METHODS FOR GPU ACCELERATION OF BIG DATA APPLICATIONS

by

Reza Mokhtari

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

Methods for GPU acceleration of Big Data applications

Reza Mokhtari
Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto
2017

Big Data applications are trivially parallelizable because they typically consist of simple and straightforward operations performed on a large number of independent input records. GPUs appear to be particularly well suited for this class of applications given their high degree of parallelism and high memory bandwidth. However, a number of issues severely complicate matters when trying to exploit GPUs to accelerate these applications. First, Big Data is often too large to fit in the GPU's separate, limited-sized memory. Second, data transfers to and from GPUs are expensive because the bus that connects CPUs and GPUs has limited bandwidth and high latency; in practice, this often results in data-starved GPU cores. Third, GPU memory bandwidth is high only if data is layed out in memory such that the GPU threads accessing memory at the same time access adjacent memory; unfortunately this is not how Big Data is layed out in practice.

This dissertation presents three solutions that help mitigate the above issues and enable GPU-acceleration of Big Data applications, namely **BigKernel**, a system that automates and optimizes CPU-GPU communication and GPU memory accesses, **S-L1**, a caching subsystem implemented in software, and a **hash table** designed for GPUs. Our key contributions include: (i) the first automatic CPU-GPU data management system that improves on the performance of state-of-the-art double-buffering scheme (a scheme that overlaps communication with computation to improve the GPU performance), (ii) a GPU level 1 cache implemented entirely in the software that outperforms hardware L1 when used by Big Data applications and, (iii) a GPU-based hash table (for storing key-value pairs popular in Big Data applications) that can grow beyond the available GPU memory yet retain reasonable performance. These solutions allow many existing Big Data applications to be ported to GPUs in a straightforward way and achieve performance gains of between 1.04X and 7.2X over the fastest CPU-based multi-threaded implementations.

Acknowledgements

I consider myself the luckiest among all versions of me in all parallel universes, because I was fortunate enough to get research and life advice from my advisor, Professor Michael Stumm. I truly believe he has a very special blend of wisdom, open-mindedness, perfectionism, and kindness. He looks at the world uniquely in that he has a talent in easily dissecting complex concepts and getting to their basic elements. He then rebuilds his own version of the concept from ground up using those elements. He has frequently amazed me with this talent, which is also very helpful in doing advanced research.

During my PhD years, I have often been asked by people around me whether if I would choose to go for a PhD again if I were given the chance to go back five years; it gives me a great deal of comfort in thinking and saying "I would do it in a heartbeat, if I'm given the opportunity to do it under the advice of Michael." Indeed, I am deeply indebted to him, beyond what words can describe, and will always be sincerely grateful for his contribution to my development, both as a researcher and as a person.

I would also like to address a special thought to my Masters thesis co-supervisor, Reza Azimi, without whom I would definitely not be where I am now. He guided me during my first steps in research and showed me how I can have fun doing it. I will always be grateful for his trust and thankful for his help.

Much gratitude goes to my thesis committee members and associated faculty, Andreas Moshovos, Ding Yuan, Ashvin Goel, and John Owens for their valuable feedback and support.

To all my friends, especially Roja, Sepehr(s), and Nass, your care and encouragement were always a source of joy and comfort. I am lucky to have in my life.

Finally, I would like to thank my parents and my sister, who, despite being far from me, have been a constant source of love, support, approval, and strength. I feel fortunate to have been raised in such a house that enabled me to reach this point of my life.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

An important class of computations operate on voluminous datasets in ways similar to what is sometimes referred to as ”Big Data” computations and other times (in the GPU community) referred to as streaming computations. These computations perform simple, straightforward, and independent operations on a large number of input data records. They are trivially parallelizable, and the input data exhibits no (or very low) reuse. This class of computation is large and includes computations that filter, transform, aggregate or partition large data sets.

As an example application, consider Page View Count, which counts the number of occurrences of each URL in input web-log files. This application is easily parallelized by partitioning the input data into smaller chunks, having different worker threads count the URLs in different chunks, and then, at the end, aggregate the results.

GPUs appear to be well suited for accelerating Big Data applications. The many GPU cores allow for highly parallelized computing. A GPU offers aggregate compute power an order of magnitude larger than what a CPU can; e.g., 8.3 TFLOPS vs. 1.5 TFLOPS (Nvidia GTX 1080 vs. an Intel Skylake CPU with 20 cores). And GPU memory has significantly higher theoretical bandwidth than CPU memory, since they were designed for graphics processing; e.g., 320 GB/s vs. 115 GB/s. Figures 1.1 and 1.2 depict the improvements in computing power and memory bandwidth of modern GPUs and CPUs. As shown, GPUs computational power and memory bandwidth are reaching levels previously expected only from small supercomputers.

While GPUs appear to be attractive for Big Data applications, a number of issues complicate their efficient use. First is the issue of managing the data that does not fit in GPU memory. The CPU and GPU have separate memories, requiring explicit data transfers between CPU and GPU memory, and GPU memory is limited in size (currently up to at most 16GB). As a result, substantial non-trivial effort has to go into managing data that does not fit in GPU memory. For the typical input data of Big Data applications, this means that the data needs to be explicitly partitioned into chucks and iteratively copied into GPU memory for processing there, which makes

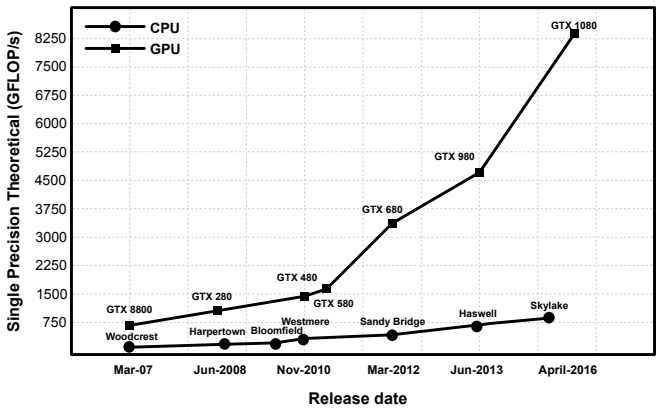

Figure 1.1: GPU and CPU computing power over the years.

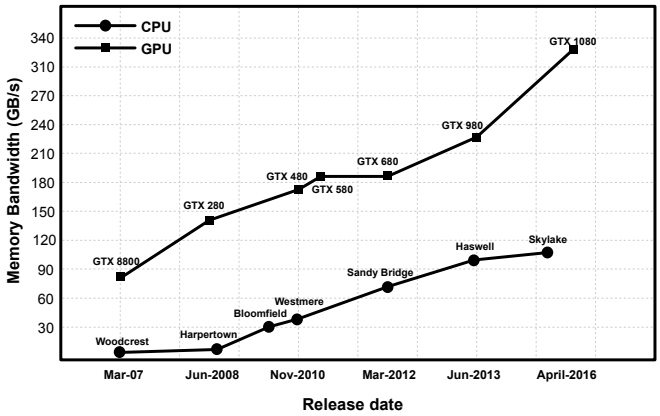

Figure 1.2: GPU and CPU memory bandwidth over the years.

GPU programming more difficult and error-prone. Further, the PCIe link that connects the two memories has limited bandwidth and, for the computations we are considering, can often be a bottleneck starving GPU cores from their data. For example, PCIe Gen 3 has a theoretical maximum throughput of 15.75 GB/s, far lower than the memory bandwidth GPU-side, and this bandwidth is difficult to exploit in practice. Indeed, while impressive speedups have been reported for many GPU applications, the speedups were often calculated without taking into account the overhead of transferring the input data to GPU memory [32].

A second issue is the fact that the high bandwidth of GPU memory can be exploited only when GPU threads executing at the same time access memory in a coalesced fashion; i.e., where the threads simultaneously access adjacent memory locations. If not coalesced, memory accesses may become serialized, resulting in significantly lower memory throughput and high access latencies. Because the applications we are targeting are not a priori structured to operate on data in a coalesced fashion, the data has to be reorganized to support coalesced accesses on the GPU, which is non-trivial.

A third issue is the fact that the GPU L1 caches are entirely ineffective [43]. For the typical number of cores

in modern GPUs, the L1 caches are too small and their cache line sizes are disproportionally large given the small cache size. For example, the Nvidia GTX 1080 has 128 cores per SMX, each of which can have multiple outstanding memory accesses at any time. Yet the maximum L1 cache size is 64KB and the cache line size is 128 bytes (for a total of 512 cache lines). Thus, cache lines are typically evicted before there is any reuse, causing a high degree of cache thrashing and an attendantly low L1 hit rate given the large number of threads executing on GPU cores, each issuing multiple memory accesses (typically to independent memory locations in our target applications).

Finally, a forth issue is that traditional data structures typically perform poorly when naïvely ported to the GPU. For example, traversing linked lists can result in low GPU core utilization because groups of GPU cores execute in SIMD fashion and all the cores in the group will have to wait for the slowest core that had to traverse the longest list. To prevent such performance degradations, data structures and their associated operations often have to be restructured to take into account the micro-architectural characteristics of the GPU. Hash table is one of those data structures that is particularly important to us because it can efficiently store key-value pairs that are widely used in Big Data applications. Hash tables that do not fit in GPU memory are particularly challenging to implement on GPUs without excessive CPU-GPU communication because there is, by design, little locality in accessing the table.

In this dissertation, we present a set of runtime/compile-time systems and libraries to address the issues mentioned above. First, we present a scheme, called BigKernel, that automates and optimizes CPU-GPU communication and GPU memory accesses. Second, we present S-L1, a level-1 (L1) cache for GPUs implemented entirely in software to address the ineffectiveness of hardware L1 caches. Finally, we present a hash table design that allows the hash table to grow beyond the size of GPU memory, and yet stay reasonably efficient. The code required for BigKernel, S-L1, and our hash table can be applied to existing Big Data applications using straightforward compiler transformations and/or by using runtime libraries.

BigKernel provides pseudo-virtual memory to Big Data GPU applications that operate on streaming data. It uses a four -stage pipeline with automated prefetching to (i) optimize CPU-GPU communication and (ii) optimize GPU memory accesses. BigKernel optimizes CPU-GPU communication by only transferring the data that will be accessed GPU-side, and it optimizes GPU memory accesses by rearranging the data in a way that coalescing is increased significantly when the GPU accesses the data. It also simplifies the programming model by allowing programmers to write kernels using arbitrarily large data structures when the data records can be operated on independently, thus relieving the programmer from having to partition the data into segments, manage buffers, transfer data between CPU and GPU, and having to invoke GPU kernels multiple times. Applying BigKernel on a set of Big Data applications shows that it outperforms both the CPU-based multi-threaded implementations (on average by 3.0X) and the GPU-based implementation of the applications that use the state-of-the-art double-

buffering scheme to transfer their data (on average by 1.7X).

S-L1 uses GPU's shared memory to provide an L1 cache implemented entirely in software. The GPU shared memory is a small software-managed memory, which is positioned at the same level as the GPU's L1 cache, and has the same access latency as the hardware L1. S-L1 determines, at run time, the proper size of cache, samples the effectiveness of caching the data of different data structures, and based on that information, decides what data to cache. On a set of Big Data GPU applications, S-L1 achieves an average speedup of 1.9 over hardware L1 and 2.1 over no L1 caching. Combining S-L1 with BigKernel leads to an average speedup of 1.19 over BigKernel alone.

Finally, our hash table is intended to be used as a key-value store for GPU-based Big Data applications. It (i) supports variable-length keys and values, (ii) can perform on-the-fly grouping of key-value pairs with the same key and, (iii) uses a model of computation we developed, called SePo, to be able to obtain reasonable performance even when the table and its data grow larger than available GPU memory. Comparing a set of GPU-based Big Data applications that use our hash table with the corresponding CPU-based multi-threaded implementations shows an average speedup of 3.5, despite having the hash table grow to up to four times larger than the space available in GPU memory.

1.1 Contributions

This dissertation makes the following five specific contributions:

- The first system to automate CPU-GPU data transfers for large datasets without requiring the programmer to split the data or annotate the code (BigKernel).

- The first scheme to improve on the performance of state-of-the-art double-buffering scheme for GPUs (BigKernel).

- A software L1 cache implemented entirely in software with several novel features including a run-time scheme to automatically determine the parameters to configure the cache (S-L1).

- A GPU-based hash table to store key-values which can grow beyond the available GPU memory and retain reasonable performance.

- A model of computation that allows certain types of applications (including Big Data applications) to run more efficiently (SePo).

1.2 Organization

The rest of this dissertation is organized as follows. We start by presenting, in chapter 2, background information on general purpose computing on graphics hardware (GPGPU) and go over some of the key challenges in achieving good performance on GPUs. This is followed in chapter 3 with a description of work related to ours. We focus specifically on existing runtime/compile-time systems that improve the performance of GPU applications and on existing core data structures used in GPU applications. This dissertation then describes the three systems we designed and implemented. BigKernel, S-L1, and our hash table design are presented in chapters 4, 5, and 6, respectively. We close with concluding remarks and possible directions for future work in chapter 7.

We would like to point out that the work we describe in chapters 4 and 5 were published in May, 2014 [73] and August, 2015 [74], respectively. The work we describe in chapter 6 is ready to be published.

Chapter 2

Background

In this chapter we provide an overview of GPGPU and then present background information on the hardware and software sides of GPU programming to help the reader better understand the key factors involved in the performance and programmability of GPUs. This chapter can be skipped by readers already familiar with GPUs.

2.1 GPGPU Overview

The term GPU was first popularized by Nvidia in 1999 when it called its Geforce 256 "the world's first 'GPU', or Graphics Processing Unit". It took less than a year for programmers to use this GPU for non-graphical applications, coining the term General Purpose GPU (GPGPU). However, to program a GPU, one had to use the graphics library interfaces, which was rather tedious. GPU Programming later became easier with the introduction of GPU software development tools tailored specifically for developing computing applications.

In 2007, Nvidia introduced a GPGPU software platform for its own line of GPU devices, which it called *Compute Unified Device Architecture* (CUDA) [77]. Geforce GTX 8800 was the first CUDA-enabled consumer GPU, offering 518 GigaFLOPs of theoretical performance [78], compared to less than 6 GigaFLOPs theoretical performance of the, at the time, state-of-the-art Intel Core 2 processor running at 3.0 GHz. This created some excitement, especially when impressive speedups of applications ported to GPU platforms, sometimes in excess of 100, were reported (see Section 3.1). The rapid improvements of raw GPU performance over the last several years has given GPUs a performance advantage over CPUs for many compute intensive applications.

In the following section, we briefly give an overview of GPU hardware and software. For the hardware overview, we often use the specifications of Nvidia GTX 680 which implements the Kepler architecture [81]. We describe GTX 680 because it is the GPU we used to evaluated BigKernel and S-L1, presented in chapters 4 and 5. Although other GPU architectures will differ in some specifications, they share many general characteristics.

GPU model	GPU Family and year	Number of cores#2	memory bandwidth	Size of memory	Size of L2 cache	Size of onchip memory
GTX 680	Kepler (2012)	1536	192GB/s	2GB	1.5MB	64KB
GTX 780	Kepler (2013)	2304	288GB/s	3GB	1.5MB	64KB
GTX Titan Black	Kepler (2014)	2880	336GB/s	6GB	1.5MB	64KB
GTX 980	Maxwell (2015)	2048	192GB/s	8GB	2MB	128KB
GTX 1080	Pascal (2016)	2560	320GB/s	8GB	4MB	128KB

Table 2.1: Comparing recent GPUs key characteristics.

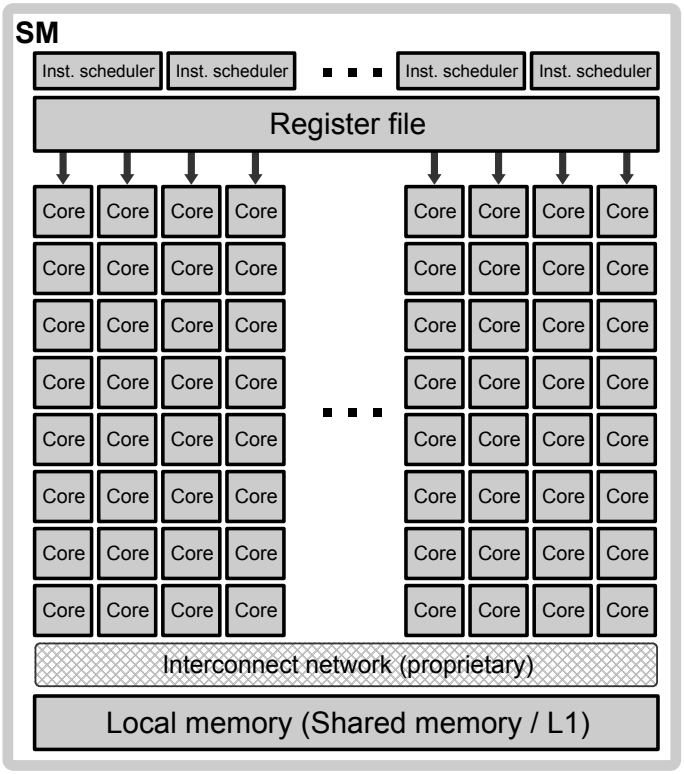

Figure 2.1: The internal structure a *streaming multiprocessor* (SMX).

Since GTX 680, Nvidia has released four newer series of GPUs, with the latest being GTX 1080. A comparison of the key characteristics of several recent GPUs is provided in Table 2.1.

In the following sections, we primarily use the terminology of Nvidia to describe the hardware and software aspects of GPUs, however we provide AMD/OpenCL's equivalent terms in footnotes as well.

2.2 GPU Overview

2.2.1 GPU Hardware

GPU compute unit consists of several streaming multiprocessors (SMX)[1], each of which contains multiple computing cores, tens of thousands of 4-byte registers, and a small (e.g. 64KB) but fast on-chip memory. Figure 2.1

[1] SMXs are called *compute units* in AMD/OpenCL's terminology.

depicts the internal structure of an SMX.

Compared to a traditional multi-core CPU, a GPU has a far higher number of cores, but each core is simpler than a conventional CPU core. GPU cores are simpler, in part because they do not have an integrated instruction scheduler. Instead, a single instruction scheduler is shared between multiple GPU cores in an SMX. For example, in Kepler architecture, every 16 cores share the same instruction scheduler. As a result, groups of threads executed by the cores that share the same instruction scheduler run in lockstep, implementing Single Instruction Multiple Data (SIMD) type execution. *Thread divergence* will occur if, on a conditional branch, threads that run in lockstep take different paths, causing the threads to serialize the execution of different paths.

The small on-chip memory in each SMX is partitioned between a hardware-managed L1 cache and a programmer-managed *shared memory*[2]. In many GPUs, the programmer can configure how the SMX on-chip memory is to be partitioned between L1 and shared memory. GTX 680, with 64KB of on-chip memory per SMX, can be configured to assign 16KB-48KB or 48KB-16KB to L1 cache and shared memory, respectively. L1 cache is used to cache register spills and stack data (and not application data) [80]. Shared memory is programmer managed memory and allows threads running on the same SMX to efficiently share data.[3]

A larger L2 cache is shared by all SMXs of the compute unit and is connected to off-chip DRAM called *global memory*. We refer to this global memory as GPU memory in this document to differentiate it from CPU main memory.

For the Nvidia GTX 680, which is considered to be a reasonably modern GPU, access latency to registers, L1, shared memory, L2 and DRAM is 10, 80, 80,210, and 340 cycles, respectively. Moreover, the theoretically maximum bandwidth from L1, shared memory, L2 and DRAM have been reported to be 190.7GB/s, 190.7GB/s, 512GB/s, and 192GB/s, respectively [81].

GPU hardware uses two strategies to hide the latency of accesses to GPU memory: (i) fast context switch to another threads on a memory access[4] and (ii) wide memory buses that enable accesses to several data elements in a single memory transaction. To exploit that latter, however, GPU programs should be optimized to access memory in a coalesced fashion.

Coalesced memory accesses are those that are simultaneously issued by concurrent threads and fall within the same aligned 128-byte region. These memory accesses are coalesced into a single memory access by a hardware *coalescing unit* before being sent to memory, resulting in only one 128-byte memory transaction. Parallel memory accesses from concurrent threads to data are defined as *n-way* coalesced if *n* of the accesses fall within the same aligned 128-byte region.

[2]Shared memory is called *local memory* in AMD/OpenCL's terminology.

[3]For threads to shared data, they not only need to run on the same SMX, but they have to be in the same *thread block*, which we describe later.

[4]The context switch between GPU threads is fast because everything is stored in registers and thus there is almost no data movement on a context switch.

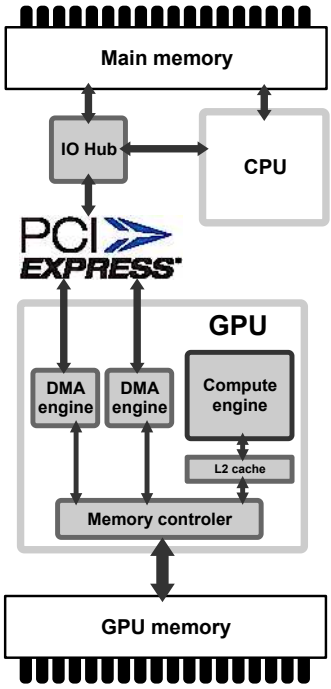

Figure 2.2: The connections between different components of the system.

Figure 2.2 illustrates the GPU compute unit, L2 cache, and DMA engines are connected to other parts of a computer system. GPUs use a PCI Express link (PCIe) to connect to CPU and CPU memory. One end of this link is connected to the IO hub located on the motherboard/CPU chip, and the other end is connected to one or two DMA engine(s) located on the GPU. The link is formed by a group of up to 16 *lanes*, each with enough wiring to support bidirectional data transfers at the same time. The latest generation of PCIe, PCIe revision 3.0, can transfer data at a theoretical speed of 15.75 GB/s in each direction if all 16 lanes are utilized.

Although GPU programs are supposed to transfer data in bulk to GPU memory before operating on it, a feature introduced in later versions of CUDA, called *zero copy*, enables GPU threads to directly access (the remote) CPU memory [77]. However, to enable zero copy, the to-be-access locations of CPU memory must be pinned so that they are not paged out by the operating system.

Finally, the latest family of CUDA-enabled GPUs (i.e., Pascal) offers hardware demand-paging support which provides a programming convenience referred to as "unified memory", in which the address space of GPU is merged with that of the CPU to create a single virtual address space. This way, GPU programs are allowed to access data located in CPU memory without first copying them to GPU memory; underneath, the demand-paging hardware pauses the execution and waits for the GPU driver to copy the required data to GPU memory before resuming execution.

2.2.2 GPU Software

A typical CUDA program consists of two parts: the main part of the program, which is executed on the host (the CPU), and the *kernel*[5], called by the main program, which is executed on the GPU by a collection of *GPU threads*[6] in parallel. The host part uses CUDA API calls to send different commands to the GPU. These commands include kernel execution control (e.g. kernel launch), memory management (e.g. memory copy to/from GPU memory), error handling (e.g. querying the GPU to check whether the kernel has terminated with an error), etc.

The programmer configures the kernel to be executed by a given number of GPU threads. A maximum of 1,024 threads are grouped into *thread blocks*[7] as configured by the programmer. Each thread block is assigned to an SMX by a hardware scheduler. Threads within a thread block can synchronize using barriers provided by the GPU hardware. However, there exists no explicit support for inter-block synchronization.

Threads of a thread block are further divided into groups of 32, called *warps*. The threads in a warp execute in lock-step because the cores share the same instruction scheduler. As described earlier, thread divergence will occur if, on a conditional branch, threads of the same warp take different paths, which can lead to serious performance degradations. Inter-warp divergence does not negatively impact performance.

A limited number of thread blocks can be scheduled on the same SMX at each time. The thread blocks currently executing on an SMX are called *online* thread blocks. The *offline* thread blocks are put in a queue and scheduled on the SMXs only after the online blocks complete their execution. The number of online blocks on each SMX depends on the available resources of the SMX and the total resource requirements of the thread blocks. Assuming that an SMX has 64K registers, for instance, no more than 4 thread blocks can be scheduled on the same SMX at a time if the size of the thread block is 256 and each thread requires 60 registers.[8]

Figure 2.3 lists a trivial CUDA program that squares the values of an array. Lines 2-6 define the kernel, which is executed on the GPU; each thread squares the value of a single array element. Lines 9-26 are the code executed on the CPU. Line 14 allocates an array in GPU memory, and data is copied into the array at line 18. Line 19 launches the kernel to be executed by 32 GPU threads. An explicit synchronization function is called at line 20 which blocks host execution until the GPU kernel finishes. After the kernel terminates, the array is copied back to main memory at line 21, and the allocated space on the GPU is freed at line 25.

[5] Kernel is called *program* in AMD/OpenCL's terminology.

[6] GPU threads are called *work-items* in AMD/OpenCL's terminology.

[7] Thread blocks are called *work-groups* in AMD/OpenCL's terminology.

[8] 4 thread blocks would require $4 \times 256 \times 60 = 60K$ registers which is available in the SMX. 5 thread blocks, however, would require $5 \times 256 \times 60 = 75K$ registers which is more than 64K registers available in the SMX.

```
1:  //GPU part
2:  __global__ void squareArray (float *a)
3:  {
4:      int idx = threadIdx.x;
5:      a[idx] = a[idx] * a[idx];
6:  }
7:
8:  //Host part
9:  int main(void)
10: {
11:     float *a_h, *a_d;
12:     int size = 32 * sizeof (float);
13:     a_h = (float *) malloc (size);
14:     cudaMalloc((void **) &a_d, size);
15:
16:     // Initialize host array ...
17:
18:     cudaMemcpy (a_d, a_h, size, cudaMemcpyHostToDevice);
19:     squareArray <<<1, 32>>> (a_d);
20:     cudaThreadSynchronize();
21:     cudaMemcpy (a_h, a_d, size, cudaMemcpyDeviceToHost);
22:
23:     // Use the results ...
24:     free(a_h);
25:     cudaFree (a_d);
26: }
```

Figure 2.3: A trivial CUDA application that squares the elements of an array

Chapter 3

Related Work

In this chapter, we present prior work related to our research. We focus on four different subareas:

1. GPU accelerated applications
2. Core data structures developed for GPU applications
3. Compile-time or runtime systems that improve the performance of existing GPU applications through various optimizations techniques
4. GPU performance models and studies that characterize GPU performance

3.1 GPU accelerated applications

Many studies have reported significant speedups when porting CPU applications to GPUs. These applications stem from a wide array of areas, including: bioinformatics [62], computational finance [15], computational fluid dynamics [93], data mining [20], defence and intelligence [19], electronic design automation [98], imaging and computer vision [25], material science [18], medical imaging [94], molecular dynamics [110], numerical analysis [7], physics [36], quantum chemistry [102], oil and gas/seismic [1], structural mechanics [84], visualization and docking [57], and weather and climate [57]. Table 3.1 summarizes a number of applications from these areas along with their achieved speedups over the corresponding CPU implementations.

In addition to algorithm changes that are required to make an application massively parallel, programmers also need to fine-tune the applications to take into account the underlying micro-architectural characteristics of the GPU in order to be able to achieve good performance. For example, Kumar et al. needed to coalesce accesses to GPU memory to improve the performance of the Expectation Maximization (EM) algorithm [52]. They observed that coalescing memory requests enabled the EM algorithm to exhibit higher computational throughput.

Application and Organization	Speed-up	Software env.	Testbed Hardware environment
Fast Seismic Modeling [1] TOTAL, avenue Larribau	10X	CUDA	GPU: NVIDIA Tesla S1070 CPU: Ten Intel Xeon quad-core 2.0GHz
Visualization of ISO-surface Extraction [8] Nancy Universite	2X	OpenGL	GPU: NVIDIA QuadroFX Go 1400 CPU: Intel Centrino 2GHz
Support Vector Machine [16] Can Tho University	70X	CUDA	GPU: NVIDIA Geforce 8800 GTX CPU: Intel Core 2 2.6 GHz
Decision Trees and Forests [89] Microsoft Research, UK	100X	CUDA	GPU: NVIDIA GTX 280 CPU: Intel Core 2 Duo 2.66 GHz
Particle Swarm Optimization [121] Peking University	11X	CUDA	GPU: NVIDIA GTX 8600 CPU: Intel Core 2 Duo 2.2GHz
Genetic Programming [87] Universit Lille Nord de France	13X	CUDA	GPU: NVIDIA GTX 8800 CPU: Intel Core 2 Duo 2.6 GHz
Genetic Programming for Bioinformatics [54] University of Essex	7.6X	RapidMind [70]	GPU: NVIDIA GTX 8800 CPU: Intel 6600 2.40 GHz
Data Mining High-throughput Screening Data [58] ChemExplorer Co. Ltd	43X-212X	CUDA	GPU: NVIDIA GTX 280 CPU: 4 Intel Xeon 5120 1.86 GHz
K-Means [114] Know-Center, Graz	14X	CUDA	GPU: NVIDIA GTX 9600 CPU: Intel Core 2 Duo E8400 CPU
Digital Forensics [69] University of New Orleans	4.6X	CUDA	GPU: NVIDIA GTX 8800 CPU: Two AMD Opteron 2218 2.6GHz
Back-Propagation Neural Network [90] University of Tecnolgica Metropolitana	63X	CUDA	GPU: NVIDIA Tesla C1060 CPU: Intel Core2 Duo E6750 2.66GHz
Molecular Dynamics Simulations [60] Nanyang Technological University	19X	CUDA	GPU: NVIDIA GTX 8800 CPU: AMD Opteron 2210 1.8 GHz
K-Means [38] Jilin University	40X	CUDA	GPU: NVIDIA GTX 8800 CPU: Intel Pentium D 965 3.7 GHz
Molecular Dynamic Simulation [24] Stanford University	700X	CUDA	GPU: NVIDIA GTX 280 CPU: Intel Core 2 Duo 2.66 GHz
Earthquake Modeling [49] Universite de Pau et des Payse de l'Adour	25X	CUDA	GPU: NVIDIA GTX 8800 CPU: Intel Xeon E5345 2.33 GHz
Quantum Monte Carlo [4] California Institute of Technology	6X	CUDA	GPU: NVIDIA GTX 7600 CPU: Intel Pentium 4 3GHz
Smith Waterman for Scanning of Sequence Databases [59] University of Warsaw	3.5X	CUDA	GPU: NVIDIA GTX 9800 CPU: Conventional CPU
Ultra-fast FFT Protein Docking [86] INRIA Nancy	45X	CUDA	GPU: NVIDIA GTX 285 CPU: Intel Xeon quad-core 2.3 GHz
Density Functional Calculations [112] Nagoya University	10X	CUDA	GPU: NVIDIA GTX 8800 CPU: Intel Core2 Duo 3.0 GHz
QM/MM-QMC Simulation [100] Japan Advanced Institute of Sci. and Tech.	23.6X	CUDA	GPU: NVIDIA GTX 275 CPU: Intel Core i7 920 2.66 GHz
Biomolecular Simulations in the Centisecond Timescale [120] University of Massachusetts	90X	CUDA	GPU: NVIDIA GTX 295 CPU: Two Intel Xeon quad-core 2.83 GHz
Single-cluster Algorithm for the Simulation of the Ising Model [51] Tokyo Metropolitan University	7.9X	CUDA	GPU: NVIDIA GTX 285 CPU: Intel Xeon W3520 2.67GHz
Numerical Solution of Stochastic Differential Equations [42] University of Silesia	675X	CUDA	GPU: NVIDIA Tesla C1060 CPU: Intel Core2 Duo E6750
Fast FEM Deformation Simulation [61] University of Macau	4X	CUDA	GPU: NVIDIA GTX 8800 CPU: Intel Core2 Quad 2.0GHz
Nonequispaced Fast Fourier Transform [92] King's Coll. London	85X	CUDA	GPU: NVIDIA GTX 8800 CPU: Intel Xeon dual-core 2.33 GHz

Table 3.1: Studies on porting applications to GPUs with the claimed speedup achieved.

In the process of developing a data mining translation system on GPUs, Ma et al. used GPU shared memory spaces to dramatically improve the application's overall performance [67]. They found, however, that since the shared memory is small, using it can be challenging.

Govindaraju et al. focused on the use of GPUs caches in a sorting algorithm [31]. They realized that by tiling the application's memory access pattern, the small GPU caches could be utilized more efficiently. This technique significantly reduces the number of GPU memory accesses and improves the performance of the sort algorithm by 2-30x and 6-25x over prior GPU-based and CPU-based sorting algorithms, respectively.

Yudanov et al. observed that the serialization effect of thread divergence hurts the performance of GPU applications noticeably [113]. The authors tracked down and resolved the source of branch divergence in a Simulation of Neural Networks application and achieved 9X speedup compared to a baseline CPU implementation.

Yang et al. designed and implemented a Linpack benchmark on a petascale CPU/GPU supercomputer system [109]. They observed that one of the primary bottlenecks was the bandwidth limitation of the CPU-GPU link. A simple software pipelining technique was used to overlap execution with host-GPU data transfers so as to better distribute data transfers over time and reduce the time the kernel spent waiting for the data. Moreover, they realized that finding the optimal configuration regarding the distribution and size of each task/data-transfer was challenging and could have benefited from an adaptive framework that does this automatically.

3.2 Work on core data structures

Performance of GPU applications often depends critically on the choice of data structure used to store data. A key challenge is that data structures used in GPU applications are typically accessed by 1000's of active GPU threads; even minor access inefficiencies thus translate into substantial performance degradation.

Regular data structures like arrays and matrices are the data structures of choice when the objective is to exploit as much of the computation power and memory bandwidth of GPUs as possible: accesses to these data structures (i) can often be coalesced into fewer (physical) memory transactions, (ii) seldomly cause thread divergence, and (iii) often do not require synchronization.

A large body of work has presented GPU-based applications that use arrays and matrices in their implementations (e.g., [7, 12, 21]). Because matrix multiplication is one of the more important operations in scientific computing, various studies have looked at ways to improve the way matrices are stored and operated on. Hall et al. looked at how matrices can be stored and accessed so as to have a high cache hit rate [33]. Dziekonski proposed a novel way to store sparse matrices on GPUs to achieve high bandwidth memory accesses [17].

Irregular data structures[1], like trees and hash tables, on the other hand, are often avoided by the GPU commu-

[1] Irregular data structures are data structures that their access pattern is data-dependent, and statistically unpredictable.

nity because (i) it is significantly harder to coalesce accesses to these data structures, (ii) costly synchronization is often required for shared accesses and, (iii) thread divergence is more likely (imagine two neighbor GPU threads traversing a tree to find two different elements at different heights). Despite these potential issues, there are cases where such irregular data structures nevertheless improve application performance, primarily because they are a great fit for the application's needs.

Irregular data structures that have been designed for GPUs can be grouped into one of the three categories: (i) *mutable GPU-based structures*, those that are kept in GPU memory and can be accessed and updated incrementally, (ii) *immutable GPU-based structures*, those that are kept in GPU memory but cannot be updated incrementally but are bulk-built and need a full rebuild to be updated, and (iii) *CPU-based structures*, those that are kept in CPU memory but are accessed by GPU cores remotely over the PCIe bus (mostly read-only accesses).

Mutable GPU-based structures: only few existing designs fall into this category. MapCG is a GPU-based MapReduce runtime system that uses a hash table to store key-value pairs [37].

MapCG has been shown to achieve 1.6-2.5X performance speedup compared to earlier implementations that used arrays to store the key-value pairs, which is perhaps surprising given the fact that a MapReduce application can generate many key-value pairs with duplicate keys and thus face a highly contended access pattern to the hash table.

In chapter 6, we present our own design of a hash table which also falls into this category. This hash table was designed primarily for key-value pairs. It (i) accepts variable-length key-value pairs, (ii) groups pairs with duplicate keys on-the-fly and, (iii) keeps the hash table in GPU memory, and while doing that stays operational (and with retained efficiency) even when the hash table contents grow larger than GPU memory.

Immutable GPU-based structures: several hash table and most tree designs fall into this category[22, 39, 45, 76, 103, 34, 83, 99, 118]. Pan et al. use a hash table for a k-nearest neighbor computation [83]. The hash table uses locality sensitive hashing (that hashes similar items to similar buckets) to cluster items that are similar to each other. Their implementation achieves more than 40X speedup over a single-core CPU based implementation.

Alcantra et al. [3] described various techniques to make the hash table more efficient on GPUs. They describe various potential structures for a hash table and analyze insert and lookup operations on each. As part of their evaluation, they reported that the hash table they built offers random access at a higher rate than binary searches through sorted arrays, despite the uncoalesced accesses inherent to hashing.

Foley et al. used a KD-tree to improve the performance of the Raytracer application [23]. They describe that even though the matrix-like grid is more "GPU-friendly", it is a suboptimal data structure for this application when rays have certain attributes. Foley claimed that using a KD-tree provided a speedup of 8 over previous

implementations using the grid data structure.

Sharp described a way to implement a decision tree on GPUs, and showed how it can accelerate an object recognition algorithm [89]. Their implementation achieved 100X speedup over the corresponding CPU implementation.

Luo et al. presented a highly parallel GPU-based implementation of R-trees [64]. R-tree is an index data structure for the efficient management of spatial data. They reported an average speedup of 25 over a CPU-based implementation of the same application.

CPU-based structures: More recent scalable hash table designs fall into this category. Stadium hashing proposes a hash table design where the hash table itself is located in a pinned portion of CPU memory, where it is directly accessed by GPU threads [47]. However, a compact indexing data structure is stored in GPU memory which is used to minimize the number of accesses to CPU memory by having one fingerprint hash token for each item stored in the hash table and having every operation on the hash table first consult the index data structure before accessing the hash table. For instance, on an insert, the GPU thread first uses the index data structure to find an empty bucket, and only then will it access CPU memory to store the data. Stadium hashing reports 2-3 times faster operations over an existing GPU parallel hashing.

Mega-KV is a CPU-based in-memory key-value store for distributed systems that uses a hash table to store the key-value pairs [116]. Mega-KV runs on a single node and stores key-value pairs that are sent to it from other nodes. The hash table is accelerated by a GPU-based index table. Similar to the Stadium hashing's approach, Mega-KV uses the GPU only for the heavy-lifting part of the operations (e.g., scanning the hash table for an empty bucket during an insert, or finding a bucket item during a lookup). However, the actual data is kept on and accessed in CPU memory.

Unlike the solution we present in chapter 6, neither Stadium hashing nor Mega-KV handle key-value pairs with duplicate keys even though they are common in Big Data analytics applications. They both store pairs with duplicate keys as if they are pairs with different keys that happen to map to the same buckets.

3.3 Runtime and Compiler-assisted Systems

A number of runtime and compiler-assisted systems have been designed for GPUs with the goal of improving the performance of existing applications. Two classes of these systems that are of particular interest to us are (i) CPU-GPU communication management systems with the objective of automating and/or optimizing CPU-GPU data communication and (ii) GPU memory optimization systems with the objective of optimizing the memory accesses of existing applications (e.g. by coalescing memory accesses or by exploiting GPU's fast shared memory).

We present two systems in chapters 4 and 5 that fall under this category. The first system, BigKernel, (i) automates and optimizes CPU-GPU communications and (ii) optimizes GPU memory accesses. The second system, S-L1, implements a level-1 cache system entirely in software to improve the performance of GPU memory subsystem.

3.3.1 CPU-GPU Communications Management Systems

Managing CPU-GPU data communications is a well-known challenge both from a programmability and from a performance point of view. Most prior work addresses this challenge only from a programmability point of view [27, 56, 108], but some also consider performance, which is more closely related to our research goals [55, 119].

Jablin et al. automates CPU-GPU communication to increase GPU programming convenience, however without having the objective of improving the performance [41]. They used compiler technology to statically analyze the code of an application and insert appropriate runtime library functions into its code to transfer data to/from GPU memory. To do this, the data structures being used by the kernel are determined based on the kernel's arguments. Next, type-inference is used on each data structure to statically determine if the data structure contains pointers or non-pointer data. If it contains non-pointer data, it is simply transferred to GPU memory before the corresponding kernel is invoked. If the data structure contains pointers, however, not only it will be transferred to GPU memory, but the data structure being pointed to by those pointers will also be transferred to GPU memory.

The static type-inference scheme the authors use makes the scheme inapplicable for applications with recursive data structures. Furthermore, this work does not address the difficulty in executing kernels that process datasets larger than the size of GPU memory. In other words, the programmer herself needs to split the input data into smaller chunks – and call the kernel function multiple times, each time to process a different chunk of data – and make sure that the total size of data required by each kernel invocation does not exceed the size of GPU memory.

In a more recent work by the same group, Jablin et al. designed another system to automatically manage recursive data structures without static analysis [40]. They realized that moving toward a runtime approach (as opposed to the static type-inference method they used in their previous work) improves the applicability of the data management system for a wider range of GPU applications. Similar to the previous work, this new system determines the data structures used based on the kernel arguments. These data structures are scanned before kernel invocation to find potential pointers. For each potential pointer found, the pointee data structure is added to the list of data structures that should be transferred to GPU memory and also scanned for potential pointers as well (to identify recursive data structures).

The new system also does not handle datasets larger than the size of GPU memory and puts the burden of

splitting and size-checking the data structures on the programmer. Additionally, it is assumed that the pointers used by an application are always stored as direct pointers (i.e., immediate address) and are not displacements (e.g. basePtr + displacement) or indexes (e.g. basePtr[index]). These limitations, for example, make the system incapable of handling applications that access a hash table since hash tables are typically indexed into using hashes of other data values.

VAST is another CPU-GPU communication management system that provides OpenCL programs the illusion of having a larger GPU memory space [55]. Based on the available GPU memory, VAST partitions the data parallel workload into chunks and extracts the working set required for the computation by first running a variation of the kernel code, called *inspector*, that compiles a list of those pages of memory being accessed by GPU threads during the computation and sends the list to the CPU. The CPU then transfers those pages to GPU memory where the data is accessed through a page table. A downside of this approach is that each data access is transformed to go through the page table before accessing the actual data, which incurs extra memory accesses. Nevertheless, the authors report an average speedup of 2.6 over the baseline CPU implementations of benchmark applications.

Komoda et al. proposed a library for OpenCL that automatically overlaps computation with data communication [50]. Using this library, programmers describe the memory usage pattern of their applications in the form of a graph abstraction called Stream Graph [96]. In this abstraction, kernel(s) and I/O streams (e.g. a stream of data in CPU memory) are nodes of the graph and communication paths amongst the nodes are the edges. When applied to four benchmark applications, the overlapped computation and communication improves the performance of the applications by an average of 58%. However, the programmer is still required for splitting the data into smaller chunks and also is required to provide information such as the size of the data structures being transferred.

Pai et al. propose a system that automates CPU-GPU memory management and also improves the performance of CPU-GPU communication by not transferring redundant data between CPU and GPU [82]. The authors observed that other systems that automate CPU-GPU data transfers often transfer data that is not used at all or transfer a data item multiple times. To reduce the volume of data transfers to what is actually necessary, this system uses a coherence model approach where data is transferred – from CPU to GPU or from GPU to CPU – only if it is does not exist at the destination or if its available version is stale. Using this system, a set of benchmark applications exhibit an average speedup of 1.29 over the system proposed by Jablin et al., described above [41]. The system proposed by Pai et al. does not overlap computation and data communication.

Finally, Nvidia recently equipped its GPUs with a demand-paging hardware mechanism, enabling a single virtual space across CPU and GPU memories [79]. This single address space allows a GPU program to access data located in CPU memory without first having to explicitly copy it to GPU memory; the demand-paging hardware will pause the execution if the accessed data is in CPU memory and transparently copy it to GPU memory before resuming the execution. A preliminary study on this demand-paging support, however, report on

noticeable performance inefficiencies, suggesting that some optimizations are still needed before the mechanism will perform reasonably well for different types of applications [119].

3.3.2 GPU Memory Optimization Systems

Despite having high theoretical memory bandwidth, GPU memory is often unable to efficiently satisfy the high number of concurrent memory requests made by running GPU threads. Two ways this issue has been addressed are (i) coalescing memory accesses to GPU memory and (ii) using GPU shared memory or L1 caches to cache the data and, consequently, reduce the number of memory accesses that need to be serviced by GPU memory.

CUDA-lite is an enhancement to CUDA developed by Ueng et al. that uses shared memory as a fast scratch-pad to reduce the number of accesses to GPU memory [101]. In CUDA-lite, a source-to-source translator is used to transform the code of a loop-based kernel[2] so it can achieve optimized tiling of GPU memory data. More precisely, a tile of data, which is predicted to be used in future iterations of the kernel's loop, is read in a coalesced fashion, from GPU memory using a single memory transaction and stored in the shared memory from where it is then accessed more efficiently. To predict which data is used in future iterations of the kernel's loop, the programmer has to annotate the source code to specify the loop configuration (e.g. start and end iterations) and the location of the data item. They were able to obtain 2-17X speedup over applications with non-coalesced memory accesses [101].

Yang et al. proposed a number of compiler optimizations to increase the effective utilization of GPU memory bandwidth and improve the efficiency of thread parallelism on the GPU devices [111]. Similar to CUDA-lite, they load data tiles (that are going to be used in future loop iterations) from GPU memory in a coalesced fashion and store them in the shared memory for later use. Additionally, to improve thread parallelism efficiency, threads in different thread blocks are analyzed for their memory access patterns (at compile-time) and those with sharing locality are merged into one thread, enabling data reuse through registers and shared memory.

Zhang et al. conducted a comprehensive study on the difficulties and benefits of removing irregularity from memory accesses [115]. The authors noted that since irregular memory accesses by the threads of a warp are often to disjoint locations of memory, their performance is significantly lower than for coalesced memory accesses. To address this, the authors proposed a software-based system in which, at runtime, the CPU reorders the data of irregular data structures before sending them to the GPU to co-locate the data items that will be accesses at the same time by the threads of a warp, causing the accesses to GPU memory to be coalesced. The downside of this work is that the CPU (which is often less efficient for parallel workloads) is required to partially run the kernel code to extract the irregular addresses the GPU threads will access, thus limiting the performance gains.

[2] A kernel that does the core of the computation inside a loop.

Nevertheless, applying their system on a variety of applications, speedups between 1.07 and 2.5 were reported.

Finally, CudaDMA provides a library that allows the programmer to copy data to shared memory and use the data from there [6]. They adopt a *producer-consumer* approach in which threads executing a kernel are divided into two groups: (i) *producer threads* whose only job is to load the data from GPU memory into shared memory and (ii) *consumer threads* that will do the computation using the data in shared memory. Producer and consumer threads are put in different warps so as to avoid thread divergence. To employ CudaDMA, the programmer uses the provided APIs to determine producer and consumer threads and also to register the data that needs to be loaded to shared memory. The downside of this work is that the programmer is responsible for setting the number of producer and consumer threads, and assigning the proper size of shared memory to different threads which is a non-trivial task. Speedups of between 1.15 and 3.2 were reported when applying CudaDMA on several kernels from scientific applications.

3.4 GPU Performance Characterization

GPU performance models and other studies that characterize GPU performance are often used to help GPU programmers predict and optimize their applications. These studies often analyze many of the micro-architectural characteristics of a GPU (sometimes obtained through reverse-engineering efforts) and provide an in-depth understanding as how each component of a GPU can become performance-limiting. This section describes some of the more important studies that characterize and predict GPU performance.

Wong et al. used micro-benchmarking to understand the architectural characteristics of a modern GPU [106]. The collected measurements are useful to those wishing to model GPU performance, since most of the GPU micro-architectural characteristics are not publicly disclosed by the GPU vendors. For example, while there is no information on the existence of TLBs for GPU memory in GPU documentation, this work reports on the measured effect of two TLB levels for GPU memory.

One of the more interesting (and for us relevant) performance modeling efforts is the one by Gomez et al., who modelled asynchronous data transfers between host and GPU memory [30]. Based on the execution time of the applications and the time it takes to transfer data between host and GPU memory, their model estimates the potential performance improvement that would be obtained from overlapping computation and data communication. In addition, the model enables programmers to determine how best to partition the application's computation.[3]

Baghsorkhi et al. propose an analytical model to predict the performance of a GPU kernel executing on a generic GPU architecture based on its (a) compute-to-memory-access ratio, (b) coalesced memory accesses,

[3]Given an application which processes D data records that can be processed independently, a programmer could decide to cluster them into n segments. Thus, each of the segments will contain $\frac{D}{n}$ of the data records.

(c) thread divergence ratio, and (d) shared memory usage of each thread block [5]. These four criteria are sufficient to determine the performance of a GPU application when its memory footprint is smaller than GPU memory size.

Others have also studied the characteristics of GPU memory and GPU caches [26, 106]. Jia et al. characterized GPU L1 cache locality and provided a taxonomy for reasoning about different types of access patterns and how they might benefit from L1 caches [43]. Tore et al. provided insights into how to tune the configuration of GPU threads to achieve higher cache hit rates. They also showed how the L1 impacts the performance of a handful of simple kernels [97].

Table 3.2 lists other studies that characterize GPU performance.

Description	**Organization**
GPU power efficiency modeling framework [85]	University of Utah
GPU Statistical power consumption analysis and modeling [68]	University of Houston
Performance modeling in multi-GPU environments [88]	Northeastern University
Thread divergence modeling [13]	Universidade Federal de Minas Gerais
GPU Statistical power modeling [75]	Tokyo Inst. of Technology
Modeling of CPU-GPU workloads [46]	Georgia Institute of Technology
Memory performance modeling and estimation [48]	Arizona State University
Visual performance analysis and memory access modeling [2]	Virginia Tech
GPU power and performance characterization [44]	Virginia Tech
Thread divergence modeling and optimization [14]	Universidade Federal de Minas Gerais
Modeling the performance of GPU application based on its CPU code [71]	Argonne National Library
Statistical model on power and performance of GPU executions [117]	Louisiana State University
Memory access model [65]	Washington University in St. Louis
GPU power analysis using tree-based methods [10]	University of Florida
Performance modeling and evaluation of memory hierarchy [63]	Washington University in St. Louis
A performance analysis framework [91]	Georgia Institute of Technology
Performance model for bandwidth constrained applications [66]	Washington University in St. Louis
Performance models for atomic operations on GPU shared memory [29]	University of Cordoba
Timing model for GPU performance [53]	University de Rennes

Table 3.2: Several other studies that characterize the performance of GPUs.

Chapter 4

BigKernel

In this chapter, we describe *BigKernel*, a mixed compile-time, runtime scheme that aims to optimize CPU-GPU data communication as well as GPU memory accesses in Big Data applications. We first present a short overview of BigKernel in Section 4.1, and then present the design and implementation in more detail in Section 4.2. Section 4.3 describes various optimizations we applied to BigKernel. We close the chapter by presenting our evaluation of BigKernel in Section 4.4.

4.1 Overview

BigKernel's goal is to optimize CPU-GPU data communication as well as GPU memory accesses by prefetching data within a four-stage pipeline. The key idea behind BigKernel is to have GPU threads identify ahead of time, yet online, which data will accessed by which threads in their near-term computations, transfer this information to the CPU, and then have the CPU assemble the data and transfer it to GPU memory prior to when the GPU threads access the data.

This scheme has a number of potential advantages. First, the amount of data transferred over the PCIe link from CPU memory to GPU memory is often reduced, because only the data being accessed GPU-side is transferred (as opposed to all data). Secondly, it allows the CPU to assemble the data in a way that increases coalesced data accesses on the GPU for more efficient GPU memory accesses.[1] Finally, BigKernel significantly simplifies the programming model, since the programmer need not deal with and manage (i) buffers, (ii) the transfers of data between CPU and GPU, and (iii) the reorganization of data so as to enable coalesced accesses.

[1]Note that many of our target applications are inherently incapable of exhibiting coalesced memory accesses in their original form. The records being processed are often large and therefore, only a few of them can be accessed in each memory transaction, causing the memory accesses of consecutive threads to be non-coalesced. Moreover, in applications with variable-length records, consecutive threads cannot be easily assigned to process consecutive records in an interleaved fashion because it is difficult for consecutive threads to identify the starting memory location of consecutive records without accessing the previous records.

BigKernel allows the programmer to write a GPU program with arbitrarily large regular data structures as if (pseudo-) virtual memory were available GPU side. The program is compiler-transformed to one that automates the management of buffers, the transfers, and the layout of GPU-side data in a way that is transparent to the programmer. Moreover, the fact that the transformed program only invokes a single kernel once for the entire computation means that the kernel context (i.e., registers and GPU shared memory) is available throughout the entire computation without having to manage it separately, as would be the case when kernels are invoked iteratively.

On the other hand, one should note that BigKernel is not a general framework. It targets only data processing applications that operate on independent input records, although we argue that this is a large and important subset of applications. Moreover, BigKernel involves a number of tradeoffs. For example, BigKernel uses twice as many GPU threads, potentially limiting the degree of parallelism GPU-side, although we have found this not to be an issue in practice since GPU core utilization tends to be low for the class of computations being considered. BigKernel also uses more CPU-side resources compared to traditional schemes – i.e., more CPU threads, more memory accesses, and more buffers that are pinned so that they cannot be paged out – which may impact other concurrently running processes on the CPU.

Our performance evaluation, which we present in Section 4.4, shows that across the 6 benchmarks we studied, BigKernel outperforms corresponding single and double buffering (a scheme that overlaps communication with computation to improve the GPU performance) implementations by up to 4.6X and 3.1X, respectively, and on average by 2.6X and 1.7X, respectively. Compared to corresponding multi-threaded CPU implementations, BigKernel executes up to 7.2 times faster and 3.0 times faster on average.

As far as we know, BigKernel[2] is:

1. the first scheme to improve on the performance of state-of-the-art double-buffering schemes for GPUs;
2. the first scheme to automate CPU-GPU data transfers for large data sets without requiring the programmer to split the data or annotate the code;
3. the first scheme to provide the continuous execution of a single kernel on arbitrarily large input/output data sets.

4.2 Design and Implementation

BigKernel organizes a computation into four pipeline stages, illustrated in Figure 4.1:

[2]The name *BigKernel* was chosen because (i) its target applications are those with Big Data-style processing of large datasets, (ii) the programmer can write a single ”big” kernel that can operate on all data, even if the data does not fit in memory, and (iii) the kernel that is generated is big compared to a traditionally implemented kernel; e.g., a kernel that is implemented in 70 LOC is transformed into one that has over 500 LOC.

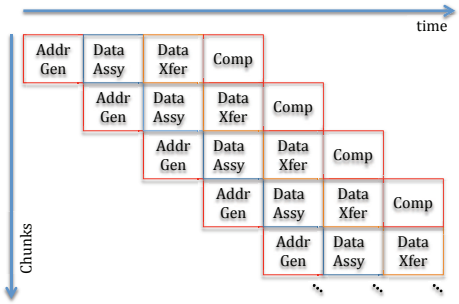

Figure 4.1: Four-stage pipeline.

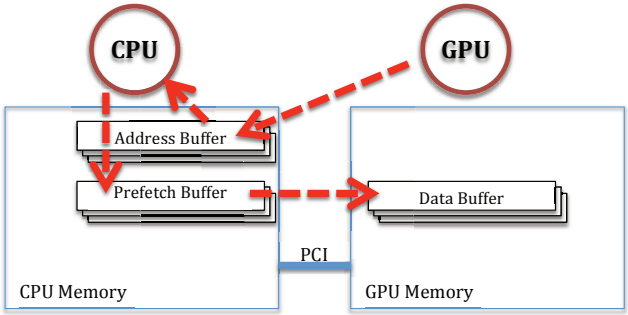

Figure 4.2: BigKernel buffers

1. **Prefetch address generation** (GPU-side): GPU threads calculate the addresses of the data needed by the computation threads later in Stage 4. It does this by running a variant of the computation that collects the addresses of memory accesses ahead of the actual computation that performs the memory accesses. The code to generate the memory addresses is obtained from the original GPU kernel source code by removing all statements except (i) those that contribute to control flow, (ii) those that contribute to memory access address calculations, and (iii) the memory accesses themselves. The memory access instructions are then transformed to record the address of each access instead of making the memory access. The addresses are recorded in a CPU-side *address buffer* (See Figure 4.2).

2. **Data assembly** (CPU-side): Based on the addresses generated in Stage 1, the prefetch data is assembled into a *prefetch buffer* in CPU memory. .

3. **Data transfer**: the GPU DMA engine transfers the contents of the prefetch buffer to a *data buffer* in GPU memory.

4. **Kernel computation** (GPU-side): GPU threads execute the actual computation using the prefetched data. The code for the computation is obtained by transforming the original GPU kernel to use the prefetched

data in the data buffer instead of the data at original target memory address.

4.2.1 A simple motivating example

To provide more detail using an example, consider part of a K-means computation:

```
clusterKernel(particles, numP, clusters) {
   for( i = 0; i < numP; i++)
       particles[i].cid = findClosestCluster(
                particles[i].x, particles[i].y,
                particles[i].z, clusters);
}
```

Figure 4.3: K-means computation.

where `particles` represents a particle array with `numP` elements that does not fit in GPU memory, `clusters` represents an array of clusters that fits entirely in GPU memory, and `findClosestCluster()` returns the id of the cluster closest to the target particle. The function `clusterKernel()` iterates on the particles array to assign them to the closest cluster.

Traditional solution

Because the particle array does not fit in GPU memory, the array would traditionally be processed in chunks with the following GPU code invoked iteratively:

```
clusterKernel(particles, numP, clusters) {
   start = myParticleStartIndex( threadId, numP );
   end = myParticleEndIndex( threadId, numP );
   for(i = start; i < end; i++)
      findClosestCluster(particles[i].x,
      particles[i].y, particles[i].z, clusters);
}
```

Figure 4.4: K-means GPU-side code.

In each iteration, the above code first determines the start and end of its assigned chunk before updating the cluster values in the particles array as before

The corresponding CPU code (i) allocates space for the cluster array GPU-side and copies the cluster array to GPU memory, and then (ii) iteratively copies the next chunk of the particles array to GPU memory before invoking the GPU `clusterKernel()` on the chunk (`pElements` is equal to the number of particles that fit in the GPU buffer `pBuf`):

```
cudaMalloc(d_clusters, clSize);
cudaMemcpy(d_clusters, clusters, clSize);
cudaMalloc(d_pBuf, GPUBufSize);
for(offset=0;offset<pSize;offset+=GPUBufSize) {
  cudaMemcpy(d_pBuf, particles + offset, GPUBufSize);
  clusterKernel<<<>>>(d_pBuf, pElements, d_clusters);
}
```

Figure 4.5: K-means CPU-side code.

Note that the CPU and GPU code above uses single buffering, where there is no communication and computation overlap. In practice, double buffering would be used for efficiency, but this would make the code significantly more complex.

BigKernel solution

Using BigKernel, the programmer no longer needs to partition the large particle array into chunks and manage the transfers. Instead, she would provide the following CPU-side code that assumes the existence of an arbitrarily large `d_particles` array in GPU memory that is (virtually) allocated with `streamingMalloc()` and mapped to the CPU-side particles array `particles` with `streamingMap()`:

```
cudaMalloc(d_clusters, clArraySize);
cudaMemcpy(d_clusters, clusters, clArraySize);
streamingMalloc(d_particles, pArraySize);
streamingMap(d_particles, particles, pArraySize);
clusterKernel<<<>>>(d_particles, numP, clusters);
```

Figure 4.6: K-means CPU-side code when using BigKernel.

The corresponding GPU-side kernel code written by the programmer (shown in Figure 4.4) remains unchanged but gets invoked only once.

Note that the K-means kernel accesses two types of data structures: the cluster array which is explicitly copied to GPU memory and does not involve BigKernel, and the particle array that does not fit in GPU memory and is mapped. BigKernel manages the accesses to the latter.

Description of pipeline

We now describe each of the four stages of the BigKernel pipeline in more detail, using the above running example.

Prefetch address generation: The prefetch address generation code is obtained from the original GPU kernel

by transforming the d_particles read accesses in the for-loop to instead store the addresses in an addrBuf array CPU-side:

```
counter = 0;
for( i = start; i < end; i++ ) {
  addrBuf[counter++][threadId] = &particles[i].x;
  addrBuf[counter++][threadId] = &particles[i].y;
  addrBuf[counter++][threadId] = &particles[i].z;
}
```

Figure 4.7: GPU-side (transformed) code to generate prefetch addresses in BigKernel.

This transformation is done by the compiler. Currently, our transformations are relatively simplistic in that they cannot deal with indirections or flow control based on application data that may be modified; if this is the case, then the transformation simply defaults to fetching all data, making the resulting code similar to the double-buffering scheme.

Since GPU threads in a warp execute in lock-step, on each address generation instructions, an aggregate of *warp-size* x *address-data-size* bytes of data is transferred to CPU memory (i.e., 256 bytes if *warp-size* is 32 and *address-data-size* is 8 bytes). Storing addresses in GPU memory and then sending them to CPU memory in bulk is an alternative design choice, but we found that doing so consumes substantial portion of GPU memory bandwidth, which in turn reduces the memory bandwidth available to data accesses during the kernel computation stage, degrading the overall performance.

This data prefetching approach has three potential downsides. First, if the same data item is accessed multiple times in the code then it will be be transferred multiple times, leading to extra overhead. However, we have not found this to be the case with the applications we examined, and believe that it is rare in Big Data-style data processing applications. Second, when characters (which are typically 1-byte) are accessed then the communication overhead of transferring the addresses (which are typically 4 or 8-bytes) is far greater than the overhead of transferring the characters. We address this issue in the next section. Finally, the CPU-side address buffer must be pinned (i.e., be non-pageable) so that it can be accessed by the GPU DMA engine. While this consumes physical CPU memory that cannot be made available to other processes, this should rarely become an issue given today's CPU memory sizes.

Data assembly: A dedicated CPU thread is responsible for fetching the target data element from the particles array for each address in the address buffer and then placing the data in the prefetch buffer, which also must be a pinned buffer.

Note that the layout in the prefetch buffer automatically results in coalesced accesses after the buffer has been transferred to GPU memory. This is because the addresses were stored into the address buffer in the order they

were accessed by the GPU threads, so data accessed at the same time will be adjacent to each other.

This data assembly process has one potential disadvantage. Because data assembly occurs CPU-side, it involves twice as many memory accesses CPU-side compared to traditional GPGPU applications. In traditional GPGPU applications, data for the GPU is first copied to a pinned buffer, resulting in a CPU-side read and write for each data element. However with BigKernel, the address is first DMAed to memory by the GPU, the CPU then reads the address before copying the target data to the pinned prefetch buffer, resulting in two reads and two writes for each prefetched data element.

Data transfer: The data transfer stage is initiated by the CPU thread that performed the data assembly but will be executed by the GPU DMA engine, allowing CPU and GPU cores to concurrently do other work.

Kernel computation: The computation code is (compiler-) generated from the GPU kernel by transforming the accesses to the `particles` array in the kernel `for`-loop to instead access the data in `dataBuf`, which was previously transferred from the CPU:

```
counter = 0;
for( i=start; i < end; i ++ )
   findClosestCluster (
            dataBuf[counter++][threadId],
            dataBuf[counter++][threadId],
            dataBuf[counter++][threadId],
            clusters
            );
```

Figure 4.8: GPU-side (transformed) code to use the prefetched data.

Four-stage pipeline: Figure 4.9 shows the implementation of the four-stage pipeline depicted in Figure. 4.1, assuming a single thread block. Unlike what is shown in Figure 4.1, the time it takes for each stage to complete will vary in practice, depending on the application. However, prefetch address generation consistently takes the least amount of time across all applications we experimented with.

Additional details on Figure 4.9

The CPU launches twice as many GPU threads as specified in the original program. Half are responsible for generating the prefetch addresses and the other half are responsible for the computation. The GPU threads must be launched in a way such that each warp contains only address generation threads or only computation threads, but not both — otherwise the kernel will suffer from thread divergence.

The outermost `for`-loop of the kernel (lines 11-36) processes one chunk of data at a time. The address generation stage (lines 16-21) ends when `addrBuf` is full, at which time all the address generating threads first

```
01: //GPU side:
02: clusterKernel(particle, numP, clusters, addrBuf, addrBufSize, dataBuf, dataBufSize)
03: {
04:  // 1 addrGen and 1 comp thread assigned same tid
05:  tid = getVirtualThreadId(threadId);
06:
07:  start = myParticleStartIndex(tid, numP);
08:  end = myParticleEndIndex(tid, numP);
09:
10:  i = start;
11:  for(; i < end;) //each iteration: processing one chunk
12:  {
13:    if(isAddrGenThread(threadId))
14:    {
15:      counter = 0;
16:      for( ; (i < end) && (counter < addrBufSize); i ++)
17:      {
18:        addrBuf[counter ++][tid] = &particle[i].x;
19:        addrBuf[counter ++][tid] = &particle[i].y;
20:        addrBuf[counter ++][tid] = &particle[i].z;
21:      }
22:
23:      barrier_addrGenThreads() ;
24:      signal_addrReady() ;
25:    }
26:    else // computation thread
27:    {
28:      counter = 0 ;
29:      wait_dataReady() ;
30:
31:      for( ; (i < end) && (counter < dataBufSize); i ++)
32:      {
33:        findClosestCluster(dataBuf[counter ++][tid],
                    dataBuf[counter ++][tid], dataBuf[counter ++][tid]);
34:      }
35:    }
36:  }
37: }
38:
39: //CPU Side:
40:  cudaMalloc(d_clusters, clArraySize);
41:  cudaMemcpy(d_clusters, clusters, clArraySize);
42:
43:  cudaMalloc(d_dataBuf, dataBufSize);
44:  pinnedMalloc(h_addrBuf, addrBufSize);
45:  pinnedMalloc(h_pBuf, pBufSize);
46:
47:  clusterKernel<<<>>>(d_particle, numP, d_clusters, numCl, h_addrBuf,
          addrBufSize, d_dataBuf, dataBufSize);%
48:
49: while(GPUKernelisRunning())
50: {
51:   wait_addrReady();
52:   for(i = 0; i < numAddr; i++)
53:         h_pBuf[i] = *(particles + addrBuf[i]);
54:
55:   cudaMemcpyAsync(d_dataBuf, h_pBuf);
56:   signal_dataReady() ;
57: }
```

Figure 4.9: Implementation of the four-stage pipeline.

barrier (line 23) and one of the threads in the thread block signals the CPU that the addresses are ready. The CPU waits until the addresses are ready (line 51) and then assembles the data (lines 52-53), copies the data to the GPU (line 55) and then signals the GPU computation threads that the data is ready (line 56), at which time the computation stage (lines 31-34) can commence.

Some details were omitted from the code in Figure 4.9 for simplicity. For example, multiple instances of each buffer are required to allow for concurrency (although the code only shows one). At minimum, two of each are required so that one can be produced while the other is still being consumed.

Writes to mapped data: Writes to mapped data are handled similarly to the way reads are handled: each write results in the writing of (i) the target address to an address buffer CPU side and (ii) the data value to a write buffer GPU side. The write buffer, once full, is transferred to CPU memory by the DMA engine. The write buffer is kept in GPU memory to exploit the high memory bandwidth of GPUs via coalesced accesses. It is then transferred in bulk to CPU memory, which better utilizes the PCIe bus. This requires two extra sets of buffers: one GPU side to collect the writes, and one CPU side to which the data is transferred. This also adds two stages to the pipeline: one for the data transfer back to CPU memory and one for a CPU thread to process the transferred data and update the target data structure. The additional buffers and the additional pipeline stages are added only if kernel code is determined to contain write instructions to the mapped data.

Multiple GPU thread blocks: The examples and code above assumed all threads were running within one GPU thread block. Supporting multiple thread blocks adds a few complications, which, however, can be handled through straightforward compiler transformations. A separate set of buffers (including address buffer, prefetch buffer, data buffer, write buffer, and another address buffer if mapped data is written to) is needed for each thread block both CPU- and GPU-side. A separate CPU thread for each GPU thread block is responsible for data assembly CPU-side. Threads within a thread block need to be organized so that half of them are responsible for prefetch address generation and the other half are responsible for computation so that each computation thread can run in the execution context of the corresponding address generation thread (but prefetching threads and computation threads must be assigned to different warps, as explained earlier).

4.3 Optimizations

In this section we present four optimizations we applied to BigKernel and explain how they contribute to the overall performance of the applications run with BigKernel.

4.3.1 Pattern recognition

To reduce the amount of address information that needs to be transferred to the CPU, the address generation stage makes use of a pattern recognition component that attempts to extract patterns from the memory addresses it generates and then only sends the pattern. Such pattern recognition, if successful, is particularly impactful performance-wise when dealing with text-based input data, since an address (4 or 8-bytes) would otherwise be required for each character accessed (1-byte).

Each address generation thread starts by generating a few addresses, storing them in a private temporary address buffer.[3] The number of addresses generated is dictated by the size of the buffer, which is typically a few tens of bytes. It then invokes a pattern recognition function to identify a potential pattern from the stored addresses. A pattern, if found, consists of a base address and a number of strides between subsequent addresses. For instance, if stored addresses are 0x00100, 0x00105, 0x00110, 0x00115, then the pattern would be *[base_address: 0x00100, stride(s): 5]*. If no pattern is found, then the addresses collected in the temporary buffer are copied to the CPU-side address buffer, and address generation continues as described earlier in Section 4.2.

If a pattern is detected, then the address generation thread continues generating data access addresses, but now verifies that each subsequently generated address follows the identified pattern. If it does not, then address generation is started over from the beginning, this time without attempting to identify a pattern and writing the addresses to the CPU-side address buffer.

If all subsequently generated addresses adhere to the pattern, then the pattern (instead of the addresses) is written to CPU memory, and a signal is sent to the CPU indicating that a pattern was found.

This pattern recognition scheme is rather simplistic, but we have found it to be effective with our benchmarks — see Section 4.4. One can easily conceive of ways to extend it and make it more versatile (e.g., allow patterns to change midstream).

4.3.2 Data locality in assembling data

Compared to traditional double-buffering implementations, BigKernel incurs extra memory accesses during the data assembly stage CPU-side. If access patterns are provided by the prefetch address generation stage then BigKernel is able to schedule memory accesses during the data assembly stage in a way that improves memory access locality and thus reduces the overhead of the data assembly.

To obtain the prefetch data specified by a pattern, instead of reading the data items in the order they are needed by the GPU computation threads, we read all of the prefetch data for one GPU thread at a time. This results in increased data locality in CPU reads, because each GPU thread tends to access consecutive data. The fetched data

[3] Preferably these temporary buffers are allocated in GPU shared memory, but if there is not enough space there because it is needed by the computation, the buffer is allocated in GPU memory.

is, however, still stored in the prefetch data buffer in the order they will be accessed GPU-side. If multiple data structures are mapped and accessed by the GPU, then we additionally read the data from each structure separately.

We focus on improving memory access locality when reading data as opposed to when writing data, because we found that the cost of these reads is far higher than the cost of the writes due to the processor's write buffers.

4.3.3 Synchronization

Synchronization in GPGPU applications is complicated by the intricacies of the GPU hardware. In particular, there is no signaling mechanism between CPU and GPU beyond using flags located in memory and busy waiting for a specific flag value. Hence, one would want to implement synchronization so that the number of memory accesses required is minimized, especially on the GPU because of the large number of threads that execute there.

The first three stages of the BigKernel pipeline are producers for their following stages: the address generation stage produces addresses for the data assembly stage, which produces data for data transfer stage, which produces data for computation stage. For each buffer used in the pipeline, proper synchronization is required to ensure that consumption of the buffer data does not commence before the data has been produced, and that data for a buffer is not produced until the buffer has previously been consumed.

The GPU signals the CPU at the end of the address generation stage by setting a flag in CPU memory. The CPU busy waits on that flag before it starts the data assembly stage. The GPU cannot signal the CPU until all of the address generating threads have completed their stage. Hence, the address generation threads first barrier at the end of their stage before one of the threads signals the CPU. We use the `bar.red` GPU instruction to barrier, because it is efficient and can barrier a given number of threads.

No synchronization is needed between the data assembly stage and the data transfer stage, because the latter is initiated by the CPU thread after it finishes assembling the data.

The DMA engine knows when the data transfer stage has completed, but there is no mechanism for the DMA engine to signal the kernel computation threads that this has occurred. Our solution to this problem relies on the fact that the DMA engine performs data transfers in order. After the CPU instructs the GPU DMA engine to transfer the data buffer (using *cudaMemcpyAsync*), it instructs the DMA engine to copy a flag to a specific location in GPU memory that indicates the data transfer has completed. The flag will not be transferred until all of the data buffer has been transferred.

Instead of having each GPU computation thread busy wait on that flag, only one computation thread is assigned that task. All the other computation threads barrier (again using `bar.red`) to ensure they do not start the computation phase until the flag has been set.

To prevent subsequent address generation stages from overwriting an address buffer that has not yet been

consumed, we barrier all threads in a thread block once for each chunk iteration.[4] Each address generation thread in iteration n synchronizes with the computation threads in iteration $n-3$. This relies on the fact that when a computation stage starts, all three stages prior to it have completed and the buffers of the previous stages can safely be overwritten.

Synchronization between threads across different thread-blocks is not needed because both computation threads and their corresponding address generation threads are packed into the same thread-block and they interact with a separate CPU thread responsible for their data prefetching.

4.3.4 Buffer allocation: active vs. inactive thread-blocks

To ensure efficient use of memory resources both CPU- and GPU-side, BigKernel allocates data and address buffers only for active thread-blocks, reusing them when inactive thread-blocks become active[5] (which only occurs when a resident active thread-block retires). The benefit of allocating buffers only for active thread-blocks is that buffers can be made larger, potentially improving performance by reducing the number of synchronization points.

We use a hybrid compile-time, runtime method to identify the number of thread-blocks that will be active. First, the resource usage required by a thread block, R_{tb}, is determined at compile-time and provided as a constant value in the application's code. The resources provided by the GPU hardware, R_{GPU}, is then probed at runtime (using provided API functions). The number of active thread-blocks is then calculated as:

$$min(numSetBlocks, (R_{tb}/R_{GPU}))$$

where $numSetBlocks$ is the number of thread-blocks set by the programmer as the argument of the kernel invocation.

4.4 Experimental Evaluation

Experimental Setup

Our baseline hardware infrastructure consists of a 3.8GHz Intel Xeon Quad Core E5 with 8 hardware threads and 10MB of combined L2/L3 cache, connected to 16GB of quad-channel memory clocked at 1800MHz. All GPU kernels were executed on an NVIDIA GeForce GTX 680 GPU with 1,536 computing cores, each running at 1020MHz, and 2GB of GPU memory. The GPU video card is connected to the system with a PCIe Gen3 x16 link interconnect.

All GPU-based applications were implemented in CUDA, using CUDA and GPU driver release 5.0.35 in-

[4] An alternative is to use full/empty flags for each buffer, but this increases the number of data transfers and the amount of busy waiting.

[5] The number of thread-blocks that become active depends on the resources (i.e. registers and shared memory) that each thread-block requires and the resources provided by the GPU.

stalled on a 64-bit Ubuntu 12.04 Linux with kernel 3.5.0-23. All applications are compiled with the corresponding version of the *nvcc* compiler using optimization level three.

For our experiments, we ran six applications with a range of different data access patterns.[6] In the description of applications, we use the term *mapped data* to describe data that is automatically managed by BigKernel. Data that is not mapped typically fits entirely in GPU memory and is manually copied to/from GPU memory by the programmer.

K-means: partitions n particles into k clusters so that particles are assigned to the cluster with the nearest mean. The mapped input data consists of an array of particles, each containing particle's coordinates, its clusterId, and a few other data values. The kernel reads particle coordinates and sets its clusterId. The clusterIds have to be written back to CPU memory, so they are dealt with as mapped output data.

Word Count: counts the number of occurrences of each word in a document. The mapped input data consists of a text file. There is no mapped output data (the result is stored in a hash table which fits entirely in GPU memory, and is copied back to CPU memory at the end of the computation).

Netflix: predicts user preferences of movies by calculating the correlation between users ratings [11]. The mapped input data consists of an array of records, each containing movie user ratings and a few other data values. There is no mapped output data (the result is stored in a table which fits entirely in GPU memory and is copied back to CPU memory at the end of the computation).

Opinion Finder: analyzes the sentiments of tweets associated with a given subject (i.e. a set of given keywords) [105]. Words from each tweet that mention the given subject are looked up in three dictionaries of positive, negative, and adverb words. Based on the identified words and their precedence, an overall sentiment score is calculated. The mapped input data consists of new-line separated text records each containing a tweet, a time-stamp, and a few other data values. The dictionaries, as well as the output data (which is a number representing the final sentiment score of tweets toward the subject) fit entirely in GPU memory and thus, are not mapped.

DNA Assembly: merges fragments of a DNA sequence to reconstruct a larger sequence [9]. In the first stage, the application hashes a portion of each fragment and stores it in a hash table to count the number of identical fragments and to remove the noisy ones. In a second stage, the hash table is used to incrementally extend each fragment by finding partial overlaps between different fragments. We only run the first stage on the GPU. The mapped input data consists of fixed-length string records, each containing a DNA fragment and a few other data values. There is no mapped output data (the hash table, which is the output of the first stage, fits entirely in GPU memory and is copied back to CPU memory at the end of the computation).

MasterCard Affinity: finds all merchants that are frequently visited by customers of a target merchant X. The application first extracts a list of customers that visited merchant X and then, with another pass over the purchase

[6]The source code for these applications as well as their input data is available at *http://www.eecg.toronto.edu/~mokhtari/bigkernel*.

Application	Data Size	Record Type	Mapped Data Access Proportion	
			Read	Modified
K-means	6.0GB	Fixed-length	50%	12%
Word Count	4.5GB	Variable-length	100%	0%
Netflix	6.6GB	Fixed-length	30%	0%
Opinion Finder	6.2GB	Fixed-length	73%	0%
DNA Assembly	4.5GB	Fixed-length	36%	0%
MasterCard Affinity	6.4GB	Variable-length	100%	0%
MasterCard Affinity (indexed)	6.4GB	Variable-length (indexed)	25%	0%

Table 4.1: Application Mapped input data. (An application may also allocate and access other non-mapped data structures.)

transactions, identifies the merchants visited by the customers from the list. The mapped input data is a new-line separated collection of purchase transactions each containing a credit card number, a payment terminal ID, and several other values. There is no mapped output data (a table of merchants visited by all customers of merchant X fits entirely in GPU memory and is copied back to CPU memory at the end of the computation).

MasterCard Affinity (indexed): as above, except that an extra index file (not mapped) is provided that contains offsets to the data-fields within the input.

Table 5.1 provides more details on the application data sets and how they are accessed. Applications that do not modify mapped data, write their results to GPU memory and then transfer them to CPU memory after all computations have completed.

To evaluate BigKernel, we implemented five different variations of our applications: (i) a CPU-based serial implementation, (ii) a CPU-based multi-threaded implementation, (iii) a GPU-based implementation that uses a single buffer for data transfers, thus serializing computation and data communication, (iv) a GPU-based implementation that uses double-buffering for data transfers in order to overlap computation with data communication, and (v) BigKernel.

All GPU-based implementations use the same kernel. Each implementation is configured to run with the number of GPU computation threads that results in the best execution time, as determined through experimentation.[7] Moreover, each implementation uses buffer sizes that result in the best execution time, given memory constraints.

The performance results presented here represent the average over ten consecutive runs.

4.4.1 Overall results

Figure 4.10 depicts the speedup of all implementations relative to the CPU-based serial implementation. To help interpret the performance behaviours of the GPU-based implementations, Figure 4.11 shows the measured

[7] GPGPU programmers typically experimentally run their applications with different thread configurations to determine the optimal number of threads and from then on run that configuration.

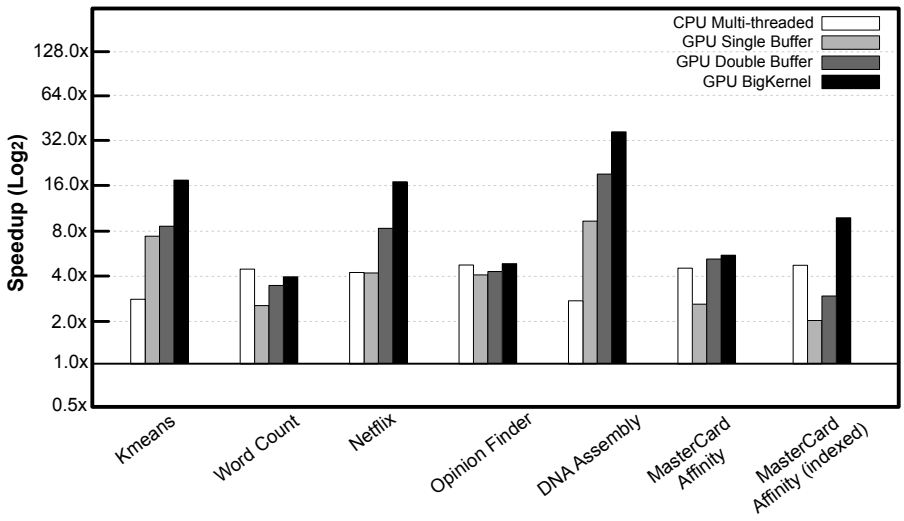

Figure 4.10: Application speedup over serial CPU implementation.

computation / communication ratio of the single-buffer implementation of these applications.

BigKernel outperforms both the single and double-buffering implementations across all applications, on average by 2.6X and 1.7X, respectively. The performance gains can intuitively be attributed to (i) overlapped computation and data communications, (ii) reducing the volume of CPU-GPU data communications and, (iii) enabling coalesced accesses to GPU memory by placing the input data of consecutive threads in interleaved data segments. We show the validity of this intuition further below.

Word Count and Opinion Finder exhibit relatively low speedup and do not appear to benefit from optimized CPU-GPU data communications, primarily because they have a dominant computation stage, which prevents improvements from overlapping computation with communication or from data transfer reductions. Word Count's computation is dominant because it uses a centralized hash table to store word counts, requiring synchronization with attendant overheads. Opinion Finder's computation is dominant because of the fairly heavy lexical analysis it conducts on input tweets.

The speedup of MasterCard Affinity is also limited due to the fact that the entire input data set has to be transferred to GPU memory. This is necessary because the variable-length records force the computation to go over all of the data to identify the individual records (which are delimiter-character separated). The small performance advantage of BigKernel over the double-buffering version is due to the effect of memory coalescing. The indexed version of MasterCard Affinity, however, achieves significant speedup, because it reduces the amount of data transferred, and because the benefits of coalesced memory accesses become more exposed with the more efficient data transfer stage.

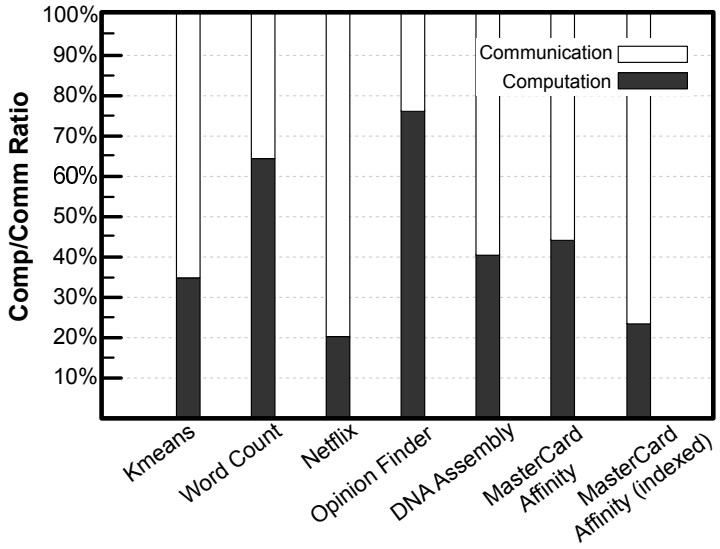

Figure 4.11: Comp/comm ratio in single-buffer implementation.

4.4.2 Performance breakdown

To gain more insight into which features of BigKernel lead to performance improvements, we ran BigKernel with certain features disabled and measured the speedup obtained over the single-buffered implementation. Specifically we compare:

1. **BigKernel overlap only**: this variant transfers all data in its original layout; i.e., no optimizations to reduce the amount of data transferred and no optimization to increase coalesced accesses. Hence, this variant only overlaps communication and computation.

2. **BigKernel transfer volume reduction**: this variant transfers only the data required by the computation but leaves the transferred data in its original layout (with the optimizations for coalesced accesses disabled).

3. **BigKernel**: the complete BigKernel implementation.

Figure 4.12 depicts the incremental speedup obtained from running one variant over the other. The figure thus gives an indication of the contribution of reduced data transfers and data layout optimized for coalesced accesses.[8]

As expected, the amount of data transferred for MasterCard Affinity and Word Count cannot be reduced and therefore they only benefit from communication/computation overlap and memory coalescing. Opinion Finder also does not exhibit performance improvements from reducing the CPU-GPU data transfers due to its dominant computation stage.

[8] It should be noted that the graph would look substantially different if the disabling of features had been done in a different order, because the contributions of each feature overlap in the pipeline.

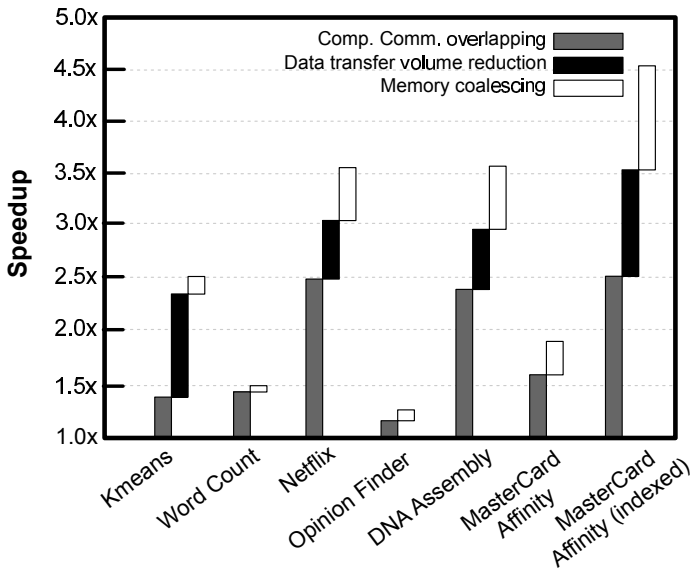

Figure 4.12: The incremental benefit of (i) overlapping computation and communication, (ii) reducing the volume of data transferred due to prefetching, and (iii) laying out the data to increased coalesced accesses.

The effect of the memory coalescing optimization varies from application to application based on a number of factors:

1. the ratio of accesses to mapped data over all data accesses in the kernel – the higher the ratio, the greater the performance benefit;

2. whether the data transfer stage dominates or not – if it does, there is no benefit from optimizing memory accesses;

3. whether or not the original layout already leads to highly coalesced accesses – if so, there is not much room for improvement.

4.4.3 Stage completion time breakdown

For optimal execution, each stage in the BigKernel pipeline would ideally take the same amount of time to complete. This is obviously not the case in practice, and the amount of time each stage requires to complete varies from application to application. For each application, we experimentally measured the time each stage required on average to complete. To measure execution breakdown, we inserted time measurements at the beginning and end of each stage. The data transfer stage, in particular, is measured by having the CPU continuously ping the status of data transfers to stop the timer when the transfer has completed.

Figure 4.13 shows, for each application, the time each stage took to complete on average relative to the stage that took the longest. The address generation stage requires only a small fraction of the total execution time

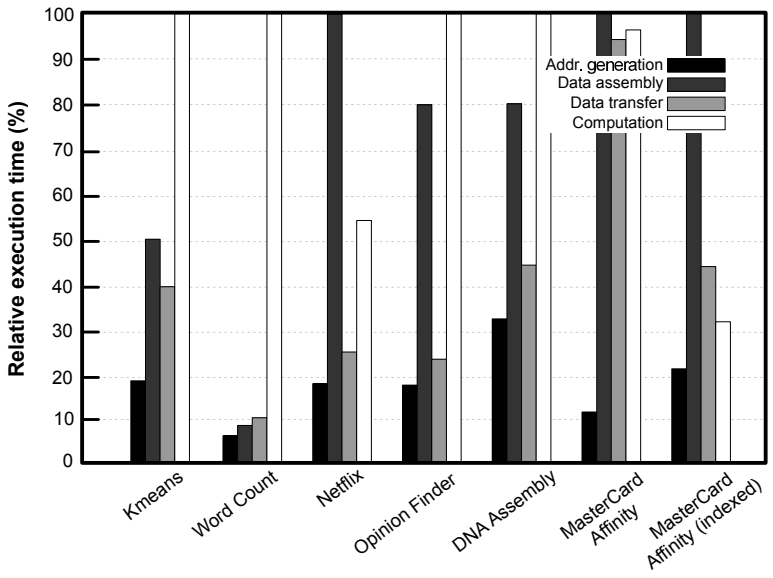

Figure 4.13: Relative completion time of each BigKernel stage.

(usually less than 20%) as it only executes those instructions that contribute to memory address calculations.

The time taken by the data assembly stage varies for the different applications based on (i) the amount of data that has to be assembled and (ii) the data locality of the data items being accessed by CPU memory and hence the cache hit rate.

The time taken for the data transfer solely depends on the size of data to be transferred because the data to be transferred is put in a contiguous pinned-buffer and therefore is efficiently transferred to GPU memory by the GPU DMA engine.

Finally, the computation itself is responsible for a considerable portion of the execution time. Clearly, this is expected for those applications that originally had a dominant computation stage. However, it is interesting to note that BigKernel significantly increased the computation / communication ratios relative to the original single-kernel implementations (Figure 4.11). The fact that the computation stage is the slowest stage for many of these applications indicates that GPU memory may be the bottleneck.

4.4.4 Pattern recognition

Recognition of access patterns during the prefetch address generation stage is a key optimization in BigKernel. This is shown in Table 4.2 that lists the performance improvements when only having to transfer patterns to CPU memory over having to send the actual addresses.

The extent to which performance is improved for each application depends on the number of addresses sent during the address generation stage, which in turn depends on the granularity of the data being accessed. For

Application	Speedup
K-means	31%
Word Count	66%
Netflix	3%
Opinion Finder	6%
DNA Assembly	7%
MasterCard Affinity	57%
MasterCard Affinity Indexed	NA

Table 4.2: Performance improvement due to the use of access patterns.

instance, in K-means, one address is sent for each *double* variable (i.e. 8-byte) while in Word Count, one address has to be sent for each required character (i.e. 1-byte). Having to send a large number of addresses relative to the amount of data to be transferred adds significant overhead to PCIe transfers (for addresses) and on CPU memory (to read addresses during the data assembly stage). Replacing the addresses with a pattern can thus have a significant performance impact. The speedup of MasterCard Affinity Indexed is not available since accesses to input data of this application do not exhibit a pattern.

Chapter 5

S-L1

In this chapter, we show that GPU hardware L1 caches are largely ineffective for Big Data applications and then address this ineffectiveness by proposing and evaluating S-L1, a level one cache implemented entirely in software. In our experimental evaluation, S-L1 achieved speedups of between 0.89 and 6.4 (2.45 avg.) on ten GPU-local applications even though each memory access requires the additional execution of a minimum of 4 instructions (and up to potentially hundreds of instructions). Combining S-L1 with BigKernel leads to speedups of between 1.04 and 1.45 (1.19 avg.) over BigKernel alone, and speedups of between 1.07 and 11.24 (4.32 avg.) over the fastest CPU multicore implementations.

In this chapter, we make the following two specific contributions:

1. we characterize the performance behavior of the GPU memory hierarchy and identify some of its bottlenecks using a number of experiments, and

2. we propose S-L1, a level-1 cache implemented entirely in software and evaluate its performance; novel features of S-L1 include a run-time scheme to automatically determine the parameters to configure the cache.

We start this chapter by presenting motivation in Section 5.1. Section 5.2 presents a detailed description of S-L1's design and implementation. We close by presenting the results of our experimental evaluation of S-L1 in Section 5.3.

5.1 Motivation

In this section we begin by showing how and why GPU hardware L1 caches are ineffective. We then present several GPU architectural trends to provide insight as to where future GPU architectures might be headed. We

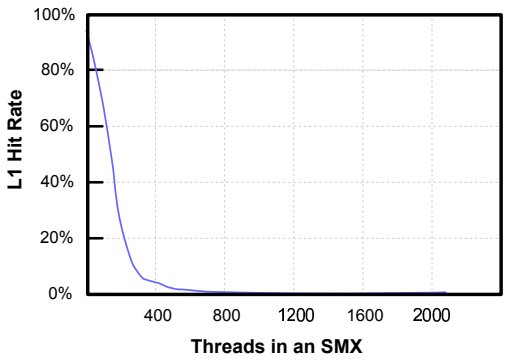

Figure 5.1: L1 hit rate when running `wc` on Titan Black GPU with 192 cores. The target data is partitioned into n chunks with each chunk assigned to a thread for processing. With effective caching, the first access of each thread results in a miss, but the subsequent 127 accesses result in cache hits. Without effective caching, these accesses result in 128 misses.

use these trends to motivate implementing our S-L1 cache. Following that we present several limitations of the GPU memory hierarchy and offer some insights into the nature of those limitations, further motivating our S-L1 design.

5.1.1 GPU Hardware L1 caches

GPU L1 caches are highly inefficient as they are implemented and configured today [43]. For the number of cores typical in modern GPUs, the hardware L1 caches are too small and their cache line sizes are disproportionally large given the small cache size. For example, the L1 on the Nvidia GTX Titan Black we used to run our experiments on can be configured to be at most 48KB per 192 cores, and the cache line size is 128B. At best, this leaves just two cache lines per core. Yet GPGPU best practices expect many threads to run simultaneously per core (supported by 340 4-byte registers per core and fast context switching), each having multiple memory accesses in-flight. Given a large number of executing threads, each issuing multiple memory accesses, cache lines are evicted before there is any reuse, causing a high degree of cache thrashing and an attendantly low L1 hit rate. As an example, Figure 5.1 depicts the L1 hit rate as a function of the number of threads executing when running the Unix word count utility, `wc`.

The hardware L1 has proven to be so ineffective that some recent GPU chip sets, like the Nvidia GTX Titan Black, disables caching of application data by L1 as the GPU's default behavior. Moreover, if historical trends are any indication (see below), we can not expect GPU L1 caches to become significantly more effective any time in the near future.

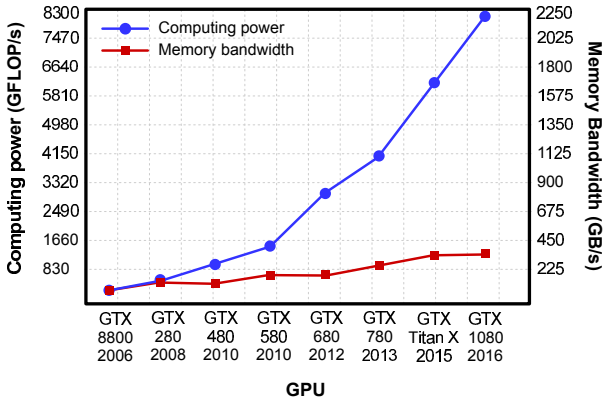

Figure 5.2: Compute power and memory bandwidth over time/GPU generations, normalized to the values of GTX 8800.

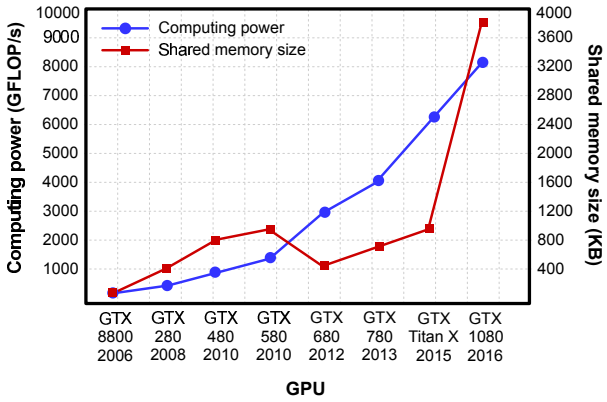

Figure 5.3: Compute power and L1 / shared memory size over time/GPU generation, normalized to the values of GTX 8800 (the rate of growth of GPU register file size has also been similar to that of L1 / shared memory).

5.1.2 Historical Trends in GPU Compute Power and Memory Hierarchy

Figure 5.2 depicts how Nvidia GPU aggregate compute power (in GFLOPS) and memory bandwidth (in GB/s) have evolved over chip generations, from their earliest CUDA-enabled version to the current version. Compute power has been increasing steadily at a steep slope. Memory bandwidth has also been increasing, but not as quickly. As a result, memory bandwidth per FLOP has been decreasing from 250 bytes/KFLOP for the GTX 8800 to 38 bytes/KFLOP for the GTX 1080 (a 6.6X reduction in bytes/KFLOP). This increases the importance of caches, if GPU cores are to be well utilized.

Figure 5.3 depicts how aggregate compute power and total size of fast on-chip memory (L1 cache and shared memory) have evolved over time. The total amount of on-chip memory has roughly kept up with compute power, even though it varies substantially. The GTX 8800 offered 380 bytes/GFLOP in 2006 and GTX 1080 offered 477 bytes/GFLOP in 2016, a 1.25X increase in bytes/GFLOP.

One can clearly see that while GPUs memory bandwidth per FLOP dropped by 6.6X over the last 10 years,

the increase in the size of GPU's on-chip memory per FLOP has only been modest (1.25X increase), making GPU cores increasingly memory starved. As a result, strategies to optimize GPU applications to make more efficient use of the memory hierarchy will likely become more important going forward if these trends continue.

Further, given the fact that future GPU generations may have smaller on-chip memory sizes, as has happened in the past, GPU programmers cannot assume the availability of a specific shared memory size. As a result, the programmer will need to design GPU applications so that they configure the use of shared memory at run-time and possibly restrict the number of threads used by the application. Or she can use run-time libraries, such as the one we are presenting in this dissertation, that automatically adjust program behavior to available hardware resources.

5.1.3 Behavior of GPU memory access performance

GPU vendors do not disclose much information on the micro-architecture of their GPUs. Hence, in order to optimize GPGPU programs so that they can more efficiently exploit hardware resources, it is often necessary to reverse engineer the performance behavior of GPUs through experimentation. In this section, we present the results of some of the experiments we ran to gain more insight into the GPU memory subsystem. All results we present here were obtained on an Nvidia GTX Titan Black. We expect similar results on other GPUs due to the similarities in the configurations of memory subsystems across various GPUs.

Memory access throughput

In our first set of experiments, we used a micro-benchmark that has threads read disjoint subsets of data located in the L2 cache as quickly as they can. The benchmark is parameterized so that the degree of coalescing can be varied. Figure 5.4 shows the maximum L2 memory bandwidth obtained, measured as number of bytes transferred over the network, when servicing 4-way coalesced accesses from the L2 cache as the number of threads running in each thread block is increased up to 1024.

Each curve in the figure represents a different number of thread blocks used, and each block uses the same number of threads. The thread blocks are assigned to SMXs in a round robin manner by the hardware. Focusing on the bottom curve, representing an experiment that has just one thread block running on one SMX, one can see that the memory throughput flattens out after about 512 threads at slightly less than 32 GB/s.[1] We observe similar behavior for DRAM, shown in Figure 5.5, when we adjusted the micro-benchmarks to only access data certain to not be in the L2 cache. In this case, however, the throughput flattens out earlier at about 480 threads, reaching a peak bandwidth of 307 GB/s with 15 blocks.

[1] Our experiments show that varying the degree of coalescing does not completely remove the flattening out behavior. However, the smaller the degree of coalescing, the earlier the curve flattens out.

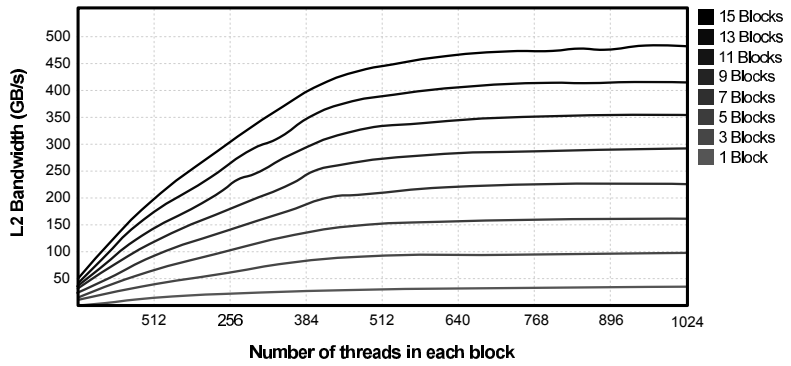

Figure 5.4: L2 memory throughput as a function of number of threads in a thread block. Each curve represents the throughput for a different number of thread blocks (1 to 15) with each thread block running 1,024 threads.

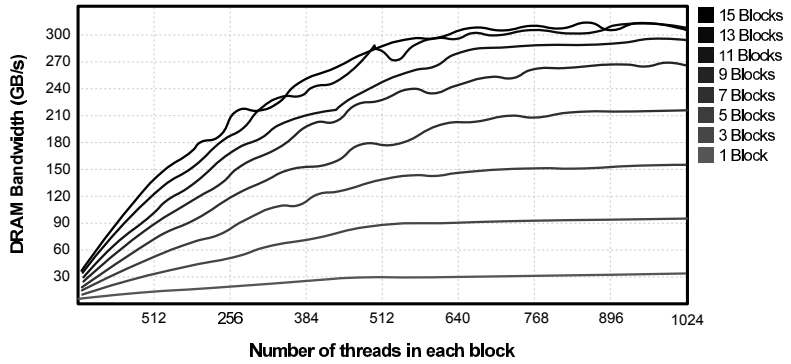

Figure 5.5: DRAM memory throughput as a function of number of threads in a thread block. Each curve represents the throughput for a different number of thread blocks (1 to 15) with each thread block running 1,024 threads.

It is difficult to assess what causes the stagnation in L2 and DRAM throughput. However the near-linear scalability with the number of thread blocks indicates that the bottleneck is in the interconnect or in the SMX itself (e.g., coalescing units) rather than in the L2 or DRAM subsystems. This is shown in Figure 5.6 where we show the throughput as a function of the number of thread blocks with each thread block running 1,024 threads. Each point along the bandwidth curves of Figure 5.6 is equal to the end point (at 1,024 threads) of the corresponding curve of Figures 5.4 or 5.5. L2 throughput increases almost linearly, reaching close to 480 GB/s with 15 blocks. DRAM throughput increases almost linearly until it reaches approximately 300 GB/s, at which point it flattens out, indicating that its bandwidth capacity has been reached at that point.

All of the results presented above measured the amount of data transferred to the SMXs by the hardware. However, depending on the application, much of this data may not actually be used by the application. For example, for non-coalesced accesses, each 4-byte integer access will result in 32 byte transfers, of which only 4 are actually used.

The throughput limitations of L2 and DRAM, as well as the fact that only some of the data transferred is used by the application, indicates that memory bandwidth actually achieved in practice will be far lower than the

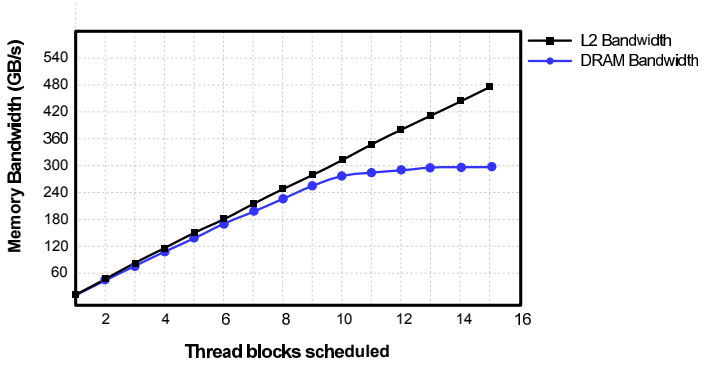

Figure 5.6: L2 and DRAM memory bandwidth as a function of number of thread blocks where each thread block is running 1,024 threads and the memory accesses are 4-way coalesced.

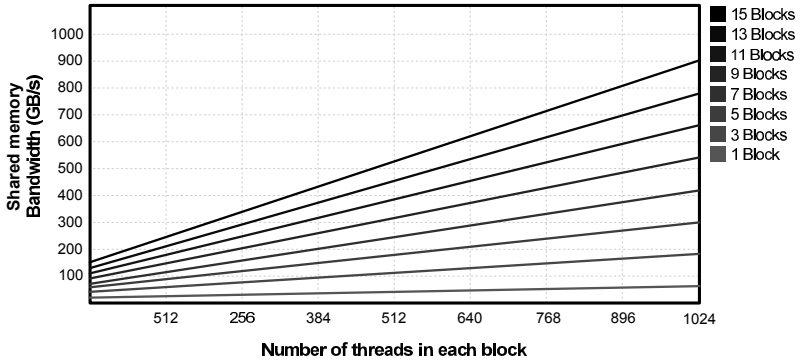

Figure 5.7: Shared memory throughput as a function of number of threads in a thread block. Each curve represents the throughput for a different number of thread blocks (1 to 15) with each thread block running 1,024 threads.

theoretical bandwidth presented in chapter 2.

In contrast to L2 and DRAM throughput, shared memory throughput within an SMX, shown in Figure 5.7, does not flatten out and reaches 60GB/s (for an aggregate throughput of close to 900GB/s with 15 SMXs).

5.2 S-L1 Design and Implementation

In this section, we present the design and implementation of S-L1.

5.2.1 Overview

S-L1 is a level 1 cache implemented entirely in software. While a software implementation of the L1 cache adds considerable base overhead that has to be amortized, we still implemented S-L1 in software because (i) software allows easier customization and (ii) a software cache can be used only when beneficial, on a per application basis.

S-L1 uses the space available in each SMX's shared memory, which has the same access latency as the hardware L1. We also considered using SMX's texture cache, but it is a hardware-managed, read-only cache and

thus, does not suit the needs of S-L1.[2] S-L1 does not require the developer to modify the GPU application code; instead the compiler inserts the code required to implement S-L1.

The design of S-L1 is guided by three key principles that deal with the small size of the shared memory:

1. **Private cache segments**: S-L1 is partitioned into thread-private cache lines instead of having all threads share the cache space, thus eliminating mapping collisions.

2. **Smaller cache lines**: the cache-line size is, at 16B, smaller than what is typical for GPUs, allowing a larger number of cache-lines to be shared per thread; and

3. **Selective caching**: the data of only a select number of data structures are cached.

The objective of our design is to significantly decrease average memory access times and minimize S-L1 cache thrashing. The specific parameters of the S-L1 cache are determined at runtime during an initial brief monitoring phase, which also identifies the potential cache hit rate of each data structure. After the monitoring phase, the computation is executed using the S-L1 cache for the n data structures that have the highest cache hit rates, where n is selected based on the amount of cache space available to each thread.

The decision to use thread-private cache segments is based on the fact that inter-thread data sharing is rare in the Big Data applications we are targeting. Therefore, the threads mostly process data independently in disjoint locations of memory. Allowing all threads to share the entire cache space would likely result in unnecessary collisions.

We use relatively small cache lines. The optimal cache line size depends to a large extent on the applications' memory access patterns. Larger cache lines perform better for applications exhibiting high spacial locality, but they perform poorer for applications with low spacial locality due to (i) the extra overhead of loading the cache lines requiring multiple memory transactions and (ii) the increased cache thrashing that occurs because fewer cache lines are available. We decided on using 16-byte cache lines after experimenting with different cache line sizes — see Section 5.3.5. This size works well because 16B is the widest load/store size available on modern GPUs, allowing the load/store of an entire line with one memory access.

We only cache some of the application's data structures.[3] The number of cache lines allocated to each thread (CLN) determines how many data structures we cache. CLN is calculated at runtime as

$$CLN = [(shMemSizePerSM/numThreadsPerSM)/cacheLineSize]$$

[2] In a separate set of experiments, we also evaluated the effectiveness of the texture cache for Big Data applications. The results show that, like the hardware L1 cache, the texture cache hit rate drops significantly when the number of online threads increase.

[3] In this context, each argument to the GPU kernel that points to data is referred to as a data structure. For example, matrix multiply might have three arguments a, b and c referring to three matrices; each is considered a data structure.

where $shMemSizePerSM$ is the amount of shared memory available per SMX, $numThreadsPerSM$ is the total number of threads allocated on each SMX, and $cacheLineSize$ is the size of the cache line; i.e., 16 bytes in our current design.

The values of the variables in the above formula are non-trivial to obtain. For example, the amount of shared memory available for the S-L1 cache depends on how much shared memory has previously been allocated by the application. An application can allocate a fixed (i.e., specifying the exact size in the code) or variable (i.e., dynamically setting the size at runtime) amount of shared memory. Hence, a mixed compile-time/runtime approach is required to identify how much shared memory remains available for S-L1. $NumThreadsPerSM$ is calculated at runtime, in part by using the configuration the programmer specified at kernel invocation and in part by calculating the maximum number of threads that can be allocated on each SMX which in turn depends on the resource usage of GPU threads (e.g. register usage) and available resources of SMX, which is extracted at compile-time and runtime, respectively.

Once the number of cache lines per thread – CLN – has been determined, up to that many data structures are marked as *S-L1 cacheable* and a separate cache line is assigned to each. In principle, multiple cache lines could be assigned to a data structure, but we found this does not benefit the Big Data applications we are targeting. Generally, cache hits occur due to locality, which either is due to data reuse (where the same data is accessed multiple time within a short period of time) or is due to accessing data (for the first time) that happens to reside in the same cache line as a recently accessed data item. In the applications we target, the latter is typically the case. Hence having multiple cache lines per data structure offers no benefit.

Data structures that are not marked as cacheable will not be cached and are accessed directly from memory. If the available size of shared memory per thread is less than $cacheLineSize$ (i.e., too much shared memory has already been allocated by the application) then S-L1 is effectively disable by assigning no cache lines to threads (i.e., $CLN = 0$).

To determine which data structures to cache, we evaluate the benefit of caching the data of each data structure using a short monitoring phase at runtime. In the monitoring phase, the core computation of the application is executed for a short period of time, during which a software cache for each data structure and thread is simulated to count the number of cache hits of each data structure during a fixed period of time. When the monitoring phase terminates, the CLN data structures with the highest cache hit counts will be marked as to be cached. The code required for the monitoring phase is injected into existing applications using straightforward compiler transformations.

5.2.2 Code Transformations

The compiler transforms the main loop(s) of the GPU kernel into two loop slices. The first loop slice is used for the monitoring phase, where the computation is executed for a short period of time using the cache simulator. After the first loop slice terminates, the data structures are ranked based on their corresponding cache hit counts, and the top CLN data structures are selected to be cached in S-L1. The second loop slice then executes the remainder of the computation using S-L1 for the top CLN data structures.

As an example, the following code:

```
//Some initialization
for(int i = start; i < end; i ++) {
     char a = charInput[i];
     int b = intInput[i];

     int e = doComputation(a, b);
     intOutput[i] = e;
}
//Some final computation
```

is transformed into:

```
//Some initialization
cacheConfig_t cacheConfig;
int i = start;

//slice 1: monitoring phase
for(; (i < end) && (counter<THRESHOLD); i ++, counter ++) {
     char a = charInput[i];
     simulateCache(&charInput[i], 0, &cacheConfig);
     int b = intInput[i];
     simulateCache(&intInput[i], 1, &cacheConfig);

     int e = doComputation(a, b);
     intOutput[i] = e;
     simulateCache(&intOutput[i], 2, &cacheConfig);
}
calculateWhatToCache(&cacheConfig, availNumCacheLines);
//slice 2: rest of the computation
for(; i < end; i ++)
{
     char a = *((char*) accessThroughCache(&charInput[i], 0,
                              &cacheConfig));
     int b = *((int*) accessThroughCache(&intInput[i], 1,
                          &cacheConfig));

     int e = doComputation(a, b);

     *((int*) accessThroughCache(&intOutput[i], 2,
                &cacheConfig)) = e;
}
flush(&cacheConfig);
//Some final computation
```

Monitoring phase

In the monitoring loop, a call to `simulateCache()` is inserted after each memory access. This function takes as argument the address of the memory being accessed, a *data structure identifier*, and a reference to the *cacheConfig* object, which stores all information collected during the monitoring phase. The *data structure identifier* is the

identifier of the data structure accessed in the corresponding memory access and is assigned to each data structure statically at compile time.

The pseudo code of `simulateCache()` is listed in Figure 5.8. This function keeps track of which data is currently being cached in the cache line, assuming a single cache line is allocated for each thread and data structure, and it counts the number of cache hits that occur. To do this, `cacheConfig` contains, for each data structure and thread, an address variable identifying the memory address of the data that would currently be in the cache, and a counter that is incremented whenever a cache hit occurs. On a cache miss, the address variable is updated with the memory address of the data that would be loaded into the cache line.

```
simulateCache(addr, accessId, cacheConfig) {
     addr /= CACHELINESIZE;

     if(addr == cacheConfig.cacheLine[accessId].addr)
          cacheConfig.cacheLine[accessId].hit ++;
     else { //a miss
          cacheConfig.cacheLine[accessId].addr = addr;
     }
}
```

Figure 5.8: The pseudo code of `simulateCache`.

The monitoring phase is run until sufficiently many memory accesses have been simulated so that the behavior of the cache can be reliably inferred. To do this, we simply count the number of times `simulateCache()` is called by each thread; once it reaches a predefined threshold for each thread, the monitoring phase is exited. This pre-defined threshold is set to 300 in our current implementation.

This method of statically setting the duration of monitoring phase works well for regular GPU applications such as the ones we are targeting, but more sophisticated methods may be required for more complex, irregular GPU applications. Moreover, while we only run the monitoring phase once, it may be beneficial to enter into a monitoring phase multiple times during a long running kernel to adapt to potential changes in the caching behavior.

Determining what to cache

In the general case, we mark the CLN data structures with the highest cache hit counts to be cached in S-L1. However, there are two exceptions. First, we distinguish between read-only and read-write data structures. Read-write data structures incur more overhead, since dirty bits need to be maintained and dirty lines need to be written back to memory. Hence, we give higher priority to read-only data structures when selecting which structures to cache. Currently, we select a read-write data structure over a read-only data structure only if its cache count rate is twice that of the read-only data structure, because accesses to read-write cache lines involve the execution of twice as many instructions on average.

Secondly, in our current implementation, we only cache data structures if it has a cache hit rate above 50%. A hit rate of more than 50% means that, on average, the cache lines are reused at least once after loading the data due to a miss. We do this because otherwise the overhead of the software implementation will not be amortized by faster memory accesses.

Computation phase

In the second loop slice, the compiler replaces all memory accesses with calls to `accessThroughCache()`. This function returns an address, which will either be the address of the data in the cache or the address of the data in memory depending on whether the accessed data structure is cached or not. A simplified version of `accessThroughCache()` is listed in Figure 5.9.

```
void* accessThroughCache(void* addr, int accessId,
                  cacheConfig_t* cacheConfig)
{
     if(cacheConfig.isCached[accessId] == NOT_CACHED) {
         return addr;
     }
     else {
         //If already cached, then simply return the
         //address within the cache line
         if(alreadyCached(addr, cacheConfig.cacheLine[accessId])) {
              return &(cacheConfig.cachelines[accessId].
                              data[addr % 16]);
         }
         //requested data is not in the cache, so,
         //before caching it we need to evict current data.
         else {
            //If not dirty, simply overwrite. If dirty,
            //first dump the dirty data to memory

            if(cacheConfig.cachelines[accessId].dirty) {
               dumpToMemory(cacheConfig.cachelines[accessId]);
            }
            loadNewData(addr, cacheConfig.cachelines[accessId]);
            return &(cacheConfig.cachelines[accessId].
                        data[addr % 16]);
        }
    }
}
```

Figure 5.9: The pseudo code of `accessThroughCache`.

S-L1 cache misses on cacheable data cause the eviction of an existing cache line to make space for the new target cache line. If the existing cache line has been modified, then it is first written back to memory. We keep a bitmap (in registers) to identify which portions of the cache line are modified, so that only those need to be written back; this also guarantees that if two different threads cache the same line (in different S-L1 lines) and modify different potions of the cache line, they will not overwrite each other's data.

A call to `flush()` is inserted after the second loop slice to flush the modified cache lines to memory and invalidate all cache lines before the application terminates.

If pointers to data structures, provided as arguments to the GPU kernel, are aliased, then the caching layer might cache the same data from the same data structure in multiple separate cache lines. This not only wastes

the cache space, it might also break the correctness of an application if those cache lines are dirty and have to be written back (as they will override each other). To prevent this from happening, the caching layer has to assume that the pointers may be aliased, and perform data lookups in all cache lines assigned to the same thread for each memory access. This is an overhead which can be avoided if the pointers are explicitly marked as not aliased. Programmers can indicate this by including the `restrict` keyword with each kernel argument.

5.2.3 S-L1 overheads

Our implementation of S-L1 introduces overheads for the monitoring phase, when determining what data structures to cache, and for each memory access. Moreover, it uses up registers which may be in short supply. We briefly describe these overheads below.

Monitoring phase: the monitoring phase has a performance overhead because it adds instructions to record the number of cache hits for each data structure. However, our experiments show that this performance overhead is relatively low in practice — an average of less than 1% was observed in the 10 applications we experimented with (see Section 5.3.4). The overhead is low because the monitoring phase only runs for a short period of time and because the code of `simulateCache()` is straightforward and typically does not incur additional memory accesses since all variables used in `simulateCache()` are located in registers.

calculateWhatToCache(): this function reads cache hit counters and picks the data structures with the highest cache hit number. This overhead is negligible since the function's logic is very simple and straightforward, and typically, the applications access only a few data structures.

accessThroughCache(): most of the overhead of the caching layer occurs in this function. For accesses to data structures that are not cached, the performance overhead entails the execution of four extra machine instructions. However, accesses to cached data structures incur significantly more overhead in some cases; e.g., when evicting a cache line. Our experiments indicate that the caching layer increases the number of instructions issued by 25% on average over the course of the entire application (see Section 5.3.4). This overhead can indeed negatively impact the overall performance of an application if it is not amortized by the lower access times offered by S-L1, and the overhead is exacerbated if the application's throughput is already limited by instruction-issue bandwidth.

Register usage

The monitoring phase requires two registers per data structure/simulated cache line: one for the mapped address of the cache line in memory and another to keep the cache hit counters. These registers are only required during the monitoring phase and will be reused after the phase terminates.

`accessThroughCache()` requires three additional registers per data structure and thread: one for the

memory address of the data currently being cached, one for the write bitmap (which also serves as the dirty bit), and one for the data structure identifier. (If the data structure is not cached, the value of the last register will be -1). As an optimization, we do not allocate bitmap registers for read-only data structures. Additionally, since data structures that are not cached do not access the bitmap and address registers, the compiler might spill them to memory, without ever accessing them, thus reducing the register usage of uncached data structures to 1.

The recent GPU architectures (e.g. *Kepler*) have 65,535 registers per SMX and can support at most 2,048 threads, in which case the S-L1 caching layer would, in the worst case, use up to 6% and 9% of the total number of available registers for cached read-only and read-write data structures, respectively.

5.2.4 Coherence considerations

Since each thread has its own private cache lines, cached data will not be coherent across cache lines of different threads. Thus, if two threads write to the same data item cached separately, the correctness of the program might be compromised. Fortunately, the loose memory consistency model offered by GPUs makes it easy to maintain the same level of consistency for S-L1 accesses. We follow two simple rules to maintain the correctness of the program: (a) we flush the threads' cache lines on *memory fence instructions* and (b) we do not cache the data of data structures that are accessed by atomic instructions.

Executing a *memory fence instruction* enforces all memory writes that were performed before the instruction to be visible to all other GPU threads before the execution of the next instruction. GPGPU programmers are required to explicitly use these instructions if application logic relies on a specific ordering of memory reads/writes. We implement this by inserting a call to `flush()` immediately before each memory fence instruction, which flushes the contents of the modified cache lines to memory and invalidates the cache lines.

By executing an *atomic instruction*, a thread can read, modify, and write back a data in GPU memory atomically. We extract the data structures that might be accessed by atomic instructions at compile time and mark them as not cacheable by S-L1.

5.3 Experimental Evaluation

5.3.1 Experimental Setup

Unless noted otherwise, all GPU kernels used to evaluate S-L1 were executed on an Nvidia GeForce GTX Titan Black GPU connected to 6GB of GPU memory with a total of 2,880 computing cores running at 980MHz. The GTX Titan Black is from the Kepler family and has 15 SMXs, each with 192 computing cores, and 64KB of on-chip memory (of which 48KB is assigned to shared memory).

Application	Description	# data structures
Upper	Converts all text in an input document from lowercase to uppercase.	2
WC	Counts the number of words and lines in an input document.	1
DNA Assembly	merges fragments of a DNA sequence to reconstruct a larger sequence [9].	3
Opinion Finder	analyzes the sentiments of tweets associated with a given subject (i.e. a set of given keywords) [105]	4
Inverted Index	Builds reverse index from a series of HTML files.	3
Page View Count	Counts the number of hits of each URL in a web log.	3
MasterCard Affinity	finds all merchants that are frequently visited by customers of a target merchant X [73]	3
Matrix Multiply	Calculates the multiplication of two input matrices. This is a naive version and does not use shared memory.	3
Grep	Finds the string matching a given pattern and outputs the line containing that string.	2 (1 in shared memory)
Kmeans	Partitions n particles into k clusters so that particles are assigned to the cluster with the nearest mean.	2 (1 in shared memory)

Table 5.1: Ten Big Data applications used in our experimental performance evaluation, their description, and the number of data structures they use in their main loop. S-L1 determines the number of data structures to cache at runtime, which could vary from run to run depending on the available size of shared memory per thread

All GPU-based applications were implemented in CUDA, using CUDA toolkit and GPU driver release 7.0.28 installed on a 64-bit Ubuntu 14.04 Linux with kernel 3.16.0-33. All applications are compiled with the corresponding version of the *nvcc* compiler using optimization level three.

For each experiment, we ran the target application using different thread configurations, and only considered the configuration with the best execution time for reporting and comparison purposes. Specifically, we tested each application using 512 different thread configurations, starting with 15 blocks of 128 threads (for a total of 1920 threads) and increased the number of threads in 128 increments, up to 480 blocks of 1024 threads (for a total of 480K threads).

We applied S-L1 to the ten data processing applications listed in Table 5.1. There is no standard benchmark suite for GPU data processing applications, so we selected 6 representative applications, 2 simple scientific applications (`MatrixMultiply` and `Kmeans`) to see how well S-L1 works on them, and 2 extreme applications to stress test S-L1, namely `wc`, which has minimal computation (only counter increments) for each character access, and `upper`, which is similar to `wc` but may modify the characters. For each experiment, the data accessed by the applications was already located in GPU memory.

5.3.2 S-L1 performance evaluation for GPU-local applications

Figure 5.10 shows the performance of our 10 benchmark applications when run with S-L1 and hardware L1 relative to the performance of the same applications run with no caching at level 1 (L2 cache is enabled in all cases). On average, the applications using S-L1 run 1.90X and 2.10X faster relative to when they use hardware L1 and no level 1 caching at all, respectively.

Using S-L1, some applications (e.g., `upper` and `wc`) run multiple times faster, while others (e.g., `grep` and `Kmeans`) experience slight slowdowns. The benefits obtained from S-L1 depends on a number of factors. First,

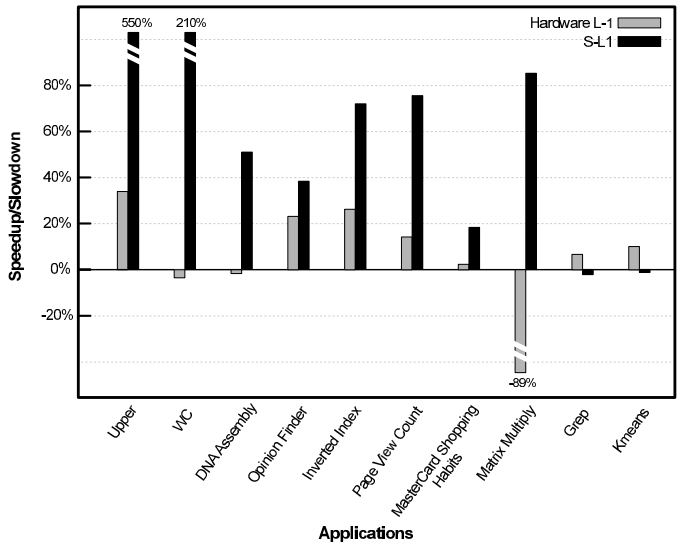

Figure 5.10: Speedup when using S-L1 relative to no L1 caching.

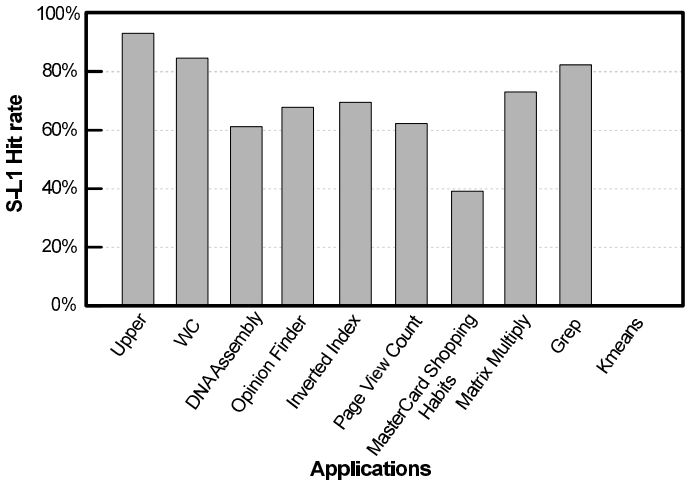

Figure 5.11: S-L1 hit rate.

the attained cache hit rate obviously has a large effect. Figure 5.11 depicts the S-L1 hit rate for all benchmarks.

Overall, the hit rate is quite high, in part because most of the applications have high spatial locality (which is to be expected for data processing applications). As an extreme example, consider `wc`, where each thread accesses a sequence of adjacent characters, so each S-L1 miss is typically followed by 15 hits, given a 16 byte cache line. Kmeans is an exception: because the application allocates much of the shared memory for its own purposes, there is insufficient space for S-L1 cache lines, and hence the effective S-L1 hit count is zero for this application.[4]

A second factor is the memory intensity of the applications; i.e., the ratio of memory access instructions to the total number of instructions executed. Some applications (e.g., `upper` and `wc`) are memory bound and hence benefit from S-L1. At the other extreme, `grep` performs worse despite having a high cache hit rate, because it

[4]Because Kmeans allocates space in shared memory dynamically at run time, the compiler cannot know that there is not enough space for S-L1 — otherwise it potentially could have avoided adding the code required for S-L1.

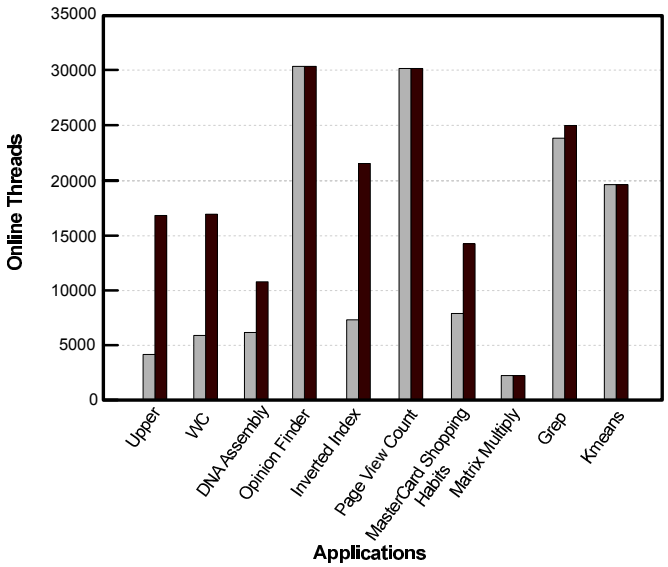

Figure 5.12: The optimal number of online threads (that leads to the best execution times) with and without S-L1 (hardware L1 is enabled).

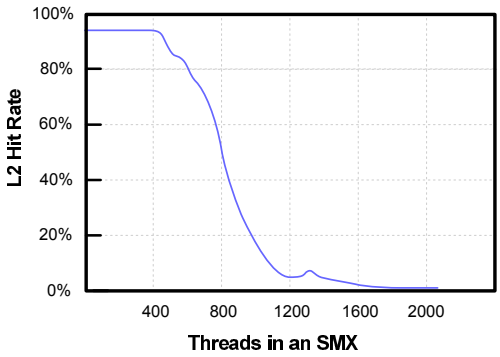

Figure 5.13: L2 hit rate for `wc`.

is compute intensive with its recursive algorithm and because it has a large number of branches with significant thread divergence. The benefits of the caching layer is negated by the extra instructions that need to be executed because of the software implementation of S-L1.

A third factor is the degree to which S-L1 enables extra thread parallelism, thus improving the utilization of the GPU cores. Figure 5.12 shows the number of *online* threads[5] that result in the best performance for each application with and without S-L1. With S-L1, applications can run with more threads without having to worry about thrashing in S-L1. Without S-L1, applications typically need to limit the degree of parallelism to prevent L2 cache thrashing. For example, Figure 5.13 shows the L2 cache hit rate of `wc` as a function of the number of threads per SMX, where the cache hit rate drops to around 10% at 1,024 threads.

Compared to S-L1, the benefit of hardware L1 cache is rather limited (shown in Figure 5.10). Overall, the performance gains with the L1 are limited to under 35% and in some cases result in slowdowns. In particular, `wc`,

[5]I.e., threads that run at the same time on all multiprocessors, the maximum of which can be 30K threads on our GPU.

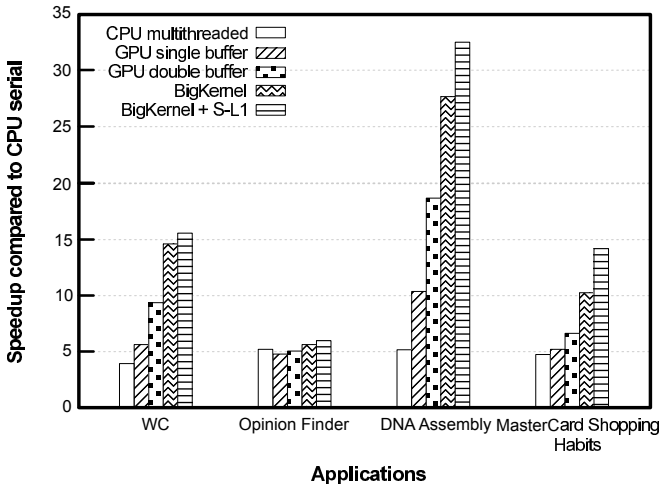

Figure 5.14: Speedup of our five scenarios relative to the CPU single core version for four GPU applications processing large data sets located in CPU memory.

`PageViewCount`, and `Matrix Multiply` all exhibit slowdown when L1 is enabled. We attribute this to the extra DRAM transactions that result from L1 cache line thrashing. Note that, when L1 is enabled, each cache miss results in four DRAM transactions (four 32-byte transactions to fill a 128-byte L1 cache line), three transactions more than what actually is required to fulfill the requesting memory access instruction. This phenomenon was originally observed by by Jia et al. [43].

5.3.3 Evaluation of S-L1 for data residing in CPU memory

For Big Data-style applications, the data will not fit in GPU memory because of the limited memory size. Hence, in this subsection, we consider the performance of four applications with data sets large enough to not fit in GPU memory. We ran these applications under five different scenarios:

1. CPU multi-threaded when run on a 3.7GHz Intel Core i7-4820K with 24GB of dual-channel memory clocked at 1.8 GHz;

2. GPU using a single buffer to transfer data between CPU and GPU;

3. GPU using state-of-the-art double buffering to transfer data between CPU and GPU;

4. GPU using BigKernel; and

5. GPU using BigKernel combined with S-L1.

We selected to combine S-L1 with BigKernel in particular, because BigKernel is currently the best performing system for data intensive GPU Big Data applications.

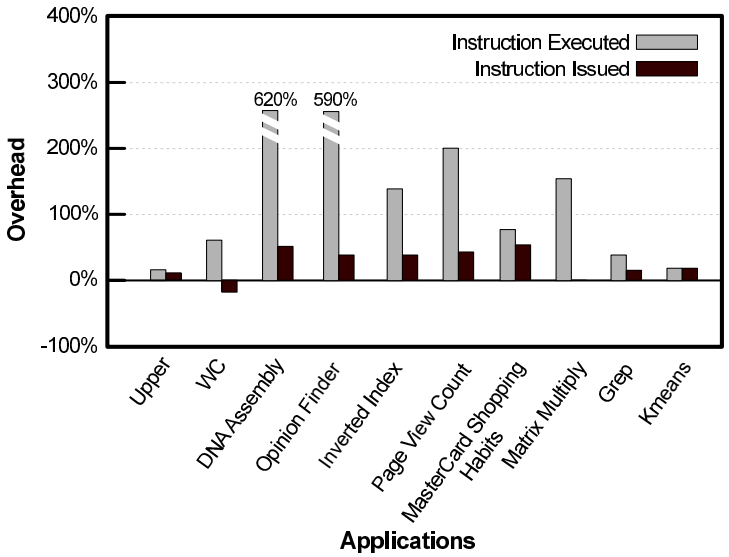

Figure 5.15: S-L1 overhead, in terms of extra instruction executed and issued (in %) when using S-L1 relative to when not using it.

Figure 5.14 shows the results. For all four applications, using BigKernel combined with S-L1 performs the best, and for all but one application the performance is an order of magnitude better than the multi-threaded CPU version. Compared to BigKernel alone, BigKernel combined with S-L1 is 1.19X faster. The primary reason preventing BigKernel from performing better when combined with S-L1, we observed, is that BigKernel uses many registers and therefore, its parallelism is limited[6].

5.3.4 S-L1 overheads

S-L1 has significant overheads because it is implemented in software and extra instructions need to be executed for each memory access: a minimum of 4 and potentially well over 100 extra instructions per memory access. Figure 5.15 depicts the increase in the number of instructions, both executed and issued, under S-L1. Executed instructions are the total number of instructions completed, while issued instructions also count the times an instruction is "replayed" because it encountered a long latency event such as a memory load.

The increase in the number of executed instructions is significant: 220% on average. The reason is obvious: each memory access instruction is transformed to additionally call a function that needs to be executed. On the other hand, the increase in the number of issued instructions is more reasonable: 25% on average. (For `wc` and `MatrixMultiply` the number of issued instructions actually decreases.) The reason issued instructions increase less than executed instructions is that S-L1 provides for improved memory performance, which reduces the number of required instruction replays.

[6]When a kernel uses high number of registers, an SMX will schedule less number of *online threads* to be able to provide them with the required number of registers.

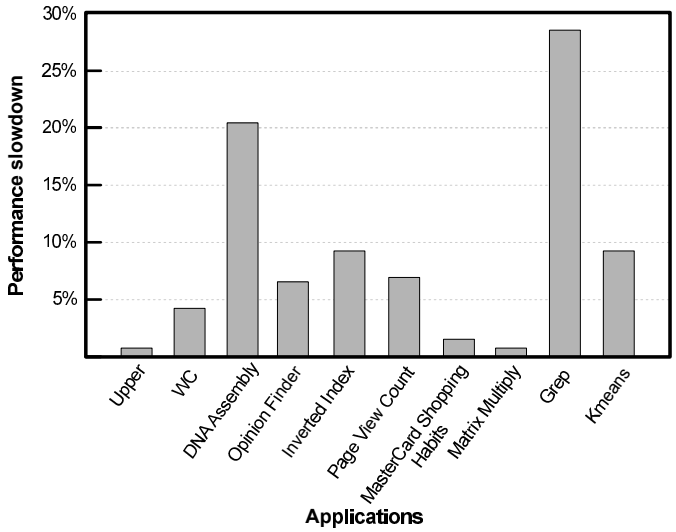

Figure 5.16: Overhead of S-L1 when S-L1 is enabled but not used to cache any data.

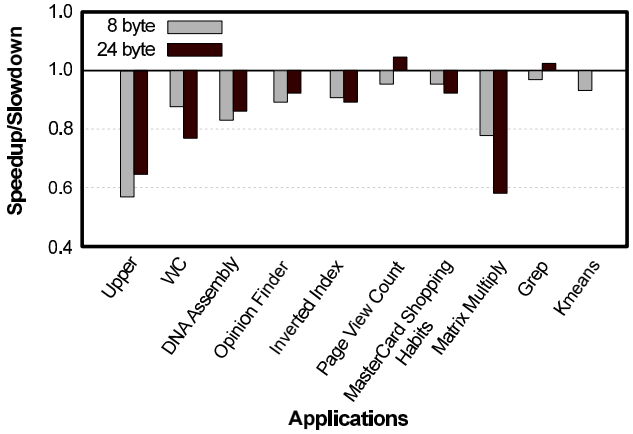

Figure 5.17: Slowdown/speedup when using 8B and 24B cache lines over using 16B cache lines.

To evaluate the amount of overhead S-L1 introduces for data structures that are not cached, we ran our benchmarks with S-L1, but with all data structures marked as non-cacheable. Figure 5.16 shows the overhead incurred in this case: 8% on average. Based on our experiments, the monitoring phase accounts for less than 1% of this overhead. The rest of the overhead is attributed to executing the memory access function that is called for each memory access. As suggested in Section 5.3.2, one potential way to avoid this overhead is have the compiler not transform memory accesses to data structures that are found not worthy of caching – e.g. a data structure that is statically known not exhibit any caching benefit.

5.3.5 Effect of S-L1 cache line size

Figure 5.17 compares the overall performance of applications when S-L1 is configured to use different cache line sizes. Specifically, we show the performance improvement/loss for 8-byte and 24-byte cache lines over 16-byte

cache lines.[7]

In most cases, 16-byte cache lines seems to be best choice. As we described in Section 5.2.1, we believe this is mainly because 16-bytes is the widest available load/store size on GPU ISA and hence, the entire cache line can be read/written with one memory access.

Decreasing the cache line size to 8-bytes impacts performance negatively in every case, since the cache then typically needs to execute the inserted memory access function twice as often for a fixed amount of streaming data to be processed by the application. Note that our benchmarks primarily consist of Big Data applications that have high spatial locality and that consume most of the data in a cache lines.

Increasing the cache line size to 24-bytes results in worse performance in all but two cases, mainly because 24 bytes do not provide much additional benefit over 16 bytes, yet require two memory accesses to fill a cache line instead of one. For example to process 48 characters accessed sequentially, a 16B cache line results in 3 misses and thus 3 L2/DRAM accesses, whereas a 24B cache line results in 2 misses and thus 4 L2/DRAM accesses.

[7] We did not choose 32 as a potential cache line size since the shared memory could not accommodate the cache lines of that size if the SMXs are fully occupied with 2048 threads.

Chapter 6

A Hash Table for GPU-based Big Data Applications

In this chapter, we present the design and implementation of a hash table that is intended to be used as a key-value store for GPU-based Big Data applications. The key characteristic of this hash table is that it retains reasonable efficiency even when it grows beyond the size of GPU memory. To be able to offer this, we use a model of computation called *SePo* that we developed to reduce the amount of data transferred between CPU and GPU memories. Even though we initially developed this model for our hash table design, we believe it can be used in other contexts as well (although it may not necessarily be beneficial for all applications).

Section 6.1 states the problem we are trying to tackle in more detail. An overview of the solution we propose for this problem is presented in Section 6.2. Sections 6.3 and 6.4 go deep into the design and implementation of our hash table. We close by presenting the results of our experimental evaluations in Section 6.6.

6.1 Problem Statement

The Big Data applications we are targeting often store their results in the form of key-value (KV) pairs. In part, this is because the Big Data ecosystem started off with MapReduce, which stores data in the key-value format. Today, the high level of storage and interoperability support for the key-value format has made it a *de facto* standard for Big Data applications. It is, therefore, highly desirable to have high-performant key-value stores on platforms that run Big Data applications, such as GPUs.

The hash table is an obvious data structure to consider for storing KV pairs efficiently. It not only allows for fast data store and lookup, but it also offers on-the-fly grouping of pairs based on their keys (where the values of

KV pairs with the same key are "grouped" or "combined"). On-the-fly grouping of pairs eliminates two overheads that might otherwise occur when grouping is postponed to a later, separate stage of execution: the overhead of storing multiple copies of the same key and the overhead of a separate grouping stage, that typically requires the data to be sorted first. In fact, despite the irregularity of memory accesses to a hash table, which is a performance hazard for GPUs, several previous studies have identified clear performance advantages when comparing hash tables to other potential data structures for key-value storage, as we described in Section 3 [35, 37, 72].

Using hash tables on GPUs has one major challenge, however. Unlike simpler data structures like arrays, hash tables cannot be broken into smaller segments that can be operated on independently, because each key may index into any location of the hash table. As a result, it is not trivial to algorithmically support a hash table that is larger than the available GPU memory.

There are two obvious system-level solutions to support hash tables that are larger than GPU memory, but both incur high data transfer overhead. The first allocates the entire hash table as a pinned region in CPU memory and has the GPU threads directly access the structure in CPU memory (remotely, over the PCIe bus). The second solution uses a hardware demand paging mechanism for GPUs which uses CPU memory as the secondary storage: if the part of the hash table being accessed is not in GPU memory, then the corresponding page(s) is paged in before the access can complete. Both solutions incur a high number of data transfers over the PCIe bus, resulting in a poor performance, as will be shown in Section 6.6.4. This overhead prevents GPU Big Data applications from using hash tables if they do not fit entirely in GPU memory. To make the matters worse, due to the dynamic memory space requirement of hash tables, there is typically no way to predict – before runtime – whether a given dataset size can be processed successfully within the available GPU memory or not. This makes GPUs an *unreliable* hardware base for real-world Big Data applications, unless one of the designs described below is used.

6.2 Solution Overview

We now introduce the SePo model of computation as a solution to allow a hash table to retain its efficiency even if it grows beyond the size of GPU memory. We first present SePo as a general model of computation and then describe how it is used specifically to support larger-than-memory hash tables for GPUs.

6.2.1 SePo Overview

In the SePo[1] model of computation, a service requestee may postpone servicing a request by declining the request and asking the requestor to re-issue the request at a later time. The requestee might, for example, postpone a request because the required resources are not available or because it is inefficient to provide the service at the

[1] SePo is short for *Selective Postponement*, which implies that the requestee can temporarily postpone providing its service.

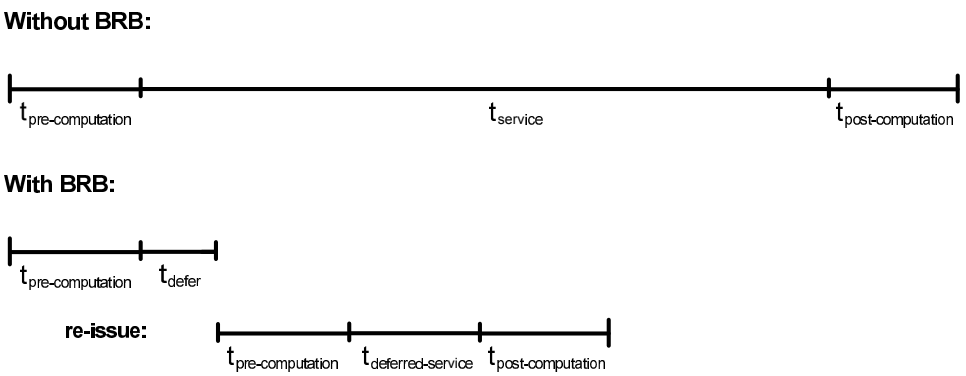

Figure 6.1: How the SePo model can improve performance.

time it is requested. This scheme is similar to the way some OS system calls return the `EAGAIN` error code, indicating that the required resources are temporarily unavailable and that the request must be re-issued. As an example use-case of SePo, assume populating a matrix so large that the OS must keep half of it on disk, and further assume the elements are inserted in a random fashion. Random accesses will likely result in excessive page faults and would cause a significant slow down. The SePo model would allow the postponement of accesses to the non-resident portions of the matrix until all accesses to the resident portions have been completed.

The SePo model of computation imposes the following two requirements on an application. First, the application's order of computation should not matter. That is, if we break the computation of the application into independent *computation units* (e.g. processing of independent input records), the order in which the units are processed should not affect the correctness of the application. Second, the application must be structured so that it can tolerate having some of its requests be postponed by the requestee. This means that the application must be able to distinguish between a unit that has been successfully processed and one that has not (due to a postponed request), and be able to re-process the postponed requests at a later time.

Data parallel applications like Big Data analytics can typically satisfy both of these requirements, because their computations contain independent operations performed on a large number of input records that can be processed in any order without affecting the correctness of the computation. Further, it is fairly straightforward to keep track of computation units in these applications, as we show in Section 6.3.3.

A key design parameter in the SePo model is determining who decides when to re-compute postponed computation units. Some applications have the requestee do this because it knows when it has the resources to better service postponed requests. Other applications pick the requestor to make the decision, perhaps because it better knows the scheduling requirements of the computation units or how long they can tolerate being postponed. Still, others involve both the requestee and requestor in making the decision, for example by having the requestee make the initial decision but allowing the requestor to override it.

Figure 6.1 shows how the SePo model can improve the performance of an application. It considers two scenarios for processing a sample computation unit. There is a basic trade-off between servicing the request inefficiently and the added overhead of some re-computation and servicing the request more efficiently. More precisely, the SePo model is effective when the following condition is true:[2]

$$(t_{pre-computation} + t_{postpone}) + (t_{pre-computation} + t_{postponed-service} + t_{post-computation}) < (t_{pre-computation} + t_{service} + t_{post-computation})$$

where $t_{pre-computation}$ is the expected time between the start of the computation unit and the time the requestor issues the request including all of the direct or indirect overheads that starting a computation unit might entail (e.g., data movement overheads); $t_{postpone}$ refers to the overhead of postponing a service including keeping track of whether the unit has been processed or not and reverting back/disposing any result that may have been produced during the corresponding $t_{pre-computation}$; $t_{postponed-service}$ and $t_{service}$ refer to the expected time to provide the service of the computation unit when it is postponed and when it is not, respectively; and finally, $t_{post-computation}$ is the time it takes to finalize the computation unit (e.g., recording a log) when the corresponding request is successfully serviced.

6.2.2 Using SePo for larger-than-memory hash tables

We use *Page View Count*, an application that stores its results in a hash table, to describe how the SePo model of computation might work on a hash table in practice. The application reads in a large log, where each line consists of an input record containing a URL. It extracts the URL and inserts the KV pair <`url, 1`> into the hash table. On each insert, the hash table automatically *combines* KV pairs with the same key, so that the hash table would store <`url, n`> if the KV pair <`url, 1`> had been inserted n times.

In this example, we assume that a hash table large enough to be able to store a pointer per unique URL easily fits in GPU memory, and that closed hashing with chaining is used to deal with collisions. All of the remaining GPU memory is allocated for a heap to be used by a memory allocator. Memory for each KV pair to be stored in the hash table is allocated dynamically from this heap.

With respect to the SePo model of computation, the application in this example is the requestor and the hash table is the requestee. For each <`url, 1`> insert request, if the key (i.e., the `url`) is already in the hash table, then its value is combined with the currently stored value of the key. If the key is not yet in the hash table, then space is allocated for the new KV pair (<`url, 1`>) before it is inserted into the hash table. If the space allocation is unsuccessful, then requestee responds to the insert request with $POSTPONE$, and the input record is marked as not having been processed.

[2]The condition might be slightly different for each application. For instance, some applications, like ours, might postpone a request multiple times before successfully servicing it.

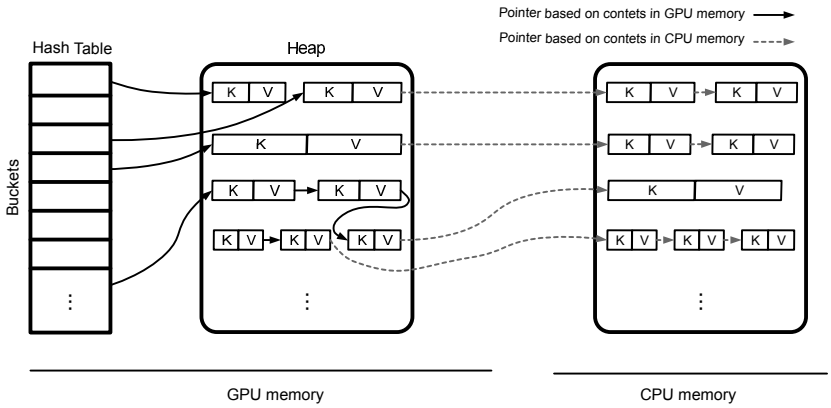

Figure 6.2: A snapshot of a hash table during a subsequent iteration of computation (not all pointers are shown in this figure).

The application iterates over the entire set of input records multiple times in sequence until all input records have been successfully processed. In each iteration, it only considers input records that have not yet been processed, and it attempts to insert the KV pair <`url,1`> for the `url` extracted from each input record in sequence. If the key being inserted has previously been inserted (from an earlier record), then an existing KV record with that key is guaranteed to be in GPU memory, and thus the value of the new KV pair (i.e., `1`) can be combined with the currently stored value. Otherwise, space for the new KV pair will have to be allocated (which, as described earlier, may or may not be successful).

Each time the application reaches the end of the set of input records, the hash table is signalled that it will no longer actively need any of the KV pairs currently located in GPU memory. The hash table then copies all of the pairs to CPU memory and frees up the heap in GPU memory to make it ready for the next iteration. Note that it will no longer need any of the KV pairs being copied back to CPU memory because all pairs (generated from the input) with the same keys will have already been successfully inserted/combined.

Figure 6.2 shows a snapshot of a hash table during a second iteration of computation. Note that new KV pairs are always inserted at the head of the bucket linked list so that there is no need to traverse the linked list elements that might no longer be in GPU memory. Moreover, our implementation stores a set of two pointers in the hash table where ordinarily one would be used: one that is based on the location of contents in GPU memory and another that is based on the eventual location of contents in CPU memory – when the hash table contents are copied to CPU memory. The hash table calculates and sets the value of CPU pointers (using simple pointer arithmetic) while inserting KV pairs. This allows the hash table to be eventually accessible from both CPU and GPU sides.

6.3 Design Overview

The hash table we designed has three key attributes: (i) it supports variable-sized keys and values, (ii) it can perform on-the-fly grouping of KV pairs with the same key and, (iii) it uses the SePo model of computation to be able to exhibit reasonable performance even if it grows larger than available GPU memory.

With a closed addressing method, our hash table uses separate chaining with linked lists. Inserted KV pairs that map to the same bucket will be stored in a linked-list of *entries* rooted in the table. The entries are dynamically allocated as KV pairs are inserted, using a custom dynamic memory allocator we designed for this purpose.

Using separate chaining with dynamic allocation of entries has a number of important advantages for our target applications. First, it allows the hash table to approach and surpass a load factor of 1 while having its performance degrade gracefully. This is an important attribute for our target applications considering that the number of key-values generated by a Big Data analytics is often difficult to predict.[3] Using an open-addressing hash table – e.g., one that uses Cuckoo hashing – can result in costly *insert* operations and even more expensive hash table re-organizations when hash table approaches a load factor of 1.

Second, dynamically allocating bucket entries allows the hash table to start with nothing but a simple array of $null$ pointers, requiring little space. This allows the array to be allocated with a large number of elements – i.e. buckets – without allocating too much memory. Having a large number of array elements reduces lock contention among GPU threads when performing hash table operations.

Third, dynamic memory allocation allows bucket entries to be allocated when they are required, and allocated exactly as large as they need to be. This not only preserves GPU memory, but also adds support for variable-sized KV pairs. A hash table that pre-allocates bucket entries has a difficult time supporting variable-sized KV pairs and often pre-allocates the entries conservatively large so that they can hold a wide range of KV pairs, hence consuming an unnecessary large amount of memory.[4]

In the following we describe the detailed design of our memory allocator, how we organize buckets, and how exactly we apply the SePo model of computation.

6.3.1 Dynamic Memory Allocator

Although a number of dynamic memory allocators designed for GPUs already existed [104], we implemented our own custom-designed dynamic memory allocater for two reasons. First, we did not require the allocator to support memory deallocation (as almost all existing dynamic memory allocators do) because our hash table only ever inserts elements and never deletes them. This simplifies the design of the dynamic memory allocator

[3] Sometimes even different input datasets result in significantly different number of KV pairs being generated by a single Big Data analytic application.

[4] For example, Inverted Index deals with URLs that are between 5 and thousands of characters and thus, requires the hash table to conservatively pre-allocate buckets of thousands of bytes.

substantially. Second, we had a reasonably good understanding of the memory allocation pattern that is exhibited by our hash table operations and thus, could design the dynamic memory allocator that can take advantage of this knowledge toward achieving a higher performance (see below).

The dynamic memory allocator we designed for our hash table uses a heap that is pre-allocated in GPU memory. The heap is partitioned into pages, from which allocation requests are serviced. To determine the largest size the heap can be allocated as, we (i) wait until all other data structures have been allocated, (ii) then query GPU memory for its remaining free space, and (iii) allocate the heap with that size.

The primary objective of our dynamic memory allocator is to achieve high performance when used by 1,000's of concurrent threads. This is essential because the dynamic memory allocator is used in the critical path of GPU threads that populate the hash table. To make the allocator's service scalable, we distribute the allocation load onto multiple pages, instead of having all allocations serviced from one page. This way, instead of accessing one free-list pointer, the accesses are distributed over multiple free-list pointers (one per accessed page), reducing memory access contention. To do this, we partition the hash table buckets into *bucket groups*, each containing n contiguous buckets, and we allocate memory for each bucket group from a different page.

While having several pages to allocate memory from improves the performance of the memory allocator, it increases the chance of memory fragmentation as some pages might not be fully used when the allocator fails to allocate memory for some allocation requests. This is a trade-off in which the right balance might be different for each application. The hash table library, therefore, allows each application to further balance this trade-off by adjusting the size of the bucket groups, which in turn changes the number of pages from which allocations are actively serviced from (e.g. a larger bucket group will have the hash table to be partitioned into fewer bucket groups and thus, distributes the allocation load onto fewer number of pages).

Figure 6.3 illustrates the possible state of a hash table during runtime, along with its bucket groups, and allocated memory pages. Initially, no page is assigned to any of the bucket groups. A page is assigned to a bucket group on the first *malloc()* done by a GPU thread for any bucket of that bucket group. Further, if on a *malloc()* the KV pair does not fit in the free space of the currently allocated page, then a new page is allocated from the memory pool and chained to the previous page(s) allocated to the bucket group. For example, the *bucket group #1* shown in Figure 6.3 has three pages where the first one (the one on the far right) was added first and the other two were added when the existing page(s) ran out of free space.

6.3.2 Bucket Organizations

Key-value pairs that are generated by Big Data analytics applications often have duplicate keys. However, different Big Data analytics applications handle such pairs differently. In our hash table design, we consider three

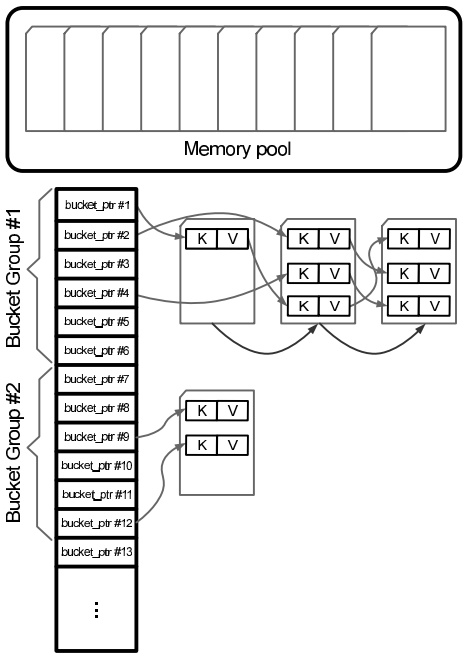

Figure 6.3: The overall structure of bucket pointers, bucket entries, and memory pages.

different bucket organizations to cater to different kinds of KV pair handling: *basic* method, *multi-valued* method, and *combining* method. We describe each in turn.

Basic method:

If two KV pairs have the same key, the two KV pairs will be stored as two separate entries in the linked list of bucket entries. This approach is generally suitable for applications that do not require the pairs with identical keys to be grouped.

Multi-valued bucket entries:

A separate list of values is associated with each key, resulting in a two dimensional linked structure with keys linked along one dimension and values linked to the their keys along a second dimension.

An example application that uses this method is *Inverted Index* which takes HTML pages as input and outputs a *1:N* mapping from the hyperlinks seen in the pages (keys) to the pages that have those hyperlinks in them (values). To do this, each time a hyperlink is found in a page, a pair in the form of <*hyperlink, pagePath*> is inserted into the hash table. At the end of the execution, each bucket entry will have one or more URLs (i.e. keys) and a list of `pagePaths` associated with each URL. For example, if the hyperlink `http://google.com` is found in documents `a.html`, `c.html`, and `d.html`, the final bucket entry will look like the structure in Figure 6.4.

We store keys and values in separate pages when the *multi-valued* method is used. This allows the set of values grow independently of the set of keys and thus, offers more flexibility in handling the keys and values, which is essential for the SePo model (see Section 6.3.3). However, separating keys from values when storing them in the

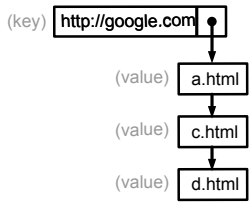

Figure 6.4: The final structure of an example bucket entry under the *multi-valued* method.

hash table increases the chances for memory fragmentation.

Combining method:

This method is inspired by the *combiner* method typically used in MapReduce applications, which is implemented to aggregate (i.e., *reduce* in MapReduce terminology) multiple values into a single value during the *map* phase [95]. Memory space needs to be allocated only the first time a KV pair with a given key is inserted. When inserting a KV pair with a key that already exists in the hash table, then it is only necessary to update the value of the existing bucket entry with the corresponding key.[5]

With the combining method, the hash table *insert* function calls a callback function to handle the corresponding update; it is called every time a new pair with a duplicate key is inserted. The function is called with the *to-be-inserted value* and a reference to the *existing value* as arguments.

To better understand how the combining method works, consider the *Page View Count* application (PVC) we presented earlier. PVC counts the number of occurrences of each URL in a large log file. The inserted KV pairs in PVC are of the form *<URL, 1>*, which means the *URL* has been seen once in a part of the input log file. To calculate the total number of times *URL* exists in the entire log file, all *1*'s (i.e., values) inserted with the same key should be summed up. To do this, the `update_value` callback function for PVC can be implemented as follows:

```
void update_value(const char* key,
                  int& value_existing,
                  const int value_new)
{
    value_existing += value_new;
}
```

Figure 6.5 shows a snapshot of the hash table when using each of the three different bucket organizations for PVC. As can be seen, providing the additional bucket organization methods can potentially save a substantial amount of memory, which is important when designing applications for GPUs. Moreover, on-the-fly grouping of entries with duplicate keys saves runtime by not requiring a separate grouping phase that is otherwise required to

[5]Our hash table does not allow the value to grow in size when being combined. An application with values that grow in size must use the *multi-valued* method and combine the values separately at the end of the execution.

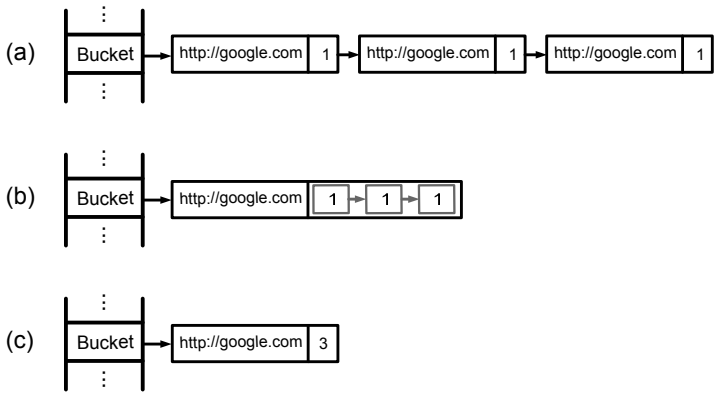

Figure 6.5: A snapshot of the hash table when filled by PVC data under three bucket organizations: (a) *basic*, (b) *multi-valued*, and (c) *combining*.

run after the hash table is fully produced.

6.3.3 Applying the SePo model of computation

Here we describe how the Big Data analytics applications that use our hash table operate with the SePo model of computation. We only focus on how the SePo model handles hash table *inserts* while the hash table is being populated, because it is typically the only operation used when Big Data analytics applications process the input data during the first phase of the application, and because the first phase is typically the one with the highest computational demand (which is why we propose to use GPUs to accelerate it). The SePo model can also be used for *lookup* operations on larger-than-memory hash tables when subsequent phases use/analyze the results but we leave that to the reader as a mental exercise.

The application starts by processing input data records, inserting the generated KV pairs into the hash table. Initially, all *inserts* will be successful, since all GPU-side pages have free space to store the inserted pairs. Every time a memory allocation request is made to a GPU-side page that is full, our dynamic memory allocator allocates a new page from the memory pool to satisfy the allocation request. After some time, however, the memory pool runs out of free pages, and then if more pages run out of free space, the hash table will be unable to to store some of the pairs the application is trying to insert. The hash table *insert* method returns a boolean value to the requestor indicating whether it has successfully stored the pair or not (i.e., $SUCCESS$ or $POSTPONE$). In our implementation of PVC, we use a bitmap array to record which computation units have been successfully processed. A $SUCCESS$ return value causes the appropriate bit to be set.

With the SePo model of computation, there may come a time when the computation will need to be halted so that the data and computation can be rearranged. For example, when no more KV pairs can be inserted into the hash table due to a full heap, the computation may need to be halted so that the heap can be copied to CPU

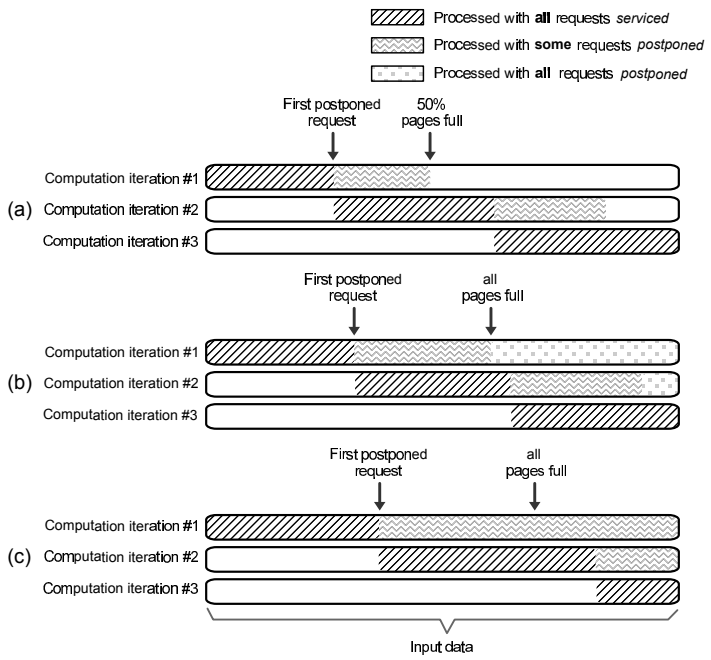

Figure 6.6: How input data is processed using each of the three bucket organization methods: (a) *basic*, (b) *multi-valued*, and (c) *combining*. Note that in (c), even after all pages get full, pairs with duplicate keys are still stored in the hash table.

memory and freed up in GPU memory before the computation can continue. Both decisions, when to halt the computation and how to rearrange the data and computation depends to a large extent on whether the *basic*, the *multi-valued*, or the *combining* bucket organization method is used. For this reason we describe them separately.

Basic method: for applications that use the *basic* method, Figure 6.6 (a) shows the points where the computation stops/restarts. The computation is allowed to continue until the requests from 50% of the bucket groups, a configurable parameter, are being postponed – i.e., when 50% of the bucket groups fail to allocate more memory for the inserted KV pairs.[6] Once the 50% threshold is reached, (i) the computation is halted, (ii) the entire heap on GPU memory is copied back to CPU memory, (iii) the heap on GPU is freed up, adding the pages back to the memory pool and, (iv) the computation restarts to process input data records from the point where a request was postponed for the first time in the previous iteration.

An alternative approach is to not halt the entire computation, but only halt the threads that are unsuccessful in allocating more memory until a page is freed up in GPU memory. However, this approach is expected to be inefficient because efficient GPU hardware interrupt support does not exists and because the cost of extra synchronization that this method needs is high.[7]

[6] We observed acceptable performance with setting the threshold to 50%.

[7] Note that, if we halt a GPU thread in software, it will have to spin-wait, which also causes all of its 31 neighbor threads in its warp to wait. Given the high latency of CPU-GPU communication, these threads will have to wait a long time before a page is freed up, which wastes a significant aggregate computing power. Even if a hardware interrupt support is provided, this alternative approach might still not work well because, as Zheng et al. envisioned, such hardware support might still halt a large number of GPU threads upon an interrupt [119].

Multi-valued method: for applications that use the *multi-valued* method, Figure 6.6 (b) shows the points where the computation stops/restarts. In each iteration, the computation processes all input records that have not successfully been inserted until the end of the input data has been reached (regardless of the percentage of the requests that are postponed). We need to do this to identify keys that have values that have not yet been inserted. At the end of each iteration (i) instead of copying the entire heap to CPU memory, only those pages are transferred that either are *value-pages* or are *key-pages* that do not contain any key that has values that have not yet been inserted, (ii) the GPU-side pages that were copied to CPU memory are freed up and added back to the memory pool, and (iii) the computation restarts to process input data records from the point where a request was postponed for the first time in the previous iteration.

Combining method: for applications that use the *combining* method, Figure 6.6 (c) shows the points where the computation stops/restarts. The *combining* method uses only one type of page to store both keys and values. Similar to the *multi-valued* method, in each iteration the computation continues to process input records until the end of the input data has been reached, because, even if the heap runs out of memory, pairs with duplicate keys can still be stored since they do not need additional memory space – they would only update the existing values. At the end of each iteration (i) the entire GPU heap is copied to CPU memory, (ii) the GPU heap is freed up, adding the freed pages back to the memory pool and, (iii) the computation restarts to process input data records from the point where a request was postponed for the first time in the previous iteration.

6.4 Implementation

In this section, we describe two of the more interesting hash table details: how we use a set of two pointers where one would otherwise use a single pointer, and how we optimize synchronization.

6.4.1 Interoperability of pointers

We use two pointers where one would ordinarily use a single pointer in the hash table (including the array bucket pointers and the pointers connecting the entries of the bucket). One pointer is GPU based and points to the target object in GPU memory. The second pointer is CPU based and points to the target object when it will eventually be located in CPU memory. To calculate the CPU-based pointer, the hash table library uses the address of the CPU-side memory where the GPU heap content will be copied to as a base address to calculate the location where the data will eventually be stored in CPU memory.

Having these two sets of pointers allows the entire hash table to be used directly both GPU side and CPU side (after the hash table has been completely moved to CPU memory at the end of the execution).

```
void atomicElementInsertion(node* prev, node* toBeInserted)
{
  do
  {
    prevNext = prev->next;
    toBeInsertedValue->next = prevNext;
    oldValue = atomicCAS(&(prev->next), prevNext, toBeInsertedValue);
  } while(oldValue != prevNext);
}
```

Figure 6.7: Pseudo code to insert the $toBeInserted$ element between two existing elements (i.e. $prev$ and $prev- > next$) in a linked list.

6.4.2 Synchronization

Concurrent accesses to a shared data structure like a hash table requires proper synchronization. Having 1,000's of GPU threads synchronize on each operation can lead to heavy lock contention if synchronization is not implemented carefully. In our implementation, we generally favor lock-free over lock-based schemes, and when locks are used, we keep the granularity of the locks as minimal as possible.

We are able to use a lock-free approach for the following operations:

- Inserting a bucket entry in the *basic* method
- Inserting a value into a bucket entry in *multi-valued* method

We used variations of the pseudo code shown in Figure 6.7 to perform these lock-free operations. Here, the atomic Compare-And-Swap instruction is used to insert the $toBeInserted$ element between two existing elements (i.e. $prev$ and $prev- > next$) in a linked list.

The following operations require locks:

- Inserting a bucket entry in the *multi-valued* method
- Inserting a bucket entry in the *combining* method

Both of these operations cannot be implemented with a single, atomic operation because each has multiple sub-operations that have to be performed as a transaction, without having other threads access the objects. For example, to insert a KV pair using the *multi-valued* method, a thread needs to search through some of the existing entries in the corresponding bucket list (those that are GPU resident[8]) to determine whether the key is already

[8]Note that, as described in Section 6.2.2, those entries of a bucket that are not GPU resident are guaranteed to have stored all of their associated KV pairs, thus are not required to be searched through when inserting a new pair.

in the list or not and then perform accordingly – during this time no other thread must be allowed to modify the linked list.

As an optimization, we use two sets of locks for the above operations to reduce the granularity of locks when possible. A *bucket lock* per bucket protects updates to the linked list of bucket entries. A *bucket entry lock* per bucket entry protects updates to the entry. These two sets of locks can be acquired independently; e.g., when locking a bucket entry, other threads can still perform operations on the bucket list or on other bucket entries.

6.5 Use case: a simple MapReduce runtime

To test our hash table infrastructure, we developed a MapReduce runtime that uses BigKernel as the input memory manager, our hash table as the KV store, and a few more lines of code to schedule *map* and *reduce* phases. The runtime leaves the core logic of the application to be implemented by the application programmer inside the *map* and *reduce/combine* functions. Some MapReduce applications do not need a *reduce* phase, in which case the *reduce/combine* function is left empty. Finally, the application programmer is asked to provide an *input data partitioner* function which partitions the input data into smaller chunks, ready to be processed by the *map* functions.

Our MapReduce runtime can be configured by the programmer to work in `MAP_REDUCE` or `MAP_GROUP` modes. `MAP_REDUCE` mode is used for MapReduce applications with a reduce phase that generate final <*key*, *value*> pairs and, `MAP_GROUP` mode is used for application with no reduce phase that generate <*key*, *value***s**> pairs. These two modes are also offered by various other MapReduce runtimes [35, 72, 95].

The flow of execution in our runtime is as follows. First, the *input data partitioner*, which runs on the CPU, is called; it splits the raw input data into smaller chunks. Next, BigKernel pipelines the chunks to the GPU cores where they are processed by a number of *map* function instances – one per input chunk created by the *input data partitioner*. Each instance is expected to generate zero or more KV pairs. The generated KV pairs are inserted into our hash table by the *map* function.

When `MAP_REDCUE` mode is used, the hash table uses the *combining* method and the provided *reduce/combine* function as its callback function to aggregate/update the values associated with each distinct key. This means that the *reduce* phase is embedded into the *map* phase (as opposed to being run only after the *map* phase ends). This saves memory and improves performance [95]. When `MAP_GROUP` mode is used, the hash table uses the *multi-valued* method to group (without reducing) all values associated to a key.

The SePo model of computation is used so that the MapReduce runtime can handle large amounts of input and result data. In fact, we believe the SePo model of computation makes our MapReduce runtime the first GPU-based MapReduce runtime that is capable of processing data for larger than what GPU memory can hold.

Application	Dataset #1	Dataset #2	Dataset #3	Dataset #4
Inverted Index	2 GB	3 GB	4 GB	5 GB
Page View Count	0.6 GB	2.2 GB	3.8 GB	5.8 GB
DNA Assembly	2 GB	4 GB	6 GB	8 GB
Netflix	1.6 GB	3.2 GB	4.8 GB	6.4 GB
Word Count (MapReduce)	0.2 GB	2 GB	3 GB	4 GB
Patent Citation (MapReduce)	0.2 GB	2.0 GB	3.4 GB	4.8 GB
Geo Location (MapReduce)	0.2 GB	1.8 GB	3.2GB	5 GB

Table 6.1: Input dataset sizes used in our experiments.

6.6 Experimental Results

In this section, we present the results of our performance evaluation. We evaluate the performance of our solution not only by measuring its *KV pair insert* rate, but also by evaluating the overall performance of several Big Data analytics applications when they are implemented on GPUs and use our hash table, compared to a corresponding CPU-based multi-threaded version. We further compare the performance of the hash table when using the SePo model of computation against the alternative solutions we described in Section 6.1.

6.6.1 Experimental Setup

We performed our experiments on a PC with a 3.8GHz Intel Xeon Quad Core E5 with 8 hardware threads and 10MB of combined L2/L3 cache, connected to 16GB of quad-channel memory clocked at 1800MHz. All GPU kernels were executed on an Nvidia Geforce GTX 780ti GPU with 2,880 cores each running at 875MHz and 3GB of DRAM with a maximum bandwidth of 336 GB/s. The GPU is connected to the rest of the system via a PCIe Gen3 x16 bus interconnect. All GPU-based applications were implemented in CUDA, using CUDA toolkit and GPU driver release 6.0.1 installed on a 64-bit Ubuntu 12.04 Linux with kernel 3.5.0-23.

For our experiments, we implemented seven applications consisting of four stand-alone Big Data analytics applications (Netflix, DNA Assembly, Page View Count, and Inverted Index), and three MapReduce applications (Word Count, Geo Location, and Patent Citation). These applications were chosen primarily due to the amount of data they need to insert into the hash table. Each application is run with a variety of input dataset sizes, which in turn results in a variable number of KV pairs that have to be inserted into the hash table. Table 6.1 provides details on the application data sets used in our experiments. We briefly describe each application.

Netflix: calculates a similarity score between each pair of users based on their movie preferences [11]. The input data consists of an array of records, each containing two ratings of a movie given by two different users and a few other data values.[9] Each KV pair inserted into the hash table is of the form *<userA&userB, similarity score between two users for a movie>*. The application uses the *combining* method in which the *similarity score* of each

[9]This input data is produced by another application which takes as input the initial ratings of users to movies and produces a result record per each two users that rated the same movie.

two users is aggregated.

DNA Assembly: merges fragments of a DNA sequence to reconstruct a larger sequence [9]. The input data consists of fixed-length string records, each containing a DNA fragment and a few other data values. Each KV pair inserted into the hash table is of the form <*part of the DNA fragment, edges of the fragment*>. The application uses the *combining* method in which the uniqueness of the edges of the DNA fragment is updated based on the new value.

Page View Count: counts the number of occurrences of each URL in a web log. The input data consists of a sequence of new-line separated records, each of which containing a URL, an IP address, and a few other values. Each KV pair inserted into the hash table is of the form <*URL, 1*>. The application uses the *combining* method to aggregate the values associated with each URL.

Inverted Index: builds a reverse index from a series of HTML files. The input data consists of a set of HTML files, each containing zero or more links (i.e., $< a >$ tags). Each KV pair inserted into the hash table is of the form <*link URL, HTML file path*>. The application uses the *multi-valued* method to associate all *HTML file paths* with the same *link URL*.

Word Count (MapReduce): counts the number of occurrences of each word in a document. The input data consists of a text file. Each KV pair inserted into the hash table is of the form <*word, 1*>. The application uses `MAP_REDUCE` mode to aggregate the values associated with each word.

Geo Location (MapReduce): groups Wikipedia articles based on the geographic location from which they have been created. The input data consists of a sequence of new-line separated records, each containing information about an article including the geographic location from which it was posted. Each KV pair inserted into the hash table is of the form <*geographic location string, article ID*>. The applications uses `MAP_GROUP` mode to group all *article IDs* with the same *geographic location string*.

Patent Citation (MapReduce): produces a reverse patent citation directory – similar to what Google Scholar offers by the "cited by" functionality. The input data consists of a text file with fixed-length records each containing a citation made by a patent. Each KV pair inserted into the hash table is of the form <*the cited patent, the citing patent*>. The application uses `MAP_GROUP` mode to group all *patents* that have cited the same *patent*.

We modified our Big Data analytics applications to use BigKernel as the underlying input memory management framework. BigKernel, as described in chapter 4, helps minimize the overhead of transferring input data from CPU to GPU memory. Having more efficient input data transfer between CPU and GPU is especially important with the SePo model of computation, because input data may be transferred to GPU memory multiple times.

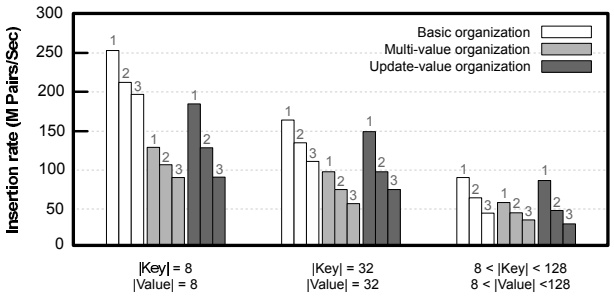

Figure 6.8: KV pair insert rate when using different key and value sizes, different bucket organization methods, and up to three iterations of computation.

6.6.2 Overall results

All of the execution times presented in this section include the input data transfer from CPU to GPU and transfer of the hash table from GPU to CPU. We believe this is the only fair way of comparing GPU implementations to CPU ones. Moreover, all CPU implementations that require dynamic memory allocation use `TCMalloc` [28] which is substantially faster than glibc's `malloc` in multi-threaded applications. Finally, all GPU-based implementations are configured to run with the number of GPU threads that result in the best execution time, as determined through experimentation.

KV pair insert rate

In our first experiment, we measure the rate at which KV pairs can be inserted into the hash table. To do this, we insert random KV pairs into the hash table. However, we make sure that there are many pairs with duplicate keys (the average number of pairs with duplicate keys depends on the dataset, as described below) so as to evaluate the efficiency of the hash table when handling pairs with duplicate keys.

Figure 6.8 presents the performance of insert operation with different key and different value sizes, when using different bucket organization methods, and for three different numbers of KV pairs in the hash table. The first number of KV pairs fits entirely in GPU memory, but the second and third number or pair need slightly less than two and three times the available size of GPU memory to be stored, thus requiring two and three iterations of computation to be completely stored, respectively. The average number of duplicate keys are 2, 5, and 10 in the three numbers of KV pairs, respectively.

Insert under the *basic* method exhibits the highest performance primarily because it does not need to search for a key during insertion and because insertion can be done using the lock-free method. Between the *multi-valued* and *combining* methods, the former exhibits the worst performance because it involves more dynamic memory allocations (it allocates memory for values of duplicate keys while the *combining* method does not) and more data copying (given the extra values).

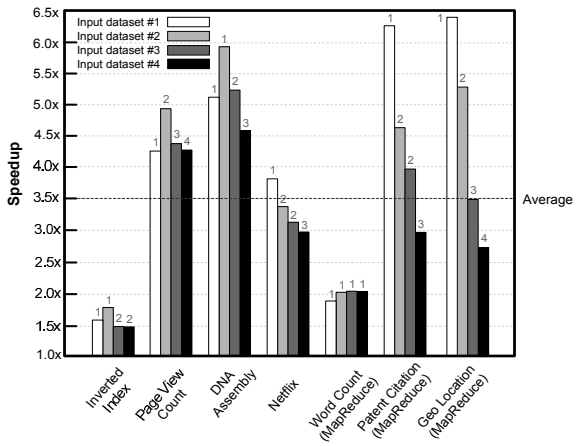

Figure 6.9: Application speedup over CPU multi-threaded implementation. For the last three, the baseline is Phoenix++.

Application performance

We compared each of the four non-MapReduce GPU accelerated applications using our hash table with a CPU-based multi-threaded implementation. The CPU-based versions use a hash table design similar to our GPU-based hash table design except that they do not use the SePo model of computation given that the entire hash table fits in CPU memory for all of our input datasets. The three MapReduce applications built with our MapReduce runtime are compared against the corresponding CPU-based applications developed using Phoenix++, a state-of-the-art MapReduce runtime for multi-core CPUs [95].[10]

Figure 6.9 depicts the achieved speedups of the GPU-based applications over their CPU-based multi-threaded counterparts for different dataset sizes. The numbers shown on top of the bars indicate the number of iterations that were necessary to successfully store all KV pairs into the hash table when using the SePo model of computation. Focusing only on the datasets that are processed in a single iteration of computation – bars with `1` shown on their top – the applications exhibit a range of performance gains when accelerated with GPUs. All applications except Inverted Index and Word Count exhibit reasonable speedups. The average speedup is 3.5X.

Inverted Index and Word Count do not perform as well on GPUs for different reasons. Inverted Index has a long *switch-case* block in its core logic, which causes a high degree of thread divergence in GPUs, negatively affecting performance. Word Count suffers from lock contention when accessing buckets because of the small number of distinct keys and large number of duplicate keys.[11] A CPU implementation also suffers from lock contention, but not as much, given the significantly lower number of threads that run on the CPU. In fact, when we artificially increased the number of distinct keys in the input dataset of Word Count (by adding random, meaningless words to the input documents), performance quickly improved (not shown).

[10]The source code for all applications is available at *https://github.com/rezafmk/SePo_HashTable*.
[11]For instance, the number of occurrences of the word 'the' in a document is high.

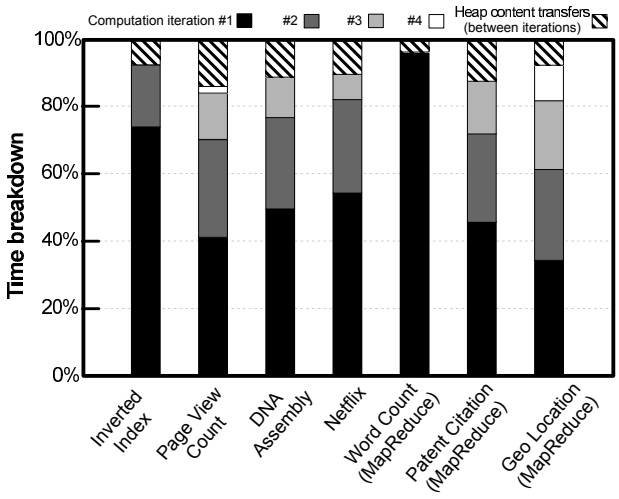

Figure 6.10: The time breakdown of the applications' runtime.

The number of iterations an application/dataset needs depends on how much memory is needed to store the KV pairs, which in turn depends on the number, size, and uniqueness of KV pairs and also on the bucket organization method used. For example, Word Count will rarely need multiple iterations even for large input datasets because (i) Word Count uses the *combining* method which saves memory by not allocating memory space for KV pairs with duplicate keys and (ii) the input dataset of Word Count typically consists of text documents which contain a limited number of distinct words no matter how large the documents is.

Figure 6.10 illustrates the breakdown of execution time when running applications with our largest available dataset – input dataset #4. We see that generally, each iteration (including the time it takes to transfer the heap contents to CPU memory after the iteration is finished) takes less time to complete. For instance, in the applications we experimented with, the second, the third, and the fourth iterations of computation are, on average, 1.94X, 3.84X, and 5.4X faster than the first iteration, respectively.[12]

6.6.3 Comparing with MapCG

We took MapCG as a state-of-the-art GPU MapReduce system and implemented our three MapReduce applications on it to compare it with the MapReduce system we built using our hash table [37]. MapCG also uses a hash table to store KV pairs generated by the *map* function instances. Similar to all existing GPU-based MapReduce runtimes that use a hash table, however, MapCG is unable to support a larger-than-memory hash table, and thus the execution fails when there is no more free memory to store newly inserted KV pairs. In fact, we were able to compare the performance of MapCG with our own MapReduce runtime only for the smallest input datasets (input

[12] One should note that these numbers will differ substantially for different applications and input datasets.

Application	Speedup
Word Count (MapReduce)	1.05X
Patent Citation (MapReduce)	2.42X
Geo Location (MapReduce)	2.55X

Table 6.2: Speedups over MapCG.

datasets between 200MB-600MB despite having a GPU with 3GB of internal memory[13]).

Not being able to process large input datasets in these experiments means that our hash table was, effectively, not using the SePo model of computation (i.e., no KV pair insertions were postponed). Consequently, the comparison with MapCG only evaluates the efficiency of the basic design of our hash table, including dynamic memory allocation and synchronization.

Table 6.2 lists the speedups of the three applications when run using our MapReduce runtime over MapCG on the same testbed. Our MapReduce runtime performs on par with MapCG for Word Count, primarily because the performance of both runtimes are limited by the heavy contention for locks during the KV pair insertions. For the other two applications, however, our MapReduce outperformed MapCG by over a factor of 2.

6.6.4 Comparing with alternative approaches

In this section, we compare the performance of our hash table to the two alternative system-level solutions we described in Section 6.1 that potentially could be used to allow a larger-than-memory hash table for GPUs, namely (i) pin the hash table in CPU memory, and (ii) using a hardware demand paging mechanism.

Hash table pinned to CPU memory

When a memory region is pinned in CPU memory, the operating system will not page it out to disk and it can be accessed directly by GPU threads over the PCIe bus. Given that typical CPU memories are much larger than GPU memories, a much larger hash table can be allocated and fully populated (by the GPU) without needing the SePo model of computation.

As an experiment we modified our dynamic memory allocator to pre-allocate its heap as a pinned CPU memory region (thus storing the content of the hash table in CPU memory). Everything else is kept in GPU memory for higher memory performance (e.g. locks). The heap is allocated sufficiently large so that the hash table's entire content can fit in it. We ran all applications with the largest dataset (i.e. input dataset #4) on this new version of the hash table and compared it with our GPU-based hash table using SePo.

Figure 6.11 shows the result of this experiment in which we show the speedups of our applications when using this modified version of the hash table as well as when using our version of the hash table. Speedup is measured

[13]Even though our testbed GPU has 3GB of memory space, its memory is shared among different data structures and thus each data structure is given a smaller space.

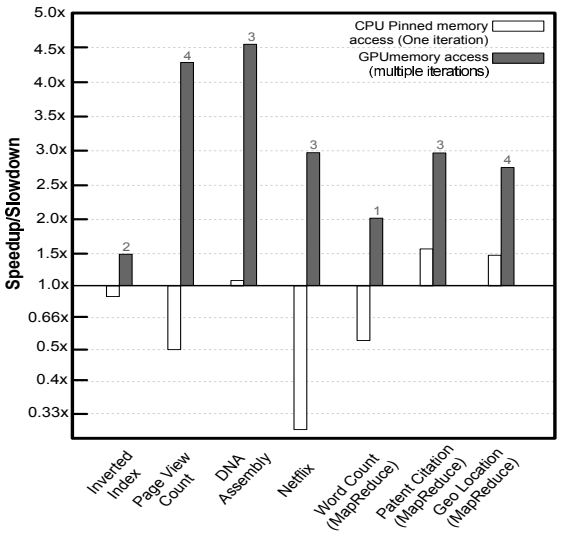

Figure 6.11: Speedups compared to the pinned version.

relative to the CPU-based multi-threaded implementation of the applications. Even though the original hash table needs multiple iterations of computation to process all of the input data, it still significantly outperforms the version that allocates the heap in CPU pinned memory. Worse, in four out of seven applications, the CPU pinned memory version of the hash table performs worse than the CPU-based multi-threaded implementations. The reason for this poor performance is not only the high volume of data that has to be transferred over the PCIe bus, but the fact that the data is transferred over many small PCIe transactions, which is much costlier than a few bulky PCIe transactions.

CPU-side hash table with demand paging

Another system-level solution that supports larger-than-GPU-memory hash tables is to use a GPU hardware that has built-in demand-paging support [79, 119]. Such GPU would allow the application to allocate more GPU memory than is physically available. It will copy pages between CPU and GPU memories as needed to ensure the accessed data is available in GPU memory prior to completing the access. Due to the irregularity of accesses to a hash table, a larger-than-memory hash table with demand paging is expected to exhibit frequent paging activity which degrades performance substantially.

So far, none of the GPUs in the market have added demand paging support, although we expect it to to be provided in the near future. In the absence of a demand paging hardware, we could have simulated the corresponding hardware with the demand paging support using a GPU simulator to measure the efficiency of this alternative solution, but instead we came up with a simple experiment that provides us with a lower bound on the overhead for this solution. In this experiment, we instrumented the code of PVC to record the access pattern to the hash table. We use this access pattern to simulate and then count the number of page replacements that demand paging

Assumed physical GPU memory	Data transfer time (1MB page size)	Data transfer time (128KB page size)	Data transfer time (4KB page size)	Total execution time with our hash table
1200	0.00s	0.00s	0.00s	1.22s
1100	14.8s	2.04s	0.07s	1.29s
1000	101.6s	11.4s	0.60s	1.37s
900	261.5s	32.5s	1.21s	1.45s
800	496.5s	62.1s	2.14s	1.56s
700	801.4s	104.4s	4.47s	1.65s
600	1178.3s	157.7s	6.12s	1.76s
500	1626.5s	128.7s	7.87s	1.89s
400	2148.3s	292.2s	10.33s	2.02s

Table 6.3: Calculated lower bound data transfer time if PVC was run on a demand paging-equipped hardware compared to the total execution time when PVC is run using our hash table.

hardware would have imposed during the runtime of the application. Multiplying this number by the page size yields the total amount of data that has to be transferred over the PCIe bus, which in turn gives us the lower bound runtime that PVC would have spent transferring data under a demand paging-equipped GPU.

Table 6.3 shows the results of this experiment. The input dataset we used for this experiment ends up populating a hash table that reaches 1.2 GB in size. In our simulations we initially set GPU memory to have 1.2 GB of free space (so that the entire hash table fits in GPU memory and no paging is required, considering that all pages are initially GPU resident). We then ran the experiment multiple times, each time reducing the available free space so as to increase the frequency of paging. For each run, we calculated the amount of the data that had to be transferred between CPU and GPU and, based on that, calculated the time it takes to transfer the data over the PCIe bus, and reported that number in the table as the data transfer time. Even though this data transfer time is only one of the overheads associated with demand paging (others including overhead to initiate PCIe transactions and overhead of page fault interrupt handling) it still, in many cases – including all cases where the hash table is about 1.5 times or more larger than the available GPU memory – exceeds the total execution time of running the application using our hash table.

Chapter 7

Concluding Remarks

GPUs have, so far, rarely been used to accelerate real-world Big Data applications despite their enormous computing power and high memory bandwidth [37, 107]. Three main challenges stood in the way: (i) Big Data is often too large to fit in the GPU's separate, limited-sized memory, (ii) data transfers to and from GPUs are expensive because the PCIe bus that connects the CPU and GPU has limited bandwidth and high latency, and (iii) Big Data is not naturally layed out to allow for coalesced accesses, which is necessary to exploit the high GPU memory bandwidth.

In this dissertation we have presented the design and implementation of BigKernel, S-L1, and a hash table to store key-values. All three aim to address the above challenges so as to make GPUs more suitable for real-world Big Data applications, as described below.

BigKernel is a scheme that provides pseudo-virtual memory to GPU-based applications that operate on Big Data. It uses a four-stage pipeline with automated prefetching to (i) optimize CPU-GPU communication and (ii) optimize GPU memory accesses. It simplifies the programming model by allowing programmers to write kernels using arbitrarily large data structures for applications that process data records independently, thus relieving the programmer from having to partition the data into segments, manage buffers, transfer data between CPU and GPU, and having to invoke GPU kernels multiple times. Straightforward compiler transformations are used to transform traditional GPU kernels into BigKernel. On six benchmark applications, we experimentally showed that BigKernel achieves an average speedup of 1.7 over implementations that use double buffering, and an average speedup of 3 over multi-core CPU implementations. We also showed that BigKernel largely migrated the bottleneck away from the PCIe bus to GPU memory.

S-L1 is a GPU level 1 cache which is implemented entirely in software using SMX shared memory. S-L1 determines, at runtime, the proper size of cache, samples the effectiveness of caching the data of different data

structures, and based on that information, decides what data to cache. Although the software implementation adds 8% instruction overhead to the 10 applications we tested, our experimental results showed that this overhead is amortized by faster average memory access latencies for most of these applications. Specifically, when using S-L1, ten GPU-local Big Data applications achieved speedups of between 0.86 and 4.30 (1.90 avg.) over hardware L1 and between 0.95 and 6.50 (2.10 avg.) over no L1 caching. Combining S-L1 with BigKernel led to speedups of between 1.07 and 1.45 (1.19 avg.) over BigKernel alone, and speedups of between 1.07 and 6.37 (3.7 avg.) over the fastest CPU multi-core implementations.

The GPU-based hash table we developed for storing key-value pairs of Big Data analytics applications is capable of retaining reasonable efficiency even when its data grows beyond the size of GPU memory. This is made possible with the help of SePo, a model of computation we developed to reduce the overhead of CPU-GPU data transfers when the hash table does not fully fit in GPU memory. Under our SePo model of computation, a larger-than-memory hash table will postpone certain operations (i.e., insert or lookup) if they attempt to access non-resident portions of the hash table. Such operations are postponed until the requested portions become resident. Our experimental results comparing GPU-based Big Data analytics applications to their CPU-based multi-threaded counterparts, showed that an average speedup of 3.5 is achieved, despite having the hash table grow up to four times larger than the available GPU memory.

If we consider CPU memory as just another level in the memory hierarchy (behind GPU shared memory, L1 cache, L2 cache, and GPU memory), but with lower bandwidth and higher latency, then the general goal of this dissertation has been to prevent the GPU cores from being data-starved by caching data closer to the computing cores. With BigKernel we use GPU memory to cache CPU memory and with S-L1, we use GPU shared memory to cache GPU memory. And with our hash table, we try to cache as much of the accessed data in GPU memory as possible while also preventing a potential thrashing of the GPU memory.

A recurring theme in all of our solutions is that we try to use some of the GPU's computational power to improve the efficiency of the GPU memory hierarchy. This makes sense because it is not possible to keep all GPU cores busy on the application's computation given that their hunger for data cannot be satisfied. In BigKernel, half of the GPU threads are used to generated prefetching addresses. In S-L1, some of the computational power is used to manage the cache (e.g., executing a monitoring phase and fetching/evicting cache lines based on the memory accesses). And in our hash table, some of the GPU's computational power is used to re-generate certain key-value pairs. Going forward, we believe this theme will be adopted more widely, given how much faster GPU computational power is increasing relative to GPU memory bandwidth.

Finally, while it is understandable that GPU designers need to prioritize optimizations for graphical processing and maintain commodity pricing, we believe that our work provides some indications of how GPU designers could enhance current GPU designs to make them more effective for Big Data applications. In particular, we believe the

following improvements would allow for more efficient execution of Big Data applications on GPUs:

1. One of the most straightforward enhancements would be to improve the bandwidth and reduce the latency of the PCIe bus – its current bandwidth has not improved in the last 6 years and is now 20 times lower than GPU memory bandwidth.

2. Another straightforward enhancement would be to significantly increase the size of the L1 – its current size only supports 0.25 of a cache line per thread when applications run with the maximum number of online threads allowed.

3. We also would like to see the GPU on-chip cache geometry to be more configurable, particularly allowing the cache lines to be smaller.

4. Currently, GPU hardware demand paging support is still very immature and inefficient [119]. We expect it to become more efficient in the near future. Nonetheless, while providing demand paging substantially improves programming convenience, we believe fetching the data from CPU memory to GPU memory "on-demand" is detrimental to GPU performance due to the high latency of the PCIe bus. We believe fetching the data from CPU memory on demand should be considered only as a last resort – only when other methods failed to cache/prefetch the data. Therefore, we believe techniques to automatically prefetch data to GPU memory or to allow the programmer to assist demand paging (e.g., with hints) would be critically important.

Future directions

Given the experience we have gained using GPUs to accelerate Big Data applications with our work on BigKernel, S-L1, and our GPU-based hash table, we see various promising avenues for future work. Two incrementally enhance our existing solutions:

1. We believe S-L1 performance can still be improved substantially. When CPU-GPU communication is not the bottleneck, the GPU memory bandwidth is almost always the bottleneck for Big Data applications given the fact that they are typically memory-intensive applications. Therefore, minimizing GPU memory access overhead should be the primary objective to improve the performance of these applications. S-L1 was developed precisely towards this end. As future work, we intend to reduce the overhead of S-L1 by relying more on compile-time analysis. Using compiler technology, we could avoid transforming memory accesses that access data structures statically known to have poor caching behavior. Moreover, if accesses to all data structures can be statically analyzed, the monitoring phase might also become unnecessary. Depending on

the application access pattern, the benefit of this static analysis will vary. At minimum, it can remove the overhead of executing the instructions that are currently used to access data structures that are determined to not be worthy of caching.

2. We expect GPUs to support demand paging in the near future. Hence one would want to explore modifying BigKernel and our hash table to work more efficiently under demand paging-equipped GPU hardware. For example, BigKernel may be able to prefetch a large indirectly-indexed data structure (e.g., an indirectly-indexed array of records) by first fetching its index data on-demand from CPU memory[1] – assuming that the index is much smaller than the data structure itself.

Generalizing our work to other domains, we believe that the SePo model of computation may also be effective in other contexts, as mentioned in Chapter 6. Both requirements of this model, namely having the application be able to tolerate (i) changing the order of its computation and (ii) postponing some of its computations to a later time, are met by most data parallel applications (including Big Data analytics and many Machine Learning applications). It would therefore be interesting to explore the potential benefits of applying the SePo model of computation to data parallel frameworks (e.g. Apache Spark, Memcached, or Hadoop). These frameworks often have certain operations/tasks that can be performed more efficiently if postponed.

Finally, it would be interesting to use BigKernel, S-L1, and perhaps our hash table to develop "accelerated frameworks" for other computation domains. We demonstrated this by developing a MapReduce runtime out of the union of BigKernel and our hash table that was able to achieve considerable speedups on a number of MapReduce applications (of between 1.9 and 6.4 over the state-of-the-art CPU-based MapReduce runtime). Another interesting domain to target is Machine Learning that has a number of existing frameworks that could be ported to the GPU (e.g. Apache Singa and Spark MLlib). Machine learning applications are similar to Big Data applications in that they deal with large datasets and most likely will also benefit from the GPU's high memory bandwidth.

[1] As described in Chapter 4, currently BigKernel is unable to prefetch indirectly-index data structures.

Bibliography

[1] R. Abdelkhalek, H. Calandra, O. Coulaud, J. Roman, and G. Latu. Fast seismic modeling and reverse time migration on a GPU cluster. In *Proc. of the Intl. Conf. on High Performance Computing & Simulation*, pages 36–43, 2009.

[2] A.M. Aji, M. Daga, and W. Feng. CampProf: a visual performance analysis tool for memory bound GPU kernels. *Tech. Rep. retrieved from http://eprints.cs.vt.edu/archive/00001123/*, 2010.

[3] D. Alcantara, V. Volkov, S. Sengupta, M. Mitzenmacher, J. Owens, and N. Amenta. Building an efficient hash table on the GPU. *GPU Computing Gems*, 2:39–53, 2011.

[4] A.G. Anderson, W.A. Goddard III, and P. Schröder. Quantum Monte Carlo on graphical processing units. *Computer Physics Communications*, 177(3):298–306, 2007.

[5] S.S. Baghsorkhi, M. Delahaye, S.J. Patel, W.D. Gropp, and W.W. Hwu. An adaptive performance modeling tool for GPU architectures. *ACM Sigplan Notices*, 45(5):105–114, 2010.

[6] M. Bauer, H. Cook, and B. Khailany. CudaDMA: Optimizing GPU Memory Bandwidth Via Warp Specialization. In *Proc. of the 2011 Intl. Conf. on High Performance Computing, Networking, Storage and Analysis*, pages 12–21, 2011.

[7] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. In *Proc. of the ACM Transactions on Graphics (TOG)*, pages 917–924, 2003.

[8] L. Buatois, G. Caumon, and B. Lévy. GPU accelerated isosurface extraction on tetrahedral grids. In *Proc. of the Intl. Symp. on Visual Computing*, pages 383–392, 2006.

[9] J.A. Chapman, I. Ho, S. Sunkara, S. Luo, G.P. Schroth, and D.S. Rokhsar. Meraculous: De Novo Genome Assembly with Short Paired-End Reads. *PLoS ONE*, 6(8):1–13, 2011.

[10] J. Chen, B. Li, Y. Zhang, L. Peng, and J. Peir. Statistical GPU power analysis using tree-based methods. In *Proc. of the 2011 Intl. Conf. on Green Computing (IGCC)*, pages 1–6, 2011.

[11] S. Chen and S.W. Schlosser. Map-Reduce meets wider varieties of applications. *Intel Research Pittsburgh, Tech. Rep. IRP-TR-08-05*, 2008.

[12] J. W Choi, A. Singh, and R. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *Proc. of the ACM Sigplan Notices*, pages 115–126, 2010.

[13] B. Coutinho, D. Sampaio, F.M. Pereira, and W. Meira. Performance debugging of GPGPU applications with the divergence map. In *Proc. of the 22nd Intl. Symp. on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 33–40, 2010.

[14] B. Coutinho, D. Sampaio, F.M.Q. Pereira, and W. Meira. Divergence analysis and optimizations. In *Proc. of the 2011 Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 320–329, 2011.

[15] S. Díaz-Pier, S.E. Venegas-Andraca, and J.L. Gómez-Muñoz. Classical simulation of quantum adiabatic algorithms using mathematica on GPUs. *arXiv preprint*, 2011.

[16] T.N. Do, V.H. Nguyen, and F. Poulet. Speed up SVM algorithm for massive classification tasks. In *Proc. of the 4th Intl. Conf. on Advanced Data Mining and Applications*, pages 147–157, 2008.

[17] A. Dziekonski, A. Lamecki, and M. Mrozowski. A memory efficient and fast sparse matrix vector product on a GPU. *Progress in Electromagnetics Research*, 116:49–63, 2011.

[18] M. Eisenbach. Future proofing WL-LSMS: preparing for first principles thermodynamics calculations on accelerator and multicore architectures. Oak Ridge National Labratory, Tech. Rep., 2011.

[19] B.R. Epstein and D.L. Rhodes. GPU-accelerated ray tracing for electromagnetic propagation analysis. In *Proc. of the 2010 IEEE Intl. Conf. on Wireless Information Technology and Systems (ICWITS)*, pages 1–4, 2010.

[20] W. Fang, K. Lau, M. Lu, X. Xiao, C. Lam, P. Yang, B. He, Q. Luo, P. Sander, and K. Yang. Parallel data mining on graphics processors. *Hong Kong University of Science and Technology, Tech. Rep. HKUST-CS08-07*, 2008.

[21] K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In *Proc. of the ACM SIGGRAPH/EUROGRAPHICS Conf. on Graphics Hardware*, pages 133–137, 2004.

[22] J. Fix, A. Wilkes, and K. Skadron. Accelerating braided B+ tree searches on a GPU with CUDA. In *Proc. of the 2nd Workshop on Applications for Multi and Many Core Processors*, 2011.

[23] T. Foley and J. Sugerman. KD-tree acceleration structures for a GPU raytracer. In *Proc. of the 2005 Conf. on Graphics Hardware*, pages 15–22, 2005.

[24] M.S. Friedrichs, P. Eastman, V. Vaidyanathan, M. Houston, S. Legrand, A.L. Beberg, D.L. Ensign, C.M. Bruns, and V.S. Pande. Accelerating Molecular Dynamic Simulation on Graphics Processing Units. *Journal of Computational Chemistry*, 30(6):864–872, 2009.

[25] J. Fung and S. Mann. OpenVIDIA: parallel GPU computer vision. In *Proc. of the 13th Annual ACM Intl. Conf. on Multimedia*, pages 849–852, 2005.

[26] J. Gaur, Raghuram S., S. Subramoney, and M. Chaudhuri. Efficient Management of Last-level Caches in Graphics Processors for 3D Scene Rendering Workloads. In *Proc. of the 46th Annual IEEE/ACM Intl. Symp. on Microarchitecture*, pages 395–407, 2013.

[27] I. Gelado, J.E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.W. Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. In *Proc. of the 15th Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 347–358, 2010.

[28] Sanjay Ghemawat and Paul Menage. Tcmalloc: Thread-caching malloc, 2007. *Retrieved from: http://goog-perftools.sourceforge.net/doc/tcmalloc.html*.

[29] J. Gómez-Luna, J.M. González-Linares, J.I. Benavides, and N. Guil. Performance modeling of atomic additions on GPU scratchpad memory. *IEEE Transactions on Parallel and Distributed Systems*, 24(11):1–13, 2012.

[30] J. Gómez-Luna, J.M. González-Linares, J.I. Benavides, and N. Guil. Performance models for asynchronous data transfers on consumer graphics processing units. *Journal of Parallel and Distributed Computing*, 72(9):1117–1126, 2012.

[31] N.K. Govindaraju, N. Raghuvanshi, M. Henson, and D. Manocha. A cache-efficient sorting algorithm for database and data mining computations using graphics processors. *University of North Carolina, Tech. Rep*, 2005.

[32] C. Gregg and K. Hazelwood. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In *Proc. of the 2011 IEEE Intl. Symp. on Performance Analysis of Systems and Software (ISPASS)*, pages 134–144, 2011.

[33] J. Hall, N. Carr, and J. Hart. Cache and bandwidth aware matrix multiplication on the GPU. *University of Illinois, Tech. Rep.*, 2003.

[34] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-accelerated software router. In *Proc. of the ACM SIGCOMM Conf.*, pages 195–206, 2010.

[35] B. He, W. Fang, Q. Luo, N.K. Govindaraju, and T. Wang. Mars: a MapReduce framework on graphics processors. In *Proc. of the 17th Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 260–269, 2008.

[36] E. Hermann, B. Raffin, F. Faure, T. Gautier, and J. Allard. Multi-GPU and multi-CPU parallelization for interactive physics simulations. *Proc. of the 2010 Euro-Par*, pages 235–246, 2010.

[37] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin. MapCG: writing parallel program portable between CPU and GPU. In *Proc. of the 19th Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 217–226, 2010.

[38] B. Hong-Tao, H. Li-li, O. Dan-tong, L. Zhan-shan, and L. He. K-means on commodity GPUs with CUDA. In *Proc. of the 2009 WRI World Congress on Computer Science and Information Engineering*, pages 651–655, 2009.

[39] D. Horn, J. Sugerman, M. Houston, and P. Hanrahan. Interactive KD-tree GPU raytracing. In *Proc. of the 2007 Symp. on Interactive 3D Graphics and Games*, pages 167–174, 2007.

[40] T. B Jablin, J. A Jablin, P. Prabhu, F. Liu, and David I August. Dynamically managed data for CPU-GPU architectures. In *Proc. of the 10th Intl. Symp. on Code Generation and Optimization*, pages 165–174, 2012.

[41] T.B. Jablin, P. Prabhu, J.A. Jablin, N.P. Johnson, S.R. Beard, and D.I. August. Automatic CPU-GPU communication management and optimization. In *Proc. of the ACM SIGPLAN Notices*, pages 142–151, 2011.

[42] M. Januszewski and M. Kostur. Accelerating numerical solution of stochastic differential equations with CUDA. *Computer Physics Communications*, 181(1):183–188, 2010.

[43] W. Jia, K. Shaw, and M. Martonosi. Characterizing and improving the use of demand-fetched caches in GPUs. In *Proc. of the 26th ACM Intl. Conf. on Supercomputing*, pages 15–24, 2012.

[44] Y. Jiao, H. Lin, P. Balaji, and W. Feng. Power and performance characterization of computational kernels on the GPU. In *Proc. of the 2010 IEEE/ACM Intl. Conf. on Cyber, Physical and Social ComputingGreen Computing and Communications (GreenCom)*, pages 221–228, 2010.

[45] T. Karras. Maximizing parallelism in the construction of BVHs, octrees, and k-d trees. In *Proc. of the 4th ACM SIGGRAPH/Eurographics conf. on High-Performance Graphics*, pages 33–37, 2012.

[46] A. Kerr, G. Diamos, and S. Yalamanchili. Modeling GPU-CPU workloads and systems. In *Proc. of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 31–42, 2010.

[47] F. Khorasani, M. Belviranli, R. Gupta, and L. Bhuyan. Stadium Hashing: scalable and flexible hashing on GPUs. In *Proc. of the 2015 Intl. Conf. on Parallel Architecture and Compilation (PACT)*, pages 63–74, 2015.

[48] Y. Kim and A. Shrivastava. Memory performance estimation of CUDA programs. In *Proc. of the ACM Transactions on Embedded Computing Systems (TECS)*, pages 1–22, 2013.

[49] D. Komatitsch, D. Michéa, and G. Erlebacher. Porting a high-order finite-element earthquake modeling application to Nvidia graphics cards using CUDA. *Journal of Parallel and Distributed Computing*, 69(5):451–460, 2009.

[50] T. Komoda, S. Miwa, and H. Nakamura. Communication Library to Overlap Computation and Communication for OpenCL Application. In *Proc. of 26th IEEE Intl. Parallel and Distributed Processing Symp. Workshops PhD Forum (IPDPSW)*, pages 567–573, 2012.

[51] Y. Komura and Y. Okabe. GPU-based single-cluster algorithm for the simulation of the Ising model. *Journal of Computational Physics*, 231(4):1209–1215, 2011.

[52] N. Kumar, S. Satoor, and I. Buck. Fast parallel expectation maximization for gaussian mixture models on GPUs using CUDA. In *11th IEEE Intl. Conf. on High Performance Computing and Communications*, pages 103–109, 2009.

[53] J. Lai and A. Seznec. *Teg: GPU performance estimation using a timing model*. PhD thesis, INRIA, 2011.

[54] W.B. Langdon and A.P. Harrison. GP on SPMD parallel graphics hardware for mega bioinformatics data mining. *Soft Computing-A Fusion of Foundations, Methodologies and Applications*, 12(12):1169–1183, 2008.

[55] J. Lee, M. Samadi, and S. Mahlke. VAST: the illusion of a large memory space for GPUs. In *Proc. of the 23rd Intl. Conf. on Parallel Architectures and Compilation*, pages 443–454, 2014.

[56] S. Lee, S. Min, and R. Eigenmann. OpenMP to GPGPU: a Compiler Framework for Automatic Translation and Optimization. In *Proc. of the 14th Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 101–110, 2009.

[57] A.E. Lefohn, J.M. Kniss, C.D. Hansen, and R.T. Whitaker. Interactive deformation and visualization of level set surfaces using graphics hardware. In *Proc. of the 14th IEEE Visualization (VIS'03)*, pages 11–23, 2003.

[58] Q. Liao, J. Wang, Y. Webster, and I.A. Watson. GPU accelerated support vector machines for mining high-throughput screening data. *Journal of Chemical Information and Modeling*, 49(12):2718–2725, 2009.

[59] L. Ligowski and W. Rudnicki. An efficient implementation of Smith Waterman algorithm on GPU using CUDA for massively parallel scanning of sequence databases. In *Proc. of the IEEE Intl. Symp. on Parallel & Distributed Processing (IPDPS)*, pages 1–8, 2009.

[60] W. Liu, B. Schmidt, G. Voss, and W. Müller-Wittig. Accelerating molecular dynamics simulations using graphics processing units with CUDA. *Computer Physics Communications*, 28(14):634–641, 2008.

[61] Y. Liu, S. Jiao, W. Wu, and S. De. GPU accelerated fast FEM deformation simulation. In *Proc. of the IEEE Asia Pacific Conf. on Circuits and Systems (APCCAS)*, pages 606–609, 2008.

[62] Y. Liu, B. Schmidt, and D.L. Maskell. CUSHAW: a CUDA compatible short read aligner to large genomes based on the Burrows–Wheeler transform. *Bioinformatics*, 28(14):1830–1837, 2012.

[63] J. Lobeiras, M. Amor, and R. Doallo. Performance evaluation of GPU memory hierarchy using the FFT. In *Proc. of the 11th Intl. Conf. on Computational and Mathematical Methods in Science and Engineering (CMMSE)*, pages 750–761, 2011.

[64] L. Luo, M. Wong, and L. Leong. Parallel implementation of R-trees on the GPU. In *Proc. of the 17th Conf. of Asia and South Pacific on Design Automation*, pages 353–358, 2012.

[65] L. Ma, K. Agrawal, and R.D. Chamberlain. A memory access model for highly-threaded many-core architectures. *Future Generation Computer Systems*, 30:202–215, 2014.

[66] L. Ma and R.D. Chamberlain. A performance model for memory bandwidth constrained applications on graphics engines. In *Proc. of the Intl. Conf. on Application-specific Systems, Architectures and Processors*, pages 24–31, 2012.

[67] W. Ma and G. Agrawal. A translation system for enabling data mining applications on GPUs. In *Proc. of the 23rd Intl. Conf. on Supercomputing*, pages 400–409, 2009.

[68] X. Ma, M. Dong, L. Zhong, and Z. Deng. Statistical power consumption analysis and modeling for GPU-based computing. In *Proc. of the ACM SOSP Workshop on Power Aware Computing and Systems (HotPower)*, pages 1–5, 2009.

[69] L. Marziale, G.G. Richard III, and V. Roussev. Massive threading: Using GPUs to increase the performance of digital forensics tools. *Digital Investigation*, 4:73–81, 2007.

[70] M. McCool. Data-parallel programming on the Cell BE and the GPU using the RapidMind development platform. In *Proc. of the Conf. on GSPx Multicore Applications*, 2006.

[71] J. Meng, V.A. Morozov, K. Kumaran, V. Vishwanath, and T.D. Uram. GROPHECY: GPU performance projection from CPU code skeletons. In *2011 Intl. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11, 2011.

[72] R. Mokhtari, A. Abbasi, F. Khunjush, and R. Azimi. Soren: Adaptive MapReduce for Programmable GPUs. In *Proc. of the 4th Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG)*, pages 118–134, 2011.

[73] R. Mokhtari and M. Stumm. BigKernel – High Performance CPU-GPU Communication Pipelining for Big Data-Style Applications. In *Proc. of the IEEE 28th Intl. Parallel and Distributed Processing Symp. (IPDPS)*, pages 819–828, 2014.

[74] R. Mokhtari and M. Stumm. S-L1: A software-based GPU L1 cache that outperforms the hardware L1 for data processing applications. In *Proc. of the 2015 Intl. Symp. on Memory Systems*, pages 121–132, 2015.

[75] H. Nagasaka, N. Maruyama, A. Nukada, T. Endo, and S. Matsuoka. Statistical power modeling of GPU kernels using performance counters. In *Proc. of the Intl. Conf. on Green Computing*, pages 115–122, 2010.

[76] N. Nakasato. Implementation of a parallel tree method on a GPU. *Journal of Computational Science*, 3(3):132–141, 2012.

[77] Nvidia. Compute Unified Device Architecture (CUDA) programming guide. 2007.

[78] Nvidia. Geforce 8800 Technical Briefs. Retrieved from: $http : //www.nvidia.ca/page/8800\_tech\_briefs.html$, 2007.

[79] Nvidia. GP100 Pascal Whitepaper. Retrieved from: $https : //images.nvidia.com/content/pdf/tesla/whitepaper/pascal - architecture - whitepaper.pdf$, 2016.

[80] Nvidia. Tuning CUDA Applications for Kepler. 2016.

[81] L. Nyland. Inside Kepler. Retrieved from: $http : //gpu.cs.uct.ac.za/Slides/Kepler.pdf$, 2013.

[82] S. Pai, R. Govindarajan, and M.J. Thazhuthaveetil. Fast and efficient automatic memory management for GPUs using compiler-assisted runtime coherence scheme. In *Proc. 21st Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 33–42, 2012.

[83] J. Pan and D. Manocha. Fast GPU-based locality sensitive hashing for k-nearest neighbor computation. In *Proc. of the 19th ACM SIGSPATIAL Intl. Conf. on Advances in Geographic Information Systems*, pages 211–220, 2011.

[84] M. Papadrakakis, G. Stavroulakis, and A. Karatarakis. A new era in scientific computing: Domain decomposition methods in hybrid CPU–GPU architectures. *Journal of Computer Methods in Applied Mechanics and Engineering*, 200(13):1490–1508, 2011.

[85] K. Ramani, A. Ibrahim, and D. Shimizu. PowerRed: a flexible modeling framework for power efficiency exploration in GPUs. In *Proc. of the Workshop on General Purpose Processing on GPUs, (GPGPU 07)*, 2007.

[86] D.W. Ritchie and V. Venkatraman. Ultra-fast FFT protein docking on graphics processors. *Bioinformatics*, 26(19):2398–2405, 2010.

[87] D. Robilliard, V. Marion, and C. Fonlupt. High performance genetic programming on GPU. In *Proc. of the 2009 Workshop on Bio-inspired Algorithms for Distributed Systems*, pages 85–94, 2009.

[88] D. Schaa. *Modeling execution and predicting performance in multi-GPU environments*. PhD thesis, Northeastern University, 2009.

[89] T. Sharp. Implementing decision trees and forests on a GPU. In *Proc. of the 2008 European Conf. on Computer Vision*, pages 595–608, 2008.

[90] X. Sierra-Canto, F. Madera-Ramirez, and V. Uc-Cetina. Parallel training of a back-propagation neural network using CUDA. In *Proc. of the 9th Intl. Conf. on Machine Learning and Applications (ICMLA)*, pages 307–312, 2010.

[91] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc. A performance analysis framework for identifying potential benefits in GPGPU applications. In *Proc. of the 17th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 11–22, 2012.

[92] TS Sorensen, T. Schaeffter, K.O. Noe, and M.S. Hansen. Accelerating the nonequispaced fast Fourier transform on commodity graphics hardware. *IEEE Transactions on Medical Imaging*, 27(4):538–547, 2008.

[93] K. Spafford, J. Meredith, J. Vetter, J. Chen, R. Grout, and R. Sankaran. Accelerating S3D: a GPGPU case study. In *Proc. of the Euro-Par Workshop on Parallel Processing*, pages 122–131, 2010.

[94] S. Stone, J. Haldar, S. Tsao, W. Hwu, B Sutton, and Z. Liang. Accelerating advanced MRI reconstructions on GPUs. *Journal of Parallel and Distributed Computing*, 68(10):1307–1318, 2008.

[95] J. Talbot, R.M. Yoo, and C. Kozyrakis. Phoenix++: modular MapReduce for shared-memory systems. In *Proc. of the 2nd Intl. workshop on MapReduce and its Applications*, pages 9–16, 2011.

[96] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *Proc. of the Intl. Conf. on Compiler Construction*, pages 179–196, 2002.

[97] Y. Torres, A. Gonzalez-Escribano, and D. Llanos. Understanding the impact of CUDA tuning techniques for Fermi. In *Proc. of the 2011 Intl. Conf. on High Performance Computing and Simulation (HPCS)*, pages 631–639, 2011.

[98] I. Torunoglu, A. Karakas, E. Elsen, C. Andrus, B. Bremen, and P. Thoutireddy. OPC on a single desktop: a GPU-based OPC and verification tool for fabs and designers. In *Proc. of the SPIE Conf. on Advanced Lithography*, pages 76–90, 2010.

[99] S. Tzeng and L. Wei. Parallel white noise generation on a GPU via cryptographic hash. In *Proc. of the 2008 Symp. on Interactive 3D Graphics and Games*, pages 79–87, 2008.

[100] Y. Uejima, T. Terashima, and R. Maezono. Acceleration of a QM/MM-QMC simulation using GPU. *Journal of Computational Chemistry*, 32(10):2264–2272, 2011.

[101] S.Z. Ueng, M. Lathara, S. Baghsorkhi, and W. Hwu. CUDA-lite: Reducing GPU programming complexity. In *Proc. of the Intl. Workshop on Languages and Compilers for Parallel Computing*, pages 1–15, 2008.

[102] I.S. Ufimtsev and T.J. Martinez. Quantum chemistry on graphical processing units. 1. Strategies for two-electron integral evaluation. *Journal of Chemical Theory and Computation*, 4(2):222–231, 2008.

[103] V. Vineet, P. Harish, S. Patidar, and P. Narayanan. Fast minimum spanning tree for large graphs on the GPU. In *Proc. of the 2009 Conf. on High Performance Graphics*, pages 167–171, 2009.

[104] M. Vinkler and V. Havran. Register Efficient Dynamic Memory Allocator for GPUs. In *Computer Graphics Forum*, volume 34, pages 143–154, 2015.

[105] T. Wilson, P. Hoffmann, S. Somasundaran, J. Kessler, J. Wiebe, Y. Choi, C. Cardie, E. Riloff, and S. Patwardhan. OpinionFinder: a System for Subjectivity Analysis. In *Proc. of the HLT/EMNLP Conf. on Interactive Demonstrations*, pages 34–35, 2005.

[106] H. Wong, M.M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *Proc. of the 2010 IEEE Intl. Symp. on Performance Analysis of Systems & Software (ISPASS)*, pages 235–246, 2010.

[107] R. Wu, B. Zhang, and M. Hsu. Clustering Billions of Data Points Using GPUs. In *Proc. of the Combined Workshops on UnConventional High Performance Computing Workshop Plus Memory Access Workshop*, pages 1–6, 2009.

[108] Y. Yan, M. Grossman, and V. Sarkar. JCUDA: a programmer-friendly interface for accelerating java programs with CUDA. In *Proc. of the 2009 Euro-Par–Parallel Processing*, pages 887–899. 2009.

[109] C. Yang, F. Wang, Y. Du, J. Chen, J. Liu, H. Yi, and K. Lu. Adaptive optimization for petascale heterogeneous CPU/GPU computing. In *Proc. of the 2010 IEEE Intl. Conf. on Cluster Computing (CLUSTER)*, pages 19–28, 2010.

[110] J. Yang, Y. Wang, and Y. Chen. GPU accelerated molecular dynamics simulation of thermal conductivities. *Journal of Computational Physics*, 221(2):799–804, 2007.

[111] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A GPGPU compiler for memory optimization and parallelism management. In *Proc. of the ACM Sigplan Notices*, pages 86–97, 2010.

[112] K. Yasuda. Accelerating density functional calculations with graphics processing unit. *Journal of Chemical Theory and Computation*, 4(8):1230–1236, 2008.

[113] D. Yudanov, M. Shaaban, R. Melton, and L. Reznik. GPU-based simulation of spiking neural networks with real-time performance & high accuracy. In *Proc. of the 2010 IEEE World Congress on Computational Intelligence*, pages 1–8, 2010.

[114] M. Zechner and M. Granitzer. Accelerating K-means on the graphics processor via CUDA. In *Proc. of the 1st Intl. Conf. on Intensive Applications and Services.*, pages 7–15, 2009.

[115] E. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. On-the-fly Elimination of Dynamic Irregularities for GPU Computing. In *Proc. 16th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 369–380, 2011.

[116] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang. Mega-KV: a case for GPUs to maximize the throughput of in-memory key-value stores. In *Proc. of the VLDB Endowment*, pages 1226–1237, 2015.

[117] Y. Zhang, Y. Hu, B. Li, and L. Peng. Performance and power analysis of ATI GPU: a statistical approach. In *Proc. of 6th IEEE Intl. Conf. on Networking, Architecture and Storage (NAS)*, pages 149–158, 2011.

[118] Y. Zhang, F. Mueller, X. Cui, and T. Potok. GPU-accelerated text mining. In *Proc. of the Workshop on Exploiting Parallelism Using GPUs and Other Hardware-assisted Methods*, pages 1–6, 2009.

[119] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. Keckler. Towards high performance paged memory for GPUs. In *Proc. of the 2016 Intl. Symp. on High Performance Computer Architecture (HPCA)*, pages 345–357, 2016.

[120] A. Zhmurov, RI Dima, Y. Kholodov, and V. Barsegov. Sop-GPU: Accelerating biomolecular simulations in the centisecond timescale using graphics processors. *Proteins: Structure, Function, and Bioinformatics*, 78(14):2984–2999, 2010.

[121] Y. Zhou and Y. Tan. GPU-based parallel particle swarm optimization. In *Proc. of the IEEE Congress on Evolutionary Computation*, pages 1493–1500, 2009.