * Bs[k][threadIdx.x];
14     }
15     __syncthreads();
16   }
17   C[Row * n + Col] = temp;
18 }
```

Listing 4.8: The shared memory implementation of MM on a GPU.

Lines 9 and 10 load blocks of **A** and **B** into shared memory. This kernel assumes $dBlk.x = dBlk.y$ and slight modifications are performed to Line 10 for loading **B** if $dBlk.x \neq dBlk.y$. Line 11 forces all threads within a block to wait until shared memory is loaded by all HWs within that block. Partial C_{ij}s are computed in Line 13. Line 15 forces all threads within a block to wait until all HWs have performed their respective computations with the current data in shared memory before new data is loaded.

Calculating the number of reads for the shared memory implementation of MM in Listing 4.8 yields

$$\text{Gld}_{32B} = \frac{n^3}{dBlk.x}\left(\frac{1}{dBlk.x} + \frac{1}{dBlk.y}\right) \quad \text{if } dBlk.x \leq 8 \tag{4.4}$$

and

$$\text{Gld}_{64B} = \frac{n^3}{16}\left(\frac{1}{dBlk.x} + \frac{1}{dBlk.y}\right) \quad \text{if } dBlk.x \geq 16. \tag{4.5}$$

If $dBlk.x \geq 16$, all global memory accesses are $64B$ and the number of accesses to global memory is reduced by a factor of $\frac{2dBlk.x \times dBlk.y}{dBlk.x + dBlk.y}$ compared to the global memory implementation.

Figure 4.4 depicts maximum effective bandwidth of the global memory implementation of MM (Listing 4.7) and the shared memory implementation (Listing 4.8). For MM, effective bandwidth is calculated as $\frac{8n^3 bytes}{T^{GPU}_{comp.}}$ as discussed in Section 3.5.

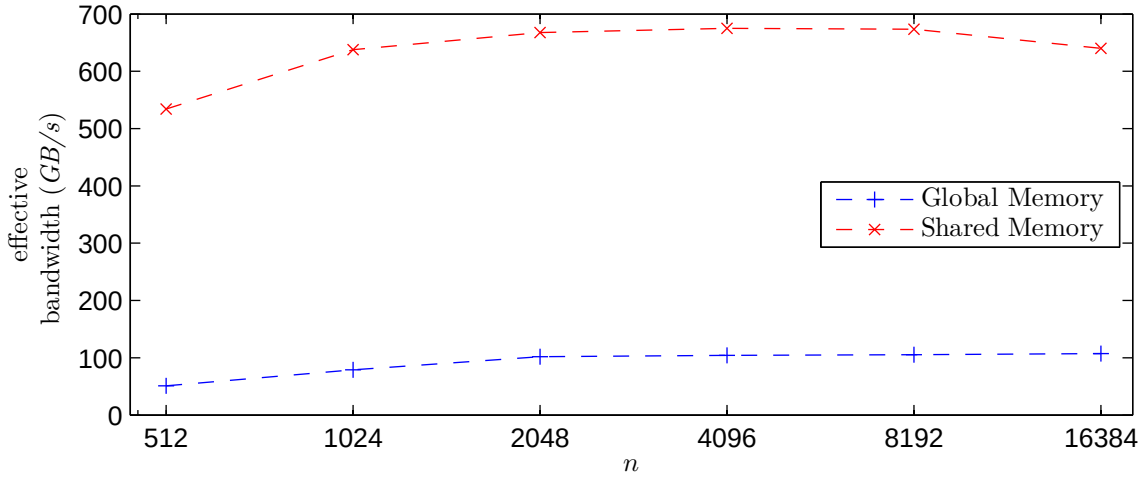


Figure 4.4: Comparison of GPU memories: Maximum effective bandwidth (GB/s) for MM on the T10 GPU.

Similar to Mv, the shared memory implementation yields a significant speedup compared to the global memory implementation. Utilizing shared memory reduces the number of global memory transactions by data reuse, thus reducing computation time and increasing effective bandwidth. The shared memory implementation yields a speedup, on average, of 7.32 compared to the global memory implementation. The minimum (n = 16384) and maximum (n = 512) speedups are 5.96 and 10.46, respectively. Therefore, for MM, Step 1 of the procedure significantly reduces GPU computation time.

4.2 Computation Patterns

Computation patterns are determined by the code and represent the manner and order in which results are computed. For Mv, if $thread_j$ computes one value of $\mathbf{c}$, c_j, where $\mathbf{c}$=$\mathbf{Ab}$, there are no varying computation patterns. Likewise, for MM, if $thread_{ij}$ computes one value of $\mathbf{C}$, C_{ij}, where $\mathbf{C}$=$\mathbf{AB}$, there are no varying computation patterns. However, if

threads compute more or less than one value, the order in which threads compute creates computation patterns.

Since computation patterns are affected by partition camping, a derivation of the computation pattern which minimizes partition camping is necessary. In addition, computation patterns can affect the number of accesses to global memory. Therefore, it is necessary to derive the optimal computation patterns for matrix-based computations. Deriving the optimal pattern requires an examination of the various patterns.

4.2.1 Mv

For Mv, there are two computation patterns if $thread_j$ computes multiple values of **c** as depicted in Figure 4.5. A computation pattern where each thread computes neighboring

(a) Grouped. (b) Spread.

Figure 4.5: Computation patterns: two computation patterns for Mv for computing multiple C_js.

values of **c**, *grouped*, is depicted in Figure 4.5a. A computation pattern where each thread computes values of **c** which are not neighboring, *spread*, is depicted in Figure 4.5b.

Figure 4.6 depicts the average computation time for Mv on the T10 GPU if each thread computes one or more values of **c** utilizing the spread computation pattern. $dGrd.x \times dBlk.x$ is the number of total threads. Each thread computes at least one value of **c** and therefore, the maximum number of total threads is n. The average computation time decreases as the number of total threads increases. The minimum time is achieved, for all data sizes tested, when the number of total threads equals n. This is due to the memory bus not being fully utilized when $dGrd.x \times dBlk.x \leq 8192$.[2]

[2]Discussion on the number of threads necessary to saturate the memory bus is in Section 4.5.

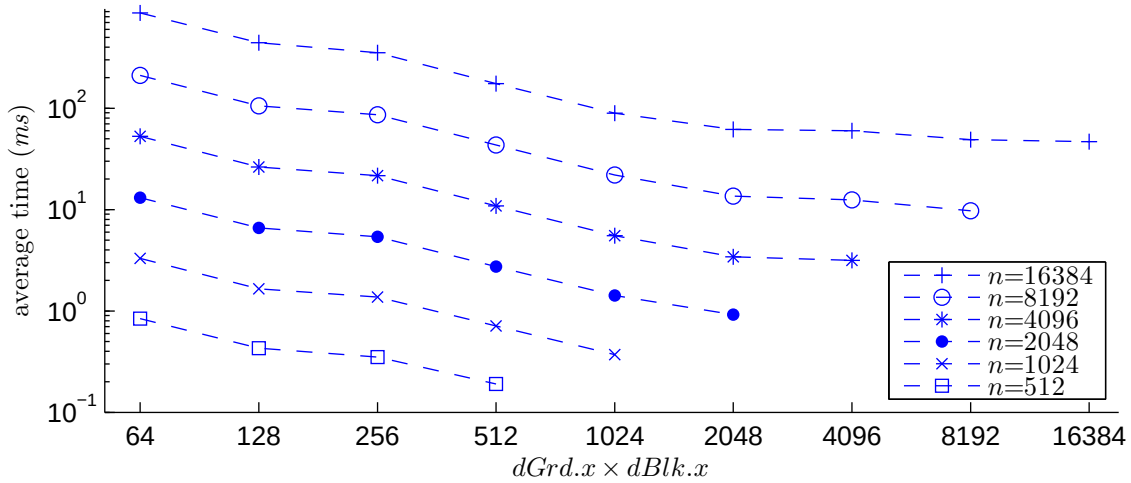


Figure 4.6: Average computation time (ms) varying the number of total threads for Mv on the T10 GPU.

Optimized

To increase the number of total threads, it is necessary that multiple threads are utilized to compute one c_j. Therefore, the optimized computation pattern consists of each thread computing a part of c_j as depicted in Listing 4.9. In Listing 4.9, each thread computes a part of c_j and each block computes $dBlk.x$ c_js. Shared memory is allocated in Line 3 and therefore,

$$\text{SMemPerBlk} = dBlk.x^2 \times 4.$$

In Line 5, *index* is calculated, which specifies the group of c_js that are computed by each block. Lines 7-9 load parts of **A** into shared memory. All threads within a block are synchronized in Line 10 to ensure shared memory is loaded before computation. Each thread computes a part of c_j in Lines 11-13. Threads are again synchronized in Line 14 to prevent shared memory being reloaded before each thread completes computation with the current data in shared memory. At the end of the *for*-loop in Line 15, each thread contains a part of a c_j in *temp*. Threads with $threadIdx.y \neq 0$ store their parts of c_j into shared memory. The shared memory which is utilized for portions of **A** is reused for the parts of c_j computed by each thread. Threads with $threadIdx.y = 0$ are utilized to sum the parts of c_j stored in

```
1  __constant__ float b[n];
2  __global__ void Mv(float* A, float* c, int n) {

3    __shared__ float As[blockDim.x][blockDim.x];

4    float temp = 0.0;
5    int index = blockIdx.x * blockDim.x;

6    for(int j = 0; j < n; j+= blockDim.x) {
7      for(int k = 0; k < blockDim.x / blockDim.y; k++) {
8        As[k + threadIdx.y * blockDim.x / blockDim.y][threadIdx.x] =
       ↳ A[(index + k + threadIdx.y * blockDim.x / blockDim.y) * n +
       ↳ threadIdx.x + j];
9      }
10     __syncthreads();

11     for(int k = 0; k < blockDim.x / blockDim.y; k++) {
12       temp += As[threadIdx.x][k + threadIdx.y * blockDim.x / blockDim.y]
             ↳ * b[j + k + threadIdx.y * blockDim.x / blockDim.y];
13     }
14     __syncthreads();
15   }

16   if(threadIdx.y != 0) {
17     As[threadIdx.y-1][threadIdx.x] = temp;
18   }
19   __syncthreads();
20   if(threadIdx.y == 0) {
21     for(k = 0; k < blockDim.y-1; k++) {
22       temp += As[k][threadIdx.x];
23     }
24     c[index + threadIdx.x] = temp;
25   }
26 }
```

Listing 4.9: The shared memory implementation of Mv on a GPU utilizing the optimized computation pattern. Constant memory is utilized for **b**.

shared memory in Lines 20-23. Lastly, in Line 24, these threads store the final c_js in **c** in global memory.

Since **b** is stored in constant memory, global memory reads are issued only for accessing **A**. In Line 8, the number of accesses to **A** by each HW is dependent on $dBlk.x, dBlk.y$ and $threadIdx.y$. Therefore,

$$\text{Gld}_{32B} = \frac{n^2}{dBlk.x} \quad \text{if } dBlk.x \leq 8 \tag{4.6}$$

and

$$\text{Gld}_{64B} = \frac{n^2}{16} \quad \text{if } dBlk.x \geq 16 \tag{4.7}$$

which is equivalent to the number of accesses in the shared memory implementation of Mv without the optimized computation pattern (Listing 4.6).

Figure 4.7 illustrates the maximum effective bandwidth of Mv before (Listing 4.6) and after (Listing 4.9) utilizing the optimized computation pattern. As discussed in Section 3.5, for Mv, effective bandwidth is calculated as $\frac{8n^2 bytes}{T^{GPU}_{comp.}}$.

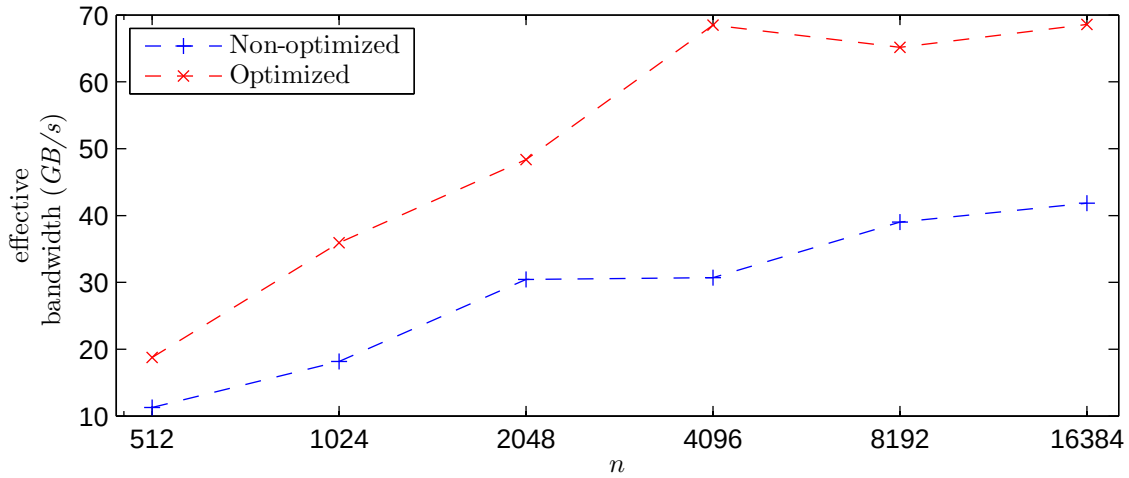


Figure 4.7: Comparison of computation patterns: Maximum effective bandwidth (GB/s) for Mv on the T10 GPU.

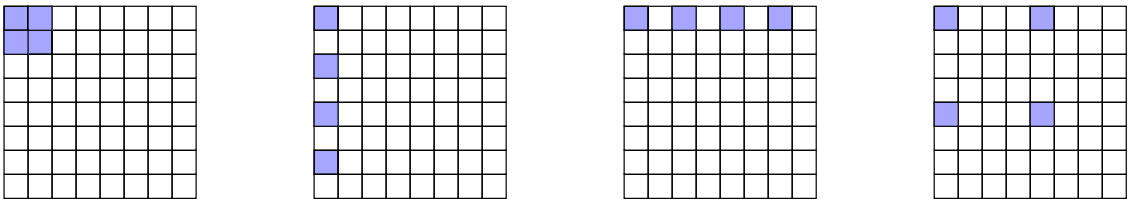
Since the optimized computation pattern increases the number of total threads, the effective bandwidth is significantly increased. The optimized computation pattern yields a speedup, on average, of 1.80 compared to the non-optimized pattern. The minimum

(n = 2048) and maximum (n = 4096) speedups are 1.59 and 2.23, respectively. Therefore, Step 2 of the parallelization procedure, deriving the optimized computation pattern, is valid and necessary for minimizing GPU computation time of Mv.

4.2.2 MM

For MM, rather than Mv, increasing the number of total threads does not necessarily yield the minimum computation time due to saturation of the memory bus. With a saturated memory bus, the effects of fragmentation and partition camping are more apparent. Therefore, if $thread_{ij}$ computes more than C_{ij}, the computation pattern, or order in which the threads compute results, affects the computation time.

Four computation patterns for MM are illustrated in Figure 4.8 if each thread computes multiple C_{ij}s. Figure 4.8a depicts a computation pattern where each thread computes



(a) Grouped. (b) Column-major. (c) Row-major. (d) Hybrid.

Figure 4.8: Computation patterns: four computation patterns for MM for computing multiple C_{ij}s.

neighboring C_{ij}s, defined as *grouped*. Figures 4.8b and c, defined as *column-major* and *row-major*, respectively, depict computation patterns where each thread computes multiple C_{ij}s per column or row, respectively. Merging column- and row-major patterns yields the last computation pattern, defined as *hybrid*, as depicted in Figure 4.8d.

Grouped

The grouped computation pattern, depicted in Figure 4.8a, causes a reduction of memory coalescing. Coalescing as many memory reads as possible at the HW level is ideal to minimize the computation time. Considering an example where each thread is to compute

4 C_{ij}s in a grouped pattern, C_{ij}, $C_{i(j+1)}$, $C_{(i+1)j}$ and $C_{(i+1)(j+1)}$, to compute the first C_{ij}, $thread_{ij}$ accesses one row of **A**, row_i and one column of **B**, col_j. Since threads are assigned to warps in row-major order, the neighboring thread of $thread_{ij}$, $thread_{i(j+1)}$, accesses the same row of **A**, row_i, but a differing column of **B**, col_{j+2} to compute $C_{i(j+2)}$ [3]. In general, each HW read will only use $\frac{1}{x}$ of the bytes where x is the number of C_{ij}s computed in the x-dimension, thus reducing the number of coalesced memory accesses to **B** by a factor of x. Therefore, this computation pattern is not included in the results as it demonstrates the effects of memory coalescing more than partition camping.

Column-major

The row- and column-major computation patterns do not cause a reduction of memory coalescing, as the grouped pattern does, if each block has at least 8 threads in the x-dimension, $dBlk.x \geq 8$. For the column-major computation pattern, depicted in Figure 4.8b, $thread_{ij}$ computes $C_{(i+y\times dGrd.y\times dBlk.y)j}$ for $y = 0$ to nele_y where nele_y is the number of C_{ij}s to compute. For this pattern, col_j is repeatedly read by $thread_{ij}$ while the row that is read varies from i to (nele_y$\times dGrd.y \times dBlk.y$). If $n\%512 = 0$, all accesses to col_j remain in the same partition.

Row-major

For the row-major computation pattern, depicted in Figure 4.8c, $thread_{ij}$ computes $C_{i(j+x\times dGrd.x\times dBlk.x)}$ for $x = 0$ to nele_x where nele_x is the number of C_{ij}s to compute. Therefore, to compute C_{ij} to $C_{i(j+dBlk.x-1)}$, $thread_{ij}$ to $thread_{i(j+dBlk.x-1)}$ read row_i and the column reads vary in a coalesced pattern from col_j to $col_{j+dBlk.x-1}$. However, for computing additional C_{ij}s, the accesses to each partition for reading **B** vary depending on which C_{ij} is being computed. Accesses to each partition for reading **A** always vary as the program execution continues as it is not possible for an entire row of **A** to be stored in one partition if

[3] $thread_{i(j+1)}$ computes $C_{i(j+2)}$ instead of $C_{i(j+1)}$ because of the grouped computation pattern. In this case, $thread_{ij}$ is computing $C_{i(j+1)}$ as well as $C_{(i+1)j}$ and $C_{(i+1)(j+1)}$.

$n > 8$ without transposing **A**. However, an entire column of **B** is stored in one partition if $n\%512 = 0$. For this computation pattern, as the program execution continues, each HW is accessing varying partitions to read **B** dependent on the current C_{ij} being computed.

Hybrid

For the hybrid computation pattern, depicted in Figure 4.8d, it is necessary to determine the order each C_{ij} is computed. Two computation patterns are defined and illustrated in Figure 4.9: *hybrid-column* and *hybrid-row*. The hybrid-column computation pattern allows each thread to compute multiple C_{ij}s in each row and column but computes each first in the y-dimension. The hybrid-row is similar except each thread computes each C_{ij} first in the x-dimension. Therefore, the column-major computation pattern is a subset of hybrid-column, and the row-major is a subset of hybrid-row. Because of this, hybrid-column and hybrid-row computation patterns are presented in the results, and row- and column-major are not.

(a) Hybrid-column. (b) Hybrid-row.

Figure 4.9: Computation patterns: hybrid computation patterns for MM for computing multiple C_{ij}s.

Similar to row- and column-major computation patterns, both hybrid patterns, defined in Figure 4.9, benefit from memory coalescing if $dBlk.x \geq 8$. If $(dGrd.x \times dBlk.x)\%512 = 0$, then all C_{ij}s computed by $thread_{ij}$ will be in the same partition. Therefore, with regards to the distribution of the HWs to all partitions, and from the GPU's perspective, the hybrid-row and hybrid-column patterns are identical. In addition, in this case, both patterns are identical to the column-major pattern.

However, if $(dGrd.x \times dBlk.x)\%512 \neq 0$, there is a difference between the hybrid-row and hybrid-column pattern. In both, $thread_{ij}$ computes $C_{(i+y\times dGrd.y\times dBlk.y)(j+x\times dGrd.x\times dBlk.x)}$

for y = 0 to nele_y and x = 0 to nele_x where nele_x and nele_y are the number of C_{ij}s to compute in the x- and y-dimensions, respectively. If all partitions are not initially accessed by the HWs, the hybrid-row pattern accesses the unused partitions quicker than the hybrid-column pattern thus reducing partition camping.

Optimized (column-major)

Memory coalescing and the order of computation are considered to derive the optimized computation pattern for MM. The row- and column-major patterns, in addition to the hybrid patterns, do not reduce memory coalescing as the grouped pattern does. From the kernel in Listing 4.8, the shared memory implementation of MM on a GPU, global memory is accessed in Lines 9 and 10. Computation is then performed in Line 13 before reloading shared memory. In this pattern, little overlap exists between accessing global memory and computation. Therefore, the optimized computation pattern consists of an algorithm maximizing the overlap of global memory accesses and computation such as the one described in [63] and utilized in [19], [20], and [21]. In this implementation, only one matrix is placed in shared memory. In addition, the aforementioned implementations assume column-major storage for matrices. In this work, row-major storage is assumed due to the row-major storage of the GPU. Although the algorithm for MM is similar to the aforementioned implementations, other steps of the parallelization procedure augment the algorithm to further minimize GPU computation time. Utilizing this algorithm and the optimized computation pattern depicted in Figure 4.8b yields Listing 4.10. In this implementation of MM, all memory accesses are coalesced since the column-major computation pattern is utilized. Shared memory is allocated for parts of **A** in Line 2. Therefore, the amount of shared memory allocated is modified from Equation (4.3) to yield

$$\text{SMemPerBlk} = (n\_ele \times dBlk.x) \times 4 \tag{4.8}$$

```
1  __global__ void MM( float *A, float *B, float *C, int n) {
2    __shared__ float As[n_ele][blockDim.x];
3    float temp[N_ele];
4    for(int i = 0; i < n_ele; i++) {
5      temp[i] = 0.0;
6    }
7    for(int j = 0; j < n / blockDim.x; j++) {
8      for(int i = 0; i < n_ele / blockDim.y; i++) {
9        As[threadIdx.y + i * blockDim.y][threadIdx.x] = A[threadIdx.x +
             ↳ (blockIdx.y * n_ele + threadIdx.y) * n + (i * blockDim.y * n
             ↳ + j * blockDim.x)];
10     }
11     __syncthreads();
12     for(int i = 0; i < blockDim.x; i++) {
13       for(int k = 0; k < n_ele; k++) {
14         temp[k] += As[k][i] * B[(blockIdx.x * blockDim.x * blockDim.y +
       ↳ threadIdx.y * blockDim.x + threadIdx.x) + (i * n + j * n *
       ↳ blockDim.x)];
15       }
16     }
17     __syncthreads();
18   }
19   for (int i = 0; i < n_ele; i++) {
20     C[i * n + blockIdx.x * blockDim.x * blockDim.y + threadIdx.y *
         ↳ blockDim.x + threadIdx.x + blockIdx.y * n_ele * n] = temp[i];
21   }
22 }
```

Listing 4.10: The shared memory implementation of MM on a GPU utilizing the optimized computation pattern.

where n_ele is the number of C_{ij}s computed per thread. In Lines 3-6, each thread allocates and initializes temporary variables for each C_{ij} being computed. A portion of **A** is loaded into shared memory in Lines 8-10. This implementation assumes $dBlk.y \leq n\_ele$. Slight modifications are made to loading **A** into shared memory in Lines 8-10 if $dBlk.y > n\_ele$. Threads within a block synchronize in Line 11 to ensure **A** is loaded. In Lines 13-15, each thread accesses **B** once and performs computation with that value and shared memory n_ele times. **B** is not read in every iteration of k in Line 14 as the access to **B** is independent of the value of k. Therefore, the overlap between accessing global memory and computation is greatly increased. In addition, this implementation requires that only portions of one matrix be loaded into shared memory thus reducing the amount of shared memory necessary.

The number of global memory accesses to **A** and **B** is dependent on n, $dBlk.x$, $dBlk.y$, and n_ele. After simplifying,

$$\text{Gld}_{32B} = \frac{n^3}{dBlk.x}\left(\frac{1}{dBlk.x \times dBlk.y} + \frac{1}{n\_ele}\right) \quad \text{if } dBlk.x \leq 8 \tag{4.9}$$

and

$$\text{Gld}_{64B} = \frac{n^3}{16}\left(\frac{1}{dBlk.x \times dBlk.y} + \frac{1}{n\_ele}\right) \quad \text{if } dBlk.x \geq 16. \tag{4.10}$$

Therefore, the number of global memory accesses is reduced by a factor of $\frac{n\_ele(dBlk.x+dBlk.y)}{n\_ele+dBlk.x\times dBlk.y}$ compared to the shared memory implementation of MM without the optimized computation pattern (Listing 4.8).

Figure 4.10 depicts maximum effective bandwidth of MM before (Listing 4.8) and after (Listing 4.10) utilizing the optimized computation pattern. The effective bandwidth is calculated in Section 3.5 as $\frac{8n^3 bytes}{T^{GPU}_{comp.}}$.

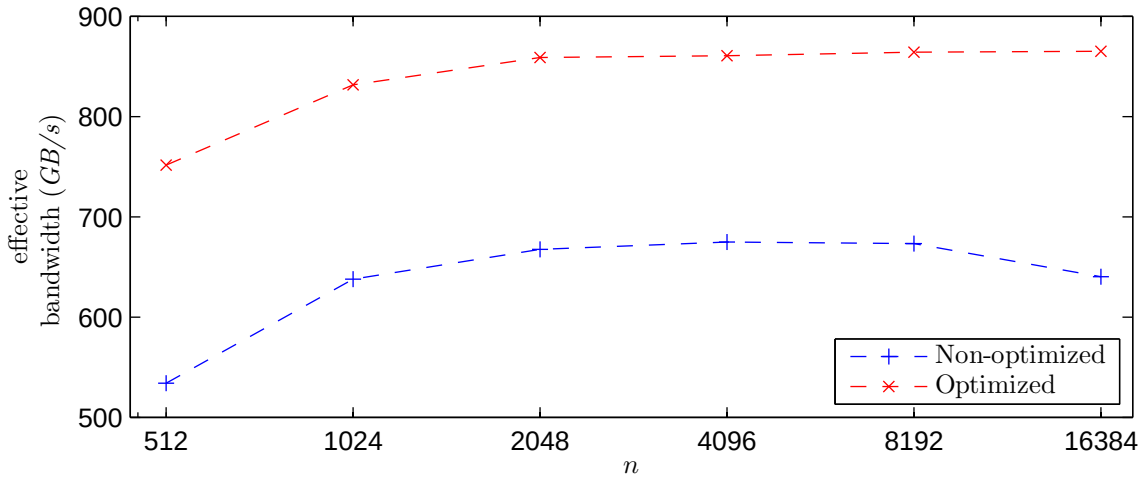


Figure 4.10: Comparison of computation patterns: Maximum effective bandwidth (GB/s) for MM on the T10 GPU.

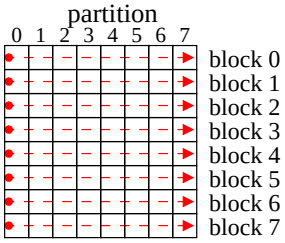
Similar to Mv, the optimized computation pattern yields a significant speedup. In the figure, the optimized computation pattern yields a speedup, on average, of 1.32 compared to the non-optimized pattern. The speedup is due to the reuse of data by the optimized computation pattern. As mentioned, the optimized computation pattern reduces the number of global memory transactions and therefore, the effective bandwidth is increased. The minimum (n = 4096) and maximum (n = 512) speedups are 1.28 and 1.41, respectively. Therefore, Step 2 of the parallelization procedure is valid and necessary for reducing GPU computation time of MM.

4.3 Access Patterns

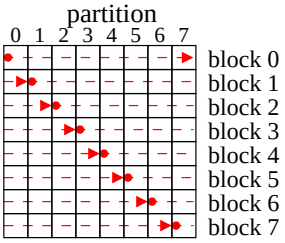
The third step of the parallelization procedure specifies that the optimized access pattern to data in memory be derived. The optimized access patterns are derived to minimize partition camping (Section 3.1) and eliminate bank conflicts (Section 3.2). Therefore, the GPU computation time is reduced.

For matrix-based computations such as Mv and MM, all HWs begin by initially reading the first value of their respective rows of **A**. Therefore, distributing the accesses to **A** at the block level to all partitions reduces partition camping. Figure 4.11a, assuming $n \geq 512$,

illustrates an access pattern to **A** of 8 blocks for Mv or MM. At the beginning of execution, block_i reads the first value of row_i from **A**, which is stored in partition 0. As execution continues, block_i accesses row_i of **A** where row_i is stored across all partitions. However, since all blocks begin accessing the first value of **A**, all blocks begin reading from partition 0 and thus, partition camping occurs. Therefore, an optimized access pattern which minimizes partition camping for accessing **A** is derived by distributing blocks to all partitions at the beginning of execution. This is illustrated in Figure 4.11b.



(a) Standard access pattern.



(b) Optimized access pattern.

Figure 4.11: Access patterns: example of block-level access patterns to **A** for matrix-based computations. Each column is one partition of global memory. Each row is 512 values of type float.

4.3.1 Mv

To implement Figure 4.11b for the shared memory implementation of Mv utilizing the optimized computation pattern, the kernel in Listing 4.9 is modified to yield the kernel in Listing 4.11. This kernel utilizes the optimized access pattern. In Line 6, the partition from which each block begins accessing **A**, P, is calculated. The *for*-loop from Line 6 in Listing 4.9 is split into 2 *for*-loops, Line 7 and 17 of Listing 4.11, such that blocks begin accessing **A** from varying partitions, P. P is computed using $blockId.x$ mod 8 since there are 8 global memory partitions. This is multiplied by 64 since each partition is 256 bytes wide and values are of type float.

The allocation of shared memory in Line 3 of Listing 4.11 is greater than the allocation of shared memory in Listing 4.9. Therefore, Equation (4.2), which defines the amount of

```
1  __constant__ float b[n];
2  __global__ void Mv(float* A, float* c, int n) {
3    __shared__ float As[blockDim.x][blockDim.x+1];
4    float temp = 0.0;
5    int index = blockIdx.x * blockDim.x;
6    int P = (blockIdx.x  % 8) * 64;
7    for(int j = P; j < n; j+= blockDim.x) {
8      for(int k = 0; k < blockDim.x / blockDim.y; k++) {
9        As[k + threadIdx.y * blockDim.x / blockDim.y][threadIdx.x] =
             ↳ A[(index + k + threadIdx.y * blockDim.x / blockDim.y) * n +
             ↳ threadIdx.x + j];
10     }
11     __syncthreads();
12     for(int k = 0; k < blockDim.x / blockDim.y; k++) {
13       temp += As[threadIdx.x][k + threadIdx.y * blockDim.x / blockDim.y]
             ↳ * b[j + k + threadIdx.y * blockDim.x / blockDim.y];
14     }
15     __syncthreads();
16   }
17   for(int j = 0; j < P; j+= blockDim.x) {
18     for(int k = 0; k < blockDim.x / blockDim.y; k++) {
19       As[k + threadIdx.y * blockDim.x / blockDim.y][threadIdx.x] =
             ↳ A[(index + k + threadIdx.y * blockDim.x / blockDim.y) * n +
             ↳ threadIdx.x + j];
20     }
21     __syncthreads();
22     for(int k = 0; k < blockDim.x / blockDim.y; k++) {
23       temp += As[threadIdx.x][k + threadIdx.y * blockDim.x / blockDim.y]
             ↳ * b[j + k + threadIdx.y * blockDim.x / blockDim.y];
24     }
25     __syncthreads();
26   }
27   if(threadIdx.y != 0) {
28     As[threadIdx.y-1][threadIdx.x] = temp;
29   }
30   __syncthreads();
31   if(threadIdx.y == 0) {
32     for(k = 0; k < blockDim.y-1; k++) {
33       temp += As[k][threadIdx.x];
34     }
35     c[index + threadIdx.x] = temp;
36   }
37 }
```

Listing 4.11: The shared memory implementation of Mv on a GPU utilizing the optimized computation and access pattern. Constant memory is utilized for **b**.

shared memory used per block for the shared memory implementation of Mv utilizing the optimized computation pattern is modified to yield

$$\text{SMemPerBlk} = (dBlk.x^2 + dBlk.x) \times 4. \tag{4.11}$$

Although $dBlk.x^2 \times 4$ bytes of shared memory are necessary for the kernel in Listing 4.11, $(dBlk.x^2 + dBlk.x) \times 4$ bytes are allocated. The additional allocation in the kernel is to reduce bank conflicts. An example of bank conflicts for Mv is depicted in Figures 4.12-4.15. In the figures, $dBlk.x = 16$. Figure 4.12 depicts shared memory being loaded with **A** for the shared memory implementation of Mv utilizing the optimized computation pattern but not the optimized access pattern. This implementation utilizes $dBlk.x^2 \times 4$ bytes of shared memory. Since the threads within a HW access different banks of shared memory, no bank conflicts occur for loading shared memory.

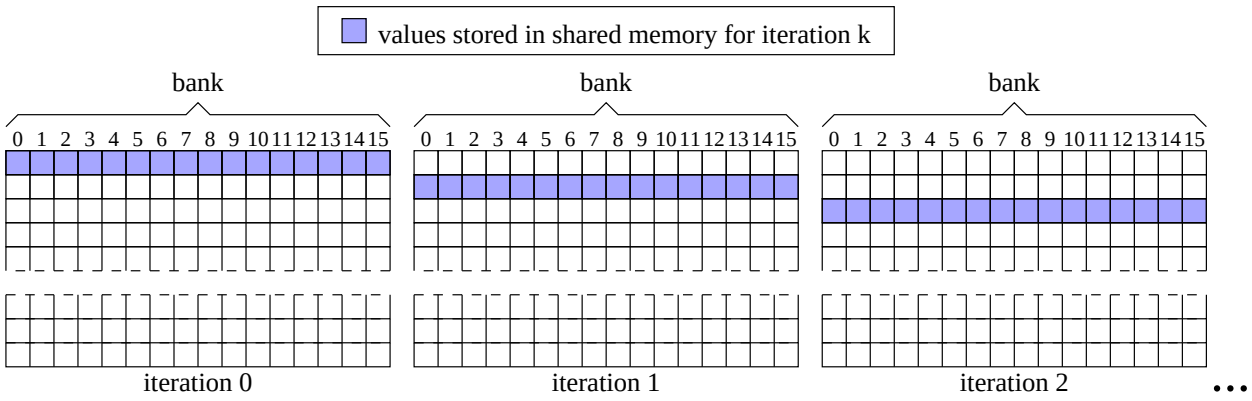


Figure 4.12: Example of loading **A** into shared memory for the shared memory implementation of Mv utilizing the optimized computation pattern but not the optimized access pattern. No bank conflicts occur. $dBlk.x = 16$.

However, for reading the values from shared memory, the HW accesses varying rows and one column of shared memory in each iteration as depicted in Figure 4.13. As illustrated, each value resides in the same shared memory bank and thus bank conflicts occur.

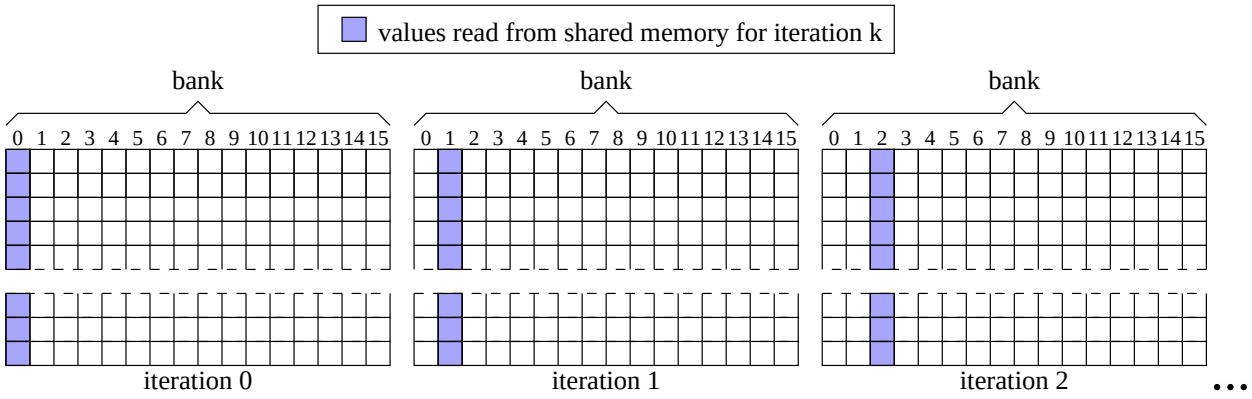


Figure 4.13: Example of reading shared memory for the shared memory implementation of Mv utilizing the optimized computation pattern but not the optimized access pattern. Bank conflicts occur. $dBlk.x = 16$.

Figure 4.14 depicts shared memory being loaded with **A** for the shared memory implementation of Mv that utilizes the optimized computation and access pattern. $(dBlk.x^2 + dBlk.x) \times 4$ bytes of shared memory are allocated. Since the threads within a HW access all shared memory banks, no bank conflicts occur.

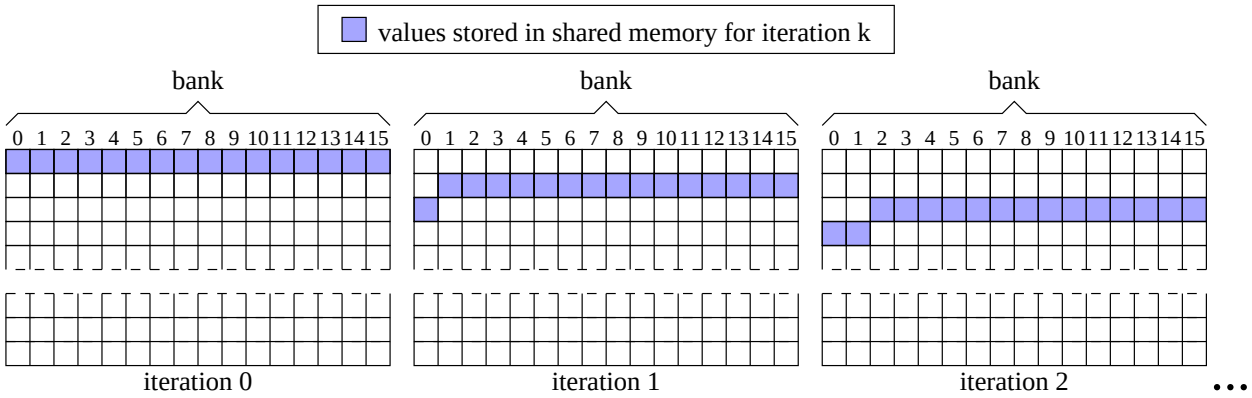


Figure 4.14: Example of loading **A** into shared memory for the shared memory implementation of Mv utilizing the optimized computation and access pattern. No bank conflicts occur. $dBlk.x = 16$.

For reading the values from shared memory, the HW accesses varying rows and the same column of shared memory in each iteration as depicted in Figure 4.15. As illustrated, each value resides in a different shared memory bank and therefore, no bank conflicts occur.

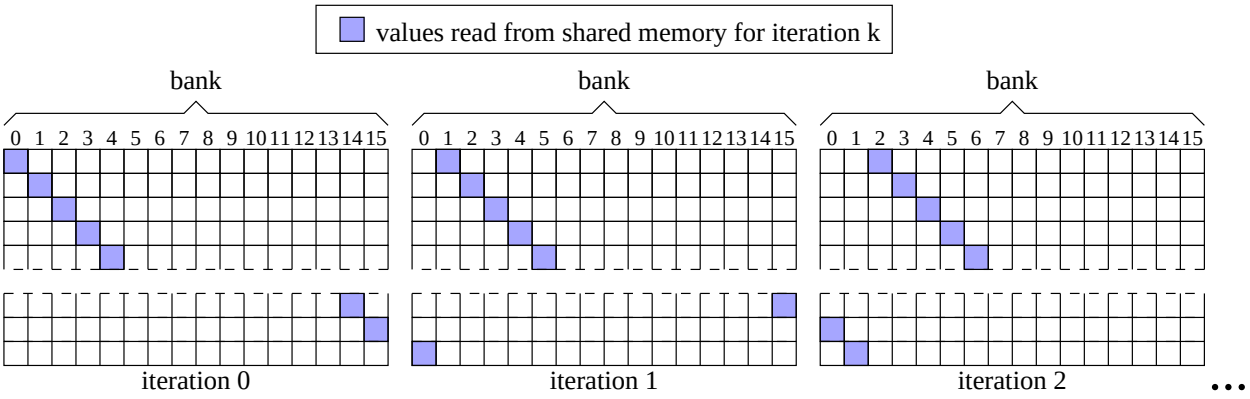


Figure 4.15: Example of reading shared memory for the shared memory implementation of Mv utilizing the optimized computation and access pattern. No bank conflicts occur. $dBlk.x = 16$.

Therefore, the optimized access pattern for Mv reduces partition camping and eliminates bank conflicts. Figure 4.16 depicts maximum effective bandwidth of Mv before (Listing 4.9) and after (Listing 4.11) utilizing the optimized access pattern. Effective bandwidth is calculated as $\frac{8n^2 bytes}{T^{GPU}_{comp.}}$ as discussed in Section 3.5.

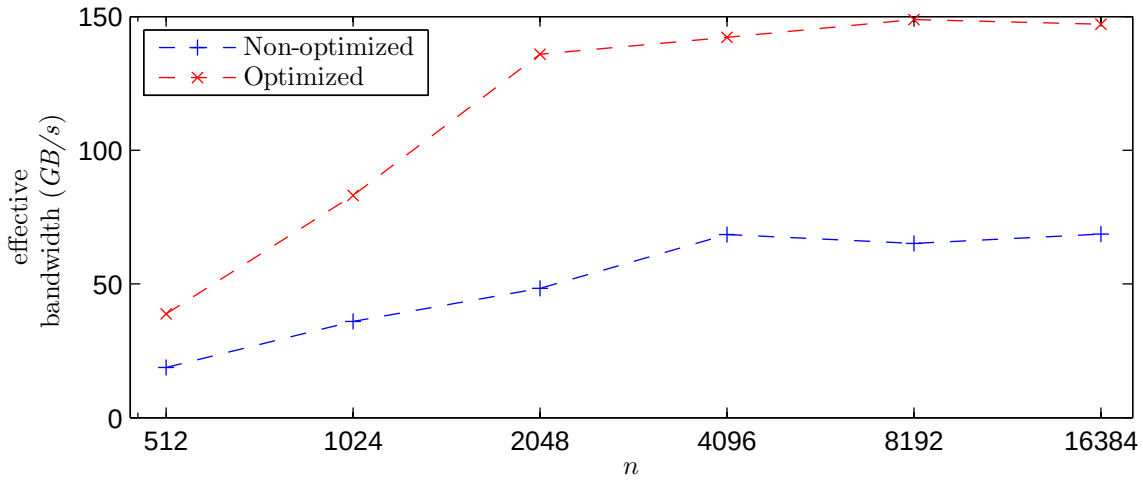


Figure 4.16: Comparison of access patterns: Maximum effective bandwidth (GB/s) for Mv on the T10 GPU.

In the figure, the optimized access pattern yields a speedup, on average, of 2.28 compared to the non-optimized pattern. The significant speedup is due to the optimized access pattern reducing partition camping and eliminating bank conflicts. The minimum ($n = 512$) and

maximum (n = 2048) speedups are 2.06 and 2.81, respectively. The significant increase in effective bandwidth shows the validity and necessity of Step 3 of the procedure for Mv, deriving the optimized access pattern.

4.3.2 MM

To reduce partition camping by implementing Figure 4.11b for the shared memory implementation of MM utilizing the optimized computation pattern, the kernel in Listing 4.10 is modified to yield Listing 4.12. This kernel utilizes the optimized access pattern. In Listing 4.12, $dBlk.y \leq n\_ele$. Slight modifications are performed to Lines 9-11 and 21-23 for loading **A** if $dBlk.y > n\_ele$. In this kernel, additional computation is required compared to Listing 4.10 to determine which value of **A** each block reads at the beginning of execution. Line 7 of Listing 4.12 assigns P, a variable used to specify which partition is accessed for reading **A** at the beginning of execution. To calculate P, a linear block index is calculated as $blkIdx.x + dGrd.x \times blkIdx.y$. The linear block index mod 8, the number of partitions, is calculated such that blocks are assigned to each partition at the beginning of execution in round-robin order. This is multiplied by $\frac{64}{blockDim.x}$ since there are 64 values of type float per row of each partition. This kernel utilizes the same amount of shared memory as the kernel in Listing 4.8. Therefore, Equation (4.8) defines the amount of shared memory utilized per block. Splitting the *for*-loop in Line 7 of Listing 4.10 as shown in Lines 8-20 of Listing 4.12 distributes the blocks to all partitions for reading **A** at the beginning of execution. Therefore, this optimized access pattern reduces partition camping and thus reduces the computation time.

Shared memory is allocated in Line 2 of Listing 4.12 the same as in Line 2 of Listing 4.10. Therefore, the amount of shared memory allocated per block is defined by Equation (4.8). In Lines 10 and 22, values from **A** are loaded into shared memory. If $dBlk.x \geq 16$, threads within a HW have the same value for $threadIdx.y$. Therefore, no bank conflicts occur for loading shared memory as all banks are utilized. When reading from As in Lines 15 and

```
1  __global__ void MM( float *A, float *B, float *C, int n) {
2    __shared__ float As[n_ele][blockDim.x];
3    float temp[N_ele];
4    for(int i = 0; i < n_ele; i++) {
5      temp[i] = 0.0;
6    }
7    int P = ((blockIdx.x + gridDim.x * blockIdx.y) % 8) * (64 / blockDim.x);
8    for(int j = P; j < n / blockDim.x; j++) {
9      for(int i = 0; i < n_ele / blockDim.y; i++) {
10       As[threadIdx.y + i * blockDim.y][threadIdx.x] = A[threadIdx.x +
     ↳ (blockIdx.y * n_ele + threadIdx.y) * n + (i * blockDim.y * n + j *
     ↳ blockDim.x)];
11     }
12     __syncthreads();
13     for(int i = 0; i < blockDim.x; i++) {
14       for(int k = 0; k < n_ele; k++) {
15         temp[k] += As[k][i] * B[(blockIdx.x * blockDim.x * blockDim.y +
     ↳ threadIdx.y * blockDim.x + threadIdx.x) + (i * n + j * n *
     ↳ blockDim.x)];
16       }
17     }
18     __syncthreads();
19   }
20   for(int j = 0; j < P; j++) {
21     for(int i = 0; i < n_ele / blockDim.y; i++) {
22       As[threadIdx.y + i * blockDim.y][threadIdx.x] = A[threadIdx.x +
     ↳ (blockIdx.y * n_ele + threadIdx.y) * n + (i * blockDim.y * n + j *
     ↳ blockDim.x)];
23     }
24     __syncthreads();
25     for(int i = 0; i < blockDim.x; i++) {
26       for(int k = 0; k < n_ele; k++) {
27         temp[k] += As[k][i] * B[(blockIdx.x * blockDim.x * blockDim.y +
     ↳ threadIdx.y * blockDim.x + threadIdx.x) + (i * n + j * n *
     ↳ blockDim.x)];
28       }
29     }
30     __syncthreads();
31   }
32   for (int i = 0; i < n_ele; i++) {
33     C[i * n + blockIdx.x * blockDim.x * blockDim.y + threadIdx.y *
         ↳ blockDim.x + threadIdx.x + blockIdx.y * n_ele * n] = temp[i];
34   }
35 }
```

Listing 4.12: The shared memory implementation of MM on a GPU utilizing the optimized computation and access pattern.

27, each HW reads the same value, row k and column i. Since the T10 GPU supports shared memory broadcasting, threads within a HW accessing the same address in shared memory cause no bank conflicts. Therefore, no further optimizations to the access pattern are necessary.

Figure 4.17 depicts maximum effective bandwidth of MM before (Listing 4.10) and after (Listing 4.12) utilizing the optimized access pattern. For MM, effective bandwidth is calculated in Section 3.5 as $\frac{8n^3 bytes}{T^{GPU}_{comp.}}$.

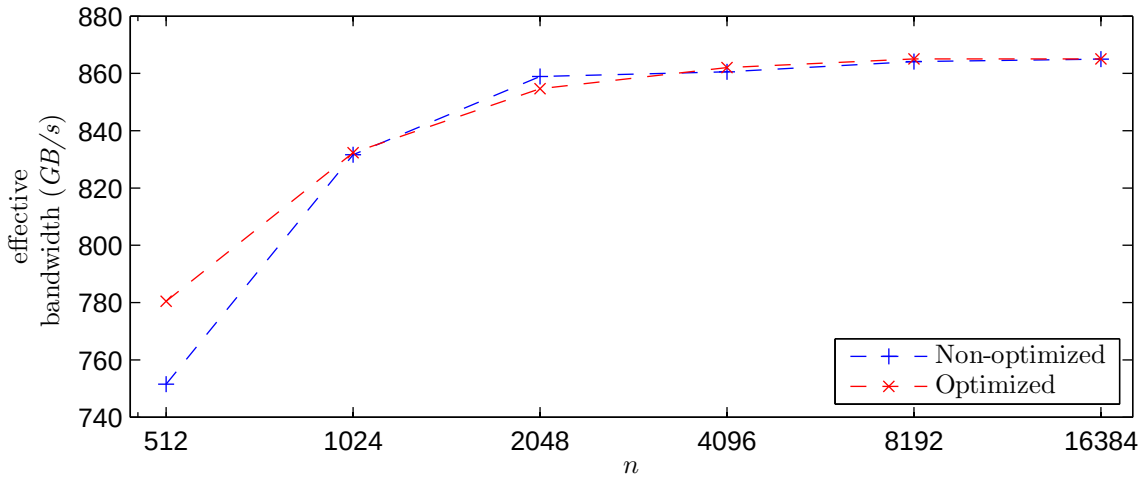


Figure 4.17: Comparison of access patterns: Maximum effective bandwidth (GB/s) for MM on the T10 GPU.

The optimized access pattern yields a speedup, on average, of 1.01 compared to the non-optimized pattern. The minimum (n = 2048) and maximum (n = 512) speedups are 1.00 and 1.04, respectively. Comparing results from Figure 4.16 with Figure 4.17, the optimized access pattern yields a significant speedup for Mv but a modest speedup for MM. Since bank conflicts do not occur for MM, these results suggest that bank conflicts affect computation time more significantly than partition camping. Regardless, the optimized access pattern increases maximum effective bandwidth, on average, for both. Therefore, Step 3 of the parallelization procedure, deriving the optimized access pattern, is valid and necessary.

4.4 Fine-tuning

The fourth step of the parallelization procedure specifies that fine-tuning is performed on each kernel to reduce computation and resource allocation. Fine-tuning is defined as optimizations to the kernel to reduce time, such as reducing index calculations and loop unrolling. Reducing index calculations, particularly in inner loops, reduces the time necessary to issue global memory transactions thus reducing the computation time.

Loop unrolling also reduces computation time by increasing the instruction mix of memory transactions and computation instructions. In addition, loop unrolling can reduce the number of registers used per thread since the compiler can reuse registers. When the compiler exceeds the maximum register allocation per thread, local memory is allocated. Local memory is a section of global memory reserved for each thread as storage for temporary variables. The usage of local memory instead of registers greatly increases the computation time as global memory accesses are orders of magnitude slower than register accesses. Since the implementation of MM utilizes many registers, the effect is more apparent than for Mv. Since loop unrolling can reduce the number of registers used per thread due to register reuse, it can eliminate allocation of local memory thus greatly reducing the computation time.

For matrix-based computations, index calculations are necessary for each thread to determine which value of each matrix is accessed. Index calculations typically consist of several multiply and addition instructions inside a loop. The kernel definitions of matrix-based computations specify that pointers to each matrix are passed. Since each thread contains a pointer to each matrix, some of the multiplication instructions can be reduced to addition. Listing 4.13 depicts sample code for a typical matrix-based computation. In

```
1 __global__ void Example(float* A, float* B, int n) {
2   for(int i = 0; i < n; i++) {
3     for(int j = 0; j < n; j++) {
4       B[threadIdx.y * n + j] = A[i * n + threadIdx.x];
5     }
6   }
7 }
```

Listing 4.13: An example of a matrix-based computation without fine-tuning.

Line 4, to calculate a part of the index of **B**, each thread calculates $threadIdx.y * n$ which is performed n^2 times. Since the value is constant for each thread, this calculation can be performed once before the *for*-loops. Likewise, each thread adds $threadIdx.x$ which is calculated n^2 times for a part of the index of **A**. This calculation can also be performed once outside of the *for*-loops. Similarly, each thread calculates $i * n$ which is performed n^2 times for a part of the index of **A**. Therefore, this calculation can be replaced by adding n to the pointer to **A** inside the outer *for*-loop. Now, each thread performs only n additions of n. The kernel in Listing 4.13 is modified accordingly to yield the fine-tuned kernel, with respect to index calculations, in Listing 4.14.

```
1  __global__ void Example(float* A, float* B, int n) {
2    A += threadIdx.x;
3    B += threadIdx.y * n;
4    for(int i = 0; i < n; i++) {
5      for(int j = 0; j < n; j++) {
6        B[j] = A[0];
7      }
8      A += n;
9    }
10 }
```

Listing 4.14: An example of a matrix-based computation with index calculations minimized.

As mentioned, loop unrolling can reduce computation time by increasing the instruction mix and reducing the number of registers used per thread. Reducing the number of registers used per thread can significantly reduce computation time since the number of active threads can be dependent on register usage. Therefore, NVIDIA provides a *#pragma unroll* directive to instruct the compiler to unroll a loop. From the CUDA programming guide [64], "By default, the compiler unrolls small loops with a known trip count." However, it is undefined as to what constitutes a small loop. Therefore, *#pragma unroll* is inserted before each loop in the kernel. The compiler unrolls inner loops first and stops unrolling loops once the maximum instruction limit is reached. Listing 4.14 is modified to instruct the compiler to unroll the *for*-loops as depicted in Listing 4.15.

```
__global__ void Example(float* A, float* B, int n) {
  A += threadIdx.x;
  B += threadIdx.y * n;
  #pragma unroll
  for(int i = 0; i < n; i++) {
    #pragma unroll
    for(int j = 0; j < n; j++) {
      B[j] = A[0];
    }
    A += n;
  }
}
```

Listing 4.15: An example of a matrix-based computation after fine-tuning. Index calculations are minimized and all loops are unrolled

4.4.1 Mv

Fine-tuning is performed to the shared memory implementation of Mv on a GPU utilizing the optimized computation and access pattern, Listing 4.11, to yield Listing 4.16.

```
__constant__ float b[n];
__global__ void Mv(float* A, float* c, int n) {

  __shared__ float As[blockDim.x][blockDim.x+1];

  float temp = 0.0;
  int index = blockIdx.x * blockDim.x;
  int P = (blockIdx.x  % 8) * 64;
  int calc1 = blockDim.x / blockDim.y;
  int calc2 = calc1 * threadIdx.y;

  A += threadIdx.x + index * n + calc2 * n + P;
  c += index + threadIdx.x;

  #pragma unroll
  for(int j = P; j < n; j+= blockDim.x) {
    #pragma unroll
    for(int k = 0; k < calc1; k++) {
      As[k + calc2][threadIdx.x] = A[0];
      A += n;
    }
    A += blockDim.x;
    A -= n * calc1;
    __syncthreads();

    #pragma unroll
    for(int k = 0; k < calc1; k++) {
      temp += As[threadIdx.x][k + calc2] * b[j + k + calc2];
    }
    __syncthreads();
  }
  A -= n;

  #pragma unroll
```

```
29    for(int j = 0; j < P; j+= blockDim.x) {
30      #pragma unroll
31      for(int k = 0; k < calc1; k++) {
32        As[k + calc2][threadIdx.x] = A[0];
33        A += n;
34      }
35      A += blockDim.x;
36      A -= n * calc1;
37      __syncthreads();

38      #pragma unroll
39      for(int k = 0; k < calc1; k++) {
40        temp += As[threadIdx.x][k + calc2] * b[j + k + calc2];
41      }
42      __syncthreads();
43    }

44    if(threadIdx.y != 0) {
45      As[threadIdx.y-1][threadIdx.x] = temp;
46    }
47    __syncthreads();
48    if(threadIdx.y == 0) {
49      #pragma unroll
50      for(k = 0; k < blockDim.y-1; k++) {
51        temp += As[k][threadIdx.x];
52      }
53      c[0] = temp;
54    }
55  }
```

Listing 4.16: The shared memory implementation of Mv on a GPU utilizing the optimized computation and access pattern after fine-tuning. Constant memory is utilized for **b**.

No modifications are performed to the allocation of shared memory thus Equation (4.11) defines shared memory usage. Since modifications are performed only to index calculations and loop unrolling, the number of global memory reads is defined by Equations (4.6) and (4.7). *calc*1 and *calc*2 in Lines 7 and 8 are commonly used calculations in the *for*-loops. By defining temporary variables for the commonly used calculations, the amount of computation in the *for*-loops is reduced. Index calculations are reduced by initializing pointers to **A** and **c** in Lines 9 and 10. Likewise, pointer arithmetic is performed in Lines 16, 18, 19, 27, 33, 35, and 36 to further reduce index calculations.

In Listing 4.16, *for*-loops are preceded with the *#pragma unroll* directive in Lines 11, 13, 21, 28, 30, 38, and 49. This instructs the compiler to unroll as many loops as possible to increase the instruction mix and reduce the number of registers used per thread.

Figure 4.18 depicts maximum effective bandwidth of Mv before (Listing 4.11) and after (Listing 4.16) fine-tuning. As mentioned, effective bandwidth is calculated as $\frac{8n^2 bytes}{T^{GPU}_{comp.}}$.

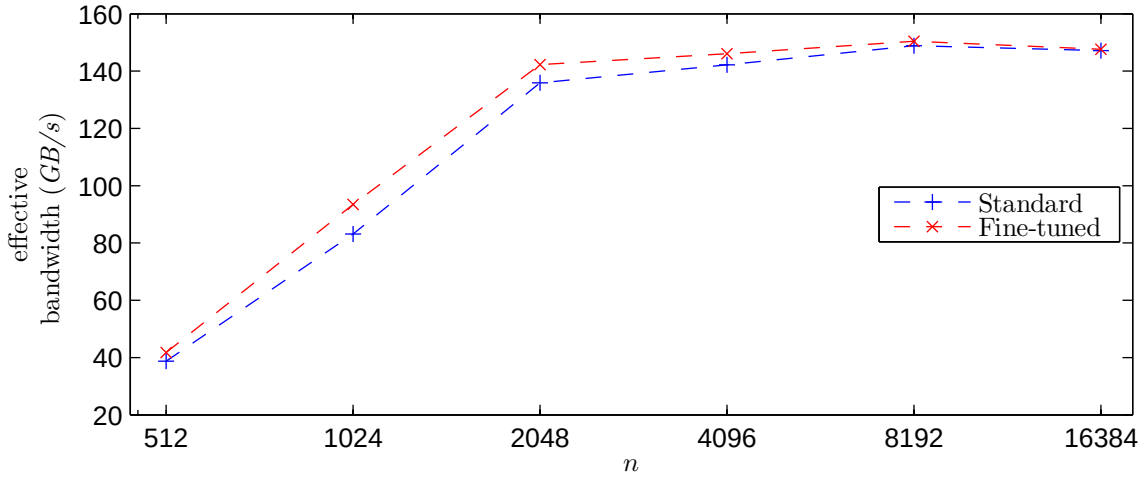


Figure 4.18: Comparison of fine-tuning: Maximum bandwidth (GB/s) for Mv on the T10 GPU.

In the figure, the kernel, after fine-tuning, yields a speedup, on average, of 1.05 compared to before fine-tuning. The minimum (n = 16384) and maximum (n = 1024) speedups are 1.00 and 1.12, respectively. As mentioned, the usage of local memory instead of registers greatly increases the computation time as global memory accesses are orders of magnitude slower than register accesses. However, for Mv, a small amount of registers are allocated and therefore, local memory is not utilized. Therefore, fine-tuning Mv yields a reduction in only the amount of index calculations and thus a modest speedup. Regardless, the maximum effective bandwidth is increased and therefore, Step 4, fine-tuning the kernel, is valid for Mv.

4.4.2 MM

Fine-tuning is performed to the shared memory implementation of MM on a GPU utilizing the optimized computation and access pattern, Listing 4.12, to yield Listing 4.17. In this kernel, $dBlk.y \leq n\_ele$. Slight modifications are performed to Lines 15-18 and 37-40 for loading **A** if $dBlk.y > n\_ele$.

```
1  __global__ void MM( float *A, float *B, float *C, int n) {
2    __shared__ float As[n_ele][blockDim.x];
3    float temp[N_ele];
4    #pragma unroll
5    for(int i = 0; i < n_ele; i++) {
6      temp[i] = 0.0;
7    }
8    int P = ((blockIdx.x + gridDim.x * blockIdx.y) % 8) * (64 / blockDim.x);
9    A += P * blockDim.x + threadIdx.x + (blockIdx.y * N_ele + threadIdx.y)
       ↳ * n;
10   B += P * n * blockDim.x + blockIdx.x * blockDim.x * blockDim.y +
       ↳ threadIdx.y * blockDim.x + threadIdx.x;
11   C += threadIdx.x + blockIdx.x * blockDim.x * blockDim.y + threadIdx.y *
       ↳ blockDim.x + blockIdx.y * N_ele * n;
12   #pragma unroll
13   for(int j = P; j < n / blockDim.x; j++) {
14     #pragma unroll
15     for(int i = 0; i < n_ele / blockDim.y; i++) {
16       As[threadIdx.y + i * blockDim.y][threadIdx.x] = A[0];
17       A += blockDim.y * n;
18     }
19     A -= blockDim.y * n * N_ele / blockDim.y;
20     __syncthreads();
21     #pragma unroll
22     for(int i = 0; i < blockDim.x; i++) {
23       #pragma unroll
24       for(int k = 0; k < n_ele; k++) {
25         temp[k] += As[k][i] * B[0];
26       }
27       B += n;
28     }
29     A += blockDim.x;
30     __syncthreads();
31   }
32   A -= n;
33   B -= n * n;
34   #pragma unroll
35   for(int j = 0; j < P; j++) {
36     #pragma unroll
37     for(int i = 0; i < n_ele / blockDim.y; i++) {
38       As[threadIdx.y + i * blockDim.y][threadIdx.x] = A[0];
39       A += blockDim.y * n;
40     }
41     A -= blockDim.y * n * N_ele / blockDim.y;
42     __syncthreads();
43     #pragma unroll
44     for(int i = 0; i < blockDim.x; i++) {
45       #pragma unroll
```

```
46        for(int k = 0; k < n_ele; k++) {
47          temp[k] += As[k][i] * B[0];
48        }
49        B += n;
50      }
51      A += blockDim.x;
52      __syncthreads();
53    }

54    #pragma unroll
55    for (int i = 0; i < n_ele; i++) {
56      C[0] = temp[i];
57      C += n;
58    }
59  }
```

Listing 4.17: The shared memory implementation of MM on a GPU utilizing the optimized computation and access pattern after fine-tuning.

No modifications are performed to the allocation of shared memory, thus Equation (4.8) defines shared memory usage. Since modifications are performed only to index calculations and loop unrolling, the number of global memory reads is defined by Equations (4.9) and (4.10). Index calculations are reduced by initializing pointers to **A**, **B**, and **C** in Lines 9-11. Likewise, pointer arithmetic is performed in Lines 17, 19, 27, 29, 32, 33, 39, 41, 49, 51, and 57 to further reduce index calculations.

The *for*-loops in Listing 4.17 are preceded with the *#pragma unroll* directive in Lines 4, 12, 14, 21, 23, 34, 36, 43, 45, and 54 to increase the instruction mix and reduce the number of registers used per thread. However, the *for*-loops in Lines 13 and 35 are dependent on P which is calculated in Line 8 and dependent on block indices. Since P is calculated at run-time, its value is unknown during compilation. Therefore, the *for*-loops in Lines 13 and 35 are not unrolled by the compiler since the trip count is unknown.

Figure 4.19 depicts maximum effective bandwidth of MM before (Listing 4.12) and after (Listing 4.17) fine-tuning. As discussed, maximum effective bandwidth is $\frac{8n^3 bytes}{T^{GPU}_{comp.}}$ for MM.

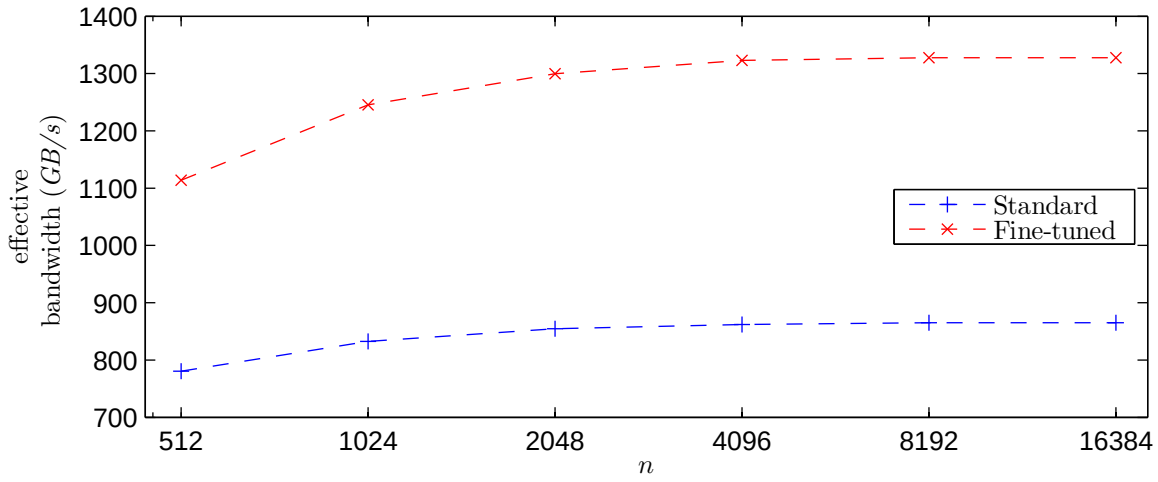


Figure 4.19: Comparison of fine-tuning: Maximum effective bandwidth (GB/s) for MM on the T10 GPU.

For MM, register utilization is high since each thread is computing multiple results and utilizing a register for each partial result. As shown in Figure 4.19 the kernel after fine-tuning yields a speedup, on average, of 1.51 compared to before fine-tuning. The minimum (n = 512) and maximum (n = 16384) speedups are 1.43 and 1.53, respectively. Since fine-tuning the kernel reduces the amount of register utilization, a significant improvement in maximum effective bandwidth is realized for MM. Therefore, Step 4 of the parallelization procedure, fine-tuning the kernel, is necessary and valid for MM.

4.5 Input Parameters

The penultimate step of the parallelization procedure, Step 5, specifies that optimal input parameters are derived. Input parameters are defined as the dimensions of the grid and blocks assigned for execution on the GPU. Through analysis of measured data for Mv and MM, input parameters significantly affect the GPU computation time. Therefore, optimal input parameters are derived as the input parameters which yield the minimum computation time. The derivation is performed utilizing the execution metrics formulated in Section 3.3. Since input parameters are dependent on the kernel, the derivation is performed after the

kernel is written, which considers the placement of data, optimized computation and access patterns, and fine-tuning adjustments.

A list of five steps is presented, in order, to derive the optimal input parameters:

1. Saturate the memory bus.
2. Maximize shared memory utilization.
3. Minimize the amount of shared memory used per block.
4. Maximize the number of global memory partitions accessed.
5. Minimize the number of fragmented blocks.

The order of deriving optimal input parameters is developed through analysis of measured data for Mv and MM. Step 1 is listed to maximize the overlap of computation and memory accesses by ensuring the memory bus is fully utilized. Step 2 ensures that data which is loaded into shared memory is reused as much as possible. Step 3 is listed such that a minimum amount of shared memory is assigned per block. This increases the number of blocks assigned to each SM. In addition, this reduces the amount of time each HW in a block must wait for synchronization since there are less HWs per block. Step 4 maximizes the number of global memory partitions accessed such that partition camping is reduced. Lastly, Step 5 ensures the contribution to computation time from fragmented blocks is minimized.

Due to limitations of the T10 GPU, a maximum of 512 threads can be assigned per block. Therefore,

$$dBlk.x \times dBlk.y \leq 512. \tag{4.12}$$

Since threads are assigned to HWs in row-major order and the minimum global memory transaction size is $32B$,

$$dBlk.x \geq 8. \tag{4.13}$$

Since values of type float are used, this equation ensures that one $32B$ transaction can service half of the threads in a HW.

Step 1 specifies that the memory bus is saturated which requires analysis of $\text{Thds}_{GPU}^{active}$ which is dependent on $\text{Blks}_{GPU}^{active}$. In the GPU documentation provided by NVIDIA [64], it is stated that each global memory access takes approximately 400-800 clock cycles. Therefore, there needs to exist a minimum of $\frac{400}{8}$ warps per partition, or 12800 $\text{Thds}_{GPU}^{active}$, to fully saturate the memory bus. Since the maximum value of $\text{Thds}_{GPU}^{active}$ is determined by the product of the number of SMs and the maximum number of threads per SM,

$$12800 \leq \text{Thds}_{GPU}^{active} \leq 30720 \tag{4.14}$$

Therefore, optimal input parameters are derived such that $\text{Thds}_{GPU}^{active}$ is greater than the minimum necessary to fully saturate the memory bus. Optimal input parameters for each matrix-based computation are derived in the following sections. Sections 4.5.1 and 4.5.2 include the derivation of optimal input parameters for Mv and MM, respectively.

4.5.1 Mv

In this section, optimal input parameters are derived for the kernel in Listing 4.16, which is the shared memory implementation of Mv on a GPU utilizing the optimized computation and access pattern after fine-tuning. The amount of shared memory allocated per block is defined by Equation (4.11). Since each SM has a maximum of $16KB$ of shared memory, the maximum value of $dBlk.x$ is 32. Combining with the minimum value of $dBlk.x$ from Equation (4.13) yields

$$8 \leq dBlk.x \leq 32. \tag{4.15}$$

From the kernel in Listing 4.16 and the maximum value of $dBlk.y$ determined from solving for $dBlk.y$ in Equation (4.12),

$$dBlk.y \leq \min\left(\frac{512}{dBlk.x}, dBlk.x\right) \tag{4.16a}$$

$$dGrd.x = \frac{n}{dBlk.x} \tag{4.16b}$$

$$dGrd.y = 1.$$

The number of registers used per thread for the kernel varies from 10 to 15 dependent on $dBlk.x$ and $dBlk.y$. Assuming the maximum is used, $Blks_{SM}^{active}$ from Equation (3.7) is not limited by register usage. Therefore, Equation (3.8) simplifies to

$$\text{Blks}_{SM}^{active} = \min\left(\left\lfloor\frac{16384}{\text{SMemPerBlk}}\right\rfloor, 8, \frac{1024}{dBlk.x \times dBlk.y}\right). \tag{4.17}$$

Since $dGrd.y = 1$, Equation (3.8) simplifies to

$$\text{Blks}_{GPU}^{active} = \min\left(30 \times \text{Blks}_{SM}^{active}, dGrd.x\right).$$

Since

$$\text{Thds}_{GPU}^{active} = \text{Blks}_{GPU}^{active} \times dBlk.x \times dBlk.y, \tag{4.18}$$

then, if $dGrd.x \leq \text{Blks}_{GPU}^{active}$,

$$\text{Thds}_{GPU}^{active} = dGrd.x \times dBlk.x \times dBlk.y.$$

Substituting $dGrd.x = \frac{n}{dBlk.x}$ from Equation (4.16b) and the lower limit of n defined by Equation (4.1) yields

$$\text{Thds}_{GPU}^{active} \geq 512 \times dBlk.y.$$

Since $dBlk.x \leq 32$, the maximum value of $dBlk.y$ from Equation (4.16a) yields

$$\text{Thds}_{GPU}^{active} \geq 8192.$$

Substituting into Equation (4.18) and substituting for $\text{Blks}_{GPU}^{active}$ yields

$$8192 \leq \min\left(30 \times \text{Blks}_{SM}^{active}, dGrd.x\right) \times dBlk.x \times dBlk.y. \tag{4.19}$$

If $dGrd.x \leq 30 \times \text{Blks}_{SM}^{active}$, rearranging Equation (4.19) and solving for $dBlk.y$ yields the lower limit as

$$\frac{8192}{n} \leq dBlk.y. \tag{4.20}$$

The upper limit of $dBlk.y$ is not modified and therefore is defined by Equation (4.16a).

However, if $dGrd.x > 30 \times \text{Blks}_{SM}^{active}$, then substituting for $\text{Blks}_{SM}^{active}$ from Equation (4.17) into Equation (4.19) yields

$$8192 \leq 30 \times \min\left(\left\lfloor \frac{16384}{\text{SMemPerBlk}} \right\rfloor, 8, \frac{1024}{dBlk.x \times dBlk.y}\right) \times dBlk.x \times dBlk.y. \tag{4.21}$$

From Equation (4.11), SMemPerBlk = $(dBlk.x^2 + dBlk.x) \times 4$. After substituting SMemPerBlk into Equation (4.21), solving for $dBlk.y$, given the possible values of $dBlk.x$ from Equation (4.15), yields the lower limit as

$$\max\left(4, \frac{64}{dBlk.x}\right) \leq dBlk.y. \tag{4.22}$$

Combining lower limits of $dBlk.y$ from Equations (4.20) and (4.22) with the upper limit from Equation (4.16a) yields

$$\max\left(\frac{8192}{n}, 4, \frac{64}{dBlk.x}\right) \leq dBlk.y \leq \min\left(\frac{512}{dBlk.x}, dBlk.x\right). \tag{4.23}$$

Step 2 of deriving optimal input parameters specifies shared memory utilization is maximized to ensure data is reused as much as possible. From Lines 22 and 39 of the kernel in Listing 4.16, the trip count for the inner loops where shared memory is accessed is dependent on $\frac{dBlk.x}{dBlk.y}$. Therefore, the maximum of $dBlk.x$ from Equation (4.15) and the minimum of $dBlk.y$ from Equation (4.23) yield

$$\begin{aligned} dBlk.x &= 32 \\ dBlk.y &= \max\left(\frac{8192}{n}, 4\right) \\ dGrd.x &= \frac{n}{32} \\ dGrd.y &= 1. \end{aligned} \tag{4.24}$$

Given n, all input parameters are constant and therefore, no further steps are necessary to derive optimal input parameters for Mv. However, as realized through measured data for MM, additional steps are necessary to derive optimal input parameters for MM. Regardless, Equation (4.24) is the derivation of optimal input parameters for the shared memory implementation of Mv on a GPU utilizing the optimized computation and access pattern after fine-tuning.

Figure 4.20 depicts measured computation time of all input parameters for GPU computation of Mv. The figure illustrates the necessity of optimal input parameters as the GPU computation time of Mv varies significantly depending on the input parameters. In the figures in this section, the x-axis represents the number of threads per block ($dBlk.x \times dBlk.y$). Therefore, there are several combinations of $dBlk.x$ and $dBlk.y$ which yield an equivalent number of threads per block.

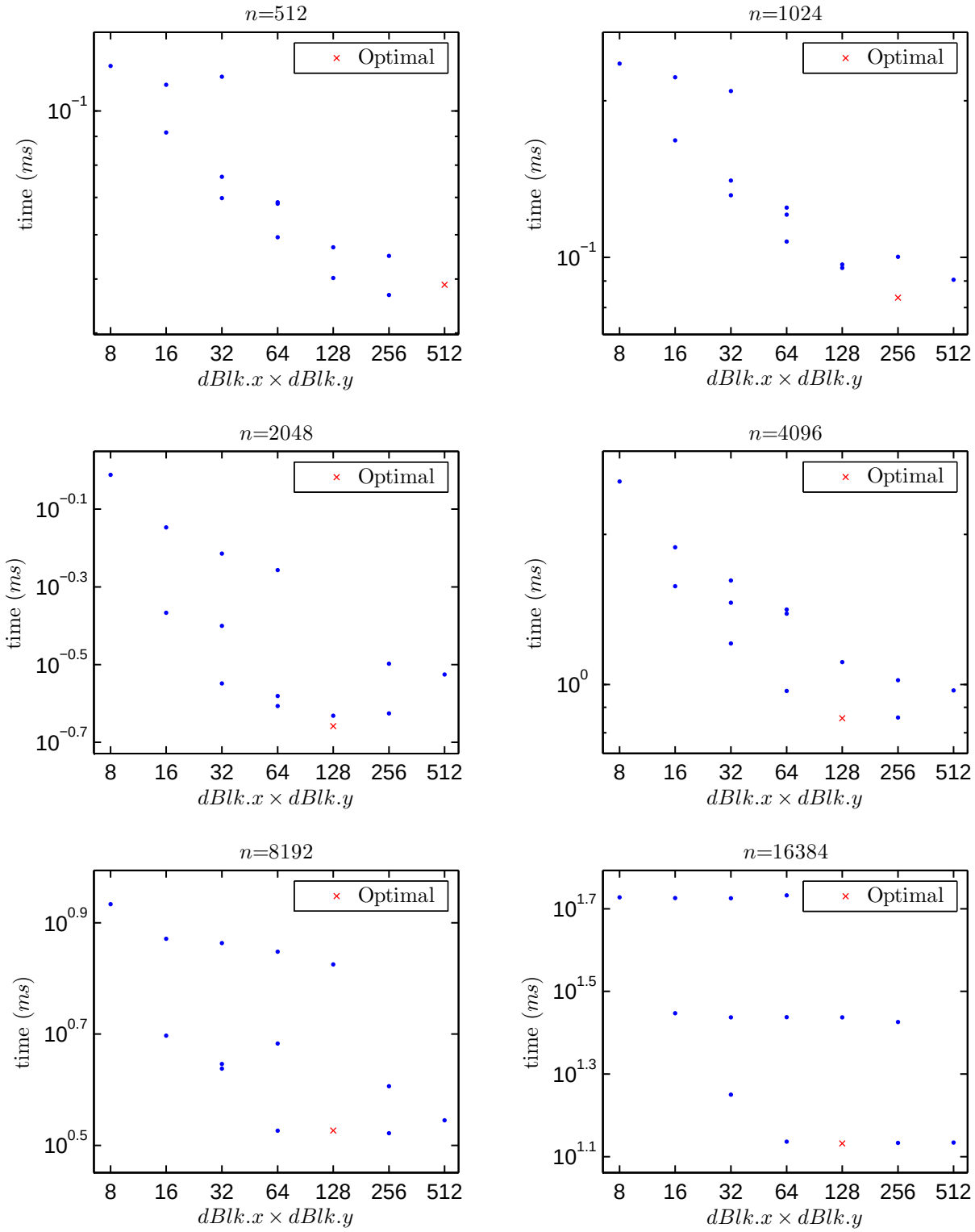


Figure 4.20: Comparison of input parameters: Computation time (ms) of all input parameters for Mv on the T10 GPU.

As illustrated, derivation of optimal input parameters yields computation time, on average, within 1.8% of the minimum measured time. In the worst-case (n = 8192), utilizing optimal input parameters yields time within 4.1% of the minimum. Best-case utilization of optimal input parameters ($n = \{1024, 2048, 16384\}$) yields the minimum. Therefore, Step 5 of the parallelization procedure, deriving optimal input parameters, yields minimum or near-minimum GPU computation times for Mv.

4.5.2 MM

In this section, optimal input parameters are derived for the kernel in Listing 4.17, which is the shared memory implementation of MM on a GPU utilizing the optimized computation and access pattern after fine-tuning. From the kernel and Equations (4.12) and (4.13),

$$8 \leq dBlk.x \leq 512 \tag{4.25a}$$

$$1 \leq dBlk.y \leq \frac{512}{dBlk.x} \tag{4.25b}$$

$$dGrd.x = \frac{n}{dBlk.x \times dBlk.y} \tag{4.25c}$$

$$dGrd.y = \frac{n}{n\_ele} \tag{4.25d}$$

where $1 \leq n\_ele \leq 16$ due to resource constraints. From Equation (4.8), SMemPerBlk = $(n\_ele \times dBlk.x) \times 4$. From compilation of the kernel, RegsPerThd varies from 14 to 38.

From the minimum of SMemPerBlk and RegsPerBlk, $dGrd.x \times dGrd.y > 30 \times \text{Blks}_{SM}^{active}$. Therefore, Equation (3.8) simplifies to

$$\text{Blks}_{GPU}^{active} = 30 \times \min\left(\left\lfloor \frac{16384}{\text{RegsPerBlk}} \right\rfloor, \left\lfloor \frac{16384}{\text{SMemPerBlk}} \right\rfloor, 8, \frac{1024}{dBlk.x \times dBlk.y}\right). \tag{4.26}$$

Substituting $\text{Blks}_{GPU}^{active}$ from Equation (4.26) and the minimum of $\text{Thds}_{GPU}^{active}$ from Equation (4.14) into Equation (3.9) yields

$$12800 \leq 30 \times \min\left(\left\lfloor \frac{16384}{\text{RegsPerBlk}} \right\rfloor, \left\lfloor \frac{16384}{\text{SMemPerBlk}} \right\rfloor, 8, \frac{1024}{dBlk.x \times dBlk.y}\right) \times dBlk.x \times dBlk.y. \tag{4.27}$$

Solving for $dBlk.y$ modifies the lower limit of $dBlk.y$ from Equation (4.25b) to yield

$$\max\left(\frac{12800}{30dBlk.x\left\lfloor \frac{16384}{\text{RegsPerBlk}} \right\rfloor}, \frac{12800}{30dBlk.x\left\lfloor \frac{16384}{(n\_ele \times dBlk.x)\times 4} \right\rfloor}, \frac{64}{dBlk.x}\right) \leq dBlk.y \leq \frac{512}{dBlk.x}. \tag{4.28}$$

Step 2 of deriving optimal input parameters specifies shared memory utilization is maximized to ensure data is reused as much as possible. From Lines 25 and 47 of the kernel in Listing 4.17, the trip count for the inner loops where shared memory is accessed is dependent on n_ele. Since $1 \leq n\_ele \leq 16$, $n\_ele = 16$. Substituting $n\_ele = 16$ into Equation (4.25d) yields

$$dGrd.y = \frac{n}{16} \tag{4.29}$$

Since $\text{SMemPerBlk} = (n\_ele \times dBlk.x) \times 4$, $dBlk.x \leq 64$. However, if $dBlk.x = 64$ and $dBlk.y$ is the maximum from Equation (4.28), then $\text{Blks}_{GPU}^{active} = 30$ due to register usage. Therefore, $\text{Thds}_{GPU}^{active} = 7680$ and is less than the minimum from Equation (4.14). Therefore, $dBlk.x \leq 32$. However, if $dBlk.x \geq 16$, due to register usage, only two values exist for $dBlk.y$ which satisfy Equation (4.28). Therefore, Equation (4.28) simplifies to

$$\frac{64}{dBlk.x} \leq dBlk.y \leq \frac{512}{dBlk.x} \quad \text{if } dBlk.x = 8. \tag{4.30}$$

and

$$dBlk.y = \begin{cases} 32 & \text{if } dBlk.x = 16 \\ 2 & \text{if } dBlk.x = 32. \end{cases}$$

Step 3 of deriving optimal input parameters specifies the minimum amount of shared memory is allocated per block to increase the number of blocks assigned to each SM. In addition, minimizing the amount of shared memory allocated reduces time each HW in a block waits for synchronization since there are fewer HWs per block. Since n_ele = 16, SMemPerBlk = $64dBlk.x$. Therefore, the minimum value of $dBlk.x$ from Equation (4.25a) is utilized. Since $dBlk.x$ = 8, Equations (4.30) and (4.25c) are evaluated to yield

$$8 \leq dBlk.y \leq 64 \tag{4.31a}$$

$$dGrd.x = \frac{n}{8dBlk.y} \tag{4.31b}$$

Step 4 of deriving optimal input parameters specifies the maximum number of global memory partitions, PartsPerGPU, are accessed to reduce partition camping. PartsPerGPU is dependent on PartsPerBlk, $dGrd.x$, and $\text{Blks}_{GPU}^{active}$. Since $dBlk.x$ and n_ele are constant, RegsPerThd is constant. From compilation, 32 RegsPerThd are allocated so RegsPerBlk = $256dBlk.y$. Since RegsPerBlk is the only limiting factor of Equation (4.26), the equation simplifies to

$$\text{Blks}_{GPU}^{active} = \frac{1920}{dBlk.y}. \tag{4.32}$$

If $dGrd.x > \text{Blks}_{GPU}^{active}$, substituting $dBlk.x$ = 8 and $\text{Blks}_{GPU}^{active}$ from Equation (4.32) into Equation (3.13) and solving for $dBlk.y$ yields[4]

$$dBlk.y \leq 16. \tag{4.33}$$

If $dGrd.x \leq \text{Blks}_{GPU}^{active}$, substituting $dBlk.x$ = 8 and $dGrd.x$ from Equation (4.31b) into Equation (3.13) and simplifying yields

$$\text{PartsPerGPU} = \min\left(\frac{n}{64dBlk.y}, 8\right). \tag{4.34}$$

[4]The solution is $dBlk.y \leq 30$. However, $dBlk.y$ is a power of two so it is rounded down to the nearest power of two.

Substituting the minimum value of $dBlk.y$ from Equation (4.31a) into Equation (4.34) yields

$$dGrd.x\frac{dBlk.x}{64} \geq \min\left(\frac{n}{512}, 8\right). \tag{4.35}$$

Substituting $dBlk.x = 8$ and $dGrd.x$ from Equation (4.31b) into Equation (4.35) and solving for $dBlk.y$ yields

$$dBlk.y \leq \frac{n}{64\min\left(\frac{n}{512}, 8\right)}. \tag{4.36}$$

Combining the upper limits of $dBlk.y$ from Equations (4.33) and (4.36) with the lower limit of $dBlk.y$ from Equation (4.31a) yields

$$8 \leq dBlk.y \leq \min\left(16, \frac{n}{64\min\left(\frac{n}{512}, 8\right)}\right). \tag{4.37}$$

Step 5 of deriving optimal input parameters specifies the number of fragmented blocks, $\text{Blks}_{GPU}^{frag.}$ from Equation (3.10), is minimized. Substituting $dGrd.x$ from Equation (4.31b), $dGrd.y$ from Equation (4.29), and $\text{Blks}_{GPU}^{active}$ from Equation (4.32) into Equation (3.10) yields

$$\frac{n^2}{128dBlk.y} \bmod \frac{1920}{dBlk.y}. \tag{4.38}$$

In general, $\frac{x}{y}$ can be expressed as $x = Qy + R$ where Q is the quotient, $Q = \left\lfloor\frac{x}{y}\right\rfloor$, and R is the remainder. Since $\text{Blks}_{GPU}^{frag.} = R$, $\text{Blks}_{GPU}^{frag.} = x - Qy$. Substituting for x, y, and Q from Equation (4.38) and simplifying yields

$$\text{Blks}_{GPU}^{frag.} = \frac{n^2}{128dBlk.y} - \left\lfloor\frac{n^2}{245760}\right\rfloor\frac{1920}{dBlk.y}. \tag{4.39}$$

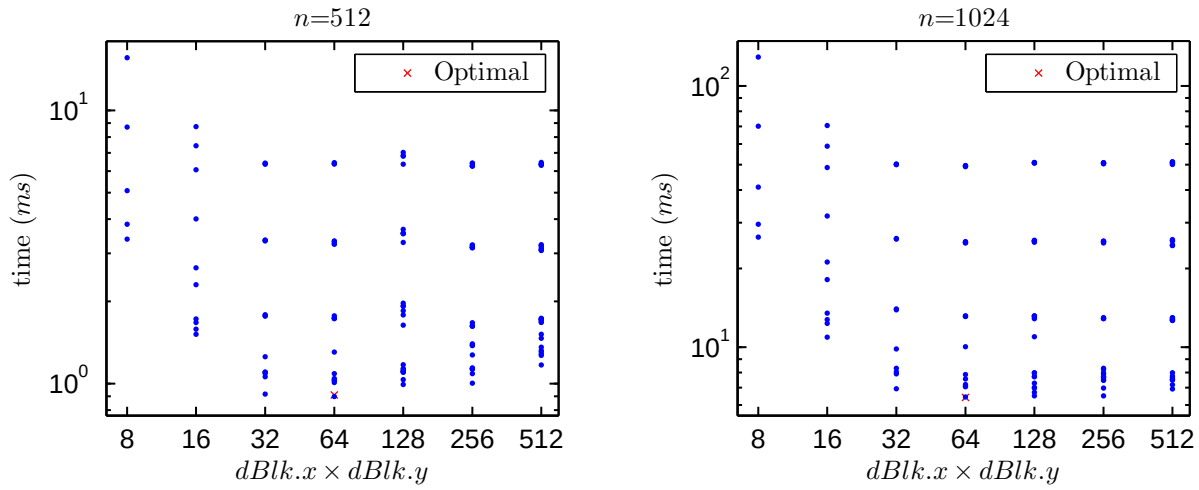
Therefore, to minimize $\text{Blks}_{GPU}^{frag.}$, the maximum value of $dBlk.y$ from Equation (4.37) is utilized and

$$dBlk.y = \min\left(16, \frac{n}{64\min\left(\frac{n}{512}, 8\right)}\right). \tag{4.40}$$

Since $dBlk.x = 8$ and from Equations (4.40), (4.25c), and (4.29), the optimal input parameters for the shared memory implementation of MM on a GPU utilizing the optimized computation and access pattern after fine-tuning are

$$\begin{aligned} dBlk.x &= 8 \\ dBlk.y &= \min\left(16, \frac{n}{64\min\left(\frac{n}{512}, 8\right)}\right) \\ dGrd.x &= \frac{n}{8Blk.y} \\ dGrd.y &= \frac{n}{16}. \end{aligned} \tag{4.41}$$

Figure 4.21 depicts measured GPU computation time of all input parameters for MM. The figure illustrates the necessity of optimal input parameters as the computation time of MM varies significantly depending on the input parameters.



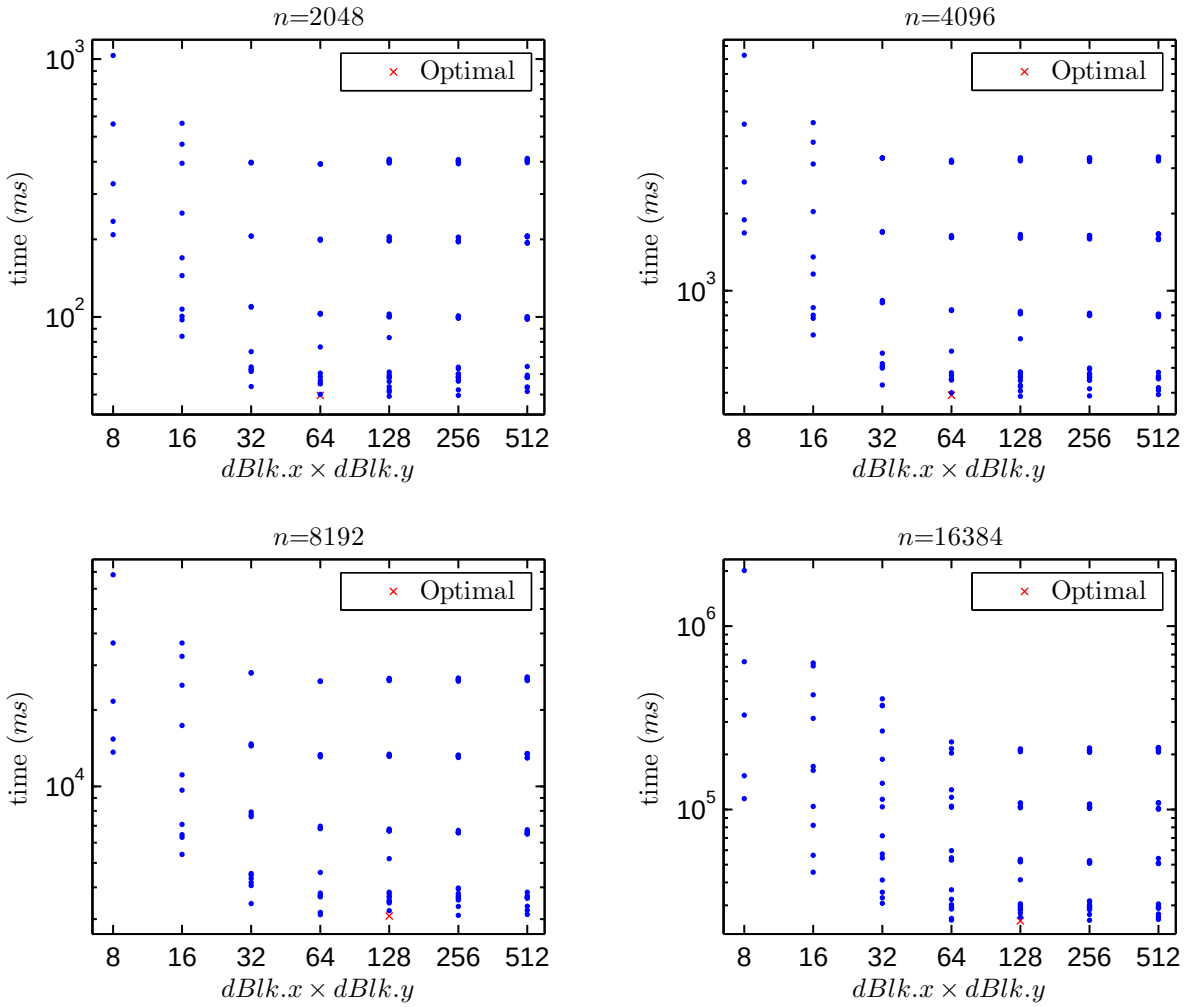


Figure 4.21: Comparison of input parameters: Computation time (ms) of all input parameters for MM on the T10 GPU.

Derivation of optimal input parameters yields computation time, on average, within 0.5% of the minimum measured time. In the worst-case (n = 512), utilizing optimal input parameters yields time within 1.2% of the minimum. Best-case utilization of optimal input parameters ($n = \{1024, 8192, 16384\}$) yields the minimum. As depicted, deriving optimal input parameters yields minimum or near-minimum GPU computation time for MM. Therefore, Step 5 of the parallelization procedure is valid and necessary for MM.

4.5.3 GPU Computation Summary

After Step 5, the parallelization procedure is developed to determine the optimal partitioning of computation between the CPU and GPU. Therefore, the procedure with regards to minimizing GPU computation time is complete. Therefore, this section illustrates maximum effective bandwidth of each aforementioned step of the parallelization procedure as it pertains to GPU computation time. In addition, a comparison of the procedure to theoretical bandwidth and CUBLAS effective bandwidth is illustrated. For each figure in this subsection, *Naïve* denotes the global memory implementation, *Sh. Mem.* denotes the shared memory implementation, *O.C.P.* denotes the optimized computation pattern utilizing shared memory, and *O.A.P.* denotes the optimized access pattern utilizing the optimized computation pattern and shared memory. *P.P.* denotes the entire parallelization procedure with regards to GPU computation which utilizes shared memory, optimized computation and access patterns, fine-tuning adjustments, and optimal input parameters.

Mv

Figure 4.22 depicts effective bandwidth of Mv for each step of the parallelization procedure in addition to CUBLAS effective bandwidth and theoretical bandwidth.

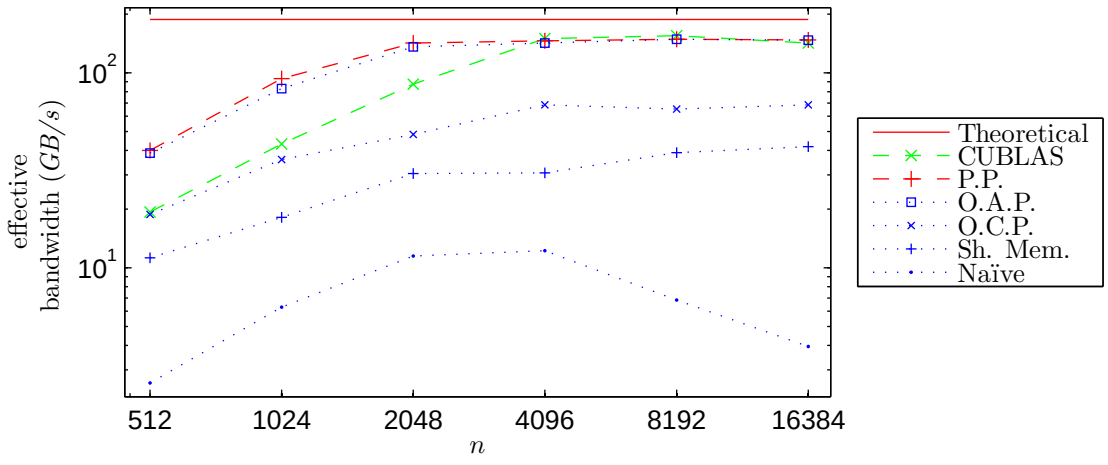


Figure 4.22: Comparison of effective bandwidth (GB/s) for Mv on the T10 GPU.

The parallelization procedure yields a speedup, on average, of 18.98 compared to the naïve implementation. The minimum (n = 4096) and maximum (n = 16384) speedups are 11.92 and 37.35, respectively. As illustrated, each step of the procedure increases the effective bandwidth. On average, the procedure yields a speedup of 1.47 compared to the CUBLAS implementation with a minimum (n = 8192) and maximum (n = 1024) speedup of 0.96 and 2.17, respectively. Compared to theoretical bandwidth, the procedure yields effective bandwidth, on average, as 63.7% of the theoretical bandwidth. Utilizing the procedure yields a worst-case (n = 512) effective bandwidth of 21.3% of theoretical. Best-case utilization of the procedure (n = 8192) yields an effective bandwidth of 79.1% of the theoretical. The procedure is verified to yield increases in speedup compared to the optimized CUBLAS GPU implementation of Mv. More importantly, the procedure is verified to yield significant increases in speedup compared to the naïve GPU implementation of Mv.

MM

Figure 4.23 depicts effective bandwidth of MM for each step of the parallelization procedure in addition to the theoretical and CUBLAS effective bandwidth.

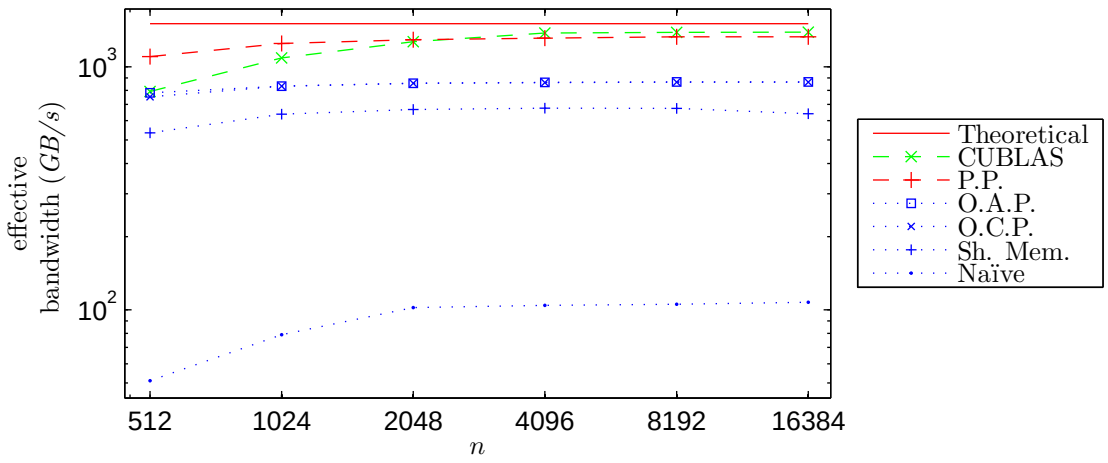


Figure 4.23: Comparison of effective bandwidth (GB/s) for MM on the T10 GPU.

Similar to Mv, the procedure applied to MM yields a significant speedup, on average, of 14.58 compared to the naïve implementation. The minimum (n = 16384) and maximum

(n = 512) speedups are 12.36 and 21.56, respectively. In Figure 4.23, each step of the procedure increases the effective bandwidth. The largest increase in bandwidth occurs after the first step of the procedure. This is due to the first step significantly reducing the number of global memory transactions. Compared to the CUBLAS implementations, the procedure yields a speedup, on average, of 1.07 with a minimum (n = 4096) and maximum (n = 512) speedup of 0.95 and 1.39, respectively. As illustrated, the procedure yields bandwidth near the theoretical. On average, the procedure yields 84.2% of the theoretical bandwidth. In the worst-case (n = 512), utilizing the procedure yields effective bandwidth of 73.2% of the theoretical. Best-case utilization of the procedure (n = 16384) yields an effective bandwidth of 88.3% of the theoretical. Similar to Mv, the procedure is verified to yield increases in speedup compared to the optimized CUBLAS GPU implementation of MM. Likewise, the procedure is verified to yield significant increases in speedup compared to the naïve GPU implementation of MM.

Therefore, for Mv and MM, the procedure yields GPU computation times competitive with the CUBLAS implementation and significantly lower than the naïve implementation.

4.6 Computation Partitioning

The last step, Step 6, of the parallelization procedure to minimize execution time is to determine the partitioning of computation between the CPU and GPU. For small matrix-based computations, the communication time may exceed the CPU computation time as measured for certain data sizes of Mv. Therefore, determining which computations are performed by the CPU and GPU requires accurate estimates of communication and computation times.

From accurate estimates of communication and computation times, it is possible to determine the appropriate computations to be performed by the CPU and GPU. The algorithm in Listing 4.18 determines which matrix-based computations are performed by the CPU and GPU. $T_{comp.}^{CPU}$ and $T_{comm.}$ are presented in Chapter 3.

```
1 if T_comp.^CPU ≤ T_comp.^GPU + T_comm.
2     perform computation on CPU.
3 if T_comp.^CPU > T_comp.^GPU + T_comm.
4     perform computation on GPU.
```

Listing 4.18: Algorithm to determine which computations are performed by the CPU and GPU.

Since GPU computation time is estimated assuming maximum bandwidth, it is necessary to determine the number of bytes accessed from global memory for each matrix-based computation. The theoretical bandwidth is partially calculated by dividing the number of bytes necessary for computation by the number of bytes accessed from global memory. This fraction is multiplied by the maximum bandwidth to global memory ($94GB/s$) to yield the theoretical bandwidth.

4.6.1 Mv

For Mv, optimal input parameters define $dBlk.x = 32$ from Equation (4.24). Therefore, from Equation (4.7), the number of $64B$ memory transactions is $4n^2$. Since $2n^2$ float values are necessary for computation and $4n^2$ bytes are accessed from global memory, the theoretical bandwidth for Mv is $\frac{8n^2}{4n^2}(94GB/s)$. Therefore, $T_{comp.}^{GPU} = \frac{\frac{8n^2}{1024^3}}{188}$. Communication requires two CPU to GPU transfers and one GPU to CPU transfer. Substituting for b, c_0, and c_1 into Equations (3.14) and (3.15) and simplifying yields the communication time in seconds as

$$T_{comm.} = 5.00 \times 10^{-9} n + \begin{cases} 3.02 \times 10^{-9} n^2 + 5.00 \times 10^{-5} & \text{if } n < 256 \\ 1.08 \times 10^{-9} n^2 + 9.60 \times 10^{-5} & \text{if } 256 \leq n < 512 \\ 1.08 \times 10^{-9} n^2 + 2.27 \times 10^{-4} & \text{if } n \geq 512. \end{cases}$$

From Lines 1-2 of Listing 4.18, computation is partitioned for CPU execution if $T_{comp.}^{CPU} \leq T_{comp.}^{GPU} + T_{comm.}$. Therefore, after substitution, if

$$\frac{\frac{8n^2}{1024^3}}{\min\left(\max\left(\frac{16384}{n}, 2\right), 8\right)} \leq \frac{\frac{8n^2}{1024^3}}{188} + T_{comm.},$$

computation is performed on the CPU to minimize execution time. Solving for n yields $T_{comp.}^{CPU} \leq T_{comp.}^{GPU} + T_{comm.}$ if $n \leq 2048$. Therefore, if $n > 2048$, the computation is performed on the GPU to minimize execution time.

Figure 4.24 illustrates measured execution time for Mv. CPU (ATLAS) execution time is measured time of the CPU utilizing ATLAS to perform Mv. GPU (CUBLAS) execution time is measured time for communication between the CPU and GPU in addition to measured GPU computation time utilizing CUBLAS.

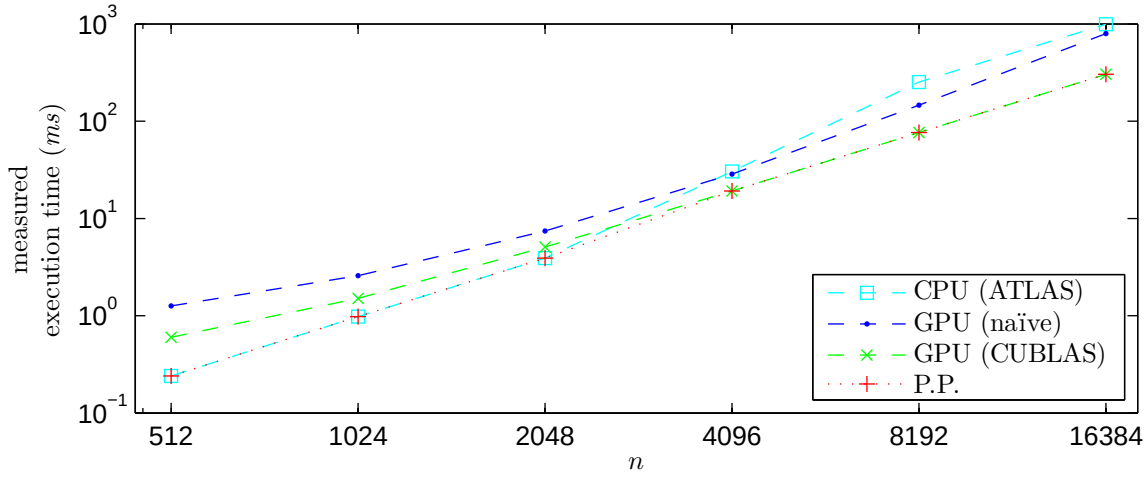


Figure 4.24: Comparison of execution time (ms) for Mv on the T10 GPU.

As mentioned, if $n \leq 2048$, Mv is performed on the CPU and therefore, the procedure yields execution time identical to the ATLAS implementation. If $n > 2048$, Mv is performed on the GPU utilizing the kernel developed in the procedure (Listing 4.16). The procedure yields a speedup, on average, of 1.68 compared to the ATLAS implementation, 2.63 compared to the naïve GPU implementation (Listing 4.2), and 1.39 compared to the CUBLAS implementation. The speedup compared to the naïve and CUBLAS implementations in Figure 4.24 is less than in Figure 4.22 due to communication time. For Mv, the communication time is a significant portion of execution time. Therefore, large speedups measured for Mv computation have less of an impact on execution time since the execution time includes communication and computation time.

From Figure 4.24, the procedure yields execution time, on average, within 0.8% of the theoretical minimum execution time. Utilizing the procedure yields a worst-case (n = 8192) time within 1.9% of the theoretical minimum. In the best-case ($n = \{512, 1024, 2048\}$), utilizing the procedure yields the theoretical minimum. This is possible since the theoretical minimum for $n \leq 2048$ is identical to the CPU utilizing ATLAS. This is because the theoretical minimum for the CPU is calculated utilizing effective bandwidths of the CPU. Therefore, Step 6 of the parallelization procedure, determine the optimal partitioning of computation, is valid and necessary for Mv.

4.6.2 MM

For MM, $2n^3$ float values are necessary for computation. If n_ele = 16 as defined in Section 4.5.2 (optimal input parameters), data is reused via shared memory. Since each thread in a HW reuses 16 float values, $\frac{8n^3}{16}$ bytes are necessary from global memory. Assuming the maximum global memory bandwidth of $94GB/s$, the theoretical bandwidth for MM is $1504GB/s$. Therefore, $T_{comp.}^{GPU} = \frac{\frac{8n^3}{1024^3}}{1504}$. Communication requires two CPU to GPU transfers and one GPU to CPU transfer. Substituting for b, c_0, and c_1 into Equations (3.14) and (3.15) and simplifying yields the communication time in seconds as

$$T_{comm.} = \begin{cases} 8.02 \times 10^{-9} n^2 + 5.00 \times 10^{-5} & \text{if } n < 256 \\ 4.14 \times 10^{-9} n^2 + 1.42 \times 10^{-4} & \text{if } 256 \leq n < 512 \\ 3.38 \times 10^{-9} n^2 + 5.94 \times 10^{-4} & \text{if } n \geq 512. \end{cases}$$

From Lines 1-2 of Listing 4.18, computation is partitioned for CPU execution if $T_{comp.}^{CPU} \leq T_{comp.}^{GPU} + T_{comm.}$. Therefore, after substitution, if

$$\frac{\frac{8n^2}{1024^3}}{51} \leq \frac{\frac{8n^2}{1024^3}}{1504} + T_{comm.},$$

computation is performed on the CPU to minimize execution time. Solving for n yields $T_{comp.}^{CPU} \leq T_{comp.}^{GPU} + T_{comm.}$ if $n \leq 64$. Therefore, if $n > 64$, the computation is performed on the GPU to minimize execution time.

Figure 4.25 illustrates measured execution time for MM. CPU (ATLAS) execution time is measured time of the CPU utilizing ATLAS to perform MM. GPU (CUBLAS) execution time is measured time for communication between the CPU and GPU in addition to measured GPU computation time utilizing CUBLAS.

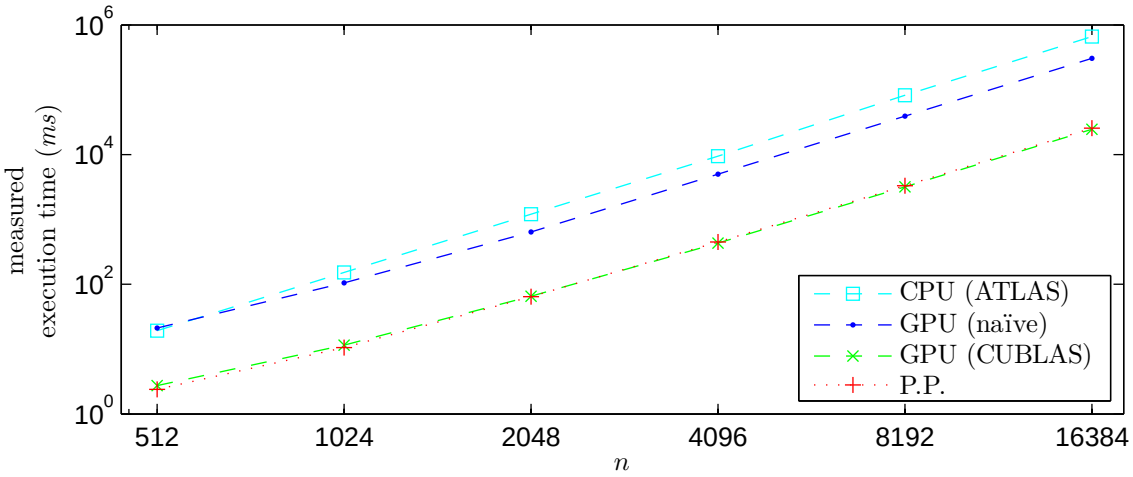


Figure 4.25: Comparison of execution time (ms) for MM on the T10 GPU.

For MM, the parallelization procedure yields execution time similar to the CUBLAS implementation for all data sizes tested. The speedup, on average, is 1.02. The procedure yields a speedup, on average, of 18.81 compared to the ATLAS implementation, and 10.60 compared to the naïve implementation. Similar to Mv, the speedup compared to the naïve and CUBLAS implementations in Figure 4.25 is less than in Figure 4.23 due to communication time. However, differing from Mv, the communication time is not a significant portion of execution time. Therefore, the reduction in speedup, due to communication, is less than for Mv.

Similar to Mv, the procedure yields execution time, on average, within 10.9% of the theoretical minimum execution time. Worst-case utilizing the procedure (n = 4096) yields time within 11.4% of the theoretical minimum. Best-case utilizing the procedure (n = 512)

yields time within 10.4% of the theoretical minimum. For MM, Step 6 of the procedure, determine the partitioning of computation, is not necessary for MM since $n > 64$. However, Step 6 is necessary to minimize execution time for Mv and thus is included.

Chapter 5

Performance Analysis

The parallelization procedure is developed and verified through Mv and MM. Therefore, Section 5.1 of this chapter is the application of the parallelization procedure to convolution. Each step is illustrated in a subsection and results are presented. In addition to verification through convolution, in Section 5.2, execution time of the conjugate gradient method [65] utilizing the parallelization procedure is compared with execution time of the CPU utilizing ATLAS and the GPU utilizing CUBLAS.

All results in this chapter were gathered using the NVIDIA Tesla S1070 which includes a pair of T10 GPUs connected to a quad-core $2.26GHz$ Intel Xeon E5520 CPU at Alabama Supercomputer Authority [62]. The height or width of a square matrix, n, is varied from 512 to 16384 as defined by Equation (4.1). $dBlk.x$ is varied from 8 to 512, the maximum allowable value for the T10.

5.1 Convolution

As mentioned, the parallelization procedure developed in Chapter 4 is applied to convolution to show the validity of the procedure. Three various filter sizes ($FS = \{3, 63, 513\}$) were selected for testing convolution.

5.1.1 Placement of Data

Step 1 of the parallelization procedure is to determine the optimal placement of data in GPU memory. Listing 5.1 is the kernel for the global memory implementation of convolution on a GPU where each thread computes one C_{ij}. The size of the filter, or the height or width of $\mathbf{B}$, is passed to the kernel as FS. The radius of the filter is computed in Line 2. Each thread

```
1  __global__ void Conv(float* A, float* B, float* C, int n, int FS) {
2    int radius = (FS-1)/2;
3    int indexCol = blockIdx.x * blockDim.x + threadIdx.x;
4    int indexRow = blockIdx.y * blockDim.y + threadIdx.y;
5    float temp = 0.0;
6    for(int k = 0; k < FS; k++) {
7      int Col = indexCol + k - radius;
8      for(int l = 0; l < FS; l++) {
9        int Row = indexRow + l - radius;
10       if(Col >= 0 && Col < n) {
11         if(Row >= 0 && Row < n) {
12           temp += A[Row * n + Col] * B[l * FS + k];
13         }
14       }
15     }
16   }
17   C[indexRow * n + indexCol] = temp;
18 }
```

Listing 5.1: The global memory implementation of convolution on a GPU.

computes the column and row index of the C_{ij} to compute in Lines 3 and 4. Lines 10 and 11 perform boundary checking to ensure no address is accessed which is out of range. k and l determine which row and column of the filter, respectively, are being accessed. The order of the *for*-loops in Lines 6 and 8 determines the order in which computation is performed and thus affects the computation time. The order of computation, or the computation pattern, is discussed in Section 5.1.2.

In Line 3 of Listing 5.1, *indexCol* is calculated utilizing *threadIdx.x*. Therefore, *Col*, Line 7, is also dependent on *threadIdx.x*. Since HWs are assigned in row-major order (*threadIdx.x* first), accesses to **A** in Line 12 are coalesced. The number of global memory accesses for accessing **A** is dependent on the number of threads per HW with differing *threadIdx.y* values. This is because each HW has *threadIdx.y* unique values for *Row* from Line 4. Therefore, the number of global memory accesses for accessing **A** is approximately[1]

$$\text{Gld}_{32B} = \frac{n^2 FS^2}{dBlk.x} \quad \text{if } dBlk.x \leq 8$$

[1]For simplicity, the equations are approximations. Due to boundary checking, not all reads defined by the equations are necessary or issued.

and

$$\text{Gld}_{64B} = \frac{n^2FS^2}{16} \quad \text{if } dBlk.x \geq 16.$$

Since each thread in a HW accesses the same value of **B** in Line 12, the number of global memory accesses for accessing **B** is approximately[1]

$$\text{Gld}_{32B} = \frac{n^2FS^2}{16} \tag{5.1}$$

regardless of the value of $dBlk.x$.

Lines 4-6 of Listing 4.1, the algorithm to determine the placement of data, specify that shared memory is utilized for reused data which is larger than the size of constant memory. It is not assumed the filter is smaller than the size of constant memory [2]. Therefore, Line 5 specifies that data which is reused is partitioned into blocks smaller than the size of shared memory. Therefore, **A** and **B** are partitioned into blocks of shared memory to reduce the number of global memory accesses as shown in Listing 5.2. Shared memory is allocated for parts of **A** and **B** in Lines 2 and 3, respectively. Therefore,

$$\text{SMemPerBlk} = \left(2 \times dBlk.y \times dBlk.x + dBlk.y\right) \times 4. \tag{5.2}$$

Every $dBlk.y$ iteration, parts of **A** and **B** are loaded into shared memory as shown in Lines 12-28. Since all HWs within a block access the same part of **B**, only some threads are utilized to load **B** into shared memory. In this kernel, threads of each HW with *threadIdx.x*=0 are utilized to load part of **B** into shared memory as shown in Lines 14-18. Boundary checking is performed in Lines 15, 19, 20, 23, 29, and 30. Similar to the global memory implementation, the order of the *for*-loops in Lines 8 and 10 determines the computation pattern, which is discussed in Section 5.1.2.

[2]If constant memory is utilized to store the filter, the maximum size of the filter is limited to 128.

```
__global__ void Conv(float* A, float* B, float* C, int n, int FS) {

  __shared__ float As[blockDim.y * 2][blockDim.x];
  __shared__ float Bs[blockDim.y];

  int radius = (FS-1)/2;
  int indexCol = blockIdx.x * blockDim.x + threadIdx.x;
  int indexRow = blockIdx.y * blockDim.y + threadIdx.y;

  float temp = 0.0;
  for(int k = 0; k < FS; k++) {
    int Col = indexCol + k - radius;

    for(int l = 0; l < FS; l++) {
      int Row = indexRow + l - radius;

      if((l % blockDim.y) == 0) {
        __syncthreads();
        if(threadIdx.x == 0) {
          if(l + threadIdx.y < FS) {
            Bs[threadIdx.y] = B[(l + threadIdx.y) * FS + k];
          }
        }
        if(Col >= 0 && Col < n) {
          if(Row >= 0 && Row < n) {
            As[threadIdx.y][threadIdx.x] = A[Row * n + Col];
          }
          if((Row + blockDim.y) >= 0 && (Row + blockDim.y) < n ) {
            As[threadIdx.y + blockDim.y][threadIdx.x] = A[(Row +
                ↳ blockDim.y) * n + Col];
          }
        }
        __syncthreads();
      }

      if(Col >= 0 && Col < n ) {
        if(Row >= 0 && Row < n) {
          temp += As[threadIdx.y + (l % blockDim.y)][threadIdx.x] * Bs[(l
              ↳ % blockDim.y)];
        }
      }
    }
  }
  C[indexRow * n + indexCol] = temp;
}
```

Listing 5.2: The shared memory implementation of convolution on a GPU.

Since global memory is accessed every $dBlk.y$ iteration, the number of global memory accesses for accessing **A** is dependent on the size of **B**, FS, and the number of threads per HW with differing *threadIdx.y* values. This is because each HW has *threadIdx.y* unique values for *Row* from Lines 6 and 11. Therefore, the number of global memory accesses for accessing **A** is approximately[3]

$$\text{Gld}_{32B} = \frac{2n^2FS}{dBlk.x}\left\lceil\frac{FS}{dBlk.y}\right\rceil \quad \text{if } dBlk.x \leq 8 \tag{5.3}$$

and[4]

$$\text{Gld}_{64B} = \frac{2n^2FS}{16}\left\lceil\frac{FS}{dBlk.y}\right\rceil \quad \text{if } dBlk.x \geq 16. \tag{5.4}$$

If $dBlk.x \geq 16$, this reduces the number of global memory accesses for accessing **A** by a factor of $\frac{FS}{2\left\lceil\frac{FS}{dBlk.y}\right\rceil}$ compared to the global memory implementation. Since global memory is accessed every $dBlk.y$ iteration, the number of global memory accesses for accessing **B** is approximately[3]

$$\text{Gld}_{32B} = \frac{16n^2FS}{dBlk.x^2}\left\lceil\frac{FS}{dBlk.y}\right\rceil \quad \text{if } dBlk.x \leq 8 \tag{5.5}$$

and

$$\text{Gld}_{32B} = \frac{n^2FS}{dBlk.x}\left\lceil\frac{FS}{dBlk.y}\right\rceil \quad \text{if } dBlk.x \geq 16. \tag{5.6}$$

This reduces the number of $32B$ accesses for accessing **B** by a factor of $\frac{FSdBlk.x}{16\left\lceil\frac{FS}{dBlk.y}\right\rceil}$ compared to the global memory implementation.

Figures 5.1-5.3 depict maximum effective bandwidth of the global memory implementation of convolution (Listing 5.1) and the shared memory implementation (Listing 5.2). Effective bandwidth for convolution is calculated in Section 3.5 as $\frac{8n^2FS^2bytes}{T^{GPU}_{comp.}}$. Figure 5.1 depicts maximum effective bandwidth for a filter size, FS, of 3.

[3]For simplicity, the equations are approximations. Due to boundary checking, not all reads defined by the equations are necessary or issued.

[4]Since FS is not a power of two, some reads are $32B$ transactions if $dBlk.x \geq 16$.

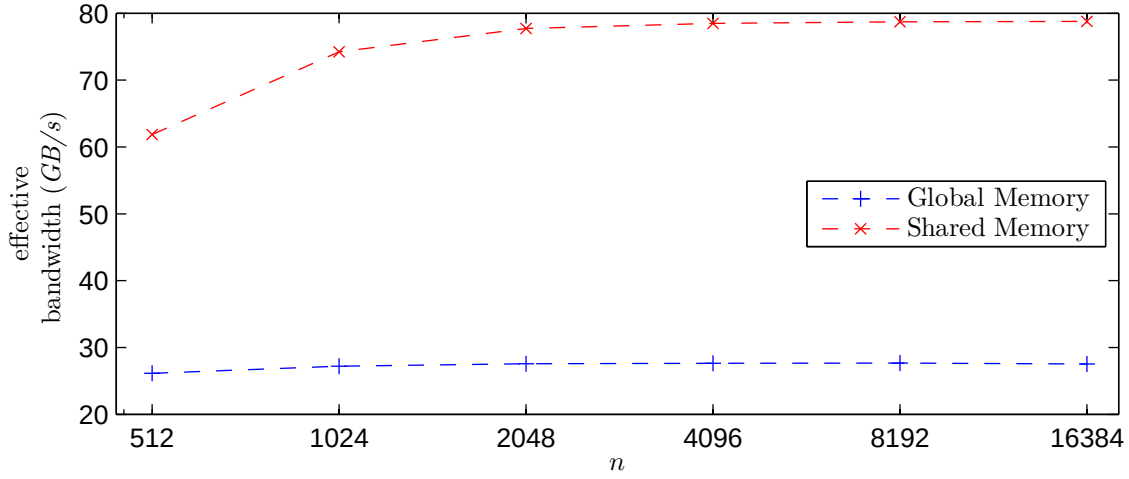


Figure 5.1: Comparison of GPU memories: Maximum effective bandwidth (GB/s) for convolution on the T10 GPU. FS=3.

As illustrated, the reduction in global memory accesses significantly increases the effective bandwidth for convolution with a small filter size. The shared memory implementation yields a speedup, on average, of 2.74 compared to the global memory implementation. The minimum (n = 512) and maximum (n = 16384) speedups are 2.37 and 2.86, respectively. Figure 5.2 depicts maximum effective bandwidth for a filter size of 63.

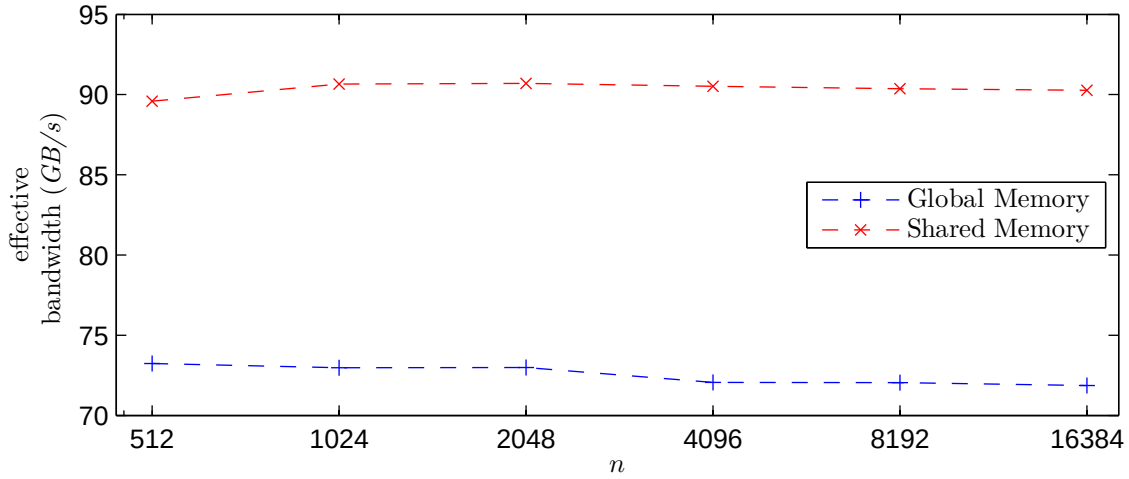


Figure 5.2: Comparison of GPU memories: Maximum effective bandwidth (GB/s) for convolution on the T10 GPU. FS=63.

For a filter size of 63, utilizing shared memory increases effective bandwidth due to the reduction in global memory accesses. However, comparing Figures 5.1 and 5.2 shows

the increase in bandwidth is less as the filter size has increased. Regardless, in Figure 5.2, the shared memory implementation yields a speedup, on average, of 1.25 compared to the global memory implementation. The minimum (n = 512) and maximum (n = 4096) speedups are 1.22 and 1.26, respectively. Although the increase in effective bandwidth for the shared memory implementation is less than for a small filter size, Step 1 of the procedure is valid for a filter size of 63. Lastly, maximum effective bandwidth is illustrated in Figure 5.3 for a filter size of 513.

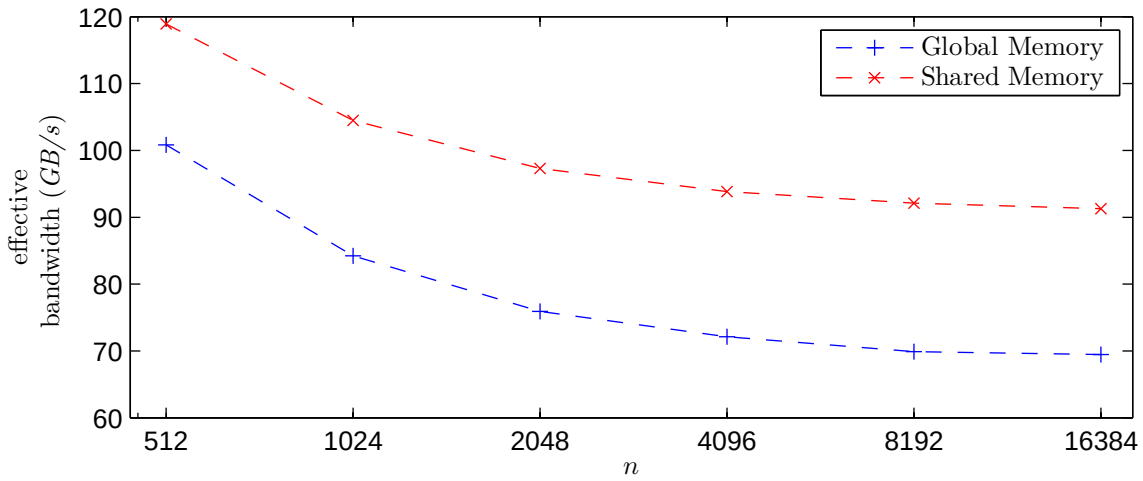


Figure 5.3: Comparison of GPU memories: Maximum effective bandwidth (GB/s) for convolution on the T10 GPU. FS=513.

Similar to a filter size of 63, the shared memory implementation yields a speedup, on average, of 1.27 compared to the global memory implementation for a large filter size. Likewise, the minimum (n = 512) and maximum (n = 8192) speedups are 1.18 and 1.32, respectively. These results show that Step 1 of the procedure yields larger reductions of computation time for smaller filter sizes. Regardless, Step 1 is verified to reduce computation time for all data and filter sizes tested.

5.1.2 Computation Patterns

Step 2 of the procedure specifies that the optimized computation pattern is derived. As mentioned, computation patterns are determined by the code and represent the manner and

order in which results are computed. For convolution, if $thread_{ij}$ computes C_{ij} where $\mathbf{C} = \mathbf{A} * \mathbf{B}$, the order in which C_{ij} is computed creates varying computation patterns.

For convolution, the filter, $\mathbf{B}$, is centered over each value of the input, $\mathbf{A}$. The products of the overlapping values of $\mathbf{A}$ and $\mathbf{B}$ are summed to compute one value of $\mathbf{C}$, C_{ij}, where $\mathbf{C} = \mathbf{A} * \mathbf{B}$. This is depicted in Figure 5.4. The order in which overlapping values are multiplied

(a) Column-major. (b) Row-major.

Figure 5.4: Computation patterns: two computation patterns for convolution.

determines the computation pattern. A pattern where each partial sum is computed first in the y-dimension, as depicted in Figure 5.4a, is the column-major computation pattern. Figure 5.4b depicts the computation pattern where each value is computed first in the x-dimension, row-major.

Column-major

The shared memory implementation of convolution utilizing the column-major computation pattern depicted in Figure 5.4a is shown in Listing 5.2 in Section 5.1.1. The amount of shared memory allocated per block is defined by Equation (5.2) and the number of global memory accesses for accessing $\mathbf{A}$ and $\mathbf{B}$ is approximated[5] in Equations (5.3)-(5.6).

[5]Due to boundary checking, not all reads are issued.

Optimized (row-major)

The shared memory implementation of convolution utilizing the row-major computation pattern depicted in Figure 5.4b requires several minor modifications to Listing 5.2. Therefore, the shared memory implementation of convolution utilizing the row-major computation pattern is shown in Listing 5.3. Shared memory is allocated for parts of **A** and **B** in Lines 2 and 3, respectively. Therefore,

$$\text{SMemPerBlk} = \left(2 \times dBlk.x \times dBlk.y + dBlk.x\right) \times 4. \tag{5.7}$$

Every $dBlk.x$ iteration, parts of **A** and **B** are loaded into shared memory as shown in Lines 12-28. Since all HWs within a block access the same part of **B**, only some threads are utilized to load **B** into shared memory. In this kernel, threads of each HW with *threadIdx.y*=0 are utilized to load part of **B** into shared memory as shown in Line 14. Boundary checking is performed in Lines 15, 19, 20, 23, 29, and 30.

Since global memory is accessed every $dBlk.x$ iteration, the number of global memory accesses for accessing **A** is dependent on the size of **B**, FS, and the number of threads per HW with differing *threadIdx.y* values. This is because each HW has *threadIdx.y* unique values for *Row* from Lines 6 and 9. Therefore, the number of global memory accesses for accessing **A** is approximately[8]

$$\text{Gld}_{32B} = \frac{2n^2FS}{dBlk.x}\left\lceil\frac{FS}{dBlk.x}\right\rceil \quad \text{if } dBlk.x \leq 8 \tag{5.8}$$

and[6]

$$\text{Gld}_{64B} = \frac{2n^2FS}{16}\left\lceil\frac{FS}{dBlk.x}\right\rceil \quad \text{if } dBlk.x \geq 16. \tag{5.9}$$

If $dBlk.x \geq 16$, this reduces the number of global memory accesses for accessing **A** by a factor of $\frac{FS}{2\left\lceil\frac{FS}{dBlk.x}\right\rceil}$ compared to the global memory implementation. However, if $dBlk.x =$

[6]Since FS is not a power of two, some reads are $32B$ transactions if $dBlk.x \geq 16$.

```
__global__ void Conv(float* A, float* B, float* C, int n, int FS) {

  __shared__ float As[blockDim.y][blockDim.x * 2];
  __shared__ float Bs[blockDim.x];

  int radius = (FS-1)/2;
  int indexCol = blockIdx.x * blockDim.x + threadIdx.x;
  int indexRow = blockIdx.y * blockDim.y + threadIdx.y;

  float temp = 0.0;
  for(int k = 0; k < FS; k++) {
    int Row = indexRow + k - radius;

    for(int l = 0; l < FS; l++) {
      int Col = indexCol + l - radius;

      if((l % blockDim.x) == 0) {
        __syncthreads();
        if(threadIdx.y == 0) {
          if(l + threadIdx.x < FS) {
            Bs[threadIdx.x] = B[k * FS + l + threadIdx.x];
          }
        }
        if(Row >= 0 && Row < n) {
          if(Col >= 0 && Col < n) {
            As[threadIdx.y][threadIdx.x] = A[Row * n + Col];
          }
          if((Col + blockDim.x) >= 0 && (Col + blockDim.x) < n ) {
            As[threadIdx.y][threadIdx.x + blockDim.x] = A[Row * n + Col +
               ↳ blockDim.x];
          }
        }
        __syncthreads();
      }

      if(Row >= 0 && Row < n ) {
        if(Col >= 0 && Col < n) {
          temp += As[threadIdx.y][threadIdx.x + (l % blockDim.x)] * Bs[(l
             ↳ % blockDim.x)];
        }
      }
    }
  }
  C[indexRow * n + indexCol] = temp;
}
```

Listing 5.3: The shared memory implementation of convolution on a GPU utilizing the optimized computation pattern.

$dBlk.y$, there is no reduction in the number of accesses compared to the shared memory implementation utilizing the column-major computation pattern.

Since global memory is accessed every $dBlk.x$ iteration, the number of global memory accesses for accessing $\mathbf{B}$ is approximately[8]

$$\text{Gld}_{32B} = \frac{n^2FS}{dBlk.x \times dBlk.y}\left\lceil\frac{FS}{dBlk.x}\right\rceil \quad \text{if } dBlk.x \leq 8. \tag{5.10}$$

and

$$\text{Gld}_{64B} = \frac{n^2FS}{16dBlk.y}\left\lceil\frac{FS}{dBlk.x}\right\rceil \quad \text{if } dBlk.x \geq 16. \tag{5.11}$$

If $dBlk.x \geq 16$, $32B$ accesses for accessing $\mathbf{B}$ are replaced with $64B$ accesses compared to the shared memory implementation utilizing the column-major computation pattern. The number of accesses to global memory is reduced by a factor of $\frac{16dBlk.y\left\lceil\frac{FS}{dBlk.y}\right\rceil}{dBlk.x\left\lceil\frac{FS}{dBlk.x}\right\rceil}$. Therefore, if $dBlk.x = dBlk.y = 16$, this computation pattern reduces the number of accesses to global memory by a factor of 16 and the number of bytes accessed by a factor of 8. Therefore, the row-major pattern is the optimized computation pattern.

Figures 5.5-5.7 depict maximum effective bandwidth of convolution before (Listing 5.2) and after (Listing 5.3) utilizing the optimized computation pattern. The optimal placement of data is utilized in both listings. As mentioned, effective bandwidth for convolution is $\frac{8n^2FS^2bytes}{T^{GPU}_{comp.}}$. Figure 5.5 depicts maximum effective bandwidth for a filter size, FS, of 3.

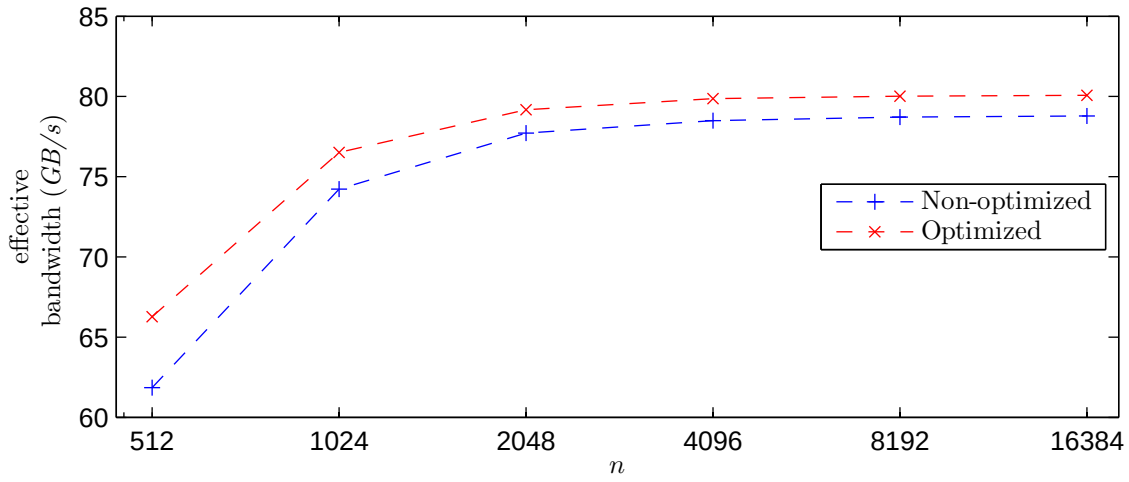


Figure 5.5: Comparison of computation patterns: Maximum effective bandwidth (GB/s) for convolution on the T10 GPU. FS=3.

Shown in Figure 5.5, Step 2 of the procedure yields a modest speedup, on average, of 1.03 compared to the non-optimized pattern. As mentioned, for the optimized computation pattern, the number of accesses to global memory is reduced by a factor of $\frac{16dBlk.y\lceil\frac{FS}{dBlk.y}\rceil}{dBlk.x\lceil\frac{FS}{dBlk.x}\rceil}$ compared to the non-optimized pattern. Since maximum effective bandwidth is illustrated and $dBlk.x$ and $dBlk.y$ vary, the optimized computation pattern yields a modest speedup. The minimum (n = 16384) and maximum (n = 512) speedups are 1.02 and 1.07, respectively. For a filter size of 63, the maximum effective bandwidth is illustrated in Figure 5.6.

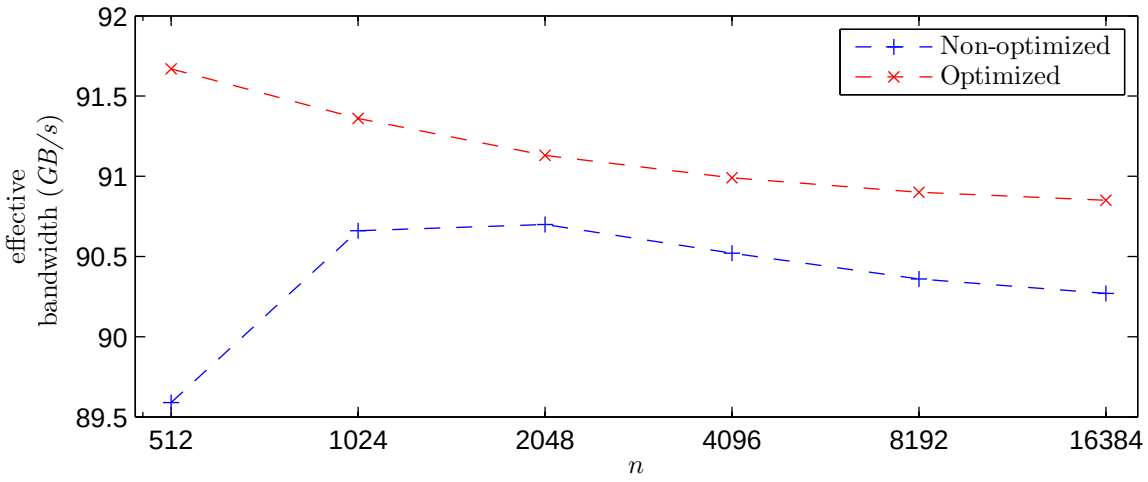


Figure 5.6: Comparison of computation patterns: Maximum effective bandwidth (GB/s) for convolution on the T10 GPU. FS=63.

In Figure 5.6, the effective bandwidth, before and after utilizing the optimized computation pattern, shows a differing behavior for the varying data sizes. However, given the scale of the figure, the difference is small and not explored further. Similar to a small filter size, the optimized computation pattern for a filter size of 63 yields a near-constant speedup, on average, of 1.01 compared to the non-optimized pattern. The minimum (n = 2048) and maximum (n = 512) speedups are 1.00 and 1.02, respectively. Lastly, Figure 5.7 depicts maximum effective bandwidth for a filter size of 513.

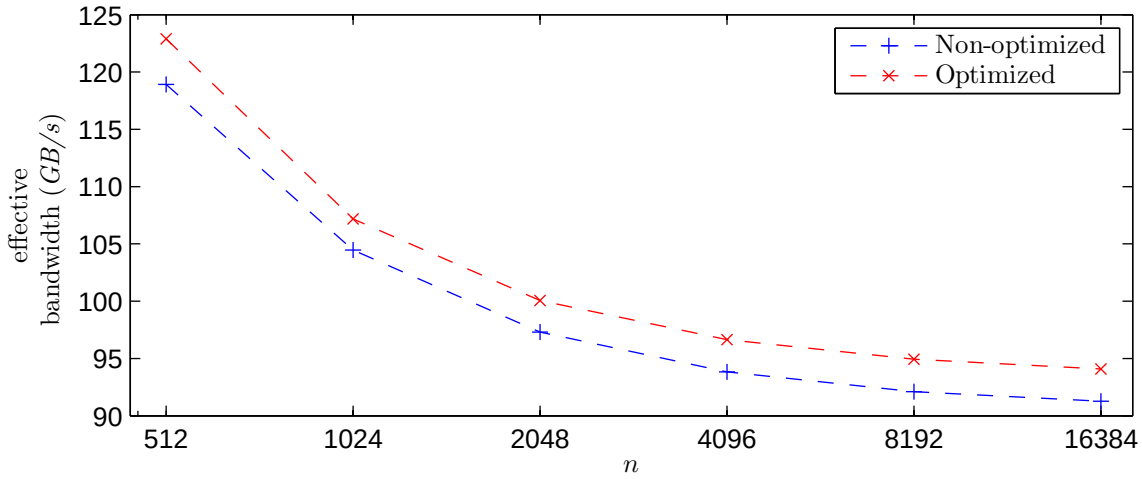


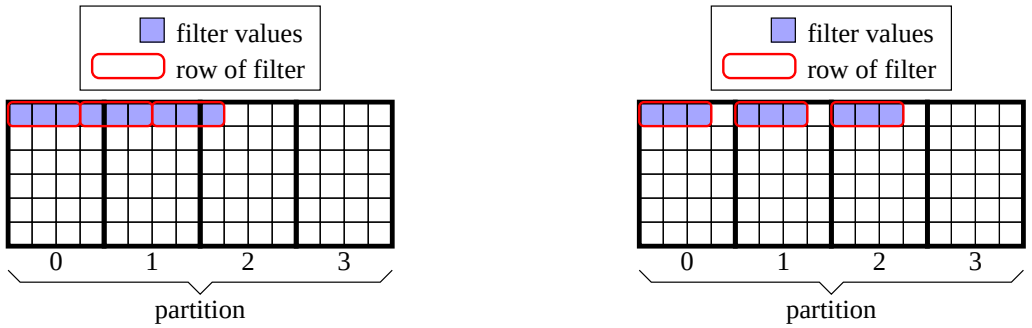
Figure 5.7: Comparison of computation patterns: Maximum effective bandwidth (GB/s) for convolution on the T10 GPU. FS=513.

As illustrated in Figure 5.7, the optimized computation pattern yields a constant speedup of 1.03 compared to the non-optimized pattern. The effective bandwidth for all filter sizes tested is roughly equivalent and a modest increase is shown when utilizing the optimized computation pattern. Although modest speedups are measured for all filter sizes, the bandwidth is increased for all data and filter sizes tested. Therefore, Step 2 of the parallelization procedure is shown to yield increases in effective bandwidth, and thus reduce the computation time, for convolution.

5.1.3 Access Patterns

Step 3 of the parallelization procedure is to derive the optimized access pattern that minimizes partition camping and eliminates bank conflicts. For Mv and MM, HWs begin execution by reading the first value of their respective rows of **A**. However, for convolution, HWs begin by reading varying columns of their respective rows of **A**. Therefore, HWs are spread to varying partitions for accessing **A** without an optimized access pattern.

Figure 5.8a illustrates a 3x3 filter, **B**, stored in global memory. Padding the filter with zeros such that each row resides in the next partition of global memory reduces partition camping. This is illustrated in Figure 5.8b. In addition, padding the filter aligns each



(a) Non-padded filter in global memory.

(b) Padded filter in global memory.

Figure 5.8: Access patterns: example of a 3x3 filter, **B**, stored in global memory. Each row of a partition is 4 values of type float.

row to a specified boundary. From the CUDA programming guide [64], "Any access (via a variable or a pointer) to data residing in global memory compiles to a single global memory instruction if and only if the size of the data type is 1, 2, 4, 8, or 16 bytes and the data is naturally aligned (i.e., its address is a multiple of that size)." Therefore, padding the filter with zeros ensures the data is naturally aligned thus reducing the number of global memory transactions.

Similarly, padding **A** with zeros eliminates the boundary checking instructions in Lines 15, 19, 20, 23, 29, and 30 of Listing 5.3. From the CUDA programming guide [64], "If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each

branch path taken." Eliminating boundary checking instructions via padding eliminates thread divergence. Therefore, modifying Listing 5.3 to accommodate padding of **A** and **B** yields Listing 5.4 which is the shared memory implementation of convolution utilizing the optimized computation and access pattern. Shared memory is allocated in Lines 2 and 3 of

```
1  __global__ void Conv(float* A, float* B, float* C, int n, int FS) {

2    __shared__ float As[blockDim.y][blockDim.x * 2];
3    __shared__ float Bs[blockDim.x];

4    int padding = 64 - FS % 64;
5    int indexCol = blockIdx.x * blockDim.x + threadIdx.x;
6    int indexRow = blockIdx.y * blockDim.y + threadIdx.y;

7    float temp = 0.0;
8    for(int k = 0; k < FS; k++) {
9      int Row = indexRow + k;

10     for(int l = 0; l < FS; l++) {
11       int Col = indexCol + l;

12       if((l % blockDim.x) == 0) {
13         __syncthreads();
14         if(threadIdx.y == 0) {
15             Bs[threadIdx.x] = B[k * (FS + padding) + l + threadIdx.x];
16         }
17         As[threadIdx.y][threadIdx.x] = A[Row * (n + FS - 1) + Col];
18         As[threadIdx.y][threadIdx.x + blockDim.x] = A[Row * (n + FS - 1)
     ↳ + Col + blockDim.x];
19         __syncthreads();
20       }

21       temp += As[threadIdx.y][threadIdx.x + (l % blockDim.x)] * Bs[(l %
           ↳ blockDim.x)];
22     }
23   }
24   C[indexRow * n + indexCol] = temp;
25 }
```

Listing 5.4: The shared memory implementation of convolution on a GPU utilizing the optimized computation and access pattern.

Listing 5.4 the same as in Lines 2 and 3 of Listing 5.3. Therefore, the amount of shared memory allocated per block is defined by Equation (5.7). In Line 4, *padding* instead of the filter radius is calculated. *padding* specifies the number of zeros added to each row of the filter. The index calculations for accessing **A**, Lines 9 and 11, no longer include the radius of the filter, as **A** is padded. Since **A** is padded by the size of the filter, the length of each row is $n + FS - 1$ rather than n as shown in Lines 17 and 18.

Figures 5.9-5.11 depict maximum effective bandwidth of convolution before (Listing 5.3) and after (Listing 5.4) utilizing the optimized access pattern. The optimal placement of data and optimized computation patterns are utilized. Figure 5.9 depicts maximum effective bandwidth for a filter size of 3.

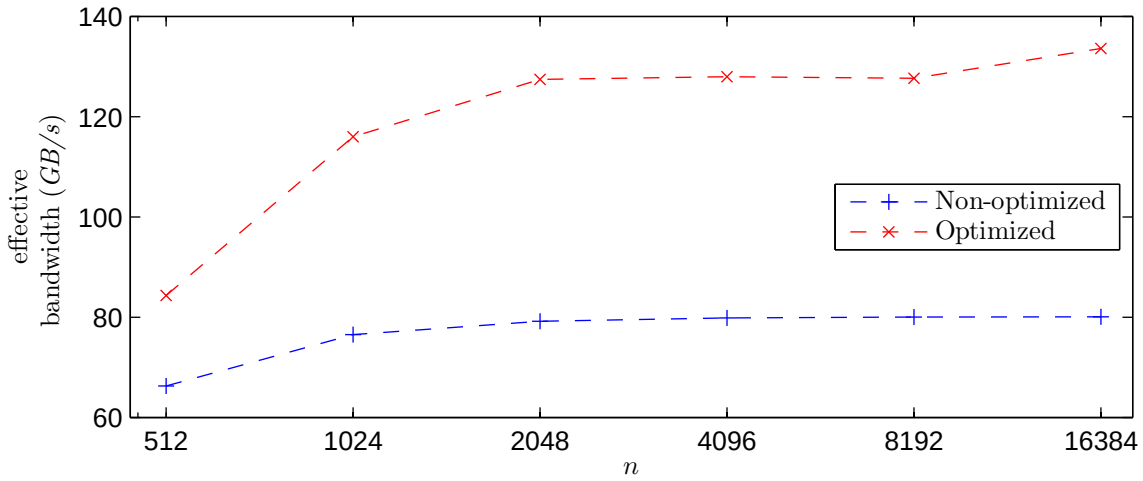


Figure 5.9: Comparison of access patterns: Maximum effective bandwidth (GB/s) for convolution on the T10 GPU. FS=3.

As mentioned, the optimized access pattern, which consists of padding **A** and **B** with zeros, reduces partition camping. In addition, the optimized access pattern eliminates boundary checking, and thus, reduces thread divergence. As illustrated in Figure 5.9, the optimized access pattern yields a speedup, on average, of 1.54 compared to the non-optimized pattern for a small filter size. The minimum (n = 512) and maximum (n = 16384) speedups are 1.27 and 1.67, respectively. Similar to Mv and MM, the optimized access pattern, which reduces partition camping, significantly increases bandwidth for convolution with a small filter size. Maximum effective bandwidth for a filter size of 63 is depicted in Figure 5.10.

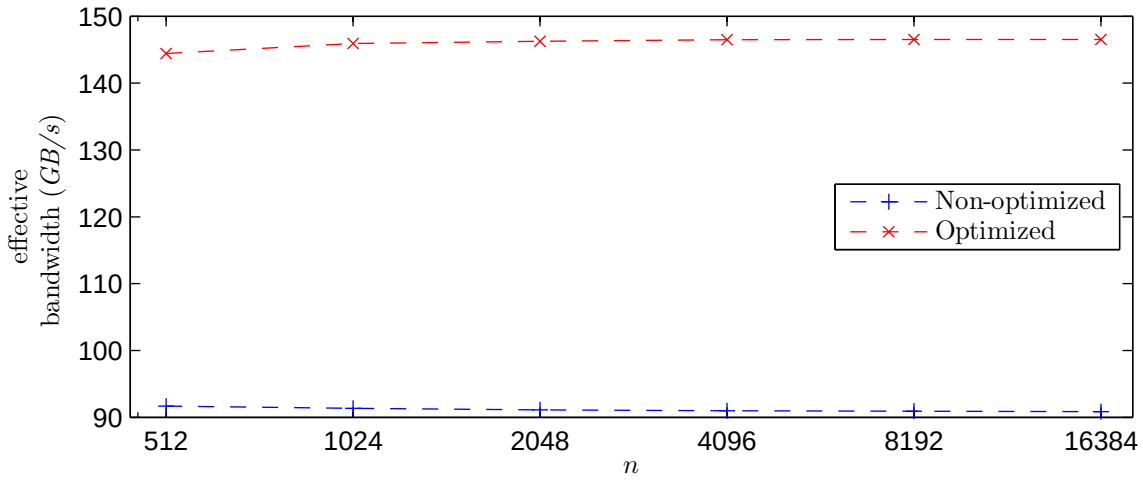


Figure 5.10: Comparison of access patterns: Maximum effective bandwidth (GB/s) for convolution on the T10 GPU. FS=63.

Similar to a filter size of 3, the optimized access pattern for a filter size of 63 yields a speedup, on average, of 1.60 compared to the non-optimized pattern. The minimum (n = 512) and maximum (n = 16384) speedups are 1.58 and 1.61, respectively. Lastly, Figure 5.11 depicts maximum effective bandwidth for a filter size of 513.

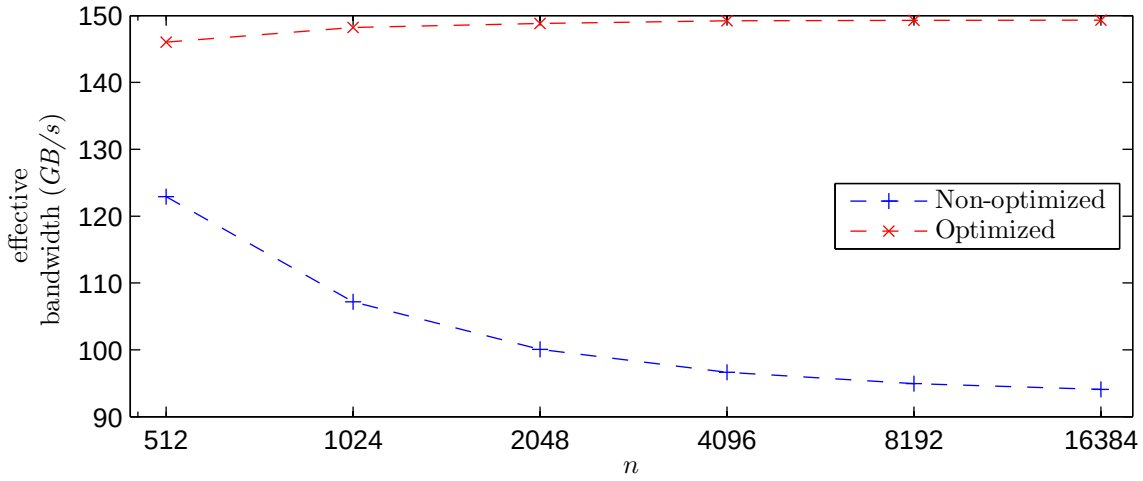


Figure 5.11: Comparison of access patterns: Maximum effective bandwidth (GB/s) for convolution on the T10 GPU. FS=513.

As shown in Figure 5.11, the optimized access pattern yields a speedup, on average, of 1.46 compared to the non-optimized pattern. The minimum (n = 512) and maximum (n = 16384) speedups are 1.19 and 1.59, respectively. For each filter size tested, Step 3 of

the parallelization procedure, derive the optimized access pattern, yields significant increases in maximum effective bandwidth. In addition, the optimized access pattern yields similar increases in speedup for all data and filter sizes tested.

5.1.4 Fine-tuning

Step 4 of the parallelization procedure is to fine-tune the kernel by unrolling loops and minimizing index calculations. As mentioned, loop unrolling reduces computation time by increasing the instruction mix of memory transactions and computation instructions. Minimizing index calculations, particularly in inner loops, reduces the time necessary to issue global memory transactions, thus reducing the computation time.

Step 4, fine-tuning, is applied to the shared memory implementation of convolution on a GPU utilizing the optimized computation and access pattern, Listing 5.4, to yield Listing 5.5. No modifications are performed to the allocation of shared memory and therefore, Equation (5.7) defines the amount of shared memory utilized. Since modifications are performed only to index calculations and loop unrolling, the number of global memory reads is defined by Equations (5.8)-(5.11). In Line 8, *calc*1 is calculated outside of the *for*-loops as it is a commonly used calculation in the *for*-loops for index calculations of **A** and **B**. Index calculations are reduced by initializing pointers to **A**, **B**, and **C** in Lines 9-11, thus eliminating the need for *Row* and *Col* from Listing 5.4. Pointer arithmetic is performed in Lines 21, 23, 29, and 30 to further reduce index calculations.

In Listing 5.5, *for*-loops are preceded with the *#pragma unroll* directive in Lines 12 and 14 to increase the instruction mix and reduce the number of registers used per thread. Since the loops are dependent only on FS, the compiler unrolls the loops until the maximum instruction limit is reached. Therefore, the amount of loop unrolling performed is dependent on FS.

```
__global__ void Conv(float* A, float* B, float* C, int n, int FS) {
  __shared__ float As[blockDim.y][blockDim.x * 2];
  __shared__ float Bs[blockDim.x];

  int padding = 64 - FS % 64;
  int indexCol = blockIdx.x * blockDim.x + threadIdx.x;
  int indexRow = blockIdx.y * blockDim.y + threadIdx.y;
  float temp = 0.0;
  int calc1 = blockDim.x * (floorf(FS / blockDim.x) + 1);
  A += indexRow * (n + FS-1) + indexCol;
  B += threadIdx.x;
  C += indexRow * n + indexCol;

  #pragma unroll
  for(int k = 0; k < FS; k++) {

    #pragma unroll
    for(int l = 0; l < FS; l++) {

      if((l % blockDim.x) == 0) {
        __syncthreads();
        if(threadIdx.y == 0) {
            Bs[threadIdx.x] = B[0];
        }
        B += blockDim.x;
        As[threadIdx.y][threadIdx.x] = A[0];
        A += blockDim.x;
        As[threadIdx.y][threadIdx.x + blockDim.x] = A[0];
        __syncthreads();
      }
      temp += As[threadIdx.y][threadIdx.x + (l % blockDim.x)] * Bs[(l %
          ↪ blockDim.x)];
    }
    A += n + FS-1 - calc1;
    B += FS + padding - calc1;
  }
  C[0] = temp;
}
```

Listing 5.5: The shared memory implementation of convolution on a GPU utilizing the optimized computation and access pattern after fine-tuning

Figures 5.12-5.14 depict the maximum effective bandwidth of convolution before (Listing 5.4) and after (Listing 5.5) fine-tuning. Figure 5.12 depicts maximum effective bandwidth for a filter size, FS, of 3.

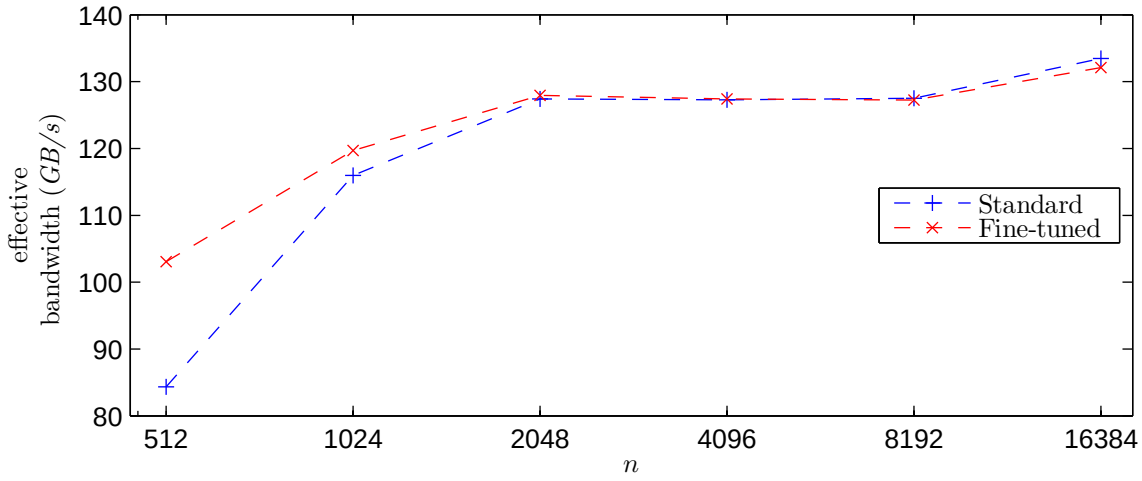


Figure 5.12: Comparison of fine-tuning: Maximum effective bandwidth (GB/s) for convolution on the T10 GPU. FS=3.

In Figure 5.12, the kernel after fine-tuning yields a modest speedup, on average, of 1.04 compared to before fine-tuning. The minimum (n = 16384) and maximum (n = 512) speedups are 0.99 and 1.22, respectively. Similar to Mv, the figure suggests Step 4 of the procedure, fine-tuning the kernel, has a minimal impact on computation time for a small filter size. However, as illustrated in Figure 5.13, Step 4 of the procedure increases bandwidth more for larger filter sizes.

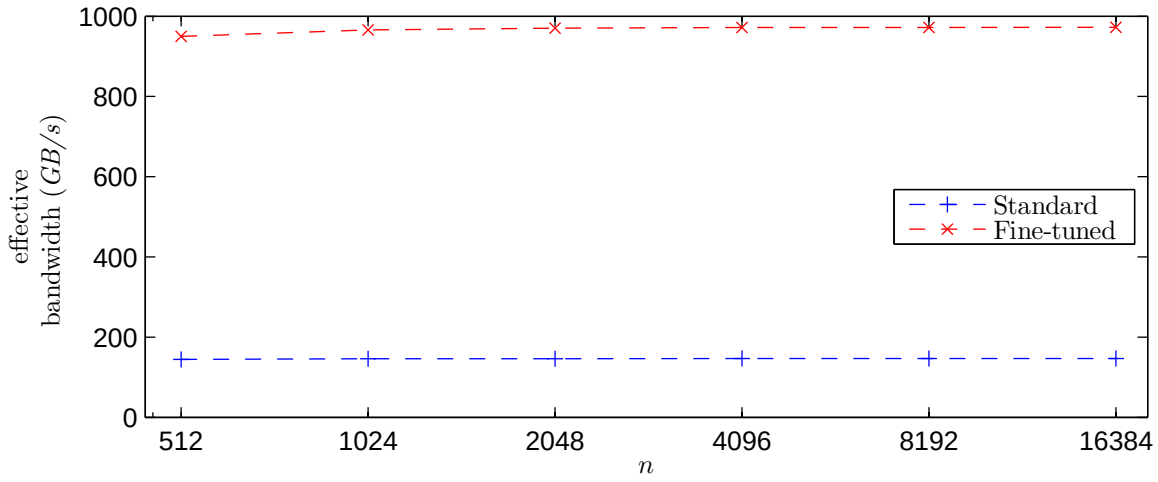


Figure 5.13: Comparison of fine-tuning: Maximum effective bandwidth (GB/s) for convolution on the T10 GPU. FS=63.

The kernel after fine-tuning yields a speedup, on average, of 6.62 compared to before fine-tuning for a filter size of 63. The minimum (n = 512) and maximum (n = 16384) speedups are 6.58 and 6.64, respectively. The speedup is significantly higher than for a filter size of 3 presumably due to loop unrolling. As mentioned, the compiler automatically unrolls small loops. The results in Figures 5.12 and 5.13 suggest that for a small filter size, the compiler automatically unrolls both loops depicted in Listing 5.4. Therefore, fine-tuning the kernel has a minimal impact for a small filter size. However, as the filter size increases, results indicate the compiler does not automatically unroll the *for*-loops. Therefore, the procedure yields significant speedups by forcing the compiler to unroll the loops. Lastly, Figure 5.14 depicts effective bandwidth for a filter size of 513.

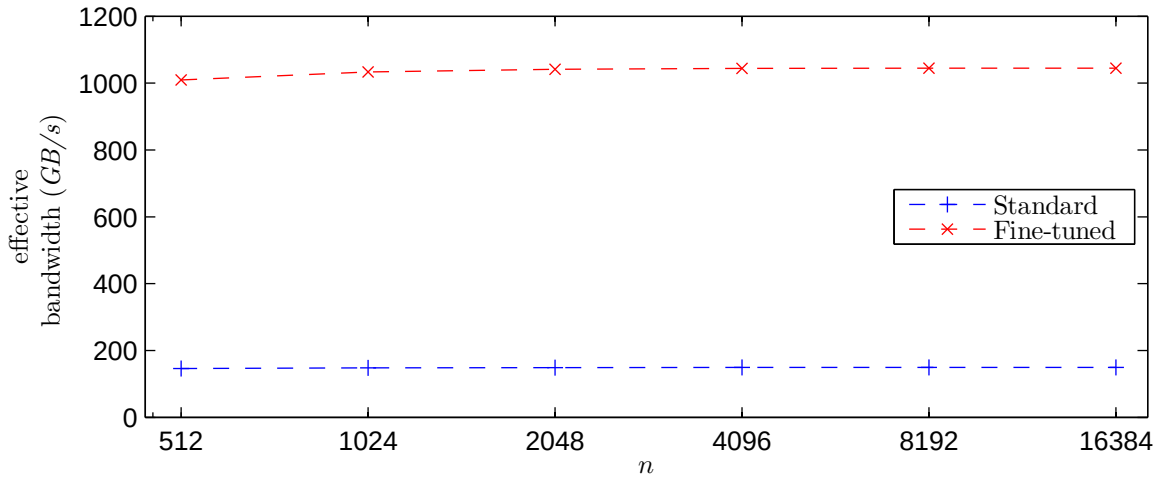


Figure 5.14: Comparison of fine-tuning: Maximum effective bandwidth (GB/s) for convolution on the T10 GPU. FS=513.

Similar to a filter size of 63, for a filter size of 513, the kernel after fine-tuning yields a speedup, on average, of 6.98 compared to before fine-tuning. The minimum (n = 512) and maximum (n = 8192) speedups are 6.91 and 7.00, respectively. Again, the speedup is significantly higher for a filter size of 513 than for a filter size of 3. Since the amount of loop unrolling performed is dependent on FS, results suggest that loop unrolling occurs without instructing the compiler for small filter sizes but not for larger sizes. Therefore, significant speedup from Step 4 is attained only for larger filter sizes. Therefore, the speedup from Step 4 of the procedure, fine-tuning the kernel, is dependent on the filter size. Regardless, Step 4 of the procedure yields an increase in effective bandwidth, and thus a decrease in computation time, on average, for all data and filter sizes tested.

5.1.5 Input Parameters

Step 5 of the parallelization procedure specifies that optimal input parameters are derived. A list of five steps to derive optimal input parameters is presented in Section 4.5. Since the derivation is Step 5 of the procedure, optimal input parameters are derived for the kernel in Listing 5.5 which is the shared memory implementation of convolution on a GPU utilizing the optimized computation and access pattern after fine-tuning. From the kernel

and Equations (4.12) and (4.13),

$$8 \leq dBlk.x \leq 512 \tag{5.12a}$$

$$1 \leq dBlk.y \leq \frac{512}{dBlk.x} \tag{5.12b}$$

$$dGrd.x = \frac{n}{dBlk.x} \tag{5.12c}$$

$$dGrd.y = \frac{n}{dBlk.y} \tag{5.12d}$$

From Equation (5.7), SMemPerBlk = $(2 \times dBlk.x \times dBlk.y + dBlk.x) \times 4$. From compilation of the kernel, RegsPerThd = 10 and is constant regardless of n, FS, and input parameters. Therefore, Equation (3.7) simplifies to

$$\text{Blks}_{SM}^{active} = \min\left(8, \frac{1024}{dBlk.x \times dBlk.y}\right). \tag{5.13}$$

Substituting $\text{Blks}_{SM}^{active}$ from Equation (5.13), $dGrd.x$ from Equation (5.12c), and $dGrd.y$ from Equation (5.12c) into Equation (3.8) yields

$$\text{Blks}_{GPU}^{active} = \min\left(30 \times \min\left(8, \frac{1024}{dBlk.x \times dBlk.y}\right), \frac{n^2}{dBlk.x \times dBlk.y}\right). \tag{5.14}$$

Since $n \geq 512$ from Equation (4.1) and $dBlk.x \times dBlk.y \leq 512$ from Equation (4.12), Equation (5.14) simplifies to

$$\text{Blks}_{GPU}^{active} = 30 \times \min\left(8, \frac{1024}{dBlk.x \times dBlk.y}\right). \tag{5.15}$$

Substituting $\text{Blks}_{GPU}^{active}$ from Equation (5.15) and the minimum of $\text{Thds}_{GPU}^{active}$ from Equation (4.14) into Equation (3.9) yields

$$12800 \leq 30 \times \min\left(8, \frac{1024}{dBlk.x \times dBlk.y}\right) \times dBlk.x \times dBlk.y. \tag{5.16}$$

If $\frac{1024}{dBlk.x \times dBlk.y} \geq 8$, then Equation (5.16) is true. Solving for $dBlk.y$ in Equation (5.16) if $\frac{1024}{dBlk.x \times dBlk.y} < 8$ yields

$$dBlk.y \geq \frac{12800}{240 dBlk.x}.$$

Since $dBlk.y$ is a power of two, this simplifies to $dBlk.y \geq \frac{64}{dBlk.x}$. Combining this with the limits of $dBlk.y$ from Equation (5.12b) yields

$$\max\left(\frac{64}{dBlk.x}, 1\right) \leq dBlk.y \geq \frac{512}{dBlk.x}. \tag{5.17}$$

Step 2 of deriving optimal input parameters specifies shared memory utilization is maximized to ensure data is reused as much as possible. From Line 15 of the kernel in Listing 5.5, the trip count for the inner loop, where shared memory is accessed, is dependent on FS. From the number of accesses to global memory from Equations (5.8)-(5.11), $dBlk.x \geq 16$. Since accesses are performed $\left\lceil \frac{FS}{dBlk.x} \right\rceil$ times in the inner loop, $dBlk.x > FS$. Therefore, $dBlk.x \geq \max(16, FS)$. However, if $dBlk.x > FS$, $dBlk.x \times dBlk.y$ values are accessed for **A** and $dBlk.x$ values are accessed for **B**, but only FS values are necessary. Therefore, $dBlk.x > FS$ but less than the next power of two. So, $dBlk.x = \max\left(16, 2^{\lceil \log_2(FS) \rceil}\right)$. If FS is greater than the upper limit of $dBlk.x$ from Equation (5.12a), then $dBlk.x = 512$. Therefore,

$$dBlk.x = \min\left(\max\left(16, 2^{\lceil \log_2(FS) \rceil}\right), 512\right). \tag{5.18}$$

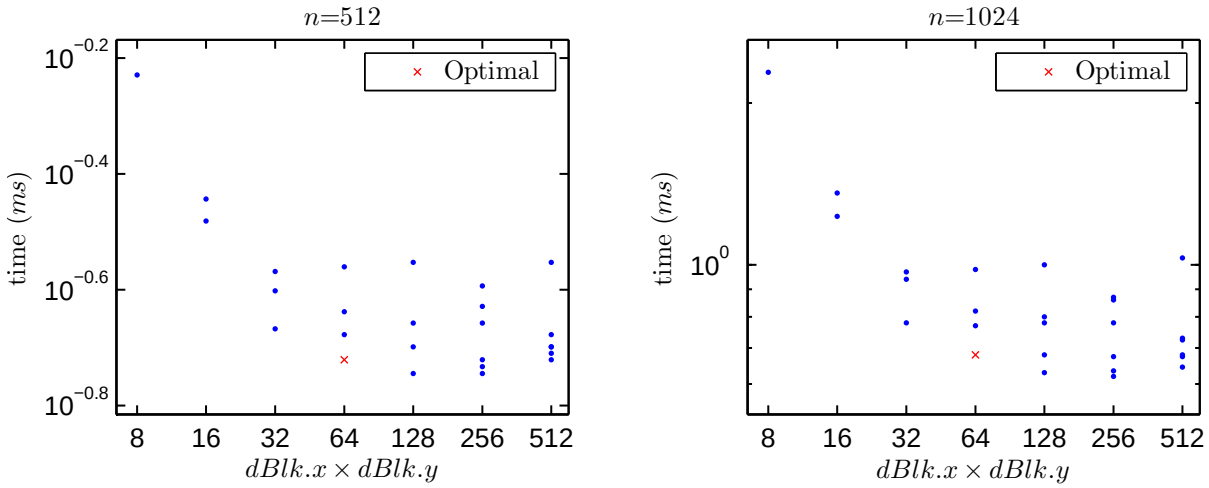
Step 3 of deriving optimal input parameters specifies the minimum amount of shared memory is allocated per block to increase the number of blocks assigned to each SM. In addition, minimizing the amount of shared memory allocated reduces time each HW in a block waits for synchronization since there are fewer HWs per block. Since SMemPerBlk = $(2 \times dBlk.x \times dBlk.y + dBlk.x) \times 4$ and $dBlk.x$ is constant from Equation (5.18), the minimum value of $dBlk.y$ from Equation (5.17) is utilized. Since $dBlk.y = \max\left(\frac{64}{dBlk.x}, 1\right)$ and from

Equations (5.18), (5.12c), and (5.12c),

$$
\begin{aligned}
dBlk.x &= \min\left(\max\left(16, 2^{\lceil \log_2(FS) \rceil}\right), 512\right) \\
dBlk.y &= \max\left(\frac{64}{dBlk.x}, 1\right) \\
dGrd.x &= \frac{n}{dBlk.x} \\
dGrd.y &= \frac{n}{dBlk.y}.
\end{aligned} \tag{5.19}
$$

Given n and FS, all input parameters are constant and therefore, no further steps are necessary to derive the optimal input parameters. So, Equation (5.19) defines the optimal input parameters for the shared memory implementation of convolution on a GPU utilizing the optimized computation and access pattern after fine-tuning.

Figures 5.15-5.17 depict measured computation time of all input parameters for GPU computation of convolution. In the figures in this section, the x-axis represents the number of threads per block ($dBlk.x \times dBlk.y$). Therefore, there are several combinations of $dBlk.x$ and $dBlk.y$ which yield an equivalent number of threads per block. Figure 5.15 depicts time for a filter size of 3.



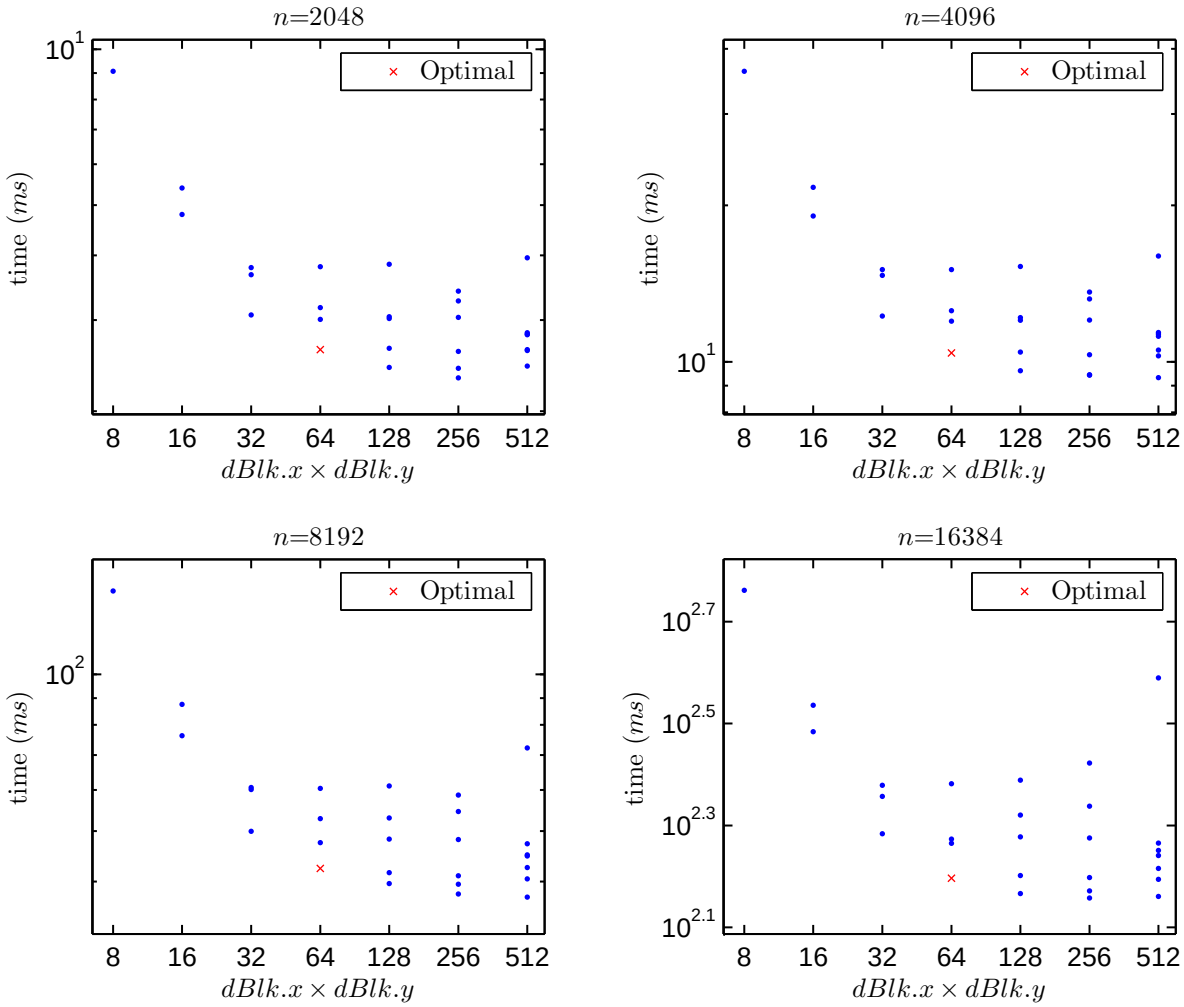


Figure 5.15: Comparison of input parameters: Computation time (ms) of all input parameters for convolution on the T10 GPU. FS=3.

For a filter size of 3, derivation of optimal input parameters yields computation time, on average, within 9.6% of the minimum measured time. Although the percentage difference between the optimal input parameters and the minimum time is greater than Mv and MM, the worst-case utilizing optimal input parameters (n = 8192) yields time within 12.1% of the minimum. In the best-case (n = 512), utilizing optimal input parameters yields time within 5.3% of the minimum. Computation time for a filter size of 63 is illustrated in Figure 5.16.

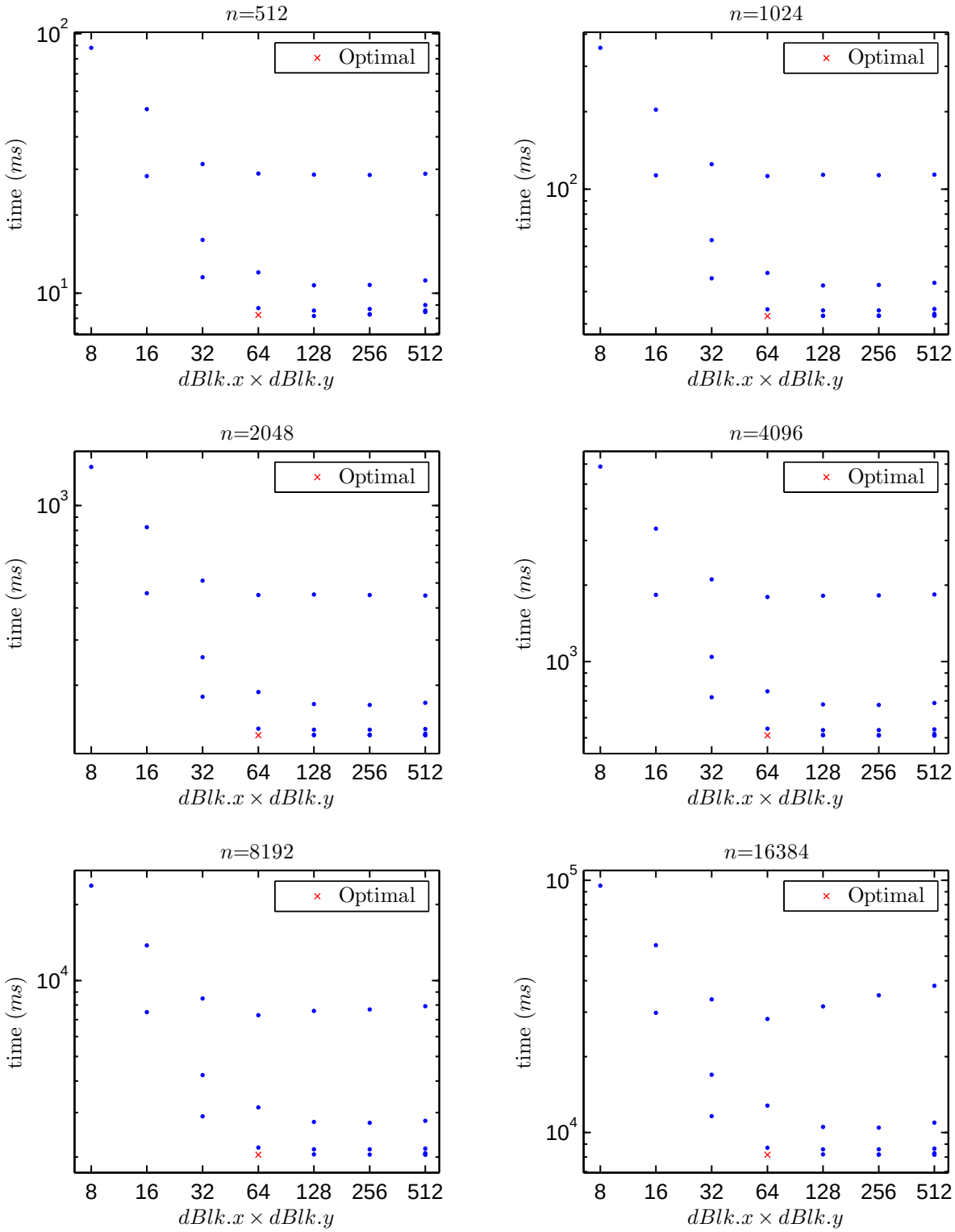
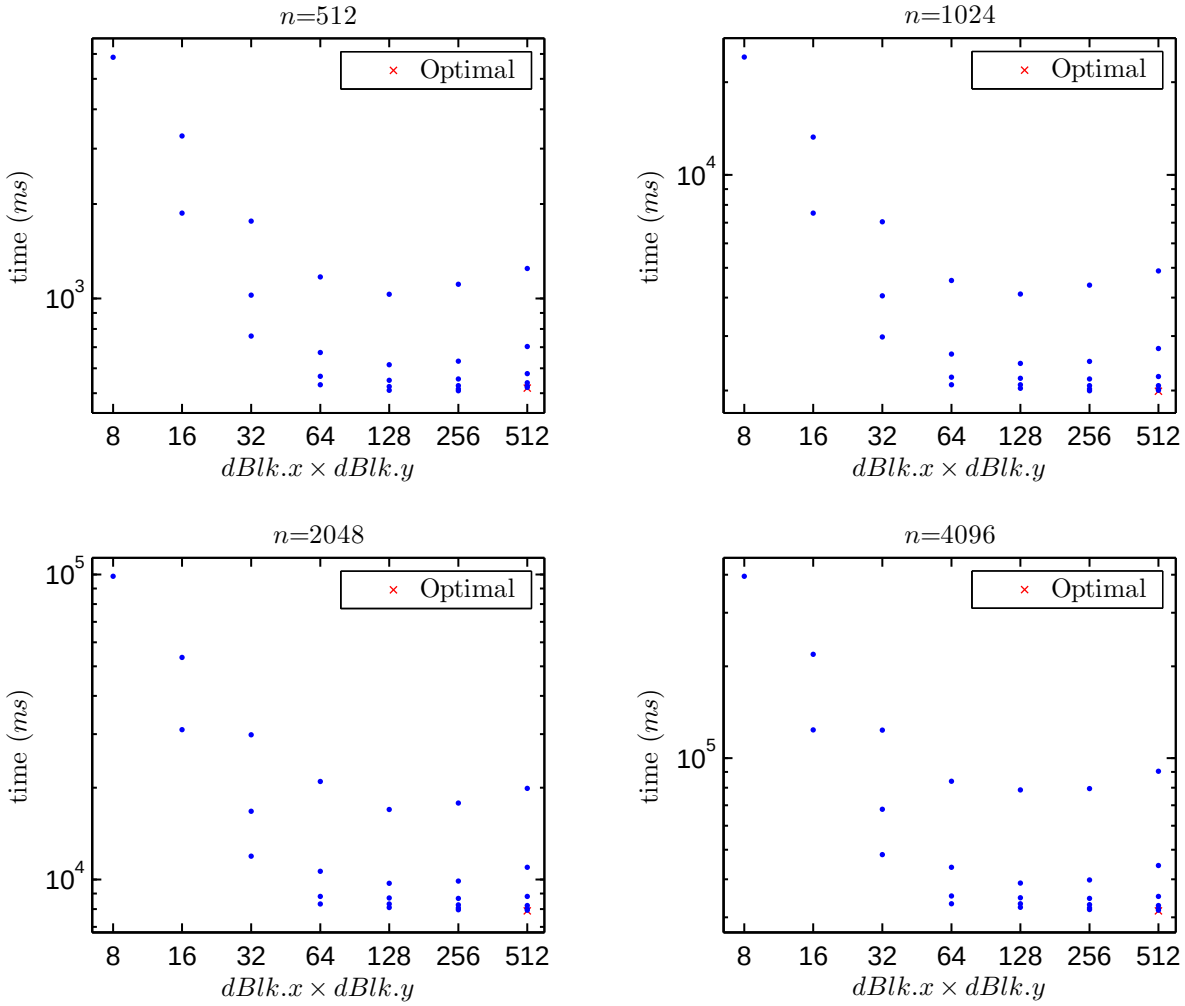


Figure 5.16: Comparison of input parameters: Computation time (ms) of all input parameters for convolution on the T10 GPU. FS=63.

As illustrated in Figure 5.16, for a filter size of 63, derivation of optimal input parameters yields computation time, on average, within 0.2% of the minimum measured time. The percentage difference between optimal input parameters and minimum time is much less for a filter size of 63 than for a filter size of 3. In addition, the percentage difference is roughly equivalent to the difference measured for Mv and MM. In the worst-case utilizing optimal input parameters ($n = 512$), the time is within 1.0% of the minimum measured. The time for the best-case utilizing optimal input parameters ($n = \{1024, 2048\}$) is the minimum measured. Figure 5.17 depicts computation time for a filter size of 513.



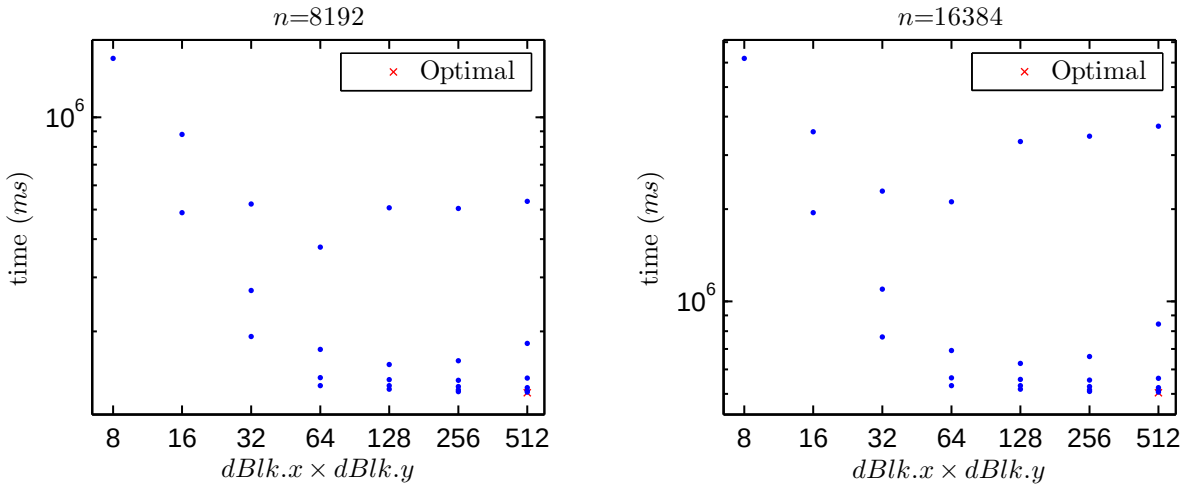


Figure 5.17: Comparison of input parameters: Computation time (ms) of all input parameters for convolution on the T10 GPU. FS=513.

For a filter size of 513, derivation of optimal input parameters yields computation time, on average, within 0.3% of the minimum measured time. In the worst-case (n = 512), utilizing optimal input parameters yields time within 1.9% of the minimum measured. In the best-case ($n = \{1024, 2048, 4096, 8192, 16384\}$), utilizing optimal input parameters yields the minimum measured time. Similar to a filter size of 63, the percentage difference between optimal input parameters and minimum time is much less for a filter size of 513 than for a filter size of 3. Likewise, the percentage difference is roughly equivalent to the difference measured for Mv and MM. For all data and filter sizes tested, the worst-case time utilizing optimal input parameters occurs for a filter size of 3 and yields time within 12.1% of the minimum measured.

GPU Computation Summary

After Step 5, the parallelization procedure is developed to determine the optimal partitioning of computation between the CPU and GPU. Therefore, the procedure with regards to minimizing GPU computation time is complete. So, this section illustrates maximum effective bandwidth of each aforementioned step of the parallelization procedure as it pertains

to GPU computation time. In addition, a comparison of the procedure to theoretical bandwidth is illustrated. For each figure in this subsection, *Naïve* denotes the global memory implementation, *Sh. Mem.* denotes the shared memory implementation, *O.C.P.* denotes the optimized computation pattern utilizing shared memory, and *O.A.P.* denotes the optimized access pattern utilizing the optimized computation pattern and shared memory. *P.P.* denotes the entire parallelization procedure with regards to GPU computation which utilizes shared memory, optimized computation and access patterns, fine-tuning adjustments, and optimal input parameters.

Figures 5.18-5.20 depict maximum effective bandwidth of convolution for each step of the parallelization procedure, in addition to the theoretical bandwidth. Figure 5.18 depicts maximum effective bandwidth for a filter size, FS, of 3.

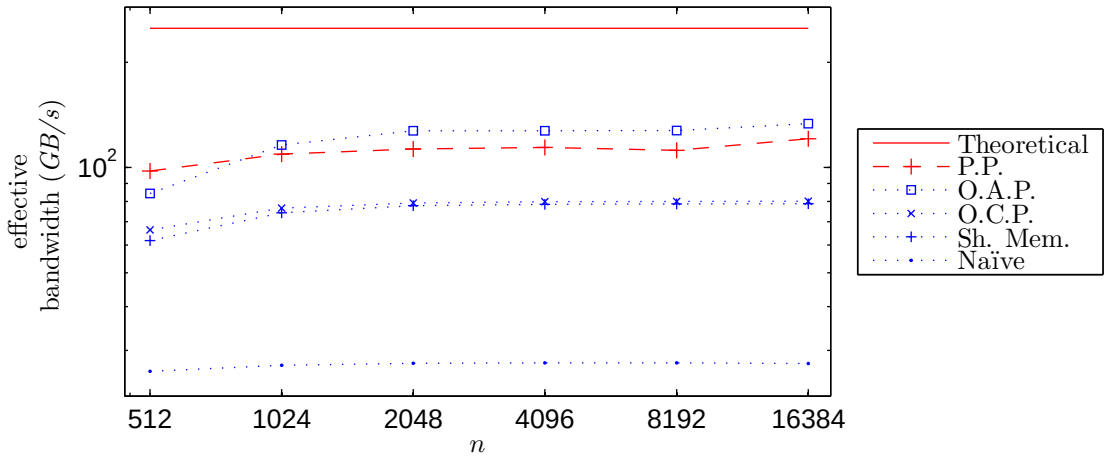


Figure 5.18: Comparison of effective bandwidth (GB/s) for convolution on the T10 GPU. FS=3.

The parallelization procedure yields a speedup, on average, of 4.07 compared to the naïve implementation. The minimum (n = 512) and maximum (n = 16384) speedups are 3.74 and 4.39, respectively. The procedure yields effective bandwidth, on average, as 39.0% of the theoretical bandwidth. Worst-case (n = 512) and best-case (n = 16384) utilizing the procedure yield effective bandwidths of 39.0% and 48.2% of the theoretical, respectively. As mentioned, the procedure yields computation time utilizing optimal input parameters, on

average, within 9.6% of the minimum. Therefore, as shown in Figure 5.18, the maximum effective bandwidth is slightly higher for the *O.A.P.* than for the *P.P.* which includes the optimal input parameters. Regardless, the procedure yields a significant speedup compared to the naïve implementation. Figure 5.19 depicts effective bandwidth for a filter size of 63.

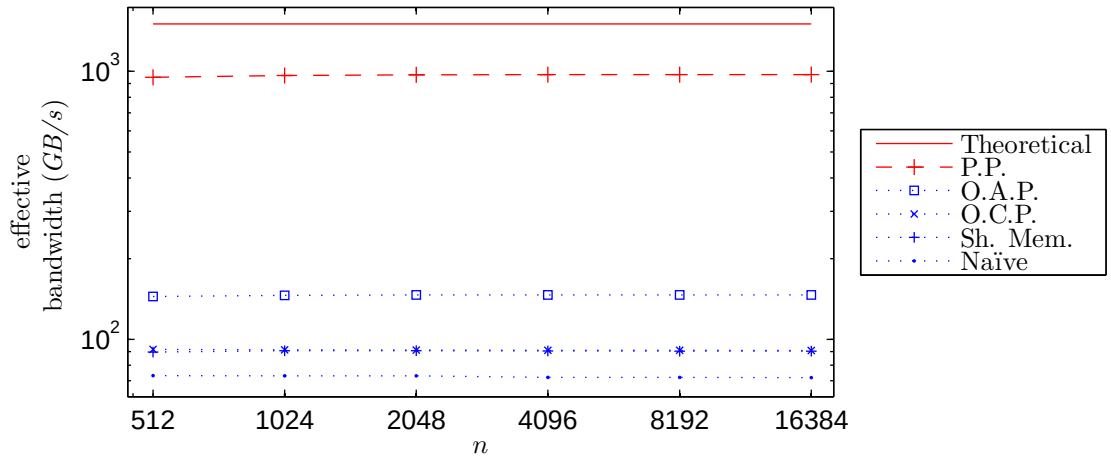


Figure 5.19: Comparison of effective bandwidth (GB/s) for convolution on the T10 GPU. FS=63.

In Figure 5.19, the large difference between the *O.A.P.* and the procedure is due to the effect on computation time from fine-tuning being dependent on filter size. For a small filter size, Figure 5.18, the *O.A.P.* slightly outperforms the procedure for some data sizes. However, for a filter size of 63, Figure 5.19, the procedure significantly outperforms the *O.A.P.* for all data sizes. Regardless, the procedure yields a speedup, on average, of 13.31 compared to the naïve implementation. The minimum (n = 512) and maximum (n = 16384) speedups are 12.85 and 13.52, respectively. The procedure yields effective bandwidth, on average, of 64.2% of the theoretical bandwidth. In the worst-case (n = 512), effective bandwidth is 62.6% of the theoretical. In the best-case (n = 16384), the procedure yields 64.6% of the theoretical bandwidth. Lastly, effective bandwidth for a filter size of 513 is depicted in Figure 5.20.

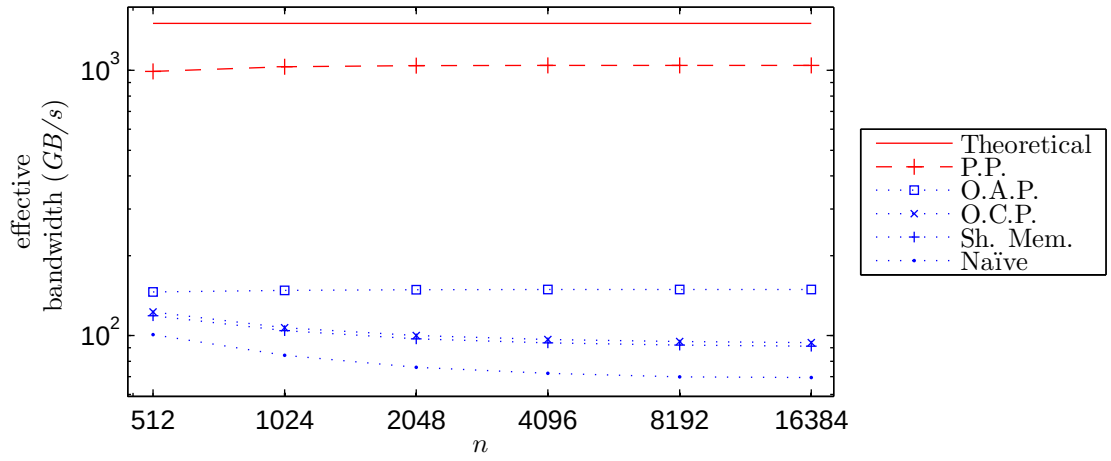


Figure 5.20: Comparison of effective bandwidth (GB/s) for convolution on the T10 GPU. FS=513.

Similar to a filter size of 63, the large difference between the $O.A.P.$ and the procedure is due to the effect on computation time from fine-tuning being dependent on filter size. The procedure yields a speedup, on average, of 13.38 compared to the naïve implementation. The minimum (n = 512) and maximum (n = 16384) speedups are 9.82 and 15.04, respectively. The procedure yields effective bandwidth, on average, of 68.7% of the theoretical bandwidth. In the worst-case utilizing the procedure (n = 512), effective bandwidth is 65.9% of the theoretical. In the best-case (n = 16384), the procedure yields 69.5% of the theoretical bandwidth. Therefore, for all data and filter sizes tested, the parallelization procedure significantly reduces GPU computation time compared to the naïve implementation.

5.1.6 Computation Partitioning

The last step of the parallelization procedure, Step 6, is to determine the optimal partitioning of computation between the CPU and GPU. For convolution, $2n^2FS^2$ float values are necessary for computation. If $FS \leq 15$, each thread in a HW reuses FS float values from global memory. However, if $FS > 15$, each thread in a HW reuses 16 float values from global memory. Therefore, $\frac{8n^2FS^2}{\min(FS,16)}$ bytes are necessary from global memory and the theoretical bandwidth is $\min(FS,16)94GB/s$. Therefore, $T_{comp.}^{GPU} = \frac{\frac{8n^2FS^2}{1024^3}}{\min(FS,16)94}$. Communication requires

two CPU to GPU transfers and one GPU to CPU transfer. Substituting for b, c_0, and c_1 into Equations (3.14) and (3.15) and simplifying yields the communication time in seconds. Since communication time is dependent on FS and n, if $FS < 256$,

$$T_{comm.} = 3.02 \times 10^{-9} FS^2 + \begin{cases} 5.00 \times 10^{-9} n^2 + 5.00 \times 10^{-5} & \text{if } n < 256 \\ 3.06 \times 10^{-9} n^2 + 9.60 \times 10^{-5} & \text{if } 256 \leq n < 512 \\ 2.30 \times 10^{-9} n^2 + 4.17 \times 10^{-4} & \text{if } n \geq 512. \end{cases}$$

If $256 \leq FS < 512$,

$$T_{comm.} = 1.08 \times 10^{-9} FS^2 + \begin{cases} 5.00 \times 10^{-9} n^2 + 9.60 \times 10^{-5} & \text{if } n < 256 \\ 3.06 \times 10^{-9} n^2 + 1.42 \times 10^{-4} & \text{if } 256 \leq n < 512 \\ 2.30 \times 10^{-9} n^2 + 4.63 \times 10^{-4} & \text{if } n \geq 512. \end{cases}$$

If $FS \geq 512$,

$$T_{comm.} = 1.08 \times 10^{-9} FS^2 + \begin{cases} 5.00 \times 10^{-9} n^2 + 2.27 \times 10^{-4} & \text{if } n < 256 \\ 3.06 \times 10^{-9} n^2 + 2.73 \times 10^{-4} & \text{if } 256 \leq n < 512 \\ 2.30 \times 10^{-9} n^2 + 5.94 \times 10^{-4} & \text{if } n \geq 512. \end{cases}$$

From Lines 1-2 of Listing 4.18, computation is partitioned for CPU execution if $T_{comp.}^{CPU} \leq T_{comp.}^{GPU} + T_{comm.}$. Therefore, after substitution, if

$$\frac{\frac{8n^2FS^2}{1024^3}}{\left(\frac{3}{\sqrt{FS}} - \frac{n}{20000} + 2.7\right)} \leq \frac{\frac{8n^2FS^2}{1024^3}}{\min(FS, 16)94} + T_{comm.},$$

computation is performed on the CPU to minimize execution time. Assuming the minimum for FS (3), solving for n yields $T_{comp.}^{CPU} \leq T_{comp.}^{GPU} + T_{comm.}$ if $n \leq 64$. Therefore, if $FS = 3$ and $n > 64$, computation is performed on the GPU to minimize execution time. If $FS \approx n$,

solving for n yields $T_{comp.}^{CPU} \leq T_{comp.}^{GPU} + T_{comm.}$ if $n \leq 11$. Therefore, if $FS \approx n > 11$, computation is performed on the GPU.

Figures 5.21-5.23 illustrate measured execution time for convolution. Since BLAS routines do not include convolution, CPU execution time is measured time for a non-optimized C implementation. Likewise, CUBLAS does not include convolution, so it is excluded. Figure 5.21 depicts execution time for convolution with a filter size of 3.

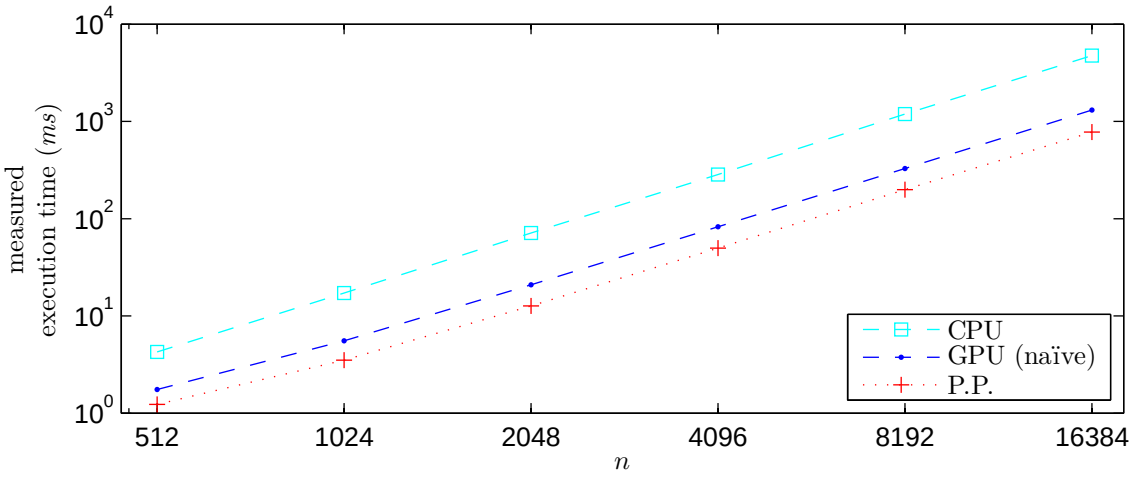


Figure 5.21: Comparison of execution time (ms) for convolution on the T10 GPU. $FS = 3$.

In Figure 5.21, the parallelization procedure yields a speedup, on average, of 5.30 compared to the CPU implementation, and 1.61 compared to the naïve implementation. The procedure yields execution time, on average, within 11.9% of the theoretical minimum execution time. In the worst-case utilizing the procedure ($n = 8192$), execution time is within 13.2% of the theoretical minimum. Best-case utilizing the procedure ($n = 512$) yields time within 11.1% of the theoretical minimum. Similar to Mv, the speedup compared to the naïve implementation in Figure 5.21 is less than in Figure 5.18 due to communication time. For convolution with a small filter size, the communication time is a significant portion of execution time. Therefore, large speedups measured for computation of convolution, with a small filter size, have less of an impact on execution time since the execution time includes communication and computation time.

Execution time for convolution with a filter size of 63 is illustrated in Figure 5.22.

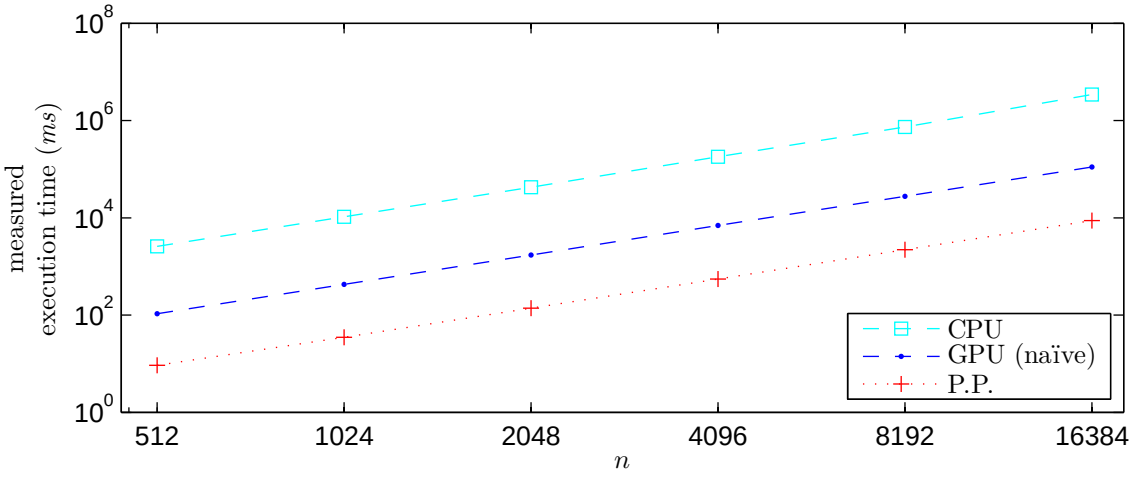


Figure 5.22: Comparison of execution time (ms) for convolution on the T10 GPU. FS = 63.

The procedure yields a speedup, on average, of 323.21 compared to the CPU implementation, and 12.33 compared to the naïve implementation. In Figure 5.22, for a filter size of 63, communication time is a small portion of execution time compared to a filter size of 3. Therefore, the procedure yields a speedup in execution time similar to the speedup achieved for computation only. The procedure yields execution time, on average, within 33.0% of the theoretical minimum execution time. Worst-case utilizing the procedure (n = 512) yields time within 33.4% of the theoretical minimum. In the best-case (n = 1024), the procedure yields time within 32.9% of the theoretical minimum. The significant speedup compared to the CPU implementation is presumably due to the non-optimized CPU implementation. Lastly, figure 5.23 depicts execution time for convolution with a filter size of 513.

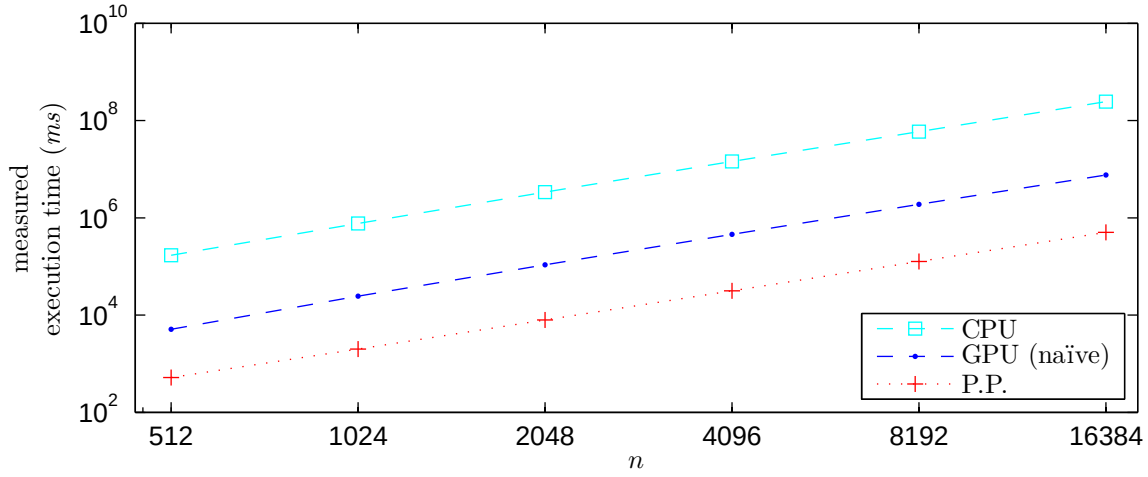


Figure 5.23: Comparison of execution time (ms) for convolution on the T10 GPU. $FS = 513$.

As illustrated in Figure 5.23, the procedure yields a speedup, on average, of 424.27 compared to the CPU implementation, and 13.36 compared to the naïve implementation. Similar to a filter size of 63, the procedure yields a speedup in execution time similar to the speedup achieved for computation only. This is due to communication time being a small portion of execution time. The procedure yields execution time, on average, within 31.3% of the theoretical minimum execution time. In the worst-case ($n = 512$), the procedure yields time within 34.0% of the theoretical minimum. In the best-case utilizing the procedure ($n = 16384$), the time is within 30.5% of the theoretical minimum. As mentioned for $FS = 63$, the significant speedup compared to the CPU implementation is presumably due to the non-optimized CPU implementation.

For all data and filter sizes tested, the parallelization procedure yields a worst-case speedup of execution time, on average, of 1.61 compared to the naïve GPU implementation. Therefore, results show the procedure is valid for convolution.

5.2 Conjugate Gradient

This section is an illustration of execution time of the parallelization procedure applied to an application utilizing a matrix-based computation. The conjugate gradient method [65] is utilized to further validate the procedure.

The conjugate gradient method is an iterative algorithm for solving a system of linear equations consisting of several vector-based computations (Level 1 BLAS) and Mv (Level 2 BLAS). The GPU (CUBLAS) implementation presented in the results utilizes CUBLAS routines for each vector- and matrix-based computation. Therefore, execution time is the sum of GPU computation and communication time. The CPU (ATLAS) implementation utilizes BLAS routines via ATLAS for all vector- and matrix-based computation and no communication is necessary. The naïve implementation and parallelization procedure utilize the GPU for matrix-based computations and CPU for vector-based computations. The naïve implementation utilizes a naïve GPU kernel. The procedure utilizes the GPU kernel developed by the procedure. Since the matrix in Mv is constant, it is necessary to transfer it to the GPU once. However, for the vector in Mv, it is necessary to transfer it to the GPU in each iteration. Likewise, it is necessary to transfer the result of Mv to the CPU in each iteration. Therefore, execution time of the procedure includes GPU computation, CPU computation, and all data transfers.

For the conjugate gradient method, the number of iterations necessary for the solution to converge is dependent on data size and input. Therefore, execution time is depicted for 8 and 256 iterations. Figure 5.24 illustrates measured execution time for the conjugate gradient method through 8 iterations.

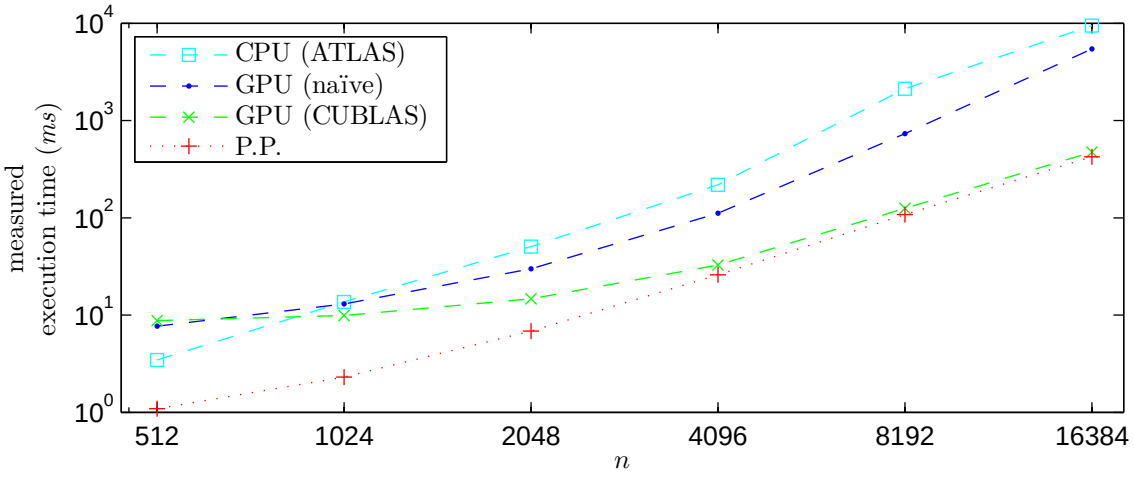


Figure 5.24: Comparison of execution time (ms) for 8 iterations of the conjugate gradient method on the T10 GPU.

As illustrated in Figure 5.24, the parallelization procedure yields a significant speedup, on average, of 11.15 compared to the ATLAS implementation. The minimum ($n = 512$) and maximum ($n = 16384$) speedups are 3.16 and 22.30, respectively. The procedure yields a speedup, on average, of 6.84 compared to the naïve implementation. The minimum ($n = 4096$) and maximum ($n = 16384$) speedups are 4.30 and 12.88, respectively. The procedure yields a speedup, on average, of 3.00 compared to the CUBLAS implementation. The minimum ($n = 16384$) and maximum ($n = 512$) speedups are 1.12 and 8.02, respectively. Although communication time is included in Figure 5.24, the procedure still yields significant increases in speedup of execution time for a small number of iterations of the conjugate gradient method.

Figure 5.25 illustrates measured execution time for the conjugate gradient method through 256 iterations.

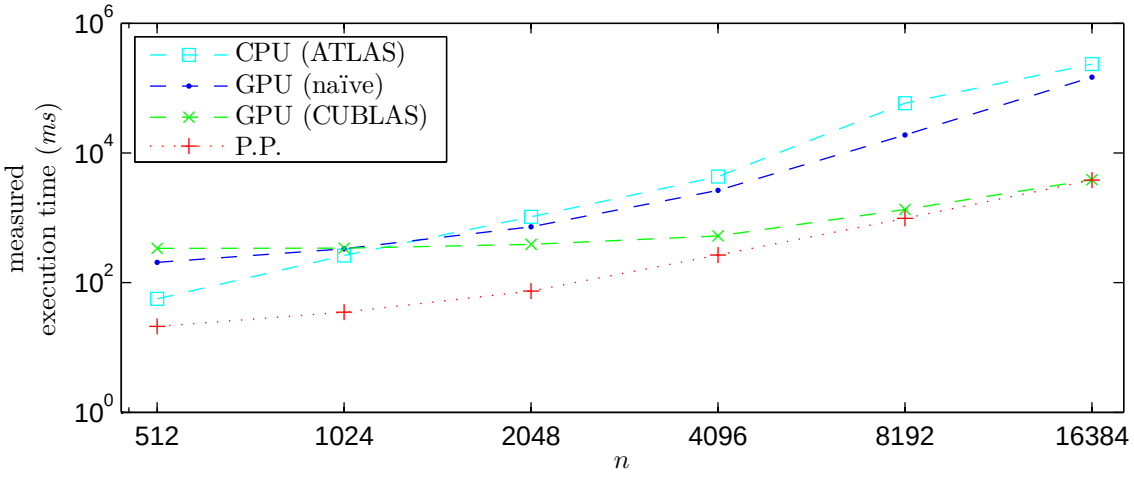


Figure 5.25: Comparison of execution time (ms) for 256 iterations of the conjugate gradient method on the T10 GPU.

The parallelization procedure yields a speedup, on average, of 26.89 compared to the ATLAS implementation. The minimum (n = 512) and maximum (n = 16384) speedups are 2.67 and 61.56, respectively. The procedure yields a speedup, on average, of 16.17 compared to the naïve implementation. The minimum (n = 1024) and maximum (n = 16384) speedups are 9.48 and 38.74, respectively. The procedure yields a speedup, on average, of 5.88 compared to the CUBLAS implementation. The minimum (n = 16384) and maximum (n = 512) speedups are 1.01 and 15.99, respectively. Similar to a small number of iterations of the conjugate gradient method, the procedure yields significant speedup of execution time for a large number of iterations. Comparing Figures 5.24 and 5.25 shows the speedup achieved by the procedure increases as the number of iterations increases. This is due to the amortization of communication time for transferring the matrix to the GPU. As mentioned, for the conjugate gradient method, the matrix is constant and must be transferred to the GPU only once. As the number of iterations increases, the contribution to execution time for transferring the matrix to the GPU decreases. Therefore, the speedup in execution time achieved by the procedure increases as the number of iterations increases.

As illustrated in Figures 5.24 and 5.25, the parallelization procedure yields minimum execution time of the conjugate gradient method for all data sizes and iterations tested.

Chapter 6

Conclusion

In this work, a parallelization procedure is developed to minimize execution time for matrix-based computations on a GPU. The procedure considers such factors as the placement of data, computation patterns, access patterns, fine-tuning, input parameters, communication time, and computation partitioning. The procedure is applied to Mv, MM, convolution, and the conjugate gradient method.

An accurate examination of the layout of GPU memory is presented. Partition camping in global memory is examined and the effects on GPU computation time are shown. Bank conflicts in shared memory are analyzed and effects are shown. Following the memory layout, execution metrics are formulated as functions of the input parameters to accurately represent the computational behavior of the GPU. The necessity of execution metrics is shown. In addition to representing computational behavior of the GPU, communication time between the CPU and GPU is modeled and estimates are presented. Computation time is modeled for the CPU utilizing ATLAS and for the GPU assuming a maximum bandwidth.

From the layout of GPU memory, execution metrics, and communication and computation time estimates, a parallelization procedure is developed through analysis and testing of Mv and MM to minimize execution time for matrix-based computations. The procedure determines placement of data in GPU memory and derives the optimized computation and access patterns to reduce computation time. Fine-tuning is performed on the GPU code to further reduce time. From the execution metrics, optimal input parameters are derived to yield the minimum computation time. Results show that utilizing optimal input parameters for Mv yields computation time, on average, within 1.8% of the minimum measured. For MM, results show that optimal input parameters yield time, on average, within 0.5%

of the minimum. Therefore, the derivation of optimal input parameters yields minimum or near-minimum computation time for each matrix-based computation tested.

For Mv, results show the parallelization procedure yields a speedup of GPU computation, on average, of 36.37 compared to the ATLAS CPU implementation, 18.98 compared to the naïve GPU implementation, and 1.47 compared to the CUBLAS GPU implementation. Results show for MM the procedure yields a speedup of GPU computation, on average, of 24.47 compared to the ATLAS CPU implementation, 14.58 compared to the naïve GPU implementation, and 1.07 compared to the CUBLAS GPU implementation.

Since the parallelization procedure is developed with analysis of measured data for Mv and MM, the procedure is applied to convolution and the conjugate gradient method to further validate the procedure. Results show for convolution that optimal input parameters yield time, on average, within 9.6% of the minimum for small filters and within 1.0% for large filters. For convolution with a small filter, results show the procedure yields a speedup of GPU computation, on average, of 26.75 compared to the non-optimized CPU implementation, and 4.07 compared to the naïve GPU implementation. For convolution with a large filter, results show the procedure yields a speedup of GPU computation, on average of 424.92 compared to the non-optimized CPU implementation, and 13.38 compared to the naïve GPU implementation.

For a small number of iterations of conjugate gradient, results show the procedure yields a speedup of execution time, on average, of 11.15 compared to the ATLAS CPU implementation, 6.84 compared to the naïve GPU implementation, and 3.00 compared to the CUBLAS implementation. For a large number of iterations of conjugate gradient, results show the procedure yields a speedup of execution time, on average, of 26.89 compared to the ATLAS CPU implementation, 16.17 compared to the naïve GPU implementation, and 5.88 compared to the CUBLAS implementation.

Therefore, the parallelization procedure developed minimizes execution time for matrix-based computations on a GPU.

Bibliography

[1] K. Fatahalian, J. Sugerman, and P. Hanrahan, "Understanding the efficiency of gpu algorithms for matrix-matrix multiplication," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, ser. HWWS '04. New York, NY, USA: ACM, 2004, pp. 133–137. [Online]. Available: http://doi.acm.org/10.1145/1058129.1058148

[2] C. Jiang and M. Snir, "Automatic tuning matrix multiplication performance on graphics hardware," in *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 185–196. [Online]. Available: http://dx.doi.org/10.1109/PACT.2005.10

[3] D. Tarditi, S. Puri, and J. Oglesby, "Accelerator: using data parallelism to program gpus for general-purpose uses," *SIGARCH Comput. Archit. News*, vol. 34, no. 5, pp. 325–335, Oct. 2006.

[4] W. Liu, Muller-Wittig, and B. Schmidt, "Performance predictions for general-purpose computation on gpus," in *Parallel Processing, 2007. ICPP 2007. International Conference on*, Sept. 2007, p. 50.

[5] S. Hong and H. Kim, "An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness," in *Proceedings of the 36th annual international symposium on Computer architecture*, ser. ISCA '09, 2009, pp. 152–163.

[6] ——. (2009) Memory-level and thread-level parallelism aware gpu architecture performance analytical model. [Online]. Available: http://www.cc.gatech.edu/~hyesoon/hong_report09.pdf

[7] ——, "An integrated gpu power and performance model," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 280–289, June 2010.

[8] K. Kothapalli, R. Mukherjee, M. Rehman, S. Patidar, P. Narayanan, and K. Srinathan, "A performance prediction model for the cuda gpgpu platform," in *High Performance Computing (HiPC), 2009 International Conference on*, Dec. 2009, pp. 463 –472.

[9] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded gpu using cuda," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, ser. PPoPP '08. New York, NY, USA: ACM, 2008, pp. 73–82. [Online]. Available: http://doi.acm.org/10.1145/1345206.1345220

[10] S. Ryoo, C. I. Rodrigues, S. S. Stone, J. A. Stratton, S.-Z. Ueng, S. S. Baghsorkhi, and W.-m. W. Hwu, "Program optimization carving for gpu computing," *J. Parallel Distrib. Comput.*, vol. 68, no. 10, pp. 1389–1401, Oct. 2008. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2008.05.011

[11] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W.-m. W. Hwu, "Program optimization space pruning for a multithreaded gpu," in *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, ser. CGO '08. New York, NY, USA: ACM, 2008, pp. 195–204. [Online]. Available: http://doi.acm.org/10.1145/1356058.1356084

[12] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu, "An adaptive performance modeling tool for gpu architectures," *SIGPLAN Not.*, vol. 45, no. 5, pp. 105–114, Jan. 2010.

[13] W.-M. Hwu, C. Rodrigues, S. Ryoo, and J. Stratton, "Compute unified device architecture application suitability," *Computing in Science Engineering*, vol. 11, no. 3, pp. 16 –26, May-June 2009.

[14] S.-Z. Ueng, M. Lathara, S. S. Baghsorkhi, and W.-M. W. Hwu, "Languages and compilers for parallel computing," J. N. Amaral, Ed. Berlin, Heidelberg: Springer-Verlag, 2008, ch. CUDA-Lite: Reducing GPU Programming Complexity, pp. 1–15. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-89740-8_1

[15] *CUBLAS*, NVIDIA, 2013. [Online]. Available: https://developer.nvidia.com/cublas

[16] "An updated set of basic linear algebra subprograms (blas)," *ACM Trans. Math. Softw.*, vol. 28, no. 2, pp. 135–151, Jun. 2002. [Online]. Available: http://doi.acm.org/10.1145/567806.567807

[17] *MAGMA*, Innovative Computing Laboratory at the University of Tennessee, 2013. [Online]. Available: http://icl.cs.utk.edu/magma/docs/

[18] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, 3rd ed. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999.

[19] R. Nath, S. Tomov, and J. Dongarra, "Accelerating gpu kernels for dense linear algebra," in *Proceedings of the 9th international conference on High performance computing for computational science*, ser. VECPAR'10. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 83–92. [Online]. Available: http://dl.acm.org/citation.cfm?id=1964238.1964250

[20] ——, "An improved magma gemm for fermi graphics processing units," *Int. J. High Perform. Comput. Appl.*, vol. 24, no. 4, pp. 511–515, November 2010. [Online]. Available: http://dx.doi.org/10.1177/1094342010385729

[21] Y. Li, J. Dongarra, and S. Tomov, "A note on auto-tuning gemm for gpus," in *Proceedings of the 9th International Conference on Computational Science: Part I*, ser.

ICCS '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 884–892. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-01970-8_89

[22] R. Nath, S. Tomov, T. T. Dong, and J. Dongarra, "Optimizing symmetric dense matrix-vector multiplication on gpus," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 6:1–6:10. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063392

[23] R. Nath, S. Tomov, and J. Dongarra, "Blas for gpus," in *Scientific Computing with Multicore and Accelerators*, 2010, ch. 4, pp. 57–80.

[24] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, "Dense linear algebra solvers for multicore with gpu accelerators," in *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, 2010, pp. 1–8.

[25] S. Tomov, R. Nath, and J. Dongarra, "Accelerating the reduction to upper hessenberg, tridiagonal, and bidiagonal forms through hybrid gpu-based computing," *Parallel Comput.*, vol. 36, no. 12, pp. 645–654, Dec. 2010.

[26] S. Tomov, J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid gpu accelerated manycore systems," *Parallel Comput.*, vol. 36, no. 5-6, pp. 232–240, Jun. 2010.

[27] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov, "A hybridization methodology for high-performance linear algebra software for gpus," vol. 2, pp. 473–484, 2011.

[28] ——, "Faster, cheaper, better - a hybridization methodology to develop linear algebra software for gpus," 2010.

[29] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical linear algebra on emerging architectures: The plasma and magma projects," in *Journal of Physics: Conference Series*, vol. Vol. 180, 2009.

[30] F. Song, S. Tomov, and J. Dongarra, "Efficient support for matrix computations on heterogeneous multi-core and multi-gpu architectures," 2011.

[31] J. Kurzak, S. Tomov, and J. Dongarra, "Autotuning gemm kernels for the fermi gpu," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 23, no. 11, pp. 2045–2057, 2012.

[32] M. Horton, S. Tomov, and J. Dongarra, "A class of hybrid lapack algorithms for multicore and gpu architectures," in *Proceedings of the 2011 Symposium on Application Accelerators in High-Performance Computing*, ser. SAAHPC '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 150–158. [Online]. Available: http://dx.doi.org/10.1109/SAAHPC.2011.18

[33] H. Ltaief, S. Tomov, R. Nath, P. Du, and J. Dongarra, "A scalable high performant cholesky factorization for multicore with gpu accelerators lapack working note #223."

[34] A. D. Malony, S. Biersdorff, S. Shende, H. Jagode, S. Tomov, G. Juckeland, R. Dietrich, D. Poole, and C. Lamb, "Parallel performance measurement of heterogeneous parallel systems with gpus," in *Proceedings of the 2011 International Conference on Parallel Processing*, ser. ICPP '11, 2011, pp. 176–185.

[35] J. W. Choi, A. Singh, and R. W. Vuduc, "Model-driven autotuning of sparse matrix-vector multiply on gpus," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '10, 2010, pp. 115–126.

[36] D. Grewe and A. Lokhmotov, "Automatically generating and tuning gpu code for sparse matrix-vector multiplication from a high-level representation," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-4, 2011, pp. 12:1–12:8.

[37] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder, "Sparse matrix solvers on the gpu: conjugate gradients and multigrid," *ACM Trans. Graph.*, vol. 22, no. 3, pp. 917–924, Jul. 2003. [Online]. Available: http://doi.acm.org/10.1145/882262.882364

[38] B. Boyer, J.-G. Dumas, and P. Giorgi, "Exact sparse matrix-vector multiplication on gpu's and multicore architectures," in *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, ser. PASCO '10, 2010, pp. 80–88.

[39] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving gpu performance via large warps and two-level warp scheduling," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44 '11, 2011, pp. 308–317.

[40] A. Bulu, J. R. Gilbert, and C. Budak, "Gaussian elimination based algorithms on the gpu," 2008.

[41] L. Y. Kah, A. Akoglu, I. Guven, and E. Madenci, "High performance linear equation solver using nvidia gpus," in *Adaptive Hardware and Systems (AHS), 2011 NASA/ESA Conference on*, 2011, pp. 367–374.

[42] A. El Zein and A. Rendell, "From sparse matrix to optimal gpu cuda sparse matrix vector product implementation," in *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, 2010, pp. 808–813.

[43] D. Q. Ren and R. Suda, "Power efficient large matrices multiplication by load scheduling on multi-core and gpu platform with cuda," in *Computational Science and Engineering, 2009. CSE '09. International Conference on*, vol. 1, 2009, pp. 424–429.

[44] S. Solomon and P. Thulasiraman, "Performance study of mapping irregular computations on gpus," in *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, 2010, pp. 1–8.

[45] G. Cummins, R. Adams, and T. Newell, "Scientific computation through a gpu," in *Southeastcon, 2008. IEEE*, 2008, pp. 244–246.

[46] Y. Sun and Y. Tong, "Cuda based fast implementation of very large matrix computation," in *Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2010 International Conference on*, 2010, pp. 487–491.

[47] N. P. Karunadasa and D. N. Ranasinghe, "Accelerating high performance applications with cuda and mpi," in *Industrial and Information Systems (ICIIS), 2009 International Conference on*, 2009, pp. 331–336.

[48] A. Bustamam, K. Burrage, and N. Hamilton, "Fast parallel markov clustering in bioinformatics using massively parallel graphics processing unit computing," in *Parallel and Distributed Methods in Verification, 2010 Ninth International Workshop on, and High Performance Computational Systems Biology, Second International Workshop on*, 2010, pp. 116–125.

[49] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel computing experiences with cuda," *Micro, IEEE*, vol. 28, no. 4, pp. 13 –27, July-Aug. 2008.

[50] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, April 2009, pp. 163 –174.

[51] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *Micro, IEEE*, vol. 28, no. 2, pp. 39 –55, March-April 2008.

[52] J. Nickolls and W. Dally, "The gpu computing era," *Micro, IEEE*, vol. 30, no. 2, pp. 56 –69, March-April 2010.

[53] D. B. Kirk and W. mei Hwu, *Programming Massively Parallel Processors*, 1st ed., 2010.

[54] G. Ruetsch and P. Micikevicius, *Optimizing Matrix Transpose in CUDA*, NVIDIA, 2009.

[55] A. M. Aji, M. Daga, and W.-c. Feng, "Bounding the effect of partition camping in gpu kernels," in *Proceedings of the 8th ACM International Conference on Computing Frontiers*, ser. CF '11. New York, NY, USA: ACM, 2011, pp. 27:1–27:10. [Online]. Available: http://doi.acm.org/10.1145/2016604.2016637

[56] J. Wu and J. JaJa, "Optimized strategies for mapping three-dimensional ffts onto cuda gpus," in *Innovative Parallel Computing (InPar), 2012*, May 2012, pp. 1 –12.

[57] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A gpgpu compiler for memory optimization and parallelism management," in *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '10. New York, NY, USA: ACM, 2010, pp. 86–97. [Online]. Available: http://doi.acm.org/10.1145/1806596.1806606

[58] A. Dasgupta, "Cuda performance analyzer," Master's thesis, Georgia Institute of Technology, May 2011. [Online]. Available: http://smartech.gatech.edu/jspui/bitstream/1853/39555/1/dasgupta_aniruddha_s_201105_mast.pdf

[59] N. B. Lakshminarayana and H. Kim, "Effect of instruction fetch and memory scheduling on gpu performance," in *Workshop on Language, Compiler, and Architecture Support for GPGPU*, 2010.

[60] R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimization of software and the ATLAS project," *Parallel Computing*, vol. 27, no. 1–2, pp. 3–35, 2001. [Online]. Available: http://math-atlas.sourceforge.net/

[61] T. Bradley and P. Micikevicius. (2009) Advanced c for cuda. NVIDIA. [Online]. Available: http://www.gputechconf.com/object/gtc2009-on-demand.html

[62] *HPC User Manual*, The Alabama Supercomputer Authority, 2010.

[63] V. Volkov and J. W. Demmel, "Benchmarking gpus to tune dense linear algebra," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 31:1–31:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=1413370.1413402

[64] *CUDA Programming Guide*, NVIDIA, 2013. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/

[65] M. R. Hestenes and E. Stiefel, "Methods of Conjugate Gradients for Solving Linear Systems," *Journal of Research of the National Bureau of Standards*, vol. 49, pp. 409–436, Dec. 1952.