arXiv:1702.07961v1 [cs.DS] 26 Feb 2017

An Efficient Multiway Mergesort for GPU Architectures

Henri Casanova*[1], John Iacono†[2], Ben Karsin‡[1], Nodari Sitchinava§[1] and Volker Weichert¶[3]

[1]University of Hawaii at Mānoa, USA
[2]New York University, USA
[3]University of Frankfurt, Germany

February 28, 2017

Abstract

Sorting is a primitive operation that is a building block for countless algorithms. As such, it is important to design sorting algorithms that approach peak performance on a range of hardware architectures. Graphics Processing Units (GPUs) are particularly attractive architectures as they provides massive parallelism and computing power. However, the intricacies of their compute and memory hierarchies make designing GPU-efficient algorithms challenging. In this work we present *GPU Multiway Mergesort* (MMS), a new GPU-efficient multiway mergesort algorithm. MMS employs a new partitioning technique that exposes the parallelism needed by modern GPU architectures. To the best of our knowledge, MMS is the first sorting algorithm for the GPU that is asymptotically optimal in terms of global memory accesses and that is completely free of shared memory bank conflicts.

We realize an initial implementation of MMS, evaluate its performance on three modern GPU architectures, and compare it to competitive implementations available in state-of-the-art GPU libraries. Despite these implementations being highly optimized, MMS compares favorably, achieving performance improvements for most random inputs. Furthermore, unlike MMS, state-of-the-art algorithms are susceptible to bank conflicts. We find that for certain inputs that cause these algorithms to incur large numbers of bank conflicts, MMS can achieve a 33.7% performance improvement over its fastest competitor. Overall, even though its current implementation is not fully optimized, due to its efficient use of the memory hierarchy, MMS outperforms the fastest comparison-based sorting implementations available to date.

*henric@hawaii.edu
†iacono@nyu.edu
‡karsin@hawaii.edu
§nodari@hawaii.edu
¶weichert@cs.uni-frankfurt.de

1 Introduction

Sorting is a fundamental primitive operation. Consequently, much effort has been devoted to developing efficient algorithms and their implementations on a wide range of hardware architectures, and in particular on the Graphics Processing Units (GPUs) that have become mainstream for High Performance Computing. A key challenge when designing GPU algorithms is exploiting the memory hierarchy efficiently [22, 11]. It is well-known that certain patterns when accessing data stored in *global memory* result in *coalesced* memory accesses, leading to greatly increased memory throughput [12]. Access patterns are also important when accessing the faster *shared memory*: if multiple threads attempt to access elements in the same *shared memory bank*, a *bank conflict* occurs and accesses are then serialized. Bank conflicts can lead to significant performance degradation, which is often overlooked when designing algorithms for GPUs [24, 13]. To illustrate the impact of bank conflicts on performance, Figure 1 shows the runtime of the current mergesort implementation in the modernGPU (MGPU) library [5], along with the number of bank conflicts as reported by an execution profiler, when sorting 10^8 4-byte integers on a K40 “Kepler” GPU [31]. Results shown are averaged over 10 trials on different input sets of varying levels of “sortedness.” All input sets are created using a sorted list of unique items and applying some number of inversions between random pairs. The x-axis corresponds to the number of such inversions, indicating the level of sortedness of the input list. The results indicate that the runtime of MGPU mergesort increases as the input list becomes less sorted. Furthermore, the number of bank conflicts closely tracks the runtime, which suggests that bank conflicts are performance drivers. Despite the potential performance loss due to bank conflicts, however, MGPU mergesort remains among the best-performing comparison-based sorting implementations available for GPUs [27].

The number of global memory accesses has been shown to impact performance for a range of applications [12, 34, 10] and, while many algorithms employ efficient access patterns, they fail to minimize the total number of accesses. Ideal global memory access patterns allow blocks of B elements to be retrieved in a single accesses. This pattern closely matches that of external disks, and if we equate global memory accesses to input/output operations (I/Os), we can analyze our algorithm using the Parallel External Memory (PEM) model [4]. Therefore, we can use the PEM model to design I/O efficient algorithms for the GPU that achieve an optimally minimal number of global memory accesses.

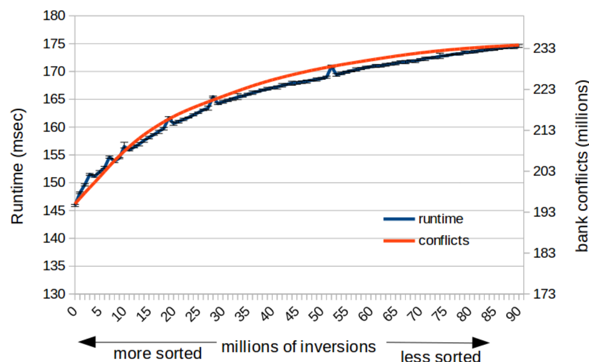


Figure 1: MGPU runtime (in ms) and number of bank conflicts vs. input sortedness when sorting 10^8 4-byte integers.

In this work, we develop a new variant of the multiway mergesort (MMS) algorithm for the GPU that (i) is bank conflict-free, (ii) achieves an asymptotically optimal number of global memory accesses, and (iii) leverages the massive parallelism available on modern GPUs. To date, one can argue that there is no true consensus regarding the use of theoretical models of computation for designing efficient GPU algorithms. As a result, most GPU algorithms are

designed empirically. By contrast, in this work we perform detailed asymptotic analysis of I/O complexity using the PEM model. Analysis shows that our MMS algorithm is more I/O-efficient than the standard pairwise mergesort approach used in state-of-the-art GPU libraries. We develop an initial implementation of our algorithm, and show via experiments in three different GPU platforms that our implementation is competitive with and often outperforms the fastest available comparison-based sorting GPU implementations. More specifically, this work makes the following contributions:

- We propose the first, to the best of our knowledge, I/O efficient and bank conflict-free (parallel multi-merge) sorting GPU algorithm;
- We show via theoretical performance analysis that this algorithm asymptotically outperforms the pairwise mergesort approaches currently used in state-of-the-art GPU libraries;
- We show experimentally that, when sorting large random input, an implementation of this algorithm is competitive with highly optimized GPU libraries (MGPU [5] and Thrust [17]) as well as a previously proposed I/O-efficient parallel sample sort implementation [24];
- We also show experimentally that, because our algorithm's runtime does not depend on the sortedness of the input due to it being bank conflict free, for some input it can achieve peak sorting throughput up to 35.2% higher than the fastest available GPU libraries.

The rest of this paper is organized as follows. Section 2 provides background information on the PEM model and GPUs. Section 3 reviews related work. Section 4 describes our proposed algorithm. Section 5 provides comparative theoretical performance analyses. Section 6 presents experimental results. Section 7 concludes with a brief summary of results and perspectives on future work.

2 Background

In this section we first review the model we use to analyze the I/O-complexity of our algorithm and its competitors, and then provide relevant information on GPU architecture and the programming model.

2.1 Parallel External Memory Model

The external memory model [3] is a well-known model analyzing the performance of algorithms that run on a computer with both a fast internal memory and a slow external memory, and whose performance is bound by the latency for slow memory access, or I/O. The Parallel External Memory (PEM) model [4] extends this model by allowing multiple processors to accesses external memory in parallel. In this work we use the PEM model to design algorithms that efficiently use the global memory system of modern GPU architectures. The PEM model relies on the following problem/hardware-specific parameters:

- N: problem input size,
- P: number of processors,
- M: internal memory size, and
- B: block size (the number of elements read or written during one external memory access).

In the PEM model, algorithm complexity is measured by the total number of I/Os. For example, scanning an input in parallel would cost $O(\frac{N}{PB})$ I/Os. Since this work is primarily

concerned with sorting, we note that the lower bound on number of I/Os to sort an input of size N is [15]:

$$\text{sort}_{\text{PEM}}(N) = \Omega\left(\frac{N}{PB}\log_{\frac{M}{B}}\frac{N}{B} + \log N\right)$$

2.2 GPU Overview

The PEM model was designed for multi-core systems with caches and RAM. Modern GPU architectures, however, contain complex compute and memory hierarchies (illustrated in Figure 2). To design efficient GPU algorithms, one must consider each level of these hierarchies.

The compute hierarchy of most modern GPUs is designed to accommodate thousands of compute threads [30]. Physically, a GPU contains a number of *streaming multiprocessors* (SMs), each of which contains: instruction units, a shared memory cache, and hundreds of compute cores. Logically, we consider the following computational units for GPUs:

- threads: single threads of execution,
- warps: groups of W threads that execute in SIMT (Single Instruction Multiple Threads) lockstep fashion [32] ($W = 32$ for most GPUs), and
- thread blocks: groups of one or more warps that execute on the same SM.

In addition to the compute hierarchy, modern GPUs employ a user-controlled memory hierarchy [11]. Figure 2 illustrates a high-level view of the typical GPU memory hierarchy.

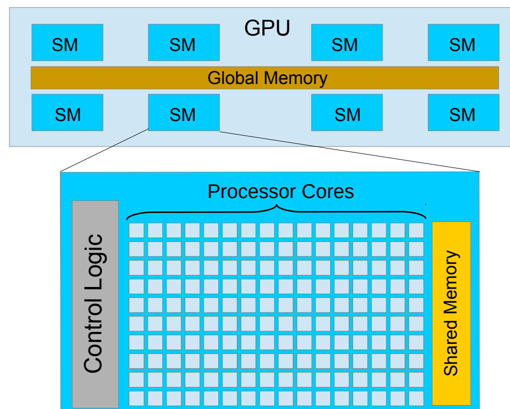


Figure 2: Illustration of a modern GPU architecture.

The largest component of the GPU memory hierarchy, *global* memory, is accessible by every thread executing on the GPU. To achieve high memory throughput, all threads within a *warp* must access together consecutive elements in global memory. This is called a *coalesced* memory access, allowing a warp to reference W elements in a single access. If we consider a warp to be a single unit of execution, we say that global memory accesses are performed in *blocks* [32], where $B = W$ elements are read from a single access, as with the PEM model discussed in Section 2.1. Therefore, we can apply the PEM model by equating global memory accesses as I/Os.

The smaller, fast *shared* memory of the GPU is shared among threads within a thread block. All threads within a warp can each access shared memory locations in parallel. However, as mentioned in Section 1, shared memory is organized in memory banks. If multiple threads

within a warp attempt to access the *same* memory bank, a *bank conflict* occurs and memory access are serialized [32]. Therefore, if all threads within a warp attempt to access the same memory bank, a W-way bank conflict occurs, effectively reducing shared memory throughput by a factor of W.

Finally, each thread has access to a set of fast private registers. The number of registers is limited and their access pattern must be known at compile time. Due to such limitations, it is difficult to model registers for algorithm design and their utilization in CUDA programs can be viewed as a low-level optimizations rather than a part of algorithm design. Thus, we do not consider registers when designing algorithms, although we use them in actual implementations.

3 Related Work

Over the past decade, many works have focused on designing efficient algorithms to solve a range of classical problems on the GPU [9, 26, 37, 20, 36, 39, 13]. These works have introduced several optimization techniques, such as coalesced memory accesses [12, 34, 10], branch reduction [24, 21], and bank conflict avoidance [21, 7]. Several empirical models for specific GPUs have been proposed that use micro-benchmarking [41, 19, 6], and several fast GPU algorithms have been produced [10, 20, 39] via the use of empirical benchmarks [40] and the application of hardware-specific optimization techniques to existing algorithms. While all these approaches can boost performance, abstract performance models are necessary to guide the design of provably efficient GPU algorithms. Several authors have proposed such models, attempting to capture salient features of the compute and memory hierarchies of modern GPUs in a way that balances accuracy and simplicity [23, 18, 1, 28, 29]. Despite these efforts, to date no model has been established as the definitive GPU performance model.

Since the problem of sorting has been extensively studied over the past half-century, in this section we focus on previous work relevant to sorting on the GPU [13, 5, 25, 24, 8, 27, 38]. According to a recent survey of several GPU libraries [27] the fastest currently-available sorting implementations include the CUB [25], modernGPU (MGPU) [5], and Thrust [17] libraries. CUB employs a GPU-optimized radix sort, and thus can only be applied to primitive datatypes. MGPU and Thrust use variations of mergesort (based on Green et al. [13]) and many hardware-specific optimizations to achieve peak performance. While highly optimized, these mergesort implementations issue sub-optimal numbers of global memory accesses and incur shared memory bank conflicts. Leischner et al. [24] introduced *GPU samplesort*, a distribution sort aimed at reducing the number of global memory accesses. Their work was continued by Dehne et al. [8] with a deterministic version of the samplesort algorithm. The work of Sitchinava et al. [38] focuses on shared memory only and presents an algorithm that sorts small inputs in shared memory without bank conflicts. Despite these efforts, no unified, provably efficient, and practical sorting algorithm has been presented. Thus, mergesort remains the algorithm of choice in top-performing GPU libraries [17, 5]. The sorting algorithm introduced in this work illustrates an analytical approach to designing GPU algorithms. This algorithm minimizes global memory accesses, incurs no shared memory bank conflicts, and outperforms state-of-the-art implementations in practice.

4 Multiway Mergesort

Mergesort is one of the most frequently used sorting algorithms today. It is simple, easily parallelizable [14] and load-balanced, and has optimal work complexity for comparison-based sorting. However, most mergesort implementations rely on pairwise merging, resulting in $\log N$ merge rounds to sort N values. In the context of the PEM model, described in Section 2.1, sequential pairwise merging requires $O(\frac{N}{B}\log\frac{N}{B})$ I/Os, and parallel mergesort requires $O(\frac{N}{PB}\log\frac{N}{B})$

I/Os [4]. Note that this is a factor $\log \frac{M}{B}$ more memory accesses than optimal. An alternative that achieves the lower bound for I/O complexity is *multiway mergesort*.

Like standard (pairwise) mergesort, multiway mergesort relies on repeated merge rounds, however, at each round, K lists are merged into a single list (to achieve optimal I/O complexity, $K = \frac{M}{B}$). To merge I/O efficiently, we must read and write only blocks of B consecutive elements and can only store a limited number of elements from each list in internal memory. Sequentially, this can be accomplished simply by using a minHeap and an output buffer. Multiway merging in parallel, however, is difficult and may require increased internal computation (which is ignored in the PEM model). On modern GPUs, both computation and memory accesses impact performance, so we develop a new method of parallelizing multiway merging.

4.1 Parallel Partitioning

A large amount of parallelism is required to effectively use the computational power of modern GPUs. Traditionally, multiway mergesort is not easily parallelized, so we introduce a technique of partitioning the problem into independent tasks that can be executed in parallel. We use the technique proposed by Hayashi et al. [16] to find the median among K lists (this is a generalization of the technique used by MGPU [14]). The technique is general and can be applied to find any i-th order statistic among K lists. This method can be used to provide us with P disjoint sets of elements (taken from all K merge lists), $s_0, s_1, ..., s_{p-1}$, where P is the number of warps executing on the GPU. Each of these sets, s_i, should have the property that its elements are *non-overlapping* w.r.t a given comparison function. Formally, we say sets s_i and s_j are non-overlapping sets iff for all $a \in s_i$ and $b \in s_j$, $a \leq b$ implies $i \leq j$.

Once each set s_i is sorted, the concatenation of $s_0 s_1 ... s_{p-1}$ will be sorted, as in quicksort. Since elements in each s_i are taken from the K lists, sorting each set becomes a smaller K-way merging problem. We now have p such smaller merging tasks, so each warp can independently merge its own subset of the overall problem. As described in the work of Hayashi et al. [16], we find pivot indices in each of our K list by performing a binary search on each. This can be accomplished in $O(K \log N)$ time, where N is the length of each list and K is the total number of lists. Figure 3 provides an example of such a set of pivots.

This process of partitioning can be executed in parallel for an arbitrary number of processors (or warps), P. Each warp performs a series of binary searches across K lists to find its set of pivots in $O(K \log N)$ I/Os. Each warp can then proceed with merging the values within its partition.

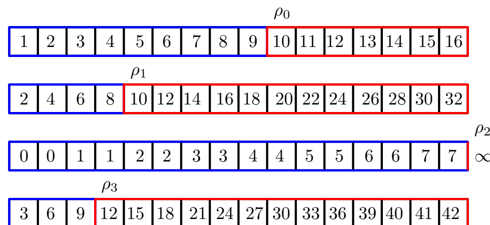


Figure 3: Example of a list of pivots creating 2 partitions.

4.2 I/O Efficient Merging

Using the parallel partitioning method described above, we are able to obtain independent work for each warp. Therefore, we need only a warp-level multiway merging algorithm to merge K lists using P warps. To do this I/O efficiently, however, we must read and write blocks of B consecutive elements at a time. As mentioned, this can be done sequentially using a heap and an output buffer, however the GPU provides additional parallelism within the warp that we need to exploit. Since B consecutive elements must be read by a *warp* at a time, independent

threads cannot work on separate sections and therefore must work in parallel on the same data. We do this using a new data structure called a *minBlockHeap*.

A *minBlockHeap* is a binary heap data structure where each node contains B sorted values. For any node v with elements $v[0], \cdots, v[B-1]$, and child nodes u and w, all elements contained in u or w must be greater than those contained in v (i.e., $v[B-1] \leq u[0]$ since each nodes list is sorted). Therefore, our heap property If this property is satisfied for our entire minBlockHeap, the root always contains the B smallest elements.

We define the *fillEmptyNode* operation that fills an empty node (i.e., a node without any elements in its list). Consider v to be an empty node with non-empty children u and w. W.l.o.g. assume that $u[B-1] > w[B-1]$. The $fillEmptyNode(v)$ operation is performed as follows: merge the lists of u and w, fill v with the B smallest elements, fill u with the B largest elements, and set w as empty. Since, prior to merging, u had the largest element ($u[B-1]$), its new largest element has not changed and the heap property holds for u. We can continue down the tree by calling $fillEmptyNode(w)$ until we reach a leaf, which we can fill by loading B new elements from memory.

To apply this data structure to multiway merging, we assign each of our K input lists to a leaf and build the minBlockHeap bottom-up using the *fillEmptyNode* operation. Each time a leaf node is empty, we fill it by reading B new elements from the corresponding list in global memory. Once the minBlockHeap is built, we begin writing the root to global memory (in blocks of B) as our sorted output. We propagate the resulting empty node down to a leaf node and fill it from global memory. We repeat this process until we have merged all K lists. Note that since we have k leaf nodes, our minBlockHeap has a height of $\log K$ and a total of $B(2K-1)$ total elements. Thus if $K = \frac{M}{2B}$, our minBlockHeap will require $B(\frac{M}{B} - 1) = M - B$ elements. The total number of I/Os of our multiway mergesort algorithm is $O(\frac{N}{PB} \log_{\frac{M}{B}} \frac{N}{B})$, as we show in Section 5.1.

4.3 Internal Memory Sort

The multiway mergesort algorithm described in Section 4 is a recursive algorithm that sorts a list after $O(\log_k N)$ merge rounds. However, if we employ an efficient internal memory sorting algorithm as a base case to the recursion, we can reduce the number of merge rounds to $O(\log_k \frac{N}{M})$. Since we will be performing this sorting strictly in shared memory, we wish to avoid bank conflicts (described in Section 2.2). We employ a variant of the *shearsort* algorithm [35] that efficiently uses GPU hardware and is bank conflict-free. We note that, a work-efficient variant of shearsort was introduced by Afshani et al. [2] and may be leveraged to improve performance. However, we leave this optimization to future work. The shearsort algorithm considers an input of n values as a $\sqrt{n} \times \sqrt{n}$ matrix. Shearsort sorts rows in alternating (ascending and descending) order, then columns in ascending order. It repeats this process $\log \sqrt{n}$ times and, after a final sort of rows in ascending order, the input is sorted.

For our base case, we have each warp perform a shearsort on a $W \times W$ grid of elements, where $W = 32$ for most modern GPUs. If we consider the grid of values to be in column-major order, each row corresponds to one of our W memory banks. Therefore, each thread can sort a row independently without any bank conflicts. Sorting columns, however, will clearly result in bank conflicts. Since transposition can be performed efficiently without any bank conflicts [7], we transpose our matrix, allowing us to sort columns without any bank conflicts. We repeat this process $\log W$ times to sort our base case of W^2 elements. Each warp can work on an independent set of elements, giving us a GPU efficient base case. Note that we can extend the size of our base case by sharing data between pairs of warps using bitonic merge. This allows us to achieve an ideal base case size to minimize the number of merge rounds for a given k and input size. We discuss this and other optimizations in more detail in Section 6.2.

5 Performance Analysis

In this section, we discuss the asymptotic performance of our multiway merge algorithm as compared to a standard pairwise merging technique. As mentioned in Section 2.1, we consider performance in the context of the PEM model. However, since modern GPUs employ a hierarchy of memory and computation, we also look at the total work done by each algorithm and the number of shared memory accesses and bank conflicts that may occur.

5.1 I/O Complexity

In the context of the PEM model, we need only consider global memory accesses when measuring algorithm complexity. As discussed in Section 2.2, global memory is accessible by all threads running on the GPU. To achieve peak throughput, all threads in a warp must access consecutive elements together. When this occurs, we have a *coalesced* memory access and all W threads receive their data in a single memory access. This access pattern can be seen as blocked access in the PEM model, so that a single I/O accesses B elements (where $B = W$). Therefore, we measure algorithm I/O complexity as the number of such accesses to global memory.

5.1.1 Pairwise Mergesort

The best-performing sorting algorithms available on the GPU today utilize pairwise mergesort. In terms of I/Os, this is rather simple to analyze. If we consider a total of P warps running concurrently on the GPU and an input of size N, the number of global memory accesses necessary for pairwise mergesort defines the recurrence relation:

$$Q(N) = \begin{cases} Q(\frac{N}{2}) + O(\frac{N}{PB}), & \text{if } N > M. \\ O(\frac{N}{P}), & \text{if } N \leq M. \end{cases}$$

$$= O\left(\frac{N}{PB}\log_2\frac{N}{M}\right)$$

This is the best we can hope to accomplish for pairwise mergesort, since we must access at least N elements during each merge round.

5.1.2 Multiway Mergesort

The multiway mergesort algorithm introduced in Section 4 aims to reduce the I/O complexity by performing a K-way merge at each round. At each merge round, each warp performs a binary search in global memory to find its partition, as outlined in Section 4.1. This results in $O(K \log N)$ I/Os, therefore the total I/O complexity for our multiway mergesort is:

$$Q(N) = \begin{cases} Q(\frac{N}{K}) + O(\frac{N}{PB} + K\log N), & \text{if } N > M. \\ O(\frac{N}{P}), & \text{if } N \leq M. \end{cases}$$

$$Q(N) = O\left(\frac{N}{PB}\left(\log_K\frac{N}{M} + 1\right) + K\log N\log_K\frac{N}{M}\right)$$

Assuming that K is smaller than $\frac{N}{PB\log N}$, the cost of merging dominates, and our asymptotic I/O complexity becomes:

$$Q(N) = O\left(\frac{N}{PB}\log_K\frac{N}{M}\right)$$

Asymptotically, multiway mergesort has a factor $O(\log K)$ less I/Os than a standard pairwise mergesort. To obtain optimal I/O complexity, $K = \frac{M}{B}$. However, on modern GPU hardware it is not clear how much internal memory we can assign to each warp while maintaining enough parallelism for peak performance. As discussed in Section 2.2, each SM has a fixed amount of shared memory, which can limit the number of warps running concurrently on each SM. Since GPUs depends on hyperthreading to hide memory latency, the amount of memory per warp (i.e., M), and therefore K, must be carefully selected. In Section 6.2 we empirically measure performance for a range of values for K.

5.2 Internal Computation

Since algorithm runtime on modern GPUs may not depend on global memory accesses alone, we also consider shared memory accesses during algorithm analysis. As mentioned in Section 2.2, the usability of registers is limited and we consider their use to be an optimization technique. Therefore, we can consider shared memory access to be our smallest unit of work in internal computation.

5.2.1 Pairwise Mergesort

Rather than analyzing the details of a particular pairwise mergesort implementation available in one of the GPU libraries, we consider the work complexity of a general pairwise mergesort algorithm. We note that Thrust and MGPU, two of the fastest mergesort implementations on the GPU, use a variation of the Merge Path [14] technique to achieve parallelism. Therefore, our analysis of a general mergesort algorithm includes this technique as well. Since the number of shared memory bank conflicts for most implementations are data dependent, we do not attempt to analytically model their impact and instead measure the impact empirically in Section 6. However, we expect for an average input, bank conflicts will increase the number of shared memory accesses needed by at least a factor of 2.

At each merge round, our general pairwise mergesort algorithm must: 1) find pivot points for each *warp* in global memory, 2) load portions of each list into shared memory and find pivots for each *thread*, and 3) have each thread merge its own section. Thus, the total parallel internal computation time is:

$$T(N) = \begin{cases} T(\frac{N}{2}) + O\left(\frac{N}{PW} + \log N + \frac{N}{PM}\log M\right), & \text{if } N > M. \\ O(N\log N), & \text{if } N \leq M. \end{cases}$$

$$= O\left(\frac{N}{PW}\log_2 N\right)$$

Note that PW is the total number of threads running on the GPU, making this algorithm asymptotically optimal in terms of parallel internal computation.

5.2.2 Multiway Mergesort

Unlike a typical pairwise mergesort algorithm, each step of our MMS algorithm employs data structures internally. While this may not increase the number of global memory accesses (I/Os),

Table 1: Specifics of our three hardware platforms.

Name	GPU Model	Architecture	Global memory	SMs	Cores per SM	Shared memory
Gibson	GTX 770	Kepler	4GiB	8	192	48 KiB
Algoparc	M4000	Maxwell	8GiB	13	128	96 KiB
UHHPC	K40m	Kepler	12GiB	15	192	48 KiB

it may increase the cost of internal computation. Therefore, we analyze each step of MMS to determine the total asymptotic parallel cost of internal computation.

At each merge round, each warp first finds K pivots to define its partition. This is done by $K \log N$ search steps, described in Section 4.1. Once the partitions are found, each warp performs its K-way merge using the minBlockHeap described in Section 4.2. Blocks of B elements are read in to a leaf node, must pass through the heap, and are outputted from the root. Recall that each time we output a block of B elements from the root, we call the *fillEmptyNode* method on each of the $\log K$ levels of the heap. Each call to *fillEmptyNode* involves merging two nodes of B elements each. We do this in parallel using all $W = B$ threads of a warp using a bitonic merging network in $2 \log W$ time. Thus, the total parallel internal computation of each K-way merge is:

$$T(N) = \begin{cases} T(\frac{N}{K}) + O\left(\frac{N}{PW} \log K \log W + K \log N\right), & \text{if } N > M. \\ O(N \log N), & \text{if } N \leq M. \end{cases}$$

$$= O\left(\frac{N}{PW} \log_2 N \log W\right)$$

We see that MMS requires an additional factor $\log W$ internal computation, compared with pairwise mergesort. However, on modern GPU hardware, $W = 32$, so $\log W = 5$. Furthermore, MMS is bank conflict-free, while pairwise mergesort implementations may incur up to W-way bank conflicts, resulting in a potential performance loss of $O(W)$. In Section 6 we attempt to verify this hypothesis by measuring the empirical performance of pairwise mergesort implementations.

6 Empirical Performance Results

The analysis in the previous section indicates that our MMS algorithm provides key advantages over a pairwise mergesort algorithm. In this section, we evaluate the performance of our implementation of MMS on a range of hardware platforms and input. We measure execution time, throughput (number of elements sorted per second), as well as numbers of bank conflicts and numbers of global memory accesses.

6.1 Methodology

We consider three hardware platforms, each with a different modern graphics card. All computations are performed on on the graphics cards, and no attempt is made to use CPU compute resources. Furthermore, 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. The specifications of the GPUs of our three platforms are listed in Table 1. On all platforms we use GCC 4.8.1 and CUDA 7.5, and all experiments are compiled with the -O3 optimization flag. Performance and other metrics are obtained via the nvprof profiling

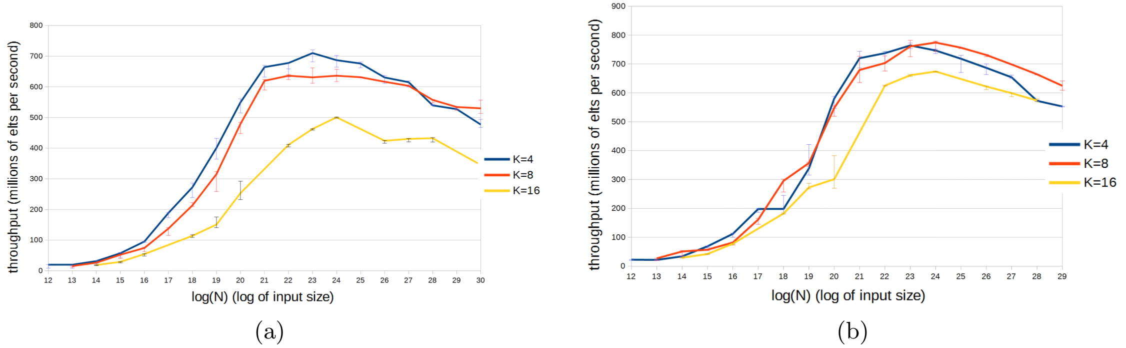


Figure 4: Average throughput vs. N for $K = 4, 8, 16$ on (a) UHHPC and (b) Algoparc.

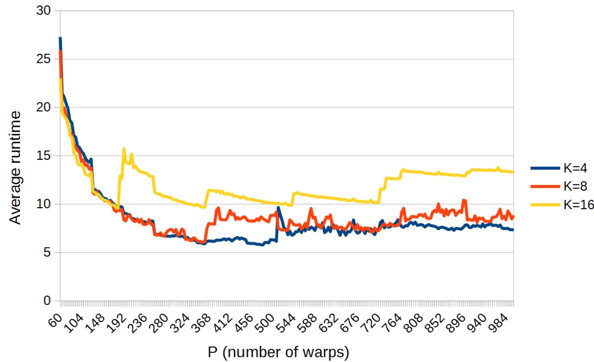


Figure 5: Average runtime vs. P for $K = 4, 8, 16$ on UHHPC. Error bars are omitted for readability.

tool [33], included in the CUDA 7.5 toolkit. Each experiment is repeated ten times, and we report on mean values, showing min-max error bars when non-negligible.

We compare the performance of MMS with three leading GPU sorting libraries: Thrust 1.8.1 [17], modernGPU (MGPU) 2.10 [5], and CUB 1.6.4 [25]. Thrust and MGPU implement pairwise mergesort algorithms and provide the fastest comparison-based sorts available on the GPU. CUB provides the highest-performing radix sort. Although CUB is not a comparison-based sort, and is therefore limited to primitive datatypes, we include it in some of our experiments for completeness. We also include in some of our experiments the I/O-efficient samplesort implementation in [24].

6.2 Implementation Optimization and Parameter Tuning

We employ several typical optimization techniques in our implementation of MMS [1]. First, when working independently threads make use of registers, which is straightforward to implement and improves performance significantly. Second, we use the warp shuffle [32] operation that has been available in nVidia GPUs since the Kepler architecture. This operation lets threads within a warp communicate between registers without relying on shared memory. The shuffle operation leads to performance improvements when the register access patterns of particular threads are deterministic. As a result, we use it for to implement the bitonic merge used by the minBlockHeap data structure (described in Section 4.2), and to avoid the transposition step of our internal memory shearsort (described in Section 4.3). An third straightforward optimization is to vary the base case size depending on N, so as to avoid an additional merge round when

[1] Note, however, that our implementation remains relatively un-optimized compared to the MGPU and Thrust implementations.

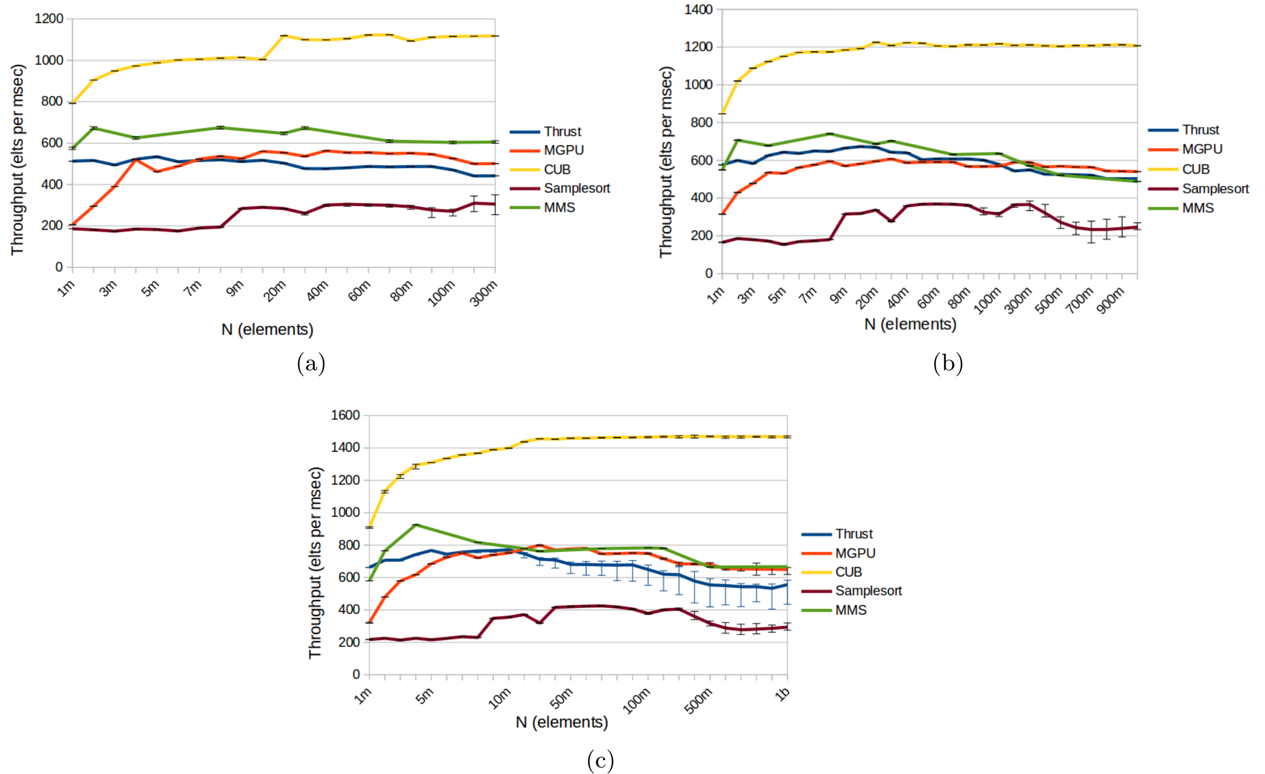


Figure 6: Average throughput vs. input size for fully random input on (a) Gibson, (b) UHHPC, and (c) Algoparc.

input sizes do not precisely fit. MMS performs $\lceil \log_K \frac{N}{|\text{basecase}|} \rceil$ merge rounds. For some input sizes, one merge round will involve fewer than K lists and work will be wasted. The larger K, the larger the performance loss due to this wasted work. Therefore, we simply increase the base case size so as to reduce the number of necessary merge rounds.

Two parameters that have a key influence on the performance of MMS are K (the number of lists merged at each round) and P (the number of warps). We determine good values for these parameters empirically for each platform. We measure average execution time over 10 trials for K and P value combinations ($K \in \{4, 8, 16\}$ and $P \in [60, 1000]$) when sorting an input of size $N = 10^7$. For each K value we then determine the empirically best P value. Figure 5 shows results of this experiment on the UHHPC platform.

On each platform, for a given K value, we set P to the best value determined in the previous experiment, and measure average throughput as N varies. We repeat this for each $K \in \{4, 8, 16\}$. Figure 4 shows results of these experiments for the UHHPC and Algoparc platforms. We do not show results for the Gibson platform because it is similar to UHHPC (both are “Kepler” generation GPUs), and thus lead to the same conclusion that the smallest K value leads to the best average performance (except for very large inputs). However, for Algoparc (which is a “Maxwell” generation GPU), $K = 8$ is best for most values of N. The primary difference between the Maxwell and Kepler GPUs is the amount of shared memory per SM. Our Maxwell GPU has 96KiB of shared memory per SM, while our Kepler GPUs have only 48KiB per SM. The larger K, the larger the amount of shared memory utilized by each warp. The size of the shared memory thus limits the number of warps that can be running concurrently on each SM, explaining why a larger shared memory allows for the effective use of a larger K value.

6.3 Experimental Results

In this section our main performance metric is the average throughput, which we measure for each sorting algorithm implementation for various input and input sizes. As explained in Section 1, we generate input of varying "sortedness," since sortedness has a large impact on the performance of Thrust and MGPU. By permuting each element in our initial sorted list at least once, we generate a fully random, uniformly distributed input (without repeats). Unless specified otherwise, results are obtained by sorting input that consists of 4-byte integers.

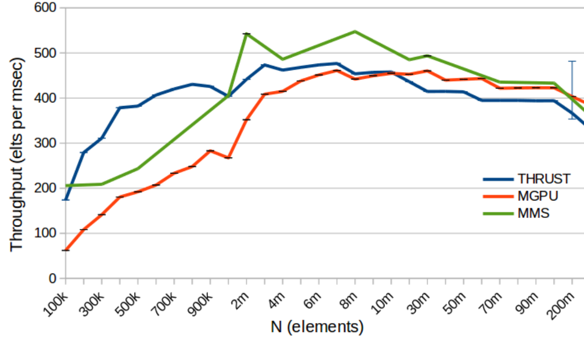
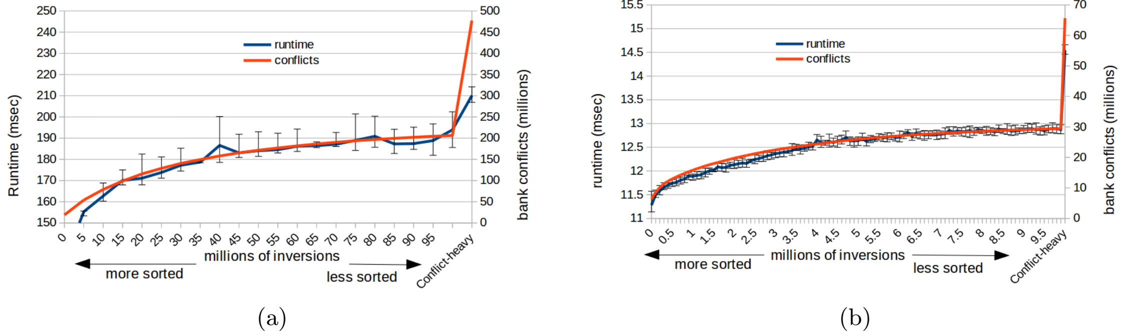


Figure 7: Average throughput vs. input size for full random inputs of 8-byte long datatypes on UHHPC.



(a) (b)

Figure 8: Average runtime vs. input sortedness for (a) sorting 10^8 items using MGPU on the Gibson platform and (b) sorting 10^7 items using Thrust on the Algoparc platform. Results for conflict-heavy input are shown as the rightmost datapoint on the horizontal axes.

Figure 6 shows the throughput achieved by each implementation when applied to fully random input of increasing sizes for each of our three platforms. These results show that MMS performs comparably to the MGPU and Thrust comparison-based sorting implementations. MMS outperforms both MGPU and Thrust for large input sizes. This indicates MMS provides better scalability, due to its optimal number of global memory accesses and to the fact that it avoids all bank conflicts. As expected, CUB, being a radix sort, achieves much higher throughput across all input sizes. The samplesort implementation performs significantly worse than all its competitors across the board.

Figure 7 shows results on UHHPC for 8-byte integers for MMS, MGPU, and Thrust (similar results are obtained on Gibson and Algoparc). The trend in these results are similar to those observed for 4-byte integers, with MMS comparing favorably with its two competitors.

6.4 Impact of Bank Conflicts

One key feature of MMS is that it is shared memory bank conflict-free for any input. MGPU and Thrust, however, have memory access patterns that depend on the input. As seen in Section 1, MGPU performs increasingly worse as the input is more random (i.e., unsorted). Since the memory access patterns of MGPU and Thrust are deterministic, we should be able to generate input that will cause these algorithms to incur large numbers of bank conflict. While such *conflict-heavy* input may result in performance degradation for Thrust and MGPU, the performance of MMS, because bank conflicts are completely avoided, does not depend in the input.

To generate conflict-heavy input, we analyze the memory access pattern of the mergesort algorithm in MGPU. While Thrust employs a similar algorithm, the code is more difficult to analyze. However, it turns out that conflict-heavy generated based on our MGPU analysis is also conflict-heavy for Thrust. MGPU performs merging in shared memory by having each thread merge two lists. The lengths of these lists depend on the GPU generation. On Kepler, resp. Maxwell, each thread merges lists of 11, resp. 15, items at each round. Note that these values are chosen to be co-prime with 32 (the number of banks) so as to reduce bank conflicts. Each thread then uses the Merge Path [14] method to find pivots and then merges its items in shared memory. Since the location of the elements accessed by each thread depends on the input, we can generate conflict-heavy input that result in large numbers of bank conflicts at every memory access. We create a small conflict-heavy input by hand and generate larger conflict-heavy input by copying and interleaving smaller conflict-heavy input. This methods generates input with numbers of items that are powers of two.

Figure 8 shows average execution time and number of bank conflicts vs. input sortedness, as defined in Section 1, for two sample experiments (MGPU on Gibson and Thrust on Algoparc). The rightmost data point on the horizontal axis corresponds to the conflict-heavy input generated as described above. These results confirm that our conflict-heavy input does indeed lead to large numbers of bank conflict for both MGPU and Thrust. In both cases it leads to more than twice as many bank conflicts as when sorting fully random input. These results further corroborate the preliminary results in Figure 1: shared memory bank conflicts are one of the key drivers of algorithm performance.

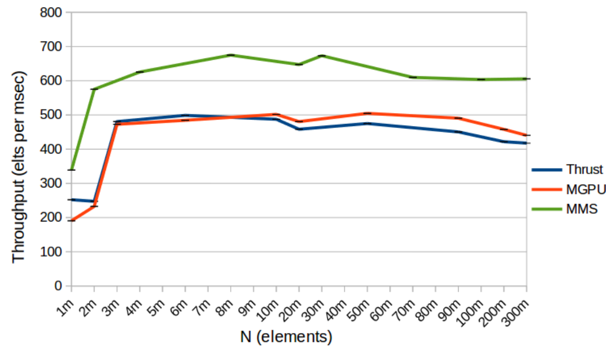


Figure 9: Average throughput vs. input size for *conflict-heavy* input on the Gibson platform.

Using our conflict-heavy input, we compare the average throughput of each implementation. Figure 9 presents results on the Gibson platform. Overall, on conflict-heavy input, MMS achieves up to 33.7% and 35.2% higher throughput than MGPU and Thrust, respectively. We obtain similar results on our two other platforms. On UHHPC, MMS outperforms MGPU and Thrust by up to 32.3% and 34.6%, respectively. On Algoparc MMS achieves up to 27.8% and 28.3% improvements over MGPU and Thrust, respectively.

6.5 Performance Bottlenecks for MMS

MGPU and Thrust provide highly optimized sorting implementations, and their performance shortcomings are due to the nature of the sorting algorithm itself rather than to overlooked implementation details. More specifically, these implementations use a pairwise mergesort algorithm, which does not achieve the I/O complexity lower bound and causes shared memory bank conflicts. By contrast, MMS uses the memory hierarchy efficiently, but could likely benefit from various optimizations in addition to those mentioned in Section 6.2. In this section we identify the bottleneck component in our current MMS implementation.

Recall from Section 4 that the MMS algorithm consists of a partitioning phase and a merging phase at each recursive level. Using the nvprof [33] profiler, we find that, for the ideal P values determined as explained in Section 6.2, the partitioning phase makes up less than 2% of the overall execution time. Furthermore, we find that the node merging component of the *fillEmptyNode* heap method (described in Section 4.2) contributes to more than 60% of the overall runtime. This is due to the communication required between threads. More specifically, 32 threads (one warp) work on merging two lists of 32 elements each, which is done using a bitonic merging network, thus requiring a large amount of thread communication. Although we leave the optimization of this component of MMS for future work, its optimization should greatly increase overall performance.

7 Conclusions

In this work we present MMS, a new GPU-efficient multiway mergesort algorithm. By using the PEM model and considering shared memory access patterns, we show that MMS achieves an optimally minimal number of global memory accesses and does not cause any shared memory bank conflicts. Furthermore, through the use of a new parallel partitioning method, MMS exposes the high level of parallelism needed to approach peak GPU performance.

We perform a detailed empirical analysis and compare performance results of MMS with two highly optimized comparison-based sorting implementations, MGPU and Thrust. Our results show that MMS exhibits performance comparable to MGPU and Thrust on randomly generated input, and outperforms them for most large input. A performance shortcoming of MGPU and Thrust is bank conflicts, which we highlight by generating input that causes these implementations to incur larger numbers of bank conflict. On such input MMS, whose performance is not sensitive to the input, achieves throughput up to 35.2% higher when sorting 4-byte integer items.

Since MMS is issues the optimal number of global memory accesses and avoids bank conflicts, we expect it to scale more efficiently than standard pairwise mergesort, such as MGPU and Thrust, algorithms as hardware improves and memory capacities grow. In addition, larger shared memories will enable MMS to utilize a larger branching factor, further improving its performance.

We leave several performance improvements to MMS as future work. Some of the hardware-specific optimizations used by MGPU and Thrust may yield performance gains for MMS as well. Since merge operations on the minBlockHeap structure are a significant bottleneck (see Section 6.5), techniques such as pipelining should improve MMS performance as well. Finally, we may be able to develop a variation of MMS that avoids the use of minBlockHeap altogether and instead relies on the parallel partitioning technique to perform multiway merging. Such an algorithm may be able to avoid the additional internal work and other drawbacks of using a heap, while retaining I/O efficiency.

References

[1] A Memory Access Model for Highly-Threaded Many-Core Architectures. *Future Generation Computer Systems*, 30:202–115, 2014.

[2] P. Afshani and N. Sitchinava. Sorting and permuting without bank conflicts on gpus. In *Proc. of ESA*, pages 13–25, 2015.

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

[4] L. Arge, M. Goodrich, M. Nelson, and N. Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *Proc. of SPAA*, pages 197–206, 2008.

[5] S. Baxter. Modern GPU. http://nvlabs.github.io/moderngpu/, 2013.

[6] N. Bombieri, F. Busato, and F. Fummi. A fine-grained performance model for gpu architectures. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2016.

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

[8] F. Dehne and H. Zaboli. Deterministic sample sort for GPUs. volume abs/1002.4464, 2010.

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

[10] P. Enfedaque, F. Auli-Llinas, and J. Moure. Implementation of the DWT in a GPU through a register-based strategy. *IEEE Trans. PDS*, PP(99), 2014.

[11] J. N. et al. Scalable parallel programming with CUDA. In *ACM Queue*, volume 6, pages 40–53, 2008.

[12] 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.

[13] O. Green, R. McColl, and D. A. Bader. GPU merge path: a GPU merging algorithm. In *Proc. of ICS*, pages 331–340, 2012.

[14] O. Green, S. Odeh, and Y. Birk. Merge path - A visually intuitive approach to parallel merging. *CoRR*, abs/1406.2628, 2014.

[15] G. Greiner. *Sparse matrix computations and their I/O complexity.* PhD thesis, Technische Universitt Mnchen, Mnchen, 2012.

[16] T. Hayashi, K. Nakano, and S. Olariu. Weighted and unweighted selection algorithms for k sorted sequences. In *Proceedings of the 8th International Symposium on Algorithms and Computation*, pages 52–61, London, UK, UK, 1997. Springer-Verlag.

[17] J. Hoberock and N. Bell. Thrust: A parallel template library. http://thrust.github.io/, 2010. Version 1.7.0.

[18] S. Hong and H. Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *Proc. of the 36th Intl. Symp. on Computer Architecture (ISCA)*, pages 152–153, 2009.

[19] S. Hong and H. Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *Proc. of ISCA*, 2009.

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

[21] B. Karsin, H. Casanova, and N. Sitchinava. Efficient batched predecessor search in shared memory on GPUs. In *Proc. of HiPC*, pages 335–344, 2015.

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

[23] K. Kothapalli, R. Mukherjee, S. Rehman, S. Patidar, P. Narayanan, and K. Srinathan. A performance prediction model for the CUDA GPGPU. In *Proc. of the Intl. Conf. on High-Performance Computing (HiPC)*, 2009.

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

[25] D. Merrill. Cub: Cuda unbound. http://nvlabs.github.io/cub/, 2015.

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

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

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

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

[30] J. Nickolls and W. Dally. The gpu computing era. In *IEEE Micro*, volume 30, pages 56–69, 2010.

[31] NVIDIA. nVidia Tesla K40 specifications. http://www.nvidia.com/content/tesla/pdf/nvidia-tesla-kepler-family-datasheet.pdf, 2014.

[32] NVIDIA. CUDA programming guide 7.0. http://docs.nvidia.com/cuda, 2015.

[33] NVIDIA. Nsight. http://www.nvidia.com/object/nsight.html, 2015.

[34] 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*, pages 73–82. ACM, 2008.

[35] S. Sen, I. Sherson, and A. Shamir. Shear sort: A true two-dimensional sorting techniques for vlsi networks. In *International Conference on Parallel Processing*, pages 903–908, 1986.

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

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

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

[39] J. Soman, K. 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 J. D. Owens. A quantitative performance analysis model for GPU architectures. In *Proc. of HPCA*, 2011.