Evaluation of Modern GPU Architecture Features to Design Efficient Algorithms

Ben Karsin

October 9, 2015

Abstract

In recent years, many-core Graphics Processing Units (GPUs) have been increasing applied to solving genera-purpose problems. However, due to the massively parallel design and unique architectural features of the GPU, many algorithms are not well-suited. Additionally, the relative newness of modern GPUs results in insufficient understanding to design GPU-efficient algorithms. For this work, we propose an in-depth study of key features of the modern GPU architecture, measuring the impact of each on overall performance, and using a series of fundamental algorithms to verify analytic results.

Thus far, we have completed our study of the GPU memory hierarchy, with the results of our shared memory analysis recently published [1]. These results are summarized in this paper, and indicate that efficient shared memory use can significant improve overall algorithm performance. We have begun extending this work with an analysis of other memory components, with promising preliminary results. Concurrently, we have begun analysis of some aspects of the GPU execution pipeline, identifying features that may significant impact performance. We plan to study these features with a detailed analysis, as well as empirical tests, to determine the relative impact of each feature. Once our analysis of the modern GPU architecture is complete, we plan to devise several novel GPU-efficient algorithms to solve several fundamental algorithms that we have identified.

1 Introduction

Many-core Graphics Processing Units (GPUs) are increasingly being used for general-purpose computing [2–5]. The thousands of compute cores that modern GPUs employ enable them to out-perform traditional CPUs for many applications [6–8]. For some applications, GPUs can achieve an order-of-magnitude performance speedup over comparable CPU systems. However, many details of the GPU architecture, such as the memory hierarchy and execution pipeline, have not been extensively studied. Therefore, many algorithms implemented for the GPU suffer from avoidable performance loss. In recent years, several works have attempted to accurately model the GPU architecture [9–11]. However, no one model accurately captures all relevant aspects of the GPU, resulting in an incomplete picture of the factors that impact GPU performance. Therefore, algorithms designed for the GPU often achieve sub-optimal performance.

We propose the detailed study of a wide range of aspects of the modern GPU architecture using micro-benchmarks, sample applications, and experimentation. Through this study, we hope to get a more complete view of the factors that impact GPU performance for a range of application types. Finally, we plan to apply this knowledge to build GPU-efficient algorithms to solve several important, computationally difficult problems that have previously been overlooked on the GPU.

So far, we have completed the study of the shared memory component of the GPU memory hierarchy. We have developed a novel binary search algorithm that avoids shared memory bank conflicts and significantly improve performance. When solving the *batched predecessor search* problem, our algorithm achieves a speedup of up to 2.93, compared with a naive binary search algorithm. The details of this portion of our work is available in our paper, published in HiPC 2015 [1]. We have began extending this work to incorporate an analysis of the global memory system by developing a B-tree data structure that can efficiently search larger data sets on the GPU. Thus far, the contributions of this work include:

- We provide an overview of the GPU memory hierarchy, with an emphasis on the aspects that may cause performance loss.

- Via a detailed analysis of the shared memory system, we determine that the naive binary search algorithm experiences poor performance due to bank conflicts.
- We propose two novel search algorithms that improve shared memory usage, and thus performance. With one algorithm being completely bank conflict-free.
- We present experimental results that validate our analysis and illustrate that shared bank conflicts have a significant impact on algorithm performance.
- Through an initial analysis of *global memory* access patterns and latency, we determine that naive access patterns may provide performance degradation.
- We present and overview of preliminary results that indicate that, by leveraging the B-tree data structure, we can improve search performance on data sets too large for shared memory.

We plan to continue this work by evaluating the key aspects of GPU execution. The single instruction multiple threads (SIMT) execution paradigm is at the heart of GPU performance, and marks a major difference from CPU architectures. Therefore, applying known algorithms designed for the CPU can result in *execution* or *memory divergence*. We propose a series of experiments and micro-benchmarks aimed at evaluating the relative impact of these factors, along with possible causes and solutions. Using the results of our analyses, we plan on developing algorithms to solve several fundamental computational geometry problems. Each of these problems requires different algorithmic techniques to solve, giving us a broad view of the real-world application of our work. Specifically, our planned contributions are:

- Via analysis of SIMT execution divergence, we hope to determine the optimal solution for algorithms that require conditional or branch decisions.
- We will investigate the effects of Instruction Level Parallelism (ILP) on GPU performance for a range of situations to determine the performance benefits and drawbacks of this approach.
- We will present methods of overcoming the negative effects of memory divergence caused by the use of pointers, data structures, and common access patterns.
- By developing algorithms to solve fundamental computational geometry problems, we will verify the impact of our previous findings.
- We will present our novel algorithms designed for the GPU and compare their performance with that of existing solutions to their respective problems.

This paper is organized as follows: Section 2 provides background about the GPU architecture, with particular emphasis on key aspects that may affect the performance of naive algorithms. Section 3 contains an overview of related work. Section 4 details the work completed thus far and Section 5 outlines the scope and planned future research. Finally, Section 6 presents conclusions and possible future work beyond the scope of this study.

2 GPU Architecture

In this section we provide an overview of modern GPU architectures, highlighting key features that we analyze in more detailed in Section 4 or plan to investigate, as outlined in Section 5. For more information about the GPU architecture or any features we discuss, see standard references [3,4]. Modern GPUs comprise thousands of physical processing cores, organized hierarchically into *streaming multiprocessors* (SMs). Each SM contains a small, fast shared memory, and a large, slower global memory is shared among all SMs.

2.1 Execution organization

To achieve optimal utilization of all of its thousands of processor cores, as well as hide memory and instruction latency, GPUs support the execution of many more threads than physical processor cores. To manage the execution of so many threads, the programmer groups threads into cooperative thread arrays (CTAs), also called *thread blocks*. The GPU schedules all threads of a CTA to a single SM, allowing them to access and communicate via the same shared memory partition. The GPU further partitions CTAs into warps, each containing w threads. All threads within a warp execute in SIMT fashion, requiring they execute the same commands at each step. *Branch divergence*, caused when threads' execution diverges, results in serialization execution of each path, which, in the worst case, can result in an order-of-magnitude performance loss [12]. Previous works consider methods of overcoming this issue, though for many algorithms a performance loss is unavoidable [12, 13]. In Section 5.1 we consider the methods of avoiding branch divergence and measure its

effects on overall performance. For more detailed information on the SIMT execution paradigm and branch divergence, we direct interested readers to [14].

2.2 Memory Hierarchy

As discussed in Section 1, the modern GPU architecture includes a memory hierarchy with each level having different latency, throughput, access scope, and optimal access pattern. Although there are additional memory types and caching mechanisms, for this work we focus on the pervasive global memory and shared memory.

Global memory is the primary way to communicate between threads of different CTAs, but previous works suggest that it is at least an order of magnitude slower than other memory types [3, 15], such as shared memory. Consequently, global memory use must be limited to achieve peak performance. Research focused on modeling the GPU hierarchy [2,7,9,10,16] demonstrate that, to obtain close to peak throughput, global memory access must be *coalesced*. Memory access is coalesced when all threads within a warp all access only elements within a 32-element *page*. In this case, all threads obtain the desired element within a single concurrent memory access, where as non-coalesced access results in separate memory accesses for each 32-element page being accessed. See [10,15] for a more in-depth discussion on the global memory access pattern.

Shared memory is a smaller, faster memory that is private to each SM. Each CTA designates its required usage and that much shared memory is assigned to that CTA while it is resident on the SM. Consequently, if a CTA requires a large amount of shared memory, it may prevent additional CTAs from being scheduled on the same SM, potentially reducing performance. The shared memory of each SM is implemented as a series of w *memory banks*, each of which can be accessed independently in parallel. However, if threads within the same warp attempt to access the same memory bank, a conflict occurs and accesses are *serialized*. Shared memory is also capable of *multicasting*, allowing the same address to be accessed by multiple threads at once, thus only concurrent accesses to*distinct* cells of a memory bank cause conflicts. Note that for most modern GPUs, the number of memory banks and threads per warp is equal. Thus, we assume both the number of memory banks and threads per warp is w. We study the impact of bank conflicts on algorithm performance in Section 4.1.

3 Related Work

Although GPUs have only been applied to general-purpose computing for the past 10 years, there have been many studies aimed at improving execution efficiency. Many of these papers focus on efficient solutions to classic problems, including parallel scan and prefix sums [7, 17–19], sorting [13, 16, 20, 21], and graph algorithms [22–24]. However, many of these works focus on engineering a fast solution for a specific problem, with little emphasis put on general techniques for improving GPU algorithm design. While the design and implementation of GPU-efficient algorithms is within the scope of this work, our central focus is understanding the key factors that effect GPU performance.

In recent years, there have been several attempts to analytically model GPU performance [9–11, 25, 26], though no single model has emerged as a standard. The work in [9, 10] focus on modeling the memory hierarchy, with [10] focusing exclusively on the global memory system. While they offer an elegant model of GPU memory, no empirical results are provided. In [26], the authors develop a general model for private-cache multiprocessors and present several algorithms they prove to be optimal, though GPUs have unique features not discussed. In [11], the authors attempt to develop a model that encapsulates the execution pipeline, in addition to memory. However, this work is highly hardware-specific and incompatible with the hardware details of more recent GPUs. Similarly, the work in [25] models specific hardware components, though they focus on modelling GPU power consumption. Our work, rather than trying to model the complex GPU system, aims at identifying the most significant factors that effect performance on the general GPU architecture, regardless of hardware specifics.

Several authors have also empirically studied the global memory system [8, 27]. In [27], the authors develop and run a series of benchmarks to gain insight into the GPU. The work of Ryoo et al. [8] contains a series of optimization techniques centered on global memory usage. While these works provide useful insight

into the global memory of the GPU architecture, they neglect many other aspects of the GPU that can impact algorithm performance, such as shared memory and the execution pipeline.

While we plan on analyzing many aspects of GPU memory and computation, the work completed thus far has focused on shared memory and, specifically, how it can be leveraged for searching. The details of this work is contained in our paper [1], and we provide an overview in Section 4.1.2. While, to our knowledge, no work has focused on efficient searching in shared memory, many works make use of binary search as a subroutine [13,20,28]. In the process of developing a GPU Sample Sort, Leischner et al. [13] eliminate branch divergence during SIMT execution of a binary search within a warp by developing an implementation of the search that uses predicates to eliminate conditional branches. But their implementation does experience loss of parallelism due to shared memory bank conflicts, which our work attempts to prevent. We plan to also investigate the performance implications of the use of predicates to eliminate branch divergence.

We have began extending the work in [1] for searching large data by using data structures to improve memory accesses and shared memory caching. Other works that have looked at using GPU-efficient data structures to improve searching performance have largely focused on global memory access patterns [29–33], while ignoring many other aspects of the GPU that may also impact performance. For *dynamic datasets*, indexing is the most common optimization technique. Kaczmarski [29, 30] studies the construction of the B^+-tree data structure to index data in GPU global memory. By focusing on the B^+-tree, the author takes advantage of coalesced global memory accesses. Similarly, Shekhar [31] proposes GPU-efficient global memory data structures designed to improve performance of the IP lookup operation. Kim et al. [32] create a hybrid CPU/GPU data structure to achieve high peak query throughput. Soman et al. [33] address the limited memory on the GPU by looking at compressing search tree data structures.

In Section 5.4, we outline two computational geometry problems that we plan on solving with GPU-efficient algorithms based on the insights gained from this work. While there have been several recent works that look at solving computational geometry problems on the GPU [34–36], few focus on the problems we outline [37]. To our knowledge, no attempts have been made to solve the problem of dominance counting on the GPU. The work in [37] implements a GPU-assisted Voronoi tessellation algorithm, though they focus on the application itself, rather than the factors that may effect GPU performance.

4 Completed Work

In this section, we provide an overview of the work completed thus far, including research avenues for which we have preliminary results. While some aspects may be yet unknown, we anticipate that all work discussed in this section may yield meaningful results.

4.1 Shared Memory

Shared memory, acting as a fast user-controlled cache, must be efficiently utilized to achieve peak performance. Previous works indicate that algorithms that do not consider the details of shared memory may suffer from performance loss due to bank conflicts [13, 38]. Recall from Section 2 that shared memory on each SM is partitioned into w memory banks, with each bank containing $\frac{M}{w}$ cells, where M is the total shared memory size per SM. Therefore, we must consider both the shared memory size, M, and its relative latency, L_S when evaluating its impact on performance. Using a series of micro-benchmarks, we are able to measure L_S for a range of access patterns and determine the impact of bank conflicts.

4.1.1 Bank Conflicts

Recall from Section 2 that bank conflicts occur when multiple threads of a warp attempt to access *unique* cells of the same memory bank. We consider a range of cases where anywhere from 2 to all w threads of a warp are in conflict with each other. When k threads are in conflict, we say there is a k-way bank conflict. Using micro-benchmarks, we measure the time required for each of 2- to w-way bank conflicts. The results indicate that, if a memory access with *no* bank conflicts takes L_S time, a k-way conflict takes approximately $L_S + \frac{(k-1)L_S}{2}$. This tells us that, while the additional memory accesses required by conflicts are faster than the initial access, their individual cost is a constant factor of L_S. Therefore, worst-case access patterns that result in many w-way conflicts will cause a a memory latency increase by a factor approximately $\frac{w}{2}$. We have

confirmed this result on 3 unique GPU environments and, though other hardware may differ in constant factors, we expect the linear slowdown caused by bank conflicts to be present.

4.1.2 Predecessor Search

To measure the relative impact of bank conflicts on GPU runtime, we consider an algorithm that leverages shared memory. Therefore, we look at a ubiquitous algorithm, *binary search*, on a list stored in shared memory. In this section we provide only an overview of this work, and more detail can be found in our recent publication [1]. To empirically measure the impact of bank conflicts on a parallel application involving binary search, we implement a naive solution to the *batched predecessor search* (BPS) problem:

Definition 4.1 *Given a list $\mathcal{K}$ of K keys $\mathcal{K}[0]$, $\mathcal{K}[1]$, $\cdots$, $\mathcal{K}[K-1]$, sorted in non-decreasing order, and a set $\mathcal{Q}$ of Q queries, the BPS problem asks to find for each $q \in \mathcal{Q}$ the largest i, such that $\mathcal{K}[i] \leq q$.*

An easily parallelizable problem, we divide $\mathcal{Q}$ equally among t threads and copy $\mathcal{K}$ to the shared memory of all CTAs. We perform a parallel binary search (PBS) on for each of $\frac{Q}{t}$ queries per thread. $\mathcal{Q}$ is initially in global memory, and the result of each search is copied back to global memory.

Through a detailed analysis of the memory access pattern of binary search, we show that, for a range of query sets, the naive PBS algorithm suffers from a large number of bank conflicts. We omit the details of this analysis and instead direct interested readers to [1]. In response to the sub-optimal access pattern of PBS, we present two novel search algorithms, the conflict-free (PBS-CF) and conflict-limited (PBS-CL) algorithms. These two algorithms divide the search procedure into two stages based on the number of memory banks, w, and offset the cells accessed by each thread within a warp to reduce bank conflicts. The PBS-CF algorithm eliminates all bank conflicts by performing a linear search for a portion of the algorithm, resulting in a sub-optimal number of operations. PBS-CL, however, suffers from few bank conflicts while remaining work-optimal, requiring $O(\log K)$ steps per query.

Using performance analysis software, we measure the number of bank conflicts of the PBS algorithm and, when compared with average execution time, we find a strong correlation. We compare average execution time of our three algorithms on a range of inputs and find a significant performance disparity, further indicating the impact of bank conflicts. Our results indicate that, while PBS-CF eliminates bank conflicts, the additional work required results in a performance loss. PBS-CL, however, suffers from few bank conflicts and obtains a maximum speedup of 2.93 over PBS. Such a significant speedup tells us that, for applications that can effectively utilize shared memory, we must consider memory access patterns and reduce bank conflicts. For more complete results and experimental details, see [1].

4.2 Global Memory

Most meaningful computation requires data, and solving large problems typically takes large amounts of data. Therefore, it is important to make efficient use of the largest memory system on the GPU, global memory. As outlined in Section 2, global memory is accessible by all SMs (CTAs) and is much larger than shared memory.

4.2.1 Coalesced Access

Since, like shared memory, global memory has a unique optimal access pattern, we must consider the access pattern when designing efficient GPU algorithms. Furthermore, since global memory is many times slower than shared memory, we presume that poor usage may significantly degrade performance. Previous works [2, 15, 16] show that optimal global memory access requires *coalesced* memory accesses, requiring that all w threads within a warp access consecutive elements in global memory. However, through experimentation, we find that global memory access can be better represented by *aligned paged* memory access. If we consider global memory as organized in pages of w elements, each memory access from a warp can load all w elements. If, however, threads attempt to access elements located in different pages, access is un-coalseced and requires additional memory accesses. Furthermore, our experiments indicate that if accessed items are not aligned to a multiple of w, two pages must be loaded.

4.2.2 B-Trees

While the results in Section 4.1.2 indicate that search performance is dependent on shared memory bank conflicts, those experiments assume that the search list $\mathcal{K}$ fit in shared memory. This allowed us to use global memory only for the queries, $\mathcal{Q}$, which required only minimal interaction. Extending this, we now consider when $\mathcal{K}$ does not fit in shared memory, and must reside in global memory. Therefore, we again consider the *batched predecessor search* problem, though with the assumption that $\mathcal{K} > M$.

As a performance baseline, we create a naive global memory search (GMS) algorithm that divides $\mathcal{Q}$ among threads, each of which repeatedly performs a binary search on $\mathcal{K}$ in shared memory. Clearly, this will lead to un-coalesced memory access and we presume will perform poorly. As a method of improving performance, we consider shared memory as a cache and devise methods of using it to reduce un-coalesced global memory access. We consider the b-tree data structure [39] as a possible method of improving performance. Thus far, we have implemented the static b-tree data structure and are currently developing query algorithms that reduce overall global memory usage. Preliminary results indicate that querying the static b-tree is faster than the using the simple GMS algorithm. We plan on developing several b-tree node-caching strategies and comparing the results to determine the performance impact of un-coalesced global memory access, total memory accesses, and shared memory usage. The results of these experiments, combined with those from Section 4.1, will give us the relative performance impact of each key aspect of the GPU memory hierarchy.

5 Planned Work

The work performed thus far has enabled us to identify aspects of modern GPU architecture that, if not considered during algorithm design, can significantly degrade performance. We have focused our efforts on the memory hierarchy, as it requires specific access patterns for optimal performance. In this section, we identify other potentially important characteristics of the GPU and identify ways of evaluating their impact on algorithm performance. Furthermore, we identify several fundamental algorithms that we plan on implementing on the GPU, using the techniques developed throughout this work, to determine the combined potential performance gains.

5.1 Branch Divergence

As outlined in Section 2, the GPU is a SIMT architecture, requiring each thread within a warp to execute in lock-step (i.e., one instruction issued to all threads). When the execution paths of threads within the same warp diverge, this can lead to *branch divergence*, causing some threads to remain idle while others follow their execution path. In the worst-case, this can lead to serialized computation, greatly reducing parallelism and performance [13, 40].

Leischner et al. [13] suggest the use of predicates, rather than conditional branches, to prevent branch divergence and improve performance. We employed this technique to improve the performance of all binary search algorithms in Section 4.1.2 (for further details, see [1]). This approach, however, has not been well-studied, though it may greatly impact performance. While, for certain algorithms, predicates can eliminate branch divergence and improve performance, for other execution patterns we may be required to, effectively, perform the work of all execution paths. Thus, we propose a series of experiments to quantify the performance impact of branch divergence, compared with predicate-based solutions. We plan to identify several simple, non-deterministic algorithms that make heavy use of conditional branches, and compare performance with a predicate-based implementation.

We have began an analysis of branch divergence in hopes of developing an estimate of the performance loss. By considering the probability of each conditional path, together with the cost of execution of that path, we can estimate which paths may be traversed and if a predicate-based variation will improve performance. This work is still in the preliminary stages, however, and we plan to develop a more complete model and verify its accuracy with a series of experiments.

5.2 Instruction-level parallelism

One aspect of the GPU that is similar to the CPU is that they can both leverage instruction-level parallelism (ILP). With ILP, independent instructions are pipelined, increasing instruction throughput and improving performance. However, to leverage this, consecutive instructions must be independent. While the compiler assists with this by re-ordering instructions, there are many cases where instruction dependency is unavoidable.

We propose a series of experiments and analysis of GPU algorithms to measure the potential performance gains associated with ILP. We can use simple algorithms, such as scan, to evaluate the impact of ILP on instruction throughput and overall execution time. By looking at varying instruction orderings and leveraging performance analysis tools, we hope to develop techniques that enable the GPU to efficiently pipeline instructions and reduce overall idle time.

5.3 Data structures and memory

Although we have completed much of our study of the GPU memory hierarchy, there are cases where we cannot adhere to the restrictive optimal access patterns of each GPU memory type. The use of data structures can easily lead to threads having to access memory in sub-optimal patterns. This type of *memory divergence* is to be avoided by GPU-optimal algorithms. Few works have been able to efficiently implement complex data structures on the GPU [29–31]. To address this difficulty, we propose the study of the memory access patterns required by common data structures and their resulting performance on the GPU memory hierarchy.

5.3.1 Pointers

Many data structures make use of pointers to maintain their structure and allow dynamic changes. This, however, can cause non-consecutive memory locations to be used, leading to memory divergence on the GPU. During our investigation of data structure performance on the GPU, we examine the use of pointers and their role in the necessity of divergent memory access patterns. We plan to find memory layouts that provide a more efficient access pattern for shared and global memory on the GPU.

5.3.2 Packed Memory Arrays

One alternative to using pointers for dynamic data structures is the use of cache-oblivious packed memory arrays [41]. With packed memory arrays, we use a larger memory block that can grow as necessary to allow dynamic insertion and deletion of elements. This allows us to use consecutive memory addresses for dynamic structures. Packed memory arrays have been actively studied [42], though, to our knowledge, no works have attempted to apply them to the GPU. We plan on building several data structures using this principle and comparing the performance of various operations with static and pointer-based versions.

5.4 Empirical Evaluation

The work completed thus far has demonstrated that key details of the GPU architecture can have significant consequences on algorithm performance. We expect that our planned investigation of the other key GPU features will provide us with more insight into the design of GPU-efficient algorithms. Using all of the techniques developed by this work, we plan on designing algorithms to solve several fundamental problems on the GPU. By comparing the performance of these algorithms with fast CPU solutions, we will determine the combined benefits of all aspects of this work on algorithm design for modern GPUs.

5.4.1 Dominance Counting

One problem that we identify and, to our knowledge, has never been applied to GPUs, is that of *dominance counting*. Dominance counting is a fundamental problem in computational geometry and can be extended to solve a range of other important problems. The problem can be described as:

Definition 5.1 *Given N points, for each point n_i, determine how many other points it* dominates*, where n_i* dominates *n_j iff, for each coordinate, $n_i > n_j$.*

Despite the seeming lack of dominance counting solutions on the GPU, parallel algorithms that solve the problem have been leveraged for a wide range of applications, including orthogonal line segment intersection [43], orthogonal range searching [44], and spatial skyline queries [45]. Several parallel algorithms exist to solve this problem [43] and, as we continue investigating key aspects of GPU performance, we will determine the best approach and develop and evaluate a GPU-efficient algorithm.

5.4.2 Voronoi Diagram

Another problem with far-reaching applicability is that of *Voronoi diagram* generation. Although computationally difficult to generate, Voronoi diagrams are used in a wide range of fields [46], making fast solutions to this problem important. Informally, we describe a Voronoi diagram as a partitioning of a plane into regions based on the location of a set of points. Like dominance counting, there are several parallel algorithms for generating Voronoi diagrams [47]. We plan on analyzing these algorithms to determine the most appropriate solution for the GPU. This is planned work that falls within the scope of this research, though the details remain mostly unknown.

6 Conclusion

The trend in computing is clearly moving toward parallelism to obtain the largest possible performance gains. The modern GPU architecture represents the leading edge of this trend, providing large amount of compute power at low cost. However, the divergence from CPU architectures, complexity, and relative newness of the GPU paradigm contribute to a lack of understanding. This lack of understanding results in many applications being unable to efficiently leverage GPUs, when they could provide performance that far outstrips comparable CPU systems.

The goal of this work is to analyze a diverse set of GPU architecture features and attempt to determine how they impact performance for a range of application types. Thus far, we have studied the GPU memory hierarchy and identified several key methods of improving performance. Through our study of shared memory bank conflicts, we developed two novel search algorithms that reduce bank conflicts and improve performance. Our novel algorithms achieve a maximum speedup of 2.93 over a naive parallel binary search, the details of which are recently published [1]. We have begun extending this work by incorporating our study of efficient global memory utilization with shared memory caching with B-tree data structures.

We outline plans for future work within the scope of this study that involves analysis of a range of other GPU details. By experimenting with aspects of the GPU execution pipeline, instruction order, memory access patterns, and data structures, we hope to gain valuable knowledge about the factors that most contribute to GPU performance. Using is knowledge, we plan on designing, implementing, and evaluating several GPU-efficient algorithms to solve fundamental problems.

References

[1] B. Karsin, H. Casanova, and N. Sitchinava, "Efficient batched predecessor search in shared memory on gpus," in *Proc. of HiPC*, 2015.

[2] T. A. T. Han, "hiCUDA: High-level GPGPU programming," in *Proc. of TPDS*, vol. 22, 2010, pp. 78–90.

[3] NVIDIA, "CUDA programming guide 7.0," 2015. [Online]. Available: http://docs.nvidia.com/cuda

[4] D. B. Kirk, *Programming Massively Parallel Processors*. Elsevier Science, 2012.

[5] NVIDIA, "CUDA CUFFT library," 2007. [Online]. Available: http://docs.nvidia.com/cuda/cufft

[6] S. Baxter, "Modern GPU," 2013. [Online]. Available: http://nvlabs.github.io/moderngpu/

[7] D. Merrill and A. Grimshaw, "Parallel Scan for Stream Architectures," Department of Computer Science, University of Virginia, Tech. Rep. CS2009-14, 2009.

[8] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in *Proc. of PPoPP*. ACM, 2008, pp. 73–82.

[9] K. Nakano, "The hierarchical memory machine model for GPUs," in *Proc. of IPDPSW*, May 2013, pp. 591–600.

[10] ——, "Simple memory machine models for GPUs," in *Proc. of IPDPSW*, May 2012, pp. 794–803.

[11] Y. Zhan and J. Owens, "A quantitative performance analysis mdoel for gpu architectures," in *Proc. of HPCA*, 2011, pp. 382–393.

[12] W. Fung, I. Sham, G. Yuan, and T. Aamodt, "Dynamic warp formation and scheduling for efficient gpu control flow," in *MICRO*, Dec 2007, pp. 407–420.

[13] N. Leischner, V. Osipov, and P. Sanders, "GPU sample sort," in *Proc. of IPDPS*, April 2010, pp. 1–10.

[14] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, 2008.

[15] N. Fauzia, L. N. Pouchet, and P. Sadayappan, "Characterizing and enhancing global memory data coalescing on GPUs," in *Proc. of CGO*, 2015, pp. 12–22.

[16] D. G. Merrill and A. S. Grimshaw, "Revisiting Sorting for GPGPU Stream Architectures," in *Proc. of PACT*. ACM, 2010, pp. 545–546.

[17] S. Sengupta, M. Harris, and M. Garland, "Efficient parallel scan algorithms for GPUs," NVIDIA Technical Report NVR-2008-003, 12 2008.

[18] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for GPU computing," *Graphics Hardware*, 2007.

[19] Y. Dotsenko, N. K. Govindaraju, P. Sloan, C. Boyd, and J. Manfedelli, "Fast scan algorithms on graphics processors," in *ICS*, 2008.

[20] F. Dehne and H. Zaboli, "Deterministic sample sort for GPUs," vol. abs/1002.4464, 2010. [Online]. Available: http://arxiv.org/abs/1002.4464

[21] N. Sitchinava and V. Weichert, "Provably efficient GPU algorithms," *CoRR*, vol. abs/1306.5076, 2013. [Online]. Available: http://arxiv.org/abs/1306.5076

[22] Z. Wei and J. JaJa, "Optimization of linked list prefix computations on multithreaded GPUs using CUDA," in *Proc. of IPDPS*, 2010.

[23] A. Davidson, S. Baxter, M. Garland, and J. Owens, "Work-efficient parallel GPU methods for single-source shortest paths," in *Proc. of IPDPS*, 2010, pp. 78–90.

[24] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," *SIGPLAN Not.*, vol. 47, no. 8, pp. 117–128, Feb. 2012. [Online]. Available: http://doi.acm.org/10.1145/2370036.2145832

[25] S. Hong and H. Kim, "An integrated gpu power and performancemodel," in *Proc. of ISCA*, 2010.

[26] L. Arge, M. Goodrich, M. Nelson, and N. Sitchinava, "Fundamental parallel algorithms for private-cache chip multiprocessors," in *Proc. of SPAA*, Munich, Germany, 2008, pp. 235–246.

[27] H. Wong, "Demystifying GPU microarchitecture through microbenchmarking," in *Proc. of ISPASS*, 2010, pp. 235–246.

[28] O. Green, R. McColl, and D. A. Bader, "GPU merge path: a GPU merging algorithm," in *Proc. of ICS*, 2012, pp. 331–340.

[29] K. Kaczmarski, "Experimental B$^+$-tree for GPU," in *Proc. of ADBIS*, vol. 2, Rome, Italy, 2011, pp. 232–241.

[30] ——, "B-tree optimized for GPGPU," in *Proc. of OTM 2012*, Rome, Italy, 2012, pp. 843–854.

[31] A. Shekhar, "Parallel binary search trees for rapid IP lookup using graphic processors," in *Proc. of IMKE*, 2013, pp. 176–179.

[32] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. Nguyen, T. Kaldeway, V. Lee, S. Brandt, and P. Dubey, "FAST: fast architecture sensitive tree search on modern CPUs and GPUs," in *Proc. of SIGMOD*, Indianapolis, Indiana, USA, 2010.

[33] J. Soman, K. Kothapalli, and P. J. Narayanan, "Discrete range searching primitive for the GPU and its applications," *J. Exp. Algorithmics*, vol. 17, pp. 4.5:4.1–4.5:4.17, Oct. 2012.

[34] M. Tang, J. yi Zhao, R. feng Tong, and D. Manocha, "Gpu accelerated convex hull computation," in *Proc. of SMI*, 2012, pp. 498–506.

[35] A. Stein, E. Geva, and J. El-Sana, "Cudahull: Fast parallel 3d convex hull on the gpu," *Applications of Geometry Processing*, vol. 36, pp. 265–271, 2012.

[36] M. Qi, T.-T. Cao, and T.-S. Tan, "Computing 2d constrained delaunay triangulation using the gpu," *IEEE Transactions on Visualization and Computer Graphics*, vol. 19, pp. 736–748, 2012.

[37] G. Rong, Y. Lio, W. Wang, and X. Yin, "Gpu-assisted computation of centroidal voronoi tessellation," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, pp. 345–356, 2010.

[38] B. Catanzaro, A. Keller, and M. Garland, "A decomposition for in-place matrix transposition," in *Proc. of PPoPP*, 2014.

[39] D. Comer, "The ubiquitous b-tree," *ACM Computing Surveys*, vol. 11, pp. 121–137, 1979.

[40] T. D. Han and T. S. Abdelrahman, "Reducing branch divergence in gpu programs," in *Proc. of GPGPU-4*, 2011.

[41] M. A. Bender, E. D. Demaine, and M. Farach-Colton, "Cache-oblivious b-trees," *SIAM Journal on Computing*, vol. 35, pp. 341–358, 2005.

[42] M. A. Bender and H. Hu, "An adaptive packed-memory array," in *Proc. of SIGMOD*, 2006.

[43] J. JaJa, *An Introduction to Parallel Algorithms*. Reading, Mass.: Addison-Wesley Publishing Co., 1992.

[44] H. Edelsbrunner and M. H. Overmars, "On the equivalence of some rectangle problems," *Information Processing Letters*, vol. 14, pp. 124–127, 1982.

[45] M. J. Atallah and Y. Qi, "Computing all skyline probabilities for uncertain data," in *Proc. of PODS*, 2009, pp. 279–287.

[46] A. Okabe, B. Boots, K. Sugihara, and S. N. Chiu, *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*, 2nd ed. John Wiley & Sons, 2009.

[47] Q. Du, V. Faber, and M. Gunzburger, "Centroidal voronoi tessellations: Applications and algoritms," *SIAM Rev.*, vol. 41, pp. 637–676, 1999.