Accelerating GPU Programs by Reducing Irregular Control Flow and Memory Access

January 2013

Tomohiro OKUYAMA

Accelerating GPU Programs by Reducing Irregular Control Flow and Memory Access

Submitted to
Graduate School of Information Science and Technology
Osaka University

January 2013

Tomohiro OKUYAMA

Published Papers

Journal Papers

1. Tomohiro Okuyama, Fumihiko Ino, and Kenichi Hagihara, "A Task Parallel Algorithm for Finding All-Pairs Shortest Paths Using the GPU," *International Journal of High Performance Computing and Networking*, vol.7, no.2, pp.87–98, April 2012.

2. Tomohiro Okuyama, Fumihiko Ino, and Kenichi Hagihara, "Fast Blocked Floyd-Warshall Algorithm on the GPU," *IPSJ Transactions on Advanced Computing Systems*, vol.3, no.2, pp.57–66, June 2010 (In Japanese).

International Conference Papers

1. Tomohiro Okuyama, Fumihiko Ino, and Kenichi Hagihara, "A Task Parallel Algorithm for Computing the Costs of All-Pairs Shortest Paths on the CUDA-compatible GPU," *Proceedings of the 6th International Symposium on Parallel and Distributed Processing with Applications (ISPA 2008)*, pp.284–291, Sydney, Australia, December 2008.

Domestic Conference Papers

1. Tomohiro Okuyama, Fumihiko Ino, and Kenichi Hagihara, "Fast Blocked Floyd-Warshall Algorithm on the GPU," *Proceedings of High Performance Computing Symposium 2010 (HPCS 2010)*, pp.9–16, January 2010 (In Japanese).

Oral Presentations

1. Kentaro Shigeoka, Tomohiro Okuyama, Fumihiko Ino, and Kenichi Hagihara, "A Parallel Method for Accelerating Parameter Sweep on the GPU," *IPSJ SIG Notes*, 2012-HPC-136, 8 pages, September 2012 (In Japanese).

2. Yoshiyuki Asai, Takeshi Abe, Masao Okita, Tomohiro Okuyama, Nobukazu Yoshioka, Shigetoshi Yokoyama, Masaru Nagaku, Kenichi Hagihara, and Hiroaki Kitano, "Multilevel modeling of Physiological Systems and Simulation Platform: PhysioDesigner, Flint and Flint K3 service," *Proceedings of the 12th IEEE/IPSJ International Symposium on Applications and the Internet (SAINT 2012)*, pp.215–219, Izmir, Turkey, July 2012.

3. Tadashi Yoshikawa, Tomohiro Okuyama, Masao Okita, Yoshiyuki Asai, Takeshi Abe, Taishin Nomura, Tetsuya Yagi, and Kenichi Hagihara, "Preliminary Evaluation of OpenMP-based Parallel Simulation by Focusing on Similarity of Formulas in Biophysical Models," *Proceedings of the 10th Symposium on Advanced Computing Systems and Infrastructures (SACSIS 2012)*, 2 pages, May 2012 (In Japanese).

4. Nobuya Fukui, Tomohiro Okuyama, Masao Okita, Takeshi Abe, Yoshiyuki Asai, Taishin Nomura, and Kenichi Hagihara, "An Agent Allocation for a parallel biophysical simulator *insilico*Sim," *Proceedings of the IEICE General Conference*, 2 pages, March 2012 (In Japanese).

5. Hiroto Kanda, Tomohiro Okuyama, Fumihiko Ino, and Kenichi Hagihara, "An Instrumentation Method for Analyzing Efficiency of Memory Access in CUDA Programs," *IPSJ SIG Notes*, 2012-HPC-133, 8 pages, March 2012 (In Japanese).

6. Masao Okita, Tomohiro Okuyama, Nobuya Fukui, Ryu Matsui, Takeshi Abe, Heien Eric, Yoshiyuki Asai, Taishin Nomura, and Kenichi Hagihara,"Automatic Parallelization of Heterogeneous Biological Simulator *insilico*Sim," *Proceedings of the 24th Bioengineering Conference, 2012 Annual Meeting of BED/JSME*, 2 pages, January 2012 (In Japanese).

7. Kentaro Shigeoka, Tomohiro Okuyama, Fumihiko Ino, and Kenichi Hagihara, "Evaluation of A Parallelization Technique for Parameter Sweep Applications using CUDA," *Proceedings of High Performance Computing Symposium 2012 (HPCS 2012)*, p.74, January 2012 (In Japanese).

8. Tomohiro Okuyama, Masao Okita, Takeshi Abe, Yoshiyuki Asai, Taishin Nomura, and Kenichi Hagihara, "Accelerating General and Heterogeneous Biophysical Simulation Using the GPU Interpreter for Solving ODEs," *Proceedings of the 4th Global COE International Symposium on Physiome and Systems Biology for Integrated Life Sciences and Predictive Medicine*, 1 pages, November 2011.

9. Ryu Matsui, Tomohiro Okuyama, Masao Okita, Takeshi Abe, Yoshiyuki Asai, Taishin Nomura, and Kenichi Hagihara, "Focusing on dependencies in biophysical models to improve scheduling in *insilico*Sim," *Proceedings of the Annual Meeting of IPSJ Kansai Branch 2011*, 7 pages, September 2011 (In Japanese).

10. Hiroto Kanda, Tomohiro Okuyama, Fumihiko Ino, and Kenichi Hagihara, "A Low Overhead Method for Timestamp Logging in CUDA Programs," *Proceedings of the Annual Meeting of IPSJ Kansai Branch 2011*, 3 pages, September 2011 (In Japanese).

11. Tomohiro Okuyama, Masao Okita, Takeshi Abe, Yoshiyuki Asai, Taishin Nomura, and Kenichi Hagihara, "Accelerating Interpreter of General Biophysical Simulatior '*insilicoSim*' on the GPU," *IPSJ SIG Notes*, 2011-HPC-130, 8 pages, July 2011 (In Japanese).

12. Tomohiro Okuyama, Fumihiko Ino, and Kenichi Hagihara, "Auto Tuned Floyd-Warshall Algorithm on the GPU," *Proceedings of the Work in Progress Session held in connection with the 19th Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP 2011)*, 2 pages, Ayia Napa, Cyprus, February 2011.

13. Hiroto Kanda, Tomohiro Okuyama, Fumihiko Ino, and Kenichi Hagihara, "A Log Generation Tool for Analyzing the Performance of CUDA Kernels," *IPSJ SIG Notes*, 2010-HPC-126, 7 pages, July 2010 (In Japanese).

14. Hiroto Kanda, Tomohiro Okuyama, Fumihiko Ino, and Kenichi Hagihara, "Preliminary Evaluation of A Trace Generation Tool for Time Series Analysis of CUDA kernels," *Proceedings of High Performance Computing Symposium 2010 (HPCS 2010)*, p.62, January 2010 (In Japanese).

15. Tomohiro Okuyama, Fumihiko Ino, and Kenichi Hagihara, "Accelerating Floyd Warshall Algorithm Using CUDA," *Proceedings of the Annual Meeting of IPSJ Kansai Branch 2008*, pp.35–38, October 2008 (In Japanese).

16. Tomohiro Okuyama, Fumihiko Ino, and Kenichi Hagihara, "Accelerating All-Pairs Shortest Path Problem Using CUDA," *IPSJ SIG Notes*, 2008-HPC-114, pp.145–150, March 2008 (In Japanese).

17. Tomohiro Okuyama, Fumihiko Ino, and Kenichi Hagihara, "Comparing Implementations of All-Pairs Shortest Path Problem on the GPU using CUDA," *Proceedings of High Performance Computing Symposium 2008 (HPCS 2008)*, p.58, January 2008 (In Japanese).

Abstract

The graphics processing unit (GPU) is recently used as a massively parallel processor to speed up general computation. However, the GPU can decrease the performance of irregular computation, because the GPU is based on the single instruction, multiple data (SIMD) architecture. The irregular computations here are conditional branches and memory accesses, which vary the behavior of threads depending on the input data. In particular, different control flow between threads causes redundant computations to follow each control flow. Moreover, uncoalesced memory accesses waste the memory bandwidth of the GPU. Therefore, there are many challenges to accelerate applications that depend on irregular computation.

This thesis presents GPU-based acceleration methods for three applications, aiming at developing techniques to efficiently process irregular computation on the GPU. We focus on irregular GPU programs that have similar threads in the entire program, although naive parallelization methods fail to exploit the similarity of threads. Our main approach is to gather similar threads for the SIMD operations before executing threads on the GPU. We achieve this preprocessing by observing the similarity of memory access pattern for the first application. For the third application, we use the similarity of operations that are executed by threads. For the second application, we evaluate another approach, which employs an algorithm that eliminates the irregularity by using a regular data structure instead of a pointer-based data structure. The details are described below.

First, we describe an acceleration method for finding the all-pairs shortest paths (APSPs) using the GPU. The APSP problem is a graph operation that finds shortest paths between all two vertices in a graph. This computation requires many uncoalesced memory accesses to refer to the graph data, while the memory bandwidth bounds the performance. Our method is based on an iterative algorithm that repeatedly solves the single-source shortest path (SSSP) problem in parallel on the GPU. We exploit the coarse-grained parallelism by using a task parallelization scheme that associates a task with an SSSP problem, in addition to the fine-grained parallelism in an SSSP problem. This scheme solves multiple SSSP problems at a time, allowing us to share the graph data on a fast on-chip memory, as well as reducing irregular memory accesses. As a result, the speedup over the existing SSSP-based

implementation ranges from a factor of 2.8 to that of 13, depending on the graph topology.

We next present acceleration methods for the Floyd-Warshall (FW) algorithm using the GPU, which is another algorithm to solve the APSP problem. This algorithm uses a matrix representation of a graph, which eliminates irregular control flow and memory accesses. The proposed method contains two variations, both designed to reduce data access to off-chip memory based on an iterative blocked FW (BFW) algorithm. The first method also reduces the number of instructions using registers rather than the shared memory. The other method increases the block size because it is inversely proportional to the amount of off-chip memory access. For graphs with 256–1024 vertices, both methods are 4% faster than an existing recursive BFW method. The first method achieves approximately 70% of peak computational performance.

Finally, we demonstrate a GPU-based general biophysical simulator, called Flint. With this application, the program for threads depends on the input data, as well as the data values. Therefore, it is required to reduce the difference of control flow between threads. Flint handles heterogeneous biophysical models described by a large set of ordinary differential equations (ODEs). It uses an internal bytecode representation of simulation-related expressions to handle various biophysical models built for general purposes. The interpretation of bytecodes causes a heavy use of conditional branches. To reduce the irregular branches, we preprocess the bytecodes, which groups the similar bytecodes to assign a bytecode group to a SIMD core of the GPU. In addition, each group is unified to a unified bytecode to reduce memory accesses. We then implement two acceleration methods for Flint using a GPU. The first method interprets multiple bytecodes in parallel on the GPU. The second method translates a model into a source code through the internal bytecode, which speeds up the compilation of the generated source codes, because the code size is diminished by the bytecode unification. The first method simulates a model containing approximately 40,000 expressions 24 times faster than that on a CPU core. The second method achieves a performance of 2.4 times higher than that of the former method.

These results show that the GPU can be used for accelerating applications that include irregular computation. In particular, the task parallel scheme used for the APSP problem can improve the throughput of computation that includes the same type of independent subproblems. The technique used for our biophysical simulator will be applied to other ODE-based simulations. Moreover, it can be applied to an application that assigns different operations to threads. These findings will contribute to the realization of a general technique for efficient processing of irregular computation on the GPU and other accelerators.

Acknowledgements

I would first like to express gratitude to my advisor, Professor Kenichi Hagihara, for his guidance and support during my years at Hagihara laboratory, and for advices and suggestions throughout this work. I would also like to sincerely thank the member of the thesis committee, Professor Toshimitsu Masuzawa, for taking the time to read this thesis and provide helpful questions and comments. I also offer my sincere thanks to Associate Professor Fumihiko Ino, who has guided my undergraduate study and this work and has gave me helpful advice and suggestions. I am deeply grateful to Assistant Professor Masao Okita, who has helped me in research skills.

I would also like to offer thanks to Professor Taishin Nomura of Graduate School of Engineering Science at Osaka University and Dr. Yoshiyuki Asai of the Open Biology Unit at Okinawa Institute of Science and Technology, for giving me the opportunity to join the collaborative project on biophysical simulations. I am deeply grateful to Mr. Takeshi Abe of the Open Biology Unit at Okinawa Institute of Science and Technology, for his support in developing the biophysical simulator Flint. I would like to thank Dr. Eric Heien, who has firstly developed the simulator that became a basis for Flint.

Finally, I am indebted to members of Hagihara laboratory, for their daily support and creative discussions.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

The graphics processing unit is a dedicated hardware component to accelerate graphics tasks, which is used with specialized application programming interfaces (APIs) such as OpenGL [3] and DirectX [4]. In recent years, the general-purpose computation on the GPU (GPGPU) [5] employs this hardware to speed up general scientific computation using the high computational performance of the GPU. For this purpose, NVIDIA provides a software development environment called the compute unified device architecture (CUDA) instead of graphics APIs. This environment allows us to use the GPU as a massively parallel processor that runs thousands of threads on hundreds of cores.

The parallel computing model of the CUDA-compatible GPU is called single instruction, multiple threads (SIMT) [6] model. With this model, the GPU runs multiple threads that process the same program, namely kernel. The GPU assigns a batch of threads called warp to a group of cores. Although, these cores process threads in the single instruction, multiple data (SIMD) fashion, programmers do not have to mind the SIMD width and the number of cores in the GPU. They can write a kernel as a sequential program. Each thread has a thread index to control its behavior, for instance, accessing different data by threads.

This characteristic facilitates the development of GPU programs. However, to achieve higher performance, developers should consider the underlying SIMD architecture. For example, if threads in a warp branch in a different way, the warp decreases the computational efficiency because it follows each control flow of threads. These branches are called divergent branches. In addition, the memory coalescing is important to utilize the wide memory bandwidth of the GPU. The coalesced memory access is a localized memory access from a warp.

Therefore, the GPU has been applied to applications that have data parallelism,

because these applications have a regular memory access pattern and all threads follow the same control flow. The regular memory access facilitates the coalescing of memory access on the GPU. Many researches show that GPUs achieve substantial performance for these applications, which reaches ten times or more speedups compared to that on the CPU.

Meanwhile, there are many challenges to accelerate irregular programs using SIMD processors. Irregular programs typically use pointer-based data structures or more complex data structures. These programs cause many irregular computation including irregular control flow and memory accesses, because the behavior of programs deeply depends on the data. There have been many researches to process these irregular programs on SIMD machines. A general approach is to emulate a multiple instruction, multiple data (MIMD) machine on a SIMD machine [7]. For the SIMD machines, a typical register-based programming model provides less flexibility, which requires programmers to explicitly operate on vector registers. To provide more flexibility, Shu and Wu [8] developed a runtime system to emulate MIMD threads on a SIMD machine. There is also another approach that statically translates a thread-based program to the program that runs on a SIMD machine [9].

For the GPU, CUDA provides a more flexible thread-based programming model compared to that for SIMD machines described above. However, the irregular computation decreases the computational efficiency of the GPU. For the GPU program, irregular control flow can cause divergent branches because threads can follow different flow each other. Irregular memory access can prevent memory coalescing. In addition, irregular computation can prevent the utilization of the fast on-chip shared memory. This memory enables us to reduce off-chip memory accesses by sharing data between threads. However, threads tend to refer to different data each other in irregular programs, resulting in no data that can be shared between threads.

In this thesis, we describe GPU-based acceleration methods for three applications that have irregular computation. The first two applications are programs to find all-pairs shortest paths (APSPs) using different algorithms. The third application is a general biophysical simulator. These applications have threads that operate in the similar way on the GPU, although naive parallelization methods run threads that execute different operations on SIMD cores of the GPU. Our basic approach is to statically rearrange the threads and data assignment to reduce irregular computation before executing the program on the GPU. Therefore, we also assume that the operations of threads are reproducible for the same input data. In other words, the thread having the same thread index executes the same operation, if we give the same input data. In particular, the control flow of thread is independent of random numbers and data that are written by concurrent memory writes without exclusive control.

The details of these applications are described below.

All-Pairs Shortest Path Problem Graph operations are typical irregular computation. These operations cause many irregular memory accesses to chase the pointers, because graphs are commonly represented by pointer-based data structures. In addition, the control flow depends on the given graph, which may cause many divergent branches on the GPU.

The APSP problem is a graph operation that finds shortest paths between all two vertices in a given graph. This problem has been applied to a wide variety of fields such as bioinformatics [10] and computer aided design (CAD) [11].

However, the APSP problem requires a large amount of computation. For instance, the Floyd-Warshall (FW) [12, 13] algorithm solves this problem in $O(|V|^3)$ time, where $|V|$ represents the number of vertices in a graph. A straightforward method for solving the APSP problem is to iteratively compute single-source shortest path (SSSP) for every source vertex. Dijkstra's algorithm [14] accelerated with a Fibonacci heap is known as a fast method to find an SSSP for a sparse graph. Using this algorithm, we can find APSPs in $O(|V||E| + |V|^2 \log |V|)$ time, where $|E|$ represents the number of edges in a graph. In contrast to algorithmic studies mentioned above, many researchers are trying to accelerate the algorithms using various accelerators, including GPUs [1, 15, 16], field-programmable gate arrays (FPGAs) [17], and clusters [18].

General Biophysical Simulator Numerical integration, referred to here as a simulation, of biophysical and physiological models enables researchers to analyze and understand various physiological functions. With the advance of measurement technology in physiology, the degrees of freedom used for mathematical modeling have rapidly increased. Large models now contain thousands of dynamic variables and mathematical expressions. Generally, the development of such models requires a number of model modifications, each of which is followed by a simulation that runs for millions of time steps. These time-consuming processes motivated us to accelerate lengthy simulations.

A biophysical simulator, called Flint, is designed to process general and heterogeneous biophysical models. This simulator numerically integrates a model described by a large set of ordinary differential equations. Flint uses an internal interpreter to simulate a variety of models built for general purpose. The interpreter causes many branches depending on the bytecodes that represent simulation related expressions.

1.2 Objectives

We focus on accelerating three applications, aiming at discussing techniques to implement irregular programs that run efficiently on the GPU.

Fast Computation of APSPs by SSSP-based Algorithm Using CUDA, Harish and Narayanan [1] present a fast SSSP-based method for large graphs, which iteratively computes SSSPs on the GPU. However, with this algorithm, threads perform in different ways because a thread is corresponding to a vertex in a graph. In particular, it can be further accelerated by memory access optimization. For example, the algorithm may be modified such that it fully uses the entire memory hierarchy, including fast but small on-chip shared memory. Therefore, the goal is to accelerate the computation of APSPs on the GPU by reducing irregularity in the memory accesses. For this purpose, we use a coarse-grained parallelism that exists between different SSSP problems.

Computation of APSPs Using Floyd-Warshall Algorithm For sparse graphs, our task parallel algorithm achieves reasonable speedup using the GPU. However, this method increases the execution time for dense graphs because its complexity depends on the number of edges, in addition to the number of vertices. On the other hand, the complexity of the FW algorithm depends only on the number of vertices. Therefore, we evaluate the performance of the FW algorithm on the GPU.

General Biophysical Simulator Using the GPU This application is an example of more complex and advanced application compared to the APSP problem. The computation, as well as the data, of this application depends on the given biophysical model. Therefore, the simulator is required to automatically parallelize the simulation for the GPU. In addition, the input model includes a variety of mathematical expressions. Consequently, a naive parallelization causes numerous divergent branches and degrades the performance of the GPU. This simulator must coordinate thread assignment to reduce divergent branches for efficiently utilizing the GPU.

1.3 Contributions of Thesis

The main results of this thesis are summarized as follows.

Task Parallel Algorithm for Finding APSPs Using the GPU We have developed a task parallel algorithm that computes multiple SSSPs in parallel on the GPU. Our method uses the coarse-grained parallelism between SSSP problems, in addition to the fine-grained parallelism in each SSSP problem. A task parallelization scheme is used to exploit the coarse-grained parallelism, which associates a task with an SSSP problem.

This scheme enables us to efficiently access graph data by sharing the data between threads on the GPU. Moreover, the use of two-level parallelism increases the number of threads on the GPU, which is beneficial to an efficient computation.

The experimental result shows that our method is 2.8 to 13 times faster than the iterative SSSP-based method, although the speedup depends on the graph topology.

Iterative Blocked FW Algorithm Using the GPU We have implemented a fast iterative blocked Floyd-Warshall (BFW) algorithm on the GPU. The proposed method contains two variations, both designed to reduce data access to off-chip memory because the bandwidth limits the performance of the FW algorithm. The first method applies a fast matrix multiplication routine to the computation, aiming at reducing the number of instructions. The other method uses a two-level blocking technique to reduce the on-chip memory usage.

As a result, both methods show 4% faster performance than an existing fast recursive BFW algorithm for 256–1024 vertices. For larger graphs, our matrix multiplication based method shows similar performance as that of the recursive method. The effective performance of this method is approximately 70% of peak computational performance, which indicates that our implementation successfully derives the performance of the GPU.

GPU-based Fast Simulation of General Biophysical Models We have developed two acceleration methods for Flint using the GPU. The first one simultaneously interprets multiple bytecodes on the GPU. It automatically parallelizes the simulation using a level scheduling algorithm. The bytecodes are reordered by their similarity to assign similar bytecodes to a warp, which reduces divergent branches. In addition, this method also unifies the similar bytecodes to a unified bytecode to reduce data amount of memory accesses. The second method translates a model into a CUDA code. This method generates the code from the unified bytecodes to diminish the generated code size. Otherwise, the compilation time increases to an impractical range.

The interpreter-based method achieves 24 times speedup over the CPU based simulation for a model with approximately 40,000 expressions. The translator-based method is up to 2.4 times faster than the interpreter-based method.

1.4 Outline of Thesis

The rest of this thesis is organized as follows. Chapter 2 summarizes an overview of GPU and CUDA. Chapter 3 and Chapter 4 present the acceleration methods for the APSP algorithm using the GPU and evaluate their performance. We describe an SSSP based method that explores the task parallelism in Chapter 3. The acceleration

of FW algorithm is described in Chapter 4. Chapter 5 presents a fast general biophysical simulator using the GPU and shows experimental results. Chapter 6 concludes this thesis and discusses the future work.

Chapter 2

Compute Unified Device Architecture

The GPU is a hardware component to accelerate graphics processing. The CUDA [6] framework provides a general-purpose parallel computing environment for the GPU, which enables the execution of thousands or more threads in parallel on the GPU.

Figure 2.1 shows an overview of the architecture of CUDA-compatible GPU. The GPU employs a hierarchical architecture that consists of several streaming multiprocessors (SMs), each having CUDA cores for processing threads. In an SM, these cores simultaneously execute the same instruction as single instruction, multiple data (SIMD) processors. The number of CUDA cores in an SM depends on the GPU architecture, while the number of SMs in a GPU varies by the GPU. The CUDA cores within the same SM are allowed to share on-chip memory called shared memory, which is as fast as registers. This memory hierarchy is useful to save the memory bandwidth between CUDA cores and off-chip memory, because it can be used as a software cache shared by multiple CUDA cores belonging to the same SM.

On the other hand, the off-chip memory, called device memory, has larger latency that is 400–600 clock cycles per memory transaction [6]. CUDA logically partitions this memory in some memory areas having different functions. The most general area is called global memory. All CUDA cores can read and write data to this memory. The CPU also can transfer data between the main memory and the global memory. CUDA also uses a part of device memory as the local memory of each thread, which is a thread local storage for temporal variables if the thread lacks registers.

Corresponding to its hierarchical processor architecture, CUDA has a hierarchical thread programming model (Figure 2.2). That is, threads are structured into equal-sized groups, each called a thread block (TB), while all threads execute the same program, namely a kernel. Programmers write the kernel as a C-like function. All threads simultaneously execute the same kernel with different thread indices to

perform SIMD computation. Threads in the same TB run on a single SM and have synchronization capability within them. These threads can also share data using the shared memory.

Meanwhile, a kernel cannot synchronize different TBs on the GPU, because each TB will be independently assigned to an SM. Therefore, developers have to write their kernel such that there is no data dependence among different TBs. Due to the same reason, the GPU does not have a mechanism that synchronizes all threads. Such global synchronization involves splitting the kernel into two pieces, which are then launched sequentially from the CPU. The synchronization is accomplished by terminating the first piece of kernel.

An SM processes a TB in the following way. Supposing that a TB is assigned to an SM, the SM splits the TB into groups of threads called warps. The number of threads in a warp, which is defined as 32 threads in current hardware, is called as the warp size. Each of warps is then processed by the SM in a SIMD fashion. Therefore, branching threads in the same warp will divergent the warp. Such divergent warps [6] significantly degrade the computational efficiency, because instructions must be serialized due to different control flows.

Generally, each SM executes multiple TBs in parallel, because the GPU architecture is designed to hide the memory access latency with independent computation of different warps. This also explains why TBs must be independent. Such independent TBs are useful to allow SMs to continue computation by switching the TBs that have to wait data from device memory. Therefore, it is better to assign multiple TBs to every SM. However, memory resources such as the shared memory and registers usually limit the number of TBs per SM.

To achieve higher performance, it is necessary to utilize the device memory bandwidth, which reaches more than 100 GB/s. For the global memory access, localized memory access from a warp is important to achieve the full utilization of this wide memory bus. This technique is called memory coalescing [6]. Using this technique, the global memory accesses issued from threads in a (half-) warp can be coalesced into a single memory transactions if the source/destination address satisfies an alignment requirement. Otherwise, the GPU issues multiple memory transactions if a warp accesses a wide address range. The requirements for memory coalescing differ according to the generation of GPU architecture, summarized in the following.

G80 and G90 GPUs (Tesla architecture) The G80 architecture is the first CUDA-compatible GPU architecture. The G90 GPUs add the support for atomic functions on the global memory, which provides atomic read-modify-write capabilities. These architectures have the most restricted coalescing rule; a thread with ID N within the half-warp should access address $base + N$, where $base$ is a multiple of 16 bytes. They also have no cache for the device memory. GPUs of these architectures have 8 CUDA cores per SM. The shared memory

size is 16 KB per SM.

We use these architectures for evaluations in Chapter 3 and Chapter 4. It should be noted that the proposed algorithm in Chapter 3 assumes the strict requirement to be executable on these older GPUs.

GT200 GPUs (Tesla architecture) The requirements of coalesced memory accesses are relaxed, such that threads in a half-warp can access to data in any order by a memory transaction if the data is in the same memory segment of size 32, 64, or 128 bytes. GT200 GPUs also have 8 CUDA cores and 16 KB of shared memory per SM.

We use this architecture in Chapter 4.

GF100 GPUs (Fermi architecture) These GPUs have 2-level cache for the device memory. Consequently, a global memory transaction is issued for a cache line (128 bytes). GPUs of this architecture have 32 or 48 CUDA cores per SM. The L1 cache and the shared memory share 64 KB of on-chip memory. The shared memory size is configurable: 16 KB or 48 KB per SM.

We evaluate our biophysical simulator described in Chapter 5 using a Fermi GPU having 32 CUDA cores per SM.

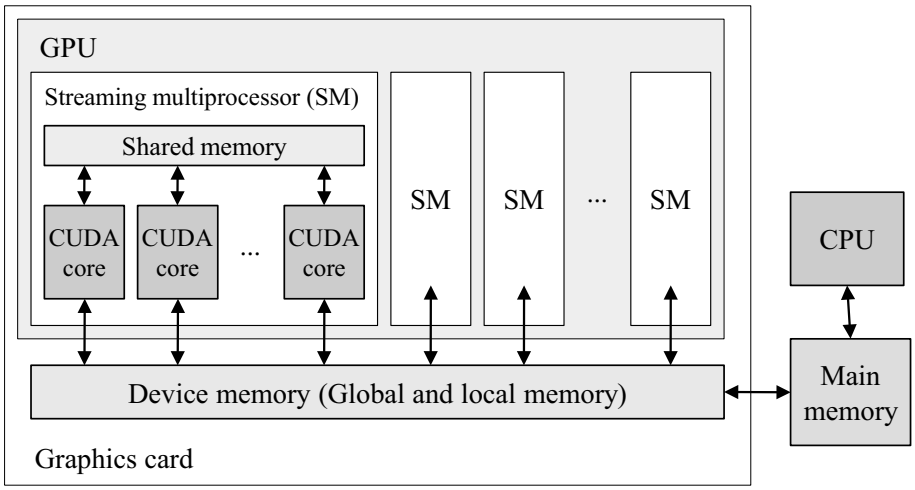


Figure 2.1: CUDA hardware model.

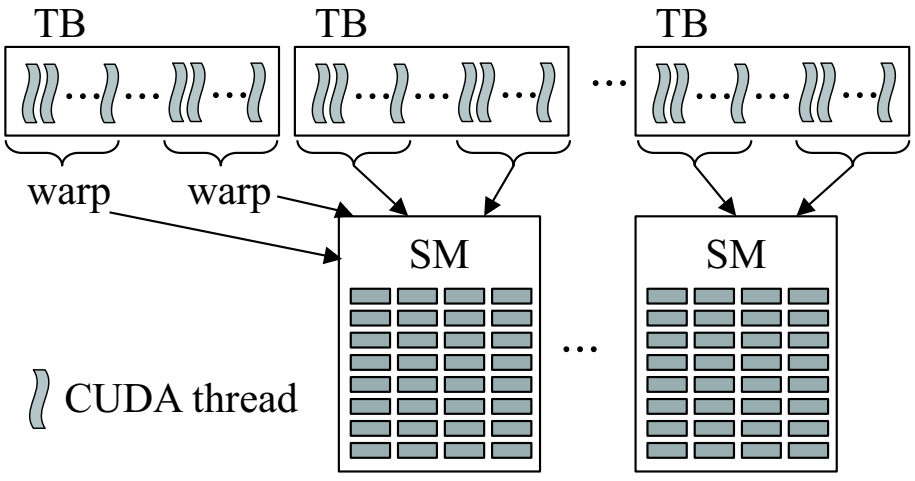


Figure 2.2: CUDA programing model.

Chapter 3

Task Parallel Algorithm for Finding APSPs

3.1 Introduction

In this chapter, we propose an SSSP-based algorithm to accelerate the cost computation of APSPs. The cost of a path here is given by the sum of the weights of edges composing the path. Our method enhances Harish's method [1] by exploiting not only off-chip memory but also on-chip memory. However, we use a different parallelization scheme for computing APSPs in order to save the bandwidth between off-chip memory and CUDA cores. In addition to the fine-grained parallelism exploited by the previous method, we exploit the coarse-grained parallelism existing between different SSSP problems. That is, the proposed scheme exploits task parallelism so that it solves in parallel multiple SSSP problems with different sources. This allows CUDA cores to simultaneously access the same data because each CUDA core takes the responsibility for solving one of the task-parallel problems. Such common access leads to an efficient use of on-chip shared memory, which is useful to reduce data accesses to off-chip memory. Furthermore, the proposed scheme contributes to achieve higher speedup with more parallel tasks and less synchronization on the GPU. We also describe how the paths can be recorded with their costs.

The rest of the chapter is organized as follows. Section 3.2 gives an introduction of related work. Section 3.3 describes the APSP problem and summarizes the iterative SSSP-based method. Section 3.4 presents our algorithm and Section 3.5 shows experimental results. Finally, Section 3.6 concludes the chapter.

3.2 Related Work

There are two widely known algorithms to find APSPs: an SSSP-based iterative algorithm and the FW algorithm. The former algorithm repeatedly solves SSSP problems with varying the source vertex $s \in V$ on a graph $G = (V, E, w)$, where V and E are the sets of vertices and edges in G, respectively. The function $w : E \to W$ assigns a weight to each edge, where W is a set of edge weights. The SSSP problem finds the shortest paths from s to all other vertices $v \in V$. A fast solution of SSSP problems is required to speed up the SSSP-based iterative algorithm. The Dijkstra's algorithm [14] and the Bellman-Ford algorithm [19, 20] are widely known algorithms to find SSSPs.

The Dijkstra's algorithm finds SSSPs for a directed graph having non-negative edge weights. This algorithm can reduce its computational amount using a priority queue. Using Fibonacci heap [21] to implement the priority queue, the complexity of this algorithm is reduced to $O(|V| \log |V| + |E|)$. However, the binomial heap is commonly used as the priority queue rather than the Fibonacci heap, because the Fibonacci heap uses a complex data structure. Using the binomial heap, the complexity of the Dijkstra's algorithm is $O((|V| + |E|) \log |V|)$. On the other hand, these improvements might not be suitable for the parallel computing, because it is not easy to efficiently operate on these heaps in parallel.

The Bellman-Ford algorithm requires $O(|E||V|)$ computations, which is larger than that of the Dijkstra's algorithm. However, this algorithm can handle graphs having negative edge weights. This algorithm has doubly-nested loops. The inner loop can be easily parallelized because there is no data dependency. However, the SSSP-based iterative algorithm using the Bellman-Ford algorithm tends to have larger complexity of $O(|E||V|^2)$ than that of the FW algorithm, because the number $|E|$ of edges is greater than the number $|V|$ of vertices in general.

Harish and Narayanan [1] present two APSP algorithms, namely the FW algorithm and the SSSP-based iterative algorithm, both implemented using CUDA. They demonstrate that the SSSP-based implementation takes approximately 10 seconds to obtain APSP costs for a graph of $|V| = 3072$ vertices. The speedup over the CPU-based FW implementation reaches a factor of 17. With respect to memory consumption, their SSSP-based algorithm requires $O(|V|)$ space while the FW algorithm requires $O(|V|^2)$ space, because the FW algorithm represents a graph in a matrix of size $|V|$, called adjacency matrix. This advantage allows us to deal with larger graphs, up to $|V| = 30,720$ vertices processed within two minutes. However, only off-chip memory is used because (1) there is no data that can be shared between CUDA cores and (2) the entire graph data is too large for 16 KB of on-chip memory.

Katz and Kinder [15] propose an optimized version of the FW implementation running on the CUDA-compatible GPU. Their method is based on an iterative blocked FW (BFW) algorithm proposed by [22], which partitions the adjacency

matrix into two dimensional tiles and iteratively updates these tiles. It takes 13.7 seconds to compute APSPs for a graph of $|V| = 4096$. A multi-GPU version is also presented to demonstrate the scalability of their method. It takes 354 seconds to process a graph of $|V| = 11,264$ vertices on two NVIDIA GeForce 8800 GT cards.

The recursive BFW method proposed by Buluç et al. [23], which recursively divides the adjacency matrix into tiles. This method uses a fast matrix multiplication method [24] to achieve higher performance. In particular, this method allows the GPU to efficiently run the matrix multiplication routine at a low depth of its recursion, because the routine processes large submatricies on the GPU. For $|V| = 4096$, their method computes APSPs in 1.01 seconds using GeForce 8800 Ultra, which reaches 126.7 Gflop/s.

Bleiweiss [25] implements Dijkstra's algorithm and A* search algorithm [26] with a priority queue using CUDA. Their implementations are designed for agent navigation in crowded game scenes, where multiple point-to-point shortest paths are simultaneously computed for smaller graphs. Thus, their problem is slightly different from our target problem.

Micikevicius [16] presents an OpenGL-based method that implements the FW algorithm on the GPU by mapping it to the graphics pipeline. The implementation runs on an NVIDIA GeForce 5900 Ultra, which demonstrates three times faster results compared with a 2.4 GHz Pentium 4 CPU. It takes approximately 203 seconds to compute APSPs for $|V| = 2048$.

Harish et al. [27] presented another matrix calculation based method for finding APSPs on the GPU. This algorithm requires $O(|V|^3 \log |V|)$ computations, which is larger than that of the FW algorithm. They adopt a lazy evaluation technique to accelerate their method for sparse graphs. This technique speeds up their method by 2–3 times. Their method finds APSPs in 0.3 seconds for a graph with $|V| = 2048$ using GeForce GTX 280, and in 2.2 seconds for $|V| = 4096$. This method can also handle a large graph that is too large to be stored in the off-chip memory of the GPU by dividing the matrix representation of the graph. For instance, this method processes a graph with $|V| = 30$K in 3072 seconds. In addition, they show that the lazy evaluation technique can be applied to the recursive BFW method by Buluç et al. and achieves a same range of 2–3 times speedup as that of their matrix calculation base method.

An FPGA-based method is proposed by [17]. They implement a tiled version of the FW algorithm and develop an analytical model to predict the performance for larger FPGAs. As compared with a CPU-based method running on a 2.2 GHz Opteron, their method reduces execution time for $|V| = 16,384$ from approximately four hours to 15 minutes, achieving a speedup of 15.2.

An auto-tuning approach is proposed by [28] to accelerate the iterative BFW algorithm on the CPU. Their method is optimized by cache blocking and single instruction, multiple data (SIMD) parallelization [29, 30]. It employs a two-level

blocking algorithm and selects the appropriate tile size for each level of cache. Using a 3.6 GHz Pentium 4 CPU, it takes 30 seconds to solve an APSP problem for $|V| = 4096$.

Finally, Srinivasan et al. [18] show a cluster approach to parallelize the FW algorithm on a distributed memory machine. However, the performance does not scale well with the number of computing nodes, because the data size $|V|$ seems to be small for the deployed cluster. A speedup of 1.2 is observed on a 32-node system when using a graph with $|V| = 4096$.

With respect to the FW implementation, the computation time mentioned above is limited by the memory bandwidth rather than the arithmetic performance: 76.8, 36.8, and 80.1 GB/s (9.6, 4.6, and 10 GFLOPS) on the FPGA [17], the CPU [28], and the GPU [15], respectively. Similarly, the SSSP-based method can be regarded as a memory-intensive application rather than a compute-intensive application. Actually, it requires two operations and at least two memory accesses to update the cost of a vertex. Thus, considering the ratio of the memory bandwidth to the arithmetic performance, the SSSP-based method requires at least 1 B/FLOP while the GPU employed by [1] provides 0.25 B/FLOP. Therefore, the performance will be increased if we save the memory bandwidth between off-chip memory and CUDA cores.

3.3 All-Pairs Shortest Path Problem

3.3.1 Definition

The APSP problem is to find the shortest paths between all pair (u, v) of vertices on a given graph $G = (V, E, w)$, where V is the set of vertices in G, E is the set of edges in G, $w : E \to W$ is a function that assigns a weight to each edge, and $u \in V$ and $v \in V$ are vertices in G. The edge $(u, v) \in E$ is a directed edge from the vertex $u \in V$ to the vertex $v \in V$. The edge weight $w(u, v) \in W$ represents the weight of the edge (u, v), where W is the set of edge weights. In this thesis, we assumes that $W = \mathbb{N}$ for simplicity. In other words, the edge weight is a non-negative integer. In this chapter, $|V|$ represents the number of vertices in G and $|E|$ is the number of edges in G.

This problem outputs the length of shortest paths between all pairs of vertices and the sequences of vertices that compose each shortest path. The path from the vertex $v_1 \in V$ to the vertex $v_k \in V$ is a sequence $v_1, v_2, \cdots, v_k$ of vertices that satisfy $(v_i, v_{i+1}) \in E$ for $1 \leq i < k$.

The length of path $v_1, v_2, \cdots, v_k$ is the sum of edge weights on the path, namely, $\sum_{i=1}^{k-1} w(v_i, v_{i+1})$. The shortest path from $u \in V$ to $v \in V$ is the path that has most shortest length of path among all paths between u and v.

Note that the sequence of vertices that compose the shortest path between any two vertices can be easily reconstructed from the cost of APSPs and the graph G. Figure 3.1 shows the reconstruction algorithm for the shortest path between vertex s and t. In this figure, D is the distance matrix, the element $D[u, v]$ of which represents the cost of the shortest path from vertex u to v.

Moreover, there is a demand to calculate the costs of paths in a short time. Therefore, the proposed methods focus on accelerating the computation of the cost of APSPs. However, to facilitate the reconstruction, we also show a method to record parent vertices with our task parallel algorithm in this chapter.

3.3.2 SSSP-based Iterative Algorithm

Harish and Narayanan [1] compute the costs of APSPs in a directed graph $G = (V, E, w)$ with positive weights. In the following, let $|V|$ and $|E|$ be the number of vertices and that of edges, respectively. Given a graph G, the method computes an SSSP $|V|$ times with varying the source vertex $s \in V$. This iteration is sequentially processed by the CPU, but each SSSP problem is solved in parallel on the GPU.

To solve an SSSP problem, an iterative algorithm [1] is implemented using CUDA. This algorithm associates every vertex $v \in V$ with cost c_v, which represents the cost of the current shortest path from the source s to the destination v. The algorithm then minimizes every cost until converging to the optimal state. This cost minimization is done by processing two phases alternatively: the scattering phase and the checking phase. In the scattering phase, all vertices try to minimize the costs of their neighbors in parallel. Figure 3.2 illustrates how this minimization works for a single vertex v. After this, the checking phase confirms whether the previous scattering phase has changed the costs of vertices.

Figure 3.3 shows this algorithm. Firstly, the cost of every vertex $v \in V$ except the source s is initialized to infinity, which means that v is not reachable from s at the initial state. On the other hand, the cost is set to zero for the source s. The cost minimization then begins at line 4 for a set M of vertices, where M is the modification set, which contains vertices whose neighbor(s) have not yet reached to the optimal state. Given such a vertex $v \in M$, the algorithm updates the cost c_n at line 9, for every neighbor $n \in V$ such that $(v, n) \in E$. The updated cost here is temporally stored to a variable u_n in order to check convergence later at line 13. Vertices that have changed their costs are added to set M for further minimization (line 14). The iteration stops when M becomes empty.

This algorithm requires synchronization between the scattering phase and the checking phase (line 12). Otherwise, some processing elements might overwrite the updated cost u_v after u_v has been confirmed to be minimal. It also should be noted that the algorithm requires atomic instructions to correctly process the scattering phase. Since multiple processing elements can update the same cost u_n at the same

time, we have to deal with the consistency of concurrent write accesses. Atomic instructions solve this issue but they are supported only on GPUs with compute capability 1.1 and higher [6]. If we lack this capability, the minimum cost u_n will be overwritten by a larger cost at line 9, resulting in a wrong result.

We now explain how Harish and Narayanan implement the algorithm on the GPU. As we mentioned earlier, there is no global synchronization mechanism in CUDA. Therefore, they develop two kernels, each for the scattering phase and for the checking phase. In both kernels, a thread is responsible for a vertex $v \in V$ in the graph. Thus, the cost minimization is parallelized using $|V|$ threads.

Figure 3.4 illustrates how a graph is represented in their kernels. They employ an adjacency list representation to store a graph in device memory. In this representation, each vertex data has a pointer to its adjacency list of edges. The adjacency list of vertex v here contains every neighboring vertex $n \in V$ such that $(v, n) \in E$. Harish and Narayanan convert these lists into arrays Va, Ea, and Wa, which store vertex set V, edge set E, and edge weights assigned by w, respectively. As shown in Figure 3.4, element $Va[v]$ has an index to array Ea, where the head of the adjacency list of v exists. Since all adjacency lists are concatenated into array Ea of size $|E|$, the adjacency list of vertex v is stored from element $Va[v]$ to $Va[v+1]-1$ in Ea. Similarly, the weight of edge $Ea[i]$ is stored in $Wa[i]$, where $0 \leq i \leq |E|-1$. In addition to the arrays mentioned above, they use additional arrays Ma, Ca, and Ua to store modification set M, current cost c_v, and updated cost u_v, respectively. Each of these arrays has $|V|$ elements and its index v corresponds to vertex v. They store these three arrays in device memory.

Figure 3.5 shows a pseudo code of the scattering kernel, which implements lines 6–11 in Figure 3.3. This kernel is invoked for every thread t_v, which is responsible for vertex $v \in V$. After this kernel execution, the CPU launches the second kernel to process the checking phase. This checking kernel updates array Ma and also sets a flag to true if any updated cost is found. The CPU then checks this flag to determine if the iteration should be stopped or not. Thus, the flag prevents the CPU from scanning the entire array Ma.

3.4 Task Parallel Algorithm

We now describe the proposed algorithm for accelerating the computation of APSPs on a directed, positively weighted graph G. Firstly, we present how our algorithm accelerates the cost computation of Harish's SSSP-based method. We then describe how the algorithm records the paths after the cost computation.

Input	D: distance matrix
	w: weight function
	s: source vertex
	t: destination vertex
Output	*path_array*: shortest path from s to t

```
1:  ReconstructPath(D, w, s, t)
2:    if (s = t)
3:      push(path_array, s)
4:      return success
5:    if (D[s,t] != ∞)
6:      push(path_array, s)
7:      for n in Successor(s)
8:        if (w(s,n) + D[n,t] = D[s,t])
9:          return SearchPath(D, n, t)
10:   return error /* The path does not exist.*/
```

Figure 3.1: Algorithm for reconstructing the sequence of vertices that compose the shortest path between s and t.

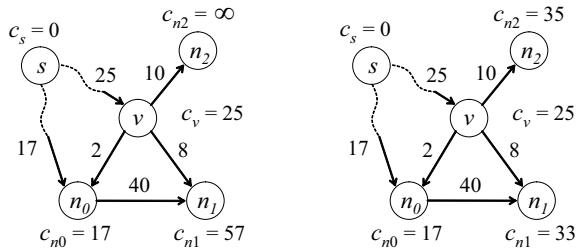


Figure 3.2: Cost minimization. (a) For each vertex v in the graph, (b) the costs of its neighbors n_0, n_1, and n_2 are updated in the scattering phase.

```
SSSP_Algorithm(s, V, E, w)                          /* s: source vertex */
 1: initialize c_v := ∞ and u_v := ∞ for all v ∈ V   /* u_v: updated cost of vertex v */
 2: c_s := 0                                        /* c_v: current cost of vertex v */
 3: M := {s}
 4: while M is not empty do
 5:     for each vertex v ∈ V in parallel do
 6:         if v ∈ M then                           /* Scattering phase */
 7:             remove v from M
 8:             for each neighboring vertex n ∈ V such that (v, n) ∈ E do
 9:                 u_n := min(c_n, c_v + w(v, n))  /* w(v, n): weight of edge (v, n) */
10:             end for
11:         end if
12:         synchronization
13:         if c_v > u_v then                       /* Checking phase */
14:             add v to M
15:             c_v := u_v
16:         end if
17:     end for
18: end while
```

Figure 3.3: Iterative algorithm for finding an SSSP from the source vertex $s \in V$.

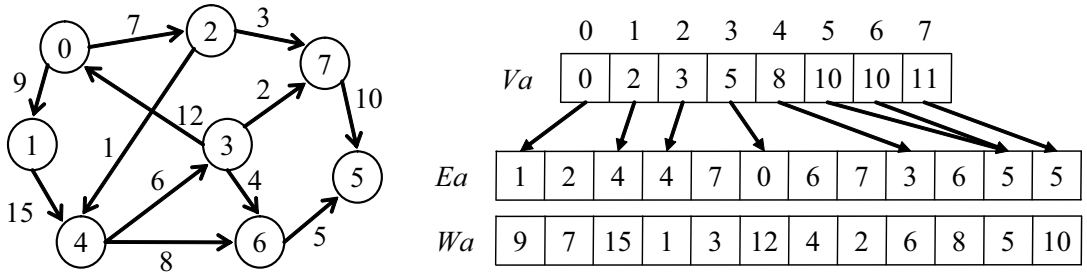


Figure 3.4: Adjacency list representation. Array Va stores the indices to the head of each adjacency list in Ea. Array Ea and Wa store adjacency lists of every vertex and edge weight, respectively.

```
SSSP_Scattering_Kernel(Va, Ea, Wa, Ma, Ca, Ua)
1: v := threadID
2: if Ma[v] = true then
3:     Ma[v] := false
4:     for i := Va[v] to Va[v + 1] − 1 do
5:         n := Ea[i]
6:         Ua[n] := min(Ua[n], Ca[v] + Wa[n])
7:     end for
8: end if
```

Figure 3.5: Pseudo code of scattering kernel [1]. This kernel is responsible for a single vertex v and updates the costs of its adjacent vertices.

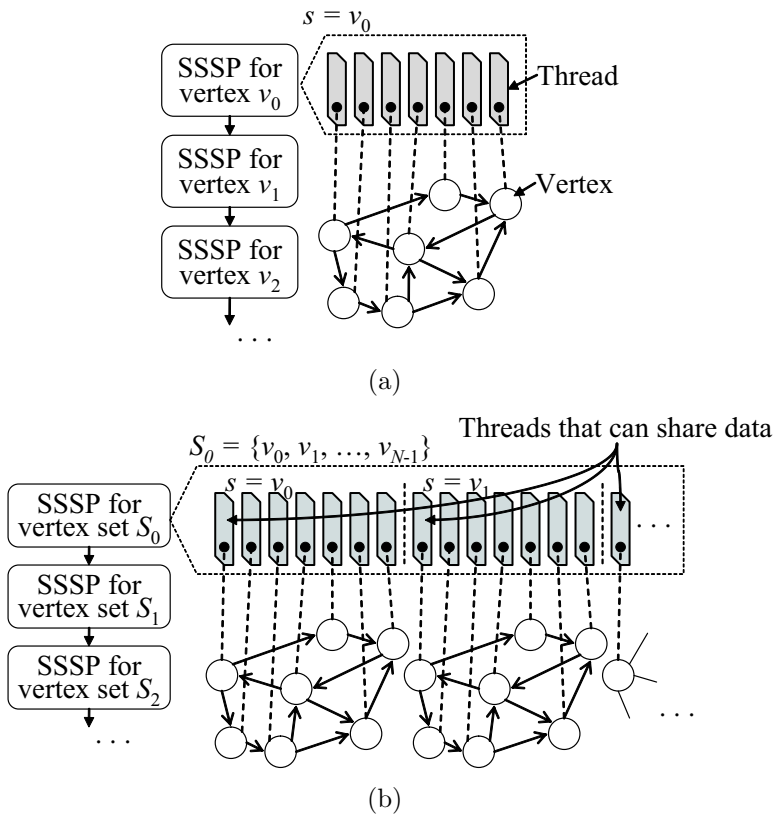


Figure 3.6: Comparison of parallelization scheme between (a) previous method [1] and (b) proposed method. Our kernel solves N SSSP problems at a time. The graph data is shared between threads that are responsible for the same vertex but in different SSSP problems.

3.4.1 Cost Computation

As shown in Figure 3.6(b), our algorithm computes N tasks in parallel, where a task deals with an SSSP problem. This task parallel scheme allows us to share graph data between different tasks. Another important benefit is that it allows the kernel to generate more threads at a launch. This leads to an efficient execution on the GPU, which employs a massively multithreaded architecture. Since the algorithm we use for a single SSSP problem is the same one developed by [1], we explain here how tasks are grouped to share the graph data.

Let p_s denote the SSSP problem with the source vertex $s \in V$. The APSP problem consists of $|V|$ SSSP problems $p_0, p_1, \ldots, p_{|V|-1}$ and there is no data dependence between them. Therefore, we can pack any N problems into a group to solve the group in parallel, where $1 \leq N \leq |V|$. Thus, the k-th group contains N SSSP problems $p_{kN}, p_{kN+1}, \ldots, p_{(k+1)N-1}$, where $0 \leq k \leq \lceil |V|/N \rceil - 1$. Let S_k denote the set of source vertices in the k-th group of N SSSP problems, where $0 \leq k \leq \lceil |V|/N \rceil - 1$. The proposed scheme then computes SSSPs from every source $s \in S_k$ on the GPU while it invokes this computation $\lceil |V|/N \rceil$ times sequentially from the CPU. We assign a vertex to a thread as Harish and Narayanan do in their algorithm. Accordingly, our kernel processes $N|V|$ threads in parallel while the previous kernel does $|V|$ threads, as shown in Figure 3.6.

Similar to Harish's algorithm, our algorithm consists of the scattering phase and the checking phase, as shown in Figure 3.7. However, our algorithm differs from the previous algorithm with respect to the use of shared memory in the scattering phase. Let $t_{v,s}$ be the thread, which is responsible for vertex $v \in V$ in problem p_s, where $s \in V$. In our algorithm, thread $t_{v,s}$ tries to update the cost $c_{n,s}$ (variable $u_{n,s}$ at line 13 in Figure 3.7), which represents the cost of neighboring vertex $n \in V$ in problem p_s. The graph data that can be shared between threads is edge (v, n) at line 12 and weight $w(v, n)$ at line 13, because both variables do not depend on the source vertex s. In order to share such s-independent data between threads, we structure a thread block (TB) such that it includes N threads $t_{v,kN}, t_{v,kN+1}, \ldots, t_{v,(k+1)N-1}$, which are responsible for the same vertex v but in different problems. Figure 3.8 shows the data structure more precisely. Note that every TB contains a multiple B of such N threads to increase the TB size for higher performance. Thus, such threads can save the memory bandwidth if they update the costs of their neighbors. However, it requires additional copy operations to duplicate data to shared memory. Therefore, threads might degrade the performance if such common access rarely occurs during execution.

With respect to the graph representation, our kernel uses a slightly different data structure from the previous kernel. We use the same arrays Va, Ea, and Wa, but they are partially shared between threads as mentioned before. The remaining arrays Ca, Ua, and Ma are separately allocated for every problem p_s, so that these

```
N_SSSPs_Algorithm(S_k, V, E, w)
 1: /* S_k: set of source vertices */
 2: /* u_{v,s}: updated cost of vertex v in problem p_s */
 3: /* c_{v,s}: current cost of vertex v in problem p_s */
 4: initialize c_{v,s} := ∞ and u_{v,s} := ∞ for all v ∈ V and s ∈ S_k
 5: initialize c_{s,s} := 0 for all s ∈ S_k
 6: add pair ⟨s, s⟩ to set M for all s ∈ S_k
 7: while M is not empty do
 8:     for each vertex v ∈ V and each source s ∈ S_k in parallel do
 9:         /* Scattering phase */
10:         if ⟨v, s⟩ ∈ M then
11:             remove ⟨v, s⟩ from M
12:             for each neighboring vertex n ∈ V such that (v, n) ∈ E do
13:                 u_{n,s} := min(c_{n,s}, c_{v,s} + w(v, n))
14:             end for
15:         end if
16:         synchronization
17:         /* Checking phase */
18:         if c_{v,s} > u_{v,s} then
19:             add ⟨v, s⟩ to M
20:             c_{v,s} := u_{v,s}
21:         end if
22:     end for
23: end while
```

Figure 3.7: Algorithm for finding SSSPs from each s of source vertices S_k.

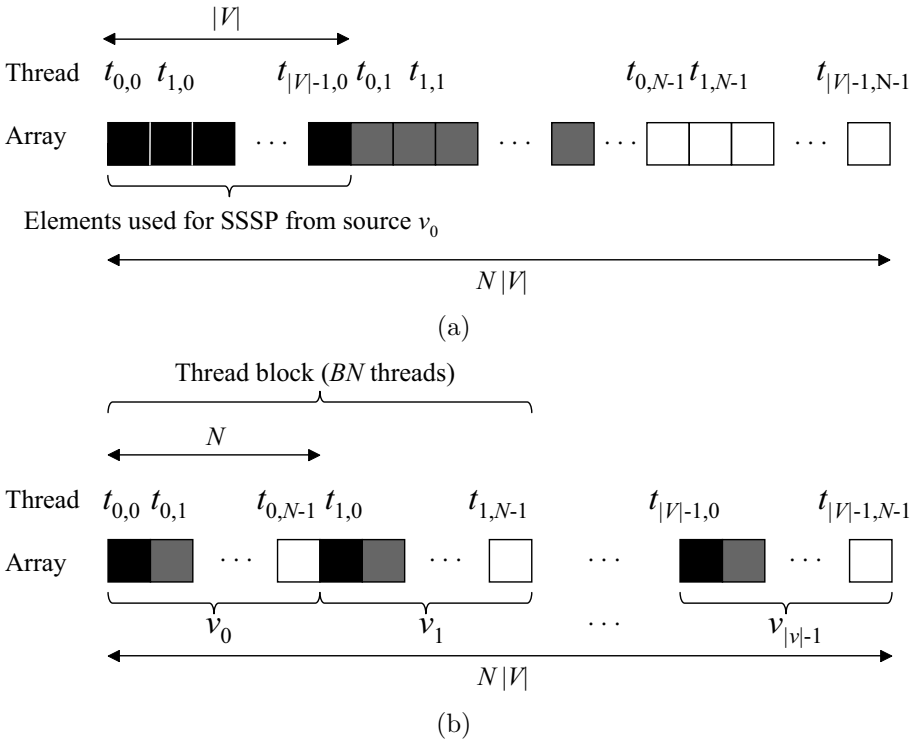


Figure 3.8: Array interleaving for coalesced memory accesses. (a) A straightforward layout for Ma, Ca, and Ua stores all data for every SSSP problems into contiguous sequences. (b) The same vertices but for different problems are stored in a contiguous address space. $B = 2$, in this case.

arrays have $N|V|$ elements as shown in Figure 3.8(b). The reason why we need such larger arrays is that the GPU does not support dynamic memory allocation though each SSSP problem can have a different number of unoptimized vertices at each iteration. Therefore, we simply use N times more arrays to provide dedicated arrays to each of N problems.

This decision might prevent us from using small shared memory for arrays Ca, Ua, and Ma. However, it is not a critical problem because each element in these arrays is accessed only by its responsible thread. Instead, as shown in Figure 3.8(b), it is important to interleave array Ma to allow threads $t_{v,0}, t_{v,1}, \ldots, t_{v,N-1}$ in the same TB to access array elements in a coalesced manner. Similarly, we arrange arrays Ca and Ua into the same structure to realize coalesced accesses in the checking kernel. This also contributes to simplify addressing for data accesses. However, it is not easy to realize coalesced accesses in the scattering kernel because every thread updates different elements of Ua.

Figure 3.9 shows a pseudo code of our scattering kernel. As we mentioned before, we use shared memory for vertex set V, edge set E, and edge weights assigned by w: arrays $from$, to, es, and ws at lines 6 and 12. In addition, we also use shared variable ms to perform reductions of modification set M at lines 7–10. That is, set M is shared among N threads, which are responsible for the same vertex v but for different problems. This means that all of such N threads must be engaged in data duplication if any of them has not yet finished the minimization. This cooperative strategy is essential to increase the number of coalesced accesses on the GPU. For memory-intensive applications, we think that CUDA cores must be used for parallelization of memory accesses, namely coalesced accesses, rather than that of computation. Note that this shared space is used only for this purpose, so that we write the new set M directly to device memory at line 22.

Similar to Harish's algorithm, our method uses a flag to check the convergence of cost computation. Since this flag has to be shared between all threads, we store the flag in global memory. However, we initialize per-TB flags on shared memory at the beginning of the checking kernel. This duplication minimizes the overhead of serialization, which can occur when many threads set the flag at the same time. After checking the convergence, one of threads in each TB writes the flag back to global memory.

3.4.2 Task Size Determination

The proposed kernel requires arrays of size $(3N + 1)|V| + 2|E|$ in device memory while the previous kernel requires those of $4|V| + 2|E|$ size. Therefore, our kernel cannot deal with larger graphs as compared with the previous kernel though it has an advantage over the FW algorithm. Thus, it is better to minimize N to compute APSPs for larger graphs. In contrast, we should maximize N to receive the timing

```
N_SSSPs_Scattering_Kernel(Va, Ea, Wa, Ma, Ca, Ua, N)
 1: v := threadID div N   /* vertex ID */
 2: s := threadID mod N /* source (problem) ID */
 3: /* vertex ID in arrays Va, Ma and Ca */
 4: vg := blockID *B + v
 5: /* Arrays in shared memory */
 6: __shared__ ms[B]
 7: ms[v] := false
 8: if Ma[vg, s] = true then
 9:     ms[v] := true
10: end if
11: if ms[v] = true then
12:     __shared__ from[N], to[N], es[B, N], ws[B, N]
13:     /* Copy data to shared memory */
14:     from[v] := Va[vg]
15:     to[v] := Va[vg + 1]
16:     neighbors := to[v] − from[v]
17:     if s < neighbors then
18:         es[v, s] := Ea[from[v] + s]
19:         ws[v, s] := Wa[from[v] + s]
20:     end if
21:     if Ma[vg, s] = true then
22:         Ma[vg, s] := false
23:         for i := 0 to neighbors − 1 do begin
24:             n := es[v, i]
25:             Ua[n, s] := min(Ua[n, s], Ca[vg, s] + ws[v, i])
26:         end for
27:     end if
28: end if
```

Figure 3.9: Pseudo code of proposed scattering kernel. This kernel solves N SSSP problems in parallel.

benefit of shared memory. Therefore, selection of N is an important issue in our algorithm.

There are two requirements that should be considered when determining N. Firstly, N must be smaller than the maximum size of a TB to share data between threads in the same TB. Since the maximum size is currently 512 threads [6], this requirement will be satisfied when $N \leq 512$. Secondly, N must be a multiple of warp size (32) to achieve coalesced accesses and to reduce divergent warps. The second requirement guarantees coalesced accesses to Ma because every warp will contain threads that process the same vertex v and access data in a contiguous address. Furthermore, such threads have the same number of iterations at line 23 in Figure 3.9 because they have to update the costs of the same neighbors. This is useful to avoid divergent warps at line 23. However, we cannot eliminate all divergent warps in the kernel because the branch instructions at lines 17 and 21 depend on each thread.

According to the design considerations mentioned above, we currently use $N = 32$ in our kernel. This configuration allows us to eliminate synchronization from the scattering kernel, because N is equivalent to the warp size. That is, all threads that must be synchronized each other belong to the same warp, where SIMD instructions guarantee implicit synchronization between threads. Otherwise, threads in the same TB have to synchronize each other after loading data in shared memory. Finally, we experimentally determine to use $B = 4$. Thus, TBs in our kernel have 128 threads.

3.4.3 Path Recording

Our path recording framework is based on a backtracing strategy that specifies the paths after the cost computation presented in Section 3.4.1. That is, this strategy computes the shortest path from the destination vertex $d \in V$ to the source vertex $s \in V$. To realize such a backtracing procedure, our method records which vertex has determined the final costs after the cost minimization. Suppose that we compute the shortest path from the source vertex s to the destination vertex n_1 in Figure 3.2(b). Our algorithm starts from vertex n_1 and finds that the neighboring vertex v determines the cost of vertex n_1. In other words, the shortest path consists of vertex v, which is the parent of vertex n_1. The algorithm then moves to parent vertex v and iterates this backtracing procedure until reaching the source vertex s. This backtracing procedure can be processed in $O(|V|)$ time for each path, so that we compute this procedure on the CPU.

In order to allow the CPU to perform backtracing, our algorithm records all of parents after the cost computation. For each source vertex s, we store parent vertices in array Pa of size $|V|$ such that element $Pa[v, s]$ has the parent of vertex v. This is done by an additional kernel, namely the recording kernel, which is invoked after the convergence of cost minimization. Since our method solves N SSSPs at a

time, the recording kernel also records all parents needed for N SSSPs. A thread in the recording kernel is responsible for a vertex $v \in V$ as same as the scattering kernel. Each thread $t_{v,s}$ records a direct predecessor $u \in V$ as its parent such that u satisfies $c_{v,s} = c_{u,s} + w(u, v)$. To find such a parent, thread $t_{v,s}$ simply checks every direct predecessor $u \in V$ such that $(u, v) \in E$. Since this operation requires predecessors for each vertex, we use inverted graph $G' = (V, E', w')$ to simplify the operation, where E' contains the inverted edge (v, u) of edge $(u, v) \in E$ and w' is a function to assign corresponding edge weights. Therefore, our method further requires $|V| + 2|E|$ space to store the adjacency list of G'. With respect to array Pa, we reuse the memory space for array Ua, so that we do not allocate additional space.

After recording all parents in Pa, the CPU backtraces the shortest path from the destination vertex d to the source vertex s [21]. Firstly, the CPU refers $Pa[d, s]$ to find the parent vertex u of vertex d. The CPU then records u and finds the parent vertex of u from $Pa[u, s]$. In this way, the CPU recursively backtraces the path until reaching the source vertex s such that $Pa[u, s] = s$.

3.5 Experimental Results

We evaluate the performance of our method by comparing it with other two methods: the SSSP-based method [1] and the Dijkstra-based method [14] running on the GPU and the multi-core CPU, respectively. Note that the comparison is done with respect to the cost computation. After this comparison, we also evaluate the overhead of path recording.

The Dijkstra-based method is accelerated using a binary heap. Furthermore, we parallelize the method using all CPU cores. In more detail, we do not parallelize this algorithm but run a sequential program with different sources on each CPU core, because there is no efficient parallelization for Dijkstra's algorithm. Thus, a CPU thread is responsible for solving SSSP problems for a subset of source vertices. CPU threads are managed using OpenMP [31].

We execute GPU-based implementations on a PC with an NVIDIA GeForce 8800 GTS (G92 architecture). This graphics card has 512 MB of device memory and 16 SMs, each having 8 CUDA cores. It also should be mentioned that G92 architecture supports atomic instructions to correctly process the scattering kernel. CUDA-based implementations run on Windows XP with CUDA 2.0 and driver version 178.28. The Dijkstra-based method is executed on an Intel Xeon 5440 2.83 GHz quad-core CPU, 12 MB L2 cache, and 8 GB RAM.

3.5.1 Performance Scalability on Graph Size

We investigate the performance with varying the graph size in terms of the number $|V|$ of vertices and that $|E|$ of edges. The graph data we used in this experiment is random graphs generated by a tool [2]. Using this tool, we generate graphs such that every weight has an integer value within the range $[1, w_{max}]$, where $w_{max} = |V|$.

Figure 3.10 shows the computation time obtained with varying the number $|V|$ of vertices. During measurements, the number $|E|$ of edges is fixed to $|E| = 4|V|$, meaning that a single vertex has four outgoing edges in average. In addition to the three methods mentioned before, we also implement an unshared version of the proposed method that uses device memory instead of shared memory. Due to the capacity of on-board memory, our method fails to solve the problem for $|V| = 1.2$M while Harish's method can deal with $|V| = 10.7$M on our machine.

As compared with the previous SSSP-based method, the proposed method achieves the best speedup of 13 when $|V| = 1$K. In particular, our method runs more efficiently than the previous method when the graph has fewer vertices. The reason for this is that our method has many threads that can be used for hiding the memory latency. For example, it generates 32K threads for 16 SMs when $|V| = 1$K, which is equivalent to 2K threads per SM. In contrast, the previous method has 64 threads per SM. Thus, more threads belonging to different TBs are assigned to every SM in our method. Such multiple assignments are essential to hide the latency with other computation, making the GPU-based methods faster than the CPU-based Dijkstra method, which is faster than the previous method (Figure 3.11).

Since the proposed method solves N SSSP problems at a time, the number of kernel launches is reduced to approximately $1/N$ as compared with the previous method. This implies that we can reduce the synchronization overhead needed at the end of a kernel execution. This reduction effects are observed clearly when $|V|$ is small, where synchronization cost accounts for a relatively large portion of total execution time. Thus, less synchronization allows threads to have shorter waiting time at the kernel completion.

By comparing the proposed two methods, the speedup achieved by shared memory ranges from a factor of 1.2 to that of 1.4. This speedup is achieved by shared memory, which eliminates approximately 16% of the data access between CUDA cores and device memory compared to the proposed method without shared memory. On the other hand, the unshared version of the proposed method is at least 1.9 times faster than the previous method. Thus, the acceleration is mainly achieved by the task parallel scheme rather than shared memory. However, the speedup achieved by the task parallel scheme decreases as $|V|$ increases, because the increase allows the previous method to assign many threads to SMs, as we do in our method. In contrast, the speedup given by shared memory increases with $|V|$. Therefore, we think that shared memory is useful to deal with larger graphs.

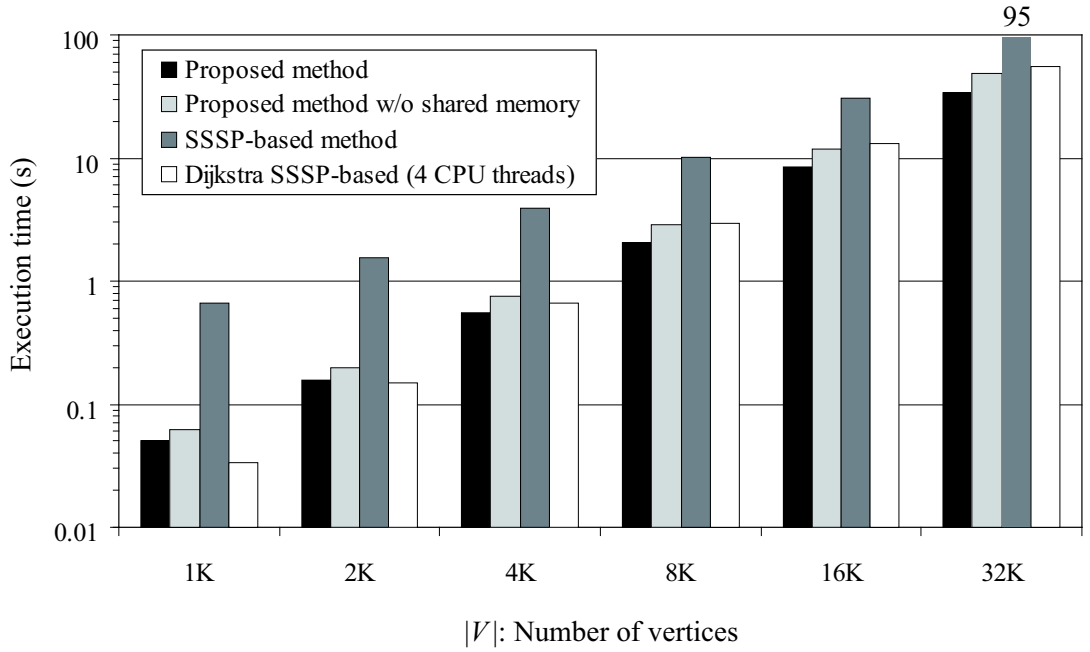


Figure 3.10: Computation time for random graphs with different number $|V|$ of vertices. Results are presented in seconds.

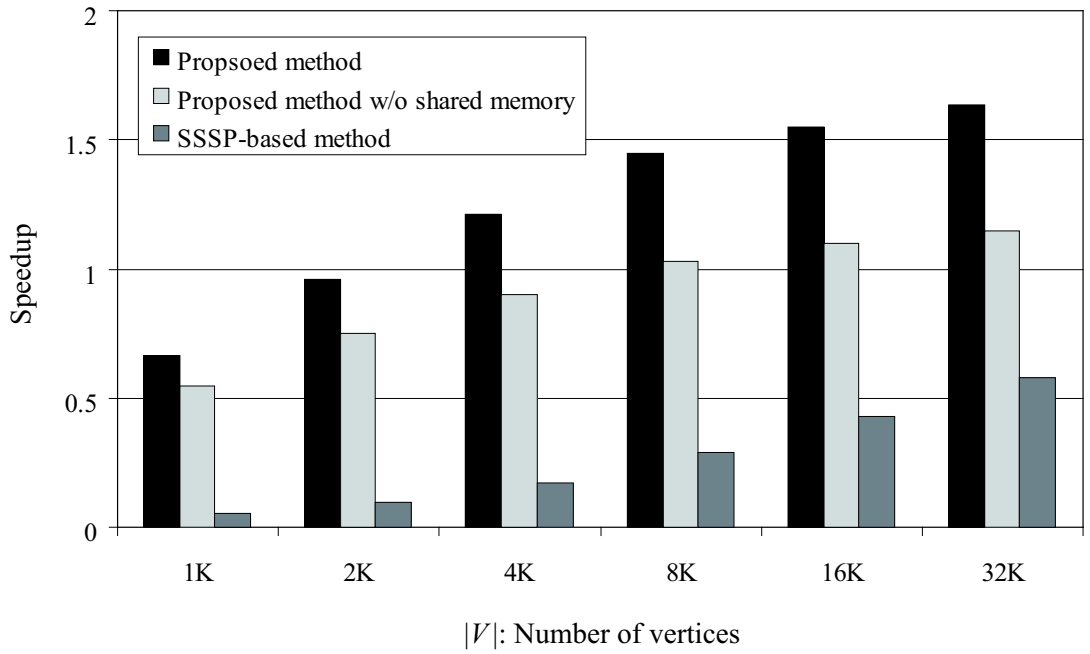


Figure 3.11: Speedup over the SSSP-based implementation running on the CPU.

Figure 3.12 shows the computation time for graphs with a different number $|E|$ of edges. Every graph has the same number of vertices: $|V| = 4$K. These results indicate that all methods increase the execution time with $|E|$. In particular, the shared version of the proposed method shows better acceleration results as $|E|$ increases. Thus, shared memory can effectively reduce the number of data reads from global memory for larger graphs with many edges and vertices. It also should be noted that the proposed method uses $O(BN)$ space in shared memory, which is independent from the graph size $|V|$ and $|E|$. Thus, the graph size is limited by the capacity of device memory rather than that of shared memory.

3.5.2 Performance Stability on Graph Attributes

Figure 3.13 shows the results obtained using graphs with a different maximum weight w_{max}. All graphs have the same numbers of vertices and edges: $|V| = 4$K and $|E| = 16$K. These results show that the execution time increases with w_{max}. However, this increasing behavior is not so sharp as compared with that shown in Figure 3.12, because w_{max} does not directly affect the execution time. The increase of w_{max} means that the graph has edge weights with a larger distribution. In such a case, we need more iterations to minimize the costs, because shorter paths having many edges can overwrite costs that have updated by longer paths having a few edges with larger weights. In addition, a change of a cost of vertex u affects the cost of vertex v, where the temporal shortest paths from source vertex to v pass through u.

We next investigate the performance on different topologies. Table 3.1 shows the computation time for various graph topologies. Every graph data has the same number $|V| = 4677$ of vertices but with different topologies: a random graph [2], a power law graph [32, 33], a ring, and a complete graph. The random graph and the power law graph have $|E| = 16$K edges with $w_{max} = 4096$. The remaining two graphs have the same weight $w_{max} = 1$ for all edges, in order to facilitate the evaluation of performance stability on these graph topologies. The power law graph has many low-degree vertices but also has less high-degree vertices. In more detail, the maximum and average outdegrees in our graph are 834 and 3.5, respectively, and 84% of vertices have a lower degree than 3.

In this table, we can see that all methods significantly vary their performance depending on the graph topology. The GPU-based methods outperform the CPU-based method with respect to the random graph and the complete graph. However, the CPU-based method provides the fastest result for the ring graph and the power-law graph. For the ring graph, this is due to the employed parallel algorithm rather than the GPU implementation, because every vertex in the ring graph has a single outgoing edge, which serializes the scattering phase. Thus, the problem is left on the parallel algorithm rather than the implementation.

For the power law graph, the CPU shows a faster result compared to that for the

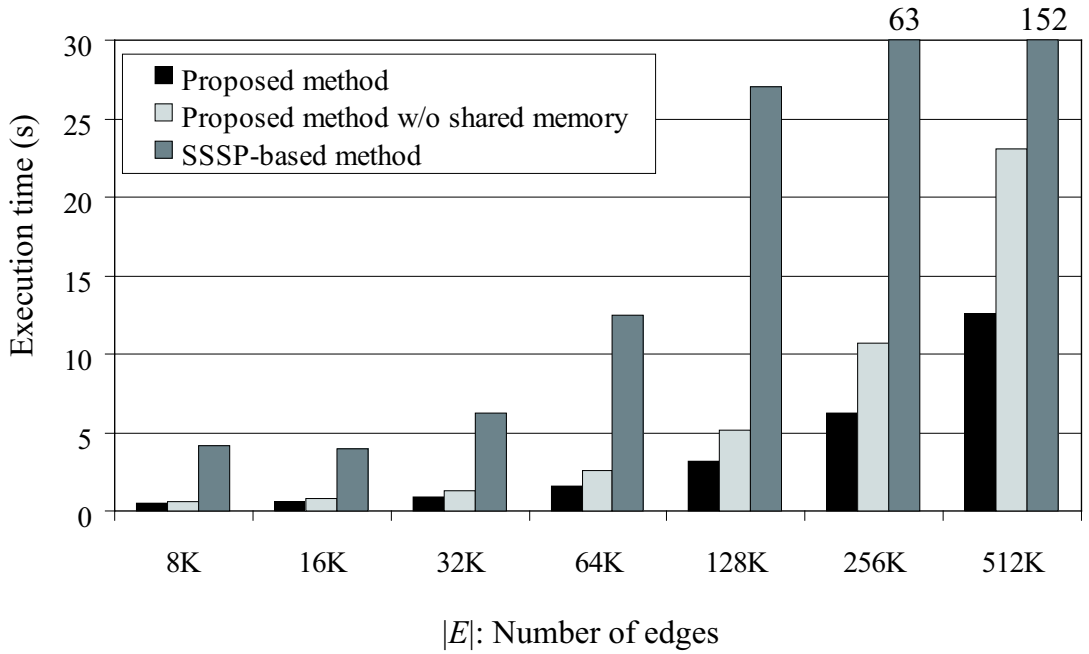


Figure 3.12: Computation time for random graphs with different number $|E|$ of edges. The number $|V|$ of vertices is fixed to $|V| = 4K$.

Table 3.1: Computation time for some graph topologies. Results are presented in seconds.

Method	**Platform**	**Topology**			
		Random	**Power law**	**Ring**	**Complete**
SSSP-based [1]	GPU	4.39	6.37	687	5730
Proposed w/o shared memory		0.884	1.73	53.7	175
Proposed		0.669	0.942	58.2	77.3
Dijkstra SSSP-based (4 threads)	CPU	0.835	0.437	0.0962	158

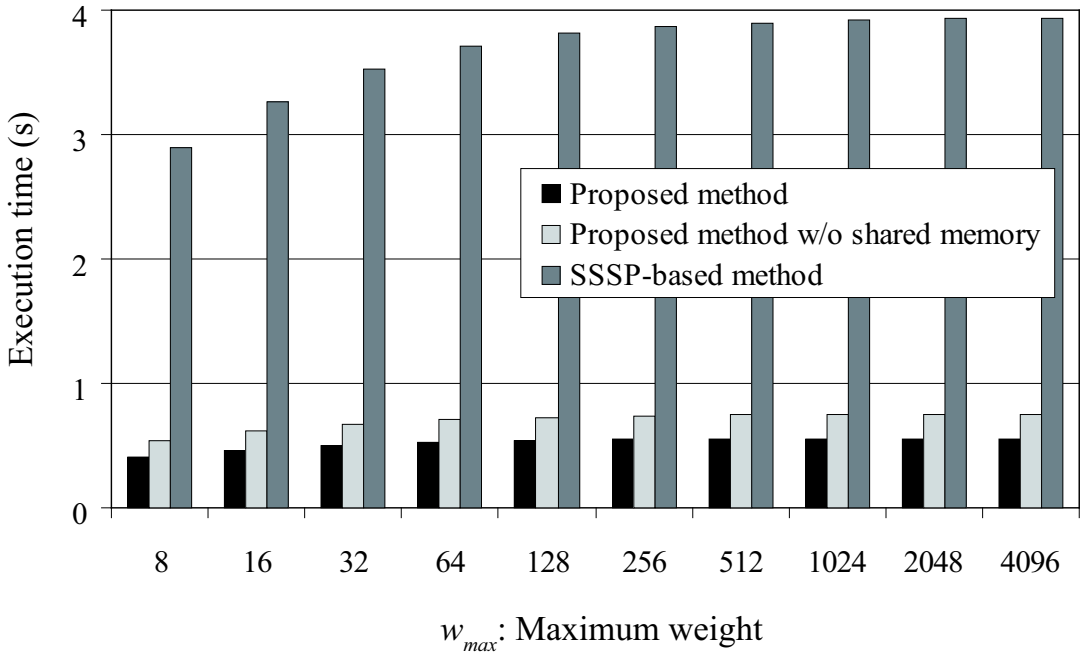


Figure 3.13: Computation time for random graphs with different maximum weight w_{max}. The number $|V|$ of vertices and the number $|E|$ of edges are fixed to $|V| = 4K$ and $|E| = 16K$.

random graph. On the other hand, the GPU-based methods basically decrease their performance as compared with that for the random graph. We think that this is due to the load imbalance in the scattering kernel, because the power law graph has wide outdegrees ranging from 0 to 834. Thus, the most loaded thread has to update 834 neighbors while 84% of threads do this only for at most 3 neighbors. Such a load imbalanced situation will increase the kernel execution time. Actually, the power law graph requires 28% less kernel launches than the random graph. Thus, the GPU-based methods can vary the performance according to the graph degree. The workload will be balanced if every vertex has the same number of outgoing vertices. Otherwise, a sorting mechanism will help us improve the performance.

The previous method takes 156 times longer time to solve the ring graph as compared with the random graph. This increasing time can be explained by the number of kernel launches. The ring graph has a unique topology where every vertex has a single edge. Since the previous kernel updates the neighboring cost of a single vertex, it has to launch the kernel $|V| = 4677$ times to compute an SSSP in this case. In contrast, it requires only 23 launches per SSSP for the random graph. Thus, the ring graph requires 203 times more launches than the random graph, increasing the execution time.

The proposed method also takes longer time for the ring graph but it is approximately 12 times faster than the previous method. We expect a 32-fold speedup over the previous method, because our method simultaneously processes $N = 32$ vertices using 128 CUDA cores. Thus, there is a gap between measured performance and expected performance. This gap can be explained by the branch overhead and the duplication overhead. Our method has more branch instructions in the scattering kernel due to the use of shared memory. Moreover, it requires duplication operations to use shared memory, but this overhead can degrade the performance as we mentioned in Section 3.4.1. The duplication overhead also explains why the unshared version outperforms the shared version when processing the ring graph.

The complete graph shows the worst result among the four topologies used in this experiment. As compared with the random graph, the previous method and our method spend 1310 times and 116 times longer time for the complete graph, respectively. In this graph, every vertex has $|V| - 1$ neighbors, so that $|V| - 1$ repetitions are required to write new costs of neighbors in the scattering kernel. Since each thread sequentially processes these repetitions, the scattering kernel spends relatively longer time. Although these repetitions are common to both of the proposed and previous kernels, our kernel demonstrates a higher tolerance to these repetitions. This tolerance is given by shared memory because each thread refers $|V| - 1$ elements of Ea and Wa, which is duplicated in on-chip shared memory.

3.5.3 Overhead of Path Recording

Finally, we analyze the overhead of path recording. Figure 3.14 shows the ratio of the recording kernel to the remaining two kernels in terms of execution time for random graphs. That is, the execution time in Figure 3.10 will be increased by the ratio in Figure 3.14 if APSPs are recorded after the cost computation. According to Figure 3.14, the overhead of path recording ranges from 3.0% to 7.7%. This overhead is mainly due to the recording kernel. In addition to this additional kernel execution, we also need to send array Pa back to the main memory. Note that we do not include the execution time spent for the backtracing in this overhead, because the number of necessary paths varies with applications of the APSP problem. The CPU takes negligibly short time for backtracing a path compared to the overhead of path recording for these random graphs.

3.6 Conclusion

In this chapter, we have proposed a fast algorithm for finding APSPs using the CUDA-compatible GPU. The proposed algorithm is based on Harish's SSSP-based method and increases the performance by on-chip shared memory. We exploit the coarse-grained task parallelism in addition to the fine-grained data parallelism exploited by the previous method. This combined parallelism makes it possible to share graph data between processing elements in the GPU, saving the bandwidth between off-chip memory and processing elements. It also allows us to run more threads with less kernel launches, leading to an efficient method for highly multi-threaded architecture of the GPU. The method is also capable of recording shortest paths as well as their costs.

The experimental results show that the proposed method is 2.8–13 times faster than the previous SSSP-based method. As compared with the previous method, the task parallel scheme demonstrates higher performance for smaller graphs. However, this advantage becomes smaller when dealing with larger graphs with more vertices. In contrast, shared memory increases its effects for larger graphs with more vertices and edges. With respect to the graph topology, we show that the ring graph serializes both the previous and proposed methods. We also demonstrate that both methods vary their performance according to the graph degree. The overhead of path recording is at most 7.7% for random graphs.

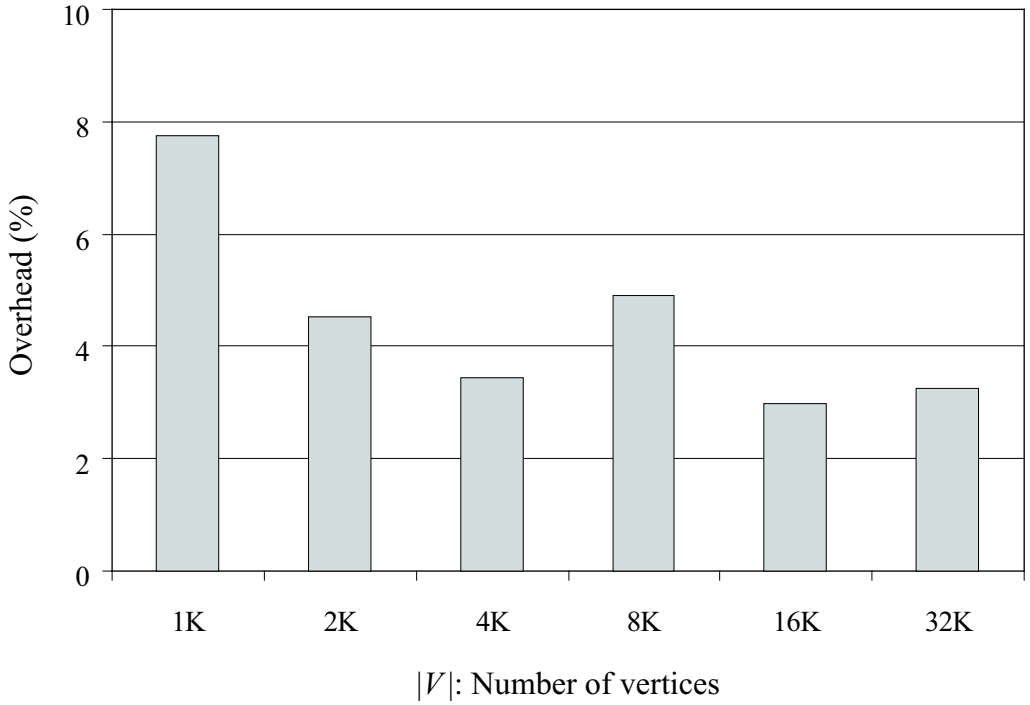


Figure 3.14: Overhead of path recording for random graphs with a different number $|V|$ of vertices. Overhead explains the increased time due to the recording kernel called after cost computation.

Chapter 4

Accelerating Floyd–Warshall Algorithm using the GPU

4.1 Introduction

The Floyd-Warshall (FW) [12, 13] algorithm is a solution to find all-pairs shortest paths (APSPs) for a directed weighted graph. This algorithm uses an adjacency matrix M that represents a given graph G as a square matrix of size n, where n is the number of vertices in the graph G. It calculates the distance matrix D that is a square matrix of order n, each element of which is the cost (distance) between two vertices. This algorithm is a memory bound operation, because it requires $2n$ arithmetic calculations and $4n$ memory accesses per element of M. However, the algorithm is easily to be parallelized for single instruction multiple data (SIMD) [29] processors as well as data partitioning. These characteristics have encouraged researchers to speed up this algorithm using accelerators including graphics processing units (GPUs) [5].

Katz and Kider [15] presented the iterative blocked FW (BFW) algorithm on the GPU using the compute unified device architecture (CUDA) [6], which divides M into equally sized square tiles and iterates the calculations of each tile. This method caches these tiles into on-chip shared memory to reduce global memory access, because the division improves the locality of memory accesses. However, the computational efficiency of their method remains at a low level of 6% compared to the peak computational performance of the GPU, which implies that this implementation can be improved for higher performance.

On the other hand, Buluç et al. [23] proposed the recursive blocking method that recursively divides M into submatrices. This method applies a fast matrix-multiply (MM) routine developed by Volkov et al. [24] to each submatrix computation. The recursive method is at least 5 times faster than the iterative method, achieving

approximately 66% of the peak computational performance. The MM routine uses registers rather than shared memory, because an operation with two operands on the shared memory decreases the instruction throughput of the GPU [23]. The instruction set architecture (ISA) of GT200 GPUs allows only one operand on the shared memory. Therefore, the GPU moves a one of shared memory operands to a register before the calculating an operation having two or more shared memory operands. This MM routine might be also helpful to improve the performance of the iterative BFW algorithm.

This chapter presents two acceleration methods for iterative BFW algorithm using the CUDA-compatible GPU: the MM based method and two-level blocking method. The MM based method applies the fast MM routine [24] to the calculation of tiles in the similar way as the recursive method. It also enlarges the calculation areas corresponding to each thread, aiming at reusing data on the shared memory and a part of calculated values. However, larger calculation area decreases the parallelism and degrades the performance. Therefore, this method uses an auto tuning technique to selects the calculation area size at runtime. For auto-tuning techniques for the GPU, there have been some researches such as stencil calculation [34] and fast Fourier transform (FFT) [35]. These methods run the program several times varying the value of the tuning parameters to find the suitable values. In contrast, our technique statically selects the parameters according to the number of vertices and the specifications of the GPU without running the program. Therefore, this method can eliminate the time to tuning the parameters when using different GPUs.

The two-level blocking method allows larger tile size than the existing iterative method [15], which reduces off-chip memory accesses. For this purpose, this method uses two-level blocking of M to decrease the required amount of the shared memory.

The rest of the chapter is organized as follows. Section 4.2 summarizes the FW algorithm. Section 4.3 presents the two proposed methods and Section 4.4 describes the auto tuning technique for the MM based method. Section 4.5 shows experimental results. Finally, Section 4.6 concludes the chapter.

4.2 Floyd-Warshall Algorithm

The FW algorithm finds APSPs in $O(n^3)$ time. This algorithm can handle a directed weighted graph including negatively weighted edges, if there is no cycle having negative cost. The FW algorithm uses a matrix representation of a graph, called adjacency matrix. The adjacency matrix M is a square matrix of size n. An element $M[u, v]$ has the weight of edge (u, v); $u \in V$ and $v \in V$, which is defined as follows.

$$M[u,v] = \begin{cases} 0, & u = v \\ w(u,v), & u \neq v, (u,v) \in E \\ \infty, & u \neq v, (u,v) \notin E \end{cases}$$

Figure 4.1 shows the FW algorithm. This algorithm overwrites M in place. After the computation, M is equal to the distance matrix D. In other words, an element $M[u, v]$ is the cost of shortest path from $u \in V$ to $v \in V$. To store M, this algorithm consumes $O(n^2)$ space on the memory.

The FW algorithm updates all elements of the matrix M and repeats this operation n times. The most outer loop, shown in line 2 in Figure 4.1, has a data dependency and cannot be parallelized. In contrast, the inner nested two loops in lines 3 and 4 can be parallelized because these loops have no dependency.

4.2.1 Iterative Blocked Floyd–Warshall Algorithm

The iterative BFW algorithm was proposed by Venkataraman et al. [22] to improve the cache utilization of the CPU. This algorithm partitions the adjacency matrix M into multiple tiles, the size of which are $t \times t$, and iteratively updates all tiles in a certain order. Figure 4.2 describes the iterative BFW algorithm and Figure 4.3 illustrates the updating order of the tiles. In these figures, $T_{I,J}$ denotes a tile on M and the upper-left tile is $T_{1,1}$. In the following, the black tile $T_{K,K}$ in Figure 4.3 is referred to as pivot tile. The gray tiles on the same block row of the pivot tile ($T_{K,J}, 1 \leq J \leq n/t$) are referred to as pivot row tiles. $T_{I,K}, 1 \leq I \leq n/t$ are also referred to as pivot column tiles. The other white tiles are referred to as non-pivot tiles.

At first, this algorithm updates the upper-left tile $T_{1,1}$ as the first pivot tile. Secondly, it updates all pivot row and column tiles that are illustrated as the gray tiles. After that, it updates non-pivot tiles. This algorithm then repeats this operation with moving the black tile $T_{K,K}$ from the upper left corner to the lower right corner.

The procedure FWI in Figure 4.2 updates each tile. The parameter A of this procedure is a tile to be updated, while B and C are tiles that are referenced to update the tile A. The tile B is a pivot column tile (or the pivot tile) in the same tile row as the updating tile A. Similarly, C is a pivot row tile in the same tile column as A. Therefore, some of these parameters point to the same tile, when this procedure updates the pivot tile or the pivot row and column tiles. For instance, when updating the pivot tile, all of three parameters point to the pivot tile. Consequently, this operation is equivalent to apply the original FW algorithm (Figure 4.1) to the pivot tile. When FWI updates a pivot row or column tile, either the parameter of B or C is equal to A and the other parameter points to the pivot tile.

This reference dependency among tiles restricts the updating order of tiles to obtain a consistent result. Tiles must be updated in the order of the pivot tile, the pivot row and column tiles, and the non-pivot tiles. On the other hand, all of pivot row and column tiles can be updated in parallel, because the updating procedures of these tiles are independent each other. Moreover, the non-pivot tiles can be also

```
1: FloydWarshall(M, n)  // M: adjacency matrix, n: # of vertices
2:   for k := 1 to n
3:     for i := 1 to n
4:       for j := 1 to n
5:         M[i,j] := min(M[i,j],  M[i,k] + M[k,j])
```

Figure 4.1: FW algorithm.

```
 1: IterativeBFW(M, n, t)  // M: adjacency matrix, n: # of vertices
 2:   for K := 1 to n/t       // t: tile size
 3:     FWI(T_{K,K}, T_{K,K}, T_{K,K}, t)    // pivot tile
 4:     for I := 1 to n/t, I != K
 5:       FWI(T_{I,K}, T_{I,K}, T_{K,K}, t)    // pivot-row tiles
 6:     for J := 1 to n/t, J != K
 7:       FWI(T_{K,J}, T_{K,K}, T_{K,J}, t)    // pivot-column tiles
 8:     for I := 1 to n/t, I != K
 9:       for J := 1 to n/t, J != K
10:         FWI(T_{I,J}, T_{I,K}, T_{K,J}, t)  // non-pivot tiles
11: FWI(A, B, C, t)
12:   for k := 1 to t
13:     for i := 1 to t
14:       for j := 1 to t
15:         A[i,j] := min(A[i,j],  B[i,k] + C[k,j])
```

Figure 4.2: Iterative blocked FW algorithm.

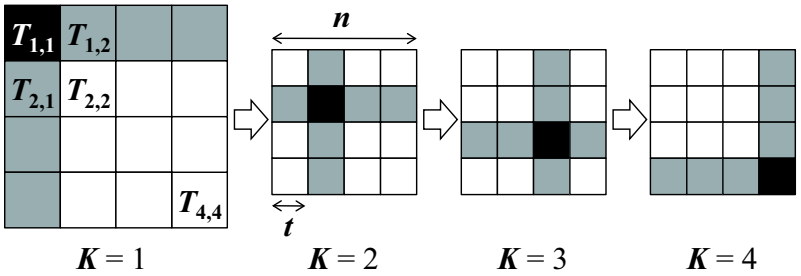


Figure 4.3: Tile updating process of the iterative BFW algorithm ($n/t = 4$). A black tile represents the pivot tile. Gray tiles are the pivot row and pivot column tiles, while white tiles are the non-pivot tiles.

updated in parallel.

FWI refers to each element of B and C t times, and updates each element of A t times. Therefore, this procedure accesses to $O(t^3)$ elements per an execution. Consequently, the iterative BFW algorithm accesses to $O(n^3)$ elements in total. On the other hand, if A, B and C are cached to a fast on-chip memory, the number of off-chip memory accesses decreases by $1/t$ per an execution of FWI. Consequently, FWI accesses $2t^2$, $3t^2$, and $4t^2$ elements on the off-chip memory with the pivot tile, a pivot row or column tile, and a non-pivot tile, respectively. Therefore, the number N_b of data that is accessed by this algorithm to the off-chip memory is given by Eq. 4.1, where $sizeof(element)$ represents the data size of matrix element.

$$N_b = (4n^3/t - 2n^2) \times sizeof(element) \tag{4.1}$$

According to Matsumoto and Sedukhin [36], the iterative BFW algorithm updates $n(n^2 - 2t + 1)$ elements in total. This algorithm uses two arithmetic operations per element; an addition and a minimization of two operands. Therefore, the number N_c of arithmetic operations is given by Eq. 4.2.

$$N_c = 2n(n^2 - 2t + 1) \tag{4.2}$$

4.2.2 Recursive Blocked Floyd-Warshall Algorithm

Figure 4.4 illustrates the recursive BFW algorithm [23]. This algorithm partitions the adjacency matrix M into four submatrices of size $n/2$. It then recursively partitions the upper-left and lower-right submatrices. Figure 4.5 shows this recursive partitioning process. In Figure 4.4, the product "$A \cdot B$" of tile A and B represents an operation that substitutes the addition and multiply of elements in a general matrix multiplication with a minimization and addition, respectively. "$A + B$" also represents an operation that applies a minimization to each element of two matrices.

The CPU recursively calls RecursiveBFW procedure to logically partition the adjacency matrix in lines 3 and 7 in Figure 4.4. The CPU then launches a kernel to update each submatrix using the GPU in lines 4–6 and 8–10 in Figure 4.4. To achieve a consistent result, synchronizations between all GPU threads are required after a launch of kernel in lines 5, 6, 9, and 10, because these kernels have data dependency with their following kernel launches. Consequently, this algorithm requires at least four kernel launches per recursion. The depth of recursion is $\log_2 n/t_R$ because a call of RecursiveBFW procedure divides the matrix into halves. t_R here is the size of submatrix that stops the recursion. When the size of submatrix is smaller than t_R, RecursiveBFW procedures updates the submatrix using the original FW algorithm. Assuming that we apply the FW algorithm to a submatrix using a kernel launch, the CPU launches the kernel n/t_R times. Consequently, the recursive method requires at least $4\sum_{i=0}^{\log_2 n/t_R}(2^i) + n/t_R = 5n/t_R - 4$ kernel launches. Note that Buluç et

al. [23] launch two kernels in lines 4 and 5, although these kernel launches can be merged to a launch. The same is true of kernel launches in lines 8 and 9. Therefore, their method launches kernels $7n/t_R - 6$ times.

On the other hand, the iterative BFW algorithm requires $3n/t$ kernel launches. This algorithm moves the pivot tile n/t times, while each step requires three synchronizations after updating the pivot tile, the pivot row and column tiles, and the non-pivot tiles. Therefore, the recursive BFW algorithm requires $4n/t-6$ or $2n/t-4$ extra kernel launches compared to the iterative BFW algorithm. Therefore, when the computation time is short and the overhead to launch kernels is relatively large, the iterative algorithm can achieve higher performance than the recursive algorithm.

4.3 Iterative Blocked Floyd-Warshall Algorithm on the GPU

The matrix multiplication (MM) based method and the two-level blocking method consist of the same procedures except for the procedure to update non-pivot tiles. In this section, the design that is common between the two methods is firstly described, and different procedures are then described.

4.3.1 Common Design

The adjacency matrix M occupies a large area $O(n^2)$ of memory, which is overwritten during the computation. Therefore, M is stored in the global memory. Our implementation firstly transfers the given graph data from the main memory to the global memory. The CPU then iteratively launches three kernels on the GPU to execute the iterative BFW algorithm. Finally, the calculated result is transferred to the main memory.

The proposed two methods use three kernels, which update the pivot tile, pivot row and column tiles, and non-pivot tiles. These three kernels are denoted by pivot kernel, pivot row and column kernel, and non-pivot kernel, respectively. These kernels employ two-level parallelism of the iterative BFW algorithm. We parallelize the loop on i and j (lines 13 and 14 in Figure 4.2) to simultaneously update multiple elements using threads in a thread block (TB). In addition, all tiles are updated in parallel using multiple TBs in the pivot row and column kernel and the non-pivot kernel (lines 4 and 6). These kernels must be terminated to synchronize all threads. This synchronization is required to obtain a consistent result.

Selecting Tile Size As described earlier, this algorithm reduces more amounts N_b of data accessed to the off-chip memory using large t. However, larger t increases the

```
1: RecursiveBFW(M, n, t_R)
2:   if n ≤ t_R return FW(M) // apply the FW algorithm to M
3:   M_{1,1} := RecursiveBFW(M_{1,1}, n/2, t_R)
4:   M_{1,2} := M_{1,1} · M_{1,2}
5:   M_{2,1} := M_{2,1} · M_{1,1}
6:   M_{2,2} := M_{2,2} + M_{2,1} · M_{1,2}
7:   M_{2,2} := RecursiveBFW(M_{2,2}, n/2, t_R)
8:   M_{2,1} := M_{2,2} · M_{2,1}
9:   M_{1,2} := M_{1,2} · M_{2,2}
10:  M_{1,1} := M_{1,1} + M_{1,2} · M_{2,1}
```

Figure 4.4: Recursive blocked FW algorithm.

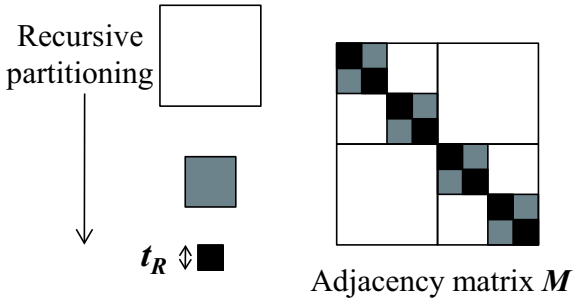


Figure 4.5: Recursive blocking of the matrix M in the recursive BFW algorithm.

use of on-chip memory and results in lower efficiency of parallel computing, because TBs shares the on-chip memory such as registers and the shared memory. Therefore, we increase t so that we can avoid that memory bandwidth becomes a theoretical performance bottleneck. Under this condition, we use t as small as possible to avoid performance degradation.

Figure 4.6 illustrates the required byte/operation ratio N_b/N_c of the iterative BFW algorithm and the byte/operation ratio O_g/B_g of each GPU. N_b represents the amount of data accessed to the off-chip memory. N_c is the number of required calculations. N_b and N_c are calculated from Eq. 4.1 and Eq. 4.2 in Section 4.2.1 with $n = 8192$ and $sizeof(element) = 4$, respectively. O_g and B_g are the values derived from the specifications of the GPU. Table 4.1 summarizes these variables for representing GPU specifications used in this chapter. Note that the MM based method requires slightly different number of memory accesses, due to the design of the MM routine. N'_b/N_c in Figure 4.6 shows the byte/operation ratio of the MM based method, where N'_b is calculated from Eq. 4.3 to be described in Section 4.3.2.

According to Figure 4.6, the byte/operation ratio N_b/N_c of the algorithm is below the byte/operation ratios O_g/B_g of all GPUs, when $t \geq 32$. This means that the bandwidth of global memory does not become a performance bottleneck of this algorithm when $t \geq 32$, even if the GPUs fully utilizes the computing performance. In order to facilitate the implementation that achieves coalesced accesses, we select t in the multiple of 16. Therefore, we select $t = 32$ for the tile size. We assign each thread to update multiple elements in each kernel, because the maximum TB size limits t to less than 23 if each thread updates only one element. The detailed design is different according to the kernel. We describe the design of each kernel in the following.

Pivot Kernel The pivot kernel is launched with a TB to update the pivot tile, where the TB includes th threads. This kernel splits the tile into t/h block rows and updates every h rows at a time (Figure 4.7). Each thread has a two-dimensional thread index $\langle i, j \rangle$ and updates t/h elements $A[i, j + hj'](0 \leq j' < t/h)$ in the tile A of FWI. We synchronize all threads in the TB after updating all elements, because FWI has data dependency in the loop on k.

Each element in the tile A is stored in a register of the thread that is responsible for updating the element. These registers are allocated as an array containing t/h elements in our implementation. Loops in the kernel are unrolled using `#pragma unroll` directive [6] so that the compiler statically calculates addresses of the array elements. This allows the compiler to optimize the code and to store the array elements in registers. Otherwise, an array of temporal variables is stored in the local memory, which has as large latency as the global memory. In our experiments, the compiler successfully stores the array using registers except for the case

Table 4.1: Symbols for representing GPU specifications.

Symbol	Description
M_g	Number of streaming multiprocessors (SMs)
S_g	Number of CUDA cores per SM
H_g	Frequency of CUDA cores
W_g	Number of threads per warp (warp size)
O_g	Peak computational performance given by $O_g = M_g S_g H_g$
B_g	Peak global memory bandwidth

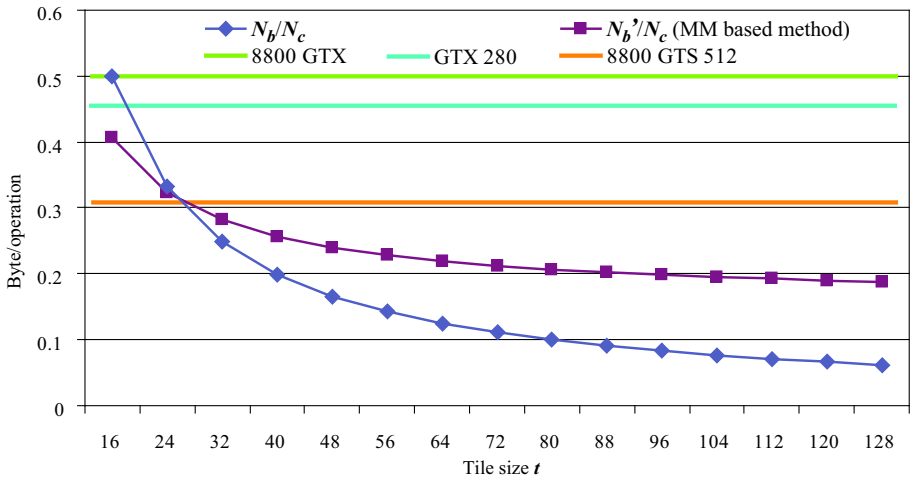


Figure 4.6: Byte/operation ratio N_b/N_c of memory access to computation demanded by the iterative BFW algorithm and that of global memory bandwidth to computational performance of the GPU ($n = 8192$).

with $t = 64, h = 1$ and $t = 64, h = 2$.

All threads commonly refer to the k-th row and k-th column in the pivot tile to update their corresponding elements. Therefore, we duplicate these common data from registers to the shared memory before processing k-th update. This duplication requires the shared memory allocation of $2t$ elements, as well as it increases the number of accesses to the shared memory. However, the shared memory usage is small enough to use larger t, because this kernel only launches a TB.

Pivot Row and Column Kernel The pivot row and column kernel simultaneously updates $2(n/t - 1)$ tiles in the corresponding region. It processes a tile using a TB, while each TB includes th threads and updates the assigned tile in the same way as the pivot kernel. The updating tile A is stored into registers and the k-th row or column of A is duplicated to the shared memory. In addition, the k-th column or row of the pivot tile is stored into the shared memory, because threads also refer to these data in common. Therefore, this kernel allocates $2t$ elements of the shared memory per TB.

Non-pivot Kernel The non-pivot kernel has no data dependency between updating elements. Therefore, some loop optimization techniques can be applied to this kernel, such as loop interchange and loop blocking. The MM based method employs a fast MM routine that uses a loop interchange and loop blocking technique. The two-level blocking method uses a loop blocking technique, in order to reduce the number of elements that are required to store on-chip memory by two-level blocking. We describe two implementations of the non-pivot kernel in the following two sections.

4.3.2 Matrix Multiplication based Iterative BFW

The update of non-pivot tiles (lines 8–10 in Figure 4.2) has a similar access pattern with a matrix multiply-accumulate operation. Concretely, it accesses to the memory in the same pattern as the operation that multiplies the pivot column by the pivot row and then accumulates the result on M. Therefore, this method applies a fast MM routine proposed by Volkov et al. [24] to the non-pivot kernel, replacing the addition and multiplication of elements with minimization and addition, respectively.

We switch the partitioning scheme according to the kernel, because the scheme of the MM routine is different from that of the pivot kernel and the pivot raw and column kernel. This scheme is essential to achieve higher performance. Figure 4.8 illustrates the logical partitioning of matrix M for the non-pivot kernel of this method. This method partitions M into submatrices containing 16×64 ele-

ments. A TB including 64 threads handles each submatrix. A TB corresponding to a submatrix $T'_{I',J'}$ updates the submatrix referring to a region $B'_{I'}$ in the pivot row and another region $C'_{J'}$ in the pivot column. The size of $B'_{I'}$ and $C'_{J'}$ are $16 \times t$ and $t \times 64$, respectively.

A thread updates all the 16 elements in a column of $T'_{I',J'}$ (Figure 4.9). It stores these elements in registers, while the region $B'_{I'}$ is stored in the shared memory. This method uses a register for an element in $C'_{J'}$ registers. Each element of $C'_{J'}$ is referenced by only one thread, because elements in a column of submatrix depend on that element while a column of submatrix updated by a thread. We thus load an element of $C'_{J'}$ from the global memory to registers and execute all computations that depend on the element. Therefore, the updating an element in line 15 in Figure 4.2 requires two elements on registers and an element on the shared memory. Consequently, this method avoids the performance degradation that is caused by an extra instruction to move a data from the shared memory to a register when both $B'_{I'}$ and $C'_{J'}$ are stored in the shared memory.

We modify the MM routine in three points to increase the performance. First, each TB operates on l submatrices. This reduces the number of instructions and memory accesses to load $B'_{I'}$ from the global memory to the shared memory by $1/l$, because updating these submatrices commonly refers to $B'_{I'}$. However, a large l decreases the number of TBs, resulting in performance degradation. The suitable size of l varies with the graph size and the GPU. Therefore, we select l at runtime using an auto-tuning technique described in Section 4.4.

Next, our method duplicates all the elements of $B'_{I'}$ into the shared memory to reduce the number of synchronization in a TB. The original MM routine only stores 16^2 elements of $B'_{I'}$ in the shared memory to reduce the shared memory usage [24]. It appropriately changes the part of $B'_{I'}$ that is stored in the shared memory according to the calculation progress. This requires the synchronization between threads in a TB after changing the stored elements. On the other hand, the size of $B'_{I'}$ is $16t$ with the non-pivot kernel, which is enough small to store on the shared memory. We thus store all the elements of $B'_{I'}$ in the shared memory at first of the kernel, and reduce the synchronization to once in the kernel.

Finally, we omit the calculation of the pivot row and column tiles, because these tiles have been updated when updating non-pivot tiles (lines 8 and 9 in Figure 4.2). To omit the calculation of pivot row tiles, we do not assign TBs to these regions by appropriately adjusting the thread block indices. On the other hand, a TB operates on a multiple tiles in the row, each thread skips to store the calculation result to the global memory, if it calculates in an element in the pivot column tiles. Although it is possible to skip the calculation of pivot column tiles, skipping the store of results is slightly faster in the experiment.

When we use larger tile size t, this method refers to the global memory more than the two-level blocking method, because the MM routine uses a fixed size partitioning.

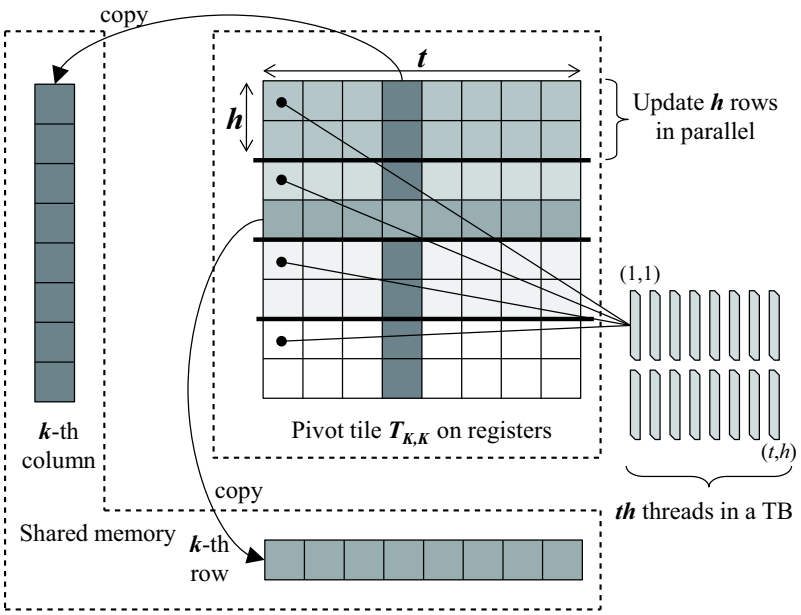


Figure 4.7: Using on-chip memory for updating the pivot tile.

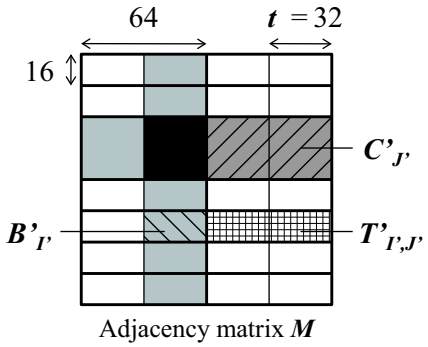


Figure 4.8: Partitioning of the computational region in the matrix multiplication based method.

This routine refers to $n^2t(1/64 + 1/16)$ elements in the pivot row and column tiles on the global memory in total. Therefore, the non-pivot kernel accesses $5n^2t/64 + 2(n-t)^2$ elements on the global memory per launch. Eq. 4.3 gives the total amount of memory accesses of the MM based method.

$$N'_b = \{(5t/64 + 2)n^3/t + 2n^2 - 2nt\} \times sizeof(element) \tag{4.3}$$

For $n = 8192$, the amount of memory accesses of the MM based method is almost equal to that of the two-level blocking method when $t = 26$. If t is larger than 26, this method accesses to the memory more than the two-level blocking method.

4.3.3 Two-level Blocking Iterative BFW

The two-level blocking method partitions each non-pivot tile to reduce the amount of data on the shared memory. For the second level partitioning, we use a two-dimensional blocking method that is commonly used for the MM.

Figure 4.11 depicts the memory usage of the non-pivot kernel, where a TB updates the tile $T_{I,J}$. Each tile is partitioned into sub-tiles of size t'. A TB includes tt' threads and simultaneously operates on t/t' sub-tiles that are arranged along the row. This thread assignment avoids too small TB size that may decrease the performance.

The tile B and C of FWI are partially duplicated to the shared memory, which are required to complete the update of t/t' tiles that are updated in parallel. In detail, we copy t' columns in B and t' rows in C from the global memory to the shared memory. When the update of t/t' tiles are completed, we copy the next t' columns and t' rows in B and C, respectively. Consequently, this method uses $2tt'$ elements of the shared memory. In order to avoid the bank conflicts [6], the t' columns in B are transposed to store in the shared memory. Nevertheless, if the sub-tile size t' is $t' < 16$, the GPU cannot achieve the coalesced memory access when loading these columns. Therefore, we use the texture cache to hide the memory latency. The pivot rows are copied to the texture memory [6] before launching the non-pivot kernel.

We also assign l tiles to a TB, although a TB cannot reuse data on the shared memory because this method stores a part of B. However, some address calculation results are reused between the operations to update a tile.

4.4 Auto-Tuning Technique for Matrix Multiplication based Iterative BFW

In this section, we describe the auto-tuning technique to determine the size of parameter l that adjusts the calculation area of a TB in the MM based method.

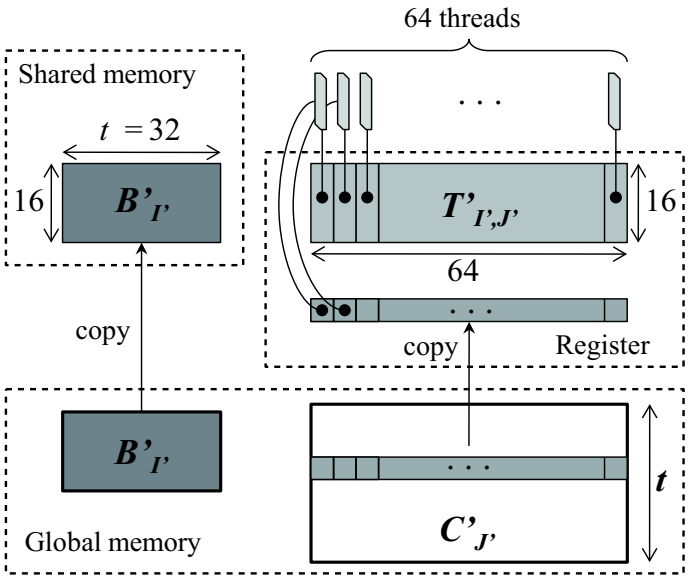


Figure 4.9: On-chip memory usage of the non-pivot kernel with the MM based method.

```
 1: int A[t,t]   // Updating tile T_{I,J}
 2: shared int B[t,t']   // Referring tile T_{I,K}
 3: shared int C[t',t]   // Referring tile T_{K,J}
 4: for k1 := 0 to t/t' − 1
 5:   Load the part of T_{I,K} from texture to shared memory B
 6:   Load the part of T_{K,J} from GM to shared memory C
 7:   for i1 := 1 to t' in parallel
 8:     for j := 1 to t in parallel
 9:       for k2 := 1 to t'
10:         k := k1 × t' + k2
11:         for i2 := 1 to t/t' − 1
12:           i := i2 × t' + i1
13:           A[i,j] := min(A[i,j], B[i,k] + C[k,j])
```

Figure 4.10: Pseudo code for the non-pivot kernel of the two-level blocking method.

As described in Section 4.3.1, the computation theoretically bounds the performance of this method with $t \geq 32$. Therefore, our auto-tuning technique selects l, which minimizes the estimated computation time that is calculated using a performance model of the GPU .

First, we describe the performance model to estimate the computation time. After that, we present the auto-tuning technique to automatically determine appropriate size of the parameter l.

4.4.1 GPU Performance Model for Computation Time Estimation

This model estimates the number of instructions executed on the GPU and the instruction throughput. The estimated computation time is calculated from these two values. It assumes that the computation time of a kernel is equal to the computation time of a SM that processes the largest number of TBs. Table 4.1 lists the symbols that are corresponding to the specification of a GPU. In addition, we use below symbols in this section, the values of which are depends on the kernel.

L : the average number of instructions executed by a thread

U : the number of TBs

X : the number of threads in a TB (TB size)

Y : the maximum number of warps that are executed on a SM at a time

The estimated execution time F is given by Eq. 4.4, where L_{TB} represents the number of executed instructions per TB, U_{SM} is the number of TBs assigned to an SM, and R_{SM} is the effective computation performance of an SM.

$$F = L_{\mathrm{TB}} U_{\mathrm{SM}} / R_{\mathrm{SM}} \tag{4.4}$$

L_{TB}, U_{SM}, and R_{SM} are calculated as follows.

$$L_{\mathrm{TB}} = LX \tag{4.5}$$
$$U_{\mathrm{SM}} = \lceil U/M_g \rceil \tag{4.6}$$
$$R_{\mathrm{SM}} = S_g H_g Z \tag{4.7}$$

L_{TB} is simply calculated by the multiplication of the number L of instructions executed by a thread and the TB size X (Eq. 4.5). We assume that a GPU assigns TBs to SMs in round-robin fashion. Therefore, the largest number U_{SM} of TBs assigned to an SM is given by Ep. 4.6.

$S_g H_g$ in Eq. 4.7 is the peak computation performance of an SM. Z in Eq. 4.7 represents the estimated instruction throughput, which is defined as Eq. 4.8.

$$Z = (Y_{\mathrm{SM}}/Y)/\lceil Y_{\mathrm{SM}}/Y \rceil \tag{4.8}$$

Y_{SM} is the number of warps assigned to an SM, which is given by $Y_{\mathrm{SM}} = U_{\mathrm{SM}} X/W_g$. CUDA-compatible GPUs hide the memory access latency by switching warps. If the number of warps that are ready to execute is less than Y, the GPU might not fully hide the memory access latency and decreases the computation throughput. Therefore, we assume that an SM processes every Y warps at a time and the SM reduces the throughput to $(Y_{\mathrm{SM}} \mod Y)/Y$ when processing the remaining $(Y_{\mathrm{SM}} \mod Y)$ warps. The entire throughput Z is estimated by Eq. 4.8.

A limitation of this model is that this model is not applicable to a program that has a performance bottleneck in memory accesses, because the model estimates the performance using the estimated computation time. In addition, it cannot handle a program, which has a potential performance bottleneck that depends on the parameter values. It also should be noted that some parameters might cause a remarkable change in the order of instruction issues. Recompiling the program might also change the order of instructions. Although, these changes will vary the performance, our model cannot detect such changes because the model estimates the throughput only based on the number of warps that are executed in parallel on the GPU. Our technique focuses on parameters that change the workload per warp at runtime.

4.4.2 Automatic Parameter Selection

Figure 4.12 illustrates the flow of auto-tuning mechanism that automatically selects the size of l. First, this mechanism collects resource consumption of kernels at a compilation time, such as the number of registers used and amount of data allocated on the shared memory. Before executing BFW algorithm, it selects appropriate size of l using the specification of the GPU and the number n at runtime. It estimates the computation time with varying l in a range of $1 \leq l \leq n/64$ to determine the size of l that minimizes the estimated computation time.

Collecting Resource Consumption at Compilation Time At compilation time, we collect the size of L, U, X, and Y. The average number L of instructions per thread is represented by a function of l and the number of instructions in each code block in the non-pivot kernel. We count the instructions in each code block in disassembled results of compiled kernels in a cubin file [6]. A tool, decuda [37], is used to disassemble the cubin file. I depends on l, because a TB updates l submatrices in the MM based method. Therefore, it is represented as $I = \alpha + \beta l$,

where α and β are constants. These constants are calculated from the number of instructions in each code block.

The number U of TBs is depends on n and l, which is given by $U = \lceil n/64/l \rceil - (n/16 - t/16)$ in the MM based method. Note that we assume that n and t are the multiple of 64 and 16 for the simplicity. The number X of threads in a TB is fixed to 64, which is determined by the algorithm.

We use a tool, called CUDA Occupancy calculator [6], to obtain the maximum number Y of warps that are executed at a time on an SM. This tool requires the number X of TB size, the number of registers used by a thread, and the amount of shared memory used by a TB as its inputs. The latter two values are available in a compiler output message.

Collecting Hardware Specification at Runtime Our method gathers the specification of the GPU at runtime that is used to run the compiled execution file. The CUDA runtime library provides `cudaGetDeviceProperties()` API function that returns the specification of GPU at runtime, including the values of M_g, H_g, and W_g. However, the library does not have a function to know the number S_g of CUDA cores in an SM. The GPUs used in the experiments have $S_g = 8$ CUDA cores per SM. Therefore, we assume $S_g = 8$ for our experiments.

4.5 Experiments

In this section, we analyze the performance of two iterative BFW algorithms. In addition, we evaluate the auto-tuning technique for the MM based method.

4.5.1 Environment

Table 4.2 and Table 4.3 summarize the experimental environments. We use three architecture generations of seven GPUs that are compatible with CUDA. Although GeForce 9800 GX2 contains two GPU chips in it, we use only a GPU of them. Table 4.3 shows the specification of a GPU chip of 9800 GX2. The warp size W_g and the number of CUDA cores per SM are $W_g = 32$ and $S_g = 8$, respectively, which are the same between seven GPUs. The unit "Gop/s" of O_g represents that the GPU executes 10^9 computations per second. We develop all programs using Microsoft Visual Studio 2008 and CUDA version 2.3. The video driver is version 190.38.

We use random graphs generated by a tool [2]. Each graph has $m = 4n$ edges, where n is the number of vertices. Edge weights are positive integers that are less than n. We implement all programs to use integer edge weights.

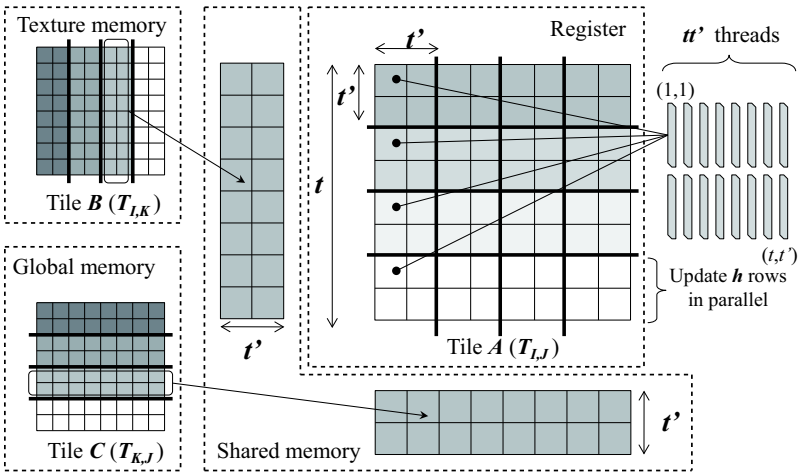


Figure 4.11: On-chip memory usage of the non-pivot kernel with the two-level blocking method.

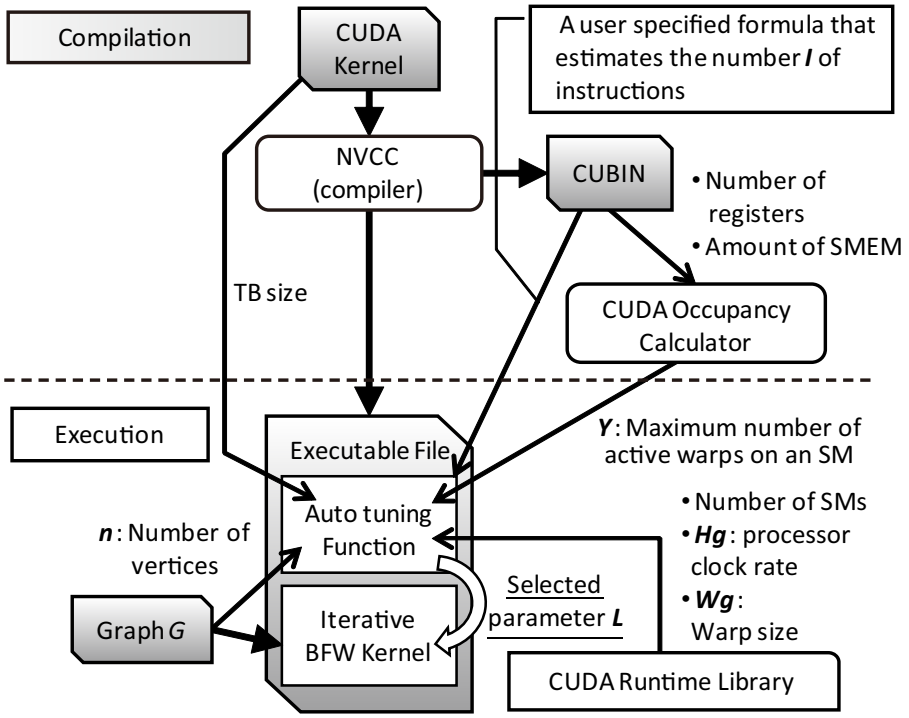


Figure 4.12: Flow of the auto-tuning mechanism for the MM based method.

Our implementation reads an input graph file and stores the graph data in the adjacency list format on the main memory. Next, it transfers the graph data to the device memory of the GPU. The GPU initializes the adjacency matrix M using the transferred data. Therefore, it consumes $O(n^2 + n + 2m)$ of the device memory, which limits the graph size. For instance, GeForce 9800 GX2 and 8800 GTS 512, which have 512 MB of device memory, cannot handle graphs with approximately $n \geq 11K$ vertices if the number of edges is $m = 4n$.

The values of parameter h of the pivot kernel and pivot row and column kernel are manually selected from $h \in \{2^i; 0 \leq i \leq 4, 2^i \leq 512/t\}$ to achieve higher performance. We select $h = 16$, 8, and 4 for the pivot kernel with $t = 16$, 32, and 64, respectively. For the pivot row and column kernel, we use $h = 4$ with all tile size. We use the same value of h for the MM based method and for the two-level blocking method.

Table 4.4 summarizes the parameters and occupancy[6] of the non-pivot kernel using GeForce GTX 280. We determine the parameter t' of the two-level blocking method by a preliminary experiment. The values of l in Table 4.5 shows the auto-tuning results for the MM based method on GeForce GTX 280, while the values of l for the two-level blocking method are manually selected by an experiment.

Eq. 4.9 and Eq. 4.10 are the functions for estimating the average number L of instructions per thread and the number U of TBs in the MM based method, respectively.

$$L = \begin{cases} 72 + 609l, & (t = 16) \\ 84 + 1166l, & (t = 32) \\ 108 + 2844l & (t = 64) \end{cases} \tag{4.9}$$

$$U = \lceil n/64/l \rceil (n/16 - t/16) \tag{4.10}$$

We compile the kernels to collect these functions and other kernel specific information such as X and Y. Then, we embed these functions and values into the CPU code and recompile the program to enable the auto-tuning mechanism for the MM based method.

We also implement the recursive BFW algorithm [23] for evaluating our method. The original design of this algorithm assumes $t_R \leq 22$ to simplify the implementation of the FW algorithm that is used to update a submatrix. We use the pivot kernel described in Section 4.3.1 to implement that with $t_R = 32$ and $t_R = 64$, instead of the original FW implementation.

In this section, we use the number of arithmetic computations of the original FW algorithm to calculate the computational efficiency, in order to compare the iterative and recursive methods. The number of arithmetic computations is $2n(n-1)^2$.

Table 4.2: Experimental environment for the GT200 GPUs.

GPU	**GTX 280**	**FX 5800**	**GTX 260**
Number M_g of SMs	30		24
Frequency H_g (MHz)	1296		1242
Device memory (MB)	1024	4096	896
Bandwidth B_g (GB/s)	141.7	102.4	111.9
O_g (Gop/s)	311.0		238.5
B_g/O_g	0.46	0.33	0.47
Expansion bus	PCI-e 2.0 x16		
CPU	Core i7 940	Xeon X5472	Xeon X5450
Frequency of the CPU	2.93 GHz	3.0 GHz	
Main memory (GB)	12	8	
OS	XP 64bit (SP2)	Vista 64bit (SP2)	XP 64bit (SP2)

Table 4.3: Experimental environment for the G90 and G80 GPUs.

GPU architecture	**G90**		**G80**	
GPU	**9800 GX2**	**8800 GTS 512**	**8800 GTX**	**8800 GTS**
Number M_g of SMs	16			12
Frequency H_g (MHz)	1500	1625	1350	1200
Device memory (MB)	512		768	640
Bandwidth B_g (GB/s)	64.0		86.4	64.0
O_g (Gop/s)	192.0	208.0	172.8	115.2
B_g/O_g	0.33	0.31	0.50	0.56
Expansion bus	PCI-e 2.0 x16		PCI-e x16	
CPU	Core2 Quad Q9550		Xeon X5472	Core2 Quad Q9550
Frequency of the CPU	2.83 GHz		3.00 GHz	2.83 GHz
Main memory (GB)	8			
OS	XP 64bit (SP2)		Vista 64bit (SP2)	XP 64bit (SP2)

Table 4.4: Parameters and resource usage of the non-pivot kernel.

		MM based			**Two-level blocking**		
$\boldsymbol{t}$		**16**	**32**	**64**	**16**	**32**	**64**
t'		—	—	—	4	2	4
TB size X		64	64	64	64	64	256
Shared memory (bytes)		1144	2232	4408	576	576	2112
Registers		39	40	40	18	30	30
Y	GT200	12	12	6	16	16	16
	G80 and G90	6	6	6	12	8	8
Occupancy[6]	GT200	38%	38%	19%	50%	50%	50%
	G80 and G90	25%	25%	25%	50%	33%	33%

4.5.2 Performance Analysis

Table 4.6 shows the computation time using GeForce GTX 280. The computation time here is that the time to execute the FW algorithms on the GPU, which is not include the initialization of the adjacency matrix on the global memory and the transfer of results from the global memory to the main memory. For $n \leq 1024$, the MM based method reduces the computation time by 20–48% and 17–45% compared to the recursive BFW method with $t_R = 32$ and 64, respectively. For $n = 256$–1024, the two-level blocking method with $t = 32$ decreases the time by 5.6–28% and 2.0–23% compared to the recursive method with $t_R = 32$ and 64, respectively. On the other hand, both iterative methods with $t = 32$ show similar performance to that of the recursive method for larger n. The MM based method and the recursive method achieves a high computational efficiency, which reaches 72.7% of peak integer computational performance for $n = 8192$ with $t = 32$ and $t_R = 32$. The computational efficiency is calculated from the number of arithmetic computations of the original FW algorithm ($2n(n-1)^2$) and the computation time. Therefore, the high efficiency suggests that these implementations fully utilize the performance of the GPU.

Comparison of Different Tile Sizes Figure 4.13 is the reduction ratio of computation time comparing $t = 32$ and $t = 16$. The both iterative methods with $t = 32$ decrease the computation time than that of $t = 16$. This result supports estimated performance improvement by a reduction of memory accesses. However, this ratio is greater than the estimated ratio described in Section 4.3.1.

In particular, the two-level blocking method shows higher reduction ratio. For example, the estimated reduction ratio is 9.0% for $n = 8192$, while the measured ratio is 19.7%. In this case, the estimated execution time is 3.88 seconds with $t = 16$. The time is derived from the estimated memory access time N_b/B_g, because the bandwidth limits the performance according to Figure 4.6. On the other hand, the estimated execution time is 3.53 seconds with $t = 32$, because the computation bounds the performance. Therefore, the estimated reduction ratio is 9.0%. The reduction in the number of kernel launches seems to cause the gap between the estimated reduction ratio and the measured ratio. The iterative method launches $3n/t$ kernels, which is inversely proportional to the tile size. Therefore, larger t makes the less overhead of kernel launches in the iterative method.

Comparing $t = 32$ and 64, the case with $t = 32$ achieve higher performance than that with $t = 64$, except for the two-level blocking method for $n = 1024$. Table 4.7 shows the computation time of each kernel to examine the cause. For the MM based method, the increase of computation time of the non-pivot kernel occupies 90% of entire increased time. This kernel decreases the performance, because the rack of on-chip memory degrades the number Y of warps that are executed on an SM at a

time. It allocates approximately 4 KB of the shared memory per TB, which reduces Y to half with $t = 64$ compared that with $t = 32$ (Table 4.4).

In addition, the computation times of the pivot row and column kernel with $t = 64$ are 49–60 milliseconds longer than that with $t = 32$ for both iterative methods. With $t = 64$, the compiler failed to fully unroll loops in this kernel, because it requires too much instructions to unroll the loops. Consequently, the kernel decreases the performance of both methods.

Comparison of Two Iterative BFW Methods In Table 4.6, the MM based method reduces the computation time by 9.0%∼32% with $t = 32$ compared to that of the two-level blocking method. The execution time of non-pivot kernel occupies a large part of the computation time with larger graphs, for instance, the ratio is 98% for $n = 8192$. Therefore, we evaluate the instruction throughputs and the number of instructions of non-pivot kernel to discuss the difference between the two iterative methods.

Table 4.8 shows the instruction throughput for $n = 8192$, measured by CUDA Visual Profiler [6]. With $t = 32$, the throughput is almost the same between the two methods.

On the other hand, the two-level blocking method executes more instructions than that of the MM based method. Figure 4.14 illustrates the total number of instructions executed on the GPU for the non-pivot kernel. To compare the two iterative methods, we set $l = 1$ for this experiment. The number of instructions for updating elements accounts for 81.6% and 74.4% of total instructions that are executed for the MM based method and two-level blocking method ($t' = 2$), respectively. The rest of instructions include address calculations, data movements, and control flow instructions, which does not contribute to the effective computational performance. In other words, the MM based method is optimized more than the two-level blocking method in the number of instructions.

Breakdown Analysis of the Execution Time Figure 4.15 shows the execution time including the initialization of the adjacency matrix and the transfer of results from the device memory to the main memory, in addition to the computational time. For $n = 8192$, the MM based method takes 4940 milliseconds using GeForce GTX 280. In this case, the computation time (Table 4.6) accounts for 98% of the execution time. The auto-tuning of l takes up to 2% of the execution time, which is not a performance bottleneck in practice. For $n \leq 512$, the computation time is 26–60% of the execution time, which implies that the initialization and data transfers will become a performance bottleneck for such small graphs.

Table 4.5: Parameter l for the non-pivot kernel using GTX 280.

Vertices n	**MM based**			**Two-level blocking**		
	$t = 16$	$t = 32$	$t = 64$	$t = 16$	$t = 32$	$t = 64$
128	1	1	1	1	1	1
256	1	1	1	1	1	1
512	2	2	3	5	1	1
1024	2	2	1	10	5	1
2048	8	8	1	10	1	1
4096	4	4	4	16	3	1
8192	16	16	4	27	6	10
9216	29	29	18	48	10	5
10240	8	8	16	64	14	7
11264	16	16	16	44	21	16

Table 4.6: Computation time for random graphs[2] with a different number n of vertices. Results are presented in milliseconds.

Vertices n	**Iterative**						**Recursive[23]**		
	MM based			**Two-level blocking**			t_R**=16**	t_R**=32**	t_R**=64**
	t**=16**	t**=32**	t**=64**	t**=16**	t**=32**	t**=64**			
128	0.425	0.404	0.638	0.696	0.569	0.793	0.783	0.561	0.533
256	0.866	0.818	1.295	1.555	1.197	1.609	2.028	1.585	1.498
512	3.096	2.555	3.705	4.987	3.405	3.934	5.645	4.738	4.415
1024	16.40	13.75	15.38	20.54	16.24	15.72	19.03	17.21	16.58
2048	95.61	86.70	104.4	122.1	97.63	100.8	95.64	91.97	91.97
4096	669.5	633.8	715.2	877.6	701.0	714.9	639.0	631.5	631.5
8192	5116	4861	5526	6731	5403	5483	4883	4866	4867
9216	7170	6966	7650	9512	7654	7751	—	—	—
10240	9885	9504	10490	13020	10480	10580	—	—	—
11264	13070	12590	13760	17280	13910	14100	—	—	—

Table 4.7: Kernel execution time of the iterative BFW methods with $n = 8192$. Results are shown in milliseconds.

	MM based			**Two-level blocking**		
t	**16**	**32**	**64**	**16**	**32**	**64**
Pivot kernel	10.3	8.71	16.7	18.1	12.5	18.7
Pivot row and column kernel	105	74.4	134	139	85.9	128
Non-pivot kernel	5020	4790	5380	6540	5270	5300

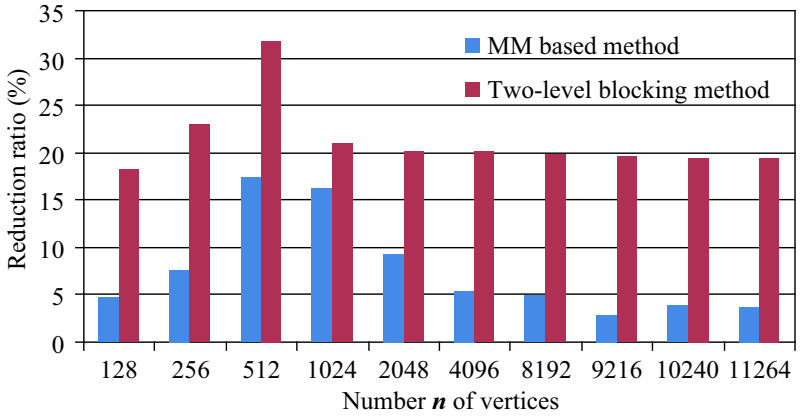


Figure 4.13: Reduction ratio (%) of computation time with $t = 32$ compared to that of with $t = 16$.

Table 4.8: Instruction throughput of the non-pivot kernel using GeForce GTX 280. The number n of vertices is $n = 8192$.

	MM based			**Two-level blocking**		
t	**16**	**32**	**64**	**16**	**32**	**64**
Throughput	0.854	0.851	0.742	0.975	0.879	0.814

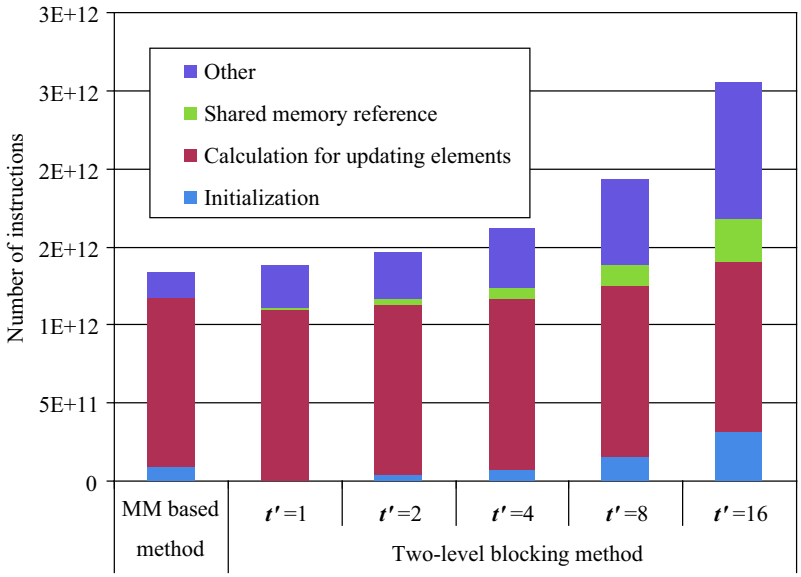


Figure 4.14: Breakdown analysis of the number of instructions in the non-pivot kernel. The number n of vertices is $n = 8192$.

Performance of the MM based Method on Different GPUs Figure 4.16(a) and Figure 4.16(b) illustrate the effective computational performance and efficiency of the MM based method using different GPUs, respectively. In these figures, we vary n by 256 in a range of $512 \leq n \leq 11264$.

GeForce GTX 280 achieves 200 Gop/s or higher performance for $n \geq 2560$, and reaches 227 Gop/s for $n = 11,264$. On the other hand, GeForce GTX 260 shows highest efficiency for $n \geq 1536$ in Figure 4.16(b), achieving 82.1% for $n = 10,752$. In addition, GeForce 8800 GTS 512 achieves the efficiency of 50% for $n \geq 2560$, although it presents the lowest efficiency among the tested seven GPUs. This GPU has the lowest byte/operation ratio between them (Table 4.3), which results in a low efficiency because it fails to fully hide the memory access latency.

In Figure 4.16(a), there are periodic performance drops using GeForce GTX 260 for some graph size, such as $n = 10240$. The partition camping [6] seems to cause these drops, which slows down the global memory accesses.

Comparing the MM based Method and A Task Parallel Algorithm Figure 4.17 shows the computation time of the MM based method ($t = 32$) and a task parallel algorithm [38] that is described in Chapter 3. The computation time is measured with varying the number of vertices using GeForce GTX 280. For $n \leq 8192$, the graphs are random graphs [2] that are used for the performance evaluation shown in Table 4.6. The graphs with $n = 16$K and 32K use the same setup of that of smeller graphs: $m = 4n$ edges and edge weights are positive integers that are less than n.

For $n \leq 2048$, the MM based method is up to 7.8 times faster than the task parallel method, because the MM based method achieves high computational efficiency. On the other hand, the MM based method takes 3.8 times longer time to process $n = 8192$ compared to that of the task parallel method. The MM based method seems to take longer computation time than that of the task parallel method with larger graphs, because the time complexity of the FW algorithm is $O(n^3)$. In addition, the MM based method cannot handle graphs with $n \geq 16$K because of the lack of device memory amount. This method consumes $O(n^2)$ space of device memory, while the task parallel method uses $O(n + m)$ space.

4.5.3 Evaluation of Auto-tuning Technique

We now evaluate the effectiveness of the auto-tuning technique for the MM based method. In this section, we fix the tile size to $t = 32$, which shows higher performance than other tile size in the performance analysis described earlier.

First, we evaluate the execution time of the non-pivot kernel with and without the auto-tuning technique. After that, we explain the effectiveness of the auto-tuning technique against the whole MM based method. We compare the below four methods that tune the parameter l of the MM based method.

Manually tuned (MT) method With this method, we manually select l for each n, which achieves highest performance in a preliminary experiment.

Fixed throughput (FT) method This method is a variation of the proposed method without throughput estimation. We estimate the number L of instructions in the same way as the proposed method. However, we assume that the throughput is constantly 1.0. This method ignores the performance degradation according to the number of warps.

Total instruction based (TI) method This method estimates the computation time by dividing the total number of instructions by the peak computational performance. The number L of instructions per thread is estimated in the same way as the proposed method. The total number of instructions is a product of L and the number UX of threads. This method ignores a load imbalance between SMs in addition to the throughput degradation.

Auto tuning (AT) method This represents the proposed auto-tuning technique.

Figure 4.18 shows the reduction ratio of computation time of the non-pivot kernel compared to that with $l = 1$ using GeForce GTX 280. The non-pivot kernel with $l = 1$ assigns a submatrix to a TB in the same way of the original MM routine. Therefore, we use this configuration as the base case that does not optimize the parameter l.

Figure 4.18 uses preliminarily measured computation time to eliminate measurement error of timings by multiple program executions. We first measure the average computation time per kernel launch for each t, n, and l. In this measurement, we repeatedly launch the kernel in approximately 1 second and calculate the average computation time. The pivot tile is fixed to the top-left corner in the matrix to simplify the measurement. The MT method use this preliminarily results to selects the size of l.

The AT method reduces the computation time by 6.8% for $n = 768$ in Figure 4.18. For $n \geq 5120$, it decreases the time by approximately 4–5%. These results show the effectiveness of the auto-tuning technique. Moreover, it achieves reasonable speedups because the difference between the reduction ratio of MT and AT method is less than 1.5%.

In Figure 4.18, the FT method significantly degrades the performance for $n = 512$ and 3328, while the AT method avoids these performance drops. This result supports that the throughput estimation of the AT method improves the prediction accuracy of the computation time.

Figure 4.19 shows the estimated and measured computation time with varying l for $n = 512$, where the FT method significantly decreases the performance. The FT method selects $l = 8$, because it minimizes the estimated computation time

with the maximum l that divides n/t with no remainder. The number of instructions decreases with the increase of l if l divides n/t with no remainder, while the throughput is fixed to 1. However, a large size of l decreases the parallelism, because the number of TBs decreases with the increase in l. For instance, the number of TBs is 30 for $n = 512$ with $l = 8$, and the number of warps is two per SM on GeForce GTX 280. On the other hand, the number Y of warps that are executed in parallel is 12. Consequently, SMs cannot fully hide the memory access latency for a small graph with a large size of l.

In Figure 4.19, the AT method estimates larger computation times than the actual time, although the suitable parameter size is correct ($l = 2$). Therefore, the AT method requires further improvement in the estimation of the instruction throughput.

The TI method tends to show lower performance compared to the other three methods in Figure 4.18. This method assumes that the computation time is determined by the total number of instructions and the peak computational performance of the GPU. Meanwhile, the total number of instructions reduces with l. Therefore, this method selects the maximum size of l for each graph size.

Finally, Figure 4.20(a) shows the reduction ratio of computation time using the AT method compared to the computation time using a fixed parameter $l = 1$. For $n \geq 2560$, the G90 and GT200 GPUs successfully reduce the computation time with the AT method. In particular, GeForce GTX 280 reduces the time by 5.4% for $n = 4352$. We also evaluate the performance on different GPUs using pre-selected parameters that have optimized for a specific GPU. Figure 4.20(b) illustrates the reduction ratio, comparing the computation time using the AT method for each GPU and that using a manually tuned l for GeForce GTX 280. For $n = 2816$, GeForce GTX 260 reduces the time by 20%. However, the G80 and G90 GPUs tend to increase the computation time. This means that the AT method requires further improvement to select appropriate parameters for each GPU.

4.6 Conclusion

In this chapter, we describe two acceleration methods of the iterative BFW algorithm using CUDA. According to the byte/operation ratio of the GPU and the algorithm, we determine the tile size as $t = 32$. To implements the iterative BFW method with this tile size, a thread updates multiple elements in the adjacency matrix. The first acceleration method applies a fast matrix multiplication routine to the computation, which improved the computational efficiency by reducing shared memory accesses. This method also uses an auto-tuning technique that automatically determines a proper calculation area size of thread block by estimating the computation time based on a performance model. The model estimates the computation time based

on the estimated number of instructions executed on a SM. The second method uses two-level blocking to cache a tile on the shared memory.

As a result, the matrix multiplication based method showed slightly higher performance than that of the two-level blocking method. The matrix multiplication based method reduced the computation time by 17–45% compared to that of an existing recursive BFW method for graphs with 256–1024 vertices. Moreover, the effective performance of this method reached to 70% of peak computational performance. The auto-tuning technique reduced the computation time by up to 5.4% using GeForce GTX 280.

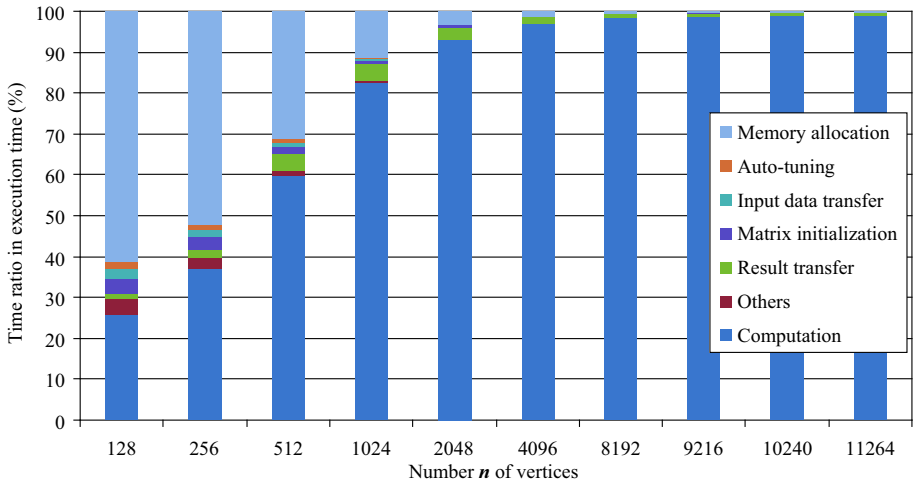
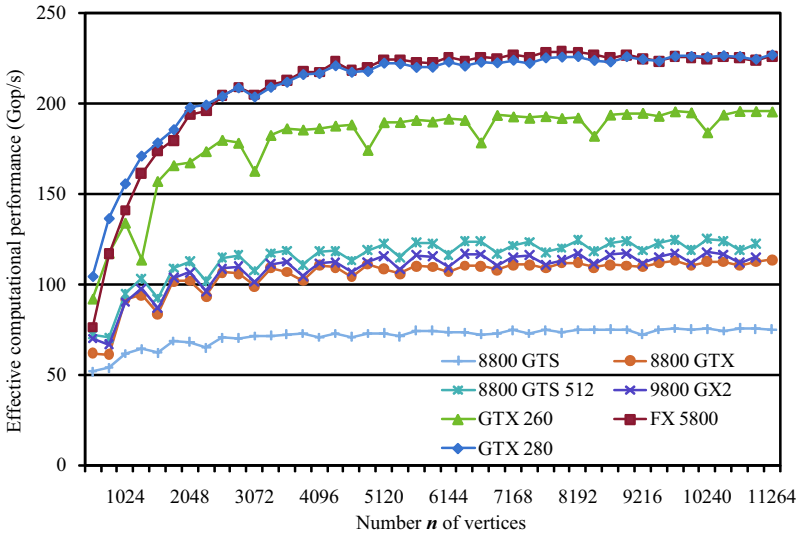
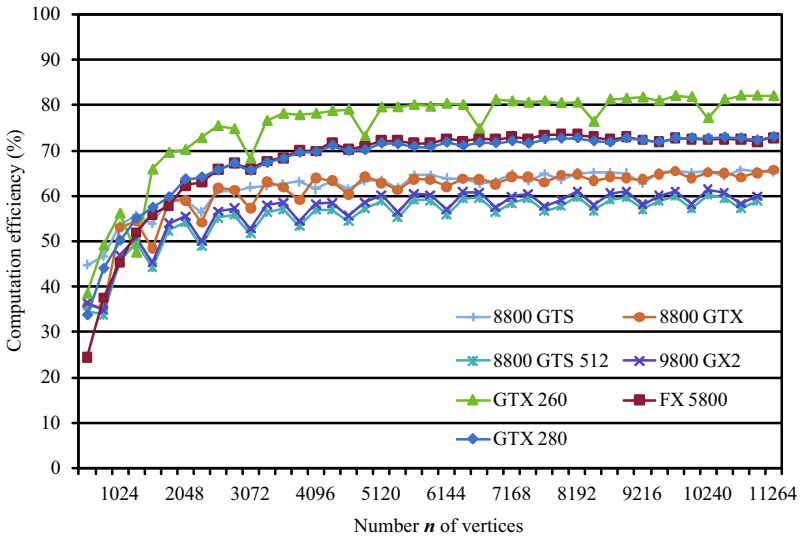


Figure 4.15: Breakdown analysis of the execution time with $t = 32$ using GeForce GTX 280.



(a) Effective computational performance.



(b) Computational efficiency.

Figure 4.16: Effective computational performance and computational efficiency of the MM based method on 7 different GPUs with $t = 32$.

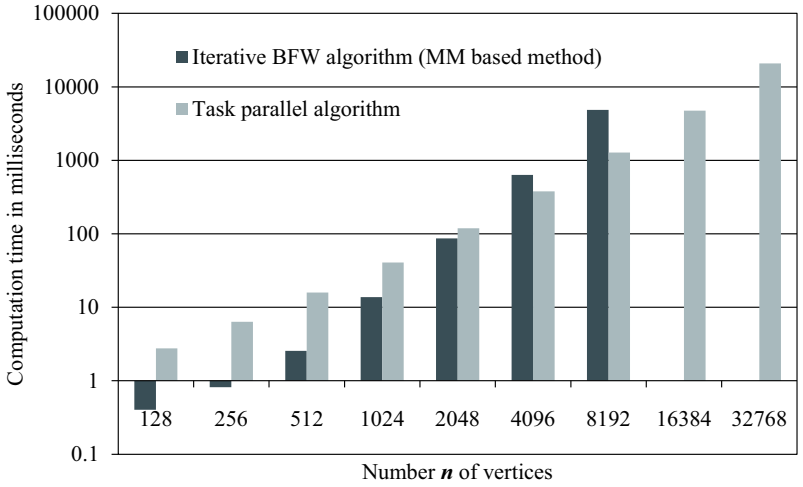


Figure 4.17: Computation time of the MM based method and an task parallel algotihm for random graphs using GeForce GTX 280.

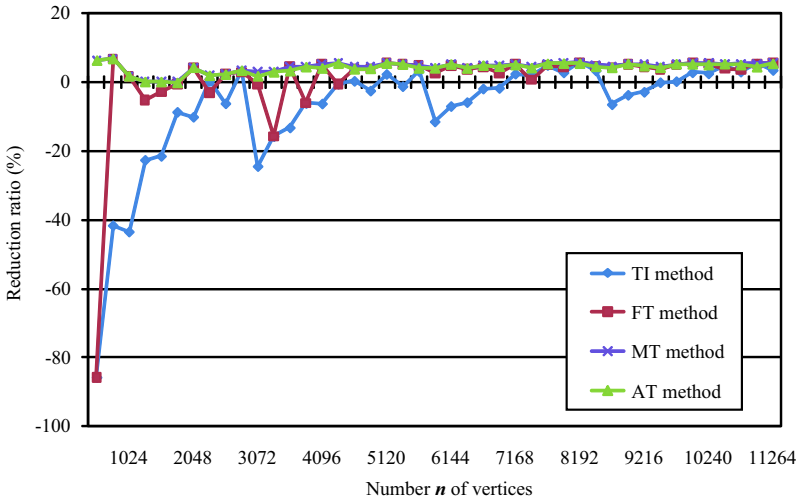


Figure 4.18: Reduction ratio of the computation time of the non-pivot kernel ($t = 32$) using GeForce GTX 280.

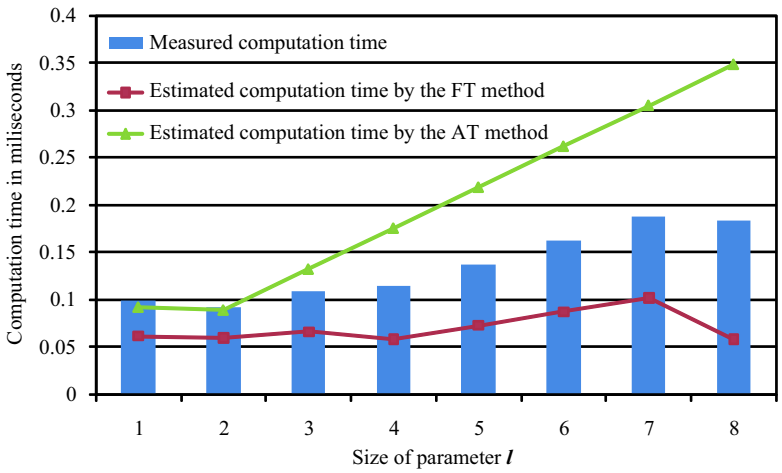
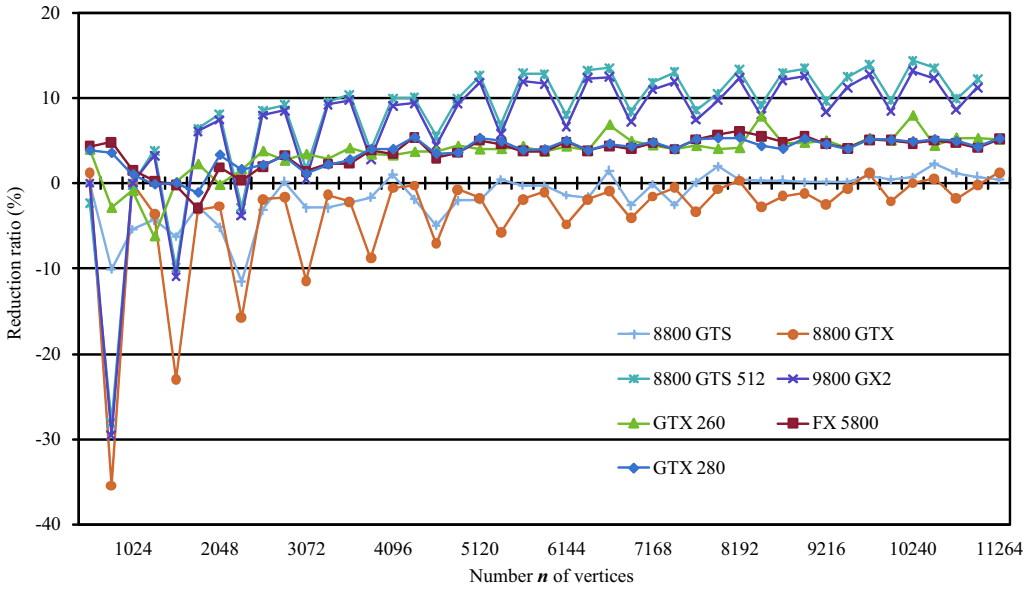
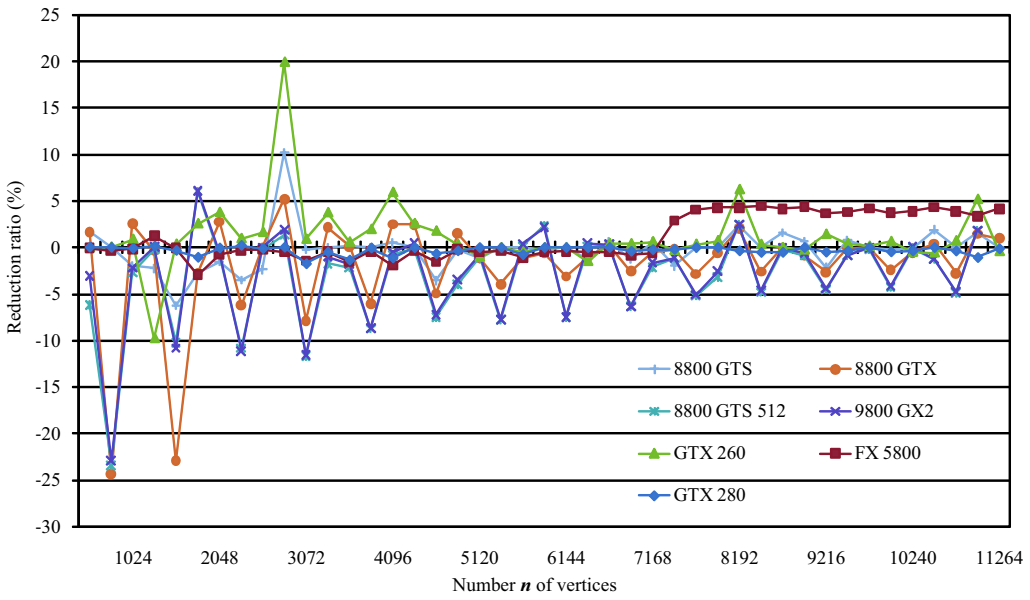


Figure 4.19: Computation time with varying the size of parameter l of the MM based method ($n = 512$).



(a) Reduction ratio using the AT method compared to the computation time using $l = 1$.



(b) Reduction ratio using the AT method compared to the computation time using the parameter l that are manually tuned for GeForce GTX 280.

Figure 4.20: Reduction ratio of the computation time of iterative BFW method with $t = 32$ on different GPUs.

Chapter 5

GPU-based General Biophysical Simulator

5.1 Introduction

There have been many studies and programs targeted at accelerating biophysical simulations via parallel computing. Some researchers speed up their simulations [39, 40, 41, 42] using graphics processing units (GPUs) [6]. However, most of these approaches are aimed at simulating a specific biological model or a specific type of model.

On the other hand, Ackermann et al. [43] proposed a translator-based simulation methodology that can translate a general model into a source code. The general biophysical model here is a set of ordinary differential equations (ODEs) and mathematical functions. It allows us to model diverse physiological functions regardless of the scale of the components, such as cells, tissues, and organs, but is not specialized in a particular function like neurons or cardiac cells. An ODE describes how physiological state, such as ion concentration and membrane potential, evolves over time. Ackermann et al. [43] demonstrated that their program can generate and compute many simulation instances with varying initial values of given ODEs, each of which runs independently using a thread on the GPU in an embarrassingly parallel fashion. Hence, this method does not parallelize each simulation instance itself. To our knowledge, there is no biophysical simulator that automatically explores the parallelism within a simulation instance of a general ODE model to run on a GPU.

Herein, we describe two acceleration methods for simulating general ODE models using a GPU, which automatically parallelize a single simulation instance. Our two methods extend a biophysical simulator called Flint [44] such that the simulation runs on a GPU. Flint is based on *insilico*Sim [45], which adopts an interpreter-based calculation technique to simulate various general ODE models. It translates

each mathematical function and ODE into an internal bytecode and executes a simulation by interpreting them. In the following, mathematical functions and ODEs are referred to as expressions.

The first new method runs the Flint interpreter on a GPU using the compute unified device architecture (CUDA) [6]. This method executes interpreters for multiple bytecodes in parallel on the GPU. Bytecodes have data dependencies between them, because an expression involves some variables, each of which is set by another expression. A level scheduling algorithm [46] is used to automatically parallelize the simulation under the constraints of these data dependencies, which is aiming at running as many threads as possible to efficiently use the GPU. Another challenge is to efficiently process irregular calculations on the GPU, because a model includes a number of different expressions. To calculate these expressions in parallel, this method assigns different bytecodes to threads using conditional branches that cause many divergent branches [6], resulting in performance degradation. To reduce the number of divergent branches, similar bytecodes are assigned to threads in a warp [6]. The required memory access is also reduced by unifying the bytecodes assigned to a warp.

The second new method translates a model into a CUDA code. This method is expected to be faster than the interpreter-based method, which has the overhead involved in interpreting the bytecodes. A naive translation method generates a C++ expression for each expression. This naive method does not seem to be applicable for a larger model, because it produces a large source code, and its compilation time becomes longer than the simulation itself. In addition, it needs as many branches as expressions to compute different expressions using threads on the GPU. Therefore, the CUDA code is generated through the internal bytecodes. Before generating the source code, bytecode unification is performed as in the interpreter-based method. C++ expressions then are generated for a unified bytecode. This technique enables us to decrease the generated code size and the number of branches required for assigning the expressions to the threads.

The rest of this chapter is organized as follows. Section 5.2 gives a brief introduction of related studies. Section 5.3 summarizes the existing implementation of Flint, and Section 5.4 introduces examples of physiological models. Section 5.5 describes the proposed two methods. Section 5.6 describes the experimental results. Section 5.7 presents the conclusions.

5.2 Related Work

There have been many studies aimed at accelerating biophysical simulations using GPUs [47]. Taylor et al. [40] implemented a soft tissue simulation based on the finite element method using a GPU. Sato et al. [39], Lionetti et al. [41], and Garcia

et al. [42] used GPUs to simulate cardiac cells or tissues. Sato et al. [39] simulate a cardiac tissue model that consists of ODEs and partial differential equations. Garcia et al. [42] presented an adaptive step size simulation to calculate the ODEs for cardiac activity on a GPU. Lionetti et al. [41] also described cardiac simulations using GPUs that can handle various mathematical models for cardiac cells, although the connections between the cells are fixed.

Ackermann et al. [43] described an acceleration method for biophysical simulations using a GPU. They generated a CUDA code from a general model written in systems biology markup language (SBML) [48]. Their method increased the simulation speed by 59.3 times compared with those using a CPU. However, they mentioned that the method may have a decreased level of performance for larger models because of the lack of registers on the GPU. On the other hand, our methods are designed to accelerate a simulation instance of a larger model using a GPU.

Heien et al. [45] pointed out that compiling a source code that is directly translated from a biophysical model takes a significantly longer time than its simulation. Therefore, they propose an interpreter-based simulation instead of translating models into C++ codes. We follow this scheme with our interpreter-based method. In addition, our translator-based method reduces the compilation time so that this method can be a practical way to simulate a large model. For this purpose, our translator decreases the size of a generated source code by unifying similar expressions and extracting constants from the source code to an external data file.

They also described a parallel computing method for Flint using message passing interface (MPI) [49]. Comparing with the serial version of Flint, it was 6.0 times faster on an 8-core CPU if node-to-node communication time was not considered, and 3.5 times faster with the communications that bound the performance. Therefore, we suggest shared memory parallel computing of Flint using the GPU.

5.3 Flint: General Biophysicl Simulator

Flint is a simulator developed concurrently with PhysioDesigner[1], an application on which users can build multilevel mathematical models of physiological functions. Models developed on PhysioDesigner are written in physiological hierarchy markup language (PHML) [50, 51], which is an XML-based language designed to describe hierarchical structure and a functional network of models. PHML enables the modeling of a wide variety of large-scale biophysical objects.

PHML is derived from and compatible with *insilico* markup language (ISML) [51]. Flint can parse models written in not only PHML but also ISML and SBML. As a distinguished feature of Flint, it can also handle SBML-PHML hybridized models

[1]Flint and PhysioDesigner are available at `http://physiodesigner.org`.

that can be developed on PhysioDesigner. SBML-PHML hybrid modeling is a notable way to describe multilevel biochemical and physiological systems. This feature can expand the scope of Flint as a simulator in the integrated life science field. In this framework, users can concentrate on modeling without worrying about numerical algorithms including parallelization, because Flint performs numerical calculations of models with parallel computing. In other words, Flint is required to simulate any models built for general purpose with high-performance parallel computing.

A PHML model is a set of modules, where a module typically represents a biophysical unit, for instance, a cell forming an organ and an organ forming a network of multiple organs. A module can encapsulate other modules to represents hierarchical physiological architecture, for instance a tissue consisting of multiple cells. The model also can specify interdependencies between modules including uni- and bi-directed interactions between any two modules, such as between cells and/or between other physiological units.

A module contains expressions, namely, ODEs and mathematical functions[2]. Each ODE represents a time evolution of a physiological state as a function of time t. A mathematical function is an explicit function, the dependent variable of which is referenced by the ODEs and the other functions. PHML allows expressions to refer to variables defined in different modules using the interdependencies of those modules that include the variables.

Flint simulates a time evolution of physiological states by numerically solving an initial value problem for a given set of ODEs of a physiological function, using the Euler or Runge-Kutta methods for numerical calculations. Figure 5.1 illustrates the flow chart of a numerical simulation of a model in Flint. First, this simulator extracts the constants, variables, and expressions from a given model written in PHML as the input. It then constructs a directed acyclic graph $G = (V, E)$, where V is the node set of G and E is the directed edge set of G. The graph G is the data dependency graph of the expressions to schedule the calculation order. A node $f \in V$ of G is an expression. A directed edge $\langle f, g \rangle \in E$ represents the dependency between the two expressions f and g and indicates that g depends on f. In other words, g refers to a variable that is set by f. Therefore, g must be calculated after f to obtain a consistent result.

Flint then partitions G into a set of subgraphs to divide the simulation, if the user requests the parallelization of the simulation using MPI. It applies a graph partitioning algorithm to G, aiming at minimizing the communication between processing elements (PEs). After the partitioning, Flint distributes the partitioned subgraphs to each PE. Each PE schedules the calculation order from the assigned subgraph. The schedule is based on a topological sort of the dependency subgraph.

[2]PHML is capable of handling scalar, vector, and matrix data types, although Flint and the proposed method currently support only scalar data.

Flint ignores dependencies of referring to states that are shown as dashed lines in Figure 5.1 when scheduling, where a state is a variable given by an ODE. With the Euler or Runge-Kutta methods, the states in the next time step are calculated from those of the current time step, while the other expressions depending on the states refer to the state values of the current time step. Therefore, the dependencies on the states can be ignored. On the other hand, an expression that refers to other functions must be calculated after the dependent functions. After the scheduling, Flint generates a bytecode per expression in a model.

After the initialization described above, Flint executes the simulation for a time interval specified in time steps of l/s by repeatedly interpreting each bytecode, where l and s are the user-specified time duration and time step of the simulation, respectively.

Flint supports a stack machine instruction set for its internal bytecode. It generates a bytecode from an expression by traversing its syntax tree in post order. The instruction set has stack operation, arithmetic, comparison, and simple conditional branching instructions. For instance, the `CONSTANT` and `VARIABLE` opcodes push a constant and a variable into the stack, respectively, while the `ASSIGN` opcode pops a computed value from the stack and sets it to a variable. These opcodes are followed by a constant value or variable. Arithmetic instructions include trigonometric, exponential, and logarithm functions. Instructions for addition and multiplication have an uncommon feature in that these operators can have multiple operands for efficiently operating sums and products on many variables. For this feature, Flint writes an opcode followed by the number of its operands into bytecodes.

5.4 Example Physiological Models

In this section, an example of a PHML model is described to show the parallelism in a model simulation. Luo and Rudy [52] introduced a mathematical model for the membrane potential of a ventricular cell. This model receives a stimulus current and simulates the change of the membrane potential using Eq. 5.1. The main expressions of this model are as follows.

$$dV/dt = -(I_{stim} + \sum_{i=1}^{6} I_i)/C, \tag{5.1}$$

$$dI_i/dt = \alpha_i(1 - I_i) - \beta_i I_i \quad (i = 1, \ldots, 6), \tag{5.2}$$

where V is the membrane potential, C is the membrane capacitance and I_{stim} is a stimulus current. The variable I_i $(i = 1, \ldots, 6)$ represents six cell ionic currents. Each ionic current refers to two variables — α_i and β_i — as in Eq. 5.2. The functions that sets α_i and β_i depend only on V; thus, these functions can be computed in parallel.

Therefore, a physiological model contains some independent expressions that can be calculated in parallel as described above. Moreover, all expressions that refer only to constants and states can be independently calculated, because the numerical calculation of the next values of states depends on their current values.

The original Luo-Rudy model consists of 30 functions and 8 ODEs including the expressions presented above. For the PHML model of a single Luo-Rudy cell, we add a function that represents the stimulus current, which feeds pulse currents to the cell. Figure 5.2 shows a coupled Luo-Rudy model that includes five Luo-Rudy cells connected by adder functions and gap junctions. The adder function and gap junction have one and two functions, respectively. In this model, a gap junction refers to the membrane potential V of the two cells, while an adder function refers to a gap junction and the V of a cell. Each cell receives a current from an adder function. Therefore, all the cells can be calculated in parallel after computing the gap junctions and adder functions. The dependency between the expressions varies according to the input model because Flint receives general physiological models. Therefore, this simulator automatically analyzes the dependency at runtime.

It should also be noted that typical large models include networks of cells, such as neurons and cardiac tissue. These models contain multiple modules of the same structure and similar expressions that equal the number of cells, because all the cell models are the same. For instance, the five Luo-Rudy cells in Figure 5.2 have the same type of ODE for each membrane potential V. This characteristic allows the unification of those expressions to improve the performance of the proposed method.

On the other hand, according to the PHML specification, we can also create a model that only consists of a module including thousands of expressions or a heterogeneous model that containing different types of modules. Therefore, our methods explore lower level of data dependencies between the expressions to automatically parallelize the simulation, instead of higher level of data dependencies between modules.

5.5 Accelerating Flint using the GPU

We now describe two proposed methods to accelerate Flint using a GPU: an interpreter-based simulation (IS) and a translator-based simulation (TS).

5.5.1 Interpreter-based Simulation Using the GPU

The IS method automatically schedules an evaluation order for the bytecodes using a level scheduling algorithm, instead of a topological sort. It splits the dependency graph into multiple phases. Each phase consists of bytecodes that are independent of each other. These initializations are processed on the CPU. The GPU then calculates

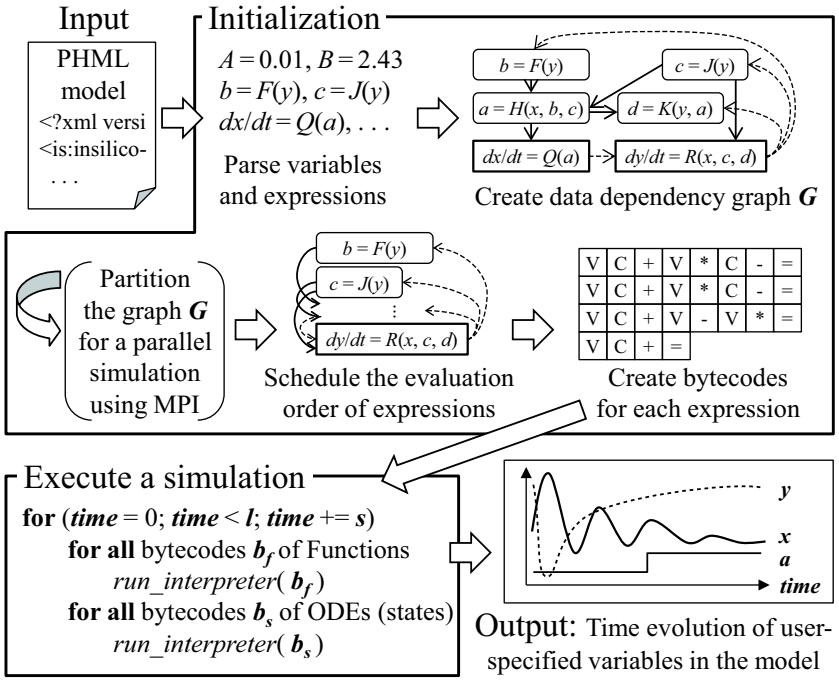


Figure 5.1: Flow chart of Flint.

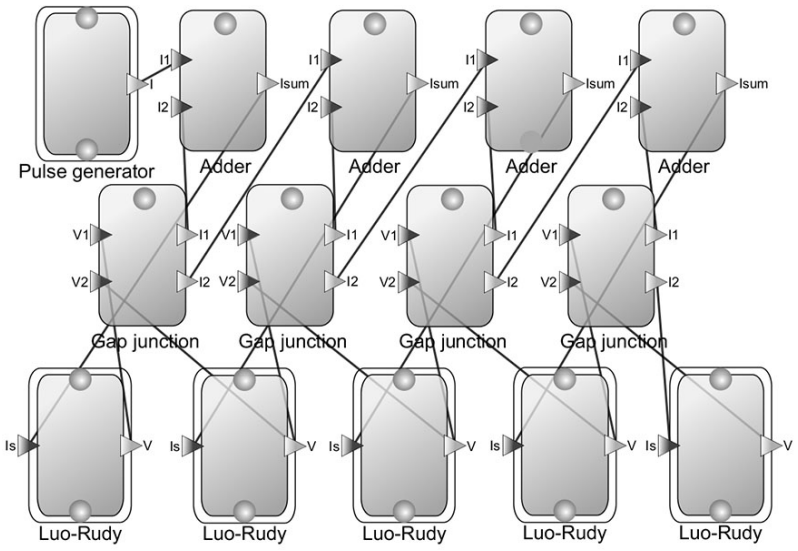


Figure 5.2: Structure of an example of a PHML model. The five rounded rectangles at the bottom represent Luo-Rudy cells.

all the bytecodes in the same phase in parallel. Each thread interprets a bytecode. All the threads are synchronized after calculating each phase by terminating the kernel, because there are dependencies between the bytecodes in the different phases.

A naive implementation has a low computational performance because of a large number of divergent branches and the need to access the global memory numerous times. Moreover, most of the CUDA cores are idle when processing smaller phases that consist of fewer bytecodes. Therefore, we apply six methods after the scheduling to improve the performance:

1. Merge small phases to reduce idle cores
2. Reorder bytecodes according to their similarity
3. Add redundant threads to uniform bytecodes
4. Unify bytecodes to reduce global memory access
5. Remap variable IDs to improve data locality
6. Parallelize the sum of many variables

To implement the IS method, scheduling, bytecode generation, and the execution of simulation were modified, but the remaining parts of Flint remain unchanged. This approach allows for the use of the same input and output facilities of the original Flint program.

Scheduling the Order of Bytecode Evaluation

Figure 5.3 shows the scheduling of the evaluation order of the bytecodes. Initially, a dependency graph G is created by analyzing the dependencies of the expressions in a model (Figure 5.3(a)). The graph is then split into the function subgraph G_F and the state subgraph G_S because we develop different kernels to evaluate the functions and ODEs. The subgraphs G_F and G_S include all the functions and ODEs in G, respectively. The last phase is generated from G_S, in which the GPU simultaneously calculates all the ODEs.

To maintain the dependencies between the functions, a level scheduling algorithm is applied to G_F (Figure 5.3(b)). This algorithm takes all the source nodes $n \in G_F$ that have no incoming edges and creates the first phase consisting of these nodes. All the sources and their outgoing edges are then removed from G_F. The second phase is created from the new sources. These operations are repeated until G_F is empty.

After the level scheduling, the proposed method merges the small function phases into their former phases. First, the proposed method attempts to shift all the

functions in the second and later phases into their former phases. The function e and f are assumed to be in phases j and k ($j < k$), respectively. The function f is shifted from phase k to phase j, if phase j does not have any dependent function of f except for e. Thus, the evaluation of e is scheduled followed by the evaluation of f. This function shifting is applied to all the functions in order, except for the functions in the first phase. It then deletes any empty phases that have no functions.

The two bytecodes that are corresponding to e and f are simply concatenated to sequentially compute these functions using a thread. However, this operation increases the load imbalance between the threads, because it assigns multiple functions to some threads. Therefore, the number of functions processed using a single thread is limited to three based on the result of preliminary experiment.

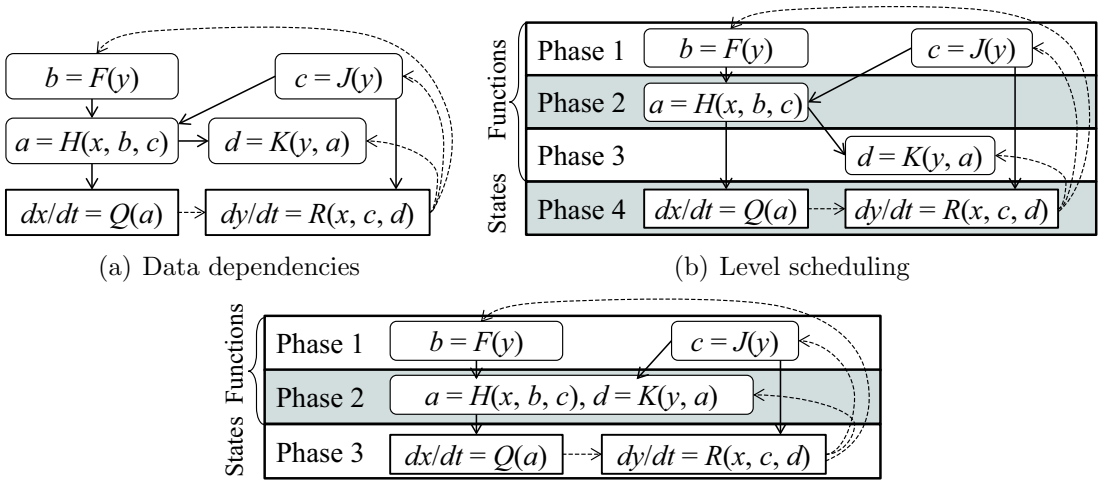
For instance, in Figure 5.3(b), the expression $d = K(y, a)$ in phase 3 refers to the variable a that is set in phase 2. The expression $d = K(y, a)$ is shifted to phase 2 because $K(y, a)$ refers to no variable set in phase 2 other than a. In contrast, $a = H(x, b, c)$ is left in phase 2, because it depends on the two variables b and c that are computed in parallel in phase 1.

Bytecode Optimization

Figure 5.4 describes how the proposed method assigns similar bytecodes to a warp to reduce the number of divergent branches. The bytecodes in each phase are reordered by comparing their opcode sequences. The opcode sequence here is a list of all operation codes in a bytecode. First, the proposed method sorts the bytecodes in each phase in the descending order based on their length. The length of a bytecode here is the number of opcodes in the bytecode. A phase is then divided into sets $B_1, B_2, \ldots$, where the set B_i consists of bytecodes having the same length. Next, the bytecodes are reordered in each set B_i according to their similarity. To achieve this reordering, all the bytecodes in B_i are actually sorted by string sorting, considering their opcode sequences as strings.

Nevertheless, divergent branches are caused if the number of bytecodes in a set B_i is not a multiples of the warp size [6]. To decrease the number of divergent branches, redundant threads are launched, and all threads in the same warp are made to process the bytecodes having the same opcode sequence (Figure 5.5).

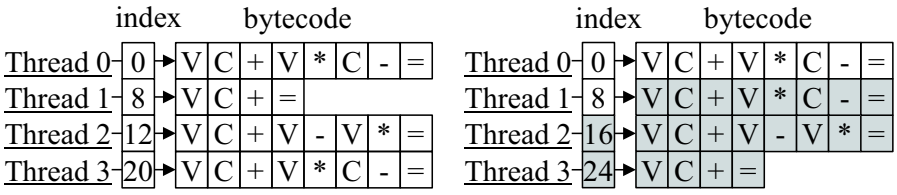
Next, the bytecodes per warp are unified to decrease the number of global memory accesses (Figure 5.6). The threads in a warp interpret the bytecodes that have the same opcode sequence due to the bytecode reordering and redundant threads. However, these bytecodes may have different constants and variables as operands of stack operations. In this case, these different operands are stored into a constant table. These different operands are indirectly referenced through the constant table. We define specific instructions to represent these indirect memory accesses: `CONSTANT_IND`, `VARIABLE_IND` and `ASSIGN_IND` instructions. `CONSTANT_IND` instruc-



(a) Data dependencies (b) Level scheduling

(c) Merge small phases with former phases

Figure 5.3: Scheduling the evaluation order of expressions. Dependencies on states (shown as dash lines) are ignored during the scheduling.



(a) Original order (b) Reordered bytecodes

Figure 5.4: Reordering bytecodes based on their similarity. "V" and "C" represent instruction for pushing a variable and constant, respectively; "index" is an array of indices pointing to the head of each bytecode.

tion is defined to push the constants from the constant table into stacks used by each thread. `VARIABLE_IND` instruction pushes the variable values into stacks referencing the addresses of the variables from the constant table. `ASSIGN_IND` instruction is also defined to store values into different variables. These operands are written in the unified bytecodes followed by an offset of referring values in the constant table.

The order of the variables in the memory is also reordered to increase the data locality by remapping their IDs. Flint assigns a variable ID to each variable and stores the variables in the order of their IDs. However, this order has a marginal relationship with the access sequence of the variables, because Flint initially determines these IDs according to their appearance in the input model. Therefore, in the proposed method, all the variables are numbered on the basis of their order of appearance in the bytecodes after the reordering of the bytecodes is completed. These numbers are then remapped as new variable IDs, and the variables are stored into an array in the order of their new IDs.

Finally, bytecodes containing sums of many variables are parallelized using threads in a warp. Typically, a model for a network of cells has sums, such as the sum of the electric currents of other cells. An expression including the sum of 32 or more variables is assigned to a warp of 32 threads. The warp splits the sum into 32 parts and simultaneously calculates each part using each thread of the warp. After calculating all the parts, the warp sums them up in a parallel reduction manner. This operation is implemented using a warp-synchronous technique [6] to eliminate explicit synchronizations between the threads within a warp.

Implementation of the Interpreter For the GPU

The IS method launches a kernel on the GPU to compute each phase. We run the same number of threads as the number of bytecodes in a phase, and each thread interprets a bytecode. The index of bytecodes, variables, constant table, and bytecodes are stored in the global memory, while the stacks are stored in the local memory [6] of each thread.

Figure 5.7 depicts the pseudo code for the kernel of the interpreter. The kernel `update_funcs` in line 33 and `update_odes_euler` in line 35 calculate the functions and ODEs, respectively. The parameter `index` for both kernels is an array of indices that points to the head of each bytecode, while `codes` is an array of bytecodes and `consts` is the constant table. The i-th thread refers to `index[i]` to get its responsible bytecode. The `input` and `output` arrays are for input and output variable values of each phase, respectively.

When bytecodes are unified per warp, `CONSTANT_IND` and `VARIABLE_IND` operators indirectly refer to a constant and variable that change according to the threads, respectively. When a warp reads these opcodes, all the threads in the warp read the following offset from the bytecode (lines 12–14). Each thread adds its lane ID to

the offset and retrieves the value from the constant table. The lane ID is the index of the thread in its belonging warp.

The CPU launches either `update_funcs` or `update_odes_euler` for each phase in each time step. The size of a thread block (TB) was set with the warp size of 32 to simplify the implementation of interpreting unified bytecodes and parallelized sums. After evaluating all the phases, the CPU transfers the computed results from the global memory to the main memory as necessary. The CPU then advances a time step and repeats the evaluation of each phase.

5.5.2 Translator-based Simulation Using the GPU

The TS method translates bytecodes into a CUDA code for the GPU. This method uses the same level scheduling and bytecode optimizations as those used in the IS method.

However, with this method, all the bytecodes in a phase that have the same opcode sequences are unified into a unified bytecode, while the bytecodes are unified per warp with the IS method. The TS method generates a conditional branch per bytecode to assign different calculation to the threads. To reduce the overhead of these branches, we unify all bytecodes that have the same opcode sequence, which also decreases the generated code size. The redundant threads are inserted so that the number of threads that process the same unified bytecode is multiple of the warp size.

The TS method also uses an auto-tuning technique to determine the TB size. It inserts a code for measuring the kernel execution time of each phase. During the early steps of the simulation, the generated simulation program measures the execution time of each phase every 100 steps, while increasing the TB size as a multiple of the warp size. This program terminates the tuning process for a phase when the execution time of the phase is longer than the previous 100 steps, because we assume that the time is convex downward with respect to the TB size. Therefore, the TB size for the previous 100 steps is selected as the tuned TB size. The program runs the phase in the remaining steps using the selected TB size.

Figure 5.8 illustrates a part of the generated code for simulating a model on the GPU. A kernel is created for each phase and different calculations are assigned to threads by branching them according to their indices, as shown in Figure 5.8 in lines 6, 9, and 11. An expression corresponding to a unified bytecode is processed using as many threads as bytecodes before unifying them to calculate all of them in parallel. For example, 128 threads are assigned to calculate line 10 in parallel, where each thread indirectly refers to different operands of the original bytecodes through the constant table `c1`.

The kernel for the state phase temporally saves the calculated states to an array `nv` of temporal values to maintain the consistency of the states. Therefore, another

kernel `state_update` is added to copy the new state values from `nv` to the variable value array `v` in line 27.

The TS method exports a data file that contains constant tables in addition to the source code including kernels. However, exporting the tables prevents optimizations of the compiler, because the compiler cannot see the constants in the tables — although there is a trade-off between the compilation time and compiler optimization. In experimental evaluations, it was observed that the compiler takes a long time to compile the generated source code without exporting the tables, which makes the TS method impractical. Exporting the constant tables enables a faster compilation by avoiding exhaustive optimizations, such as loop unrolling and constant folding, although the level of improvement depends on the compiler.

The TS method can also generate a source code of a simulation program that run on the CPU. For the CPU, we simply generate a series of C++ expressions corresponding to each bytecode in a phase. The method continuously writes all phases into a source code. A unified bytecode is translated to a loop that repeats the number of bytecodes before unifying them as shown in lines 3–4, 6–7, and 11–15 in Figure 5.9. For the CPU, this method does not use the redundant threads and parallelized sums. Some bytecodes consist of multiple expressions when we merge multiple phases. The TS method translates these bytecodes into multiple C++ expressions within a block like lines 12–15. This method parallelizes the simulation using OpenMP [31] for a multi-core CPU. It embeds OpenMP directives to parallelize the loop for unified bytecodes as in lines 2, 5, and 10.

5.6 Experimental Results

The performance of the proposed method was evaluated on a PC with an Intel Core i7 930 2.8 GHz quad-core CPU, 12GB RAM, and an NVIDIA Tesla C2070 GPU. Simultaneous multithreading (SMT) and Intel Turbo Boost technology were enabled on the CPU, because these functions slightly improve the performance of CPU-based simulations. In this section, CPU-1 and CPU-8 denote simulations using one and eight CPU threads with SMT on the CPU, respectively, although all the methods use a single CPU thread to process initialization such as reading the model, scheduling, and optimizing the bytecodes. The IS method using CPU-1 is corresponding to the existing interpreter-based simulation of Flint using a single CPU thread. The TS method using CPU-8 parallelizes the simulation using OpenMP. The implementation was developed using Visual Studio 2010 and CUDA 4.2 on Windows 7 (64 bit) with a version 301.32 video driver. For the TS method, the generated source codes were compiled using Visual C++ 2010 compiler and nvcc [6] with the `-O2` optimization flag.

Table 5.1 presents the number of expressions in physiological models used for the

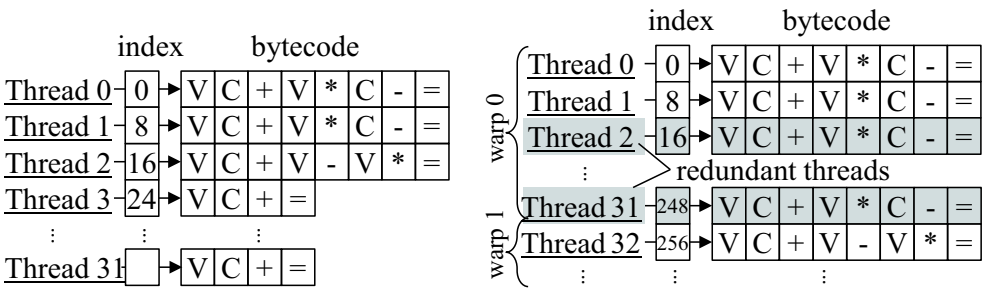
evaluation. The Luo-Rudy model is a PHML model that describes a cardiac cell [52]. The LR-Ring model consists of 80 Luo-Rudy cells in a ring connection [53]. The LR-100 and LR-1000 – LR-5000 models represent 100 and 1,000 – 5,000 cells connected in a line, respectively. The Wang [54] and Rybak [55] models are neuron models having different network connections. The Mix model is a demonstration model for processing heterogeneous physiological functions. It is a simple concatenated model of the Wang, Rybak, and LR-100 models. The proposed method is available for such various physiological models containing thousands of expressions. With the Luo-Rudy and LR-100 – LR-5000 models, we demonstrate that our methods can handle models with various scales. It can also accelerate the simulation of various cell networks containing different dependencies, such as those in the Wang, Rybak, and LR-Ring models. All the simulations were executed in double precision using the Euler method both on the CPU and GPU. In addition, all the simulations were run for 100,000 steps with a time duration $l = 1,000$ milliseconds and a time step $s = 0.01$ milliseconds, except for the experiments designed to analyze the accuracy of the method. The duration and time step were selected to be in a range similar as those values used in the Luo-Rudy [52] and Wang [54] simulations.

5.6.1 Performance Evaluation

Table 5.2 shows the total execution time including the output of the simulation results. The values of all the variables were sampled every 100 steps, while these results were readbacked from the GPU every 1,000 steps. A readback means a data transfer from the global memory on the GPU to the main memory, which has an overhead for synchronization between the CPU and GPU. The results were directly written to a file as binary data. Different parts of the methods were evaluated with respect to the time it took to complete them. The time of the IS method includes the initialization time as well as the execution time of the simulation itself. For the TS method, the code generation and compilation time were evaluated in addition to the initialization time.

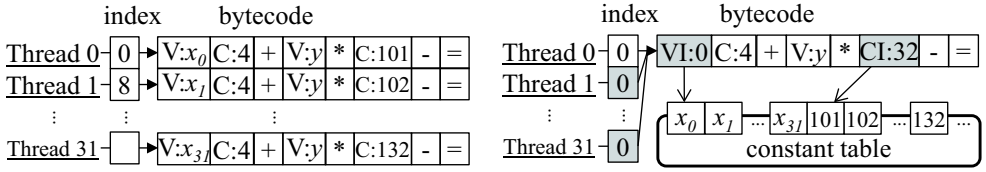
With larger models containing thousands of expressions, the IS method is 1.5–24 times faster than the existing method running on CPU-1. In contrast, the GPU is 12 times slower than the CPU with the Luo-Rudy model because it only has 39 expressions and less parallelism. On the GPU, the TS method is up to 2.4 times faster than the IS method. Note that the TS method on the CPU shows a lower performance with the LR-Ring model because the compilation time increases to 250 seconds (Table 5.8).

Table 5.3 shows the global memory usage of the IS method in the same setup as that of Table 5.2. The LR-1000 model requires 320 MB of the global memory to buffer the results and reduce the overhead of the readbacks. However, the other data only occupies approximately 2 MB. Meanwhile, users generally have an interest



(a) Original thread assignment (b) Add redundant threads

Figure 5.5: Adding redundant threads to avoid divergent branches.



(a) Original (b) Unified bytecodes

Figure 5.6: Unifying bytecodes to reduce memory accesses. "VI" and "CI" are opcodes for the indirect reference of a variable and constant, respectively.

Table 5.1: Number of functions and ODEs in the tested models.

Model	**Luo-Rudy**	**Wang**	**Rybak**	**LR-Ring**	**LR-100**	**Mix**	**LR-1000**
Functions	31	1,200	1,362	2,647	3,298	5,860	32,998
ODEs	8	400	480	640	800	1,680	8,000

```
1  template<class AssignT>
2  __device__ update(index, codes, consts, in, out) {
3    stack[STACK SIZE]; /* use local memory for the stack */
4    *sp = stack - 1;
5    id = blockIdx.x * blockDim.x + threadIdx.x;
6    lane = laneid; /* the thread's lane within the warp */
7    pc = index[id];
8    while((inst = read_int(pc, codes)) != END) {
9      switch(get opcode(inst)) {
10     case CONSTANT: /* push a constant value */
11       push(sp, read_double(pc, codes));
12     case CONSTANT_IND: /* push a constant value */
13       adr = read_int(pc, codes) + lane * sizeof(double);
14       push(sp, consts[adr]);
15     case VARIABLE: /* push a variable value */
16       push(sp, in[read_int(pc, codes)]);
17     case PLUS_NARY: /* add or sum (less than 32 operands) */
18       n = read_int(pc, codes); /* # of operands */
19       sp -= n-1; v = 0;
20       for(i=0; i<n; i++) v += sp[i];
21       push(sp, v);
22     case ASSIGN: /* assign a value to a variable */
23         AssignT::assign(read_int(pc, codes), pop(sp), in, out);
24     ...
25 } } }
26 __global__ double *delta; /* an array of step sizes */
27 struct AssignFunc {
28   __device__ assign(pos, value, in, out) {
29   out[pos] = value; } }
30 struct AssignEuler {
31   __device__ assign(pos, value, in, out) {
32   out[pos] = in[pos] + value * delta[pos]; } }
33 __global__ update_funcs(index, codes, consts, in, out) {
34   update<AssignFunc>(index, codes, consts, in, out); }
35 __global__ update_odes_euler(index, codes, consts, in, out) {
36   update<AssignEuler>(index, codes, consts, in, out); }
```

Figure 5.7: Pseudo code for the kernels of the interpreter.

```
1  __global__ void phase1(double *v, double *nv, int *c1) {
2    // v: variables
3    // nv: temporal variables to calculate states
4    // c1: the constant table for phase 1
5    id=blockIdx.x*blockDim.x+threadIdx.x;
6    if (id<3200) { const int i=id-0;
7      v[c1[i+3200]]=(0.001*(v[c1[i+0]]- (-75)))*
8        para_sum(i/32,99,c1,12296,v); // parallelized sum
9    } else if (id<3328) { const int i=id-3200;
10     v[c1[i+7040]]=((((9*v[c1[i+6400]])*v[c1[i+6528]])* ···
11   } else if (id<3648) { const int i=id-3328;
12     v[c1[i+9408]]=get(c1,i*2+7168)*exp((-(v[c1[i+7808]]+
13 ...
14 } // end phase 1
15 ...
16 int main(int argc, char** argv) {
17   ... // initialization
18   int *c1=NULL; // constant table for phase 1
19   LoadConstants(const_file); // load constant tables
20   ...
21   for(double t=s;t<=l;t+=s) {
22     // s: the simulation time step, l: the time duration
23     ...
24     phase1<<<nb1,nt1>>>(v,nv,c1);
25     phase2<<<nb2,nt2>>>(v,nv,c1);
26     ...
27     state_update<<<nb5,nt5>>>(v,nv,deltas,state_ids);
28     if (is_sampling_step())
29       cudaMemcpy(h_v,v,mem_size,cudaMemcpyDeviceToHost);
30   }
31   ...
32 }
```

Figure 5.8: Pseudo code for a GPU-enabled simulation generated by the TS method.

```
1  // Phase 1
2  #pragma omp for
3  for(i=0;i<100;i++)
4    v[c1[i+10000]]=(0.001*(v[c1[i+0]]- (-75)))*(v[c1[i+100]]+···
5  #pragma omp for
6  for(i=0;i<200;i++)
7    v[c1[i+11700]]=((*(double*)&c1[i*2+10100])*···
8  ...
9  // Phase 2
10 #pragma omp for
11 for(i=0;i<100;i++) {
12   { // concatenated expressions for merging phases
13     v[c2[i+300]]=v[c2[i+0]]/(v[c2[i+100]]+···
14     v[c2[i+900]]=((((35*v[c2[i+400]])*v[c2[i+500]])*···
15 } }
16 ...
```

Figure 5.9: Pseudo code for an OpenMP-enabled simulation generated by the TS method.

in specific variables. They also execute simulations with smaller time steps than the required time resolution to improve the accuracy of the results. Therefore, our methods can simulate larger models in a typical use case, because the memory usage for buffering results can be reduced by selecting the variables to output and using a larger sampling interval.

We next investigate the computational performance of each method. Table 5.4 and Table 5.5 present the computation time and effective bandwidth, respectively. The time of the IS method is the computation time of the simulations without the initialization and output of the results. The time of the TS method is the execution time of the generated simulation program without the output. For simulations running on the GPU, the readback of the results was omitted. We also measure the amount of memory accesses by counting the push and pop operations in the bytecodes to calculate the effective bandwidth.

In Table 5.4, the TS method using CPU-1 is 6.1–26 times faster than the IS method using CPU-1. Using CPU-8, the factor increases to 95 with the LR-5000 model. Comparing CPU-8 and GPU, the TS method does not reduce the time as much as the IS method. However, the TS method using the GPU is up to 2.4 times faster than the IS method using the GPU.

It should be noted that the simulation has relatively few calculations per variable reference, because the simulation only contains scalar expressions. Therefore, the memory bandwidth bounds the performance. The TS method for the GPU achieves the highest bandwidth of 30 GB/s with the LR-5000 model (Table 5.5), although the efficiency remains low at 21% compared with the peak global memory bandwidth of the GPU, which is 144 GB/s.

The main cause of the low efficiency is the low number of threads on the GPU. Table 5.6 shows the number of bytecodes in each phase before unifying the bytecodes, which is equivalent to the number of threads in each phase, except for redundant threads and threads for parallelizing the sums. With smaller models, each phase launch few threads per CUDA core, which prevents the GPU from achieving a higher efficiency. For these models, a kernel launch has a relatively large overhead compared with the calculation. Consequently, the overhead bounds the performance.

5.6.2 Performance Analysis of Interpreter-based Simulation

Figure 5.10 shows the performance improvement of the six optimizations applied to the IS method. *Base* in Figure 5.10 is the naive GPU implementation and the others are incrementally applied optimizations: reordering the bytecodes (*Reorder*), remapping the variable IDs (*Remap*), adding redundant threads (*Redundant*), unifying the bytecodes (*Unify*), merging the phases (*Merge*), and parallelizing the sums (*Sum*).

Table 5.2: Total execution time with output of the results. Results are presented in seconds.

Model	Interpreter		Translator		
	CPU-1	GPU	CPU-1	CPU-8	GPU
Luo-Rudy	1.14	13.5	0.324	0.759	9.25
Wang	42.1	17.6	17.8	4.70	11.3
Rybak	45.4	30.6	29.1	5.18	12.7
LR-Ring	73.0	19.5	252	34.9	12.3
LR-100	92.0	21.4	34.5	6.20	12.5
Mix	190	38.4	294	30.8	17.9
LR-1000	1550	65.8	113	42.1	39.1

Table 5.3: Global memory footprint for the IS method (KB).

Model	Luo-Rudy	Wang	Rybak	LR-Ring	LR-100	Mix	LR-1000
Bytecodes	3.05	49.2	13.3	12.9	14.6	76.9	123
Constant table	2.38	37.8	82.6	77.0	96.4	221	851
Others	3.35	56.2	50.7	90.3	112	219	1100
Results	313	12,500	14,400	25,700	32,000	58,900	320,000
(1 step)	0.313	12.5	14.4	25.7	32.0	58.9	320
Total	322	12,600	14,500	25,900	32,200	59,400	322,000

Table 5.4: Computation time without output of the results. Results are presented in seconds.

Model	Interpreter		Translator		
	CPU-1	GPU	CPU-1	CPU-8	GPU
Luo-Rudy	1.03	7.04	0.169	0.596	6.88
Wang	40.0	6.00	2.73	1.77	7.02
Rybak	44.1	18.0	2.50	2.39	7.91
LR-Ring	71.9	8.10	4.86	3.31	8.14
LR-100	90.4	7.45	6.08	3.56	7.57
Mix	186	20.4	11.3	6.76	9.58
LR-1000	1570	18.2	64.2	19.5	11.4
LR-2000	3390	33.8	134	37.5	20.0
LR-3000	5250	50.2	204	55.8	26.4
LR-4000	7120	66.8	279	75.5	30.8
LR-5000	8990	83.6	355	95.0	35.4

Reorder increases the computation time by 48% with the Wang model because of the load imbalance between the warps. This optimization assigns the bytecodes to the threads in descending order of the length of the opcode sequences. Consequently, it assigns longer bytecodes to threads with smaller thread indices, resulting in a load imbalance between the warps. On the other hand, the GPU can hide the imbalance with the LR-1000 model because there are dozens of warps per streaming multiprocessor (SM) with this model.

Redundant decreases the computation time by 45% with the Luo-Rudy model. This model has few similar expressions and a relatively high ratio of divergent branches. *Redundant* reduces the ratio from 13% to 0.2% for the first 100 time steps measured by the CUDA Visual Profiler [6]. *Redundant* has no significant effect with the other models, which have many similar expressions.

Unify reduces the computation time by 46–60% compared with that of *Redundant*, except for the Luo-Rudy and LR-Ring models, because *Unify* reduces the number of memory access instances. In fact, it decreases the global memory read requests by 61% for the first 100 time steps with the LR-1000 model. On the other hand, the number of instructions to load from the global memory increases by 2,000% with the Luo-Rudy model and 19–61% with the other models compared with that of *Base* because of the redundant threads. However, the redundant threads run in the same way and access the same data as essential threads in the same warp. Therefore, redundant threads do not increase the memory requests and transactions, because the GPU coalesces contiguous or duplicated memory accesses from a warp.

Merge decreases the computation time by up to 20%. Table 5.7 gives the number of variations of the opcode sequences. *Merge* decreases one or two of the phases compared with those in *Base*, which leads to the speed up. However, *Merge* adds to the variation of the bytecodes in a phase because it concatenates multiple bytecodes. For instance, the number of different opcode sequences increases from 14 to 16 in the first phase of the Luo-Rudy model. Note that the PHML specification allows special functions that are evaluated after the evaluation of states. The Rybak model uses this type of functions and has a special function phase after the state phase. Consequently, the Mix model in *Base* has six phases that is more than the maximum number of phases of models that compose the Mix model.

Sum improves the performance by approximately 60% with the Wang and LR-Ring models that contain the sums of dozens of variables. With the other models, this optimization has no effect because they have no sums of many variables.

5.6.3 Performance Analysis of Translator-based Simulation

Figure 5.11 shows the computation time of the TS method with large coupled Luo-Rudy models. Comparing the total execution time, which is the sum of the translation, compilation, and simulation times, the TS method using the GPU is 1.1 times

Table 5.5: Effective memory bandwidth (GB/s).

Model	Interpreter		Translator		
	CPU-1	GPU	CPU-1	CPU-8	GPU
Luo-Rudy	0.20	0.030	1.3	0.35	0.031
Wang	0.40	2.6	5.8	9.0	2.3
Rybak	0.28	0.69	4.9	5.2	1.6
LR-Ring	0.24	2.1	3.5	5.2	2.1
LR-100	0.23	2.8	3.5	6.0	2.8
Mix	0.27	2.4	4.4	7.3	5.2
LR-1000	0.14	12.0	3.3	11	19
LR-5000	0.12	13.0	3.0	11	30

Table 5.6: Number of bytecodes before unifying them. Each item shows the series of the number of bytecodes in each phase separated by right arrows.

Model	Number of threads
Luo-Rudy	22 →3 →8
Wang	1,000 →100 →400
Rybak	842 →40 →480 →200
LR-Ring	1,924 →322 →1 →640
LR-100	2,399 →399 →800
Mix	4,241 →539 →1,680 →200
LR-1000	23,999 →3,999 →8,000

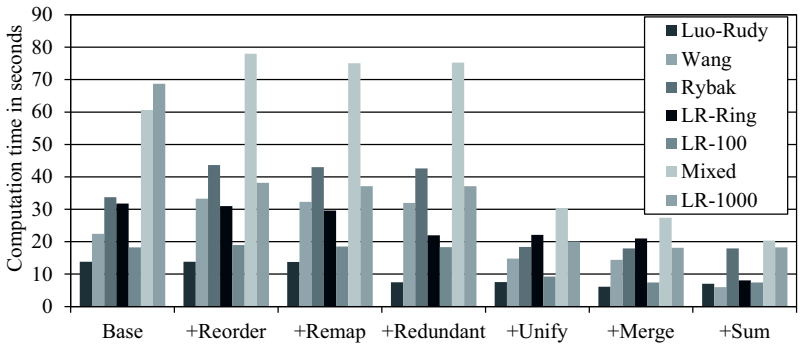


Figure 5.10: Computation time of the IS method with each optimization. Results are presented in seconds.

faster than that using CPU-8 with the LR-5000 model. With the LR-5000 model, the translation of the model takes approximately 290 seconds, which is a bottleneck for the TS method. Therefore, further improvement of this process is required, but this discussion is beyond the scope of this thesis.

However, with the LR-5000 model, the simulation program using the GPU is 250 times faster than the IS method on CPU-1 presented in Table 5.4. Moreover, the generated source code can be reused to simulate the model with different initial values of the ODEs without translating the model again. Therefore, the TS method is beneficial for a parameter sweep experiment on a model that repeatedly simulates the same model while varying the initial values.

Table 5.8 shows the compilation time of the generated source codes. *Base* shows the time of the simulation programs generated using a naive translation method without the bytecode unification and export of the constant tables to a file. *Unify* is the time with the bytecode unification, and *Export* is the time with both improvements. The compiler failed to compile the *Base* source code for the GPU with the Mix and LR-1000 models. With *Base*, the compiler took 4.4 hours to compile the source code for the CPU-1 with the LR-1000 model. The compilation time increased to 10 hours with the bytecode unification because of the folding of the constants and the unrolling of the loops for processing the unified bytecodes. On the other hand, *Export* reduced the time to 27 seconds, because it prevents the compiler optimizations described above. The compilation time of the GPU-enabled codes is typically shorter than that of the CPU-based codes. This result is due to the difference of the compilers for the CPU and GPU.

Figure 5.12 illustrates the execution time of each kernel in the simulation program for LR-5000 generated by the TS method, with varying the TB size in the multiple of the warp size of 32. The computation time is almost convex downward for phase 1 with the TB size, while execution time of other phases show no significant changes. The proposed method properly selects 160, 96, 32, and 128 as the TB size of the kernel for phase 1, 2, 3, and updating states, respectively.

5.6.4 Precision Analysis of Simulation Results

Finally, we analyze the accuracy of simulation results of the proposed method. The parallelization can change the order of floating-point calculations, which can generate some errors including loss of trailing digits, loss of significance, and round-off errors. In particular, parallelizing sums can cause these errors. In addition, the implementation of floating-point calculations might differr between the CPU and the GPU. Therefore, we compare the simulation results of the CPU-based method and that of the GPU-based method.

Table 5.9 shows the relative root mean square errors (RRMSEs) between results of the proposed method and that of the interpreter of Flint running on CPU-1. In

Table 5.7: Number of different opcode sequences in each phase. Each item shows the series of numbers separated by right arrows.

Model	Base	Merge
Luo-Rudy	14 →6 →2 →1 →3	16 →3 →3
Wang	7 →1 →1 →3	7 →1 →3
Rybak	8 →5 →1 →5 →4	11 →2 →5 →4
LR-Ring	16 →9 →2 →1 →4	19 →6 →1 →4
LR-100	15 →7 →1 →1 →3	18 →4 →3
Mix	26 →12 →3 →1 →11 →4	33 →7 →11 →4
LR-1000	15 →7 →1 →1 →3	18 →4 →3

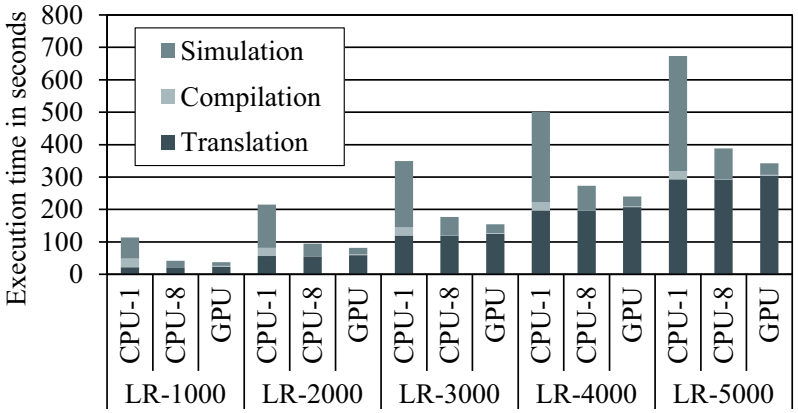


Figure 5.11: Computation time of the TS method using coupled Luo-Rudy models with varying numbers of cells.

Table 5.8: Compilation time for the source codes generated using the TS method. Results are presented in seconds.

Model	CPU-1			GPU		
	Base	Unify	Export	Base	Unify	Export
Luo-Rudy	0.49	0.12	0.13	3.2	3.1	2.6
Wang	13	120	13	72	3.6	2.3
Rybak	35	68	25	66	3.8	2.5
LR-Ring	28	100	250	630	5.1	2.8
LR-100	49	140	27	1,100	5.8	2.7
Mix	230	2,200	280	—	11	3.2
LR-1000	16,000	36,000	27	—	290	2.7

this experiment, each simulation runs 10,000 steps with $l = 100$ milliseconds and $s = 0.01$ milliseconds using the Euler method.

The IS method on the GPU has errors with Wang, LR-100, Mix, and LR-1000, although they are sufficiently small. With Luo-Rudy based models, the TS method has larger errors than the errors of the IS method on the GPU. The constant folding by the compilers may cause these errors.

On the other hand, the TS method has a significant error with Wang on the GPU. This error is caused by the parallelization of the sums because the error is eliminated without parallelizing the sums. The error firstly occurs at a particular time step and is accumulated during the simulation.

5.7 Conclusion

In this chapter, we describe two acceleration methods for the general physiological simulator Flint using a GPU. Our interpreter-based method speeds up the simulation of Flint by running its interpreter on the GPU. We use a level scheduling algorithm to automatically parallelize the evaluation of many expressions in a physiological model. The GPU calculates all the expressions in each phase using a kernel launch. To achieve a higher performance, we unify similar bytecodes of different expressions, because there are many similar expressions in a model that contains thousands of modules connected to each other. This optimization decreases the number of divergent branches and the global memory accesses.

The proposed translator-based method generates a source code from the bytecodes of our interpreter-based method. The bytecode unification reduces the generated code size, which speeds up code compilation. These methods enable a fast simulation of large, general, and heterogeneous models using a GPU. Moreover, it allows physiologists to easily accelerate a simulation using their own workstation with a GPU.

The experimental result shows that our interpreter-based method is up to 24 times faster than the existing method running on a core of a CPU. The program generated by our translator-based method accomplishes simulations 2.4 times faster than the interpreter-based method running on the GPU.

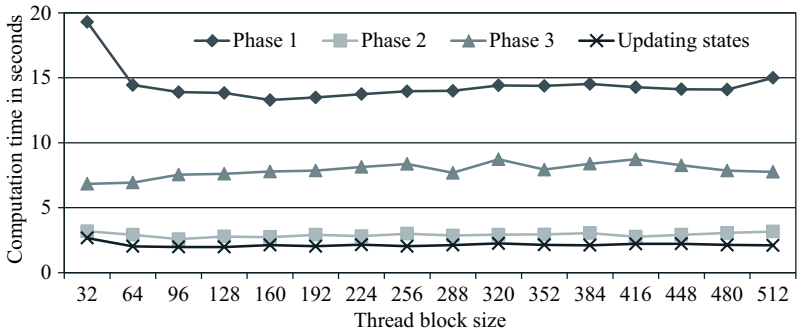


Figure 5.12: Execution time of each kernel in the simulation program for LR-5000 generated by the TS method.

Table 5.9: Relative root mean square errors.

Model	**Interpreter**	**Translator**	
	GPU	**CPU-1**	**GPU**
Luo-Rudy	0.0	9.4×10^{-4}	9.4×10^{-4}
Wang	1.4×10^{-13}	0.0	3.2×10^{-1}
Rybak	0.0	0.0	0.0
LR-Ring	0.0	8.9×10^{-7}	8.9×10^{-7}
LR-100	9.3×10^{-20}	1.3×10^{-2}	1.3×10^{-2}
Mix	4.8×10^{-20}	6.5×10^{-3}	2.9×10^{-2}
LR-1000	4.1×10^{-20}	5.6×10^{-3}	5.6×10^{-3}

Chapter 6

Conclusion

6.1 Summary of This Thesis

In this thesis, we discussed techniques for efficient computation of irregular programs on the graphics processing unit (GPU). Our approach is to statically rearrange thread assignment to gather similar threads on a single instruction, multiple data (SIMD) core before executing kernels on the GPU, aiming at reducing the number of irregular control flow and memory access. In particular, we have focused on the all-pairs shortest path (APSP) problem and a general biophysical simulator.

With the APSP problem, we examine the memory access pattern to find threads that show similar behavior. This technique can be also applied to a problem that includes the same type of multiple subproblem instances that can be computed in parallel, for instance, the single source shortest path (SSSP) problems for the APSP problem. Our technique will improve the throughput of computing subproblems if the subproblems have the similar memory access pattern, even if there are irregular and uncoalesced memory accesses in each subproblems. Note that this technique can be applied when developing a GPU program from a sequential program by carefully analyzing the control flow and data dependencies, in addition to improving the efficiency of existing parallel programs for the GPU.

The biophysical simulator varies its computation depending on mathematical expressions in the input model. For this application, we automatically rearrange thread assignment at runtime, examining the data dependency and similarity of the mathematical expressions. The simulation techniques will be applied to other ODE-based simulations. In addition, this technique can be applied to an application that significantly varies the behavior of threads for each thread, but the behavior is statically determined.

In addition, we described another acceleration method for the APSP problem based on the Floyd-Warshall (FW) algorithm on the GPU. The contribution for

each application is summarized as follows.

Task Parallel Algorithm for Finding APSPs Using the GPU We have presented a fast algorithm for finding APSPs using the compute unified device architecture (CUDA). This algorithm exploited the coarse-grained task parallelism among different single source shortest path (SSSP) problems, in addition to the fine-grained data parallelism in each SSSP problem. This combined parallelism allows threads to share data using the on-chip shared memory. It also allows us to run more threads with less kernel launches, leading to efficient use of the highly multithreaded architecture of the GPU. The experimental results show that the proposed method is 2.8–13 times faster than the iterative SSSP-based method. This technique can be also applied to a batch execution of single source based algorithms with varying the source vertex for a large sparse graph, such as SSSP and breadth first search.

Iterative Blocked FW Algorithm Using the GPU We have described two acceleration methods of the iterative Blocked FW (BFW) algorithm using CUDA. The first method applies a fast matrix multiplication routine to the computation, which improved the computational efficiency by reducing shared memory accesses. This method also has an auto-tuning technique that automatically determines a suitable calculation area size of thread block by estimating the number of instructions executed on an SM. The second method uses two-level blocking to cache a tile on the shared memory. As a result, the former method showed slightly higher performance and reduced the computation time by 17–45% compared to that of an existing recursive BFW method for graphs with 256–1024 vertices.

GPU-based Fast Simulation of General Biophysical Models We have presented two fast general physiological simulation techniques using the GPU. We use a level scheduling algorithm to automatically parallelize the evaluation of many expressions in a physiological model. In addition, we unify similar bytecodes of different expressions, to decreases the number of divergent branches and memory accesses. The first method interprets the bytecodes on the GPU and the other method generates a source code from the bytecodes. The experimental results show that the first method is up to 24 times faster than the existing method using a CPU. The second method accomplishes simulations 2.4 times faster than the first method. These methods enable a fast simulation of large, general, and heterogeneous models using a GPU. Moreover, it allows physiologists to easily accelerate a simulation using their own workstation with a GPU.

6.2 Future Work

In this work, we focus on APSP problems with a graph that can be stored on the device memory of the GPU. However, there are real-world graphs that are too large to be stored entirely on the memory. For example, there is a large graph that represents road networks in the United States of America [2]. This graph includes approximately 24,000,000 vertices and 58,000,000 edges, which has approximately 750 times as many vertices as the graph evaluated in Chapter 3. Other examples are social networks and web graphs [56], the analysis of which recently attracts the attention of researchers as a data-intensive application. Such large graphs involve data decomposition due to the lack of memory capacity of the GPU. In addition, the implementations should be developed using more high computational environments such as multiple GPUs and the cluster of GPUs. Meanwhile, the task parallel scheme used for the APSP problem can improve the throughput of computation that includes the same type of independent subproblems such as parameter sweep applications.

The GPU-based biophysical simulator described in Chapter 5 can be extended and improved in several ways. First, the bytecode optimizations are possible to improve the performance. The basic compiler data-flow optimization techniques can be applied to this process, such as common subexpression elimination among bytecodes, constant folding and dead store elimination. To process larger models, it might be required to improve the accuracy of simulation results and to use clusters of GPUs. This simulator can be extended to process other ODE-based simulations. In addition, the bytecode reordering and unification techniques can be applied to applications that statically assign different operations to each thread.

Bibliography

[1] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in *Proc. 14th Int'l Conf. High Performance Computing (HiPC'07)*, Dec. 2007, pp. 197–208.

[2] "9th DIMACS implementation challenge - Shortest paths," http://www.dis.uniroma1.it/~challenge9/download.shtml.

[3] D. Shreiner, M. Woo, J. Neider, and T. Davis, *OpenGL Programming Guide*, 5th ed. Reading, MA: Addison-Wesley, Aug. 2005.

[4] Microsoft Corporation, "DirectX," 2007. [Online]. Available: http://www.microsoft.com/directx/

[5] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[6] NVIDIA Corporation, "CUDA Programming Guide Version 4.2," Apr. 2012. [Online]. Available: http://developer.nvidia.com/cuda/

[7] M.-Y. Wu and W. Shu, "Mimd programs on simd architectures," in *6th Symp. Frontiers of Massively Parallel Computing (Frontiers '96)*, oct 1996, pp. 162–170.

[8] W. SHU and M.-Y. Wu, "Solving dynamic and irregular problems on simd architectures with runtime support," in *Int'l Conf. Parallel Processing (ICPP 1993)*, vol. 2, aug 1993, pp. 167–174.

[9] A. Di Blas and R. Hughey, "Explicit simd programming for asynchronous applications," in *IEEE Int'l Conf. Application-Specific Systems, Architectures, and Processors*, 2000, pp. 258–267.

[10] A. Nakaya, S. Goto, and M. Kanehisa, "Extraction of correlated gene clusters by multiple graph comparison," *Genome Informatics*, vol. 12, pp. 34–43, Dec. 2001.

[11] N. Shenoy, "Retiming: theory and practice," *Integration, the VLSI J.*, vol. 22, no. 1/2, pp. 1–21, Aug. 1997.

[12] R. W. Floyd, "Algorithm 97: Shortest path," *Communications of the ACM*, vol. 5, no. 6, p. 345, 1962.

[13] S. Warshall, "A theorem on boolean matrices," *Journal of the ACM*, vol. 9, no. 1, pp. 11–12, 1962.

[14] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[15] G. J. Katz and J. T. Kider, "All-pairs shortest-paths for large graphs on the GPU," in *Proc. 23rd ACM SIGGRAPH/EUROGRAPHICS Symp. Graphics Hardware (GH'08)*, Jun. 2008, pp. 47–55.

[16] P. Micikevicius, "General parallel computation on commodity graphics hardware: Case study with the all-pairs shortest paths problem," in *Proc. Int'l Conf. Parallel and Distributed Processing Techniques and Applications (PDPTA'04)*, vol. 3, Jun. 2004, pp. 1359–1365.

[17] U. Bondhugula, A. Devulapalli, J. Dinan, J. Fernando, P. Wyckoff, E. Stahlberg, and P. Sadayappan, "Hardware/software integration for FPGA-based all-pairs shortest-paths," in *Proc. 14th IEEE Symp. Field-Programmable Custom Computing Machines (FCCM'06)*, Apr. 2006, pp. 152–164.

[18] T. Srinivasan, R. Balakrishnan, S. A. Gangadharan, and V. Hayawardh, "A scalable parallelization of all-pairs shortest path algorithm for a high performance cluster environment," in *Proc. 13th Int'l Conf. Parallel and Distributed Systems (ICPADS'07)*, vol. 1, Sep. 2006, cD-ROM (8 pages).

[19] R. Bellman, "On a routing problem," *Quarterly of Applied Mathematics*, vol. 16, no. 1, pp. 87–90, 1958.

[20] J. Lester Randolph Ford and D. R. Fulkerson, *Flows in Networks*. Princeton University Press, 1962.

[21] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. The MIT Press, Sep. 2001.

[22] G. Venkataraman, S. Sahni, and S. Mukhopadhyaya, "A blocked all-pairs shortest-path algorithm," in *Proc. 7th Scandinavian Workshop Algorithm Theory (SWAT'07)*, Jul. 2000, pp. 419–432.

[23] A. Buluç, J. R. Gilbert, and C. Budak, "Gaussian elimination based algorithms on the GPU," University of California, Tech. Rep. UCSB/CS-2008-15, Nov. 2008.

[24] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *Proc. Int'l Conf. High Performance Computing, Networking, Storage and Analysis (SC'08)*, Nov. 2008, 11 pages (CD-ROM).

[25] A. Bleiweiss, "GPU accelerated pathfinding," in *Proc. 23rd ACM SIGGRAPH/EUROGRAPHICS Symp. Graphics Hardware (GH'08)*, Jun. 2008, pp. 65–74.

[26] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, Jul. 1968.

[27] P. Harish, V. Vineet, and P. J. Narayanan, "Large graph algorithms for massively multithreaded architectures," International Institute of Information Technology Hyderabad, Tech. Rep. IIIT/TR/2009/74, Feb. 2009.

[28] S.-C. Han, F. Franchetti, and M. Püshel, "Program generation for the all-pairs shortest path problem," in *Proc. 15th Int'l Conf. Parallel Architectures and Compilation Techniques (PACT'07)*, Sep. 2006, pp. 222–232.

[29] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, 2nd ed. Reading, MA: Addison-Wesley, Jan. 2003.

[30] A. Klimovitski, "Using SSE and SSE2: Misconceptions and reality," in *Intel Developer Update Magazine*, Mar. 2001.

[31] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel Programming in OpenMP*. San Mateo, CA: Morgan Kaufmann, Oct. 2000.

[32] D. A. Bader and K. Madduri, "GTgraph," http://www.cc.gatech.edu/~kamesh/GTgraph/.

[33] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Proc. 4th SIAM Int'l Conf. Data Mining (SDM'04)*, Apr. 2004, pp. 442–446.

[34] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Proc. Int'l Conf. High Performance Computing, Networking, Storage and Analysis (SC'08)*, Nov. 2008, pp. 1–12.

[35] A. Nukada and S. Matsuoka, "Auto-tuning 3-d FFT library for CUDA GPUs," in *Proc. Int'l Conf. High Performance Computing, Networking, Storage and Analysis (SC'09)*, Nov. 2009, 10 pages (CD-ROM).

[36] K. Matsumoto and S. G. Sedukhin, "A solution of the all-pairs shortest paths problem on the cell broadband engine processor," *IEICE Trans. Information and Systems*, vol. E92-D, no. 6, pp. 1225–1231, Jun. 2009.

[37] W. J. van der Laan, "decuda," http://wiki.github.com/laanwj/decuda.

[38] T. Okuyama, F. Ino, and K. Hagihara, "A task parallel algorithm for finding all-pairs shortest paths using the gpu," *Int'l Journal of High Performance Computing and Networking*, vol. 7, no. 2, pp. 87–98, Apr. 2012.

[39] D. Sato, Y. Xie, J. N. Weiss, Z. Qu, A. Garfinkel, and A. R. Sanderson, "Acceleration of cardiac tissue simulation with graphic processing units," *Medical and Biological Engineering and Computing*, vol. 47, no. 9, pp. 1011–1015, Aug. 2009.

[40] Z. Taylor, O. Comas, M. Cheng, J. Passenger, D. Hawkes, D. Atkinson, and S. Ourselin, "On modelling of anisotropic viscoelasticity for soft tissue simulation: Numerical solution and GPU execution," *Medical Image Analysis*, vol. 13, no. 2, pp. 234–244, May 2009.

[41] F. V. Lionetti, A. D. McCulloch, and S. B. Baden, "Source-to-source optimization of CUDA C for GPU accelerated cardiac cell modeling," in *Proc. 16th Int'l Euro-Par Conf. Parallel processing (EuroPar'10)*, 2010, pp. 38–49.

[42] V. Garcia, A. Liberos, A. Climent, A. Vidal, J. Millet, and A. González, "An adaptive step size GPU ODE solver for simulating the electric cardiac activity," in *Computing in Cardiology*, Sep. 2011, pp. 233–236.

[43] J. Ackermann, P. Baecher, T. Franzel, M. Goesele, and K. Hamacher, "Massively-parallel simulation of biochemical systems," in *Proc. Massively Parallel Computational Biology on GPUs*, Sep. 2009, 12 pages.

[44] Y. Asai, T. Abe, M. Okita, T. Okuyama, N. Yoshioka, M. N. Shigetoshi Yokoyama, K. Hagihara, and H. Kitano, "Spatiotemporal multilevel modeling of physiological systems and simulation platform: Physiodesigner, flint and flint k3 service," in *Proc. 12th IEEE/IPSJ Int'l Symp. Applications and the Internet (SAINT 2012)*, Jul. 2012, 5 pages.

[45] E. M. Heien, M. Okita, Y. Asai, T. Nomura, and K. Hagihara, "insilicosim: an extendable engine for parallel heterogeneous biophysical simulations," in *Proc.*

3rd Int'l Conf. Simulation Tools and Techniques (SIMUTools '10), 2010, pp. 78:1–78:10.

[46] H. El-Rewini, T. G. Lewis, and H. H. Ali, *Task Scheduling in Parallel and Distributed Systems*. Englewood Cliffs, NJ: PTR Prentice Hall, 1994.

[47] L. Dematté and D. Prandi, "GPU computing for system biology," *Brief Bioinform*, vol. 11, no. 3, pp. 323–333, May 2010.

[48] M. Hucka, A. Finney, H. M. Sauro, H. Bolouri, J. C. Doyle, H. Kitano, A. P. Arkin, B. J. Bornstein, D. Bray, A. Cornish-Bowden, A. A. Cuellar, S. Dronov, E. D. Gilles, M. Ginkel, V. Gor, I. I. Goryanin, W. J. Hedley, T. C. Hodgman, J.-H. S. Hofmeyr, P. J. Hunter, N. S. Juty, J. L. Kasberger, A. Kremling, U. Kummer, N. L. Novère, L. M. Loew, D. Lucio, P. Mendes, E. Minch, E. D. Mjolsness, Y. Nakayama, M. R. Nelson, P. F. Nielsen, T. Sakurada, J. C. Schaff, B. E. Shapiro, T. S. Shimizu, H. D. Spence, J. Stelling, K. Takahashi, M. Tomita, J. M. Wagner, J. Wang, and the rest of the SBML Forum, "The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models," *Bioinformatics*, vol. 19, no. 4, pp. 524–531, Mar. 2003.

[49] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard, Version 2.2*. Stuttgart, Germany: High Performance Computing Center Stuttgart (HLRS), 2009.

[50] "Physiodesigner," http://physiodesigner.org/, 2012, accessed July 20, 2012.

[51] Y. Asai, Y. Suzuki, Y. Kido, H. Oka, E. Heien, M. Nakanishi, T. Urai, K. Hagihara, Y. Kurachi, and T. Nomura, "Specifications of insilicoml 1.0: A multilevel biophysical model description language," *J. Physiological Sciences*, vol. 58, no. 7, pp. 447–458, Dec. 2008.

[52] C.-H. Luo and Y. Rudy, "A model of the ventricular cardiac action potential. depolarization, repolarization, and their interaction," *Circulation Research*, vol. 68, no. 6, pp. 1501–1526, Jun. 1991.

[53] F. Mahmud, N. Shiozawa, M. Makikawa, and T. Nomura, "Reentrant excitation in an analog-digital hybrid circuit model of cardiac tissue," *Chaos: An Interdisciplinary Journal of Nonlinear Science*, vol. 21, no. 2, Jun. 2011, 14 pages.

[54] X.-J. Wang and G. Buzsáki, "Gamma oscillation by synaptic inhibition in a hippocampal interneuronal network model," *J. Neuroscience*, vol. 16, no. 20, pp. 6402–6413, Oct. 1996.

[55] I. A. Rybak, N. A. Shevtsova, M. Lafreniere-Roula, and D. A. McCrea, "Modelling spinal circuitry involved in locomotor pattern generation: insights from deletions during fictive locomotion," *J. Physiology*, vol. 577, no. 2, pp. 617–639, Dec. 2006.

[56] "Stanford Large Network Dataset Collection," http://http://snap.stanford.edu/data/index.html, 2012, accessed November 30, 2012.