Analysis-driven Engineering of Comparison-based Sorting Algorithms on GPUs

Ben Karsin
University of Hawaii at Mānoa
USA
karsin@hawaii.edu

Volker Weichert
Goethe University Frankfurt
Germany
weichert@cs.uni-frankfurt.de

Henri Casanova
University of Hawaii at Mānoa
USA
henric@hawaii.edu

John Iacono
New York University
USA
iacono@nyu.edu

Nodari Sitchinava
University of Hawaii at Mānoa,
USA
nodari@hawaii.edu

ABSTRACT

We study the relationship between memory accesses, bank conflicts, thread multiplicity (also known as over-subscription) and instruction-level parallelism in comparison-based sorting algorithms for Graphics Processing Units (GPUs). We experimentally validate a proposed formula that relates these parameters with asymptotic analysis of the number of memory accesses by an algorithm. Using this formula we analyze and compare several GPU sorting algorithms, identifying key performance bottlenecks in each one of them. Based on this analysis we propose a GPU-efficient multiway mergesort algorithm, GPU-MMS, which minimizes or eliminates these bottlenecks and balances various limiting factors for specific hardware.

We realize an implementation of GPU-MMS and compare it to sorting algorithm implementations in state-of-the-art GPU libraries on three GPU architectures. Despite these library implementations being highly optimized, we find that GPU-MMS outperforms them by an average of 21% for random integer inputs and 14% for random key-value pairs.

This material is based upon work supported by the National Science Foundation under Grants CCF-1533823, CCF-1745331, CCF-1319648, CCF-1533564, CCF-0430849 and MRI-1229185, a Fulbright Fellowship and by the Fonds de la Recherche Scientifique-FNRS under Grant nÂř MISU F 6001 1.

ICS '18, June 12–15, 2018, Beijing, China
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5783-8/18/06…$15.00
https://doi.org/10.1145/3205289.3205298

KEYWORDS

GPU, sorting, mergesort, bank conflicts, I/O efficiency

ACM Reference Format:
Ben Karsin, Volker Weichert, Henri Casanova, John Iacono, and Nodari Sitchinava. 2018. Analysis-driven Engineering of Comparison-based Sorting Algorithms on GPUs. In *ICS '18: 2018 International Conference on Supercomputing, June 12–15, 2018, Beijing, China.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3205289.3205298

1 INTRODUCTION

Sorting is a primitive operation that is a building block for countless algorithms. It is therefore important to design and implement sorting algorithms that approach peak performance on a range of hardware architectures. Graphics Processing Units (GPUs) are particularly attractive architectures as they provide massive parallelism and computing power. However, designing and implementing algorithms that achieve peak performance on GPUs is a challenging task. Many factors can stifle GPU performance, including: sub-optimal memory access patterns, insufficient parallelism, overuse of scarce resources (e.g., shared memory), low *instruction-level parallelism* (ILP), and thread branch divergence. Many prior works have considered these factors individually and provided models and optimizations in an attempt to achieve peak GPU performance [1, 11, 16, 24, 25, 31, 41]. However, there is frequently a complex tradeoff between these performance factors and it is not clear which of them affect overall performance the most. For example, it is well-known that a large *multiplicity* [1] (number of threads scheduled per core) is required to hide memory latency. But excessive use of shared memory to avoid slow global memory accesses may reduce the number of threads scheduled per core due to lack of shared memory resources. As a result of these complex tradeoffs, developers frequently rely on heuristics and trial-and-error to achieve high performance

[1]Also known as *over-subscription.*

on GPUs. Instead, we propose an analytical approach to identify these tradeoffs, aiding in the development of new, GPU-efficient algorithms.

In this paper, we focus on the analysis and design of comparison-based sorting algorithms for GPUs. A natural question arises: why bother with comparison-based sorting algorithms? After all, all numbers on modern computers are stored in binary representation and one can map these representations to integers and use an integer sorting algorithm, such as radix sort, which is known to be faster in practice. To answer this question, consider the following problem, which is a natural problem in computational geometry: given a set of line segments defined by endpoints with integer coordinates, order them by their slopes. Storing the slopes as fixed precision floating point numbers can lead to rounding errors and, consequently, to erroneous result [4]. Instead, the slopes can be stored without any loss in precision as fractions, each represented by two integers: a numerator and a denominator. Then, the relative order of two fractions $\frac{a}{b}$ and $\frac{c}{d}$ can easily be determined without any loss in precision by comparing ad to bc. Observe that to use radix sort to sort these fractions, one would need to convert them into fractions with a common denominator, which can require up to $O(n)$ bits.

Comparison-based sorting is a thoroughly studied problem, with many implementations on GPUs, including the highly optimized MGPU [3] and Thrust [15] library implementations. Yet, even for such a comprehensively studied problem, the effects of the above factors and tradeoffs between them is not well-understood. Thus, we discuss these factors to better understand how to balance them when developing GPU-efficient sorting algorithms. We assume the reader is familiar with the aspects and terminology of the GPU architectures, including thread organization, memory hierarchy design and ideal data access patterns [33].

Bank conflicts. It is well-known [1, 21, 33] that when utilizing shared memory, an algorithm must be careful to avoid bank conflicts, as they negatively impact performance. Yet, sometimes completely eliminating bank conflicts requires an algorithm with higher complexity, and the increase in the number of operations can offsets the benefit of removing bank conflicts. Instead, practitioners use heuristics to reduce the number of bank conflicts by changing access patterns without drastically changing the algorithm. This rarely eliminates bank conflicts completely, and we show in Section 3.3 that even a small average number of conflicts due to randomly generated inputs can significantly impact performance. Furthermore, one can find inputs that undermine these heuristics and cause a large number of bank conflicts. In Section 6.2 we show that, without too much effort, we can generate *conflict-heavy* inputs that cause a state-of-the-art sorting library implementation to execute up to 20% slower due to bank conflicts.

Memory hierarchy utilization. The three levels of memory hierarchy on GPUs (global memory, shared memory, registers)[2] provide various access latencies and bandwidths. All else being equal, using a faster memory type will obviously result in a faster implementation. However, often the utilization of faster memory results in more complex algorithms and the tradeoff between different approaches is not as clear. For example, Leischner et al. [23] present a randomized *GPU samplesort* algorithm, which implements multiway quicksort. Extensive research on I/O-efficient algorithms predicts that such multiway sorting algorithms should fully utilize fast shared memory and reduce accesses to slow global memory. Despite these types of I/O-efficient algorithms performing well on CPUs, on the GPU the MGPU library implementation of simple two-way mergesort [3] is much faster than the multiway samplesort of Leischner et al. [23].

Bank conflict free algorithms. One of the arguments why samplesort of Leischner et al. [23] is not the fastest in practice could be attributed to large number of bank conflicts in shared memory. Therefore, Koike and Sadakane [21] developed a bank conflict free multi-way mergesort algorithm. Unfortunately, we could not obtain the source code of their implementation to compare it to MGPU experimentally. However, analysis of their results indicates that, despite superior utilization of shared memory and bank conflict avoidance, their multiway mergesort would still be slower than MGPU's two-way mergesort implementation.

Thread occupancy, multiplicity and ILP. Using I/O complexity analysis of the External Memory model [2], it can be shown that the multiway mergesort of Koike and Sadakane [21] achieves asymptotically optimal global memory accesses, while MGPU mergesort does not. However, the I/O complexity analysis ignores many other performance factors that are significant for GPUs. In Section 4.2 we determine that the data structures used by Koike and Sadakane require a large amount of shared memory, limiting *occupancy* and significantly reducing overall performance. This illustrates the importance of parallelism when designing GPU-efficient algorithms. In Section 3, we investigate the impact of parallelism on GPU performance, showing that both *multiplicity* and *ILP* are important GPU performance factors.

1.1 Our Contributions

In this paper, we present an analytical framework that we use to study a range of factors that impact the performance of GPU-efficient comparison-based sorting algorithms. In particular, we consider the relationship between multiplicity $\mathcal{X}$, instruction-level parallelism $\mathcal{I}$, bandwidth $\mathcal{B}$, and latency $\mathcal{L}$

[2]Modern GPUs contain other types of memory that we do not consider here. Caches are implemented in the shared and global memories that we do discuss, while texture and constant memories are more useful for graphics rendering than general purpose computations.

of global memory, shared memory, and registers on NVIDIA GPUs. We experimentally verify the accuracy of our framework and show a simple mathematical relationship between these parameters and how their tradeoffs affect algorithm execution times. For example, we show that on our hardware, multiplicity $\mathcal{X}$ and ILP $\mathcal{I}$ are fully interchangeable via a simple relationship: $\mathcal{I} \cdot \mathcal{X} \geq 8$.

We consider two state-of-the-art GPU-efficient sorting algorithms: modernGPU (MGPU) [3] pairwise mergesort and the multiway mergesort presented by Koike and Sadakane [21]. For each algorithm, we analytically identify performance bottlenecks. We then present GPU-MMS, a multiway mergesort algorithm that mitigates or avoids these bottlenecks. MGPU suffers from a sub-optimal number of global memory accesses, which we experimentally determine to account for up to 75% of overall runtime. GPU-MMS avoids this, achieving an asymptotically optimal number of global memory accesses at the cost of additional computation.[3] As a result, GPU-MMS achieves a balance between time spent computing and time spent accessing memory. We find that the primary bottleneck of Koike and Sadakane's multiway mergesort is a lack of sufficient parallelism. GPU-MMS mitigates this issue by increasing both multiplicity and ILP. We show empirically that GPU-MMS outperforms the fastest currently available comparison-based sorting algorithms by up to 32.7% on random integer inputs and 67.2% on worst-case input permutations of integers.

Outline. The rest of this paper is organized as follows. Section 2 reviews related work. Section 3 describes our micro-benchmarks. Section 4 presents our analysis of the existing state-of-the-art sorting algorithms, and Section 5 details our design and implementation of GPU-MMS. Section 6 presents experimental results. Finally, Section 7 concludes with a discussion of the significance of our results.

2 RELATED WORK

Over the past decade, a lot of research has focused on designing efficient algorithms to solve a range of classical problems on GPUs [9, 12, 17, 27, 36, 37, 39]. These works have introduced several optimization techniques, such as coalesced memory accesses [10, 11, 35], branch divergence elimination [19, 23], and bank conflict avoidance [1, 6, 9, 19]. Several empirical models for specific GPUs have been proposed that use micro-benchmarking [5, 41], and several fast GPU algorithms have been produced [10, 17, 39] via the use of benchmarks [40] and application of hardware-specific optimization techniques to existing algorithms. Several authors proposed abstract GPU performance models, attempting to capture salient features of the GPU architectures in a way that attempts to balance accuracy and simplicity [16, 22, 25, 30, 31]. Despite these efforts, to the best of our knowledge, no prior work analyzed the tradeoff relationship between *all* of the above performance parameters that we study here.

Sorting has been extensively studied over the past half-century, and here we only mention previous work that focuses on the GPUs [3, 8, 12, 23, 26, 29, 38]. According to a recent survey of several GPU libraries [29] the fastest currently-available sorting implementations include the CUB [26], modernGPU (MGPU) [3], and Thrust [15] libraries. CUB employs a GPU-optimized radix sort, which sorts based on the binary representation of elements. MGPU and Thrust use variations of mergesort (based on Green et al. [12]) While highly optimized, these mergesort implementations issue sub-optimal numbers of global memory accesses and incur shared memory bank conflicts. Leischner et al. [23] introduced *GPU samplesort*, discussed in Section 1. Their work was continued by Dehne et al. [8] with a deterministic version of the samplesort algorithm. The work of Afshani and Sitchinava [1] focuses on shared memory only and presents an algorithm that sorts small inputs in shared memory without bank conflicts. Koike and Sadakane [21] also present a GPU sorting algorithm that minimizes global memory accesses using a multiway mergesort that is also bank conflict-free (we analyze their algorithm Section 4.2).

3 GPU PERFORMANCE FACTORS

In this section, we use a series of micro-benchmarks to determine the performance profile of each level of the memory hierarchy (including registers). We use the results to analyze several sorting algorithms (Sections 4 and 5).

3.1 Experimental Methodology

We perform all experiments using three hardware platforms. All computations are performed on the graphics cards, and no attempt is made to use CPU compute resources. Execution times are measured as time spent computing on the GPU, while time to transfer data between the CPU and GPU is not included, as is customary in these types of experiments [12, 23]. Table 1 presents the hardware specifications of our three platforms, as well as the parameters derived via micro-benchmarks. On all platforms we use GCC 4.8.1 and CUDA 7.5, and all programs are compiled with the -O3 optimization flag. Performance metrics such as bank conflicts are obtained via the `nvprof` profiling tool [32], included in the CUDA 7.5 toolkit. Since running `nvprof` impacts performance, execution times are measured on separate runs using the *cudaEvent* timer. Unless otherwise noted, Each experiment is repeated ten times, and we report the mean values, showing min-max error bars when non-negligible.

[3]Finding a parallel sorting algorithm that is both I/O-efficient and work-efficient remains a difficult open problem.

3.2 Parallelism and Memory Performance

As discussed in Section 1, GPUs rely on *multiplicity* and *instruction-level parallelism* (ILP) to hide memory latencies and improve throughput. Formally, we say that the multiplicity $\mathcal{X}$ of an implementation is the number of threads running *per physical core* (i.e., with P cores, we have a total of $\mathcal{X} \cdot P$ threads), while the ILP, $\mathcal{I}$, is the average number of consecutive independent instructions. The practical impact of both multiplicity and ILP is to hide the latency of each memory access. For instance, if we consider a single core accessing a particular memory system $\mathcal{A}$ times, where each access has latency $\mathcal{L}$, the runtime would be $\mathcal{A} \cdot \frac{\mathcal{L}}{\mathcal{X} \cdot \mathcal{I}}$. We define the latencies, $\mathcal{L}_g$, $\mathcal{L}_s$, and $\mathcal{L}_r$ as the number of clock cycles required to access global memory, shared memory, and registers, respectively. By increasing multiplicity and/or ILP, we can decrease the time needed to perform these accesses until peak bandwidth is reached for the particular memory system. We denote the peak bandwidth of global memory, shared memory, and registers as $\mathcal{B}_g$, $\mathcal{B}_s$, and $\mathcal{B}_r$, respectively. We normalize bandwidth values to be the peak number of elements accessed *per clock cycle, per core*. Thus, with P processor cores, we estimate the total time to perform $\mathcal{A}_P$ *parallel* accesses to a memory system with latency $\mathcal{L}$ and bandwidth $\mathcal{B}$ to be:

$$T \approx \mathcal{A}_P \cdot max\left(\frac{1}{\mathcal{B}}, \left\lceil \frac{\mathcal{L}}{\mathcal{X} \cdot \mathcal{I}} \right\rceil\right).$$

To verify the above performance estimate, we perform a series of micro-benchmarks that measure the average time to access each type of memory while varying multiplicity ($\mathcal{X}$) and ILP ($\mathcal{I}$), on each of our platforms.

We measure the impact of multiplicity and ILP on global memory performance with a micro-benchmark that performs a simple copy in global memory. *Each thread* copies N elements from one array in global memory to another (also in global memory). Therefore, this micro-benchmark performs $2N$ memory accesses per thread, with $P\mathcal{X}$ total threads, or $\frac{2N \cdot \mathcal{X}P}{P} = 2N\mathcal{X}$ parallel memory accesses, so we expect it to run in $2N\mathcal{X} \cdot max\left(\frac{1}{\mathcal{B}_g}, \left\lceil \frac{\mathcal{L}_g}{\mathcal{X}\mathcal{I}} \right\rceil\right)$ clock cycles. We vary $\mathcal{X}$ by varying the number of threads and $\mathcal{I}$ by loading elements into registers before writing them into the destination array and observe their impact on memory access time.

Table 1 Specifications of three platforms, including parameters measured by benchmarks (denoted by a *) and hardware details provided by the manufacturer.

Parameter	Algoparc	Gibson	Uhhpc
NVIDIA GPU Model	M4000	GTX 770	K40m
Generation	Maxwell	Kepler	Kepler
Global Memory	8 GiB	4 GiB	12 GiB
Total Cores (P)	1664	1536	2880
Clock Rate	780 MHz	1046 MHz	745 MHz
Total smem (M)	1248 KiB	512 KiB	960 KiB
*$\mathcal{B}_g$ (elts/clock/core)	0.0301	0.0279	.0275
*$\mathcal{B}_s$ (elts/clock/core)	0.233	0.130	0.131
*$\mathcal{B}_r$ (elts/clock/core)	≈ 1	≈ 1	≈ 1
*Ł$_g$ (clocks)	269.5	267.6	291.2
*Ł$_s$ (clocks)	85.84	123.1	111.9
*Ł$_r$ (clocks)	6	10	10

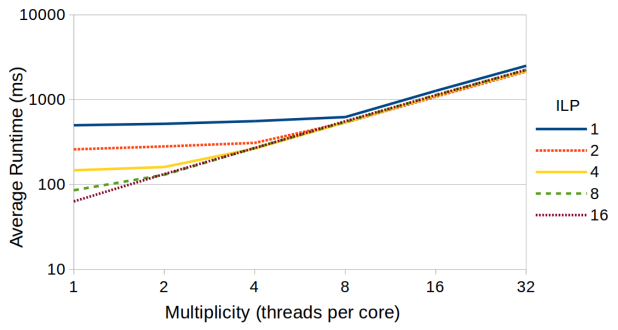

Figure 1 Average global memory micro-benchmark runtime vs. multiplicity ($\mathcal{X}$) for several ILP ($\mathcal{I}$) values. Results on Algoparc with $N = 2^{16}$ integers per thread. Once $\mathcal{X} \cdot \mathcal{I} = 8$, peak bandwidth is reached.

Figure 1 plots average execution time vs. multiplicity ($\mathcal{X}$) on Algoparc when each thread copies 2^{16} integer elements from one global memory array into another, for several ILP ($\mathcal{I}$) values. When $\mathcal{X} = 1$ and $\mathcal{I} = 1$, the average time per memory access, in clocks, is simply the latency, $\mathcal{L}_g$, and for larger amounts of ILP ($\mathcal{I}$), each access takes less time (i.e., $\frac{\mathcal{L}_g}{\mathcal{I}}$). As $\mathcal{X}$ grows, however, the total number of accesses grows as well, so constant execution time indicates reduced average time per access (i.e., $\frac{\mathcal{L}_g}{\mathcal{X}}$). Once $\mathcal{X} \cdot \mathcal{I}$ becomes large enough (~ 8 in Figure 1), the time spent per access becomes constant. At this point, increasing $\mathcal{I}$ has no impact and increasing $\mathcal{X}$ increases execution time linearly, since the number of accesses increases with $\mathcal{X}$. This corresponds to the point where $\frac{1}{\mathcal{B}_g} \geq \frac{\mathcal{L}_g}{\mathcal{X}\mathcal{I}}$, which is the inflection point in Figure 1 where execution time begins to increase. From this we estimate $\mathcal{B}_g = \frac{2N\mathcal{X}}{T}$, where T is the measured execution time. The latency and peak bandwidth for shared memory ($\mathcal{L}_s$ and $\mathcal{B}_s$), and registers ($\mathcal{L}_r$ and $\mathcal{B}_r$) are measured similarly. Table 1 gives $\mathcal{L}$ and $\mathcal{B}$ values obtained for the three levels of memory hierarchy on our three hardware platforms.

3.3 Bank Conflicts

As discussed in Section 1, shared memory must be accessed in a specific pattern to avoid *bank conflicts*. To illustrate the practical significance of bank conflicts, we consider the state-of-the-art MGPU mergesort [3] algorithm (for details see

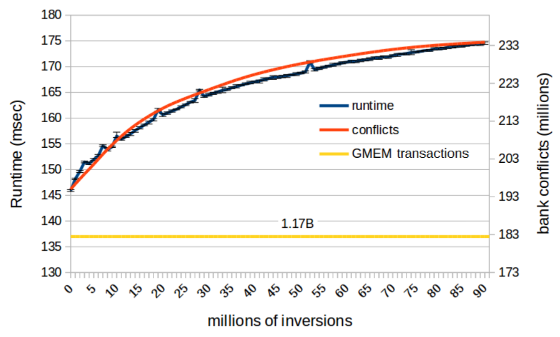

Figure 2 MGPU runtime and number of bank conflicts vs. input sortedness for $N = 10^8$. Global memory transactions are shown to illustrate that they are largely independent of input sortedness.

Section 4.1). Although MGPU mergesort uses a heuristic to reduce bank conflicts, its shared memory access pattern is *data-dependent*, so the number of bank conflicts is a function of the input. Figure 2 illustrates a strong correlation between the number of bank conflicts and overall runtime of MGPU mergesort. Results show that, on UHHPC, as we increase the level of unsortedness, i.e., the number of inversions, the number of bank conflicts (measured using an execution profiler) increases along with the overall execution time.

While we can measure bank conflicts and see that they impact GPU performance, theoretical analysis of bank conflicts on such data dependent access patterns is a challenging open problem. For this reason, we do not attempt to analyze bank conflicts theoretically. Instead, we let β denote an average number of bank conflicts that occur for a particular set of shared memory accesses, measure it empirically, and incorporate it into our performance estimate. Since β is typically only a function of the input and/or the algorithm, we can measure it on one architecture and use it to estimate the performance on a range of GPUs.

4 STATE-OF-THE-ART GPU SORTING

While a number of sorting algorithms have been proposed for GPU architectures [1, 3, 8, 15, 21, 23, 26, 28, 29], the CUB [26], Thrust [15], and MGPU [3] libraries currently provide the fastest sorting implementations. We focus our analysis on MGPU mergesort, because Thrust employs a similar comparison-based algorithm and CUB is not a comparison-based algorithm.

4.1 MGPU Mergesort

The modernGPU (MGPU) library [3] is a collection of highly optimized implementations of fundamental algorithms for modern GPU architectures. While MGPU is not frequently updated, many of its implementations remain the fastest available for NVIDIA GPUs. The sorting implementation included with MGPU is based on pairwise mergesort [7] with a high degree of parallelism to run efficiently on GPUs.

MGPU algorithm overview – The latest MGPU (version 2.10) mergesort takes an unsorted input of N elements and performs a series of merge rounds to generate a sorted output. MGPU assigns a fixed number E of elements to each thread, where E depends on the GPU hardware (either $E = 11$ or $E = 15$). Threads are grouped into thread-blocks of $t = 128$ threads each. The algorithm begins by having each thread sort E elements independently. Each thread-block then merges t lists in shared memory to obtain a sorted list of tE elements (we call this the *base case*). Pairs of lists are then merged in subsequent rounds, with increasing numbers of threads performing each merge. During each merge round, lists are partitioned using the Mergepath [13] method in both global and shared memory.

Execution Time Estimate – We omit the details of our analysis of the MGPU mergesort algorithm due to lack of space. For full details, we refer interested readers to Karsin's dissertation [18] that includes a thorough analysis of MGPU. Combining this analysis with the benchmark results from Section 3, we estimate the runtime of MGPU mergesort as:

$$\begin{aligned} T &\approx T_g + T_s \,, \\ T_g &\approx \frac{2N}{P}\left(\left\lceil \log \frac{N}{tE} \right\rceil + \frac{N \lceil \log (NtE) \rceil \lceil \log \frac{N}{2tE} \rceil}{4tE} + 1\right) max\left(\frac{1}{\mathcal{B}_g}, \frac{\mathcal{L}_g}{\mathcal{X}\mathcal{I}}\right) , \\ T_s &\approx \frac{2N}{P}\left(\beta_1 \left\lceil \log \left(\frac{N}{E}\right) \right\rceil + \beta_2 \frac{\lceil \log (tE) \rceil \lceil \log \left(\frac{N}{tE}\right) \rceil}{2E}\right) max\left(\frac{1}{\mathcal{B}_s}, \frac{\mathcal{L}_s}{\mathcal{X}\mathcal{I}}\right) , \end{aligned}$$

where T_g and T_s denotes the global and shared memory access times, respectively.

We determine $\beta_1 = 3.1$ and $\beta_2 = 2.2$ by using the nvprof tool [33] to empirically measure average numbers of bank conflicts. These are simply estimates obtained with random inputs and certain permutations may result in more or fewer bank conflicts. We investigate this further in Section 6.2 by generating inputs that result in many bank conflicts.

Since MGPU mergesort uses few registers, multiplicity $\mathcal{X}$ is limited only by shared memory usage. Each thread stores E elements in shared memory, and the GPU has a total shared memory of size M, thus $\mathcal{X} = \frac{M}{PE}$. ILP ($\mathcal{I}$) depends on optimizations and varies for each phase. MGPU employs global memory access optimizations proposed by Merrill and Grimshaw [27] that effectively doubles $\mathcal{I}$. Shared memory accesses are dependent, though additional accesses due to bank conflicts are automatically issued independently [33], so $\mathcal{I} = \beta_1$ or $\mathcal{I} = \beta_2$, depending on the kernel.

Combining the above analysis with the hardware parameters listed in Table 1, we predict the execution time of MGPU

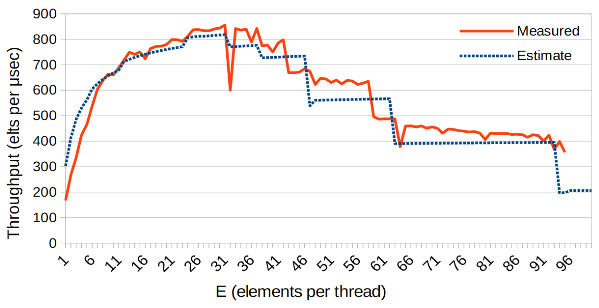

Figure 3 Estimated and measured MGPU mergesort throughput when varying E (elements per thread) on Algoparc, for $N = 100M$.

mergesort on our three hardware platforms, estimate optimal parameter values (e.g., E), and identify bottlenecks. Figure 3 shows our throughput estimate as E varies, on the Algoparc platform, along with the throughput measured when running MGPU. These results indicate that our estimate correctly predicts the best value for E to be 31, despite MGPU using $E = 15$ for Maxwell generation GPUs, like that on Algoparc. Our overall execution time estimate has an average relative error of 7.48%.

On all three platforms, our estimate indicates that between 65% and 75% of execution time is due to global memory accesses. The remaining portion of execution time is attributed to shared memory accesses. We measure β for merging and partition phases to be 3.1 and 2.2, respectively, indicating that bank conflicts contribute roughly 15-20% to overall execution time. We conclude that MGPU suffers from two primary performance bottlenecks: global memory bandwidth and shared memory bank conflicts.

4.2 Koike and Sadakane's Multiway Mergesort

Pairwise mergesort requires $O(\log_2 N)$ merge rounds, where N elements must be read from (and written to) global memory at each round, leading to the global memory bottleneck seen in our analysis of MGPU in the previous section. This naturally suggests the use of a *multiway* mergesort algorithm [20] to reduce the number of merge rounds. According to the (sequential) external memory model [2], multiway mergesort achieves optimal I/O complexity when $K = \frac{M}{B}$, where M is the size of internal memory and B is the access block size [20]. Koike and Sadakane [21] present a multiway mergesort for GPUs.

Algorithm Overview – The multiway mergesort of Koike and Sadakane [21] starts by sorting blocks of w elements in internal memory ("base case"). Groups of K sorted lists are then merged until the entire input is sorted. At each merge round, partitions are needed if there are not enough independent merge lists to satisfy all thread-blocks, where each thread-block contains $w = 32$ threads. Since groups of K lists are merged at each round, partitions are found across all K lists using a search method based on K binary searches proposed by Hayashi et al. [14].

The merging of K lists by w threads is accomplished with the use of a variation of a minHeap structure. This structure stores $2w$ elements in sorted order at each node, where all elements satisfy the heap property (every element in a node x is smaller than every element in its children y and z). This allows I/O-efficient reading and writing of blocks of w elements at a time while sorting.

Estimating performance – As with MGPU, we omit the details of our analysis of the multiway mergesort of Koike and Sadakane and refer interested readers to Karsin's dissertation [18] for the full analysis. We were unable to obtain an implementation of the algorithm presented by Koike and Sadakane [21], so we cannot accurately determine $\mathcal{I}$. We, therefore, assume $\mathcal{I} = 1$, since, while merging, each operation is dependent on the result of the previous operation. Each warp works on its own heap structure, requiring $4Kw - 2w$ elements in shared memory, limiting multiplicity to $\mathcal{X} = \frac{M}{4KP-2P}$. We estimate the execution time of the algorithm by Koike and Sadakane [21] as:

$$\begin{aligned} T &\approx T_g + T_s \,, \\ T_g &\approx \frac{2N}{P}\left(\log_K \frac{N}{w} + 1\right) max\left(\frac{1}{\mathcal{B}_g}, \left\lceil \frac{\mathcal{L}_g}{\frac{M}{4KP-2P}} \right\rceil\right) , \\ T_s &\approx \frac{4N}{P}\left(\log w \log \frac{N}{w} + \frac{\log^2 w}{2}\right) max\left(\frac{1}{\mathcal{B}_s}, \left\lceil \frac{\mathcal{L}_s}{\frac{M}{4KP-2P}} \right\rceil\right) . \end{aligned}$$

Our performance estimate indicates that, as K grows, the relative impact of shared memory accesses increases. When $K = 32$, more than 75% of the execution time is due to shared memory accesses. Furthermore, since multiplicity decreases with increased K, on Algoparc accesses to shared memory and global memory become latency-bound when $K \geq 3$ and $K \geq 6$, respectively.

5 GPU-MMS

The main bottlenecks for the MGPU mergesort are global memory bandwidth and shared memory bank conflicts. While the multiway mergesort proposed by Koike et al. [21] addresses these bottlenecks, its performance is limited by the large amount of shared memory it accesses and low multiplicity. In this section, we present GPU-MMS, our GPU-efficient multiway mergesort algorithm that avoids the performance bottlenecks of both of these algorithms.

5.1 Algorithm Overview

We use the algorithm by Koike and Sadakane [21] as a starting point, but present several improvements that address its performance bottlenecks. In a nutshell, GPU-MMS maximizes multiplicity and ILP while reducing both shared memory and global memory accesses. We present the following improvements over the algorithm of Koike and Sadakane [21].

Parallel heap – We first focus our design on reducing the shared memory usage to increase $\mathcal{X}$ and thereby improve performance for larger values of K. Recall that the heap used by Koike and Sadakane stores $2w$ elements at each node, requiring $4Kw - 2w$ elements in shared memory per heap. We present an improved heap structure, which we call a *minBlockHeap*, that requires half the shared memory, while still allowing all w threads to work cooperatively and access only blocks of w elements at a time from global memory.

Each node x contains a list of w elements $(x[0], \ldots, x[w-1])$, stored in sorted order, and we define the *fillEmptyNode* operation that fills an empty node (i.e., a node without any elements in its list). Consider x to be an empty node with non-empty children y and z. W.l.o.g. assume that $y[w-1] > z[w-1]$. The $fillEmptyNode(x)$ operation is performed as follows: merge the lists of y and z, fill x with the w smallest elements, fill y with the w largest elements, and set z as empty. Since, prior to merging, y had the largest element $(y[w-1])$, its new largest element has not changed and the heap property holds for y. We continue down the tree by calling $fillEmptyNode(z)$ until we reach a leaf, which we fill by loading w new elements from global memory. This structure provides two benefits over the heap used in [21]: (i) the shared memory required to store each heap is reduced to $2Kw - w$ elements and (ii) the number of elements merged is reduced from $4w$ and $2w$ at each level. We note that, a small K value results in a smaller heap, increasing K reduces the total number of merge rounds. Thus K provides a tradeoff between shared memory usage and global memory accesses.

Merging in registers – A primary drawback of using a minBlockHeap type structure is that, since w threads are merging pairs of w elements, a work-efficient sequential merge cannot be used. Koike and Sadakane [21] employ a bank conflict-free bitonic merge network [20] in shared memory, increasing shared memory accesses by a factor $\log w$. To reduce the cost of this work-inefficient merge, we develop a merge step for GPU-MMS that operates in *registers*. While this causes GPU-MMS to be work-inefficient, the extra work is done in low-latency registers that have much higher peak bandwidth. We accomplish this register merge with the use of `__shfl()`, a hardware instruction that lets threads within a warp access each others' registers. This enables us to reduce the number of shared memory accesses in favor of faster register operations.

Independent merging – In addition to increasing $\mathcal{X}$ we increase $\mathcal{I}$ with the following optimization to the minBlockHeap. After we extract the smallest w elements from the heap, all w threads working on the heap identify the path of merge nodes from the root to the leaf. This path corresponds to the node at each level of the heap that will be emptied during merging. Starting at the root, each thread loads one element from each child into registers, and continues down the path of the child with the smallest $(w-1)$-th element. There are total of $\log K$ nodes along this path, so each thread loads a total of $2\log K$ elements into registers. Since each merge operation is independent, we interleave the operations, thereby increasing $\mathcal{I}$ by a factor $\log K$. This comes at the cost of using additional registers. However, this is not an issue on our hardware, so shared memory remains the primary factor limiting multiplicity.

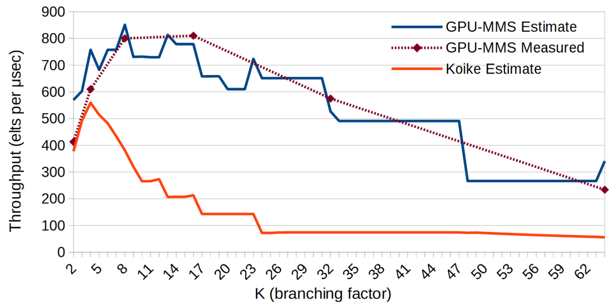

Figure 4 Estimated throughput of GPU-MMS, compared with the measured performance on ALGOPARC with $N = 2^{28}$ for a range of K values. Estimated throughput also shown for Koike and Sadakane's multiway mergesort [21].

Increased base case – MGPU mergesort sorts its base case of tE elements using pairwise mergesort, though this results in bank conflicts, reducing performance. Koike and Sadakane [21] avoid bank conflicts in the base case by letting each thread sort w elements within its own memory bank, resulting in smaller base case. We increase the base case by sorting w^2 items in shared memory by using the bank conflict-free algorithm of Afshani and Sitchinava [1].

5.2 Performance Analysis

Aside from reducing shared memory accesses by a factor $\log w$, GPU-MMS does not significantly improve asymptotic performance over [21]. However, GPU-MMS improves many of the constant factors that have a practical impact on execution time. Though we omit details due to limited space, our analysis of accesses to each type of memory results in:

$$\begin{aligned}
T &\approx T_g + T_s + T_r\,, \\
T_g &\approx \frac{2N}{P}\left(\left\lceil \log_K \frac{N}{w^2}\right\rceil + 1\right) max\left(\frac{1}{\mathcal{B}_g}, \left\lceil\frac{\mathcal{L}_g}{\mathcal{X}\mathcal{I}}\right\rceil\right), \\
T_s &\approx \frac{4N}{P}\left(\left\lceil \log \frac{N}{w^2}\right\rceil + \log w\right) max\left(\frac{1}{\mathcal{B}_g}, \left\lceil\frac{\mathcal{L}_g}{\mathcal{X}\mathcal{I}}\right\rceil\right), \\
T_r &\approx \frac{6N}{P}\left(\left\lceil \log \frac{N}{w^2}\right\rceil \log w + \log^3 w\right) max\left(\frac{1}{\mathcal{B}_g}, \left\lceil\frac{\mathcal{L}_g}{\mathcal{X}\mathcal{I}}\right\rceil\right),
\end{aligned}$$

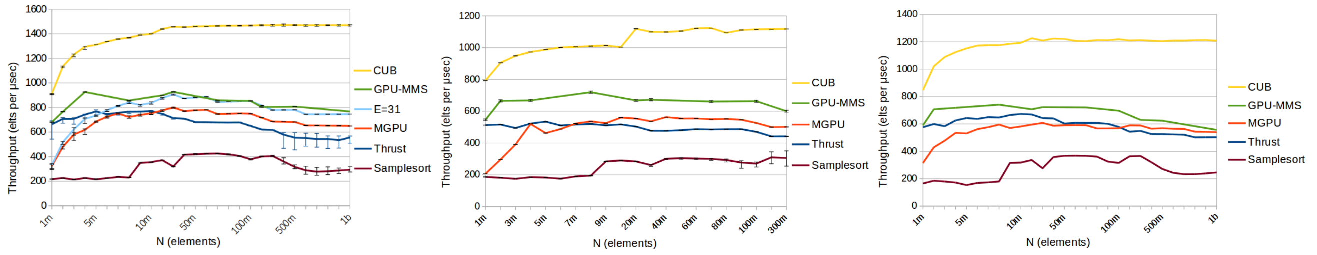

Figure 5 Comparison of average throughput for each sorting algorithm (CUB radix sort [26], Thrust [15] and MGPU [3] pairwise mergesort, GPU samplesort [8], and our GPU-MMS multiway mergesort) on inputs of random integers on Algoparc (left), Gibson (middle), and Uhhpc (right). The $E = 31$ plot refers to MGPU while setting $E = 31$, rather than the standard $E = 15$.

where T_r is the time spent performing computations in registers. Note that we require $6N$ register accesses per merge round due to the merge network requiring 3 operations: `__shfl()`, `min()`, and `max()`. Furthermore, the use of a merging network means that GPU-MMS is not work-efficient, as merging w elements takes $O(w \log w)$ work. Note that the w^2 is due to the increased size of the base case. Since our heap uses less memory, multiplicity is increased to $\mathcal{X} = \frac{M}{2KP-P}$. Furthermore, independent merging increases $\mathcal{I}$ to $\log K$.

Figure 4 shows our performance estimate and measured execution time of our GPU-MMS implementation, on Algoparc, for $N = 2^{28}$ and varying values of K. We also show a performance estimate for the algorithm of Koike and Sadakane [21] to illustrate the impact of the improvements included in GPU-MMS. Results indicate that these improvements significantly increase overall performance, especially when K is larger. However, even with the increased $\mathcal{X}$ and $\mathcal{I}$, the ideal value of K is still either 8 or 16 on Algoparc. On our other two hardware platforms, both our analysis and empirical results indicate that $K = 8$ leads to the best performance. Thus, in all that follows we use $K = 16$, $K = 8$, and $K = 8$ on Algoparc, Gibson, and Uhhpc, respectively, unless otherwise noted. To verify that our GPU-MMS algorithm does not suffer from the performance bottlenecks that we see with MGPU mergesort and Koike and Sadakane's multiway mergesort [21], we consider the estimated percentage of overall runtime due to each component of execution. Our performance estimate indicates that, unlike MGPU mergesort and Koike and Sadakane's mergesort, no single type of operation dominates execution time. Furthermore, register operations make up a significant portion of overall execution time (35%), indicating that GPU-MMS performance is not bound by high-latency memory accesses.

6 EXPERIMENTAL COMPARISONS

The analysis and results in the previous section indicate that our GPU-MMS algorithm provides key advantages over the Thrust and MGPU pairwise mergesorts, as well as the multiway mergesort presented by Koike and Sadakane [21]. While we were unable to get code for the algorithm in [21], the improvements outlined in the previous section and analytical results illustrated in Figure 4 indicate that GPU-MMS should outperform [21] in all cases. We compare GPU-MMS performance with three leading GPU sorting libraries: Thrust 1.8.1 [15], MGPU 2.10 [3], and CUB 1.6.4 [26]. As discussed in Section 4, Thrust and MGPU provide two of the fastest comparison-based sorts available for GPUs, while CUB provides the fastest radix sort. Although CUB is not a comparison-based sort (limitations discussed in Section 1), we include it in some of our experiments for completeness. We also include the I/O-efficient samplesort implementation of [23] in some of our experiments.

6.1 Sorting Random Integers

Figure 5 shows the average throughput achieved by each algorithm when applied to random inputs of 4-byte integers, on each of our hardware platforms. These results show that GPU-MMS outperforms all other comparison-based sorting algorithms for all input sizes. As expected, CUB, being a radix sort, achieves much higher throughput across all input sizes. On Algoparc, GPU-MMS outperforms MGPU an average of 32.27%, across all input sets. Even when using the improved value of $E = 31$ for MGPU (as determined by our analysis in Section 4.1), GPU-MMS still outperforms MGPU for most input sets. On Gibson and Uhhpc, GPU-MMS is 23.26% and 11.48% faster than MGPU, respectively. We note that, unlike on Algoparc, for Gibson and Uhhpc, MGPU performs best when using the hard-coded $E = 11$ value.

6.2 Impact of Bank Conflicts

A feature of GPU-MMS is that, regardless of input, it is free of shared memory bank conflicts. MGPU and Thrust, however, have memory access patterns that depend on the input and may therefore result in bank conflicts. Since the memory

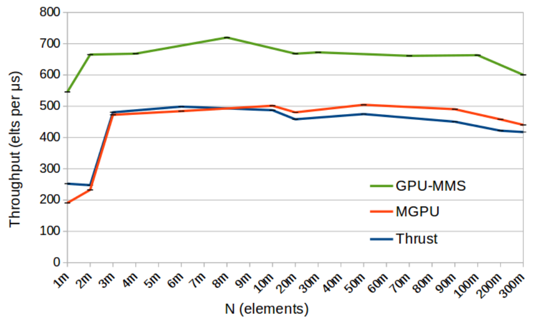

Figure 6 Average throughput vs. input size for *conflict-heavy* input on the GIBSON platform.

access patterns of MGPU and Thrust are deterministic, by carefully analyzing the access patterns of the algorithm, we generate a *conflict-heavy* input permutation that will cause these algorithms to incur large numbers of bank conflicts, for a given E. Since the optimal values of E may differ, we create conflict-heavy inputs specifically for each value of E (11 and 15). Figure 6 shows the average throughput achieved when sorting a conflict-heavy input set on the GIBSON platform. These results indicate that both Thrust and MGPU suffer from significant performance degradation due to bank conflicts on the conflict-heavy input, while GPU-MMS achieves performance similar to a random input sequence. On these conflict-heavy inputs, GPU-MMS is, on average, 71.1%, 68.3%, and 54.6% faster than MGPU (67.2%, 63.4%, and 49.5% faster than Thrust) on ALGOPARC, GIBSON, and UHHPC, respectively.

6.3 Sorting Other Data Types

As seen from Figure 5, the CUB radix sort implementation significantly outperforms all comparison-based sorting algorithms, including GPU-MMS. However, as discussed in Section 1, it is not always possible to use radix sort efficiently. In this section we present an experimental comparison of MGPU and GPU-MMS when sorting more complex data types using user-defined comparison functions. We consider sorting (a) key-value pairs, (b) points in the plane using l_1 norm (Manhattan distance) from the origin, and (c) rational numbers. Each of the three data types are stored as objects of two 32-bit integers x and y. The relative order of two objects (x_1, y_1) and (x_2, y_2) is resolved according to the outcome of comparing (a) x_1 and x_2, (b) $|x_1| + |y_1|$ and $|x_2| + |y_2|$, and (c) $x_1 \cdot y_2$ and $x_2 \cdot y_2$.

There are two factors that affect the performance of GPU-MMS on these data types. First, recall that GPU-MMS minimizes accesses to global memory by increasing utilization of shared memory, which is parameterized by K – the number of sequences being merged simultaneously. By increasing the size of the data types from a single integer to pairs of integers (or 32-bit floating point numbers), we effectively

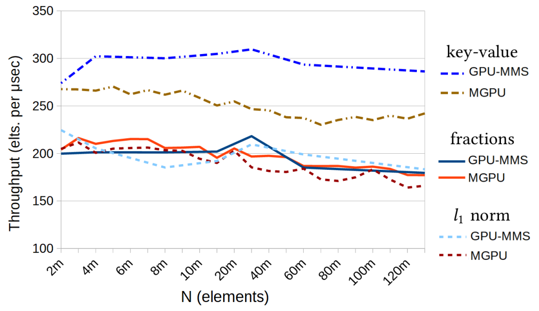

Figure 7 Average throughput vs. input size when sorting: key-value pairs, (x,y) coordinates by their l_1 distance from origin, and rational numbers, represented as integer numerator and denominator pairs.

reduce the shared memory size by a factor of 2. Thus, larger data types force GPU-MMS to use a smaller value of K to maintain adequate multiplicity. This increases the number of merge rounds and potentially degrades GPU-MMS's performance. MGPU, on the other hand, uses a small amount of shared memory and must perform a suboptimal $O(\log N)$ number of merge rounds, regardless of the size of the data types. Second, recall that, while MGPU's performance is primarily bounded by global memory accesses (Section 4.1), GPU-MMS is nearly-perfectly balanced between computations and global memory accesses (Section 5.2). As a result, the performance impact of increasing the cost of each comparison will be more apparent for GPU-MMS than MGPU.

Figure 7 plots the average throughput of GPU-MMS and MGPU when sorting random inputs of each of the datatypes we consider, on GIBSON. When sorting key-value pairs, despite the increased memory requirement (and decrease in K) we observed that GPU-MMS still outperforms MGPU on all input sizes by an average of 15.8%, 13.5%, and 12.3% on ALGOPARC, GIBSON, and UHHPC, respectively. We note that, since the value can be defined as a pointer to an arbitrary object, this experiment encompasses values of any data type. When sorting by the l_1 norm, GPU-MMS is on average, 6.6% and 1.5% faster, and 4.6% slower than MGPU on GIBSON, UHHPC, and ALGOPARC, respectively. When sorting fractions, MGPU outperforms GPU-MMS by an average of just 0.6%, 3.1%, and 5.4% on GIBSON, UHHPC, and ALGOPARC, respectively. As expected, since MGPU is memory-bound, the increased computation has less of an impact on MGPU's overall performance. Nevertheless, GPU-MMS remains competitive. This performance loss due to increased computation during a comparison confirms that GPU-MMS is not memory bound and makes more balanced use of the massive compute capabilities of modern GPUs.

7 CONCLUSIONS

This work demonstrates that even state-of-the-art GPU-efficient algorithm may suffer from avoidable performance bottlenecks. Analytical frameworks and theoretical models let us identify these bottlenecks and develop new, more effective algorithms. Furthermore, these models let us study how hardware changes would impact runtimes. Our analysis shows that a larger shared memory size would increase the relative performance improvement of GPU-MMS over MGPU. The latest NVIDIA architectures provide larger shared memory sizes than their predecessors. Thus, we expect GPU-MMS to outperform MGPU by larger margins on future systems.

While we focus on comparison-based sorting algorithms, the analytical techniques presented in this work are general enough that they may be useful for developing other GPU-efficient algorithms. In particular, the relationship between latency, bandwidth, and parallelism are key to achieving peak GPU performance, regardless of the algorithm itself. Along these lines, we have continued this work and applied this analytical framework to the problem of general matrix-matrix multiplication and determined that the currently available library implementation [34] achieves optimal performance on our GPU platforms [18].

REFERENCES

[1] Peyman Afshani and Nodari Sitchinava. Sorting and permuting without bank conflicts on GPUs. In *Proc. of ESA*, pages 13–25, 2015.

[2] A. Aggarwal and J.S. Vitter. The input/output coplexity of sorting and related problems. *Commun. ACM*, 31(11), 1988.

[3] S. Baxter. Modern GPU. URL: http://nvlabs.github.io/moderngpu/.

[4] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, Santa Clara, CA, USA, 3rd ed. edition, 2008.

[5] N. Bombieri, F. Busato, and F. Fummi. A fine-grained performance model for GPU architectures. In *Proc. of Design, Automation & Test in Europe*, pages 1267–1272, 2016.

[6] Bryan Catanzaro, Alexander Keller, and Michael Garland. A decomposition for in-place matrix transposition. In *Proc. of PPoPP*, 2014.

[7] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill, 2nd edition, 2001.

[8] Frank Dehne and Hamidreza Zaboli. Deterministic sample sort for GPUs. *CoRR*, abs/1002.4464, 2010.

[9] Y. Dotsenko, N. K. Govindaraju, P. Sloan, C. Boyd, and J. Manfedelli. Fast scan algorithms on graphics processors. In *Proc. of ICS*, pages 205–213, 2008.

[10] P. Enfedaque, F. Auli-Llinas, and J.C. Moure. Implementation of the DWT in a GPU through a register-based strategy. *IEEE Trans. PDS*, 26(12):3394–3406, 2015.

[11] N. Fauzia, L. N. Pouchet, and P. Sadayappan. Characterizing and enhancing global memory data coalescing on GPUs. In *Proc. of CGO*, pages 12–22, 2015.

[12] Oded Green, Robert McColl, and David A. Bader. GPU merge path: a GPU merging algorithm. In *Proc. of ICS*, pages 331–340, 2012.

[13] Oded Green, Saher Odeh, and Yitzhak Birk. Merge path - A visually intuitive approach to parallel merging. *CoRR*, abs/1406.2628, 2014.

[14] Tatsuya Hayashi, Koji Nakano, and Stephan Olariu. Weighted and unweighted selection algorithms for k sorted sequences. In *Proc. of International Symp. on Algorithms and Computation*, pages 52–61, 1997.

[15] Jared Hoberock and Nathan Bell. Thrust: A parallel template library, 2010. Version 1.7.0. URL: http://thrust.github.io/.

[16] S. Hong and H. Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *Proc. of ISCA*, pages 152–153, 2009.

[17] K. Kaczmarski. Experimental B^+-tree for GPU. In *Proc. of ADBIS*, volume 2, pages 232–241, Rome, Italy, 2011.

[18] Ben Karsin. *A performance model for GPU architectures: analysis and design of fundamental algorithms*. PhD thesis, University of Hawaii, 2018.

[19] Ben Karsin, Henri Casanova, and Nodari Sitchinava. Efficient batched predecessor search in shared memory on GPUs. In *Proc. of HiPC*, pages 335–344, 2015.

[20] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Publishing Co., Inc., 1998.

[21] A. Koike and K. Sadakane. A novel computational model for GPUs with applications to efficient algorithms. *International Journal of Networking and Computing*, 5(1):26–60, 2015.

[22] K. Kothapalli, R. Mukherjee, S. Rehman, S. Patidar, P. Narayanan, and K. Srinathan. A performance prediction model for the CUDA GPGPU. In *Proc. of HiPC*, 2009.

[23] N. Leischner, V. Osipov, and P. Sanders. GPU sample sort. In *Proc. of IPDPS*, pages 1–10, April 2010.

[24] L. Ma, R.D. Chamberlain, and K. Agarwal. Performance modeling for highly-threaded many-core GPUs. In *Proc. of ASAP*, 2014.

[25] Lin Ma, Kunal Agrawal, and Roger D. Chamberlain. A memory access model for highly-threaded many-core architectures. *Future Generation Computer Systems*, 30:202–115, 2014.

[26] D. Merrill. CUB: CUDA Unbound. URL: http://nvlabs.github.io/cub/.

[27] Duane Merrill and Andrew Grimshaw. Parallel Scan for Stream Architectures. Technical Report CS2009-14, Department of Computer Science, University of Virginia, 2009.

[28] Duane G. Merrill and Andrew S. Grimshaw. Revisiting sorting for GPGPU stream architectures. In *Proc. of PACT*, pages 545–546, 2010.

[29] Bruce Merry. A performance comparison of sort and scan libraries for GPUs. *Parallel Processing Letters*, 4, 2016.

[30] K. Nakano. Simple memory machine models for GPUs. In *Proc. of IPDPSW*, pages 794–803, 2012.

[31] K. Nakano. The hierarchical memory machine model for GPUs. In *Proc. of IPDPSW*, pages 591–600, 2013.

[32] NVIDIA. Nsight, 2015. URL: http://nvidia.com/object/nsight.html.

[33] NVIDIA. CUDA guide 9.0, 2017. URL: http://docs.nvidia.com/cuda.

[34] NVIDIA. CUDA cuBLAS library, 2018. URL: http://docs.nvidia.com/cuda/cublas/.

[35] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proc. of PPoPP*, pages 73–82. ACM, 2008.

[36] Shubhabrata Sengupta, Mark Harris, and Michael Garland. Efficient parallel scan algorithms for GPUs. NVIDIA Technical Report, 2008.

[37] A. Shekhar. Parallel binary search trees for rapid IP lookup using graphic processors. In *Proc. of IMKE*, pages 176–179, 2013.

[38] Nodari Sitchinava and Volker Weichert. Provably efficient GPU algorithms. *CoRR*, abs/1306.5076, 2013.

[39] Jyothish Soman, Kishore Kothapalli, and P. J. Narayanan. Discrete range searching primitive for the GPU and its applications. *J. Exp. Algorithmics*, 17:4.5:4.1–4.5:4.17, 2012.

[40] H. Wong. Demystifying GPU microarchitecture through microbenchmarking. In *Proc. of ISPASS*, pages 235–246, 2010.

[41] Y. Zhang and John D. Owens. A quantitative performance analysis model for GPU architectures. In *Proc. of HPCA*, pages 382–393, 2011.