Title
Parallel Algorithms for Medical Informatics on Data-Parallel Many-Core Processors

Author
Moazeni, Maryam

Publication Date
2013

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA

Los Angeles

Parallel Algorithms for Medical Informatics on Data-Parallel Many-Core Processors

A dissertation submitted in partial satisfaction of the

requirements for the degree Doctor of Philosophy

in Computer Science

by

Maryam Moazeni

2013

ABSTRACT OF THE DISSERTATION

Parallel Algorithms for Medical Informatics on Data-Parallel Many-Core Processors

by

Maryam Moazeni

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2013

Professor Majid Sarrafzadeh, Chair

The extensive use of medical monitoring devices has resulted in the generation of tremendous amounts of data. Storage, retrieval, and analysis of such data require platforms that can scale with data growth and adapt to the various behavior of the analysis and processing algorithms. In recent years, many-core processors and more specifically many-core Graphical Processing Units (GPUs) have become one of the most promising platforms for high performance processing of data, due to the massive parallel processing power they offer. However, many of the algorithms and data structures used in medical and bioinformatics systems do not follow a data-parallel programming paradigm, and hence cannot fully benefit from the parallel processing power of

data-parallel many-core architectures.

In this dissertation, we present three techniques to adapt several non-data parallel applications in different dwarfs to modern many-core GPUs. First, we present a load balancing technique to maximize parallelism in non-serial polyadic Dynamic Programming (DP), which is a family of dynamic programming algorithms with more non-uniform data access pattern. We show that a bottom-up approach to solving the DP problem exploits more parallelism and therefore yields higher performance. We achieve 228X speedup over an equivalent CPU implementation.

Second, we introduce a parallel hash table as a parallel-friendly lock-free dynamic hash table. The parallel hash table structure reduces the contention on the shared objects in lock-free hash table and achieves significant throughput on many-core processor architectures. To reduce the contention, it creates multiple instances of a hash table and uses a table assignment function to distribute hash table operations to different hash table instances and guarantees key uniqueness. We achieved roughly 27X speedup over counter-part multi-thread lock-free hash table on CPU.

Third, we present a memory optimization technique for the software-managed scratchpad memory based on G80, GT200, and Fermi architectures to alleviate the constraints of using scratchpad memory. We propose a memory optimization scheme that minimizes the usage of memory space by discovering the chances of memory reuse with the goal of maximizing application performance. Our solution is based on graph coloring. Our evaluations show that using this technique can reduce the execution time of applications on GPUs by up to 22% over the non-optimized GPU implementation.

In addition, by leveraging massive parallelism of GPUs, we introduce a novel time-series

searching technique for multi-dimensional time series. Searching for time series is an intuitive and practical approach to study similarity of patterns, events, and activities in patient histories. However, its computational intensity has traditionally been a constraint in the development of a complex algorithm that can handle patterns in multi-dimensional signals considering noise, scaling, and time correlation between dimensions. Using GPUs, we are able to achieve high speed up in processing signals, while improving the quality of the search algorithm and tackle problems such as noise and scaling. We used data collected from two medical monitoring devices, a Personal Activity Monitor (PAM) and Medical Shoe to evaluate our approach and show that our technique results in up to 25X speed up and up to 15 point improvement in Normalized Discounted Cumulative Gain (NDCG) for such application.

The dissertation of Maryam Moazeni is approved.

Glen Reinman

Jens Palsberg

Alex Bui

Majid Sarrafzadeh, Committee Chair

University of California, Los Angeles

2013

To my parents …

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to greatly thank everyone who was a part of my journey. First, I would like to acknowledge my advisor Professor Majid Sarrafzadeh for his advises in planning my research. His vision and great advices on selecting the right topic and taking the correct approach helped me through all these years.

I am grateful to Professor Alex Bui who was always like an advisor to me, and helped me greatly through my research. I wish to thank Professor Jens Palsberg and Professor Glen Reinman for their insightful comments which helped me greatly in planning my PhD dissertation.

I would also like to acknowledge Microsoft and my managers at Microsoft for their full support and encouragement.

I would like to thank all my friends at UCLA Computer Science Department and especially members of Embedded and Reconfigurable Lab for their valuable friendship which made my studies joyful.

I wish to express gratitude to my dear parents, Nahid and Mahmoud, who always made me and my siblings their number one priority. My mother was always my role model for a woman with high aims for education. My father was my role model in seeking perfection. Working on my PhD while having a fruitful career was never easy, and I couldn't have finished my studies without their support. I am also grateful to my brother, Ramin, who helped me start my graduate studies abroad and to my sister, Mina, who always believed in me.

I would like to acknowledge my grandparents who were and will always be a big part of my life and filled my life with their love. My grandfather, Dr. Sayed Ali Tabatabaee, rest in peace, who was a great scholar and was my role model in modesty since childhood. He used to talk to me about the value of knowledge when I was only six years old and used to help him take his daily walk around the garden.

Last but not least, I adore the love and support from my dear and loving husband, Alireza, throughout this time. I had the most joyful graduate studies at UCLA, where we fall in love and made lots of joyful

memories. He was always there for me through all the difficult times, without him it would have been unquestionably much harder to finish a PhD while having a challenging career.

VITA

2002-2006	B.Sc. (Computer Engineering) Department of Computer Engineering University of Isfahan
2009-Present	Software Developer Engineer SQL Reporting Services Microsoft Inc.
2006-2008	Teaching Associate Computer Science Department University of California, Los Angeles
2008-2009	National Library of Medicine Fellow University of California, Los Angeles

PUBLICATIONS

Maryam Moazeni, Majid Sarrafzadeh, "High Performance Multi-dimensional Signal Search With Applications in Remote Medical Monitoring", IEEE Conference on Body Sensor Networks (BSN), May 2013.

Maryam Moazeni, Majid Sarrafzadeh, "Lock-free Hash Table on Graphics Processors", 2012 Symposium on Application Accelerators in High-Performance Computing, July 2012.

Maryam Moazeni, Alex Bui, Majid Sarrafzadeh, "Non-Serial Polyadic Dynamic Programming on a Data-Parallel Many-core Architecture", 2011 Symposium on Application Accelerators in High-Performance Computing, July 2011.

Maryam Moazeni, Yi Zou, Mahsan Rofouei, Alex Bui, Majid Sarrafzadeh, Jason Cong, "A Performance Evaluation of General Purpose Applications on Multithreaded GPUs", UCLA Computer Science Technical Report, 2009.

Maryam Moazeni, Alex Bui, Majid Sarrafzadeh "A Memory Optimization Technique for Software-Managed Scratchpad Memory in GPUs", IEEE Symposium on Application Specific Processors, July 2009.

Maryam Moazeni, Alex Bui, Majid Sarrafzadeh, "Accelerating Total Variation Regularization for Matrix-Valued Images on GPUs", ACM International Conference on Computing Frontiers, May 2009.

Maryam Moazeni, Alex Bui, Majid Sarrafzadeh, "Accelerating TV Regularization for Images on GPUs", NLM Conference, June 2009.

Mahsan Rofouei, Maryam Moazeni, Majid Sarrafzadeh, "Fast GPU-based Space-Time Correlation for Activity Recognition in Video Sequences", IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTImedia), October 2008.

Maryam Moazeni, Alireza Vahdatpour, Majid Sarrafzadeh , "Communication bottleneck in hardware-software partitioning", Poster in ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), February 2008.

Alireza Vahdatpour, Foad Dabiri, Maryam Moazeni, Majid Sarrafzadeh , "Theoretical Bound and Practical Analysis of Minimum Connected Dominating Set in Ad-Hoc and Sensor Networks", International Symposium on Distributed Computing (DISC), September 2008.

Maryam Moazeni, Alireza Vahdatpour, Majid Sarrafzadeh, "HEAP: A Hierarchical Energy Aware Protocol for Routing and Aggregation in Sensor Networks", International Workshop on Performance Control in Wireless Sensor Networks (PWSN), 2007.

CHAPTER 1

Introduction

1.1 Healthcare Application Demand for High-Performance Computing

Recent advances in electronics industry have resulted in transformational change in health management and medicine. As a result of proliferation of ubiquitous sensing devices along with advances in wireless communication technology and portable devices new domains of applications, specifically in the area of health care are created. This creates a demand for methods required to optimize the acquisition, storage, retrieval, and processing of information in healthcare domain. The large health care markets, along with research opportunities, are strong motivations for researchers in computer science to develop solutions that can mitigate these challenges.

Medical Monitoring systems are based on the use of sensing technologies to constantly monitor subjects' vital signs, behavior, and activities. The proliferation of convenient hand-held devices, wearable sensors, and broadband wireless services for monitoring and guidance has led to generation of tremendous amount of data. The higher processing power and the abundance of data in the centralized medical monitoring systems allows researchers to examine various hypotheses, to extract priori unknown information from the data. Such information can be used for diagnosis, providing feedback for patients or

extending the knowledge of clinicians. This demands high-performance computing on the centralized medical systems for efficient storage, retrieval and analysis of data.

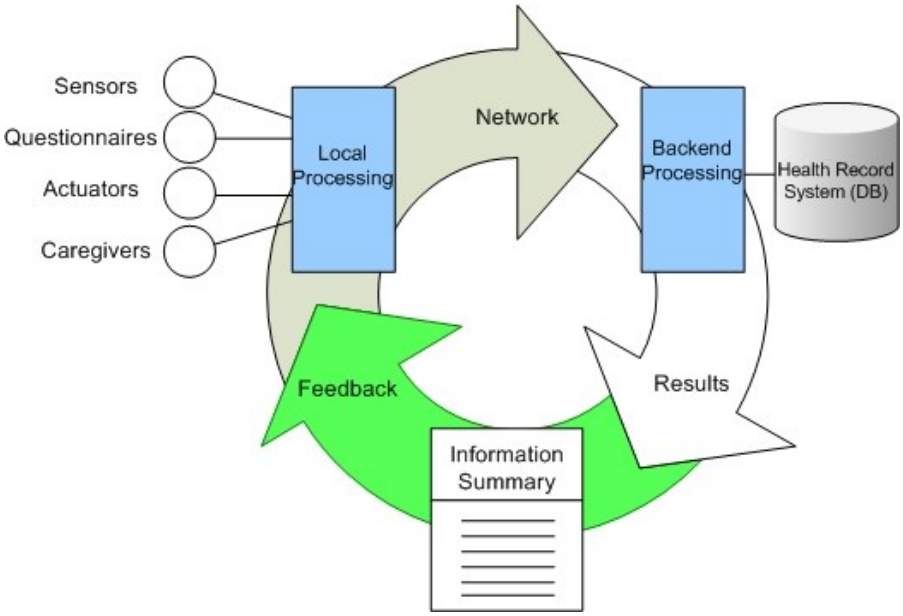

Figure 1. Typical steps for data collection, storage, analysis in medical systems.

In addition to centralized medical centers, more and more individuals and physician are demanding low cost health data mining platforms and systems which can process massive amount of data with relatively high accuracy and high speed. Low cost off-the-shelf commodity GPUs have potential to enable personal computers to process data collected from patients.

Medical imaging is also used widely in the diagnosis and treatment of most medical problems, but many advances in this field have been constrained to the research environment, due to a lack of computational power. Medical imaging applications involve computationally intensive algorithms. With advances in scanner technologies the amount of imaging data is ever growing, which makes the problem even more computationally difficult. Examples are the multi-valued imaging data, such as DTI (Diffusion Tensor Imaging) or multi-channel acquisitions, wherein each voxel is a feature vector of 6-100 dimensions. In recent years cloud computing has offered great computing power to virtually any connected computing platform. It can also be used in many bioinformatics systems. However, several

concerns regarding data privacy prevents medical systems to use centralized cloud systems for storage and processing of medical information. Hence, using low cost powerful GPUS that enables localized processing and storage clusters can be considered as a feasible alternative.

Bioinformatics is an important field in scientific computing and has high demand for computational power. Researchers in this field are dealing with computations on petabytes of and are restricted by the use of large grids and clusters, which are not easily manageable. Even though massively parallel systems are used these days to processes bioinformatics data, the sequential nature of many bioinformatics algorithms results in inefficient use of many parallel resources. Hence adapting the sequential algorithms to better leverage parallel architectures can result in higher performance and faster execution of such algorithms.

1.2 General-Purpose Computing on Graphics Processors

Increasing parallelism, rather than increasing clock-rate, has become the primary engine of processor performance growth, and this trend continues with the integration of hundreds of cores onto a single chip termed as many-core. Many-core processors can offer higher performance or power efficiency compared to current single-core or multi-core processors [Mayun]. Modern Graphic Processing Unit (GPU) has evolved into massively parallel, many-core processors with very high memory bandwidth.

However, compared with CPUs, the hardware architecture of GPUs differs significantly. For instance, current GPUs provide parallel lower-clocked execution capabilities on over a hundred cores whereas current CPUs offer out-of-order execution capabilities on a much smaller number of cores. Unlike CPU cores instructions are issued in order however and there is no branch prediction and no speculative execution. GPUs are specialized for compute-intensive computations and therefore, they are designed with more transistors dedicated to data processing rather than data caching and control flow. Therefore, memory latency can be hidden by calculations rather than big data caches. Therefore, these differences imply that parallel decomposition techniques that are used for multi-core implementations may not suffice or succeed at achieving the maximum parallelism if applied to many-core processors.

The interest in Graphics Processing Unit (GPU) programming for general-purpose computations has been driven by relatively recent improvements in the programmability and flexibility of graphics hardware. Modern GPUs offer general-purpose instruction sets for non-visual general-purpose computations and they are available as an inexpensive commodity coprocessor. CUDA [Nvidia] introduced by NVIDIA, has improved the suitability of GPUs for high-performance computing, by increasing their flexibility and programmability. A programming interface alternative to CUDA is available for AMD Stream Processor, using the R580 GPU, in the form of the Close to Metal (CTM) [Amd] compute runtime driver which, completely exposes ISA to the programmer; thus providing fine-grained control.

The processing power of GPUs has been successfully exploited in broad domains, especially in scientific, imaging and database applications [Owens, Che]. Cloud dynamics simulation using partial differential equations [Harris], CUDA-based MRI reconstruction [Stone], deformable image registration [Mayun], and cutoff pair potential for molecular modeling [Rodrigues] have also been successfully implemented on GPUs and shown impressive speedups.

1.3 Leveraging GPU for Healthcare

In this dissertation, I present structured methods for enhancing the performance of a set of algorithms, data structures or applications on data-parallel many-core processors. The evaluation of the proposed techniques is focused on health care domain applications; however, as they are enhancing generic data structural and algorithmic bottlenecks, they are not limited to this domain and can be applied to other domains as well. The techniques are concentrated on fundamental algorithms and data structures which are frequently used in medical and bioinformatics applications. These algorithms and data structures exhibit challenges for implementation on data-parallel many-core architectures and programming model as they have not been designed for many-core architectures. The goal for the proposed methods is to be simple for implementation and yet increase the performance drastically by exploiting the massive parallelism of GPUs. Our techniques achieve this by improving parallelism through load balancing,

memory optimization, and changing the underlying structure of data structures.

In addition to proposing techniques to improve existing algorithms and data structures, we have also designed an algorithm for medical informatics from scratch. The proposed method for single searching helps physician and care giver mine massive amount of data in very short time using commodity hardware. Design and leverage of such application in health care domain, although very valuable, has not been feasible previously due to limited processing power of computers used by physicians and care givers. Using GPUs which are inexpensive and available in most personal computers, we have shown the great benefit of leveraging many-care parallel processing in enabling physician and healthcare workers with more tools to monitor and analyze patient data.

1.4 Dissertation Contribution

In this dissertation, we study techniques to leverage many-core architectures and specifically many-core GPUs for algorithms and data structures that cannot be expressed as pure data-parallel computations. Focusing on properties and computations required in several medical monitoring and bioinformatics applications, the contributions of this dissertation are the following:

1- A decomposition and task distribution technique to improve the performance of dynamic programming (DP) algorithms on GPUs. As it will be discussed, algorithms that employ dynamic programming approach are frequently used in processing bioinformatics sequences and time series data. However, most of such algorithms have limited parallelism and hence cannot benefit from massive parallel computation power of GPUs (and other many-core architectures). We focused on non-serial polyadic family of dynamic programming algorithms that has more non-uniform access pattern than other classes. To demonstrate a generic technique that achieves more parallelism, we use an abstract formulation of non-serial polyadic DP, which was derived from RNA secondary structure prediction and matrix parenthesization. We present a decomposition algorithm that improves the overall performance of the DP algorithm on GPU by up to 228X over its counter-part single threaded DP algorithm on CPU.

2- A many-core friendly lock-free hash table structure. Hash table is one of the most frequently used

data structure, since it allows arbitrary insertion and deletion of data at constant average time per operation. However, the normal approach for resolving hash key conflicts in dynamic hash tables degrades the performance of hash table data structure in parallel processors. We introduce parallel hash table structure to diminish the contention on the shared objects and achieve significant throughput on many-core processors. Our method provides multiple instances of hash table to GPU threads, and uses a Table Assignment function to distribute operations among different hash table instances. Hence, parallel-friendly hash table reduces the conflicts that are caused by thread operations in the same hash buckets. Out method is especially beneficial in many-core architecture (comparing to benefit of applying the same method to multi-core architecture implementation). Our method also provides an opportunity to cope with data-skew and poor-fit hash functions, which impacts many-core implementations more severely. We show that using our technique we can improve the performance of hash tables on GPU and comparing to counterpart multi-processor implementation achieve up to 27X speed up.

3- A Scratchpad memory optimization technique. In modern GPU architectures, there are several levels of memory, which have the classic trade-off between speed and capacity. Since the fastest memory structure in GPUs (texture memory) has limited size, many applications have to use the Shared Memory. The data in shared memory can be shared among many parallel threads, enabling inter-thread data reuse in GPU. However, an incremental increase in the usage of shared memory per thread can result in a substantial decrease in the number of threads that can be simultaneously executed and thus significantly reducing the parallelism. Current GPUs offer limited resources (e.g. shared memory) available to each multiprocessor, and conversely, demand for availability of massive number of threads to achieve maximum performance. The limited size of fast-access shared memory available to each multiprocessor and its considerable impact on reducing the parallelism motivated us to develop a method to minimize the usage of shared on-chip memory space in modern GPU architectures. This method was specifically designed for the properties of the shared memory within the G80, GT200, and Fermi GPU architectures. Our evaluations show that using our technique can result in up to 22% more speed up in execution of

benchmark application on GPUs (comparing to CPU counterpart).

4- A parallel friendly multi-dimensional signal searching technique. Searching massive amount of time series data is a common task in mining and analyzing data collected from medical and health monitoring systems. In traditional text and structured searching systems (e.g. web search engines) pre-indexing data removes converts the linear execution order of search problem into an O(1) hash table operation. However, searching in time series data using indexing leads to extremely poor results in most contexts due to sensitivity of the technique to noise and scaling. One the other hand, considering multiple time series collected from several sources and the relationship between them, the task of searching for multi-dimensional patterns becomes computationally expensive. Considering the massive parallel capability of GPUs, we designed a new technique for searching multi-dimensional time series which is resilient to noise, scaling and delay. By efficiently using GPU resources, our technique not only outperforms highly optimized traditional techniques in execution time, but also shows higher accuracy.

1.5 Dissertation Organization

The organization of the rest of this dissertation is as follows. Chapter 2 provides background on Parallel processing concepts and introduces the medical and biomedical application platforms that are addressed in this dissertation for optimization. Chapter 3 introduces a novel approach for improving parallelism of dynamic programming algorithm on many core architectures. Chapter 4 contains the introduction of parallel lock free hash table. A memory reuse and optimization approach is introduced in Chapter 5. In Chapter 6, we describe how several techniques can be used to improve the performance of TV normalization on GPUs. Chapter 7 introduces signal searching in medical data and shows how GPUs can be utilized to extract and mine information from massive amount of data. Finally, Chapter 8 concludes this dissertation.

CHAPTER 2

Background

2.1 Parallel Computational Models

Parallel computation models range from very abstract to very concrete. Most models (e.g., PRAM, 2D Mesh) have two versions: a synchronous version and an asynchronous version. The most important property of a parallel model is whether it is synchronous or asynchronous.

2.1.1 PRAM Model

In its simplest form PRAM (Parallel Random Access Machine) model [Jaja] posits a set of p processors, with global shared memory, executing the same program in lockstep. Though there is some variability between PRAM definitions, the standard PRAM is a MIMD computer where each processor can execute its own instruction stream. Every processor can access any memory location in one time step regardless of the memory location. The main difference among PRAM model is how they deal with read or write memory contention. The PRAM model gives good guidelines when one takes a first look at the parallelization of an algorithm; the PRAM model is also useful for analysis of NP-Completeness. For

practical use, however, it has too strong assumptions, e.g., it only charges one time unit for communication between the processors. Some variations of the PRAM model were also proposed which try to alleviate these problems, but could not evaluate to a practical parallel model.

2.1.2 BSP Model

The BSP (Bulk-Synchronous Parallel) model was introduced in [Valiant] to overcome the shortcomings of the classic PRAM model and to make a bridge between abstract algorithms and realistic architectures for general purpose parallel computation. A BSP program consists of a sequence of parallel supersteps. A superstep is a combination of local computation steps and message transmissions. Each superstep is followed by a global check (barrier synchronization) to wait for all processors to finish the current superstep before proceeding to the next superstep. The BSP model is an abstract MIMD model since the processors can execute different instructions concurrently. It is loosely synchronous at the superstep level, in contrast to the tight synchrony in a SIMD model. The processor interaction mechanism in the BSP model is not specific and allows either shared variables or message passing. BSP therefore is a reasonable model for most current MIMD machines. Clusters and SMPs [Pfister] are currently two of the most popular architectural variants of MIMDs and are captured well by the BSP model.

2.1.3 SIMD vs. MIMD

Both SIMD (Single-Instruction Multiple-Data) and MIMD (Multiple-Instruction Multiple-Data) models have their particular characteristics and advantages. All processors of a SIMD are controlled by a central unit and operate in lockstep or synchronously. The SIMD model has advantages of being easily programmed, cost-effective, highly scalable, and especially good for massive fine grain parallelism [Parhami, Potter]. On the other hand, each of processors of a MIMD has its own program and executes independently at its own pace; i.e., asynchronously. The MIMD model has the advantages of high flexibility in exploiting various forms of parallelism, ease in using current high-speed of-the-shelf microprocessors, and being good for coarse-grain parallelism [Akl, Pfister].

2.2 GPU Architecture Highlights

This work NVIDIA GPUs as the hardware target for the studies in this dissertation. There has been three generations of NVIDIA GPUs G80, GT200 and Fermi. NVIDIA GPUs are effectively a set of Streaming Multiprocessors (SMs) with the ability to directly access a global device memory, which allows a more flexible programming model than previous generations of GPU. A Streaming Multiprocessor consists of a set of scalar Streaming Processors (SPs), special function units, multithreaded instruction unit, on-chip shared memory, and an L1 cache per SM multiprocessor and unified L2 cache that services all operations. Figure 2 depicts the specification and comparison of three NVIDIA GPU architectures.

The Streaming Multiprocessors employ the SIMT (Single Instruction Multiple Threads) architecture. The multiprocessor SIMT unit manages and executes threads in groups of 32 parallel threads called warps. Individual threads composing a SIMT warp start together at the same program address but are otherwise free to branch and execute independently. However, substantial performance improvements can be realized when threads in a warp seldom diverge. At every instruction issue time, the SIMT unit selects a warp that is ready to execute and issues the next instruction to the active threads of the warp. A warp executes one common instruction at a time, therefore, if threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken. Hence, full efficiency is achieved when all 32 threads of a warp are on the same execution path. SIMT architecture is similar to SIMD (Single Instruction Multiple Data) vector organizations in that a single instruction controls multiple processing elements. However, in contrast with SIMD vector machines, SIMT enables scalar thread processing; SIMT does not require the programmer to organize the data into vectors, and it permits arbitrary branching behavior for threads. SIMT is more efficient but it also uses more transistors, surface on the chip and the power consumption is higher because a more complex logic control is required. NVIDIA's Compute Unified Device Architecture (CUDA) [Nvidia] is a parallel programming model that provides a set of abstractions that are exposed to the programmer as a minimal set of extensions to C.

CUDA allows programmers to develop applications using a data-parallel programming model. Instead of compiling directly into the native machine code, CUDA compiler target a low-level virtual machine and Parallel Thread eXecution (PTX) instruction set. The PTX virtual machine is invisible to users and is delivered as part of the GPUs graphics driver. CUDA treats GPU as a coprocessor that executes data-parallel functions, so called kernel functions. The source program provided by the developer is divided into host (CPU) and kernel (GPU) code, which are then compiled by the host compiler and NVIDIAs compiler (nvcc) respectively. In the following sections, we discuss the CUDA threading model and the architectural features of the NVIDIA general-purpose GPUs that are most relevant to our work. More comprehensive descriptions are found in [Nvidia, Nokolls, Ryoo].

2.2.1 Architectural Features

To reduce the application's demand for off-chip memory bandwidth, each multiprocessor has on-chip memories that can be employed to exploit the data locality and data sharing. Each multiprocessor has a parallel data cache or shared memory for data that is either written and reused or shared among threads, a read-only constant cache and a read-only texture cache that are shared by all scalar processors. Reading from the constant cache is as fast as reading from a register as long as all threads in a half-warp read the same address, otherwise accesses will be serialized. The cost scales linearly with the number of different addresses read by all threads. Finally, for read-only data that is shared by many threads but not necessarily accessed simultaneously by all threads, the off-chip texture memory and the on-chip texture caches can be utilized to exploit 2-D data locality to reduce the memory latency.

In the Fermi architecture, a single unified memory request path was implemented for loads and stores, with an L1 cache per SM multiprocessor and unified L2 cache that services all operations (load, store and texture). The per-SM L1 cache is configurable to support both shared memory and caching of local and global memory operations.

GPU	G80	GT200	Fermi
Transistors	681 million	1.4 billion	3.0 billion
CUDA Cores	128	240	512
Double Precision Floating Point Capability	None	30 FMA ops / clock	256 FMA ops /clock
Single Precision Floating Point Capability	128 MAD ops/clock	240 MAD ops / clock	512 FMA ops /clock
Warp schedulers (per SM)	1	1	2
Special Function Units (SFUs) / SM	2	2	4
Shared Memory (per SM)	16 KB	16 KB	Configurable 48 KB or 16 KB
L1 Cache (per SM)	None	None	Configurable 16 KB or 48 KB
L2 Cache (per SM)	None	None	768 KB
ECC Memory Support	No	No	Yes
Concurrent Kernels	No	No	Up to 16
Load/Store Address Width	32-bit	32-bit	64-bit

Figure 2. Specification and comparison of three NVIDIA GPU architectures

Prior to Fermi architecture, the off-chip memory or global memory space is not cached, so it is all the more important to follow the right access pattern to get maximum memory bandwidth. Bandwidth to off-chip memory is quite high, but can be saturated if many threads request access within a short period of time. The GPU has a hardware feature called memory coalescing to exploit the spatial locality of memory accesses among threads. In devices with compute capability of lower than 1.2, when the addresses of the memory accesses of the multiple threads in a thread group are consecutive, these memory accesses are grouped into one. However, for devices with compute capability of higher, coalescing is made more flexible, and is achieved for any pattern of addresses requested by the half-warp, including patterns where multiple threads access the same address. This is in contrast with devices of lower compute capabilities where threads need to access words in sequence.

2.2.2 Threading Model

The threads on each multiprocessor are organized into thread blocks (TB). The thread blocks are dynamically scheduled on the multiprocessors. Each thread block is assigned to a single multiprocessor for the duration of its execution. Threads within a thread block share the computation resources such as

registers and shared memory on a multiprocessor. A thread block is divided into multiple schedule units. Given a kernel program, the occupancy of the GPU is the ratio of active schedule units to the maximum number of schedule units supported on each multiprocessor. A higher occupancy indicates that the computation resources of the GPU are better utilized.

The GPU thread is different from the CPU thread. It has low context-switch and low creation time as compared to CPUs. The batch of threads that executes the kernel is organized as a grid of thread blocks. Each kernel creates a single grid that consists of many thread blocks. Each thread-block (TB) is at most a three dimensional array of threads and has unique coordinates in the grid. Number of thread blocks that can be processed simultaneously on a multiprocessor depends on how many registers per thread and how much shared memory per thread block are required for a given kernel since the multiprocessors registers and shared memory are split among all the threads of the batch of blocks. If there are not enough registers or shared memory available per multiprocessor to process at least one block, the kernel will fail to launch. Consequently, the more resources consumed by each thread, fewer threads can be active simultaneously which results in tremendous performance loss. Therefore, there is often a trade-off between the efficiency of individual threads and thread-level parallelism. In other words, although by using more resources in each thread we may increase the performance of each thread individually, this eventually reduces the degree of parallelism, which results in reduction of the multiprocessor's occupancy [Muyan].

2.3 Programming Dwarfs

The conventional way to guide and evaluate architecture innovation is to study a benchmark suite based on existing programs, such as EEMBC (Embedded Microprocessor Benchmark Consortium) or SPEC (Standard Performance Evaluation Corporation) or SPLASH (Stanford Parallel Applications for Shared Memory). One of the biggest obstacles to innovation in parallel computing is that it is currently unclear how to express a parallel computation best. Hence, based on the report from Berkeley [Asanovic], it seems unwise to let a set of existing source code drive an investigation into parallel computing. There is a need to find a higher level of abstraction for reasoning about parallel application requirements.

The approach described in [Asanovic], is to define a number of "dwarfs", which each capture a pattern of computation and communication common to a class of important applications. The dwarfs are specified at a high level of abstraction to allow reasoning about their behavior across a broad range of applications. Programs that are members of a particular class can be implemented differently and the underlying numerical methods may change over time, but the claim is that the underlying patterns have persisted through generations of changes and will remain important into the future. The following list is the 13 Dwarfs, which consist of the Seven Dwarfs first introduced by Phil Colella [Colella] and the six additional dwarfs that was added in [Asanovic]:

1- Dense Linear Algebra

2- Sparse Linear Algebra

3- Spectral Methods

4- N-Body Methods

5- Structured Grid

6- Unstructured Grid

7- Monte Carlo

8- Combinational Logic

9- Graph Traversal

10- Dynamic Programming

11- Backtrack, Branch and Bound

12- Construct Graphical Models

13- Finite State Machine

In any case, the point of the 13 Dwarfs is to identify the kernels that are the core computation and communication for important applications in the upcoming decade, independent of the amount of parallelism.

2.4 Medical Applications and Platforms

In this section, we briefly introduce the main applications and platforms used as source of algorithms or data throughout or research.

2.4.1 Diffusion Tensor Imaging Denoising

During the last two decades, a new magnetic resonance modality called diffusion tensor imaging (DTI) has been extensively studied [Basser]. Using DTI, it is possible to noninvasively study anatomical structures such as the nerve fibers in the brain. From the developments in DTI, a need for robust regularization methods for matrix-valued images has emerged. One of the state of the art techniques for denoising such matrix valued images in the work presented in [Christiansen]. Figure 3 shows the sample DTI image from human brain, and the result of applying this denoising algorithm on it.

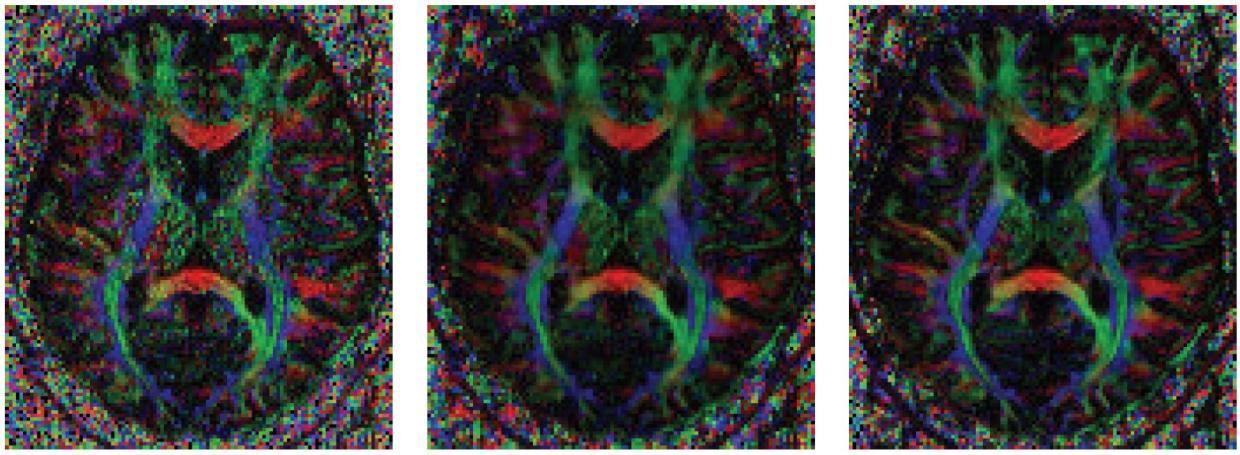

Figure 3.The noisy acquisition (left) 4-averages denoised (middle) and 18-averages denoised image (right)

2.4.2 Medical Shoe

The Medical Shoe system has been developed with the consideration of several health oriented concerns. The main component of the medical shoe systems, which distinguished it from a normal shoe is its sensor enable insole. The insole consists of a gird of 99 tiny pressure sensors, which cover the whole are of the insole. For sensor placement, we use the Pedar plantar pressure mapping system [Novel], which distributes the sensors across the foot. The Medical shoe system is equipped with wireless communication

modules and is able to send its data in real time to the base station. The applications of the medical shoe ranges from ulcer prevention in diabetes patient, fall detection in elderly and disabled, activity monitoring for general population, and professional athlete activity tuning [Dabiri, Noshadi]. Figure 4 depicts the general structure of the medical shoe system.

Figure 4. Medical Shoe and the corresponding Pedar sensor mapping

2.4.3 Personal Activity Monitor

The Personal Activity Monitor (PAM) system is a portable motion sensing device which is designed to continuously monitor motions of the subjects. PAM consists of three accelerometers, a flash memory and a USB computer interface [Pam]. To conserve energy and hence having long battery life time, PAM does not have any wireless communication system and instead, it saves all the sensor data in a flash memory. Its data can be uploaded occasionally to a base station. PAM is mainly used to monitor and record activities of subjects. An example use of such records is to measure physical activity and hence energy expenditure [Vahdatpour11]. Figure 5 represents a PAM device.

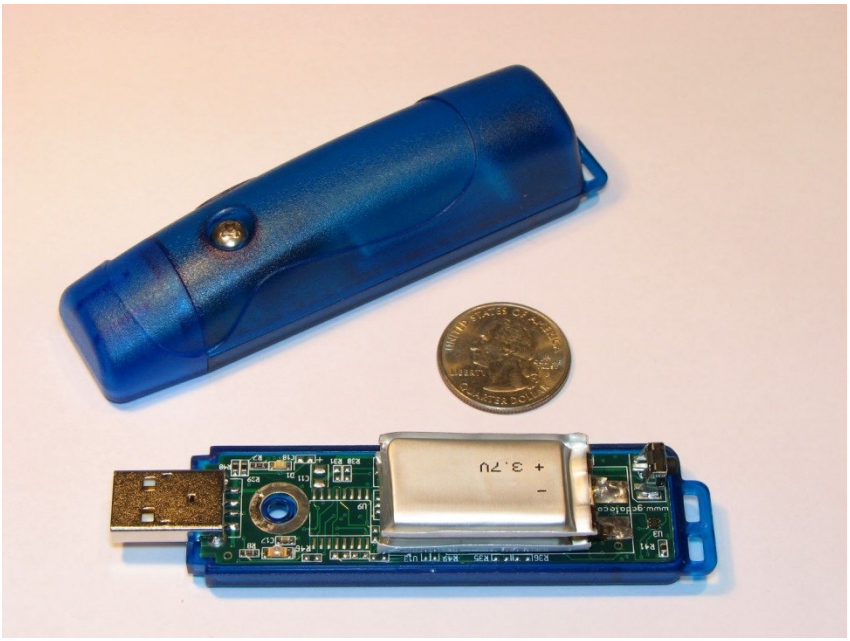

Figure 5. Personal Activity Monitor (PAM) System.

CHAPTER 3

Dynamic Programming on Data-Parallel Many-core Architectures

3.1 Overview

Modern GPUs offer massive parallelism through the use of hundreds of cores and high memory bandwidth. Achieving the maximum performance from GPUs requires exposing large amounts of fine-grain parallelism and structuring computations to regulate execution paths and memory access patterns.

To port real-world applications to GPUs, structured analysis that identifies and eliminates the bottlenecks is required. Based on [Asanovic], parallelizable applications can be categorized into 13 representative classes called dwarfs (also known as motifs), where each dwarf captures a pattern of computation and communication common to a class of important applications. The idea is that the dwarfs are specified at a high level of abstraction so that programs that are member of a particular class, while potentially being implemented differently, will still exhibit the same underlying patterns. Originally, dwarfs were used to replace the traditional benchmarks to design and evaluate parallel programming models and architectures. But looking from the applications' perspective, the idea of dwarfs can be used to develop structured

methodologies (e.g., data layout transformation, adapting or redesigning the resource allocation, load balancing) for the persisting patterns of dwarfs. In this chapter, we focus on the dynamic programming dwarf.

Dynamic programming is a method for efficiently solving a broad range of search and optimization problems that exhibit the characteristics of overlapping sub-problems. This technique is used in many application domains such as bioinformatics, VLSI design, scheduling, and inventory management. As a result, techniques for efficiently solving large-scale DP problems are often critical to the performance of many applications. In order to find efficient parallel algorithms for dynamic programming, algorithms that exhibit the same computation and communication patterns are classified in the same class [Grama]. The following criteria are used to classify dynamic programming: if the sub-problem located on all levels depends only on the results from the immediately preceding levels, it is called serial; otherwise, it is called nonserial. Typically there is a recursive equation called a functional equation, which represents the solution to the optimization problem. If a functional equation contains a single recursive term, the DP formulation is called nomadic; otherwise it is called polyadic. As such, there are four classes defined based on these classification criteria: serial monadic (e.g., single source shortest path, 0/1 knapsack problem), serial polyadic (e.g., Floyd all pairs shortest paths algorithm), nonserial monadic (e.g., longest common subsequence problem, Smith-Waterman algorithm) and nonserial polyadic (e.g., optimal matrix parenthesization problem, RNA secondary structure prediction (Zuker algorithm)).

In general, dynamic programming has limited parallelism. However, due to its importance, parallel dynamic programming has become a classic problem and it is relatively well-studied on multi-core architectures. However, optimizing dynamic programming for many-core architectures is different than multi-core architectures. A many-core implementation must distribute work among hundreds or thousands of threads. Many-core architectures are designed to hide the latency of memory accesses by means of multi-threading instead of using caches. This difference implies that parallel decomposition techniques that are used for multi-core architectures may not suffice for many-core architectures, and may not

succeed at achieving the needed level of parallel granularity.

In this chapter, we address the challenge of exploiting fine-grain parallelism of nonserial polyadic dynamic programming. We use an abstract formulation of non-serial polyadic DP, which was derived from RNA secondary structure prediction and matrix parenthesization. We present a decomposition algorithm that achieves the best overall performance with this type of workload on many-core architectures. Our optimization reasoning is based on performance data obtained via profiling and quantitative analysis. We compare a divide-and-conquer approach previously used on multi-core architectures with an iterative bottom-up approach. The divide-and-conquer approach was popular for multi-core implementations because it often has better cache performance; however, as shown, the divide-and-conquer approach results in very poor load balancing. A dynamic programming workload is not pure SIMD (single instruction, multiple data); therefore, the decomposition algorithm is an important factor in achieving optimal performance. This is in contrast to multi-core implementations that only need to create tens of executions threads. Thus, workload imbalance was not a critical concern as much as data locality and cache performance.

The rest of the chapter is organized as follows. After discussing related work in Section 3.2 we present the abstract formulation of non-serial polyadic DP in Section 3.3. In Section 3.4, we present the implementation on GPU. Section 3.5, describes our decomposition algorithm. Section 3.6 includes performance analysis and finally Section 3.7 concludes the chapter.

3.2 Related Work

In [Steffen], the authors present a framework to encode common bioinformatics problems, like RNA folding and pairwise sequence alignment, in C; and subsequently implemented a parallel GPU CUDA backend for their compiler, which launches a large number of threads. They report speedups ranging from 9.9-25.8x for the RNA folding problem using CUDA, which is significantly less than the speedups achieved in this study. The authors claim that all dynamic programming problems have similar data dependencies and use this idea to develop a generic parallelization that does not achieve the best

optimization for different classes of dynamic programming. [Xiao] proposes a fine-grain parallelization of the Smith-Waterman problem on NVIDIA GPU and Cell Broadband Engine. In their CUDA implementation, they use a set of techniques such as matrix realignment, coalesced memory access, tiling, and GPU synchronization rather than CPU synchronization. They report that tiling fails to speed up the execution, while GPU synchronization achieves a better performance than CPU synchronization, reducing the synchronization from 55.32% to 36.17%. [Che] gives a characterization of the Needleman-Wunsch (NW) problem workload on GPU compared to other applications from the 13 dwarfs. An interesting takeaway was that the persistent-thread-block technique results in poor results for NW problem, although global synchronization and many incurred kernel calls are avoided; a speedup of 8x is reported in this study.

There is no previous work that has carefully studied nonserial polyadic dynamic programming workload on GPUs. Throughout this study, we compare our approach with the multi-core implementations of RNA secondary structure prediction presented in [Tan07, Tan06], which entailed a 30x speedup. They presented a divide-and-conquer approach and a parallel pipeline for decomposing computations and improving cache performance in multi-core architectures. However, as discussed previously, parallel decomposition techniques that are used for multi-core architectures may not suffice for many-core architectures and may not succeed to expose the right level of parallel granularity. Our goal is to present these differences and our proposed approaches for many-core architectures in this chapter.

3.3 Problem Formulation

We use an abstract DP formulation that is based on a DP formulation for RNA secondary structure prediction and optimal matrix parenthesization from the nonserial polyadic family, previously used in [Tan07]. In most applications, the computation in the formulation mainly involves floating point operations. The abstract formulation is as follows, where *a(i)* is an initial value:

$$m[i,j]=\begin{cases}\min_{i\le k<j}\{m[i,j],m[i,k]+m[k+1,j]\} & 0\le i<j<n \\ a(i) & i=j\end{cases} \quad (1)$$

The data dependence in this DP exists between non-consecutive stages, which make the data access pattern non-uniform. The non-uniform data access pattern makes this problem harder to optimize for parallelization. Therefore, we also use the data transformation that was used in [Tan07] to eliminate cross block references, improving data locality (Figure 6). Assume *(i, j)* is the original coordinate in the original domain $D = \{(i, j) \mid 0 \leq i \leq j < n\}$, where n = |D| is the original problem size, *(i', j')* is the new coordinate in the transformed domain $D' = \{(i', j') \mid 0 \leq i' \leq j' < n'\}$, where $n' = n + 1 = |D'|$ is the new problem size. The iteration domain transformation is defined as follows:

$$(i', j') = f(i, j): i' = i, j' = j + 1$$

Therefore, Equation 1 is rewritten as the new Equation 2 in the transformed domain, where *a(i)* is the known initial value. The values on the new diagonal can be any value. In the new domain, the values on the new diagonal do not contribute to the computation.

$$m[i', j'] = \begin{cases} \min_{i'+1 \leq k' < j'} \{m[i', j'], m[i', k'] + m[k', j']\} & 0 \leq i' < j' < n' \\ a(i) & i' = j' \end{cases} \quad (2)$$

3.3.1 Parallelism

To exploit fine-grain parallelism we can use the blocking technique to decompose the computations. By using the transformed DP formulation, the cross block reference is eliminated in the blocked matrix.

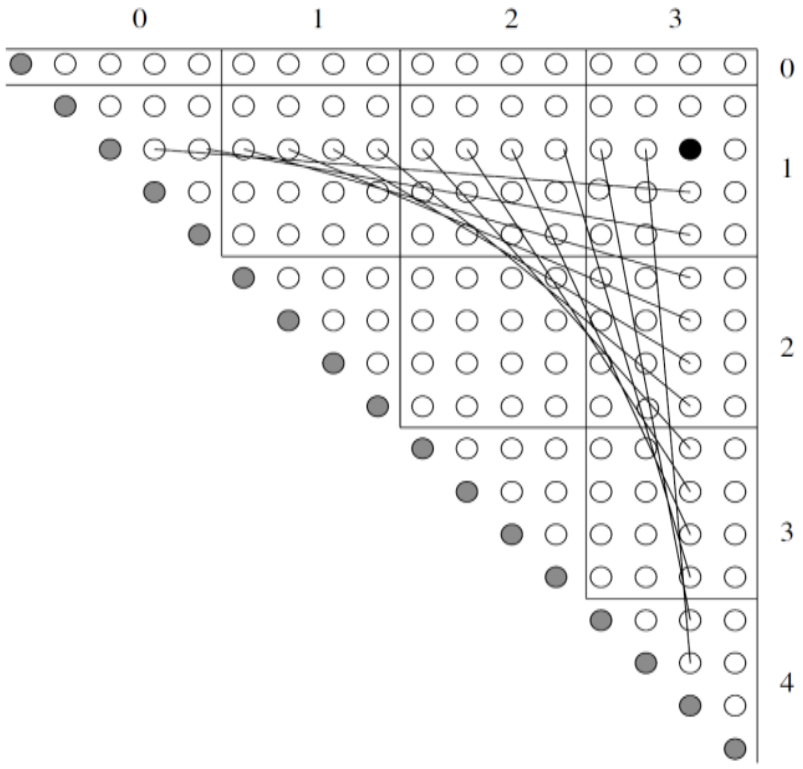

Figure 6. The blocked transformed DP matrix

The blocked algorithm can be observed as comprising many matrix block operations. Let matrices $A = (a_{ij})_{sxs}$, $B = (b_{ij})_{sxs}$, $C = (c_{ij})_{sxs}$, the tensor operations $\otimes$ and $\oplus$ for the blocked matrix is defined as follows:

Definition 1

$$\forall a_{ij} \in A, b_{ij} \in B, c_{ij} \in C, 1 \le i,\ j \le n, \text{ if } c_{ij} = \min_{k=1}^{n}\{c_{ij}, a_{ij} + b_{ij}\}, \text{ then } C = A \otimes B$$

Definition 2

$$\forall a_{ij} \in A, b_{ij} \in B, c_{ij} \in C, 1 \le i,\ j \le n, \text{ if } c_{ij} = \min\{a_{ij}, b_{ij}\}, \text{ then } C = A \oplus B$$

The formulation to compute any matrix sub-blocks (In the rest of this chapter, matrix sub-blocks are referred as blocks) A(i, j) is as follows:

$$\begin{aligned} A(i,j) &= \oplus_{k=i}^{j} (A(i,k) \otimes A(k,j)) \\ &= (\oplus_{k=i+1}^{j-1} (A(i,k) \otimes A(k,j))) \oplus (A(i,i) \otimes A(i,j)) \oplus (A(i,j) \otimes A(j,j)) \end{aligned}$$

In this equation, the calculation is divided into two parts. The first part uses the rectangular blocks in

the same row and column. We refer to this part of the computation as the rectangular computation:

$$(\oplus_{k=i+1}^{j-1}(A(i,k)\otimes A(k,j)) \qquad (4)$$

The second part of the computation depends on triangular blocks and itself. We refer to this computation as the triangular computation:

$$(A(i,i)\otimes A(i,j))\oplus(A(i,j)\otimes A(j,j)) \qquad (5)$$

These computations contribute to the partial result of the block *A(i, j)*. For each k in Equation 4 above, all the blocks *A(i, j)* in the same diagonal can be computed in parallel. In Equation 5, the two $\otimes$ operations depend on the result of *A(i, j)* from Equation 4; this computation can be performed in parallel for all the blocks *A(i, j)* in the same diagonal.

For rectangular computation, each element in one block is mapped to one thread. According to Definition 1, there is no dependency between elements in a block for $\otimes$ operation, so all threads can be computed in parallel, similar to a dense matrix multiplication.

3.4 Non-Serial Polyadic DP on GPU

There are two diverse forms of computation in this dynamic programming formulation, the rectangular computation and the triangular computation, which makes the load balancing of this workload interesting on many-core architectures.

The rectangular computation workload and data access pattern is very similar to a dense matrix multiplication and the same optimizations can be applied. The triangular computation, on the other hand, has very limited parallelism because of its data dependency between two consecutive entries: parallelism can only be exploited along the diagonal. However, triangular computations can be broken down into rectangular computations and triangular computations of a smaller size to expose more parallelism. Therefore, a good decomposition algorithm is to reduce the proportion of triangular computations.

The rectangular computation of each block is dependent on blocks in the same row and column.

Therefore, the execution of blocks proceeds along the diagonal. Because the computations for each block in the same diagonal do not have any interdependencies, they can be computed in parallel. In the multicore implementation [Tan07], on the order of 10s of threads are used (i.e., 16, 32, and 64). Therefore, in contrast to a many-core implementation, it is not possible to take advantage of the parallelism that exists for the computation of blocks in the same diagonal. However, a single GPU kernel can distribute work among thousands or tens of thousands of threads, which can exploit parallelism at this level.

For each *k* in Equation 4: all the blocks *A(i, j)* in the same diagonal are computed in a single compute_rectangular kernel execution in parallel. For Equation 5, we merge the two tensor operations in a single kernel and all the blocks *A(i, j)* in the same diagonal are computed in a single compute_triangular kernel execution in parallel.

We implement the rectangular computation as a CUDA kernel, compute_rectangular, in which we use tilling technique and loop unrolling to achieve maximum parallelism. For the triangular computation kernel, we explored different design options to find the solution resulting in the best performance, considering the characteristics of this workload. For the compute_triangular kernel, we achieve the best overall performance when we map each block *A(i, j)* to a single thread-block in CUDA with no further decomposition. Therefore, the number of thread-blocks is equal to the number of blocks *A(i, j)* in the current diagonal in the DP matrix (Figure 9). Two options were considered:

1. We use shared memory to merge and store two triangular blocks with the rectangular block. Therefore, we can take advantage of data reuse in the calculation of each point in the block along the diagonal via the fast on-chip memory.

2. We read directly from global memory, and we do not merge the two triangular blocks with the rectangular blocks. Therefore, in the CUDA kernel, we transform the code in Figure 8 to compute the result by discontinuously reading rows and columns from the three separate blocks in the matrix.

For the first option, we have a limitation of 16K shared memory per thread-block. Therefore, the size of

each thread-block cannot be more than 16×16. This limits the size of the block size in the DP matrix to 16 as well, which results in a poor load balancing on GPU. In the overall workload, the second option thus achieves the best result.

```
rectangular_computation (A, B, C, size)
{
  for (i = 0; i < size; i++)
   for (j = 0; j < size; j++)
     for (k = 0; k < size; k++)
      C[i][j] = min(C[i][j], A[i][k]+B[k][j]);
}
```

Figure 7. Pseudo code for rectangular computation

```
triangular_computation (A, B, C, size)
{
  M = merge (A, B, C);
  for (index_j = 1; index_ j < size; index_j++)
   for (i = 0; (i+index_j) < size; i++)
   {
     j = index_j + i;
     for (k = i+1; k < j; k++)
      M[i][j] = min(M[i][j], M[i][k]+M[k][j]);
   }
}
```

Figure 8. Pseudo code for triangular computation

As we explored the different design options through profiling the workload, we concluded that the decomposition algorithm plays a more important role in achieving the best performance versus micro-optimization of the kernels.

3.5 Decomposition Algorithm

The DP matrix can be filled in two fashions. First is the divide-and-conquer approach, which is also used in the multi-core implementation of the RNA secondary structure prediction in [Tan07]. We can use this divide-and-conquer approach (Figure 9) to obtain the decomposition of computations described in Section 3.3 to exploit higher fine parallelism. Another approach to achieve the same decomposition of the computations is a bottom-up iterative strategy. In the bottom-up iterative approach (Figure 11), the DP

matrix is partitioned into fixed-size blocks and all the blocks *A(i, j)* in the same diagonal are computed in each iteration based on equation 3.

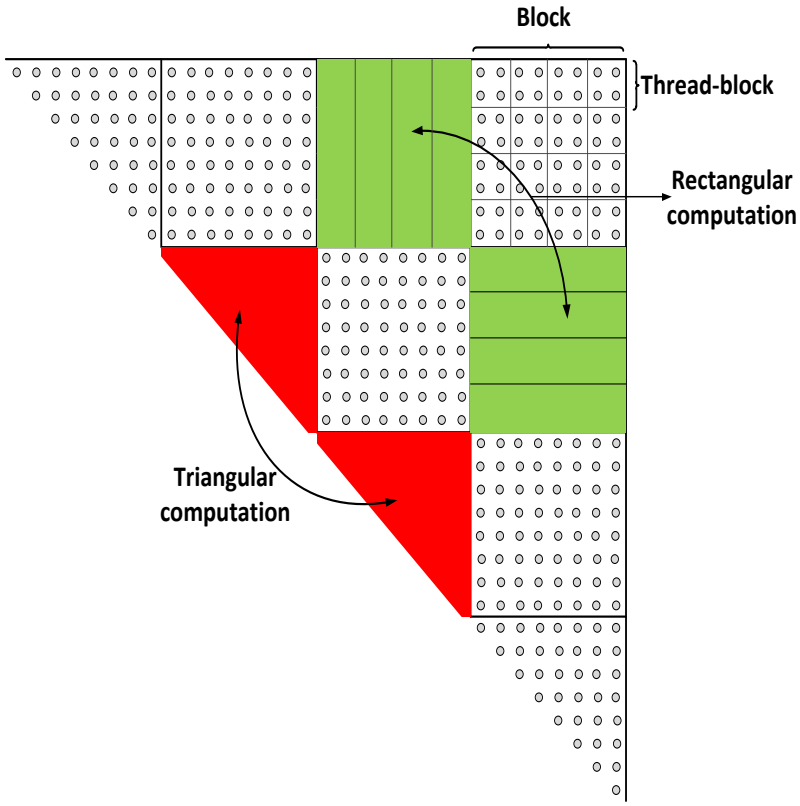

Figure 9. Blocks, thread blocks, triangular and rectangular computation for a matrix.

The best decomposition algorithm is the one that reduces the proportion of triangular computation as there is very limited parallelism in that kernel. With regards to the rectangular computations, as we increase the problem size the kernel execution time does not increase linearly as matrix multiplication is $O(n^3)$, shown in Figure 10. This is also the same for the dense matrix multiplication kernel, which has the same characteristics as the compute_rectangular kernel. As a consequence, a good decomposition algorithm should result in more calls to the compute_rectangular kernel with smaller size.

Table 1 compares the divide-and-conquer and the bottom-up iterative approaches in the number of calls to the compute_rectangular and compute_triangular kernels. As demonstrated, by increasing the problem size, the bottom-up approach results in significantly more calls to the compute_rectangular kernel. For this comparison, we launch a kernel for each block in the current diagonal-strip. In our final

implementation, we launch a single kernel for all the blocks that are in the same diagonal-strip. As depicted in Figure 9, the divide-and-conquer approach performs the rectangular computations with problem size n and divides the triangular computations into rectangular and triangular computations with problem size *n/2*. This pattern of execution results in less calls to the compute_rectangular kernel with variable sizes. On the other hand, the bottom-up approach results in more calls to the compute_rectuangular with fixed size (and smaller than those in the divide-and-conquer approach in overall). More detail on the performance evaluation of the two approaches is explained in Section 3.6.

Table 1. Iterative bottom-up approach vs. divide-and-conquer approach

	Iterative				Divide-and-conquer			
	triangular		rectangular		triangular		rectangular	
Problem size (n)	Number of calls	Time (us)	Number of calls	Time (us)	Number of calls	Time (us)	Number of calls	Time (us)
1024	6	748,822	4	2,177	6	749,021	4	2,181
2048	28	3,616,350	56	30,690	36	4,650,060	28	29,906
4096	120	15,572,900	560	306,875	216	27,870,300	172	311,124
8192	496	64,884,000	4960	2,727,580	1296	171,049,000	1036	2,923,350

Block size can also make a big difference in overall running time. Although the key to performance on this platform is using massive multithreading to utilize the large number of cores and hide global memory latency, partitioning the matrix to larger sub-matrices does not result in reduced overall running time. This is elaborated in more detail in Section 3.6 where we show the performance numbers varied with different block sizes. The reason for this is related to the fact that is depicted in Figure 10: the larger we choose the block size the larger is the load on the GPU proportional to the load on GPU with smaller block size.

```
rectangular_computation(A, B, C, partition_size)
{
    triangular_computation(A01, A00, A11, partition_size);
    triangular_computation(C10, A11, B00, partition_size);
    triangular_computation(B01, B00, B11, partition_size);

    compute_rectangular<<t, b>>(C00, A01, C10, partition_size);
    triangular_computation(C00, A00, B00, partition_size);

    compute_rectangular<<t, b>>(C11, C10, B01, partition_size);
    triangular_computation(C11, A11, B11, partition_size);

    compute_rectangular<<t, b>>(C01, A01, C11, partition_size);
    compute_rectangular<<t, b>>(C01, C00, B01, partition_size);
    triangular_computation(C01, A00, B11, partition_size);
}

triangular_computation(A, B, C, partition_size)
{
     if (partition_size > M)
          rectangular_computation(A, B, C, partition_size/2);
     else
          compute_triangular<<t,b>>(A, B, C, partition_size);
}
```

Figure 10. Pseudo code for divide-and-conquer approach

```
for (index_j = 1; index_ j < num_partition; index_j++)
{
    for (i = 0; (i+index_j) < num_partition; i++)
    {
        j = index_j + i;
        for (k = i+1; k < j; k++)
            compute_rectangular<<t, b>>(A_ik, B_kj, C_ij, parrtition_size);
    }
    compute_triangular<<t,b>>(A_ik, B_kj, C_ij, partition_size);
}
```

Figure 11. Pseudo code for iterative bottom-up approach

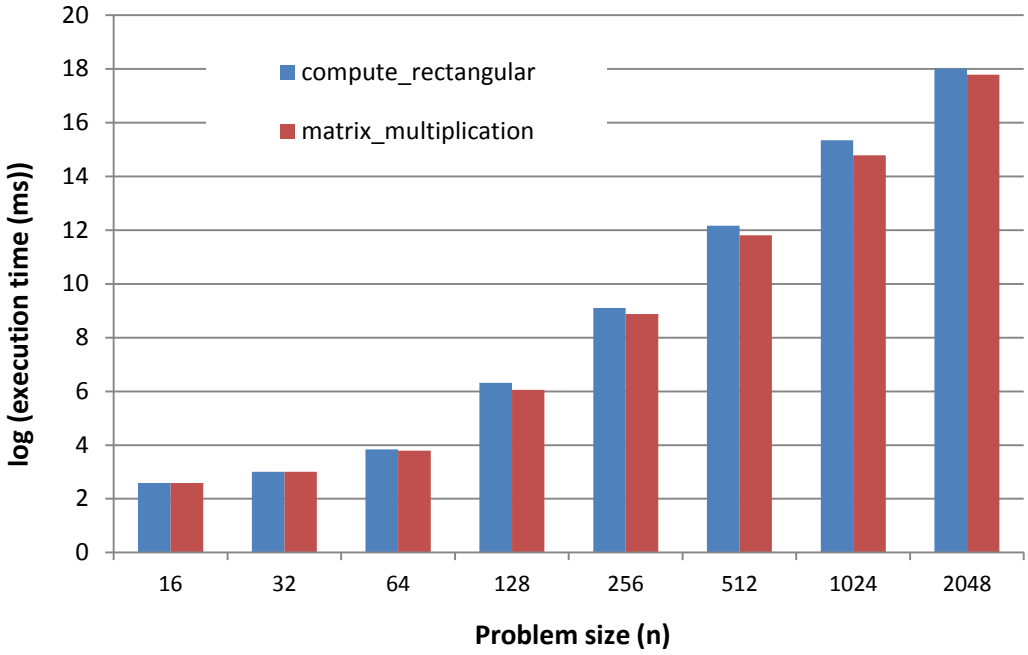

Figure 12. Execution time for the compute_rectangular and dense matrix multiplication kernels

As the running time does not increase linearly as we increase the kernel size, significantly better result is achieved with smaller block sizes compared to larger block sizes.

3.6 Performance Analysis

We evaluate the performance of the parallel non-serial polyadic dynamic programming on three different NVIDIA GPUs: Tesla C1060, Quadro FX 5600, and GeForce 8800 GT with CUDA 3.2 paired with an Intel Core i7 965 CPU. The specifications for these GPUs are summarized in Table 2. The three GPUs have a different number of parallel processor cores and clock rates. We use these differences to evaluate how the workload scales by increasing the number of available processor cores. We report performance in GFlops, determined by dividing the required arithmetic operations by the average execution time. The execution time is obtained by taking the minimum time over multiple runs. The time for transferring the data to GPU is considered in the execution time. We compare our GPU implementations with single-threaded CPU implementations in order to contrast our results with the previously reported results on multi-core architectures, which were also compared with single-threaded implementations.

Table 2. GPU specifications

GPU	Clock rate (GHz)	Parallel processor cores	Peak bandwidth (GB/s)	Driver
Tesla C1060	1.3	240	102	3.2
GeForce 8800 GT	1.5	112	57.6	3.2
Quadro FX 5600	1.35	128	76.8	3.2

3.6.1 Decomposition

In this section, using an exhaustive set of experiments we show that the iterative bottom-up approach results in better load balancing compared to the divide-and-conquer approach previously used in multi-core architectures. As mentioned earlier, a good decomposition algorithm is to reduce the proportion of triangular computations as it exposes very limited parallelism and to increase the proportion of rectangular computations.

We conducted experiments to compare the divide-and-conquer decomposition technique with the iterative bottom-up. Table 3 shows that the iterative bottom-up approach achieves significantly better results compared to divide-and-conquer. For the rectangular computations, the divide-and-conquer approach schedules less blocks with larger size, as compared to the iterative bottom-up approach. This strategy does not achieve the best results because, as depicted in Figure 12, as we increase the problem size the compute_rectangular kernel execution time does not increase linearly. The reason is that we launch $O(n^2)$ execution threads on each kernel, but the number of processor cores is constant. Therefore, execution is serialized. Consequently, as we increase the problem size n, the execution time does not increase linearly. Hence, scheduling more blocks with smaller size for rectangular computation achieves best results.

Table 3. Speedup of iterative bottom-up approach over divide-and-conquer approach

Size	1024	2048	4096	8192
Speedup	1.01X	1.49X	2.34X	3.92X

Modern GPUs are optimized for very large workloads. It is normally recommended that thousands or tens of thousands of threads needs to be executed simultaneously to hide memory latencies incurred on

the GPU. Therefore, the question is what should be the granularity of a partitioning that gives the best overall performance and whether the result is comparable to more flexible multi-core architectures with more flexible execution model?

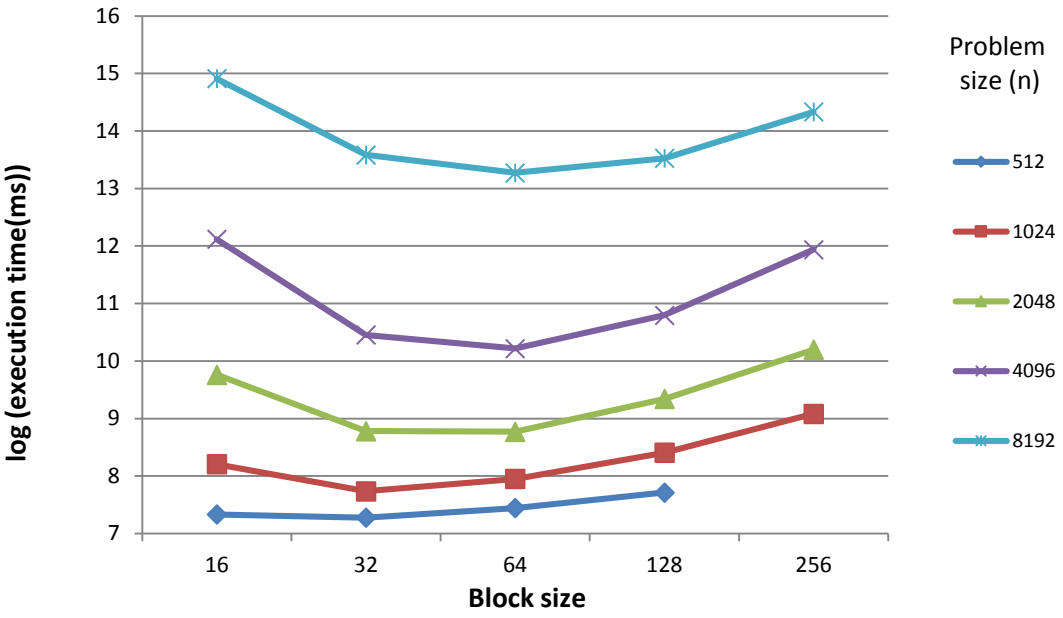

Figure 13. Execution time on TeslaC1060

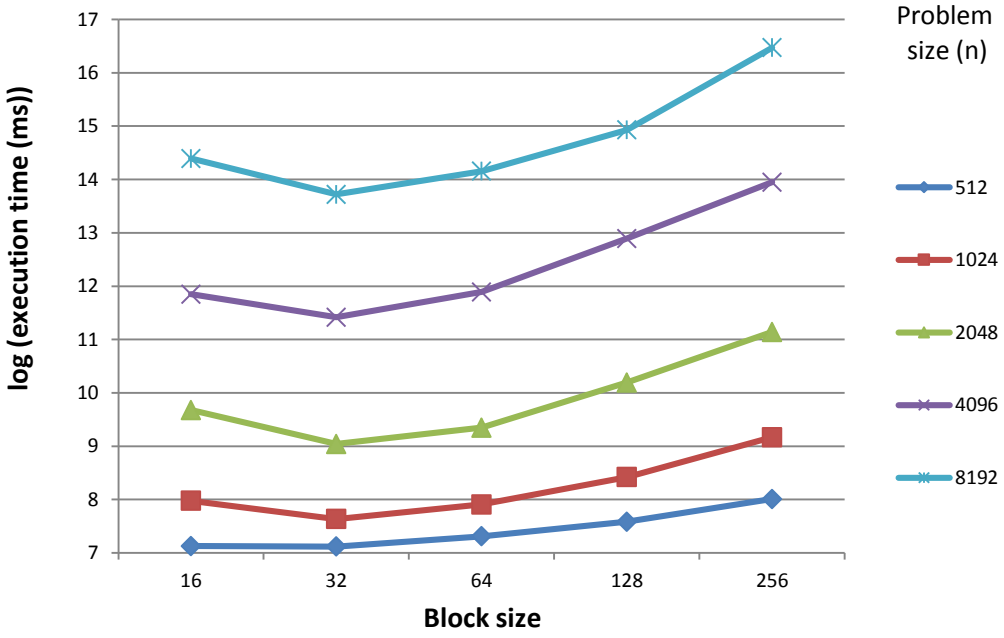

Figure 14. Execution time on GeForce 8800 GT

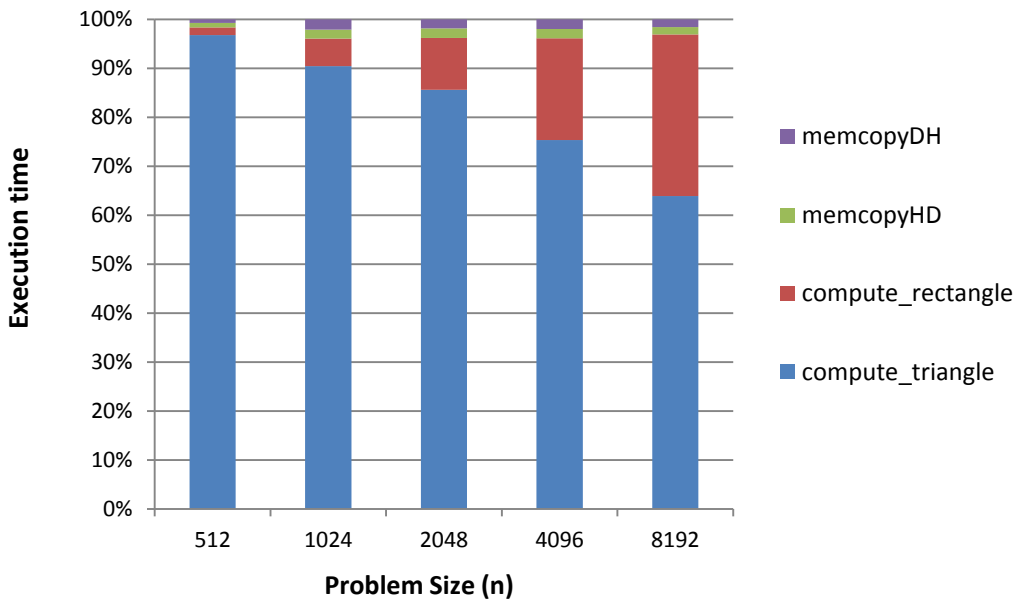

Figure 15. Contribution of different execution phases and kernels to overall execution time on Quadro FX 5600

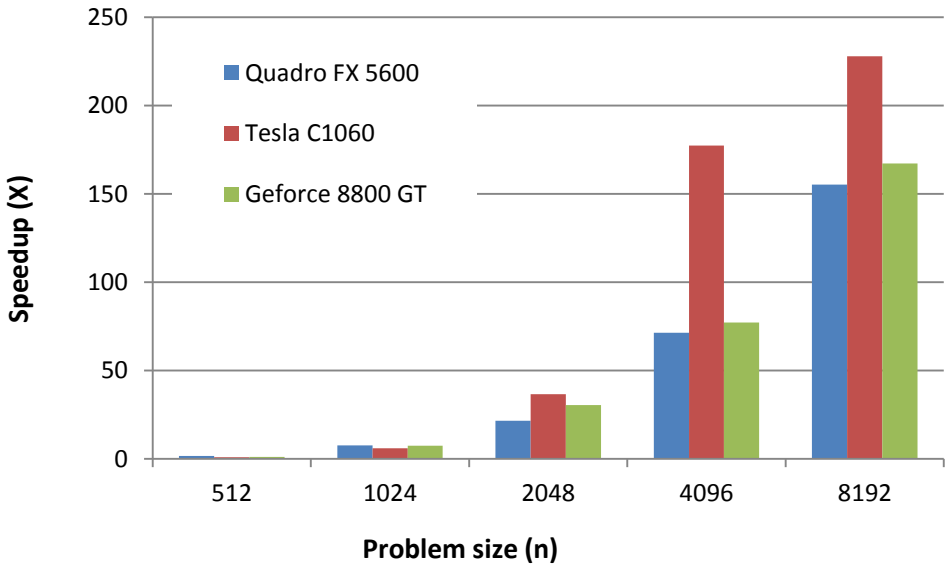

Figure 16. Speedup over single-threaded CPU implementation

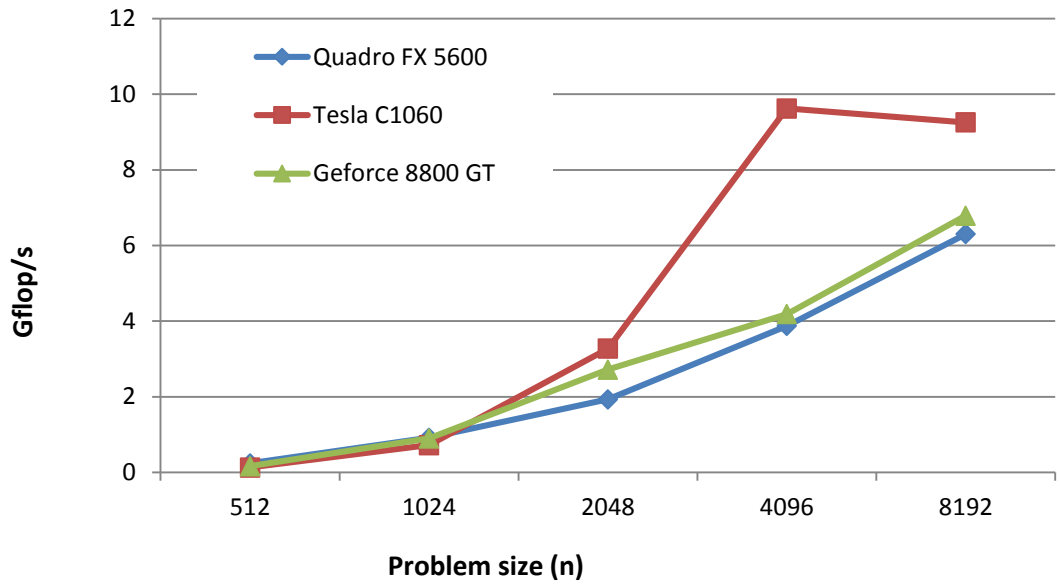

Figure 17. Performance of the three GPUs

Assume we categorize the block sizes from 16-32 to be small; block sizes from 64-128 as medium; and 256 and higher to be large. Partitioning the DP matrix into large block sizes results in fewer and larger loads on GPU. In turn, this decreases the global synchronization and kernel launch overhead incurred between diagonals. On the other hand, partitioning the DP matrix to medium and small sizes results in smaller loads on the GPU, with higher global synchronization and increased kernel launch overhead. Nonetheless, as illustrated in Figure 14, using small and medium block sizes can significantly reduce the execution time of each kernel, which benefits overall performance. Note that the block size is not equal to the size of the load on each kernel, but equal to the number of blocks in the current diagonal times the block size. Figure 13 and Figure 14 show the result of our experiments on the Tesla C1060 and the GeForce 8800 GT using iterative bottom-up decomposition. On the Tesla, for problem size $n < 512$ the block size of 32 gives best results; and for problem sizes $n > 512$, the block size of 64 gives the best results. On the GeForce, a block size of 32 always gives the best result. The Quadro FX 5600 exhibits the same behavior as the Tesla. Figure 15 also shows that as we increase the problem size, the proportion of rectangular computation is increased almost linearly.

Figure 16 presents the speedups achieved on the three GPUs as compared to a single-threaded implementation on an Intel Core i7 CPU. We achieved up to 228x speedup on the Tesla. The multi-core implementation achieves a speedup of up to 30x using 64 threads by implementing a cache-oblivious parallel fine-grain algorithm on Cyclops64 [Tan07]. Figure 17 demonstrates the performance in GFLOP/s. For the Quadro and the GeForce GPUs, performance continues to increase as we grow the problem size. However, on the Tesla, the performance remains almost constant after reaching a size of n = 4096. This workload is very interesting for data-parallel many-core architectures because even after the decomposition of operations, the workload is still not purely SIMD.

3.6.2 Scaling

We examined the scaling properties of our algorithm with respect to the number of processor cores. Figure 18 shows the execution time of our algorithm running on 112, 128, and 240 processors. Again, the iterative bottom-up approach for decomposition was used. For $n < 512$, a block size of 32 was used; and for $n > 512$, a block size of 64 was employed for optimal decomposition. As shown in this experiment, for $n < 1024$, the execution time on the Tesla GPU is greater than the execution time on the Quadro GPU. However as the problem size increases, the execution time on the Tesla GPU is between 2.4 to 1.4 times less than the Quadro and GeForce GPUs. The Tesla has 1.8 and 2.1 times more processor cores than the Quadro and GeForce GPUs, respectively. This suggests that the algorithm has potential scalability on many-core architectures, as increasing the number of processing cores results in proportionally higher performance.

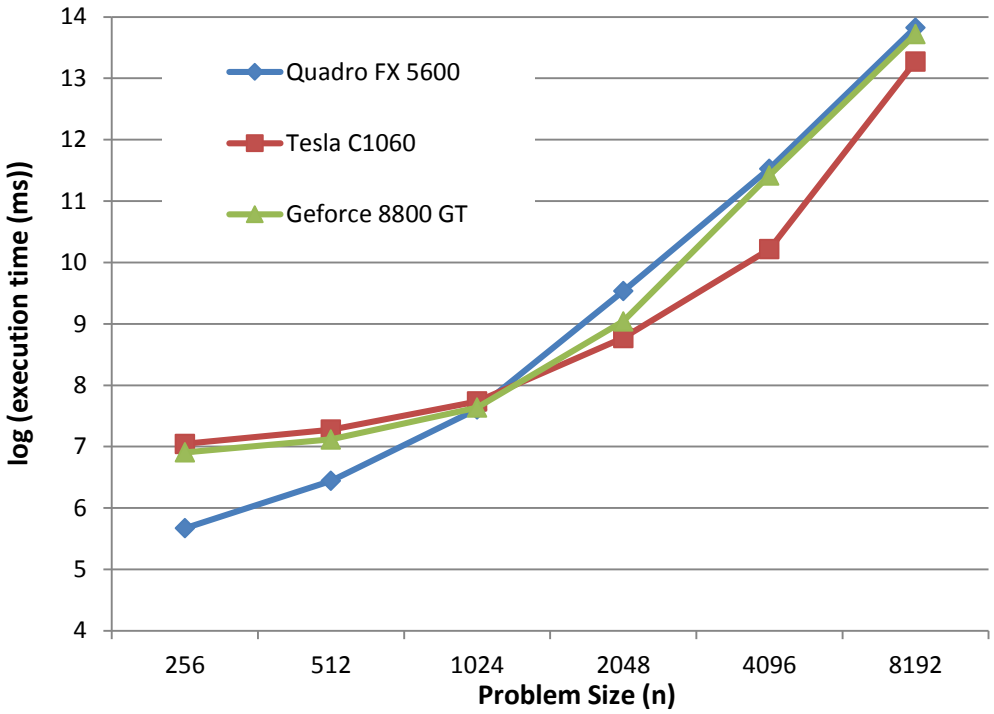

Figure 18. Comparison of execution time on three GPUs

3.7 Conclusion

We studied different approaches for porting a family of Dynamic Programming (DP) algorithms to GPUs. To do so, we used an abstract formulation of non-serial polyadic DP groups of algorithms. We presented how load balancing can be an important factor in achieving the right level of parallel granularity on many-core architectures, in contrast to multi-core architectures where locality and cache performance are more critical concerns. A comparison of a divide-and-conquer approach with an iterative bottom-up approach was presented. It is shown that significantly better load balancing is achieved using the iterative bottom-up approach. To evaluate the performance of different approaches, we used three NVIDIA GPUs: Tesla C1060, Quadro FX 5600, and GeForce 8800 GT. We achieved up to 228x speedup and 10 GFLOP/s on the Tesla C1060 compared to the multi-core implementation previously reported up to 30x speedup on a Cyclops64.

CHAPTER 4

Lock-Free Parallel-Friendly Hash Table on a Data-Parallel Many-Core Processor

4.1 Overview

GPUs (Graphics Processors) have evolved as many-core processors for general purpose computation. They offer massive parallelism through hundreds of cores and high memory bandwidth. Achieving the maximum performance from GPUs requires exposing large amounts of fine grain parallelism and structuring computations to regulate execution paths and memory access patterns.

Many-core processors are the future of computing and the domain of applications that are going to be executed on many-core processors will not be limited to pure parallel applications [Asanovic]. To be able to fully conquer the processing power of many-core processors we need to be able to utilize them for applications that have some sequential features. Hashing is one of these applications. By providing fast insert, search, and delete operations, hash tables are widely used as data structure for a lot of algorithms (e.g., set operations, associative arrays) and applications (e.g., database management) which require frequent data store and retrieval.

Chaining is a common method for resolving the conflicts of keys that are hashed to the same hash table bucket. Linked list traversal and modification is inherently a sequential process. In addition, buckets must be kept ordered to avoid duplicate keys in buckets or due to requirements of some applications (e.g., [Larson]). These inherently sequential characteristics make efficient implementation of chained hash tables on data-parallel many-core processors a challenge.

Shared sets are the building blocks of hash table bucket chains. On massively parallel architectures like GPU, to guarantee high throughput concurrent data structures, it is not viable for shared objects to be synchronized using any sort of lock. In GPUs, it is normally recommended that thousands or tens of thousands of threads needs to be executed simultaneously to hide memory latencies [Bell]. Therefore, locking is not a practical option. This is because while the thread that is holding the lock can be delayed due to memory latencies, other threads cannot make progress as well [Michael] (basically to hide the memory latency incurred by the thread holding the lock).

Lock-free (non-blocking) shared data structures however, guarantee more robust performance than locked-based implementations on parallel architectures. Lock-free set algorithms (which are building blocks of lock-free hash tables) are a well-known research area and several algorithms for lock-free set implementations have been proposed for conventional multiprocessors [Greenwald, Massalin, Valois, and Michael]. The GPU implementation for lock-free hash table [Moazeni12] based on CAS-based lock-free set algorithm [Michael] demonstrated that a lock-free approach can significantly outperform the lock-based hash table. This is despite the additional sequential overhead that is introduced by the CAS-based lock-free algorithm. This result is significant because the simplicity of chained hash tables makes them an essential candidate in many applications especially database applications [Larson]. However, although results of [Moazeni12] shows feasibility of chained hash tables in GPU by leveraging a lock-free hash table, the achieved performance is not ideal as it does not exploit the massive parallelism of GPUs. In this chapter, we demonstrate that the current best known CAS-based lock-free set algorithm [Michael] can be significantly enhanced for modern many-core GPUs.

Differences in many-core architectures compared to conventional multi-core architectures impose challenges for implementing concurrent data structures for these platforms. A many-core implementation must distribute work among hundreds or thousands of threads. Therefore, the contention on shared objects is much higher than in multi-core implementations. Contention on shared objects between threads does not allow exploiting maximum parallelism of many-core processors. In concurrent hash tables, contention between shared objects exists between objects that are hashed to the same hash bucket.

In a concurrent hash table, the problem of contention for shared objects residing in the same hash bucket chain can be escalated or alleviated with changes in data distribution. While uniform distribution of hash keys result in uniform distribution of conflict between hash table operations, a non-uniform (e.g. normal) distribution can easily exacerbate the contentions between working threads. With normal distribution of the keys, it is most possible that some of the hash table buckets become more crowded. Although a basic requirement for hash functions is that the function should provide a uniform distribution of hash values, but it is sometimes difficult to ensure uniformity especially as the distribution and type of input data may vary over time. In addition, there are applications [Larson] that require multiple versions of a key to be added to the hash table and be inserted adjacent to the all the other existing versions. In such applications, it is the input data distribution that determines the length of the bucket chain; even with perfect hashing, the length of chain in different buckets will vary.

Figure 19 shows the motivation to enhance lock-free hash tables for many-core processors. It shows the speedup of executing a batch of hash table operations using a basic GPU implementation of the CAS-based lock-free hash table over its counterpart multi-threaded implementation on a multi-core processor. As depicted, the achieved speedup is not comparable to the potentials that massively parallel architectures provide. Additionally, notice that the less the variance of hash key distribution, the less is the speed up achieved from the GPU.

As it will be discussed in the chapter, the major bottleneck that impacts GPU performance is the sequential overhead of lock-free hash table algorithm, which is introduced when there is contention in

hash buckets. Observing this, we propose a key distribution function to maximize the potential of hashing in GPUs.

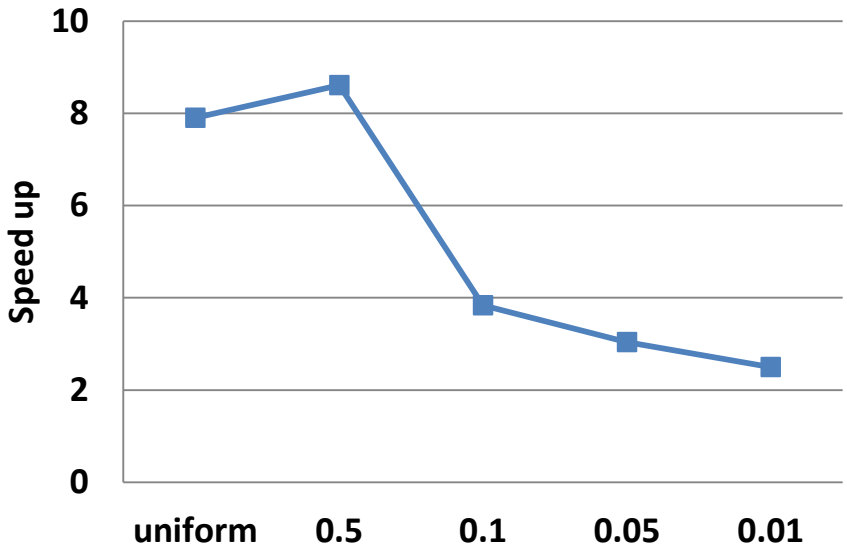

Figure 19. Speedup of basic GPU implementation of the CAS-based lock-free hash table over the counterpart multi-threaded implantation using Pthread by changing the variance in distribution of keys.

In this chapter, we introduce parallel hash table structure to diminish the contention on the shared objects and achieve significant throughput on many-core processor architectures. Our method provides multiple instances of hash table to GPU threads, and hence reduces the conflicts that are caused by thread operations in the same hash buckets. We present a key distribution technique that enables having multiple hash tables while ensuring uniqueness of keys. The proposed method supports all hash table operations (Insert, Search, and Delete).

After introducing our parallel hash table structure, we will present exhaustive analysis of its performance and characteristics. We show that despite its minimal memory overhead, this method is especially beneficial in many-core architecture (comparing to benefit of applying the same method to multi-core architecture implementation). Our method also provides an opportunity to cope with data-skew and poor-fit hash functions, which impacts many-core implementations severely.

For experimental evaluations we used NVIDIA GTX 480 with CUDA capability 2.1 and using CUDA

4.0. Our experimental results show that with combined Search, Insert and Delete workloads, we achieve a substantial improvement of 5X-27X compared to the counterpart multi-thread CPU implementation. Our solution also shows 25X-170X improvement over the non-optimized GPU implementation. In addition, ~3.5X speed up is achieved over a GPU implementation with same load factor but without our key distribution method.

The rest of the chapter is organized as follows. After discussing related work in Section 4.2 we overview the CAS-based lock-free list-based algorithm in Section 4.3. In Section 4.4, we describe the limitations of the algorithm for data parallel many-core processors, and the naïve GPU implementation of the algorithm. In Section 4.5, we introduce parallel hash table and describe its GPU implementation. Section 4.6 includes detailed performance analysis of the parallel hash table. Finally we conclude the chapter in Section 4.7.

4.2 Related Work

Hash tables are one of the most used and important data structures. Hence many studies have been focused on improving their algorithms, implementations and usage. Here we focus on most related work on performing hash table operations on multi and many-core processors.

In [Alcantara09], a data-parallel algorithm for building large hash tables is presented on the GPU. Their hash table implementation is not dynamic; if any or all of the data items in the hash table change, it rebuilds the table from scratch which is inefficient. Their approach is based on cuckoo hashing, which is done in shared memory. Shared memory is typically small, which makes this approach less practical for applications that need to store large objects in the hash table. Therefore, this hash table implementation is not general purpose (only shown useful in graphics applications) in contrast to our method. However, in this study, we focus on chaining as a perfectly practical mechanism in resolving hash conflicts in a wide domain of applications. In [Alcantara], they focus on chaining, but their main idea is to sacrifice the ability to modify the structure after the initial construction of the hash table. Parallel hashing has been studied from a theoretical perspective, mostly in the early nineties [Matias, Bast, and Gil].

Lock-free set algorithms (building block of lock-free hash tables) are a well-known research area and several algorithms for lock-free set implementations have been proposed for multiprocessors [Michael, Massalin, Valois, and Michael]. [Michael], is the lock-free hash table that is based on the current best known CAS-based lock-free list-based algorithm. This algorithm is not designed for a many-core processor. We demonstrate that the GPU implementation of this algorithm performs poorly in comparison to a multi-core implementation. Building on top of this algorithm, we propose a load balancing mechanism to enhance the algorithm for many-core architectures.

Bulk execution model is used to group multiple transactions into a bulk and execute the bulk as a single task on GPU in [He]. We use the same approach for executing operations on the hash table in GPU. Similar approaches to buffer the incoming requests are used in [Sewall]. [Moazeni12] shows that the lock-free hash table implementation based on CAS-based algorithm outperforms lock-based hash table implementation on GPUs. However, this implementation does not leverage massive parallelism of GPU, and comparing to multi-core implementation it shows minimal or no improvement depending on data distribution. By changing the underlying structure of hash table on GPUs we show that an order of magnitude improvement in performance is viable.

4.3 CAS-based Lock-free Algorithm

To make the study self-explanatory, in this section we briefly describe the architecture of CAS-based lock-free list-based set algorithm (*Michael's algorithm*). However, we refer the reader for complete description to [Michael].

The CAS-based lock-free list-based set algorithm [Michael] is the current best known algorithm for shared sets and hash tables. This CAS-based lock-free list-based algorithm is used as the building block for a lock-free hash table. It uses CAS (swap-and-compare) atomic primitive or equivalently restricted LL/SC (load-linked/store-conditional). All current major processor architectures support one of these two primitives.

A lock-free shared hash table guarantees that if more than one thread attempt to perform operation in

the same hash bucket, at least one of the threads will complete the operation in finite number of steps regardless of the other threads.

CAS-based lock-free hash table uses chaining for resolving the conflicts between the keys that hash to the same hash bucket. Figure 20 shows the pseudo-code for Insert, Search and Delete operations of the list-based set algorithm. To avoid duplicate keys in bucket, the most common method is to keep the linked list an ordered list. The function Find guarantees to capture a snapshot of a segment of the list including the node that contains the lowest key value greater than or equal to the input key and its predecessor pointer. The main idea is that the thread executing Find starts over from the beginning whenever it detects a change in **`*prev`**, in line A. This change means that some other threads have inserted element into the chain in the meantime.

The function Insert on the other hand uses the snapshot from the function Find to insert the new node. If the key already existed in the hash bucket, the Insert function will return without inserting the duplicate key. Using the CAS atomic primitive (or equivalently restricted LL/SC), it guarantees that only one thread can create a new link between **`prev`** and the new node. The failure of the CAS in line B implies that a new node was inserted immediately before **`cur`**, since the snapshot from the function Find was captured. Therefore, the thread executing Insert starts over from the beginning.

```
struct Entry {
  Key: KeyType,
  Value: ValueType,
  <Mark, Next>: <boolean,*Entry>
}

// hash function
h(key: KeyType)
{
 //any function returning value in {0…m-1}
}

// hash table operations
bool HashInsert(key: KeyType, value: ValueType)
{
  node <- AllocateNode();
  node.key <- key;
  node.value <- value;
  return Insert (&H[h(key)], node);
}

bool HashSearch(key: KeyType, out value: ValueType)
{
  success<-Search(&H[h(key)],key,out value);
  return success;
}

bool HashDelete(key: KeyType)
{
  return Delete(&H[h(key)], key);
}
```

Figure 20. Hash table operations

More than one thread can get the same snapshot from the function Find. Therefore, more than one thread can attempt to link their node to the same spot in the linked list. Using an atomic CAS primitive to link the new node to the linked list is used instead of locking to control concurrency of threads for the final insertion. The advantage of using the atomic CAS is that it is always guaranteed that one of the threads that have contention over a node will win. This prevents deadlock and offers robust performance.

The function Delete attempts to mark **`cur`** as deleted, using the CAS in line C (The Mark field

indicates that the key in the node has been deleted.). If successful, the thread attempts to remove **`cur`** by swinging **`prev.Next`** to **`next`**, while verifying **`prev.Mark`** is clear, using the CAS in line D.

All the three functions Insert, Search, and Delete invoke function Find which means they all will start over from the beginning of the linked list if they detect that another thread is making changes in the same spot.

4.4 Implementation on Data-Parallel Many-Core Architectures

In this section we discuss the limitations of the CAS-based lock-free hash table implementation on data-parallel many-core processors. We also describe our GPU implementation of the algorithm.

```
//Private variables; note that these variables are shared variables between Find,
Insert, Search and Delete functions in this pseudo code.
Prev, cur, next: *Entry
bool Find(head: *Entry, key: KeyType)
{
  try_again:
  prev <- head;
  <pmark,cur> <- prev.<Mark, Next>;
  while true
  {
     if cur = null return false;
     <cmark, next> <- cur.<Mark, Next>
A:    if *prev ≠ <0,cur>
        goto try_again;
     if (!cmark)
     {
       if (cur.key ≥ key)
          return cur.key = key;
       prev <- &cur.<Mark,Netx>;
     }
     else
     {
       if (CAS(prev, <0,cur>, <0,next>))
          DeleteNode(cur);
       else goto try_again;
     }
     cur <- next;
  }
}

bool Insert(head: *Entry, node: Entry)
{
  key <- node.key
  while true
  {
    if Find(head, key) {result <- false; break;}

     node.<Mark,Next> = <0,cur>;

B:   if (CAS(prev, <0,cur>, <0,node>))
  {result <-true; break;}
  }
  return result;
}
```

Figure 21. The CAS-based lock-free algorithm (Inset and Find)

```
bool Search(head: *Entry, key: KeyType, out value: ValueType)
{
  success <- Find(head, key);
  value <- cur.value;
  return success;
}


bool Delete(head: *Entry, key: KeyType)
{
  while true
  {
    if (!Find(head, key))
    {
        result <- false;
        break;
    }

C:   if (!CAS(&cur.<Mark,Next>, <0,next>, <1,next>)) continue;

D:   if (CAS(prev,<0,cur>,<0,next>))
        DeleteNode(cur);
     else
        Find(head, key);

     result <- true; break;
  }
  return result;
}
```

Figure 22. The CAS-based lock-free algorithm (Search and Delete)

4.4.1 Limitations on Data-Parallel Many-Core Architectures

In this study, we use NVIDIA modern GPUs as a data-parallel many-core processor. Limitations of implementing this algorithm on modern GPUs are:

1- The algorithm uses chaining for resolving hash conflicts. This is naturally not suitable for GPUs because it requires sequential access to the linked list structure on the GPU.

2- There is variable work per operation. Chaining requires traversing the linked list, which can vary in

length. This increases branch divergence in the SPMD (Single Program Multiple Data) execution model of the GPU which causes inefficiency.

3- GPU uses many more threads (hundreds or thousands of threads) for concurrent operations in the hash table compared to a multi-threaded implementation. Therefore, the probability of contention between shared objects is higher. Each conflict in the course of a Search, Insert or Delete operation means starting over from the beginning (as described in Section 4.3).

The more threads concurrently inserting or deleting in the hash table, the higher the probability of conflicts among threads performing operations in the same hash bucket becomes. Increase in the number of conflicting threads may also lead to increase in divergent branches. As more threads fail to complete their operations due to a conflict, they need to start over from the beginning (Lines A, B, C, and D in Figure 21 and Figure 23). This causes more variation in the length of linked list traversal that is needed for all threads to complete their operations. With normal distribution of the keys it is most possible that some of the hash table buckets become more crowded. Therefore, inserting into the same hash table bucket leads to higher chance of conflicts. We introduce parallel hash tables to address this issue in Section 4.5.

4.4.2 Basic GPU Implementation

In this section we describe the implementation of lock-free hash table on GPU, which uses NVIDIA's CUDA programming environment and targets GPUs that feature atomic global memory operations. This implementation naively implements the CAS-based lock-free algorithm on GPU with no further adaptation.

The implementation of the lock-free hash table on GPU requires a *bulk execution model* [He] to group multiple hash table operations within a batch and to execute the batch on GPU as a single task. The operations within the batch are executed concurrently on the GPU.

In the bulk execution model, we buffer the input operations and then send them to GPU and wait for the result of the whole bulk. Without a bulk execution model, we naturally spawn threads as requests

enter the system. However, this is not feasible for GPU-based implementations. On the other hand, in order to avoid sacrificing the average response time in a bulk execution model, we need to have a reasonable limit on the batch size.

The hash table is completely stored in the GPU Global memory. For the hash table data structure, we use a pool array of pre-allocate nodes. We attach nodes from the pool to the hash table buckets dynamically to build a linked list as each hash table bucket. For each node in the linked list instead of using a pointer to the next node, we use the index of the next node in the pool array. This is done because of limitations of CUDA and for two reasons 1) efficient memory allocation and 2) the fact that the CUDA **`atomicCAS`** does not support pointer types. If we wanted to have each thread allocate a node from the heap, the per-thread allocation is very costly. Therefore, we have to pre-allocate a pool of nodes to amortize the cost of memory allocation. In addition, to use pointer types, we needed to convert pointers to **`longlong`** and vice versa for performing basically any action on the linked list (which is very inefficient). Each dynamic node must contain the following fields: Key, Value and the <Mark, Next>. The Mark field indicates that the key in the node has been deleted. <Mark, Next> has to occupy a contiguous aligned memory block that can be manipulated atomically using **`atomicCAS`**. The Mark bit and the Next index can be placed in one word. We use the left-most bit for the Mark bit. Figure 23, shows the data structure used for implementation of each node entry in the linked list.

```
struct Entry {
  Key: KeyType,
  Value: ValueType,
  <Mark,Next>: int
}
```

Figure 23. Hash table data structures

4.5 Parallel-Friendly Lock-Free Hash Table

In this section we describe our solution for a parallel-friendly lock-free hash table for many-core processors. In many-core processor architectures, there are hundreds or thousands of threads that are

performing an operation on the lock-free hash table concurrently. Therefore, many threads can be traversing a hash bucket or making changes to the shared objects in the same hash bucket concurrently. In the CAS-based lock-free set algorithm, a thread that is making changes to a shared object for an Insert or Delete operation can cause other threads that are reading the shared object for traversal (i.e. for either Insert, Search or Delete operations) to start over from the beginning of the linked list. This is a huge overhead and may cause more threads to diverge.

4.5.1 Introducing Parallel Hash Table

We propose to make the lock-free hash table parallel friendly by introducing Parallel Hash Table *PH* that changes the underlying structure of a hash table *H* with hash function *h* with *m* hash buckets.

Parallel hash table $PH[n]$ of size n consists of n hash table ph_i each having m hash buckets as the original hash table *H*. Each key k will be inserted to hash table $ph_{TA(k)}$ using the original hash function $h(k)$ where $TA(k)$ is the Table Assignment function and $0 < TA(k) \leq n\text{-}1$ (Figure 25). To find key k, hash table $ph_{TA(k)}$ is searched for key k. A definition of parallel hash table operations Insert, Search and Delete is given in Table 4. The Table Assignment function should guarantee the uniqueness of keys inserted to the hash table. Therefore, the fundamental requirement is that Table Assignment function should map each key uniquely to a hash table instance $ph_{TA(k)}$. More details on the requirements of Table Assignment function is given later in this Section (V.B).

Note that there is no change in the original hash function h or the original number of buckets m in each hash table instance. This is a fundamental aspect of parallel hash tables, and the reason is that it is not always possible to improve the hash function in all applications to reduce conflicts nor it is desired to re-design hash functions. The parallel hash table functions listed in Figure 24 can be seamlessly integrated to any application that uses hash table, with no requirement to adapt the application to deal with the underlying structure of parallel hash table. Only the backend system needs to be aware of the existence of multiple hash tables and deal with balancing conflicting threads using its Table Assignment function.

Table 4. Parallel hash table operations.

Functions	***Description***
bool Insert (k, value, PH)	**Insert (k, value, $ph_{TA(k)}$).** Insert key/value pair (k,value) into $ph_{TA(k)}$ using the hash value h(k) if there is no duplicate key in $ph_{TA(k)}$. Returns false if found a duplicate.
bool Search(k, out value, PH)	**Search (k, out value, $ph_{TA(k)}$)** Search the hash table $ph_{TA(k)}$ using hash value h(k). Return value if search is successful. Returns false if key is not found.
bool Delete (k, PH)	**Delete (k, $ph_{TA(k)}$)** Search the hash table $ph_{TA(k)}$ using hash value h(k). Deletes key/value pair if Search is successful. Returns false if key is not found.

```
// Hash function
h(key: KeyType): 0…m-1
{
  …
}

// Table Assignment function
TA(key: KeyType): 0…n-1
{
  …
}

// hash table operations
bool HashInsert(key: KeyType)
{
  node <- AllocateNode();
  node.key <- key;
  node.value <- value;
  return Insert(&HP[TA(key)][h(key)],node);
}

bool HashSearch(key: KeyType, out value: ValueType)
{
  success<-Search(&HP[TA(key)][h(key)],key, out value);
  return success;
}

bool HashDelete(key: KeyType)
{
  return Delete(&HP[TA(key)][h(key)], key);
}
```

Figure 24. Parallel hash table operations.

For Insert operations, the main idea is that threads that are inserting to the same bucket are routed to the same bucket of different hash tables ph_i $(0 \leq i < n)$ using TA(k). It is a requirement for TA function to always map key k to a unique hash table $ph_{TA(k)}$. Therefore, if there are more than one thread simultaneously attempting to insert the same key k to the hash table, parallel hash table guarantees to route all of those threads to the same hash table $ph_{TA(k)}$. After routing all the threads to the same hash table

$ph_{TA(k)}$, Michael's algorithm guarantees that only one of the threads succeeds to insert key k.

For Search operations, the idea is to search the hash table bucket designated by the hash function h in the hash table instance that is uniquely identified by $TA(k)$. If the key k is found in hash table $ph_{TA(k)}$, it is guaranteed to be unique in the hash table. In addition, if the key k, is not found in $ph_{TA(k)}$, it is guaranteed that it does not exist in the parallel hash table.

In Section 4.6, we present experimental evaluations and more detailed performance analysis of lock-free parallel hash table.

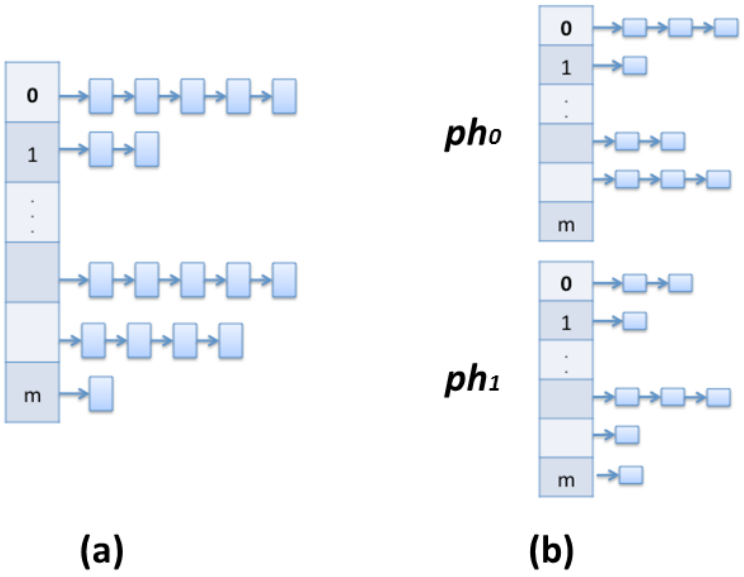

Figure 25. (a) Simple hash table. (b) Equivalent parallel hash table with two tables.

By leveraging parallel hash table, we distribute the threads that are inserting to the same hash bucket to different hash tables. Parallel hash table provide four advantages:

1- **Significantly reduces the conflicts and traversal time.** Lowers the probability of conflicts amongst threads that are operating in the same hash bucket. This increases the chance to complete hash table operations without being forced to start over due to conflicts (in lines A, B, C, and D in Figure 21 and Figure 22). Hence reducing the traversal time. This improves hash table operations Insert, Search, and Delete as they all need to traverse hash bucket linked lists.

2- **Reduces branch divergence.** Results in less variation in the length of linked list traversal, which lead to less divergence among threads.

3- **Reduces the sequential overhead of chaining.** Reduces the length of chains in the hash tables which reduces the traversal time as GPU is not efficient for serialized tasks.

4- **Improves the uniformity of the hash function**. Discussed n Section 4.5.3.

The additional memory that is required for parallel hash table is negligible compared to the benefits that it provides for reducing the conflicts between threads. To construct a parallel hash table we only need an array of header pointers per ph_i in $\{ph_0, \ldots, ph_{n-1}\}$. Comparing *PH* with a normal hash table *h* with same number of hash buckets *m*, the additional memory for *PH* is:

$$\text{Memory (PH[n])} - \text{Memory (ph)} = (n-1) \times m \times \text{Memory (header)}$$

In the experiment section, we show how this design is not as effective in improving the performance of CPU multi-threaded implementation. The main reason is that limited number of threads in multi-threaded implementation exhibit less contentions. Moreover, threads on a CPU can traverse long chains of data in buckets more efficiently, while in GPU, individual threads have much less performance and the difference between the length of chains causes branch divergence. In Section 4.6, we further discuss the impact of the Table Assignment method on performance.

4.5.2 Proof of Correctness

The parallel hash table structure is a direct extension of Michael's lock-free set algorithm [Michael]. For brevity, since it is straight forward to extend the proof, we refer the reader to [Michael] for proof of linearizability, lock freedom, and safety of the algorithm. Here, we only show the informal intuition of why adding multiple tables does not violate the unique key criteria of hash table. Since both function *h* and function *TA* are deterministic functions, Insert operation always maps redundant keys to the same instance of parallel hash tables. Hence the linearizability and safety of Michael's algorithms guarantees the uniqueness of the key in the parallel hash table.

4.5.3 Table Assignment Function

We use a function to distribute the hash table operations amongst multiple instances of the hash table structure. The goal is to distribute the hash table operations to different hash table instances, regardless of the distribution of the keys. This lets parallel threads which are operating in the same hash buckets to perform their operations on different table instances, and hence avoid conflicts. The Table Assignment function does not have knowledge of the input data. The requirements of the Table Assignment function are:

1- Must be a function over the input keys. This is to ensure each key k is mapped to a unique hash table identified by $TA(k)$.

2- To achieve best performance, $TA(k)$ should be such that for majority of input values k, if $h(k_1)=h(k_2)$ then $TA(k_1)\neq TA(k_2)$. This is to ensure that keys that are hashed to the same hash bucket are not mapped to the same hash table instance.

Adhering to these requirements, tables are assigned independent of hash values. In addition, being a separate function, TA function can be paired with any hash function with any complexity. It should be noted that the table assignment mechanism does not change the distribution of data in different buckets. However, by scattering it between different tables, it makes hash table operations less contentious. In our experimental results, we study the impact of our table assignment method with respect to both uniform distribution and normal distribution of hash keys.

Table assignment is especially suitable for data-parallel many-core processor architectures because it is performed in a single step with no conditions. Unlike collision resolution methods, Table Assignment in parallel hash tables does not require probing like open addressing or multiple hash functions like Cuckoo hashing. Table assignment is an approach that is combined with chaining and does not require table resizing.

Consider a hash function that is designed for an application. The distribution of data may changes in a way that the uniformity of the hash keys generate by hash functions is impacted. A many-core

implementation will be hit by this more than multi-thread implementation. The reason is that there are many more threads that will conflict while performing a hash table operation on the same hash bucket. Our proposed solution solves this conflict problem on many-core processors.

4.5.4 Increasing the Number of Hash Buckets

Increasing the number of hash buckets *m* can also reduce the conflicts between threads. However, note that increasing the number of hash buckets does not fully overcome the adverse effect of imbalance distribution of keys. If the designed hash function does not map incoming keys to different hash values, increasing the number of hash buckets has no impact in reducing the conflicts between threads. Multiplying the number of buckets by a factor *k*, is more helpful if it increases the variance of hash value distribution proportionally.

On the other hand, using a Table Assignment function that is independent of the hash function, it is very unlikely to have an input data distribution that is adversary to both the hash function and the TA function. Therefore, when hash function does not uniformly distribute keys because of unexpected changes to the distribution or other characteristics of the keys, using parallel hash table with a TA function with aforementioned requirements could have higher chance in preventing contention between threads. In addition, in the worst case scenario where the TA function fails to distribute keys, the performance will not be worse than the original hash table. In section 4.6 we evaluate both approaches and show the advantage of using parallel hash table over plain increase in the number of hash buckets.

4.5.5 Implementation on GPU

The lock-free parallel hash table extends the hash table structure used for our basic GPU implementation described in Section 4.4.2. Parallel hash table is also completely stored in GPU Global memory. Parallel hash table with n parallel tables is implemented as *n* arrays of header nodes. Each bucket in hash table $PH[i]$ is a linked list with the header node in the $PH[i]$ array.

4.6 Performance Analysis

We evaluate the performance of lock-free parallel hash table through exhaustive set of experiments on NVIDIA GTX 480 with CUDA 4.0. For comparison we used a Multiprocessor system with two Quad-Core Intel Xeon E5405 processors.

The execution time and throughput are obtained by taking the average over 10 runs. The time for transferring the data to GPU is not considered in the execution time, however, we study data transfer time to GPU in our evaluations.

4.6.1 Benchmarks

We generated workloads of operations (Insert and Search, and Delete) by choosing random keys (distributed between 0 and 1). To study the impact of data distribution, both uniform and normal random number generation methods are used. We used polar form of the Box-Muller transformation for normal random generation. Unless otherwise mentioned, variance is set to 0.05 in all experiments for normal distribution of keys. To reduce the artifact of table emptiness on performance, in all experiments we pre-load the hash table with 262,144 key/value pairs. The number of hash table buckets in all of our experiments is 1024. Note that the number of hash buckets is not important for comparison of different methods; it is the load factor that is important for evaluation. Unless otherwise mentioned, we used DEK [Knuth] hashing as the hash function in evaluations. We used a mod function for the TA function in all experiments: $TA(k) = k \times 10^6 \% n$, where n is the number of parallel hash tables and k is the key. We chose this function since it is simple to experiment and change its behavior. In addition, since the hash functions that are used are complex, the output of the table assignment and hash functions show independent relationship. In all figures, the number of tables is shown by variable PH. We experiment with various batch sizes and then we later show how it impacts the overall performance and the response time of the system (see section 4.4.2 for details of bulk execution model).

Five different algorithms are implemented to evaluate the performance of the lock-free hash table and lock-free parallel hash table on GPU and multi-core CPU:

Basic GPU implementation for lock-free hash table (GPU_BASIC): This is the basic lock-free hash table implemented on GPU based on [Moazeni12]. We use a thread block size equal to 512 in all experiments, as it achieves best performance in all variations. Throughout the graphs in this section, in comparison to parallel lock-free hash table we represent the basic GPU lock-free implementation as PH=1 since it can be considered as a simplification of the parallel hash table with a single hash table.

GPU implementation for lock-free parallel hash table (GPU_PH): We implemented this as described in Section 4.5. We use a thread block size equal to 512 in all experiments, as it achieves best performance in all variations. We refer to lock-free parallel hash table Implementation as GPU_PH. Throughout the graphs in this section, we represent the parallel lock-free hash table with *i* parallel hash tables as PH=*i*.

GPU implementation with larger number of hash buckets (GPU_BIG): this is the basic GPU hash table which uses the same implementation as GPU_BASIC but has uses the same amount of memory as GPU_PH.

CPU Multi-thread implementation for lock-free hash table (CPU_BASIC): Pthreads library [Butenhof] is used to implement the counter-part implementation of lock free hash table on a multi-core CPU [Michael]. The Pthreads implementation is also based on bulk execution model as we described in Section 4.4.2. Basically we group multiple hash table operations within a batch and distribute the batch to threads. We use 8 threads in this benchmark because it achieves the best results for our Pthreads implementation. The multiprocessor that is used has 2 Quad-Core Intel Xeon processors.

CPU Multi-thread implementation for parallel lock-free hash table (CPU_PH): We also implemented lock-free parallel hash table using Pthreads. The implementation uses the same table assignment function as the GPU implementation. We use 8 threads in this benchmark because it achieves the best results for our Pthreads implementation.

4.6.2 Lock-Free Parallel Hash Table Performance

In this section, through a set of exhaustive experiments we demonstrate the performance of lock-free

parallel hash table as our proposed solution for a more efficient dynamic hash table with more robust performance on many-core GPUs. We consider all hash table operations, Search, Insert and Delete operations in this evaluation. Bulk execution model is implemented equally for all the different implementations. Since normal distribution of the keys results in more contention for the GPU implementation, we mostly show data for the normal distribution of the keys throughout this section.

Table 5 presents the additional memory that is required for the lock-free parallel hash table structure. This memory footprint is negligible comparing to the scale of data that is stored in the table.

Table 5. Extra memory required for parallel hash tables. Number of hash table buckets is 1024.

# of Hash Tables	4	16	64	128	256
Extra Memory (KB)	32	128	521	1024	2048

4.6.2.1 Performance of Insert

Figure 26 presents the throughput of the Insert operations. As demonstrated, increasing the number of tables in GPU_PH significantly increases the throughput. The throughput stabilizes after increasing the number of hash tables more than some threshold, because there are sufficiently more hash table instances than active threads. This threshold depends on the batch size. Figure 27 presents the speedup of GPU_PH Insert operations (varying the number of parallel hash tables, and batch size) over the CPU_BASIC. As expected, speedup is increasing with respect to increase in the number of hash tables. Speedup is also increasing as the batch size increases. For brevity, we skip detailed analysis of Delete operation as it exhibits characteristics similar to Insert operations.

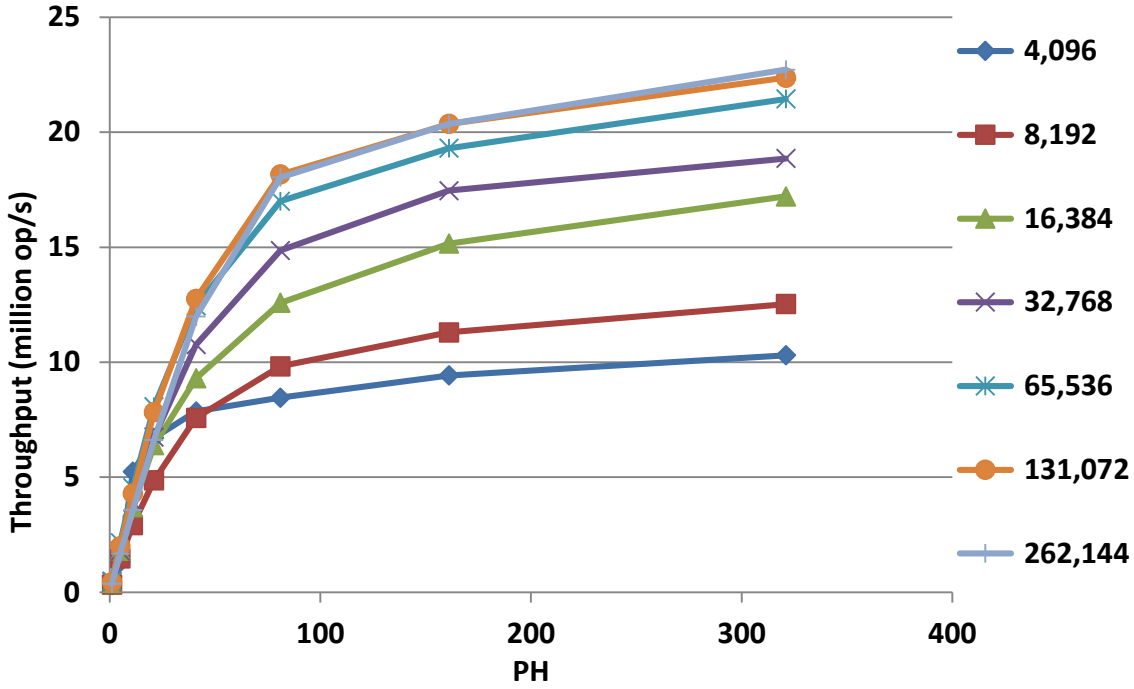

Figure 26. Throughput of Insert operation batches with normal distribution of the keys in GPU_PH (varying number of tables).

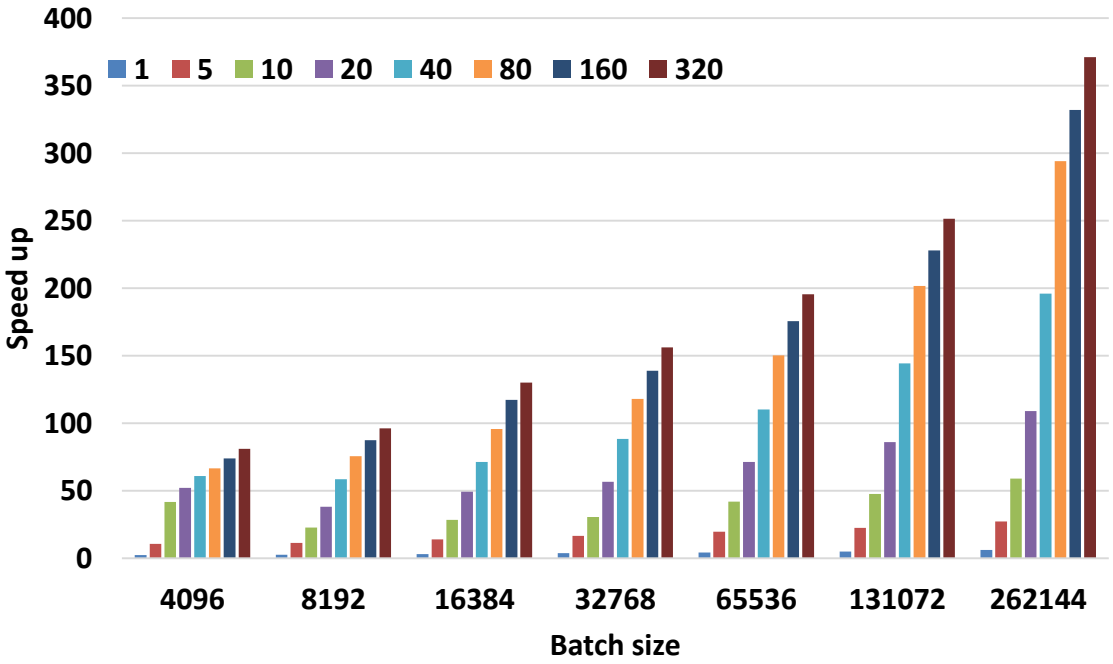

Figure 27. Speedup of Insert operation batches in GPU_PH compared to the CPU_BASIC. GPU_BASIC is also represented with PH=1 in the diagram.

Figure 28 shows how the performance of Insert operations is affected by changing the variance in the distribution of the keys. The figure shows the difference in execution time for different variances in data distribution. As expected, with lower variance we observe higher execution time. However, by increasing the number of hash tables, the execution time reaches the same number for the normal distribution with the three different variances.

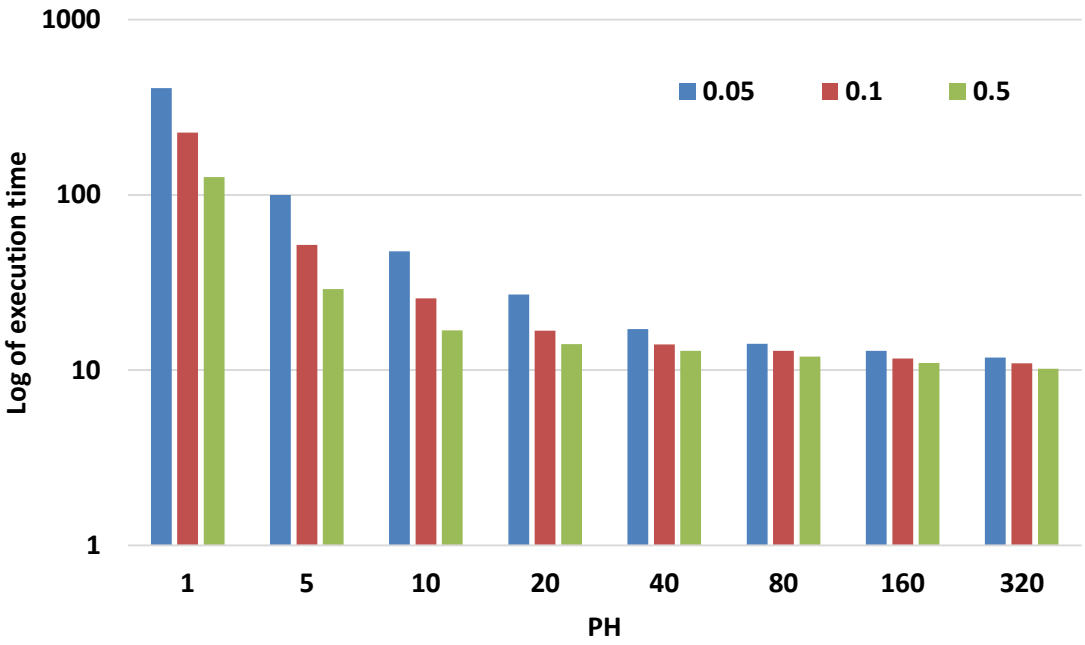

Figure 28. The execution time for a batch of Insert operations in GPU_PH by changing the variance in normal distribution of the keys.

4.6.2.2 Performance of Search

Figure 29 shows the speedup of GPU_PH over CPU_BASIC for Search operations. We observe that as we increase the number of parallel hash tables, speedup increases especially in larger batch sizes. It also shows that we achieve speedup against the CPU_BASIC in all configurations.

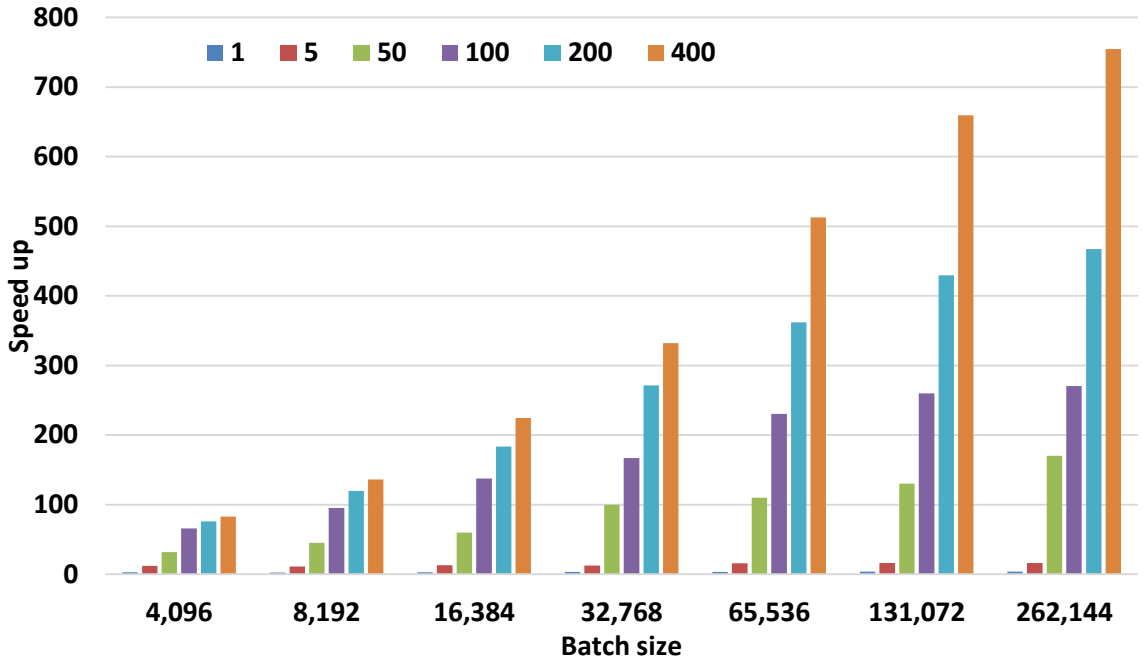

Figure 29. Speedup of Search operation batches in GPU_PH compared to CPU_BASIC.

4.6.2.3 Overall Performance

In this section we evaluate the overall performance of the GPU_PH against CPU_BASIC, GPU_BASIC, and CPU_PH. We evaluate performance with batches containing a combination of Search, Insert and Delete operations. We experiment with five variations: 1) 33% Search, 33% Insert and 33% Delete 2) 50% Search, 25% Insert and 25% Delete 3) 60% Search, 20% Insert and 20% Delete 4) 80% Search, 10% Insert and 10% Delete 5) 90% Search, 5% Insert and 5% Delete.

In Figure 30, we show the throughput for GPU_PH. Increasing the number of hash tables increases the throughput in all combinations. By increasing the number of hash tables more than a point (PH=100 in the graph), workload combinations with higher percentage of Search start to show higher increase in throughput. This is because increasing the number of hash tables more than a point does not increase Insert throughput as much as it increases the Search throughput.

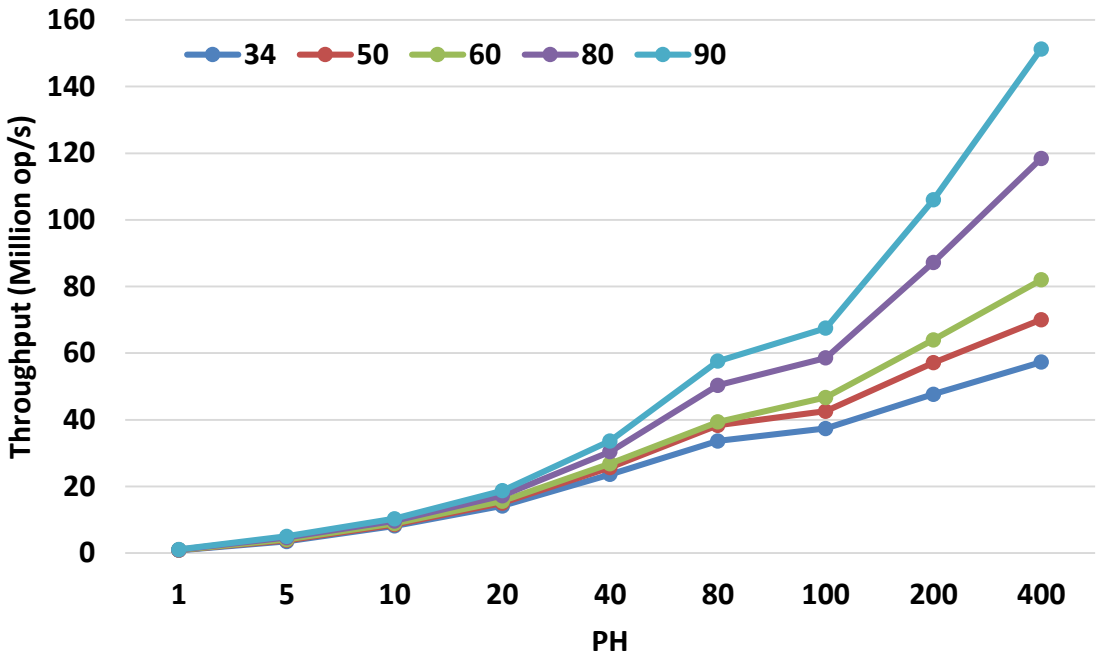

Figure 30. Throughput of combined Search, Insert and Delete operation batches with normal distribution of the keys for GPU_PH. The number in legend is search operation. Batch size is 262,144.

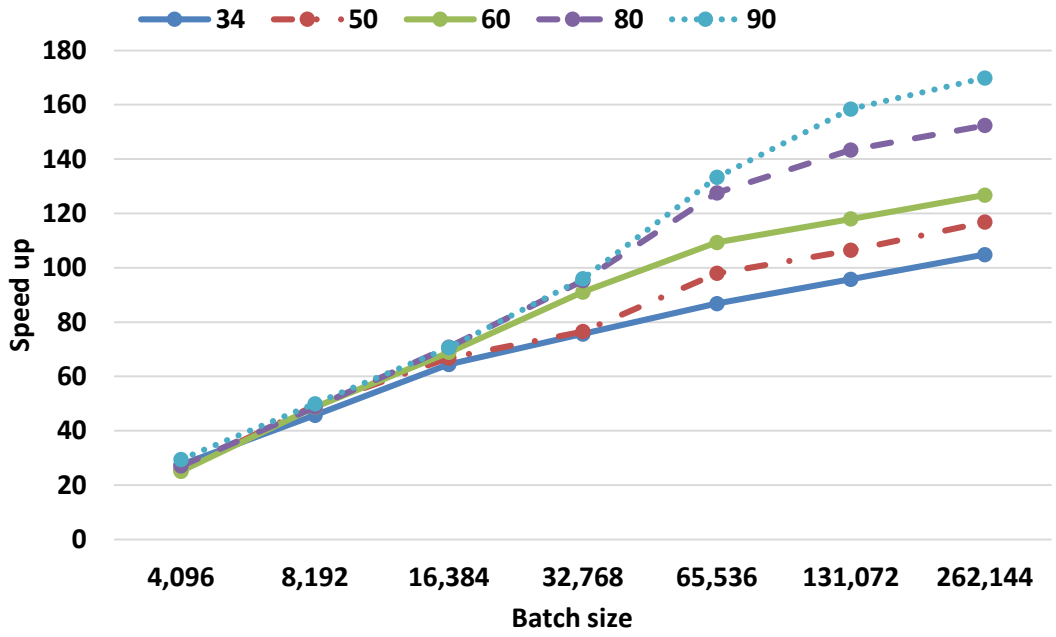

Figure 31. GPU_PH (PH=320) speedup over GPU_BASIC batches with normal distribution of the keys. Legend shows the search operation percentage in batches.

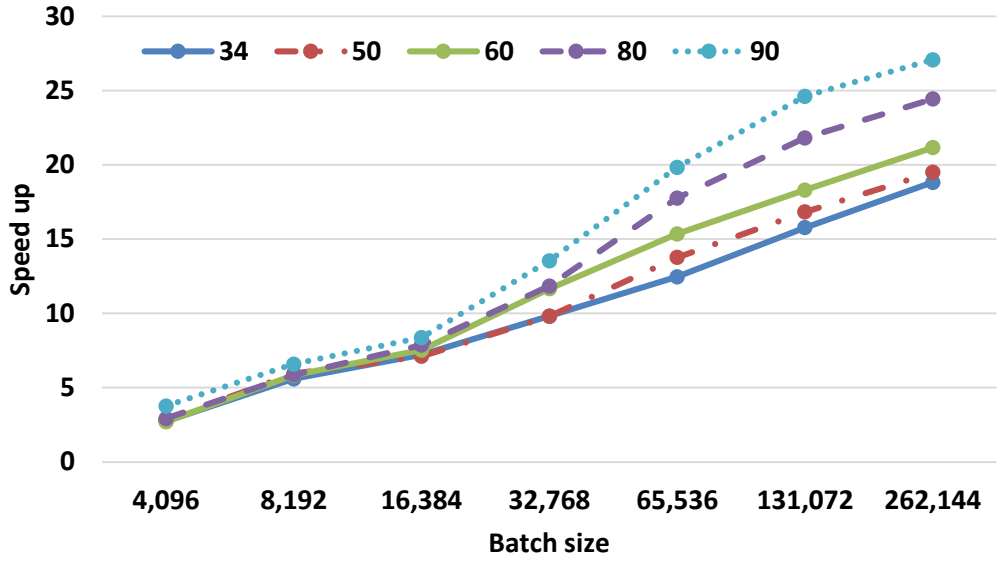

Figure 32. GPU_PH (PH=320) speedup over CPU_PH for combined batches of Search, Insert and Delete operation with normal distribution of the keys.

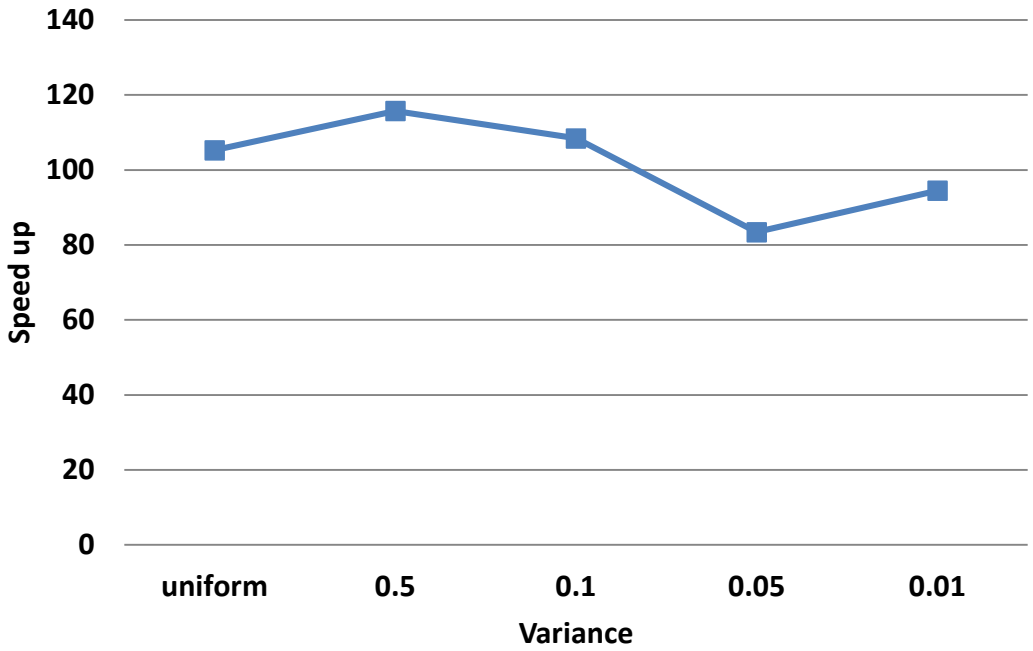

Figure 33. Overall speedup of GPU_PH (PH=320) over GPU_BASIC with both normal and uniform distribution of keys.

In Figure 31, the speedup of GPU_PH over GPU_BASIC for batches containing a combination of Search, Insert and Delete operations is shown. GPU_BASIC is configured to be at its peak performance and GPU_PH uses, and 320 hash tables. We experimented with the same five variations as in the last experiment. With the range of batch sizes in our experiment we achieve 25X-170X speedup.

In Figure 32, the speedup of GPU_PH over CPU_PH is shown for batches containing a combination of Search, Insert and Delete operations. Both benchmarks are implementations based on parallel hash table that use 320 hash tables. As it is shown CPU_PH performance is also improved compared to CPU_BASIC. However, the limited number of threads in multi-threaded implementation has smaller number of contentions. Moreover, threads on a CPU can traverse long chains of data in buckets more efficiently, while in GPU, individual threads have much less performance to traverse the chain serially. Therefore, reducing the length of chains in parallel hash table is more effective on GPU performance. With the range of batch sizes in our experiment we achieve 5X-27X speedup.

To show how changing the variance in normal distribution of the keys affects performance in GPU_PH compared to GPU_BASIC, we evaluated GPU_BASIC and GPU_PH (with 320 hash tables) by changing the variance from 0.5 to 0.01. In Figure 33, it is shown that GPU_PH achieves 80X-120X speedup over the GPU_BASIC at its peak performance. GPU_PH also demonstrates to be more stable by changing the variance in data distribution.

4.6.2.4 Impact of Hash Function

It is well known that choosing hash function indeed has impact on hash operation performances. Here we study the impact of parallel hash table on different hash functions. The hashing is done in two steps. The hash functions is independent of the hash table bucket size, and it is then reduced to an index (between 0 and the 1024, which is the size of the bucket array) using a remainder operation. Table 6 gives a description of the hash function in this experiment.

Figure 34 shows the throughput for batches containing a combination of Search, Insert and Delete operations with normal distribution of the keys. As it is shown, with all hash functions we notice steady

increase in throughput by increasing the number of hash tables, which is mainly due to the independence of TA function from the hash function. SDBM and DEK show the best overall performance with our data set. PJW shows lower overall performance as throughput stabilizes as we increase the number of hash tables.

Table 6. Hash functions description

Hash function	Description
SDBM Hash	Used in open source SDBM project [Seltzer]
PJW Hash	Proposed by P.J.Weinberger [Aho]
DEK Hash	Proposed by Donald E. Knuth [Knuth]

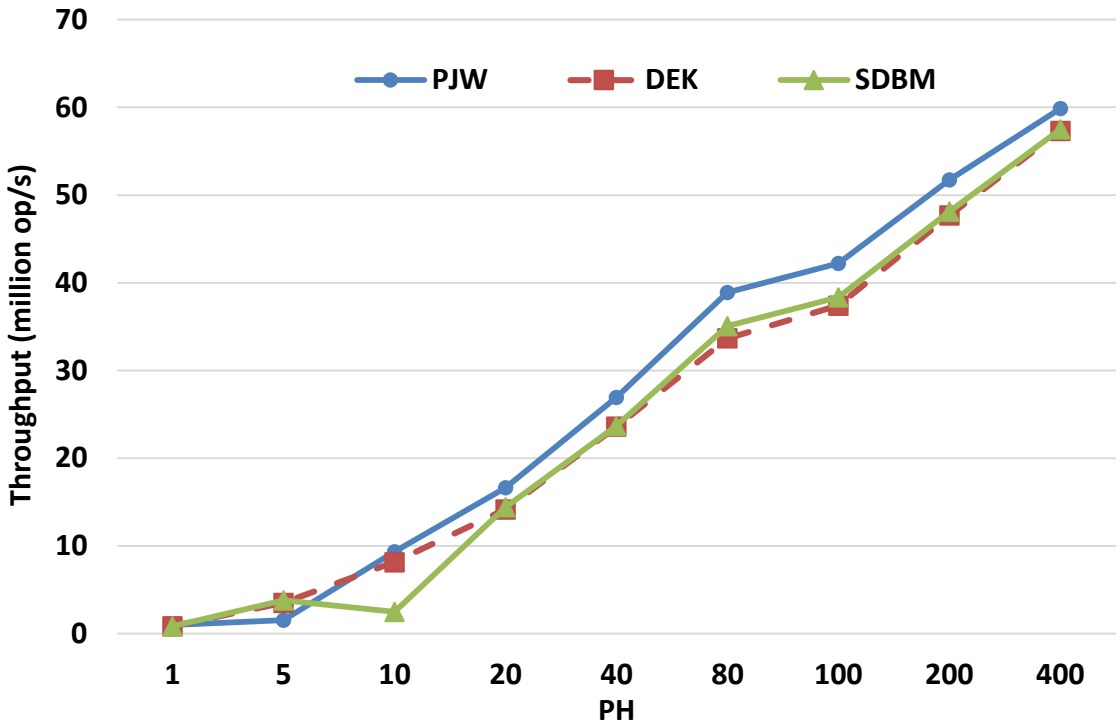

Figure 34. Throughput of combined 33%Search, 33%Insert and 33%Delete operation batch in GPU_PH with normal distribution of keys using three different hash functions.

4.6.2.5 Impact of Table Assignment Function

Table assignment method distributes the workload between the hash tables in the parallel hash table structure. It should be noted that even though such mechanism benefits all type of workloads, the impact on unbalanced workloads (e.g., workload with normal distribution of hash keys) is more. Figure 35, shows the speedup of GPU_PH over GPU_BASIC with random and normal distribution of the keys. It

depicts that we achieve higher speed up in using parallel hash table for Insert workload with normal distribution versus uniform distribution of keys. The reason is that as a thread has to spend more time traversing a longer bucket chain, there is also higher probability that the chain snapshot becomes obsolete due to operation of other threads (and chain traversal should be repeated). Hence the speed up achieved for distributing data between tables is higher for workloads with unbalance key distribution.

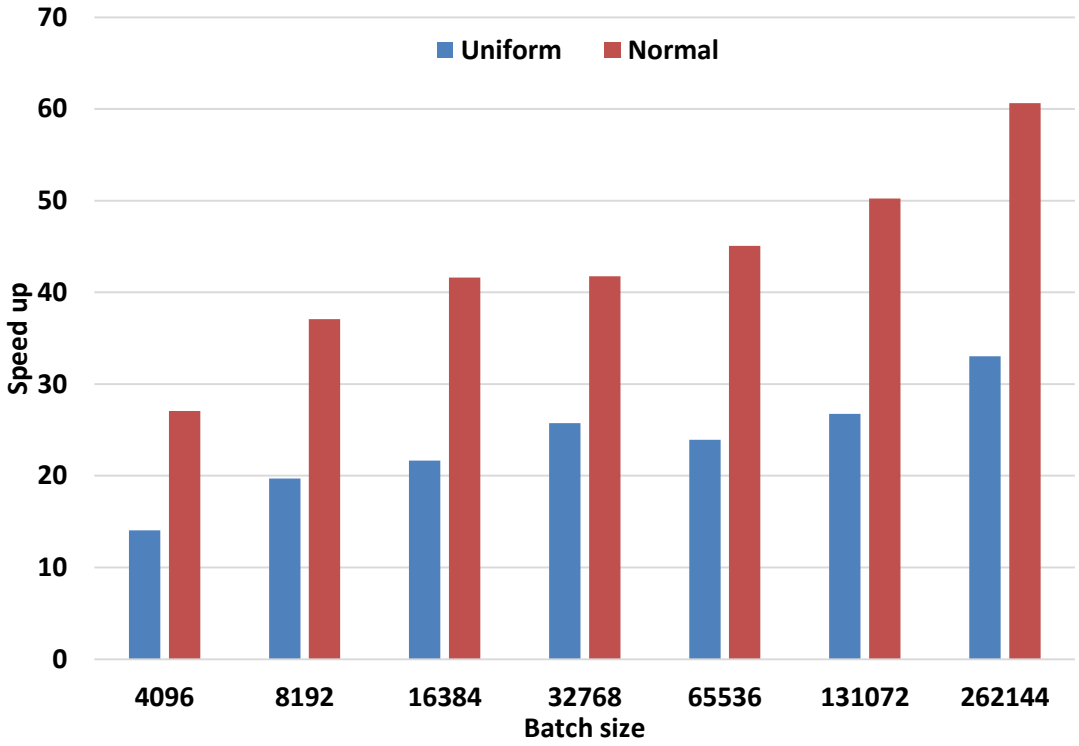

Figure 35. Effect of Table Assignment method on speedup of Insert operation batches in GPU_PH (using 320 hash tables) over GPU_BASIC.

4.6.2.6 Increasing the Number of Hash Buckets

In this section, we evaluate the effect of increasing the number of hash buckets. Increasing the number of hash buckets (size of bucket array) is one potential solution to reduce the conflicts between threads. Increasing the number of hash buckets has two drawbacks: First, it requires changing the original hash function, which is not feasible in many scenarios. Second, increasing the number of hash buckets does not necessarily increase the variance of hash key distributions proportionally. Indeed, increasing the size reduces contentions to some degree; however, if the input data distribution is an adversary case for the

hash function, the hash key distribution will still remain unbalanced. Following experiments examines latter.

In Figure 36, we compare the two methods by comparing GPU_PH with *N* hash tables and *M* hash buckets with GPU_BIG with *N***M* hash buckets. As demonstrated, GPU_PH shows ~3.5X speedup over GPU_BIG. We only show the comparison for combinations with 33% Search, 33% Insert, 33% Delete, however, all combinations show similar results. The speed up is slightly higher when the number of tables is less, as the impact of TA function in distributing keys is higher.

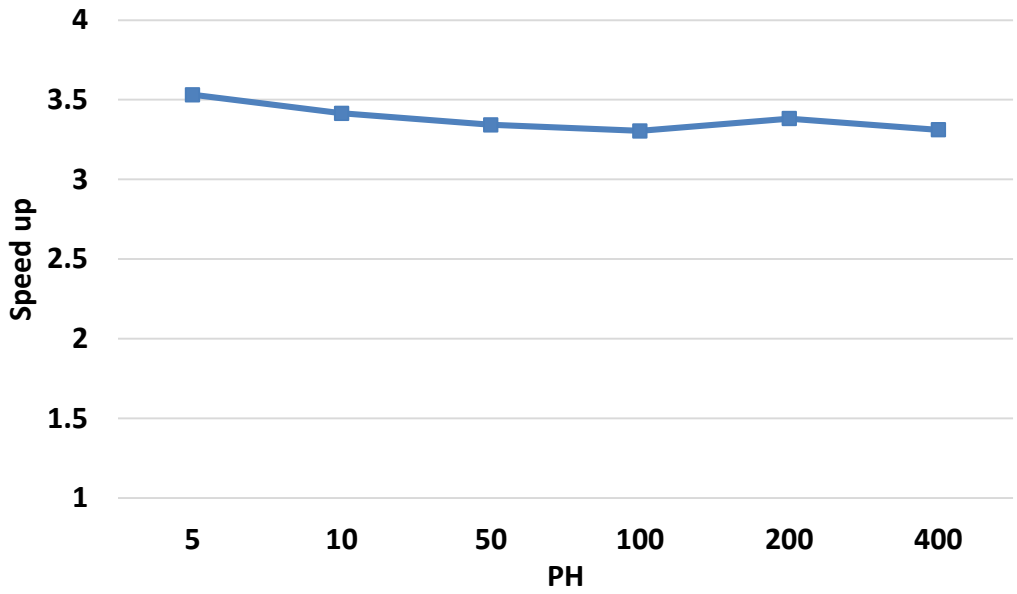

Figure 36. Speedup of GPU_PH with PH={5..400} and Bucket Size = 1024 over GPU_BIG with Bucket Size = {5..400}×1024 with normal distribution of keys.

As shown in Figure 37, both GPU_PH and GPU_BIG scale linearly with increase in number of parallel hash tables or number of hash buckets. However, TA functions ability to reduce contention results in consistent higher throughput for GPU_PH.

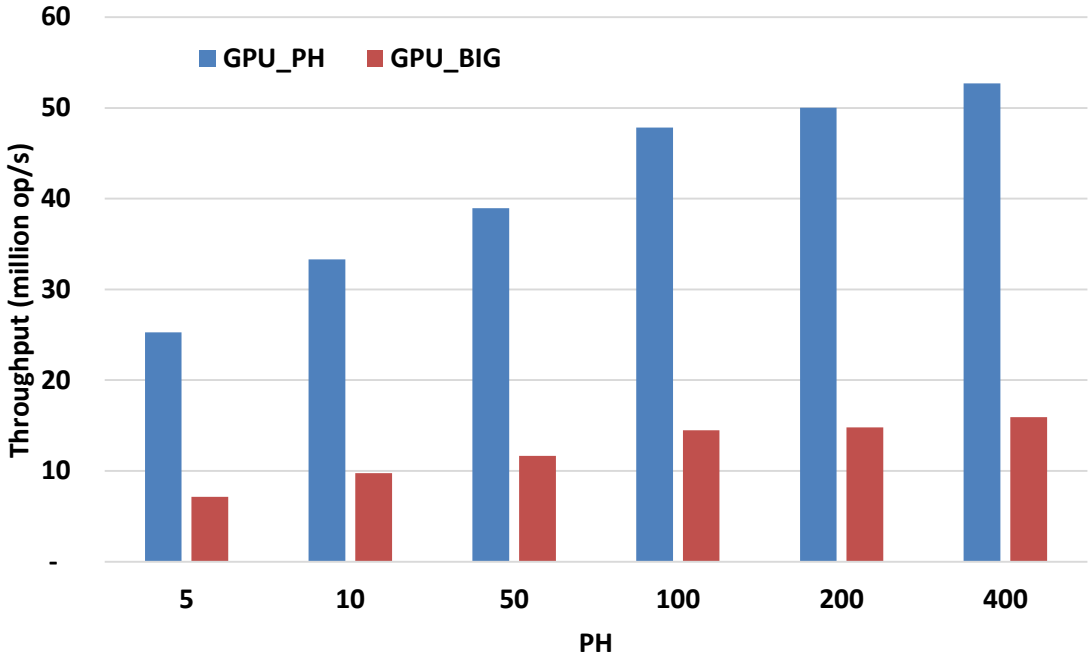

Figure 37. Throughput of GPU_PH with PH={5..400} and Bucket Size = 1024 over GPU_BIG with Bucket Size = {5..400}×1024.

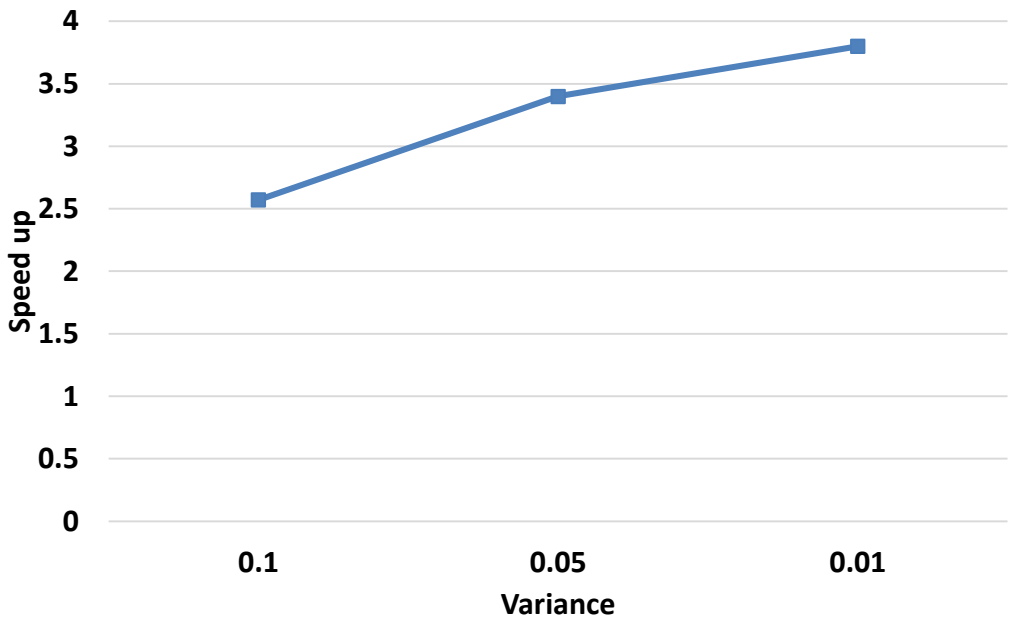

Figure 38. Speed up of GPU_PH with PH=50 and Bucket Size = 1024 over GPU_BIG with Bucket Size = 50×1024 with respect to change in Variance in normal distribution of keys.

We also compare GPU_PH and GPU_BIG by changing the variance in normal distribution of keys. In Figure 38, it is shown that by reducing the variance in normal distribution of keys GPU_PH shows higher

speedup over GPU_BIG. This also confirms earlier observation that a separate table assignment function reduces contentions between operations more efficiently. We only show the comparison for combinations with 33% Search, 33% Insert, 33% Delete, however, all combinations show similar results.

4.6.2.7 Bulk Execution Model

In Figure 39, the effect of batch size on overall performance is shown. Here we experiment with a fixed total load of 2^{20} operations that is executed on GPU with different batch sizes varying from 2^{12} to 2^{18}. As we increase the batch size, the overall execution time is reduced. Note that as previously mentioned, the selection of batch size also depends on the response time requirement of applications and the frequency of incoming hash table workload. Hence, the lower performance of smaller batches may be preferred if application has tighter demand on response time, or lower incoming workload rate.

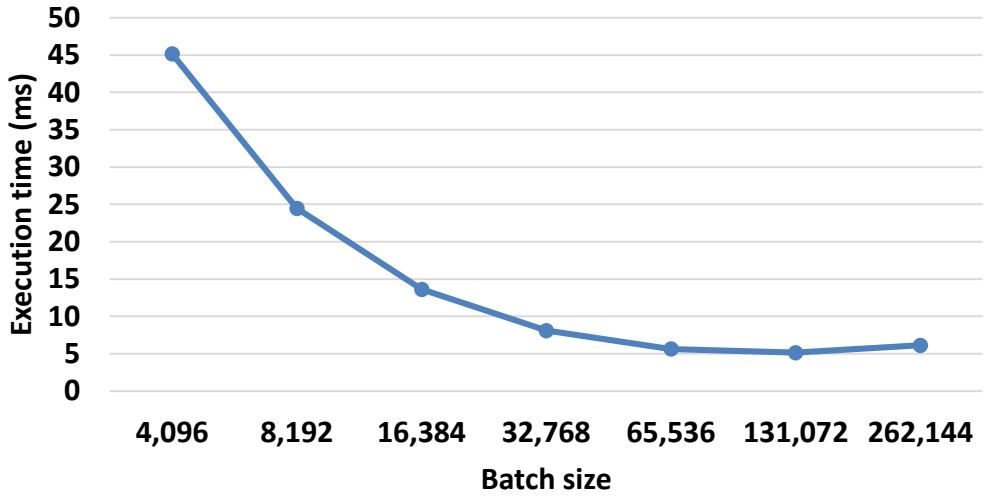

Figure 39. Effect of bulk execution model and batch size on overall performance of 1 million combined hash operations.

4.6.2.8 CPU to GPU Data Transfer

In this section, we demonstrate the CPU to GPU communication time in comparison to the GPU computation for a batch of hash table operations. In Figure 40, it is shown that in smaller batch sizes the data transfer time is almost the same as the computation time on GPU, and as we increase the batch size the GPU computation time well dominates the data transfer time by an order of magnitude. This provides

the opportunity to overlap the data transfer of the next batch with the computation of the current batch on GPU. Therefore, we pay for the data transfer time only for the first batch and the communication time is hidden. Hence, we do not consider the data transfer time in our benchmark timings.

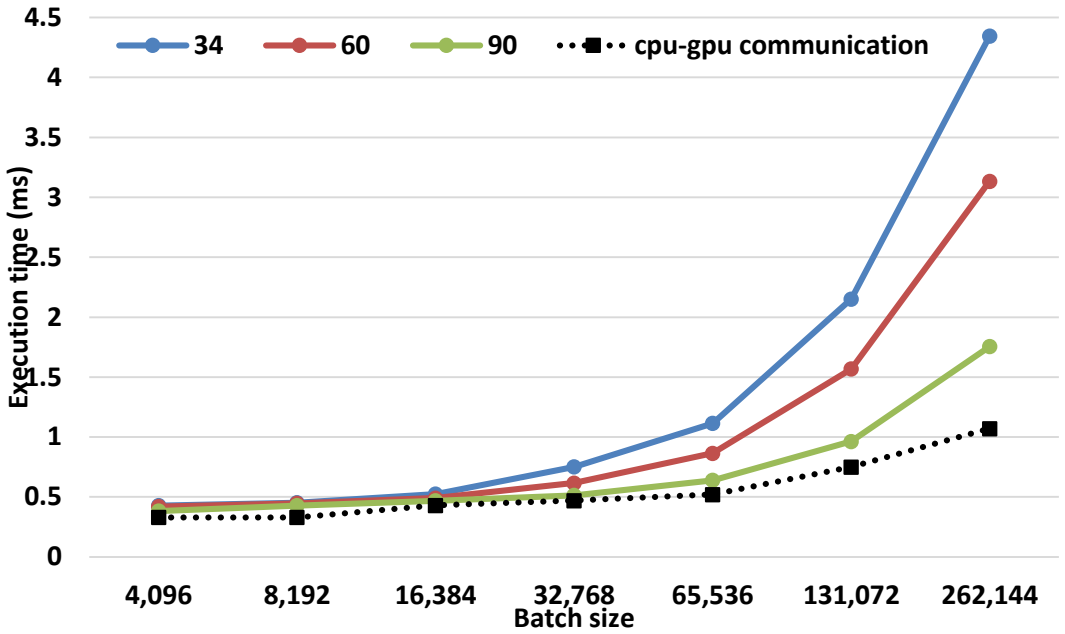

Figure 40. Execution time of different workload batch sizes and their corresponding CPU/GPU data transfer time. The legend shows search percentage of batch.

4.6.2.9 Applications

To emulate real world properties of applications and input data, we used collected data form two devices [Vahdatpour09, Vahdatpour10] used for remote health monitoring. The device data is the status (combination of activity, orientation, motions) and location of the device users. In an application, hash table is used to store and retrieve time slots where device users have certain status and location. As shown in Figure 41, in the first application, input data is combination of two normal distributions with standard deviation .05 and .1. In the second device, the data is normal distribution with standard deviation = .58. The first application has soft real time deadlines to detect emergency conditions; hence, we used batches of 4096 data point and achieved 25X speed up over GPU_BASIC. In the second application, data processing is done offline; hence, we used batches of 262K data points to maximize speedup and achieved 94X speedup.

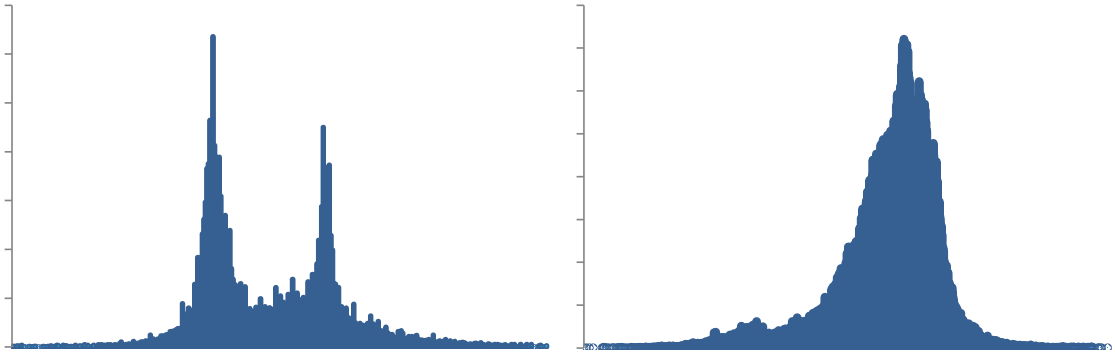

Figure 41. Histogram of data in the two motivational applications.

4.7 Conclusion

In this chapter we introduced parallel hash table and used it to reduce the contention on the shared objects in lock-free hash tables. We showed that the contention amongst threads is exaggerated in a many-core processor execution model (i.e. GPU) which makes the current CAS-abased lock-free list-based set algorithm unsuitable for many-core processor architectures. To leverage massive compute capability of GPU, we changed the underlying structure of hash table which resulted in an order of magnitude improvement in performance. This change in structure reduced the conflicts and increased parallelism in thread execution. With combined Search, Insert, and Delete workloads, we achieve 5X-27X improvement over the counterpart multi-thread CPU implementation. We also reach more than 25X improvement over the basic GPU implementation. The impact of data distribution was also studied. By emphasizing that distribution of data can significantly change the throughput, we showed how our technique is especially profitable in non-balanced data distribution scenarios.

CHAPTER 5

A Memory Optimization for Scratchpad Memory in GPUs

5.1 Overview

Modern high-performance computer architectures have increasing number of on-chip processing elements. Architects must ensure that memory bandwidth and latency are also optimized to exploit the full benefits of the available computational resources. Utilizing cache hierarchy has been the traditional way to alleviate the memory bottleneck [Kandemir]. In contrast, various modern parallel architectures such as NVIDIA G80 [Nikolls] and IBM Cell [Johns] utilize fast explicitly managed on-chip memories, often referred to as scratchpad memories, in addition to slower off-chip memory in the system to hide the memory latencies [Kandemir]. Scratchpad memories are limited in size since minimization of on-chip memories is important in reduction of manufacturing cost [Zhu].

The introduction of the IBM Cell processor with software-managed per-core memory (local store) led to the development of techniques for utilizing that memory. However, because of the architectural differences between Cell processor and NVIDIA G80, management of the software-managed on-chip

memory (shared memory) in NVIDIA G80 architecture has to be specifically studied, and the effect of imposed overheads has to be evaluated based on the architectural organization of G80. In the NVIDIA G80 architecture, shared memory is partitioned among up to 512 thread blocks that are assigned to the same multiprocessor at run-time. The data in shared memory can be shared among all threads in a thread block, enabling inter-thread data reuse. This is in contrast to single thread access to Cells local store. Moreover, in G80, an incremental increase in the usage of shared memory per thread can result in a substantial decrease in the number of threads that can be simultaneously executed and thus significantly reducing the parallelism. Current G80 architecture offers limited resources (e.g. shared memory) available to each multiprocessor, and conversely, demand for availability of massive number of threads to achieve maximum performance. The limited size of fast-access shared memory available to each multiprocessor and its considerable impact on reducing the parallelism motivates us to develop a method to minimize the usage of shared on-chip memory space in G80. This method should specifically be designed for the properties of the shared memory within the G80 architecture.

In response to this challenge, we propose a memory optimization method, which assists in increasing parallelism in applications with high data dependencies by minimizing the usage of shared on-chip memory (scratchpad memory) and increasing each multiprocessors utilization (occupancy) in the G80 architecture. We conducted a set of experiments on our image processing benchmark suite in medical imaging domain as a source for real-life and data-intensive applications.

5.2 Memory Optimization

Global memory bandwidth can limit the throughput of the system as described earlier. In G80, alleviating the pressure on global memory bandwidth generally involves using additional registers and shared memory to reuse data, which in turn can limit the number of simultaneously executing threads. Balancing the usage of these resources is often non-intuitive and some applications will run into resource limits. This section presents a memory optimization technique in the G80 architecture to further alleviate the constraints of using shared memory for data-intensive applications in the G80 architecture.

Data-intensive applications that have high usage of shared memory in their CUDA implementations are limited by low SM occupancy when ported to the G80 architecture. In the G80, as each threads resource usage (e.g. shared memory and register count) increases, the total number of threads that can occupy the SM decreases, which results in reduction of SM occupancy that results in significant performance loss. Occasionally this decrease in thread count occurs in a dramatic fashion because threads are assigned to an SM at the granularity of thread blocks; this makes the situation very critical in a sense that a small increase in threads resources could have a dramatic effect on performance. For example, consider a data-intensive application with 128 threads per block and 8KB of shared memory per thread block. This application can schedule 2 thread blocks on each SM. However, if each threads shared memory usage increases from 8KB to 10KB (an increase of 25%), the number of blocks per SM will decrease from 2 to 1 (a 50% decrease). In other words, the G80 can only assign one thread block (128 threads) to an SM because a second block would increase the amount of shared memory usage above the SM limit. This results in significant performance reduction. Therefore, allocating memory space in limited scratchpad-like memories in such data-intensive applications is highly costly in modern parallel architectures such as G80.

In order to maximize the performance, it is better to allow for two or more thread blocks to simultaneously execute. For this to happen, not only should there be at least twice as many thread blocks as there are multiprocessors in the device, but also the amount of allocated shared memory per thread block should be at most half the total amount of shared memory available per multiprocessor [Nvidia]. Therefore, it is crucial to have a mechanism to minimize the usage of shared memory. We are aiming at achieving this by reusing allocated memory spaces and avoiding the allocation of further unnecessary resources for each thread block with the goal of maximizing the performance. In our vision, by having this transformation, developers will provide a straightforward implementation of the kernel code that utilizes shared memory, and depend on this transformation to optimize the memory usage. In the following section, we propose a memory reuse scheme particularly designed for scratchpad memory in

GPU architectures. In Section 5.3, we demonstrated the effectiveness of our approach on our image processing benchmark suite.

5.2.1 Memory Reuse Scheme

Consider a motivational simple example of the shared memory reuse in Figure 42(a), where memory blocks `sA`, `sB` and `sC` are shared among all threads in a thread block, and need to be allocated to certain memory areas in shared memory. A naive allocation, as performed by almost all the software compilers, is to map each of the blocks to distinct memory locations, as shown in Figure 42(b). A careful inspection of the program reveals that memory block `sA` and memory block `sC` can in fact be shared, leading to the allocation in Figure 42(d), which can be obtained by the modified program in Figure 42(c). We refer to the `sA_shared_sC` memory block as the "reused memory block" in our scheme.

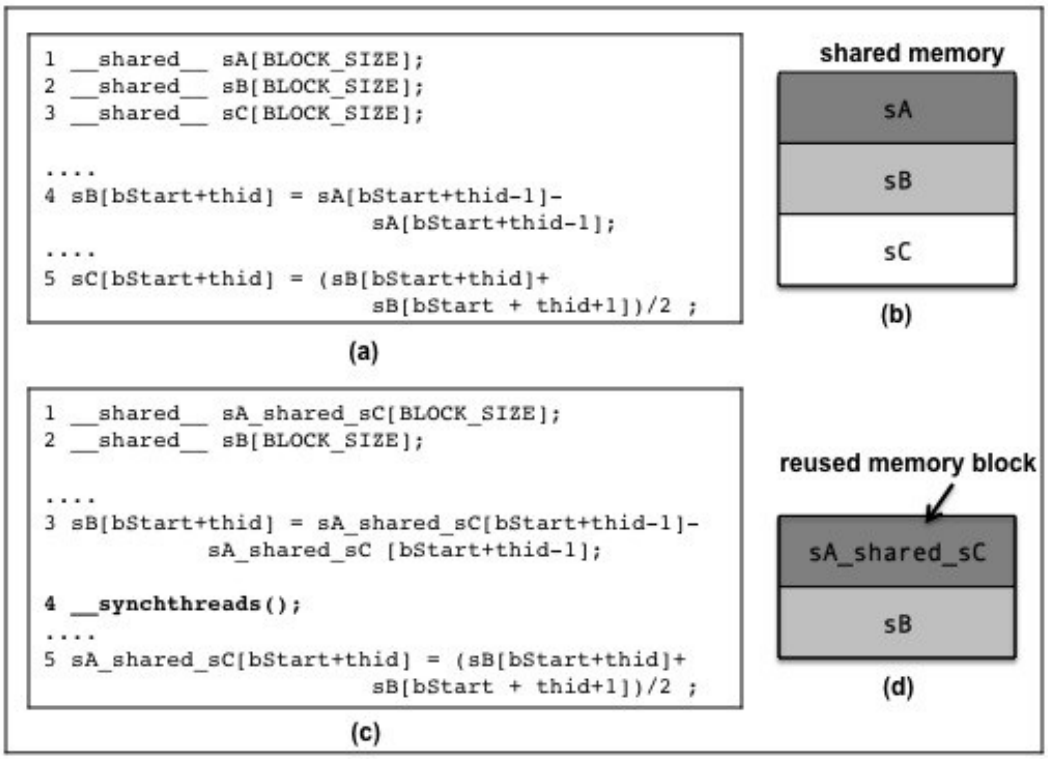

Figure 42. A motivational example in CUDA

Our ultimate goal in the memory reuse scheme is to minimize the usage of memory space without

changing the structure of the program. One might argue that the programmers should identify such opportunities of memory reuse and enforce them manually in the program. We believe this requirement is unrealistic for the following reasons: (1) the primary goal of a programmer is to specify functionality; for a programmer, readability and maintainability has higher priority than implementation details; (2) as the application complexity increases (i.e. consisting of data structures with different sizes) automated optimization tools have a better chance to find an optimal solution than the programmers; (3) in a multithreaded context it is harder for the programmer to enforce the memory sharing while maintaining the correctness of the program; (4) eventually productivity of programmers will increase by taking the burden of memory management off their shoulder.

The idea of having a memory reuse scheme is very similar to the register allocation problem in traditional compiler optimization [Briggs]. The goal in the memory reuse problem is to achieve memory minimization by discovering the chances of memory reuse with the goal of maximizing the application performance. We propose a solution for the memory reuse problem based on graph coloring described in the following section.

5.2.2 Solution Approach

Reuse Pattern As described in previous sections, our goal is to achieve memory minimization by discovering the chances of memory reuse. Since our solution is proposed for optimization in the GPU shared memory space, the desired reuse pattern in applications should be suited to the architecture of GPUs and shared memory in particular. In the G80 architecture, shared memory is shared among all threads in a thread block and we intend to leverage a reuse pattern to reuse shared memory spaces across all threads in the thread block as illustrated in the example of Figure 7. Therefore, execution of all threads in the thread block needs to be synchronized to coordinate shared memory accesses to provide means of correct and safe memory reuse, as illustrated by use of synchthreads primitive in Figure 7(c), line 4. This is due to the fact that each active thread block on a multiprocessor is split into SIMD groups of threads (warps) executed in an SIMD fashion, and all the SIMD groups from all active thread blocks on the

multiprocessor are time-sliced. Therefore, there is no explicit guaranteed ordering in the accesses to shared memory in different SIMD groups in a thread block. As a result, in order to make memory reuse a viable solution in such SIMD architecture, it is crucial to enforce synchronization after or before the points of reuse. We define points of reuse as any use or definition point of a reused memory block in the program. For example lines 3 and 5 in Figure 7(c).

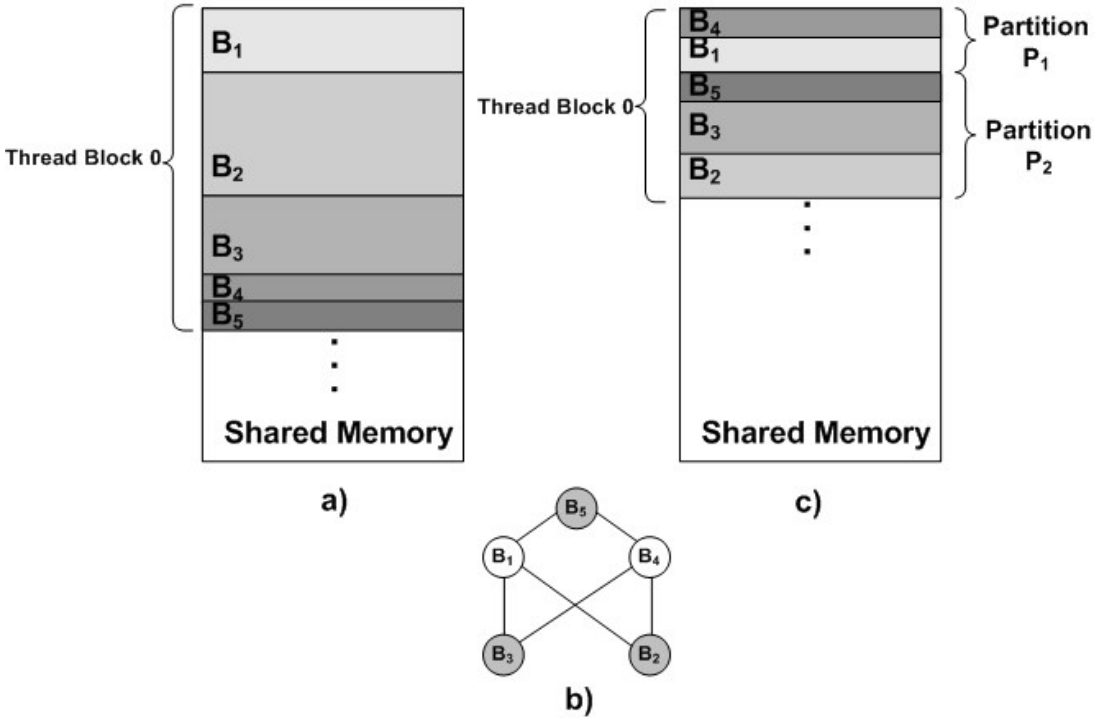

Figure 43. A memory reuse scheme for shared memory

Previous methods [Kandemir, Yang], discussed in Section 5.1, were not designed for scratchpad memories that are shared among i.e. 512 threads; therefore, synchronization of threads for coordinating the accesses to shared memory was not an issue in those studies. In our problem, memory blocks are shared among all threads in the thread block; thus, coordination of memory accesses according to the underlying threading model is critical, and needs to be explicitly added to the kernel code at points of reuse an example of this case is demonstrated in Figure 7(c). In Section 5.3, we evaluate the effect of synchronization overhead on the overall performance results.

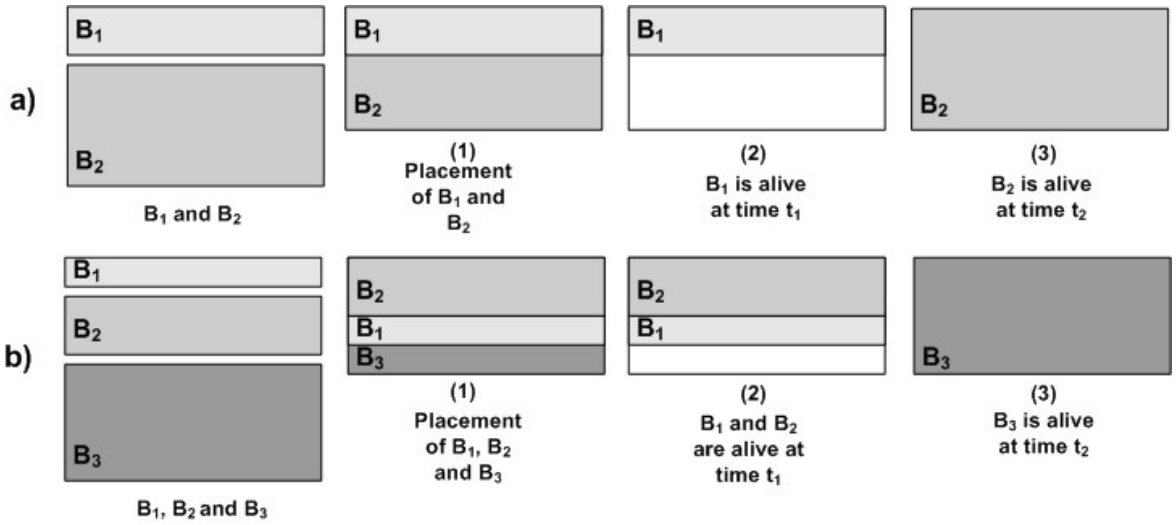

Figure 44. Configuration of memory blocks Bi in their memory partitions

Discovering the Chances of Memory Reuse Regardless of the desired reuse pattern in our scheme, we describe our solution to discovering the chances of memory reuse in this section. In our proposed solution, we define a memory block to be an arbitrary sized array of data shared among all threads in a thread block. A memory partition is a partition of shared memory to which one or more memory blocks will be assigned. Figure 43 demonstrates our memory reuse scheme. Figure 43(a) shows the placement of memory blocks without the memory reuse scheme; Figure 43(b) depicts the live range conflicts between memory blocks B_1::B_5 in the given interference graph in which two nodes each representing memory blocks are connected if their live ranges overlap. Figure 43(c) demonstrates the reduction in memory space in each thread block by leveraging the memory reuse scheme based on the given interference graph. As an example, there is no edge between memory block B_4 and B_1 in the interference graph, and hence, B_4 and B_1 are both assigned to memory partition P_1. As it is shown in the figure, B_4 is placed inside B_1's memory space in order to reuse available memory spaces. It should be noted that memory partitions should not necessarily be of the same dimension.

The difference between the proposed memory reuse technique against the well-known register allocation problem is that: inputs to the memory reuse problem are arbitrary sized memory blocks. This makes the problem different than register allocation in the sense that finding the optimal sizes of memory

partitions are now added to the problem, which makes it more of a placement problem as seen in the bin-packing problem [Vazirani].

There are two possible configurations for the placement of memory blocks in memory partitions in the memory reuse scheme. In the first configuration shown in Figure 44(a), only one of the memory blocks in a memory partition is alive at a time. In Figure 44(a) memory blocks B_1 and B_2 are sharing a memory partition where B_1 is placed inside B_2 space; B_1 is showed to be alive at time t_1 and B_2 is alive at time t_2. In the second configuration, it is possible to have more than one memory blocks that are simultaneously alive in a memory partition as depicted in Figure 44(b) in which B_1 and B_2 are alive at time t_1. In this placement B_1 and B_2 are both placed with no physical overlap inside B_3 space. Therefore, in this configuration B_1 and B_2 may have conflict in their lifetimes, but neither should have conflicts with B_3.

In order to relax the problem, we solve the problem assuming only one memory block b_i from memory partition P_j can be alive at a time. Therefore, our expected configuration for the relaxed problem is as illustrated in Figure 44(a), which our proposed algorithm is based upon. Thus, two memory blocks that are simultaneously alive cannot share a partition. Therefore, our ultimate goal is to assign memory blocks with non-overlapping lifetimes to memory partitions so that usage of memory space is minimized without changing the structure of the program. Given a program with arbitrary sized memory blocks b_i, our goal is to allocate memory partitions P_j in shared memory to fit as many memory blocks with non-overlapping lifetimes as possible in P_j.

Input:
$\{b_1, b_2, ..., b_n\}$: existing allocated memory blocks in shared memory

Output:
K: number of memory partitions to be created
$\{P_1, P_2, ..., P_k\}$: memory partitions to be allocated and their related meta-data

ReuseMemory($\{b_i, P_i\}$
Determine the live ranges of all memory blocks b_i
Build the Interference Graph $G(V, E)$
$V=\{b_1, b_2, ..., b_n\}$
Undirected edge connects two memory blocks
(b_i , b_j), if live ranges of b_i and b_j overlap in time
$B' \leftarrow \{b_0\}$

for all nodes b_i adjacent to nodes in B'
{
$B' \leftarrow B' \cup \{b_j\}$
Assign color K_j such that adjacent colored nodes has
different colors
$K = Max(K_j, K)$
$P_j \leftarrow P_j \cup \{b_j\}$
$Partition\_Size(P_j) = Max(Partition\_Size(P_j), Block\_Size(b_j))$
}

Allocate $P = \{P_1, P_2, ..., P_k\}$ in shared memory
{Physically assign members of P to the corresponding memory partition}

Figure 45. Pseudo code for memory reuse

Algorithm in Figure 45 is proposed as a solution for memory reuse problem. In Line 3, the live ranges of memory blocks are determined to identify memory blocks with non-overlapping lifetimes. Line 4, constructs an interference graph based on the output of Line 1 in which two nodes each representing memory blocks are connected if their live ranges overlap (Figure 43(b)). Steps 6-8 cluster the memory blocks with non-overlapping lifetimes by coloring the interference graph; memory blocks assigned to the same memory partition are represented by the same color after coloring the interference graph. By adding each color Kj, a new memory partition Pj is added to the solution, and memory blocks colored with color Kj are assigned to P_j in Line 10. In Line 9, the number of memory partitions K is calculated. Size of

memory partitions are calculated based on the maximum size of assigned memory blocks based on the configuration described in Figure 44(a) (Line 11).

Note that this algorithm, arbitrarily selects memory blocks to be added to the solution, and therefore, the final solution may not be optimal. However, we show the effectiveness of this algorithm through experimental evaluations. It is worth mentioning that finding the optimal size of memory partitions and placing memory blocks in a memory partition when having more than one live memory block per memory partition at a time is NP-Complete, which can be proved by reducing this problem to the bin-packing problem [Vazirani], which is not discussed in this study.

5.3 Experimental Results

This section presents the experimental results of manual evaluation of our memory optimization scheme introduced in Section 5.2. We based our experiments on our image processing bench-mark suite as a source of real-life and data-intensive applications to demonstrate how real-life applications can benefit from the introduced optimization method. These real-life applications are more interesting and useful than micro benchmarks because of their larger code sizes and data sets, and variety of instructions and control flow [Ryoo].

We used CUDA version 2.0 for our experiments. Experiments were performed on Core2 Duo running at 2.33 GHz with 8 GB of main memory and NVIDIA Quadro FX 5600 as a commodity GPU.

5.3.1 Benchmarks

Our benchmark suite consists of denoising, segmentation and registration algorithms particularly designed for medical imaging. In this section we describe the most important characteristics of the three benchmarks that are mostly relevant to our experimental evaluations of the memory reuse scheme. The denoising benchmark is a local nonlinear iterative denoising algorithm called Total Variation Regularization [Christiansen]; the segmentation benchmark is a curvature-based segmentation algorithm called Active Contour [Chan] based on geometric PDEs, and the registration benchmark is based on Biharmonic Regularization [Fischer].

Measurement of curvature exists in both denoising and segmentation benchmarks, and has high usage of shared memory in its CUDA implementation; curvature is a common measurement in image processing and computer vision algorithms [Christiansen, Chan, Kindlmann, Wang]. The denoising benchmark uses a 3D computation of curvature (Curvature3D), and the segmentation benchmark uses a 2D calculation of the curvature (Curvature2D). Detailed explanation on the measurement of GPU-based Curvature3D is presented in [Moazeni09]. We implement Curvature3D and Curvature2D as independent kernels. The Curvature kernel consists of three measurements: (1) partial derivatives, (2) gradient norm and (3) divergence where measurements at each step have dependency to the previous measurements. For example, measurement of partial derivates is dependent on two neighboring pixel values; measurement of gradient norm is dependent on four neigh-boring pixels corresponding partial derivates; measurement of the divergence is dependent on two neighboring pixels corresponding gradient norms [Christiansen]. Therefore, there is significant data reuse; thus, shared memory is utilized for storing these arrays. The pixel data is loaded from global memory collaboratively by each thread, where accesses to global memory are coalesced. It is important to note that the computation of Curvature3D is heavier than Curvature2D and involves higher synchronization overhead. The preferred thread block size is 16×16 for both implementations and the SM occupancy is 66%.

The registration benchmark consists of two computational steps, which update a displacement array in vertical and horizontal directions within each iteration; the two displacement arrays are placed in shared memory, as there is significant data reuse in computation of each pixels displacement from displacement of neighboring pixels in the previous iteration. The pixel data is loaded from texture memory in order to utilize the on-chip texture cache. The preferred thread block size is 16×24 in our implementation and the SM occupancy is 38%.

In all experiments, we consider each benchmark in its preferred configuration; for instance, in the denoising and the segmentation benchmarks, the thread block size is set to be 16×16, while in the registration benchmark, it is set to be 16×24.

5.3.2 Evaluation of the Memory Reuse Scheme

We performed a set of experiments applying our memory reuse scheme manually on our image processing benchmark suite. We compare the results of straightforward GPU-based implementations of the three benchmarks with our GPU-based implementations optimized for shared memory space based on our memory optimization technique.

Table 7. Shared memory saving for the image processing benchmark

Benchmark	Size w/o optimization (byte)	Size w/t optimization (byte)	Memory Savings
Denoising	6220	3916	37%
Segmentation	6220	3916	37%
Registration	13596	13596	0%

Table 7 shows the memory savings we can achieve for the benchmark suite. The first column gives the shared memory usage per thread block without optimization. The second column gives the shared memory usage per thread block after applying the memory optimization. The third column gives the percentage of memory saving we are able to achieve per thread block. For denoising and segmentation benchmarks the achieved memory saving is 37% in the optimized implementations, allowing the number of active thread blocks to increase from 2 to 3 (increasing the number of active threads from 512 to 768) on each multiprocessor, which increases the multiprocessor occupancy from 66% to 100%. This increase in the number of active threads increases the parallelism, which results in increasing the performance.

In the registration benchmark we cannot achieve memory saving directly from this optimization technique. However, by changing the order of computations we can apply the memory reuse technique to this benchmark as well. In spite of this, we ignore this benchmark for optimization since reordering the computations is out of the scope of this study. Nevertheless, it is worth pointing out the existing potential.

We observe that benchmarks that involve multiple steps of dependent processing benefit from our memory reuse technique to the most. For example, measuring the curvature constitutes the measurement of partial derivates, gradient norm - dependent on partial derivates - and finally, measurement of the divergence - dependent on gradient norm [Christiansen].

Table 8. GPU execution time for the curvature kernel

Data size	Exec time w/o optimization (μsec)	Exec time w/t optimization (μsec)	Percentage
16^3	25.2	19.53	22.5%
32^3	74.89	60.96	18.6%
64^3	390.7	319.59	18.2%
128^3	3033.3	2532.85	16.5%

Table 9. GPU execution time for segmentation

Data size	Exec time w/o optimization (μsec)	Exec time w/t optimization (μsec)	Percentage
64×64	35.3	22.14	37%
128×128	39	27.39	30%
256×256	70.5	61.2	13%
512×512	185	181.76	2%

Table 8 and Table 9 show the execution time before and after applying the optimization to denoising and segmentation benchmarks. In both tables, the first column gives the GPU execution time without optimization. The second column gives the GPU execution time after applying the memory optimization. The third column shows the reduction of GPU execution time in percentage. It is observed that by increase in data size, the performance increase is reduced. This demonstrates that high multiprocessor occupancy has less effect on performance as the load is increased on GPU. We observe that our optimization technique maintains more stable results particularly in the denoising benchmark with higher computational load and higher synchronization overhead compared to the segmentation benchmark. Particularly in the denoising benchmark, it is also observed that although there is a synchronization overhead involved in our optimization scheme, the overall performance results are promising.

5.4 Conclusion

In this chapter, we proposed a memory reuse scheme to minimize the usage of shared memory space in applications with high data dependencies and increasing the parallelism as a result. In the G80, alleviating the pressure on global memory bandwidth generally involves using additional registers and shared memory to reuse data, which in turn can limit the number of simultaneously executing threads, which significantly decreases the application performance. Balancing the usage of these resources is often non-intuitive and some applications will run into resource limits. Therefore, our memory optimization technique in the G80 architecture further alleviates the constraints of using shared memory for applications with high data dependencies in the G80 architecture. We evaluated our proposed memory optimization technique by a set of experiments on our image processing benchmark suite in medical imaging domain using NVIDIA Quadro FX 5600 and CUDA. Implementations based on our proposed memory reuse scheme showed up to 22.5% increase in speedup over their naïve GPU implementations.

CHAPTER 6

Accelerating Total Variation Regularization for Matrix-Valued Images on GPUs

6.1 Overview

A large number of medical imaging algorithms, including all the algorithms in the medical imaging pipeline (i.e. denoising, registration and segmentation) can benefit significantly from accelerators such as GPUs. During the last decade, a new magnetic resonance modality called diffusion tensor imaging (DTI) has been extensively studied [Basser, Bihan, Westin, Mori99, Mori02, Bammer]. The DTI images are matrix valued. In each voxel of the imaging domain, a diffusion tensor (i.e. diffusion matrix) D is constructed based on a series of K direction-specific MR measurements. All measurements contain noise, which degrades the accuracy of the estimated tensor. Compared with conventional MR, direction-sensitive acquisition has a lower signal-to-noise ratio (SNR). There are several ways to increase the accuracy of estimated tensors. The most intuitive way is to make an average of a series of repeated measurements. Alternatively, the number of gradient directions can be increased. An obvious disadvantage of both of these approaches is the increased scanner time. Best way to improve the quality of the tensor is by post-processing the data.

Hence, from the developments in DTI, there is a need for robust regularization (denoising) methods for matrix-valued images that is computationally fast. One of the state-of-the art methods for regularization of tensor-valued images is proposed in [Christiansen] as a Variational method [Lysaker, Chan, Weickert] and a natural extension of the color Total Variation model proposed by Rudin et al. [Rudin]. However, the post-processing problem is only made more computationally difficult when considering multi-valued imaging data, such as DTI or multi-channel acquisitions, wherein each voxel is a feature vector of 6-100 dimensions. In this chapter, we accelerate the Total Variation regularization algorithm [Christiansen] for DTI images. To the best of our knowledge, there have been no e orts to accelerate such fundamental algorithms for DTI images in the GPGPU community before.

For this regularization algorithm to be viable for clinical settings, significant and low-cost computational acceleration is required. We found that regularization algorithms for DTI images can significantly benefit from the advances in the architecture of GPU. Solving partial differential equations in Total Variation model poses extensive synchronization on the GPU-based Implementation of TV regularization. Hence, in this chapter, we analyzed the effect of synchronization by comparing our GPU-based implementation to a secondary approach that eliminates synchronization by dividing all computations into independent sub-blocks. Thereby, we compared the effect of excessive synchronization on our primary approach against the effect of excessive computational workload and memory load in the secondary approach that is imposed on each thread by eliminating the synchronization.

6.2 Total Variation Regularization for matrix-valued images

Image processing methods using Variational calculus and partial differential equations (PDEs) have been popular for a long time in the image processing research community. Among popular PDE methods are the Total Variation method introduced by Rudin et al. [Rudin] and various methods related to this [Nvidia, Asanovic, Christiansen]. Many of these methods were extensively studied for scalar-valued (gray-scale) images and were later generalized to vector-valued (color) images.

Emerging imaging modalities such as matrix-valued images, also, require robust image processing methods. However, when considering multi-valued imaging data, the computational complexity of these algorithms becomes significantly higher and makes them impractical for clinical purposes. One of the most fundamental image processing algorithms is denoising that is usually required as a pre-processing step for registration and segmentation of medical images. Thus, acceleration of such a fundamental algorithm for matrix-valued images such as DTI images has an immediate impact on medical imaging community. This study shows that regularization of matrix-valued images becomes viable in clinical settings when accelerated on GPUs.

We implemented the Total Variation regularization of [Christiansen] specifically for DTI images. This algorithm finds the solution to the following minimization problem for each voxel (each voxel is a feature vector of size 6)

$$min_{l_{kl}}\{\sqrt{\underbrace{\sum_{ij} TV\left[d_{ij}(l_{kl})\right]^2}_{R(u)} + \frac{1}{2}\underbrace{\sum_{ij}\left\|d_{ij} - \hat{d}_{ij}\right\|_2^2}_{F(y,f)}} \quad (6)$$

Where $\{kl\} = \{11,21,22,31,32,33\}$, $\hat{d}_{ij}$ denotes the elements of the tensor estimated from the noisy data, d_{ij} denotes the elements of matrix D and TV is the Total Variation norm of a matrix. Matrix D is defined as $D = LL^T$ where L is a lower triangular matrix. Consequently, the diffusion matrix is represented on the form of a Cholesky factorization.

The objective is to find the d_{ij} as the (unique) minimizer of Equation 6.

R(u) is the regularization functional and F(u,f) is the fidelity functional. The regularization term is a geometric functional measuring smoothness of the estimated solution and the fidelity term is a measure of fitness of the estimated solution.

Total Variation (TV) norm of a matrix $D \in R^3 \times R^3$ is defined as

$$TV[D] = (TV[d_{11}(l_{ij})]^2 + 2\,TV[d_{21}(l_{ij})]^2 + TV[d_{22}(l_{ij})]^2 + 2\,TV[d_{31}(l_{ij})]^2 + 2\,TV[d_{32}(l_{ij})]^2 + TV[d_{33}(l_{ij})]^2)^{1/2} \quad (7)$$

Following equation gives the abstract formulation of the problem.

$$\min_u \left\{ \mathbf{G}(u, f, \lambda) = \mathbf{R}(u) + \frac{\lambda}{2}\mathbf{F}(u, f) \right\} \quad (8)$$

The minimization problem described in this study therefore consists of five primary computations. Derivative of regularization functional R is given in the following equations:

$$\frac{\partial R}{\partial l_{kl}} = -\sum_{ij} \frac{1}{\alpha_{ij}} \underbrace{\underbrace{TV[d_{ij}}_{TV\ norm} \underbrace{\nabla . \left(\frac{\nabla d_{ij}}{|\nabla d_{ij}|}\right)}_{curvature}}_{P(x_{ij})} \frac{\partial d_{ij}}{\partial l_{kl}} \quad (9)$$

$$\alpha_{ij} = TV[D]$$

Throughout this chapter, ∇ denotes the spatial gradient, while $\nabla.$ denotes the divergence operator.

First, the algorithm computes each element of P as part of computing $\frac{\partial R}{\partial l_{kl}}$ given by,

$$P(\mathbf{x_{ij}}) = TV[d_{ij}] \nabla \cdot \left(\frac{\nabla x_{ij}}{|\nabla x_{ij}|} \right) \frac{\partial x_{ij}}{\partial l_{kl}} \quad (10)$$

Function P consists of two major parts, curvature and the TV norm. Curvature is the most computationally expensive function in the algorithm. α_{ij} is the scaling factor, which scales the result of P based on the total variation in the image. Second, the algorithm computes the gradient of regularization functional R

$$\frac{\partial \mathbf{R}}{\partial l_{kl}} = -\sum_{ij} \mathbf{P}(d_{ij}) \frac{\partial d_{ij}}{\partial l_{kl}}, \quad (11)$$

Because all six $\frac{\partial \mathrm{R}}{\partial \mathrm{l}_{\mathrm{kl}}}$ depends on values of $P(d_{ij})$, the values of P can be computed beforehand and then reused in the computation of $\frac{\partial \mathrm{R}}{\partial \mathrm{l}_{\mathrm{kl}}}$.

Third, the algorithm computes the gradient of the fidelity functional F

$$\frac{\partial \mathbf{F}}{\partial \mathrm{l}_{\mathrm{kl}}} = 2\sum_{\mathrm{ij}} (\mathrm{d}_{\mathrm{ij}} - \hat{\mathrm{d}}_{\mathrm{ij}}) \frac{\partial \mathrm{d}_{\mathrm{ij}}}{\partial \mathrm{l}_{\mathrm{kl}}} \qquad (12)$$

Forth, the algorithm combines the previous computations to compute the gradient $\frac{\partial \mathrm{G}}{\partial \mathrm{I}_{\mathrm{ij}}}$,

$$\frac{\partial \mathbf{G}}{\partial \mathrm{l}_{\mathrm{ij}}} = \frac{\partial \mathbf{R}}{\partial \mathrm{l}_{\mathrm{ij}}} + \frac{\partial \mathbf{F}}{\partial \mathrm{l}_{\mathrm{ij}}} \qquad \{\mathrm{ij}\} \in \{11,21,22,31,32,33\}. \qquad (13)$$

Finally, the algorithm can iteratively solve the Euler-Lagrange equation corresponding to the minimization problem using the steepest descent method with a fixed time step Δt. For this step, six equations are solved iteratively based on Equation 14.

$$\mathrm{d}_{\mathrm{ij}}^{\mathrm{n+1}} = \mathrm{d}_{\mathrm{ij}}^{\mathrm{n}} - \mathrm{dt} \frac{\partial \mathbf{G}^{\mathrm{n}}}{\partial \mathrm{l}_{\mathrm{ij}}} \qquad (14)$$

For each step of the algorithm computations are performed for six gradient directions *{11,21,22,31,32,33}*, which makes each step computationally intensive.

The complexity of Total Variation regularization as a denoising method for multi-valued imaging data such as MR diffusion tensor imaging or multi-channel acquisitions, far exceeds the complexity of the same methods for conventional vector-valued images, since each voxel can be a feature vector of 6-100 dimensions in multi-valued imaging. For this reason, denoising of high-resolution, three dimensional and multi-valued images have been impractical in clinical settings, despite the need for such imaging modalities. Our work demonstrates that such advanced denoising methods can be performed quickly and efficiently on modern GPUs, increasing their viability in clinical settings.

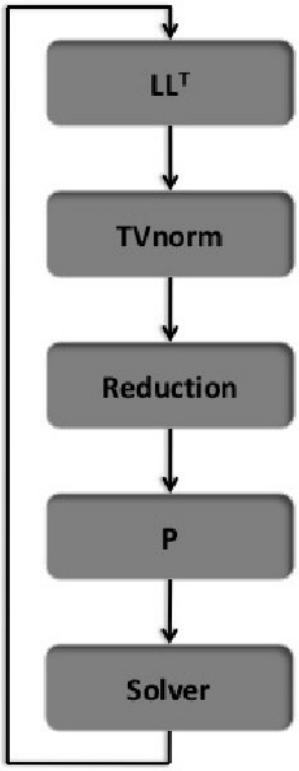

Figure 46. TV Regularization kernel control flow

6.3 GPU-based Implementation

The Total Variation (TV) regularization method for matrix-valued images described in Section 6.1 consists of five steps in its GPU-based implementations and is generally a solver that iteratively solves a minimization problem based on steepest descent method. Each step depicted in Figure 46, can be implemented as a CUDA kernel. However, data does not need to be transferred back and forth between CPU and GPU between kernel launches, hence, avoiding the overhead. We explain the kernel control flow in the following sections.

6.3.1 LL^T

After estimating the Cholesky factors L, tensor D is calculated per each voxel in the kernel LL^T. Kernel LL^T is a matrix multiplication kernel, multiplying a lower triangular matrix and its transpose, and is executed in data-parallel fashion. The output of kernel LL^T , x_{ij} is then fed to successive kernels. x_{ij} is a 3-

D matrix, which contains the element d_{ij} of the diffusion matrix D per each voxel in the image.

6.3.2 TVnorm

The kernel TV norm computes the Total Variation norm for each x_{ij} in a data-parallel fashion, which consists of computing derivatives of the 3-D image in x, y, z directions and finally calculating the norm by performing a square root operation. Computation of TV norm is followed by a global reduction operation among all thread blocks. For this, each thread block does its own share of accumulation in the shared memory, and then a global accumulation needs to be done among the single result of all thread blocks in the global memory. However, since our GPU platform does not support atomic add at this moment, we need to add an additional kernel (kernel Reduction) to our design to perform the final reduction.

6.3.3 Reduction

This kernel launches only one thread block to perform the global reduction on the set of data captured from each thread block in kernel TV norm. This step had to be added to compensate for the lack of atomic operations' support in our experimental platform.

6.3.4 P

$P(x_{ij})$ as the main function in the PDE solver is implemented as an independent kernel. Kernel P consists of two major measurements: Curvature and TV norm. TV norm is implemented as an independent kernel as described in Section 6.3.2, and its result of will be reused in kernel P.

Curvature3D: The function Curvature3D consists of three measurements:

1. spatial gradient of matrix $x_{ij}(\nabla x_{ij})$ in the x, y, z directions,
2. gradient norm ($|\nabla x_{ij}|$)
3. divergence of $\frac{\nabla d_{ij}}{|\nabla d_{ij}|}$

By finding its spatial gradient in the x, y, z directions and accumulating them as the result of

divergence. CUDA implementation of Curvature3D is depicted in Figure 45. Measurements at each step have dependency to neighboring voxels in the image. For example, measurement of spatial gradient is dependent on neighboring voxel value; measurement of gradient norm is dependent on neighboring voxels' corresponding spatial gradient; measurement of the divergence is dependent on neighboring voxels' corresponding spatial gradients and gradient norm. Therefore, all intermediate results (i.e. spatial gradient and gradient norm) need to be computed completely per neighboring voxels before proceeding to the computations in the successive steps. This requires all the threads and thread blocks to synchronize after completion of each step. However, based on the current architecture of GPUs and the insufficient synchronization primitives supported by CUDA, global synchronization of thread blocks in the GPU-based implementation of Curvature3D is impossible. This is due to the fact that global synchronization involves kernel termination and creation and it is practically impossible to per-form this synchronization after each computational step in the Curvature3D. Synchronization of threads in the same thread block is however possible but, has considerable overhead in this function.

```
__device__ float curvature3d(...)
{
   ...

   uy[index(ty,tx,tz,bStart)]=(u[index(ty-1,tx,tz,bStart)]-
u[index(ty,tx,tz,bStart)])/dy;
   ux[index(ty,tx,tz,bStart)]=(u[index(ty,tx-1,tz,bStart)]-
u[index(ty,tx,tz,bStart)])/dx;
   uz[index(ty,tx,tz,bStart)]=(u[index(ty,tx,tz-1,bStart)]-
u[index(ty,tx,tz,bStart)])/dz;

   __syncthreads();

   Ly[index(ty,tx,tz,bStart)] =
                        (uy[index(ty,tx,tz,bStart)] +
                         uy[index(ty,tx+1,tz,bStart)] +
                         uy[index(ty+1,tx,tz,bStart)] +
                         uy[index(ty+1,tx+1,tz,bStart)])/4;

   Lz[index(ty,tx,tz,bStart)] =
                        (uz[index(ty,tx,tz,bStart)] +
                         uz[index(ty,tx+1,tz,bStart)] +
                         uz[index(ty,tx,tz+1,bStart)] +
                         uz[index(ty,tx+1,tz+1,bStart)])/4;

   __syncthreads();

  normx[index(ty,tx,tz,bStart)] = ux[index(ty,tx,tz,bStart)]/
      (sqrtf( ux[index(ty,tx,tz,bStart)] *
      ux[index(ty,tx,tz,bStart)] +
            Ly[index(ty,tx,tz,bStart)] * Ly[index(ty,tx,tz,bStart)]
      +
            Lz[index(ty,tx,tz,bStart)] * Lz[index(ty,tx,tz,bStart)]
      + TINY ));

   __syncthreads();

   uxx[index(ty,tx,tz,bStart)]=
                  (normx[index(ty,tx-1,tz,bStart)]-
                  normx[index(ty,tx,tz,bStart)])/dx;

   // NOTE:same calculations repeats as above for uyy and uzz

   return( uxx[index(ty,tx,tz,bStart)]+
          uyy[index(ty,tx,tz,bStart)] +
          uzz[index(ty,tx,tz,bStart)] );
}
```

Figure 47. Curvature3D in CUDA

```
for i = 1:iter

       T = LLT(L);
      pt11=p(T(:,:,:,1,1),dx,dy,dz);
      pt21=p(T(:,:,:,2,1),dx,dy,dz);
      pt22=p(T(:,:,:,2,2),dx,dy,dz);

      // similar calculation repeats for pt31, pt32 and pt33

      drdl11=L(:,:,:,1,1).*pt11 + L(:,:,:,2,1).*pt21 + L(:,:,:,3,1).*pt31;
      drdl21=L(:,:,:,1,1).*pt21 + L(:,:,:,2,1).*pt22 + L(:,:,:,3,1).*pt32;
      drdl22=L(:,:,:,2,2).*pt22 + L(:,:,:,3,2).*pt32;

      // similar calculation repeats for drdl31, drdl32 and drdl33

      dgdl11 = 2*lambda*(T(:,:,:,1,1)-Xnoisy(:,:,:,1,1)).*L(:,:,:,1,1) + ...
                     (T(:,:,:,2,1)-Xnoisy(:,:,:,2,1)).*L(:,:,:,2,1) + ...
                     (T(:,:,:,3,1)-Xnoisy(:,:,:,3,1)).*L(:,:,:,3,1) - ...
                      2*drdl11;

      dgdl21 = 2*lambda*(T(:,:,:,2,1)-Xnoisy(:,:,:,2,1)).*L(:,:,:,1,1) + ...
                     (T(:,:,:,2,2)-Xnoisy(:,:,:,2,2)).*L(:,:,:,2,1) + ...
                     (T(:,:,:,3,2)-Xnoisy(:,:,:,3,2)).*L(:,:,:,3,1) - ...
                      2*drdl21;

      dgdl22 = 2*lambda*(T(:,:,:,2,2)-Xnoisy(:,:,:,2,2)).*L(:,:,:,2,2) + ...
                     (T(:,:,:,3,2)-Xnoisy(:,:,:,3,2)).*L(:,:,:,3,2)- ...
                      2*drdl22;

      // similar calculation repeats for dgdl31, dgdl32 and dgdl33

      L(:,:,:,1,1) = L(:,:,:,1,1)  - dt*dgdl11;
      L(:,:,:,2,1) = L(:,:,:,2,1)  - dt*dgdl21;
      L(:,:,:,2,2) = L(:,:,:,2,2)  - dt*dgdl22;

      // similar calculation repeats for 31, 32 and 33
end
```

Figure 48. Solver implementation in MATLAB.

The GPU-based implementation of the Curvature3D uses shared memory to ameliorate the effect of this global synchronization problem by overlapping the boundaries among thread blocks and creating redundant threads for computations in a given block of image. As a result, for a $N \times N \times N$ data block, we create a thread block with size $(N+2)\times(N+2)\times(N+2)$. After completing each computational step (spatial gradient, gradient norm and divergence) in Curvature3D the dimension of active threads is decreased by one. Therefore, at the beginning of the kernel P, there exists $(N+2)\times(N+2)\times(N+2)$ active threads for loading a data block of size $(N+2)\times(N+2)\times(N+2)$, while at the end there exist only $N \times N \times N$ active threads.

6.3.5 Solver

Kernel Solver is a data-parallel implementation of the PDE solver in the TV regularization algorithm. As described in Section 6.2, Equation 6 is solved iteratively by invoking this kernel to find the unique minimizer of the problem. The kernel is invoked until the number of iterations exceeds a threshold. At each iteration, the solver finds the gradient of G per voxel in multiple steps as described in Equations 9-13 to solve the Euler-Lagrange equation given by Equation 14. Results of kernel P stored in global memory is used in this kernel for computing the gradient of R based on Equation 9. Finally, x_{ij} is estimated according to the Euler-Lagrange equation given by Equation 14, based on gradient of G and the value of x_{ij} in the previous iteration. The implementation for the solver is demonstrated in MATLAB for the sake of briefness in Figure 46, which consists of function P and LL^T which are implemented as separate kernels in the CUDA implementation.

6.4 Methodology

Regularization of multi-valued images using the algorithm described in Section 6.2 imposes significant synchronization overhead to its GPU-based implementation resulting from the structure of the algorithm especially function $P(x_{ij})$, which is the most computationally expensive function in this application. We are particularly aimed at evaluating the effects of excessive synchronization in this application study. The

synchronization overhead in the Curvature3D is an interesting behavior to study in the GPGPU domain. Therefore, it is worth comparing several approaches in implementing synchronization in the Curvature3D for the sake of performance evaluations in order to learn the effect of synchronization in similar applications. This property of Curvature3D is in contrast to most of the previous GPGPU applications that are successfully ported to GPUs [Ryoo, Stone]. Therefore, in addition to GPU:GlobalSync as our primary GPU-based implementation, secondary approaches to our GPU-based implementation are demonstrated in this chapter. We consider our secondary approaches as approximate GPU-based solutions of the TV Regularization algorithm, and only present them for the sake of performance evaluations of different synchronization patterns. Therefore, they are not fully optimized. Function Curvature3D in kernel P is particularly important for our evaluation; therefore, we focus particularly on this function.

6.4.1 Primary Approach

This GPU-based implementation (GPU:GlobalSync) executes in data parallel fashion on the GPU. In this layout, each thread is responsible for computations in a single voxel. Kernel P is the most computationally intensive among other kernels. Computing the intermediate results in Curvature3D function within kernel P (i.e. spatial gradient, gradient norm, and divergence corresponding to each voxel) has substantial data reuse among threads within a thread block, therefore, placing them in shared memory hides the excessive memory latencies. Moreover, coordination among all the threads and thread blocks is necessary for consistency. In GPU:GlobalSync, global synchronization among thread blocks is achieved by overlapping the boundaries among thread blocks and redundant computation in a given block of image as elaborated in Section 6.3.4. For example, for each 6x6 data block in the image we actually create a 8x8 thread block and load 8 8 data blocks to shared memory. On the other hand, thread-level synchronization is enforced by CUDA provided synchronization primitive among threads in the thread block. Other kernels, however, do not require the same layout in terms of the work distribution among threads since no specific coordination is required among the threads nor thread blocks. All computations in kernels that will be used in successive kernels are stored in global memory, where all accesses to off-chip memory are

coalesced to conserve its bandwidth.

6.4.2 Secondary Approach

In both of our secondary approaches, since we are not mainly concerned about detailed analysis of these implementations and mainly focused on evaluating the effect of synchronization, we did not perform extensive performance tuning for our secondary implementations. Moreover, we do not follow the same kernel composition as GPU:GlobalSync described in Section 6.3. Hence, we consider only one kernel, which we refer to as Solver, and the main device functions are: function P calling Curvature3D and TV norm, and function LL^T . Both secondary approaches are approximate in the sense that the TV norm is only computed only for each thread block and therefore, no global reduction is required which eliminates the need for the Reduction kernel in this layout.

1- Eliminating Synchronization

In this GPU-based implementation (GPU.UnSync), we relax the synchronization problem in the TV regularization algorithm that existed in Curvature3D. In order to alleviate this synchronization problem, all computations are divided into independent sub-blocks (i.e. cubes for 3-D images) or sub-matrices in the image. Thus, in GPU.UnSync each thread is responsible for computations in a sub-block in contrast to GPU.GlobalSync where each thread is responsible for computations in a single voxel. In our implementation, the size of each sub-block is $2\times2\times2$. By dividing the regularization tasks into sub-blocks in the image, each sub-block (sub-image) is denoised (regularized) independent from neighboring sub-blocks, which eliminates dependency to neighboring voxels. However, the downside of GPU.UnSync is the decreased quality of the denoised image compared to the original algorithm that performs denoising at a global image-level. This approach enforces data-parallelism at a higher granularity than GPU.GlobalSync and eliminates the need for synchronization between neighboring voxels within the sub-block. However, dealing with the boundary data between threads is still a remaining challenge. We enforce padding the boundary of sub-blocks that eliminates the need to exchange boundary data between threads, and therefore, we can relax the synchronization problem between threads. Because of the high

memory load in kernel P and kernel Solver in this layout, the amount of off-chip memory latencies that can be hidden by leveraging the hardware's data transfer mechanism is limited here, because constant and texture memories are both read-only and shared memory is very limited to fit all the intermediate results. However, in order to conserve bandwidth to off-chip memory, memory coalescing is enabled as much as possible.

2- Thread-level Synchronization

This GPU-based implementation (GPU.ThreadSync) executes in data parallel fashion on the GPU. In this layout, each thread is responsible for computations in a single voxel the same as our primary (GPU.GlobalSync) implementation. Thread-level synchronization is enforced by CUDA provided synchronization primitive among threads in the thread block. On the other hand, global synchronization is still not possible in this layout.

In this layout, all the intermediate results in Curvature3D (i.e. spatial gradient and gradient norm, etc corresponding to each voxel) are stored in global memory as opposed to each thread's local memory since each thread needs to access the intermediate results computed by neighboring threads in the thread block. For the sake of performance comparison of GPU.ThreadSync with GPU.UnSync, no hardware's data transfer mechanism (e.g. utilizing shared memory) is leveraged in this layout to be consistent with GPU.UnSync; since we only want to evaluate the effect of synchronization in this study; therefore, we don't want a better memory placement option dramatically changes the result in the favor of GPU.ThreadSync. However, in order to conserve bandwidth to off-chip memory, memory coalescing is enabled as much as possible. Boundary data in the GPU.ThreadSync implementation is handled by padding the boundary of thread blocks' corresponding region that eliminates the need to exchange boundary data between thread blocks. The Kernel Solver on the other hand, exactly follows the same layout as the kernel Solver in our primary GPU-based implementations.

6.5 Experimental Results

This section presents the experimental results of accelerating the TV regularization algorithm on GPUs. In this section, we analyze different characteristics of our primary and secondary implementations. We used CUDA version 2.0 for our GPU-based implementations. Experiments were performed on Intel Core2 Duo running at 2.33 GHz with 8GB of main memory and NVIDIA Quadro FX 5600 as a commodity GPU. The CPU code is compiled under GCC with the O3 flag. Given the same input data set, the speedup is calculated by taking the wall-clock time required by the application on the CPU divided by the time required by the GPU. Times are measured after initial setup and do not include PCI-E bus transfer time. In this section we mainly analyze the Solver as the main computational kernel.

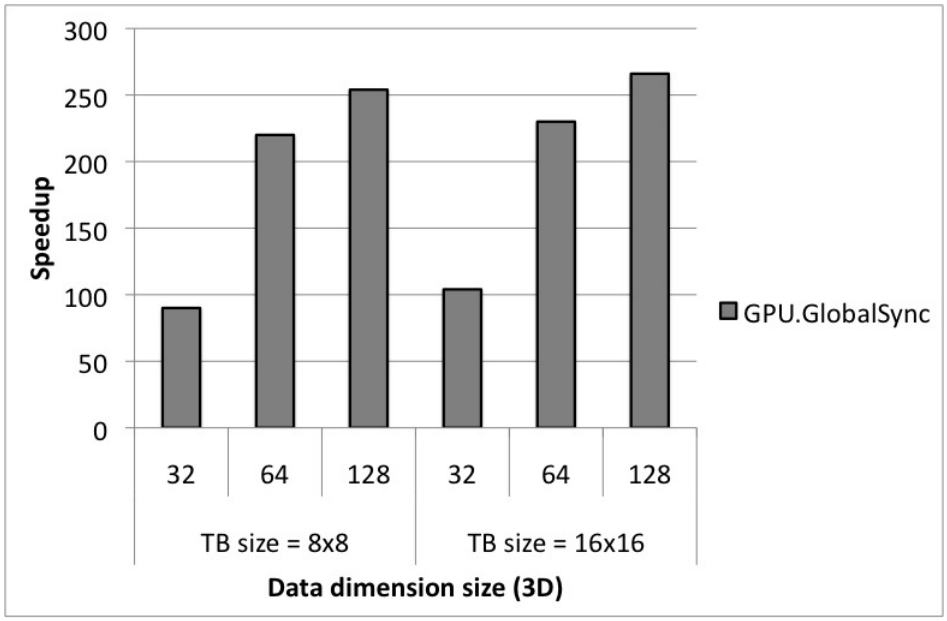

Figure 49. Kernel speedup for our primary GPU-based implementation

6.5.1 Primary Approach

As Figure 49 shows, GPU:GlobalSync achieves up to 266X speedup over the CPU version. In this version, there are 256 threads per thread block and each grid processes 128^3 matrix-valued tensors. In Kernel P , each thread uses 10 registers, and shared memory usage is 6 KB per thread block. Therefore, up to 2 thread blocks can execute on each SM simultaneously, which represents 66% utilization of the

Quadro's processor cores. In Kernel TV norm, each thread uses 10 registers, and shared memory usage is 2 KB per thread block. Therefore, up to 3 thread blocks can execute on each SM simultaneously, which represents 100% utilization of the Quadro's processor cores. Kernel Solver uses 27 registers per thread, and therefore, up to 1 thread blocks can execute on each SM simultaneously, which represents 33% utilization. Kernel Solver has high off-chip memory load. The ratio of floating-point operations to memory accesses is 0.84; therefore, knowing that the memory bandwidth is 76.8 GB/s, the upper limit on kernel Solver performance is only 16.12 GFLOPS.

Table 10. Kernel implementation performance for execution profiles

Kernel	#Calls	Execution time (ms)	Utilization	%GPU Time	Shared Mem per Thread Block (KB)	Registers per Thread
LL^T	100	258.13	100%	7%	0	9
TVnorm	600	888.21	100%	21%	2	10
Reduction	100	178.19	66%	2.86%	2	4
P	600	1800	66%	54.14%	6	10
Solver	600	557.79	33%	13.03%	0	27

Table 10 demonstrates properties of each kernel. Kernel P and kernel TV norm constitute the highest percentage of the total execution time. Kernel's execution time includes the kernel creation and termination overhead. It is observed that the highest overhead in kernel creation belongs to kernel Reduction, which only occupies one multiprocessor in the device. Kernel Solver has high usage of registers, therefore, in spite of high kernel creation and termination overhead it is essential to decompose the computation of the PDE solver into multiple kernels in this layout. Increasing the TB size from 8×8 to 16×16 does not have significant effect on overall performance as demonstrated in Figure 50. However, it is observed from Figure 51 that the increase in TB size, decreases the execution time in kernel P . This observation shows that current GPU architectures has tolerance for thread-level synchronization. On the other hand, increase in TB size increases the execution time in kernel TV norm. These observations are most evident in large data sizes. Therefore, the reverse effect of TB size increase on both kernels has somewhat diminished the effect of increasing TB size on overall performance.

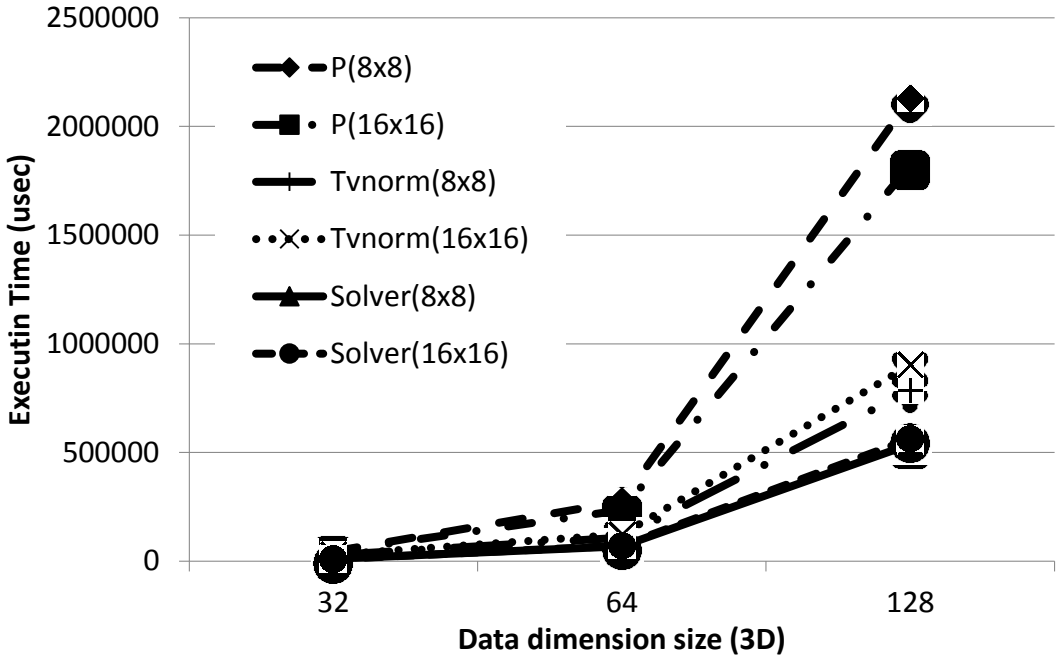

Figure 50. Kernel execution time for our primary GPU-based implementation

6.5.2 Secondary Approach

As Figure 51 shows, GPU:ThreadSync achieves up to 128X speedup over the CPU version. In this version, there are 64 threads per block and each grid processes 1283 matrix-valued tensors. Each thread uses 30 registers, and therefore, up to 8192/30=273 threads can execute on each SM simultaneously, which represents 33% utilization of the Quadro's processor cores. GPU:UnSync on the other hand achieves up to 134X speedup over the CPU version as depicted in Figure 51. It is notable that GPU:ThreadSync achieves higher speedup compare to GPU:UnSync with data size of 128^3 when TB size is 8×8. Furthermore, GPU:ThreadSync shows to scale better with the increase in data size. Figure 52 depicts the execution time of GPU:UNSync compared to GPU:ThreadSync. As it is observed from Figure 52 when TB size is increased to 16×16, by increasing the data size to 128^3 in GPU:ThreadSync, the growth in execution time has a slower slope than that of GPU.UnSync. This demonstrates that GPU architecture has better tolerance for excessive synchronization in GPU.ThreadSync rather than excessive per thread computational workload and memory load that exist in GPU.UnSync. That is due to the fact that, in GPU.UnSync, all computations in each thread are performed on 2×2×2 blocks of data as opposed

to one single voxel in GPU.ThreadSync. This is made clearer in Figure 53, where the gap between execution time of function P and the overall execution time of kernel Solver is more evident in GPU.UnSync compared to GPU.ThreadSync (computation of P is part of Solver in secondary implementations). Although execution of P is faster in GPU.UnSync, the remaining computations of Solver, takes more time to complete compared to GPU.ThreadSync duo to excessive computational workload and memory load of each thread in GPU.UnSync. Therefore, the negative effect of excessive synchronization in GPU.ThreadSync is made less evident. Overall, it is notable that the speedup achieved from GPU.UnSync is not considerably better than GPU.ThreadSync in large data sizes.

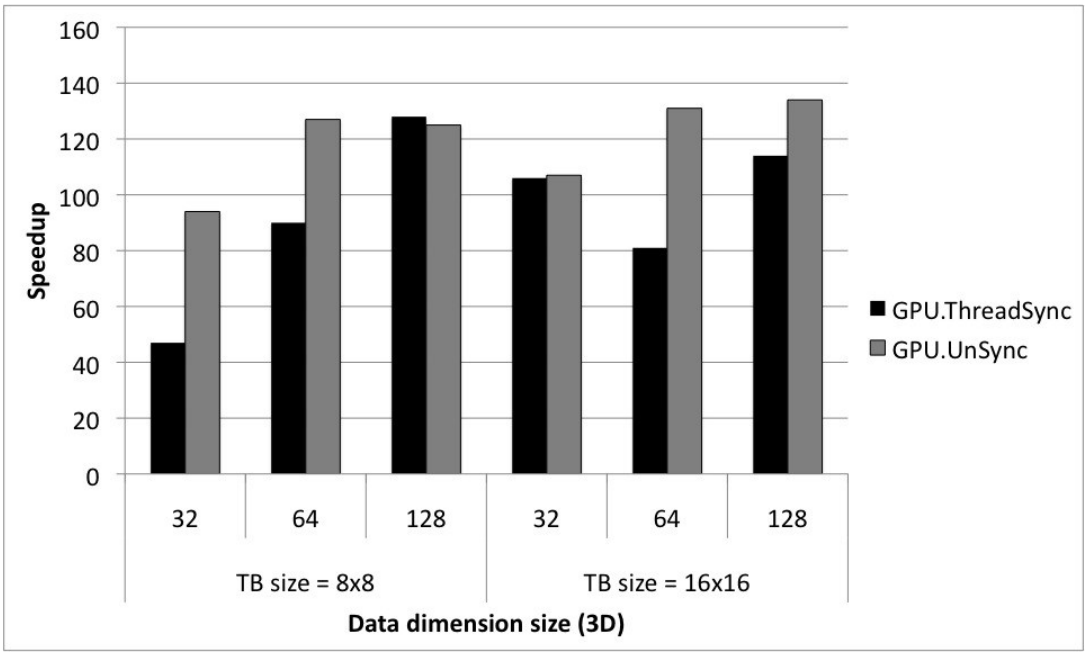

Figure 51. Kernel speedup for our secondary GPU-based implementation

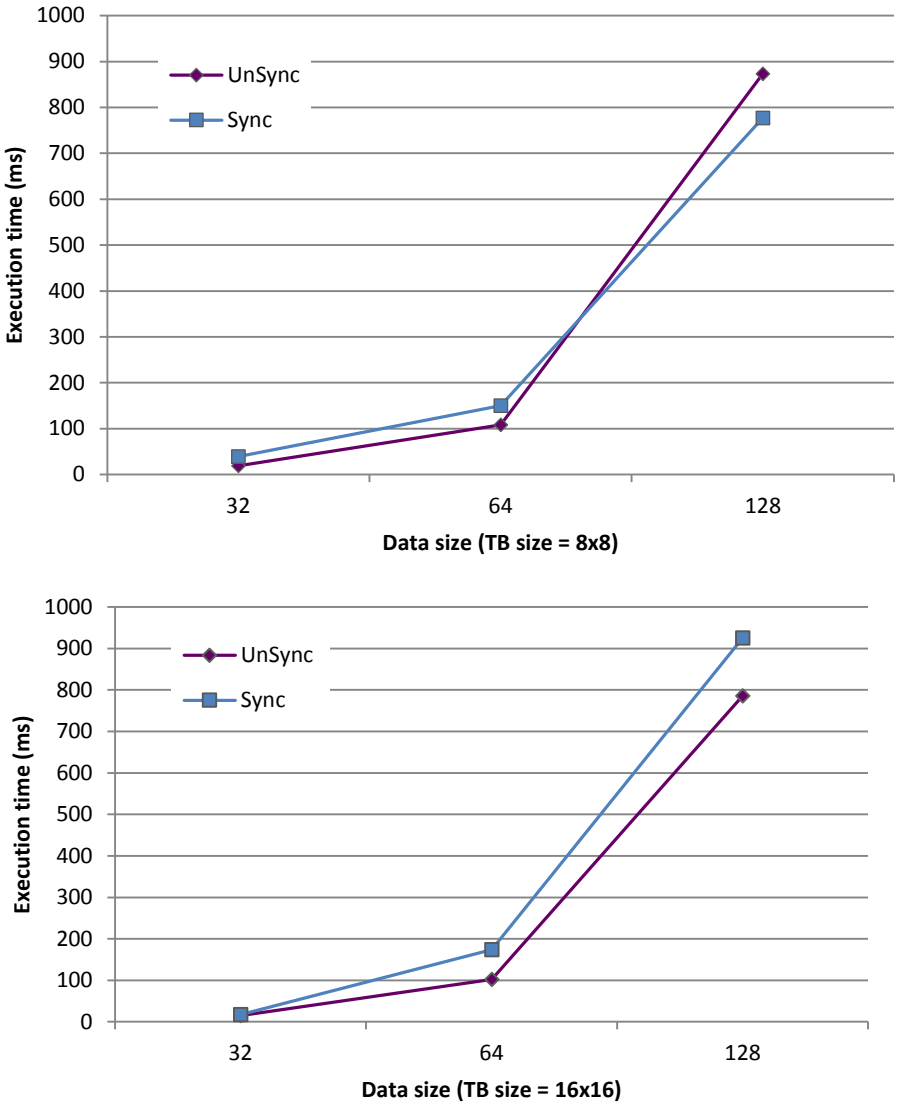

Figure 52. The trend of GPU-based solver execution time

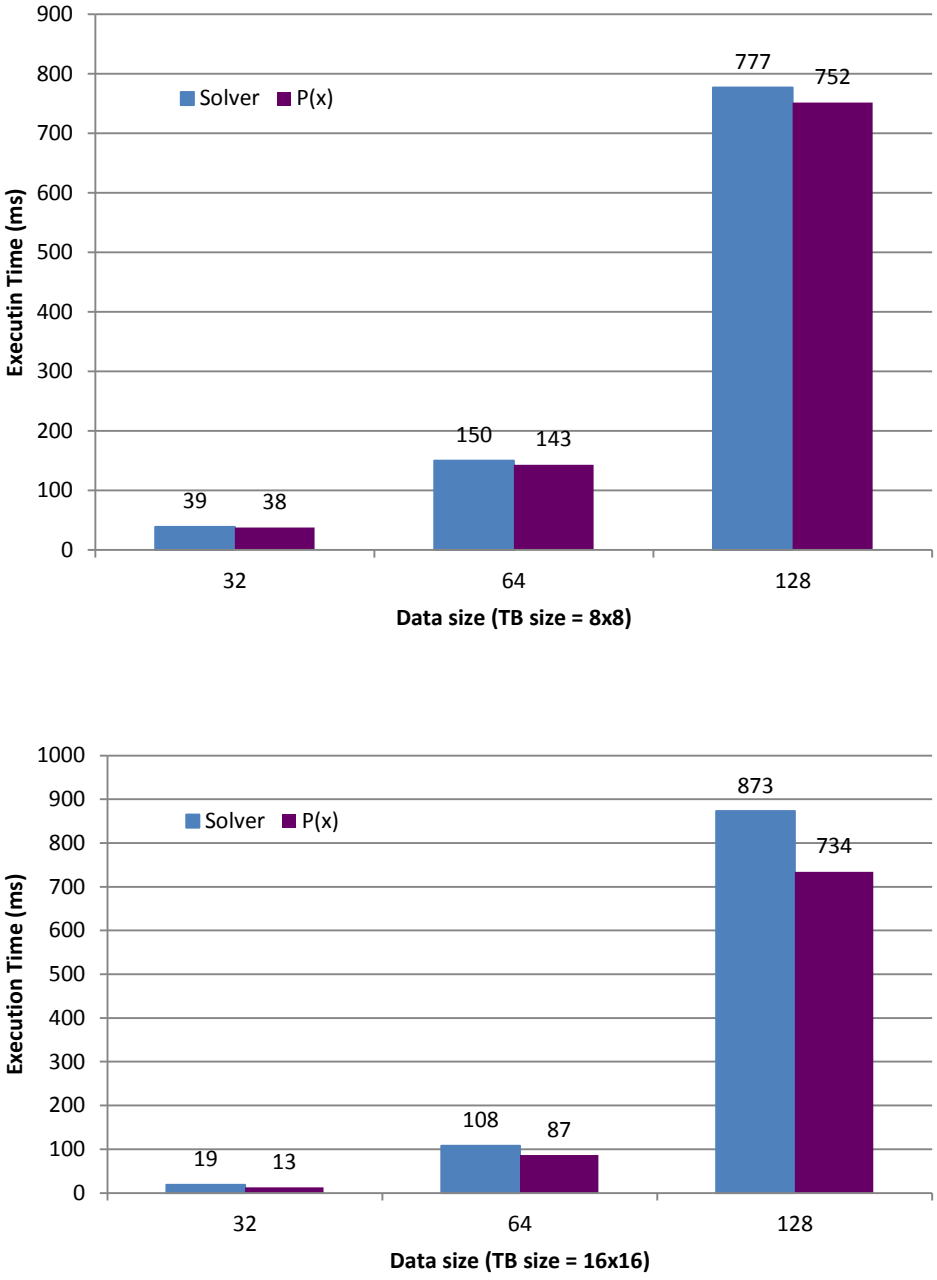

Figure 53. GPU-based solver execution time

6.6 Conclusion

Multi-valued imaging such as diffusion tensor imaging (DTI) has substantially higher noise levels compared to conventional MR imaging. Total Variation regularization, which is particularly designed for DTI images, can mitigate these limitations at the expense of substantial computation. The regularization algorithms for DTI images can significantly benefit from the advances in the architecture of GPU and reduce the execution time of regularization of matrix-valued images from 3 hours on a dual-core CPU to 1 minutes and 30 seconds, making the application of DTI images practical for many clinical settings.

We analyzed the effect of excessive synchronization in this algorithm, which results from dependence of this method to solving partial differential equations. We analyzed the effect of synchronization by comparing our GPU-based implementation to a secondary approach that eliminates synchronization. Thereby, we compared the effect of excessive synchronization on our primary approach against the effect of excessive computational workload and memory load in the secondary approach that is imposed on each thread by eliminating the synchronization. This application study showed that although the secondary approach achieves higher speedups, the primary approach scales better on the Quadro by the increase in image size.

CHAPTER 7

High Performance Signal Search

7.1 Overview

Proliferation of wearable sensors, mobile devices, and broadband wireless services in recent years has resulted in generation of tremendous amount of data. One of the most dominant types of data representation in medical and healthcare monitoring systems is time series. A time series is a sequence of data points, measured at successive points in time space. Design and development of efficient methods for mining time series has become a great interest of research community. Many studies have addressed problems such as indexing and retrieval [Pham], clustering [Vahdatpour09], and compression of time series, with special focus on biomedical related applications. Search and retrieval of similar time series subsequences is one of the most fundamental and useful functionalities. In medical monitoring systems, signal searching can be used to find certain interesting patterns or events in the patient history, comparing a patient's data with other patients to find most similar cases, detect and classify unknown events or patterns by comparing them to pre-annotated data.

Signal searching algorithms are considered as extension of time series subsequence matching, where a relatively short subsequence (query) is compared to a long time series, using a distance function. The goal of the algorithm is to find all time instances where patterns similar to those of the query are present in the longer time series. Distance function is crucial to the quality and performance of time series matching. Over the last decades there has been plethora of work to find best distance function for different application domains. Example of these distance metrics are Euclidean distance, Dynamic Time Warping (DTW) [Vlachos], and Wavelet [Chan09].

Historically, most of the data mining algorithms have been focused on single dimensional data, where the time series only contains one value at each time instance (e.g., stock market index, ECG). However, distance metrics and data mining algorithms defined for single dimensional time series are not directly applicable to most medical and healthcare monitoring systems, where multiple metrics are measured at each time instance. In such systems, to draw conclusion from one time series, one has to consider the context that is presented in other time series. For example, in a wearable system that is used to monitor heartbeat, it is important to also consider the activity that is performed by the user. A relative high heartbeat for a user who is performing sport activity should not send false alarms, while the same heartbeat, when user is not performing major physical activity could be an important sign for physicians.

The most important obstacle in front of processing multiple time series together is what is known as the curse of dimensionality [Indyk]. In most algorithms, increasing dimensions has exponential impact on execution time and since time series are usually lengthy, it makes executing the algorithm unfeasible.

In addition, algorithms for multi-dimensional time series (especially those of wearable and monitoring systems) have to address challenges associated with time synchronization and independent behavior of time series dimensions as well; Consider Figure 54, where a 3-dimensional subsequence (annotated as query) is compared to a longer time series *DB*. We have highlighted five time instances t_1, t_2, t_3, t_4, and t_5 in *DB*. t_1 represent an exact subsequence match for the query, where all subsequences in the query exist in the time series with precise time synchronicity. Finding such instances are simple by naively extending

the single dimensional subsequence matching algorithm and performing it over all dimensions. t_2 represents a *time shift* between different dimensions. As depicted, while all major patterns in the query exist in S at time t_2, however, there is time lag between their occurrences. t_3 represents a *sub-dimensional match* between the *query* and *DB* where only two dimensions from the *query* exist in *DB*. In t_4, although all patterns in the query are repeated in *DB*, however, subsequences have *variation and scaling* and do not exactly match the query subsequences. t_5 represents a multi-dimensional subsequence where although all subsequences in the query exist, however, their occurrences have *switched between different dimensions*. Depending on the application and domain, subsequences in t_2, t_3, t_4, and t_5 may be considered as good *matches* for the query. However, as it will be shown in evaluation section, naïve extension of single dimensional subsequence matching techniques fails in finding such subsequences.

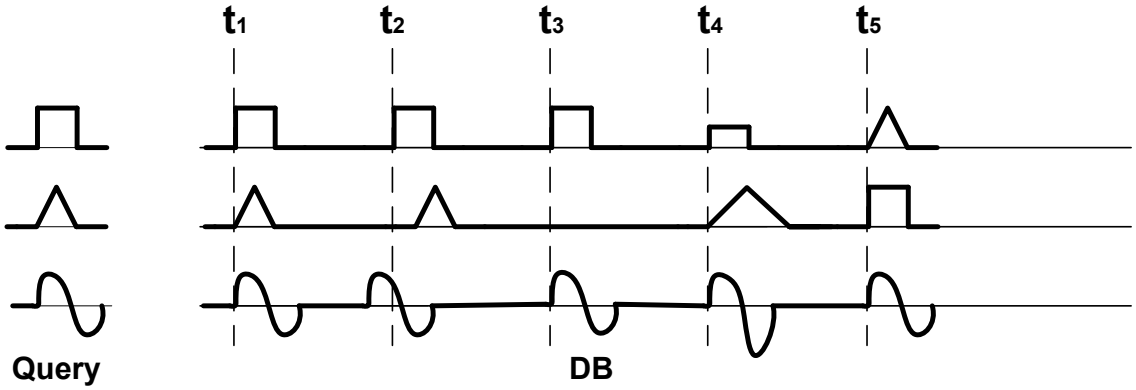

Figure 54. A query and five possible matches

In addition to find all such subsequences, it is important to rank them based on their similarity to the query (considering the application domain). In this chapter, we introduce and evaluate a method to search in multi-dimensional time series. The contributions of our multi-dimensional searching algorithm are fourfold:

1- Support for missing or switched dimensions

2- Tolerance for noise, scaling and asynchrony

3- Linear performance scalability with number of dimensions

4- Application aware rank and similarity metric

To our knowledge, this is the first work to study signal search in multi-dimensional time series. In order to show significance of the assumptions and scenarios presented above and depicted in Figure 54, we focus on two medical monitoring systems. In the evaluation section, we discuss the circumstances where each scenario is plausible and should be considered.

Medical Shoe: The medical shoe system [Novel] is used to monitor the plantar foot pressure via an array of ninety-nine pressure sensors embedded in an insole. This system has a range of applications including gait analysis, diabetic ulcer prevention, and real time fall detection and prevention.

We used multi-dimensional search functionality for two main goals: A) to classify frequent activities performed by user such as walking, jogging, and running. To do so, only one pre-annotated patterns from each activity is used as query. B) To discover all occurrence of abnormalities in users' gait. Search is especially beneficial to detect infrequent abnormalities where other supervised classification techniques underperform due to lack of pre-annotated data that can be used for classifier training.

PAM (Personal Activity Monitor) is a small wearable 3-dimensoinal accelerometer system that is used to monitor user's activity [Whi]. The main application of the system is to track and log activities performed by users during long intervals. To conserve energy, the system only collects data during long intervals. Large chunks of data are uploaded to server using USB connection on the system. Hence algorithms used to interpret this data should be efficient in processing hours of data collected from the user. A major challenge in processing accelerometer data for activity detection is the various possible placements of sensor on the body. Changing the orientation of the sensor changes the dimensions that capture body motions. In addition, depending on the placement of the sensor, scale, and the length of the patterns change as well. For example, the variation in signals is such that [Vahdatpour11] has proposed a method to detect the placement of the sensor on the body by analyzing the signals captured during walking.

The multi-dimensional signal search method provides the ability to search and discover all occurrences of any activity pattern, by using only one occurrence of such activities as the query. As will be discussed

in the evaluation section, we use our signal search algorithms to find occurrence of activities regardless of the placement and orientation of the sensors.

Structure of this chapter is as of the following; Section 7.2 covers related work and background. In Section 7.3 we formally define the problem, and in section 7.4 we introduce our method for multi-dimensional time series search. Section 7.5 contains experimental. Finally Section 7.6 concludes the study.

7.2 Background and Related Work

Subsequence search has been one of the most studied research topics in the signal processing research community.

Euclidean distance (ED) is the most basic and yet widely used function in which the distance between two subsequences is calculated by the absolute difference between corresponding points in subsequences [Faloutsos].

Numerous studies have proposed techniques to improve time series subsequence comparison, mostly by improving the distance function according to an application domain. Dynamic Time Warping (DTW) is a well-known technique for subsequence matching where a dynamic programming technique is used to find the best possible mapping between points of two subsequences. It has been shown that DTW is performs well in presence of noise, uncertainty, and scaling the time series [Rakthanmanon]. Although highly accurate, the quadratic execution time of DTW makes it unsuitable for comparison of long time series. Recently, [Keogh] has improved the performance of DTW by using lower bounding and early abandoning techniques. It is shown that using these techniques, DTW can achieve close to linear execution time.

Designing algorithms for multi-dimensional time series has recently become an interest of research society, since such data is becoming more prevalent and also computing capability has increased over years. In [Tanaka], authors use Principle Component Analysis (PCA) to convert multi-dimensional time series into single dimensional data and use single dimensional methods thereafter. A major disadvantage

of dimensionality reduction algorithms is their sensitivity on time synchronization between dimensions. [Vahdatpour09] has proposed a method for activity discovery in multi-dimensional time series. It uses a graph clustering algorithm to find related activity patterns between different dimensions. Considering properties of 2d and 3d trajectories, [Vlachos] has proposed a searching technique which supports multiple distance metrics. The study has shown that simple extension of DTW to compare 2 and 3 dimensional time series perform well. While the proposed method has general advantages of DTW in resilience toward minor variations and noise, it has limitations in handling time variation, dimension change, and scaling among dimensions. The main difference between trajectory time series and the sensor data is that trajectory time series are synchronous in all dimensions while sensor data mostly act independently (as in most cases, each dimension is measurement of independent event).

Indexing is one of the fastest techniques used for subsequence matching. Although very fast, Indexing techniques perform weak in face of noise and scaling. However, [Keogh] has shown that even though indexing is theoretically fast, in practice and for large time series, cache miss ratio (due to random and non-serial access to different locations in memory) makes it inefficient.

7.3 Problem Definition

Given a multi-dimensional query Q with k dimensions and length m and a multi-dimensional time series DB with length n ($n \gg m$) and k dimensions:

$$Q_{1..m}^{1..k} = Q_{1..m}^{1}, Q_{1..m}^{2}, \dots., Q_{1..m}^{k}$$
$$DB_{1..n}^{1..k} = DB_{1..m}^{1}, DB_{1..n}^{2}, \dots., DB_{1..n}^{k}$$

The goal is to find and rank all time instances t_i in DB, where $DB_{t_i..t_i+m}^{1..k}$ are most similar time series segments in DB to Q. Considering medical monitoring systems, the overall goal of using is to enable searching for similar activities, events, and conditions in sensor data collected form subjects, and projected on sensors. Hence the matching algorithm should consider the possibility of 1) Noise and variation in patterns, 2) temporal variation and lag between dimensions, and 3) time series dimension

change:

Noise and variation in patterns: Existence of noise is one of the most important properties of real-world signals, which if not accounted for in algorithm design, can result in significant algorithm quality degradation. Especially in human monitoring applications, where activities and measurement show huge variation depending on environmental conditions.

Temporal variation (lag between dimensions): Most systems, especially wearable systems used for human monitoring, include a number of independent sensors. Temporal variation between occurrences of patterns for similar activities in different sensors (time series dimensions) can be caused due to communication lag between sensor and data collector [Pham], delay in sensor response, or minor difference in performing the same activity [Vahdatpour09].

Time series dimension change: In systems where there are redundant sensing units with minor configuration variations, similar events and activities may be projected on different sensors in different time instances. For example, in 3d accelerometers used to monitor human activities, depending on how a device is worn by the user (the orientation of sensor), the accelerometers axis that captures motion in different directions changes.

In Figure 54, a brief summary of above properties was depicted in an artificial time series.

7.4 Overall Approach

In this chapter, we propose a search mechanism for multi-dimensional time series. Considering the characteristic introduced in Section 7.3, the algorithm is especially suitable for wearable monitoring systems. We use two of such monitoring systems to evaluate the algorithm. Our solution consists of three main phases, which we elaborate in the following sections.

7.4.1 Single Dimensional Search

The first phase of our algorithm is to search for single dimensional subsequences. In this phase, each dimension is searched independently. The intuition for this step in contrast to a multi-dimensional search is to ensure that noise in some dimensions or an independent unrelated pattern in another dimension does

not impact the matching result.

Figure 55 shows the first phase of our algorithm. In this phase, each dimension of the query Q is compared against the corresponding dimension in the time series DB via DTW to find time instances where similar patterns are discovered. A sliding windows method is used to sequentially compare all subsequences of length m in DB to Q. To find the best time instances with most similar patterns, we used a min-heap, which stores distances of subsequences at each time instance. At the end of the algorithm, we retrieve the top p time instances with minimum distance from the min heap.

In addition, in order to support the dimension change, each dimension of the query is also compared against its adjacent dimensions. The factor nr is used to determine the range of such comparison depending on the application. The result of this phase (Figure 55) is a set of 4-tuple relationships $S_u = (Q^i, DB^j, t, d)$ where Q^i is the ith dimension of query Q, DB^j is the jth dimension of the time series DB, t is the time instance in DB^i and d is the distance between $DB^j_{t..t+m}$ and $Q^i_{1..m}$. Each relationship S_u is called a single dimensional subsequence match.

To perform single dimensional search, we utilize DTW, as it has been shown to be resilient to noise and scale in the data. DTW resilience to noise and scaling comes from the fact that it finds a nonlinear alignment between two subsequences that minimizes the distance metric. To perform each DTW comparison between subsequences of length n and m, $n \times m$ calculations are needed. This is more than n comparisons needed to perform simple Euclidean distance (ED) calculation. Hence, using DTW increases the execution time. To speedup DTW computations, we used the lower bounding technique proposed in [Rakthanmanon]. This technique is an approximation that compares first and last pair of points in two subsequences. If the distance is high, it is shown that it can be used as a criterion for pruning the candidates. Figure 55 shows the integration of DTW in the first phase of our algorithm.

```
SINGLE_DIM_SEARCH(Query, DB)
  //Input: Query = {Q^1_1..m, ..., Q^dim_1..m}
  //Input: DB = {DB^1_1..n, ..., DB^dim_1..n }
  //Output: Set of matches S = {(Q^i0, DB^j0, t_i0, d_i0), …, (Q^i0, DB^j0, t_i0, d_i0)}
  //Find all single dimensional subsequence matches:
```

```
Foreach DB^i
  Foreach Q^j where i-nr < j <i+nr
    For t:1…n of length in DB^i
      Insert (t, d = DTW(DB^i 1..n, Q^j 1..m)) to MinHeap_i,j
Foreach MinHeap_i,j
  (t, d) = Pop MinHeap_ij
  S ← (Q^j, DB^i, t, d)
Return S
```

Figure 55. Single dimensional search

7.4.2 Multi-Dimensional Subsequence Construction

The next phase of the algorithm is to construct multi-dimensional subsequences from the discovered single dimensional subsequence matches in the previous phase. Performing a single dimensional search and combining the single dimensional matches to construct a multi-dimensional match in a separate phase lets us address the possibility of temporal variation between dimensions (time lag between dimensions).

The input to this phase is a set of relationships $S = \{S_u = (Q^i, DB^j, t_u, d_u),\ 1<u<k\times p\}$, each S_u representing a single dimensional subsequent match. Here, k is number of dimensions, nr is neighboring ratio and p is the max number of matches found in any dimension. We define two single dimensional match relationships ((a, b, x, y) and (c, d, p, q)) independent iff ($a \neq c$) and($b \neq d$).

We define $M \subseteq S$ a feasible multi-dimensional subsequence match for query $Q_{1..m}^{1..k}$, at time instance t_0, if for all members of M, $t_0+m < t < t_0+m$ and no other independent relationships with $t_0+m < t < t_0+m$ can be added to set M.

To clarify more, consider the example in Figure 56 where for time t, $Q_{1..m}^{2}$ is found to be similar to $DB_{t..t+m}^{1}$ and $DB_{t-m..t}^{2}$. In this example, $\{(Q^1, DB^1, t_0\text{-}m, d_1), (Q^2, DB^2, t_0, d_2), (Q^3, DB^3, t_0, d_3)\}$ and $\{(Q^2, DB^1, t_0, d_4), (Q^3, DB^3, t_0, d_5)\}$ are feasible matches in DB for query Q, as adding any other 4-tuplet will violate the independence of the relationships. However, $\{\{(Q^2, \mathrm{DB}^1, t_0, d), (Q^2, DB^2, t_0, d), (Q^3, DB^3, t_0, d)\}$ is not a feasible multidimensional subsequence match as the first and second 4-tuplets are not independent. The intuition behind the requirement for independence of relationships is that a subsequence in the query should only be considered once for comparison to a multi-dimensional subsequence at time windows $t_0+m < t < t_0+m$.

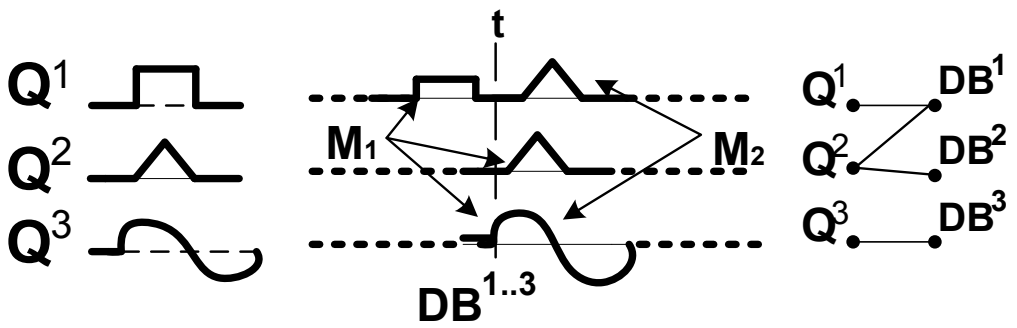

Figure 56. A simple query and two feasible matches (left and middle), the representing graph (right)

To find all such sets, our approach is to construct a graph in which nodes represent time series patterns in *DB* and *Q* overlapping time instance t, and edges represent relationships between them; finding all feasible matches is equivalent to finding all maximal independent edge sets (MIS) in the graph (or simply maximal independent vertex set in complement of graph). In general Maximal Independent Set problem is known to be NP complete. However, a relationship exists in the graph, only if a query is found to be similar to a subsequence in DB. In addition, the graph has at most $2 \times k$ nodes, where ($k << m, n$) is the number of query dimensions. Hence, the computation complexity of MIS problem on this graph is negligible comparing to the computation required in the first phase of the algorithm. First part of Figure 57 summarizes second phase of our algorithm.

```
MULTI_DIM_SEARCH(S)
  //Input: Set of single dimensional matches in 4-tuple:
  //S = {(Q^i0, DB^j0, t_i0, d_i0),…, (Q^iu, DB^ju, t_iu, d_iu)}
  //Output: Set  of all feasible multi-dimensional matches
  //M = {M1 = {Sa, Sb, …, Sc}, M2 = {Sd, Se, …, Sf}, …}
  //Find and sort all feasible multi-dimensional matches
  For t:1..n
     SEGt = {Si ∈ S | t-m < ti < t+m}
     G(V,E) = CONVERT_TO_GRAPH(SEGt):
     Add all MAXIMAL_INDEPENDENT_SET(G) to M

CONVERT_TO_GRAPH(S)
  //Input: S = {(Q^i0, DB^j0, t_i0, d_i0),…, (Q^iu, DB^ju, t_i0, d_i0)}
  //Output: G(V,E)
  Convert set S to Graph G where V = {Q^i, DB^j} and dx as edge     weight

RANK(M)
  //Input: Set of feasible multi-dimensional matches:
  //M = {M1 = {(Q^i0, DB^j0, t_i0, d_i0),…,(Q^ir, DB^jr, t_ir, d_ir)}, ...}
  //Output: Sorted list of matches:
  //L = {L1 = {(Q^i0, DB^j0, t_x, d_min0),…}, …}
  //Rank all feasible multi-dimensional matches

  For i: 1 to z
    Calculate Overall distance for Mi
  L = Sort Ms based on overall distance
  Return L
```

Figure 57. Multi-dimensional combination

7.4.3 GPU Implementation

Subsequence search is sequential in nature and is not immediately suitable for SIMD model of GPUs. However, because there are many subsequences that need to repeat the same computation, GPUs become suitable. Especially, for multi-dimensional signal search in medical monitoring applications that signals can change dimensions, each dimension of the query needs to be searched in adjacent dimensions in the time series signal. Therefore, there is potentially an order of magnitude increase in the number of subsequence searches.

In the GPU implementation, each thread is responsible for computing the DTW distance between a time series subsequence and the query. The time series signal which is long is copied to the global memory. The query however, can be copied to the shared memory to be shared among all threads in a thread block. A sliding window technique is used, therefore, each thread is assigned a window and

computes the DTW distance between its assigned window of time series and the query.

In our GPU implementation, we used an optimal DTW implementation which computes the full matrix for computing the distance between two subsequences. Note that due to high computational power requirement, for CPU implementation we use an efficient DTW algorithm which uses early abandoning techniques for more efficiency, which is an approximation that is proved to be accurate enough.

7.4.4 Query Segmentation

More efficient subsequent search on GPU allows us to do more diligent subsequence search for higher quality of search. We propose to do query segmentation as a secondary step to increase the search recall. The goal of query segmentation phase is to segment the query into multiple parts and compare the query segments to the respective segment in the time series subsequence rather than comparing the whole query for similarity to a time series subsequence. The intuition for this phase is that if some segments in the query have good similarity to a time series subsequence but the other segments in the query have poor similarity to the time series subsequence, we can still return a partial match. This will increase the search recall as it is possible to return such a candidate as a partial match.

After signal searching is completed, we start this phase by segmenting the query and starting a signal search kernel for each query segment. Because subsequent search is faster on GPU, we do not introduce additional overhead by scanning the results to determine what the segments to repeat the search for are.

7.4.5 Search Exhausting

Scaling factor (SF) which is used to overcome the scaling problem [Keogh] impacts both execution time and accuracy of single dimensional DTW algorithm. To achieve higher precision, it is preferred to have smaller SF (around .05). However, smaller SF results in lower recall rate. In this phase, we introduce a re-enforcement step, which uses the information in multi-dimensional matches to improve recall of the subsequence matching. The intuition behind search exhausting is that if in a feasible multi-dimensional match, several dimensions are missing (since single dimensional DTW has not found any matches to the patterns in the query), searching with higher SF may result in discovering matches. This technique

adaptively changes SF without impacting the overall precision.

7.4.6 Ranking

The goal of this phase is to rank multi-dimensional subsequence matches according to their similarity to the query. Consider all M_i , $1<i<z$, where each M_i is a set of independent 4-tuplets (feasible multi-dimensional sequence matches) that are found in phase two.

$$M_i = \{(Q^{i0}, DB^{i0}, d_{i0}, t_{i0}), \dots, (Q^{ir}, DB^{ir}, d_{ir}, t_{ir})\}$$

To calculate the overall distance of Mi to query Q:

$$Distance(M_i) = \sum_{j=0}^{|M_i|} f(Q^{ij}, DB^{ij}) \times d_{ij}$$

Where

$$f(Q^{ij}, DB^{ij}) = \begin{cases} 1 & if\ Q^{ij} = DB^{ij} \\ \alpha & if\ Q^{ij} \neq DB^{ij} \end{cases}$$

The overall distance of a feasible multi-dimensional subsequence from a query is sum of the distances of its single dimensional matched patterns. However, depending whether the dimensions in query and subsequence that contain a pattern are the same or different, parameter α is used to weight the distances. Function f is an application dependent function which determines the plausibility of matching patterns across dimensions. For example, if in a system, same activities can be recorded by different sensors over time, alpha can be set to a constant 1. In addition, overall ranking preference is given to feasible matches with higher number of matched patterns. As shown in Figure 57, ranking is done in reverse order of the overall distance of feasible matches to the query, meaning that a feasible match with minimum distance to Q is the highest ranked results.

7.5 Empirical Results

7.5.1 Benchmarks

To our knowledge, this work is the first study to address multi-dimensional search with broad definition of match and support for ranking. Hence, for evaluation, we compared our method to a multi-dimensional

extension of Euclidean Distance technique (referred as EDT hereafter). In this method, distance between two multi-dimensional subsequences *A* and *B* is defined to be:

$$ED(A_{1..n}^{1..k}, B_{1..n}^{1..k}) = \sum_{i=1}^{k} \sqrt{\sum_{t=1}^{n} (A_t^i - B_t^i)^2}$$

Using this function, when comparing a query *Q* to time series *DB*, the *p* best subsequences of *DB* with smallest distance are found, and the subsequence are ranked according to their distance to the query. Euclidean distance is inherently not resilient to noise and scaling. In addition, no further algorithm is applied to count for sub-dimensional queries and also to overcome time gap, and asynchrony between dimensions.

7.5.2 Metrics

Discounted Cumulative Gain (nDCG) [Jarvelin] is used to evaluate search results, as it is a common metric in evaluating search methods. Consider a sorted list of m results returned for a query. $nDCG_m$ is defined as:

$$nDCG_m = \frac{DCG_m}{DCG_{IDEAL}}, \qquad DCG_m = \sum_{i=1}^{m} \frac{2^{rel_i}-1}{\log_2(i+1)}$$

Where, DCG_{IDEAL} is DCGm for a perfect result (highest possible DCG). rel_i is the relevance of result *i*. for our experiments, if a signal is a correct match to the query (same activity or event) $rel_i = 1$, otherwise it is zero. The intuition behind this metric is to penalize non relevant subsequences that are appearing higher in the search results.

To evaluate the methods, first we use a synthetic time series to visually showcase the benefit of using the method and also measure the performance and scalability of the algorithm.

7.5.3 Synthetic Data

Figure 58 shows a time series which consists of 4 dimensions. We used the subsequence at left as a multi-dimensional query. As depicted, we have artificially inserted several scenarios including time shift, scaling, dimensions switch, and noise. The red and blue box in the figure represents the location and rank

of the found subsequence matches for the query by EDT and (our technique) MD-DTW respectively. As shown, EDT has lower recall of the possible matches. In addition, the synchrony of the patterns impact the distance function and hence the ranking.

As shown in the figure, MD-DTW has higher recall of subsequences, since not only it is resilient to time lag and scaled patterns, but also detects *sub-dimensional matches* and those subsequences with switched dimensions.

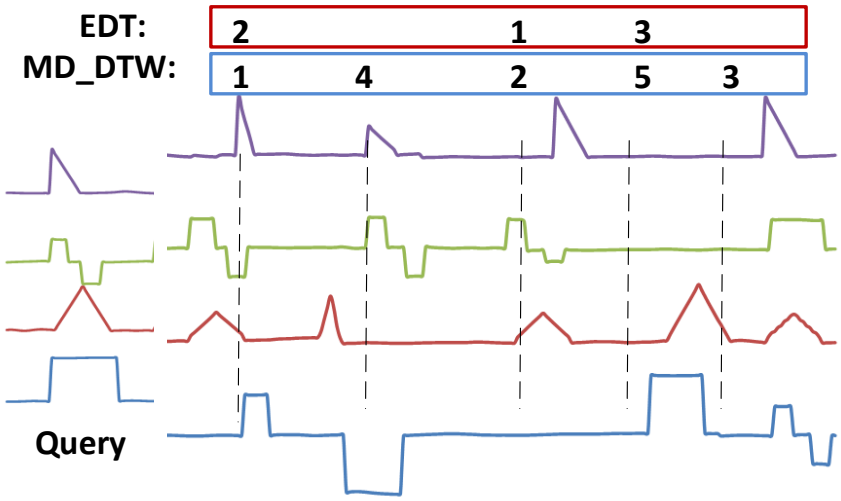

Figure 58. A synthetic time series (right) and a query (left). Red and blue boxes show rank of matched time series.

7.5.4 Real Data

To further evaluate the performance and accuracy of the method, we tailored the search method to two remote monitoring applications.

7.5.4.1 Shoe

We collected pressure data from 5 subjects, each wearing the pressure monitoring system for 10 minutes. Users were asked to perform activities including walking, running, and artificial limping in arbitrary order. Annotations were collected by a second person monitoring the subjects. Sample patterns from each activity were used as queries. Figure 59 shows the relative foot plantar pressure for two different subjects at a same corresponding moment during the walking activity on all (99) sensors. In this graph,

consecutive sensors are physically adjacent. As shown, while sensors that capture the pressure for the activity in two subjects are close to each other, however, some pressure points (especially in front foot area) are switched between subjects (due to variation in foot shapes). To capture this dimension switch, we set the *nr* to three.

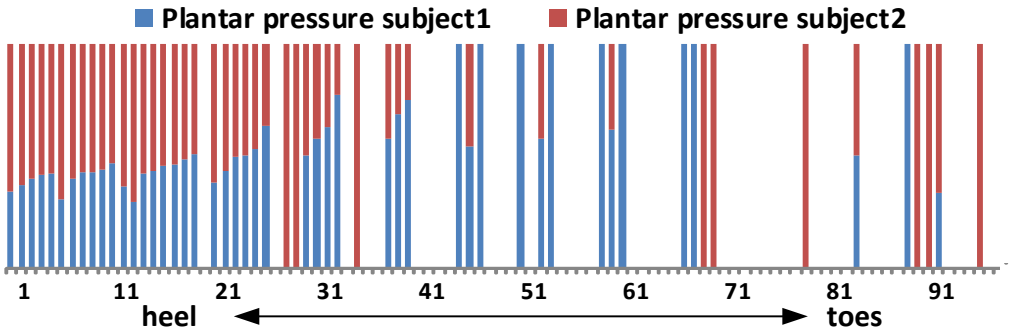

Figure 59. Plantar pressure on all sensors at a time instance for two subjects

Two set of experiments were performed. First, we used random samples of normal activities as query and searched for them in the whole corpus of data collected from all subjects. We then judged the quality of the top 20 ranked subsequence matches for MD-DTW and EDT by comparing them to annotation recorded during data collection. Table 11 shows $nDCG_{20}$ for three activities.

Table 11. NDCG of three activities

	MD-DTW	EDT
Walk	.86±.03	.78±.04
Run	.94±.02	.92±.02
Jump	.80±.04	.67±.05

As shown, MD-DTW outperforms EDT in all activities. However, for running, since multi-dimensional patterns are dominated by high amplitude patterns in the heel, both methods exhibit high quality. Table 12 presents the Recall gains achieved by using query segmentation and search exhausting techniques.

Table 12. Recall gains using query segmentation and search exhausting techniques

	MD-DTW	Query Segmentation	Search Exhausting
Walk	.80	.83	.84
Run	.86	.88	.89
Jump	.70	.72	.74

Abnormal activities have less frequency of occurrence and the variation of patterns among them is higher. To perform the next experiment, we used subsequences that represent limping (on either left or right foot) as query and searched for similar occurrences of patterns in the signal. Since only a few repetition of each abnormal activity query existed in the time series, we did not calculate NDCG. Instead, discovered subsequences are compared to annotations to evaluate precision and recall of the algorithms.

Table 13. Precision/Recall of abnormal activity detection

	MD-DTW		EDT	
	Precision	Recall	Precision	Recall
Limp Right	.86±.04	.60±.05	.73±.03	.62±.05
Limp Left	.78±.03	.59±.06	.70±.04	.63±.05

As depicted in Table 13, MD-DTW has higher precision for both experiments while its recall is on par. Figure 60 shows the execution time of signal searching using GPU using data collected form the Shoe in normal usage scenarios. As depicted, even with long time series and query length, GPU still has execution time in order of minutes.

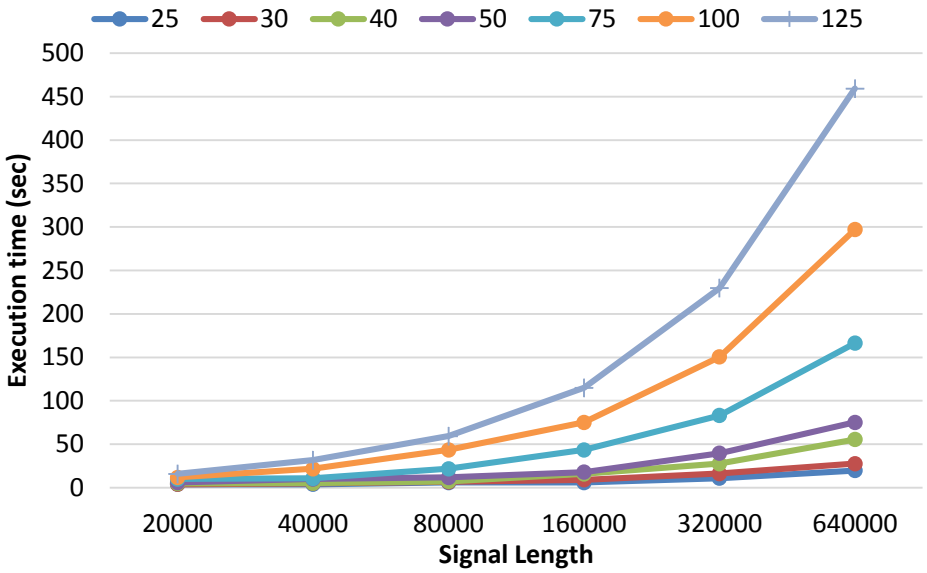

Figure 60. Execution time of MD_DTW on GPU for Shoe scenarios (legend shows the query length)

7.5.4.2 Personal Activity Monitor

We used the multi-dimensional search technique to search for walking patterns captured in the sensor placed on three different locations of the body (arm, waist, leg). For each position, the orientation and the

exact placement of the sensor was varied.

To conduct the test, we collected data from 10 different subjects, each wearing the sensor for 30 minutes and performing miscellaneous activities. For each sensor position (arm, leg, waist), we used 5 subsequences that are annotated as walking patterns as the queries and searched the signal for all similar occurrences of the signal. Since the sensor orientation was unknown for each subject, we set *nr* to 2 to compare each dimension with all other dimensions of 3d accelerometer. Table 14 represents the $NDCG_{20}$ for both methods.

Table 14. NDCG of search for walking patterns

	MD-DTW	EDT
Arm	.89±.03	.72±.04
Waist	.95±.02	.91±.02
Leg	.96±.02	.88±.03

The results show that the more the variation of patterns is for the activity, the more the advantage of MD-DTW is. For example, in leg and waist sensor placements, degree of freedom is limited; hence motion data has more limited variation and both techniques perform better in finding occurrences of similar multi-dimensional patterns.

	MD-DTW	Query Segmentation	Search Exhausting
Arm	.70	.72	.74
Waist	.90	.91	.95
Leg	.90	.92	.93

Figure 61 shows two example occurrences of walking activity projected on the device placed on a subject's waist. When the left subsequence was used a query, EDT failed to find the right subsequence as a match, even though the subsequences are very similar. As highlighted in the figure, two main reasons for this were: 1) the lag in start of the patterns in the first dimension comparing to other two dimensions. 2) the variation in the pattern in the third dimension.

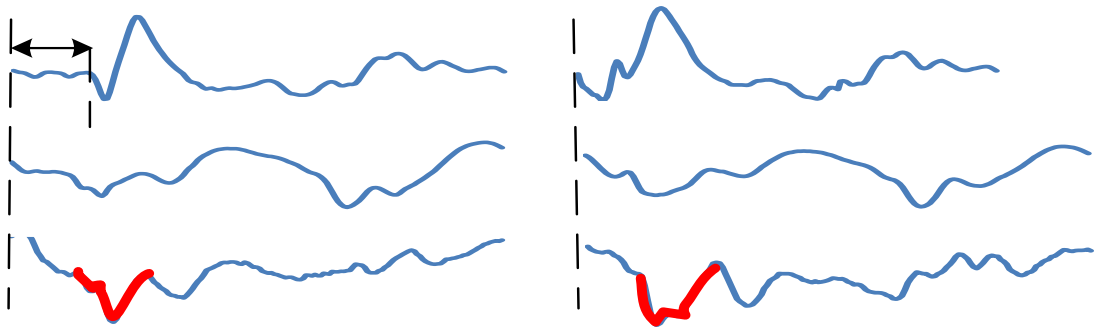

Figure 61. Two walking patterns projected on a 3d accelerometer sensor placed on waist

Figure 62 shows the execution time of the MD_DTW approach on GPU. As it is shown, since there are only three dimensions in the data, even with very long time series, the execution time in below 10 seconds.

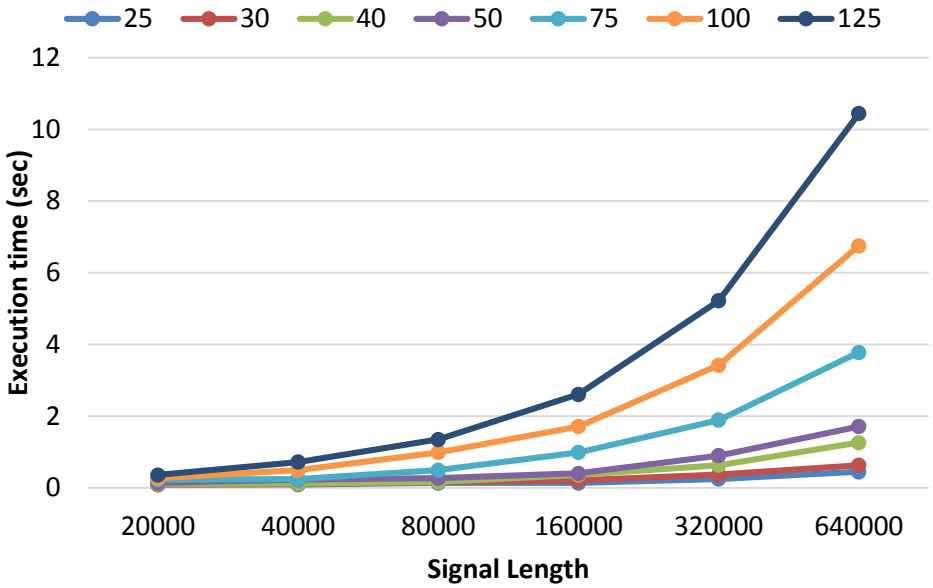

Figure 62. Execution time of MD_DTW on GPU for PAM data (legend shows the query length).

7.5.5 Overall Performance

In this section, we evaluate the overall performance of our GPU-based multi-dimensional search with its counter-part CPU implementation. The counter-part CPU implementation uses an optimized DTW implementation which is an approximation of DTW. It uses early abandoning and several lower bounding

thresholds to avoid computing the whole dynamic programming table. Our GPU-based implementation however, uses complete DTW implementation that is optimal and constructs the entire dynamic programming matrix for each subsequence, hence having higher quality of results. As depicted in Figure 63, the speed up varies based on the query and signal length. For shorter signals, longer queries yield higher speed up, since GPU has enough threads to handle the added overhead. However, for long timer time series, since the GPU is overwhelmed by computation, increasing the length of query results in over loading the GPU. As depicted, GPU-based implementation achieves up to 25X speedup. Using such analysis, it seems for longer time series, the system should break down the operation into smaller batches, to achieve the highest possible speed up.

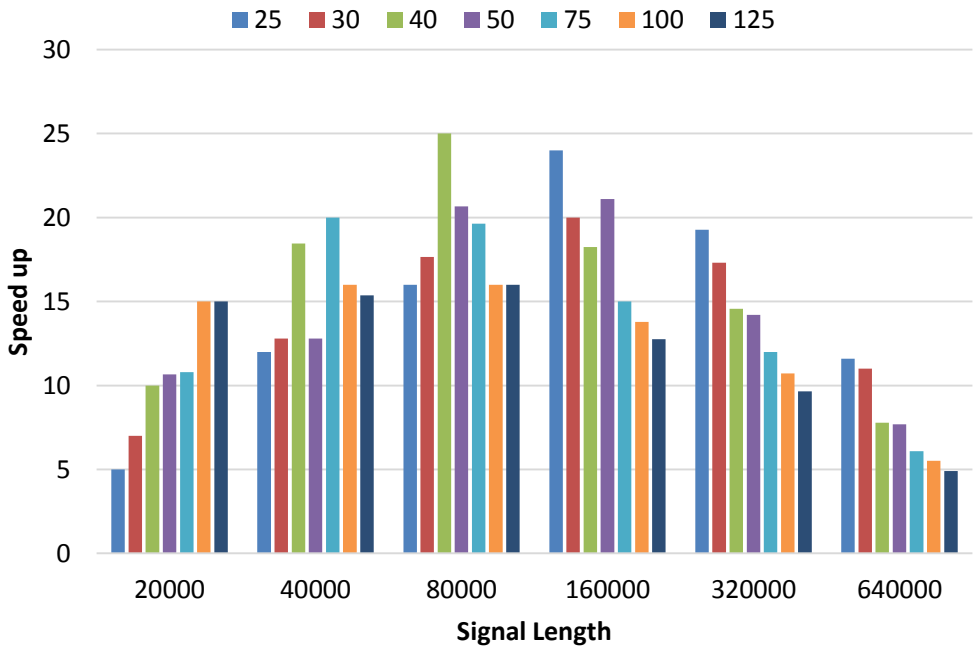

Figure 63. Speed up achieved by using GPU for MD_DTW (legend shows the query length).

Figure 64 shows the scalability of our method. If *nr* is set to a constant, increasing the number of dimensions has linear impact on overall execution time. However, if *nr* is set to be a linear function of number of dimensions, the increase in execution time is quadratic, as the number of dimensions increases.

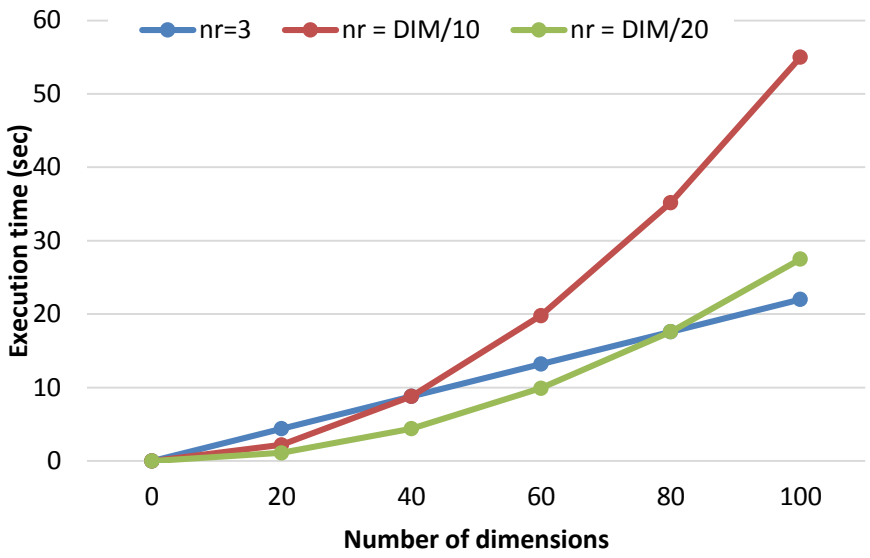

Figure 64. Scalability of the MD_DTW versus the number of dimensions.

7.6 Conclusion

In this chapter we presented the first study to address search and ranking in multi-dimensional signal. We focused on medical monitoring devices and the properties of time series generated in them. We proposed a method to efficiently and accurately search for similar time series in them. We evaluated the performance and accuracy of the method using data collected from two remote medical monitoring devices. As a future work, we plan to further speed up the execution, by using parallel processing capacity of GPUs.

CHAPTER 8

Conclusion

This dissertation presented techniques to utilize the massive parallel capability of many-core Graphics Processing Units (GPUs) for non-data parallel applications, algorithms and data structures. GPUs provide great capability to achieve high performance in inherent data parallel applications. However, their capability is underutilized in applications and data structures which do not have data parallel properties. We presented several techniques to adapt such applications to many-core GPU architecture. Parallel decomposition techniques, memory reuse and optimization, and data structure optimization and parallelization that were specialized for many-core requirements were among the techniques that we introduced to leverage the massive parallel processing power of many-core GPUs.

As medical and biomedical informatics is one of the most rapidly growing areas, we chose several applications in these domains to show case the effectiveness of our solutions. Diffusion Tensor Imaging (DTI) denoising and multi-dimensional signal searching in Medical Shoe and Personal Activity Monitor were the applications that we used. We showed that GPUs are underutilized for complex non-inherently parallel tasks and proposed techniques that can be used to improve the efficiency of these non-data parallel algorithms and speed up their execution by at least an order of magnitude. We showed that by identifying the bottlenecks in achieving the many-core requirements we can achieve drastic improvements with simple solutions. The solutions that we proposed in this dissertation can be utilized as a programming library for the use of developers.

We conclude that modern commodity GPUs are promising high-performance computing platforms and can be widely used as a powerful co-processor in different domains of computing. Especially for health care and medical informatics leveraging GPUs is very effective as they are very cost-effective, energy

efficient and are widely accessible. The SIMD-type programming model used for these GPUs is simple and makes it relatively convenient for development. We advocate for the continuation of GPGPU efforts and more systematic studies on many-core applications.

REFERENCES

[Aho] Aho, Alfred V., Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. Compilers: principles, techniques, & tools. Vol. 1009. Pearson/Addison Wesley, 2007.

[Akl] Aki, Selim G. "The design and analysis of parallel algorithms.", Prentice Hall, New Jersey, 1989.

[Alcantara09] Alcantara, Dan A., Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta. "Real-time parallel hashing on the GPU." In ACM Transactions on Graphics (TOG), vol. 28, no. 5, p. 154. ACM, 2009.

[Alcantra] Alcantara, Dan Anthony Feliciano. Efficient Hash Tables on the GPU. PhD Thesis, University of California, Davis, 2011.

[Amd] AMD Stream Processor. http://ati.amd.com/products/streamprocessor/index.html.

[Asanovic] Asanovic, Krste, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson et al. The landscape of parallel computing research: A view from Berkeley. Vol. 2. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[Bammer] Bammer, Roland. "Basic principles of diffusion-weighted imaging." European journal of radiology 45, no. 3 (2003): 169-184.

[Basser] Basser, Peter J., James Mattiello, and Denis LeBihan. "MR diffusion tensor spectroscopy and imaging." Biophysical journal 66, no. 1 (1994): 259-267.

[Bast] Bast, Holger, and Torben Hagerup. "Fast and reliable parallel hashing." In Proceedings of the third annual ACM symposium on Parallel algorithms and architectures, pp. 50-61. ACM, 1991.

[Bell] Bell, Nathan, and Michael Garland. "Implementing sparse matrix-vector multiplication on throughput-oriented processors." In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, p. 18. ACM, 2009.

[Bihan] Le Bihan, Denis, Jean-François Mangin, Cyril Poupon, Chris A. Clark, Sabina Pappata, Nicolas Molko, and Hughes Chabriat. "Diffusion tensor imaging: concepts and applications." Journal of magnetic resonance imaging 13, no. 4 (2001): 534-546..

[Butenhof] Butenhof, David R. Programming with POSIX threads. Addison-Wesley Professional, 1997.

[Chan] Chan, Tony F., and Luminita A. Vese. "Active contour and segmentation models using geometric PDE's for medical imaging." In Geometric methods in bio-medical image processing, pp. 63-75. Springer Berlin Heidelberg, 2002.

[Chan09] Chan, Kin-Pong, and Ada Wai-Chee Fu. "Efficient time series matching by wavelets." In Data Engineering, 1999. Proceedings. 15th International Conference on, pp. 126-133. IEEE, 1999.

[Che] Che, Shuai, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. "Rodinia: A benchmark suite for heterogeneous computing." In Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on, pp. 44-54. IEEE, 2009.

[Christiansen] Christiansen, Oddvar, Tin-Man Lee, Johan Lie, Usha Sinha, and Tony F. Chan. "Total variation regularization of matrix-valued images." International journal of biomedical imaging 2007.

[Dabiri] Dabiri, Foad, Alireza Vahdatpour, Hyduke Noshadi, Hagop Hagopian, and Majid Sarrafzadeh. "Ubiquitous personal assistive system for neuropathy." 2nd International Workshop on Systems and Networking Support for Health Care and Assisted Living Environments, p. 17. ACM, 2008.

[Faloutsos] Faloutsos, Christos, Mudumbai Ranganathan, and Yannis Manolopoulos. "Fast subsequence matching in time-series databases." Vol. 23, no. 2. ACM, 1994.

[Fischer] Fischer, Bernd, and Jan Modersitzki. "Curvature based image registration." Journal of Mathematical Imaging and Vision 18, no. 1 (2003): 81-85.

[Grama] Grama, Ananth. Introduction to parallel computing. Addison Wesley, 2003.

[Gil] Gil, Joseph, and Yossi Matias. "Fast hashing on a PRAM—designing by expectation." In Proceedings of the second annual ACM-SIAM symposium on Discrete algorithms, pp. 271-280. Society for Industrial and Applied Mathematics, 1991.

[Greenwald] Greenwald, Michael Barry. "Non-blocking synchronization and system design." PhD diss., Stanford University, 1999.

[Harris] Harris, Mark J., William V. Baxter, Thorsten Scheuermann, and Anselmo Lastra. "Simulation of cloud dynamics on graphics hardware." In Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, pp. 92-101. Eurographics Association, 2003.

[He] He, Bingsheng, and Jeffrey Xu Yu. "High-throughput transaction executions on graphics processors." Proceedings of the VLDB Endowment 4, no. 5 (2011): 314-325.

[Indyk] Indyk, Piotr, and Rajeev Motwani. "Approximate nearest neighbors: towards removing the curse of dimensionality." In Proceedings of the thirtieth annual ACM symposium on Theory of computing, pp. 604-613. ACM, 1998.

[Jaja] JáJá, Joseph. An introduction to parallel algorithms. Addison Wesley Longman Publishing Co., Inc., 1992.

[Jarvelin] Järvelin, Kalervo, and Jaana Kekäläinen. "Cumulated gain-based evaluation of IR techniques." ACM Transactions on Information Systems (TOIS) 20, no. 4 (2002): 422-446.

[Kandemir] Kandemir, Mahmut, and Alok Choudhary. "Compiler-directed scratch pad memory hierarchy design and management." In Proceedings of the 39th annual Design Automation Conference, pp. 628-633. ACM, 2002.

[Keogh] Keogh, Eamonn J., and Michael J. Pazzani. "Scaling up dynamic time warping for datamining applications." In Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 285-289. ACM, 2000.

[Knuth] Knuth, Donald E. The art of computer programming, Vol. 3. Addison-Wesley, Reading, MA, 1973.

[Larson] Larson, Per-Åke, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. "High-performance concurrency control mechanisms for main-memory databases." Proceedings of the VLDB Endowment 5, no. 4 (2011): 298-309.

[Massalin] Massalin, Henry, and Calton Pu. "A lock-free multiprocessor OS kernel." ACM SIGOPS Operating Systems Review 26, no. 2 (1992): 108.

[Matias] Matias, Yossi, and Uzi Vishkin. "On parallel hashing and integer sorting." Journal of Algorithms 12, no. 4 (1991): 573-606.

[Mayun] Muyan-Ozcelik, Pinar, John D. Owens, Junyi Xia, and Sanjiv S. Samant. "Fast deformable registration on the GPU: A CUDA implementation of demons." In Computational Sciences and Its Applications, 2008. ICCSA'08. International Conference on, pp. 223-233. IEEE, 2008.

[Michael] Michael, Maged M. "High performance dynamic lock-free hash tables and list-based sets." In Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures, pp. 73-82. ACM, 2002.

[Moazeni12] Moazeni, Maryam, and Majid Sarrafzadeh. "Lock-free Hash Table on Graphics Processors." In Application Accelerators in High Performance Computing (SAAHPC), 2012 Symposium on, pp. 133-136. IEEE, 2012.

[Moazeni09] Moazeni, Maryam, Alex Bui, and Majid Sarrafzadeh. "Accelerating total variation regularization for matrix-valued images on GPUs." In Proceedings of the 6th ACM conference on Computing frontiers, pp. 137-146. ACM, 2009.

[Mori99] Mori, Susumu, and Peter B. Barker. "Diffusion magnetic resonance imaging: its principle and applications." The Anatomical Record 257, no. 3 (1999): 102-109.

[Mori02] Mori, Susumu, and Peter van Zijl. "Fiber tracking: principles and strategies–a technical review." NMR in Biomedicine 15, no. 7-8 (2002): 468-480.

[Nikolls] Nickolls, John, Ian Buck, Michael Garland, and Kevin Skadron. "Scalable parallel programming with CUDA." Queue 6, no. 2 (2008): 40-53.

[Noshadi] Noshadi, Hyduke., Foad Dabiri, S. Ahmadian, N. Amini, and M. Sarrafzadeh. "HERMES: mobile system for instability analysis and balance assessment." ACM Transaction Embedded Computing Systems (2010).

[Novel] Novel.de, Pedar, 2007, http://www.novel.de/

[Nvidia] Nvidia, CUDA. "NVIDIA CUDA programming guide." (2011).

[Owens] Owens, John D., David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. "A Survey of general-purpose computation on graphics hardware." In Computer graphics forum, vol. 26, no. 1, pp. 80-113. Blackwell Publishing Ltd, 2007.

[Pam] Gulf Coast Data Concepts, http://gcdataconcepts.com/x6-2.html.

[Parhami] Parhami, Behrooz. "SIMD machines: do they have a significant future?." ACM SIGARCH Computer Architecture News 23, no. 4 (1995): 19-22.

[Pham] Pham, Tri, Eun Jik Kim, and Melody Moh. "On data aggregation quality and energy efficiency of wireless sensor network protocols-extended summary." In Broadband Networks, 2004. BroadNets 2004. Proceedings. First International Conference on, pp. 730-732. IEEE, 2004.

[Potter] Potter, Jerry L. Associative Computing: A Programming Paradigm for Massively Parallel Computers. Perseus Publishing, 1991.

[Rudin] Rudin, Leonid I., Stanley Osher, and Emad Fatemi. "Nonlinear total variation based noise removal algorithms." Physica D: Nonlinear Phenomena 60, no. 1 (1992): 259-268.

[Rakthanmanon] Rakthanmanon, Thanawin, Bilson Campana, Abdullah Mueen, Gustavo Batista, Brandon Westover, Qiang Zhu, Jesin Zakaria, and Eamonn Keogh. "Searching and mining trillions of time series subsequences under dynamic time warping." In Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 262-270. ACM, 2012.

[Rodrigues] Rodrigues, Christopher I., David J. Hardy, John E. Stone, Klaus Schulten, and Wen-Mei W. Hwu. "GPU acceleration of cutoff pair potentials for molecular modeling applications." In Proceedings of the 5th conference on Computing frontiers, pp. 273-282. ACM, 2008.

[Ryoo] Ryoo, Shane, Christopher Rodrigues, Sam Stone, Sara Baghsorkhi, Sain-Zee Ueng, and Wen-mei W. Hwu. "Program optimization study on a 128-core GPU." In The First Workshop on General Purpose Processing on Graphics Processing Units, pp. 30-39. 2007.

[Seltzer] Seltzer, Margo, and Ozan Yigit. "A new hashing package for UNIX." In Proceedings of the Winter USENIX Technical Conference, pp. 173-84. 1991.

[Sewall] Sewall, Jason, Jatin Chhugani, Changkyu Kim, Nadathur Satish, and Pradeep Dubey. "PALM: Parallel architecture-friendly latch-free modifications to B+ trees on many-core processors." Proc. VLDB Endowment 4, no. 11 (2011): 795-806.

[Steffen] Steffen, Peter, Robert Giegerich, and Mathieu Giraud. "GPU parallelization of algebraic dynamic programming." In Parallel Processing and Applied Mathematics, pp. 290-299. Springer Berlin Heidelberg, 2010.

[Stone] Stone, Samuel S., Justin P. Haldar, Stephanie C. Tsao, Wen-mei Hwu, Bradley P. Sutton, and Z-P. Liang. "Accelerating advanced MRI reconstructions on GPUs." Journal of Parallel and Distributed Computing 68, no. 10 (2008): 1307-1318.

[Tan06] Tan, Guangming, Shengzhong Feng, and Ninghui Sun. "Locality and parallelism optimization for dynamic programming algorithm in bioinformatics." In Proceedings of the 2006 ACM/IEEE conference on Supercomputing, p. 78. ACM, 2006.

[Tan07] Tan, Guangming, Ninghui Sun, and Guang R. Gao. "A parallel dynamic programming algorithm on a multi-core architecture." In Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures, pp. 135-144. ACM, 2007.

[Tanaka] Tanaka, Yoshiki, Kazuhisa Iwamoto, and Kuniaki Uehara. "Discovery of time-series motif from multi-dimensional data based on MDL principle." Machine Learning 58, no. 2-3 (2005): 269-300.

[Vahdatpour11] Vahdatpour, Alireza, Navid Amini, and Majid Sarrafzadeh. "On-body device localization for health and medical monitoring applications." In Pervasive Computing and Communications (PerCom), 2011 IEEE International Conference on, pp. 37-44. IEEE, 2011.

[Vahdatpout09] Vahdatpour, Alireza, Navid Amini, and Majid Sarrafzadeh. "Toward unsupervised activity discovery using multi-dimensional motif detection in time series." 21st international joint conference on Artificial intelligence, pp. 1261-1266. Morgan Kaufmann Publishers Inc., 2009.

[Valiant] Valiant, Leslie G. "A bridging model for parallel computation." Communications of the ACM 33, no. 8 (1990): 103-111.

[Valos] Valois, John D. "Lock-free linked lists using compare-and-swap." In Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing, pp. 214-222. ACM, 1995.

[Vazirani] Vazirani, Vijay V. Approximation algorithms. springer, 2004.

[Vlachos] Vlachos, Michail, Marios Hadjieleftheriou, Dimitrios Gunopulos, and Eamonn Keogh. "Indexing multi-dimensional time-series with support for multiple distance measures." In Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 216-225. ACM, 2003.

[Weickert] Weickert, Joachim. "A review of nonlinear diffusion filtering." In Scale-space theory in computer vision, pp. 1-28. Springer Berlin Heidelberg, 1997.

[Westin] Westin, Carl-Fredrik, S. E. Maier, H. Mamata, A. Nabavi, F. A. Jolesz, and Ron Kikinis. "Processing and visualization for diffusion tensor MRI." Medical image analysis 6, no. 2 (2002): 93-108.

[Whi] Wireless Health Institute Portal, http://www.wirelesshealth.ucla.edu/.

[Xiao] Xiao, Shucai, Ashwin M. Aji, and Wu-chun Feng. "On the robust mapping of dynamic programming onto a graphics processing unit." In Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on, pp. 26-33. IEEE, 2009.