Sorting Large Datasets with Heterogeneous CPU/GPU Architectures

Michael Gowanlock
School of Informatics, Computing, & Cyber Systems
Northern Arizona University
Flagstaff, AZ, 86011
michael.gowanlock@nau.edu

Ben Karsin
Department of Information and Computer Sciences
University of Hawaii at Manoa
Honolulu, HI, 96822
karsin@hawaii.edu

***Abstract*—We examine heterogeneous sorting for input data that exceeds GPU global memory capacity. Applications that require significant communication between the host and GPU often need to obviate communication overheads to achieve performance gains over parallel CPU-only algorithms. We advance several optimizations to reduce the host-GPU communication bottleneck, and find that host-side bottlenecks also need to be mitigated to fully exploit heterogeneous architectures. We demonstrate this by comparing our work to end-to-end response time calculations from the literature. Our approaches mitigate several heterogeneous sorting bottlenecks, as demonstrated on single- and dual-GPU platforms. We achieve speedups up to 3.47$\times$ over the parallel reference implementation on the CPU. The current path to exascale requires heterogeneous architectures. As such, our work encourages future research in this direction for heterogeneous sorting in the multi-GPU NVLink era.**

I. Introduction

Data transfers between the host (CPU) and device (GPU) have been a persistent bottleneck for General Purpose Computing on Graphics Processing Units (GPGPU). Current technology uses PCIe v.3 for data transfers, with a peak aggregate data transfer rate of 32 GB/s. NVLink increases the aggregate bandwidth to up to 300 GB/s [1]. However, the data transfer rate remains a bottleneck; for example, multi-GPU platforms will compete for bandwidth in applications with large host-device data transfers.

Due to their high throughput, accelerators are increasingly abundant in clusters. Flagship clusters are decreasing the aggregate number of nodes, and increasing per-node resources. This is important, as: (i) network communication is a bottleneck; thus, executing a larger fraction of the computation on each node instead of distributing the work reduces inter-node communication overhead [2]; (ii) reducing the number of nodes in a cluster improves node failure rates, which require expensive recovery strategies, such as checkpointing and rollback [3]; (iii) accelerators are more energy efficient than CPUs for some applications [4]. Consequently, increasing per-node capacity and decreasing node failure rates helps *pave the way to exascale*.

We examine the problem of sorting using a heterogeneous approach, defined as follows: given an unsorted input list, A, of n elements in main memory, generate a sorted output list, B, using the CPUs and GPU(s) on the platform. Depending on the strategy employed to generate B, the memory bandwidth between the CPU and main memory may also become a bottleneck. Thus, it is not clear that multi-core CPU-only parallel sorting algorithms will not outperform a hybrid CPU/GPU approach.

Recently, [5] advanced a state-of-the-art radix sort for the GPU. They apply their sort to the problem of heterogeneous sorting. While the results for the on-GPU sorting are impressive, several overheads in the hybrid sorting workflow appear to be omitted. In this work, we reproduce their results to determine the extent that missing overheads may degrade the performance of their heterogeneous sorting pipeline.

This paper has two goals: (i) advance an efficient hybrid approach to sorting data larger than GPU memory that maximizes the utilization of both the CPU and GPU; and, (ii) reproduce the method used for computing the end-to-end time in [5], and compare it to our end-to-end performance that includes all overheads. Our contributions are as follows:

- We advance a heterogeneous sort that utilizes both multi-core CPUs and many-core GPU(s) that outperforms the parallel reference implementation when sorting data that exceeds GPU global memory capacity.
- We demonstrate critical bottlenecks in hybrid sorting, and propose several optimizations that reduce the load imbalance between CPU and GPU tasks.
- We show that these bottlenecks cannot be omitted when evaluating the performance of heterogeneous sorting, as may be the case in the literature.
- Our heterogeneous sorting optimizations achieve a high efficiency relative to the modeled baseline lower bound.

Paper organization: Section II describes related work on CPU and GPU sorting. Section III advances the hybrid algorithm to sort datasets exceeding global memory. Section IV evaluates the proposed algorithm. In Section V we discuss the future of heterogeneous sorting, and conclude the paper.

II. Related Work

Sorting is a well-studied problem due to its utility as a subroutine of many algorithms [6]. In this paper, we focus on sorting large datasets using a heterogeneous platform consisting of CPUs and GPUs. Such systems are commonplace,

and at present, are the most likely path to exascale [7], [8]. We do not advance a new on-GPU or CPU sorting algorithm. Rather, we utilize state-of-the-art sorting algorithms within the broader problem of heterogeneous sorting.

A. Parallel Sorting Algorithms

Over the past several decades, parallelism has become increasingly important when solving fundamental, computationally intensive problems. Sorting is one such fundamental problem for which there are several efficient parallel solutions. When sorting datasets on keys of primitive datatypes, radix sort provides the best asymptotic performance [9] and has been shown to provide the best performance on a range of platforms [10]–[14]. When sorting based on an arbitrary comparator function, however, parallel algorithms based on Mergesort [15], Quicksort [16], and Distribution sort [17] are commonly used. Each of these algorithms has relative advantages depending on the application and hardware platform. Regarding Mergesort, parallelism is trivially obtained by concurrently merging the many small input sets. However, once the number of merge lists becomes small, merge lists must be partitioned to increase parallelism [18]. Multiway variants can similarly be parallelized and further reduce the number of accesses to slow memory at the cost of a more complex algorithm [19], [20]. Quicksort is an algorithm that is shown to perform well in practice, and can be parallelized using the parallel prefix sums operation [16]. See [9] for an overview of these algorithms.

Highly optimized implementations of parallel sorting algorithms are available for both CPU [19]–[21] and GPU [13], [22], [23]. The Intel Thread Building Blocks (TBB) [21] and MCSTL [19] parallel libraries provide highly optimized parallel Mergesort algorithms for CPUs.

Several libraries also provide optimized radix sort algorithms [20], [24]. The PARADIS algorithm [11] has been shown to provide the best performance on the CPU, but we were unable to obtain the code for evaluation.

B. GPUs and Hybrid Architectures

GPUs have been shown to provide significant performance gains over traditional CPUs on many general-purpose applications [25]–[28]. Applications that can leverage large amounts of parallelism can realize remarkable performance gains using GPUs [29], [30]. However, memory limitations and data transfers can become bottlenecks [31].

While sorting on the GPU has received a lot of attention [5], [28], [32]–[35], most papers assume that inputs fit in GPU memory and disregard the cost of data transfers to the GPU. In the GPU sorting community, this is reasonable as the overhead of CPU-GPU transfers is independent of the sorting technique used on the GPU. The Thrust library [22] provides radix and Mergesort implementations that are highly optimized for GPUs and are frequently used as baselines [33]. Several works present implementations that outperform Thrust [5], [14], [34], although their performance gains are limited and the Thrust library is continually updated and improved. We use Thrust for on-GPU sorting, as the impact of performance gains will have a small overall impact when accounting for CPU-GPU transfers.

Fully utilizing the CPUs and GPU(s) is a significant challenge, as data transfer overheads are frequently a bottleneck [31], [36]. As such, sorting using a hybrid CPU/GPU approach has been mostly overlooked. Recently, Stehle and Jacobsen [5] proposed a hybrid CPU/GPU radix sort, showing that it outperforms existing CPU-only sorts. However, as we show in Section IV, this work ignores a significant portion of the overheads associated with data transfers. In this work, we consider all potential performance loss due to data transfers to determine the efficacy of CPU/GPU sorting.

III. Heterogeneous Sorting

We present our approach to hybrid CPU/GPU sorting. We use NVIDIA CUDA [37] terminology when discussing GPU details. The approach we present relies on sorting on the GPU and merging on the CPU. We divide our unsorted input list *A* into n_b sublists of size b_s, that we call *batches*, to be sorted on the GPU. When describing our approach, we assume b_s evenly divides n (i.e., $b_s = \frac{n}{n_b}$). We merge the n_b batches on the CPU to create the final sorted list, *B*. Table I outlines the parameters and notation we use in this work.

Table I: Table of notation.

	Description
n	Input size.
n_b	Number of batches/sublists to be sorted on the GPU.
n_{GPU}	Number of GPUs used.
n_s	The number of streams used.
b_s	Size of each batch to be sorted.
p_s	Size of the pinned memory buffer.
A	Unsorted list to be sorted (n elements).
B	Output sorted list (n elements).
W	Working memory for sublists to be merged (n elements).
HtoD	Data transfer from the host to device.
DtoH	Data transfer from the device to host.
Stage	Pinned memory staging area.
MCpy	A host-to-host memory copy to or from pinned memory.
GPUSort	Sorting on the GPU.
Merge	Multiway merge of n_b batches.

A. Parallel Merging on the CPU

Sorting sublists occurs on the GPU and merging occurs on the CPU. After all n_b batches have been sorted on the GPU and transferred to the host, we use the parallel multiway merge from the GNU library. Merging the $\log n_b$ batches into the final sorted list *B* requires $O(n \log n_b)$ work and $O(\log n_b)$ time in the CREW PRAM model. Preliminary results indicate that multiway merge is more cache-efficient than pairwise merging. We use the multiway merge for merging after all batches have been produced. However, as discussed below, we will use pair-wise merges in one of our pipelining optimizations, in addition to the multiway merge.

B. Sorting on the GPU

We use the Thrust library [22] to sort sublist batches on the GPU. As with many high-performance sorting implementations, Thrust sorts out-of-place, requiring double the memory of the input list to perform the sort. This doubles the memory required on the GPU for each batch to be sorted, and thus doubles the total number batches, n_b, needed to sort the n elements in *A*. Thus, the memory requirement has a negative impact on the merging procedure, as it will increase the total amount of merging work to be performed on the CPU, as merging requires $O(n\log n_b)$ work (Section III-A).

C. Trade-Offs and Space Complexity

The memory requirements of the hybrid algorithm include the unsorted input list (*A*), working memory for the sorted sublists computed on the GPU (*W*), and the sorted output list (*B*), yielding $\sim 3n$ space. The space requirements for the GPU are not counted here, as all memory on the GPU is considered to be used temporarily. The only way to reduce the space complexity on the CPU is to perform an in-place parallel multiway merge. Merging in-place is known to be a challenging problem and leads to a decrease in performance [35], [38], as threads need their own working memory for merging the segments of the sublists. Furthermore, merging cache-efficiently, in-place, and in parallel remains an open problem. Thus, in this work, we employ out-of-place merging to achieve peak performance.

D. Approaches and Optimizations

Our baseline assumes that *A* fits in memory on a single GPU. We simply sort *A* on the GPU by copying the data from the host to the GPU, sort the data, and then transfer the result to the host, denoted as: *A* $\rightarrow$ *HtoD* $\rightarrow$ *GPUSort* $\rightarrow$ *DtoH* $\rightarrow$ *B*. All data transfers are *blocking* using *cudaMemcpy*, and the default CUDA stream is used. For further details on CUDA memory transfers and streams, see [37]. We refer to this approach as BLINE.

1) Baseline With Multiple Batches: When *A* is larger than GPU global memory ($n_b > 1$), the batches, once sorted, need to be merged on the host. Figure 1 shows an example with 6 batches ($n_b = 6$) being sorted on the GPU and then merged on the CPU (using a multiway merge). Since merging occurs after all batches have been sorted, there is load imbalance between the CPU and GPU. Figure 1 illustrates a hypothetical example where merging all batches requires $3\times$ the time required to sort a batch on the GPU. While merging on the host requires $O(n\log n_b)$ work, and sorting one batch requires $O(\frac{n}{n_b}\log\frac{n}{n_b})$ work, the asymptotic performance is inadequate to compute the load imbalance between CPU and GPU tasks, due to differing architectures, and overheads that dominate on-GPU sorting.

In the baseline approach with multiple batches, we use blocking data transfers and the default CUDA stream. The workflow is as follows: *A* $\rightarrow$ *HtoD* $\rightarrow$ *GPUSort* $\rightarrow$ *DtoH* $\rightarrow$ *W* $\rightarrow$ *Merge* $\rightarrow$ *B*. This is denoted as BLINEMULTI.

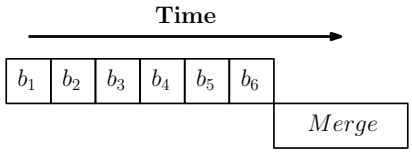


Figure 1: Example of generating $n_b = 6$ on the GPU and then merging the result with multiway merge on the CPU, if multiway merge requires $3\times$ the time to sort one batch.

2) Pipelining Data Transfers: When sorting data larger than global memory, $n_b > 1$, we can overlap data transfers when sorting sublists. To maximize bidirectional bandwidth utilization over PCIe, data can be simultaneously sent to the GPU (*HtoD*), and sent to the host (*DtoH*). To pipeline data transfers, different CUDA streams must be used. Transferring data in a stream requires using *cudaMemcpyAsync* which uses *pinned memory* allocated with *cudaMallocHost*. Pinned memory allows for faster transfer rates, but it is expensive to allocate. Pinned memory buffers are typically used as a staging area to incrementally copy data to or from a larger buffer. As a result, the host must copy data between pinned memory and a larger buffer. Here, the larger buffer is either the unsorted list, *A*, or the working memory, *W*, that will be used for the multiway merge (or *B* directly if $n_b = 1$). When using a single GPU ($n_{GPU} = 1$) each stream is assigned a number of batches to sort, n_b/n_s (assuming n_s even divides n_b). For multi-GPU systems, the number of batches to sort per stream is $n_b/(n_s n_{GPU})$.

The workflow when $n_b > 1$ is as follows: *A* $\rightarrow$ *Stage* $\rightarrow$ *HtoD* $\rightarrow$ *GPUSort* $\rightarrow$ *DtoH* $\rightarrow$ *Stage* $\rightarrow$ *W* $\rightarrow$ *Merge* $\rightarrow$ *B*. For completeness, when $n_b = 1$ and no overlapping of data transfers, the workflow is as follows: *A* $\rightarrow$ *Stage* $\rightarrow$ *HtoD* $\rightarrow$ *GPUSort* $\rightarrow$ *DtoH* $\rightarrow$ *Stage* $\rightarrow$ *B*. In this case, BLINE is used (no data transfer overlap possible).

Figure 2 illustrates pipelining in two different contexts: host-only operations, and data transfers. First, four streams, $s_1, \ldots, s_4$, are able to simultaneously copy their data from *HtoD* (bottom) and *DtoH* (top). This relies on allocating pinned memory buffers for each stream. The buffer is typically allocated to be smaller than the batch size, b_s. In Figure 2, the pinned memory buffer size is $p_s = b_s/3$. Thus, we incrementally copy data in either direction using pinned memory as a staging area. This allows for the fine interleaving of data transfers shown in Figure 2.

Operations in Figure 2 are identical in duration. In practice, the time to perform the individual operations will vary. Load imbalance between copying on the host to/from pinned memory (*MCpy*) or data transfers (*HtoD*, or *DtoH*), will impact the number of streams needed to most efficiently overlap these operations. If memory copies in/out of pinned memory are a bottleneck, then more streams can be used for

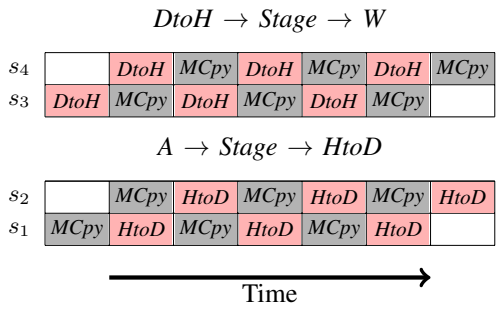


Figure 2: Illustrative example of pipelining the data transfers in streams, denoted $s_1, \ldots, s_4$. Pipeline shows incrementally copying data from *HtoD* using the pinned memory buffer (lower), and copying the data from *DtoH* using the pinned memory buffer (upper). Incremental copies (*HtoD* or *DtoH*) are interleaved with copying on the host. Red cells show data transfers and gray cells show host-side operations.

simultaneous data transfers; likewise, if the data transfers are a bottleneck, then fewer streams should be utilized. We refer the method of pipelining data transfers as PIPEDATA.

Data transfer performance cannot be improved beyond pipelining *HtoD* and *DtoH*. In contrast, if host-to-host *MCpy* to/from pinned memory in *Stage* is a bottleneck when using std::memcpy, we can parallelize this step on the host. We denote parallelizing the *MCpy* in *Stage* as PARMEMCPY.

3) Pipelining The Merge Phase: While the GPU is sorting batches, the host exclusively performs memory-intensive operations, such as memory copies to and from the GPU (*HtoD*, and *DtoH*), and host-to-host copies of data to and from pinned memory in the *Stage* phase. Thus, there is an opportunity to decrease the load imbalance shown in Figure 1 between GPU (sorting) and CPU (merging), by starting the merge stage before all batches are sorted (in addition to overlapping CPU/GPU work). This is important as n_b increases, since the amount CPU merging work increases. Once two batches are sorted and transferred back to the CPU, we can perform pair-wise merges in parallel, reducing the amount of work done by the final multiway merge. For instance, in Figure 3, m_1 merges b_1 and b_2, and m_2 merges b_3 and b_4. Thus, the final multiway merge will consist of 4 total sublists: the two merged sublists from m_1 and m_2, and, b_5 and b_6. Comparatively, using BLINEMULTI, the CPU does not begin merging until all batches are sorted, requiring that the final multiway merge combine 6 batches.

How should pairs of sorted batches be merged? An important observation is that after the last batch is sorted on the GPU, all pair-wise merging should be finished, such that the pair-wise merges do not delay the final multiway merge. As such, we utilize the heuristics below to compute the number of pair-wise pipelined merges to perform:

- When $n_{GPU} = 1$: $\lfloor \frac{n_b - 1}{2} \rfloor$. This ensures that when n_b is even, the last two pairs of batches are not pair-wise merged, and when n_b is odd, the last batch is not merged.
- When $n_{GPU} \geq 2$: $\lfloor \frac{n_b}{2n_{GPU}} \rfloor$. Because batches are sorted faster with 2 or more GPUs, there is less time for pair-wise merging on the host. Thus, fewer merges can occur before the multiway merge phase, as a function of n_{GPU}.

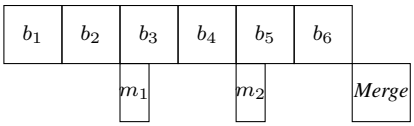


Figure 3: Pipelining merging pairs of sublists on the CPU while the GPU sorts batches. For illustrative purposes, we assume that the time needed for a pair-wise merge is half of the time to sort a batch on the GPU. Here, m_1 merges sorted batches b_1 and b_2, and m_2 merges b_3, and b_4.

We do not pipeline merging any sublists that are the product of a previous pipelined merge (e.g., the output of m_1 in Figure 3), and only merge pairs of sublists that are of size b_s. We find that merging sublists in an "online" fashion (i.e., as they are produced on the GPU), or using a merge tree to determine optimal merges, results in delaying the multiway merging procedure, and thus degrades performance. We refer to our approach that pipelines both pair-wise merges and data transfers (Section III-D2) as PIPEMERGE.

4) Summary of Approaches and Optimizations:

•BLINE– Baseline for sorting a single batch ($n_b = 1$) on the GPU using blocking calls on the host for data transfers.
•BLINEMULTI– Same as BLINE when sorting multiple batches on the GPU, and multiway merging on the CPU. Used to compare against the pipelined approaches.
•PIPEDATA– Pipeline the data to/from the device to overlap data transfers and utilize more bidirectional bandwidth over PCIe. This uses pinned memory and CUDA streams.
•PIPEMERGE– Concurrently merge pairs of sorted batches on the CPU, and sorting on the GPU to reduce the overhead of the multiway merge at the end, in addition to PIPEDATA.
•PARMEMCPY– Parallelize memory copies on the host between larger and pinned memory staging buffers.

IV. EXPERIMENTAL EVALUATION

We evaluate the performance of the heterogeneous sorting approaches. We measure the performance impact of each of the optimizations described in Section III and compare overall results with state-of-the-art CPU sorting algorithms.

A. Input Dataset

In previous works, performance is evaluated for a range of input distributions due to the inherent sensitivities of many sorting algorithms to the input characteristics [11]. However, the performance of our hybrid sorting is dominated by memory transfer time, which is independent of input distribution. Also, our approach can use any sorting algorithm

Table II: Details of hardware platforms.

	CPU				GPU			
Platform	Model	Cores	Clock	Memory	Model	Cores	Memory	Software
PLATFORM1	2×Xeon E5-2620 v4	2×8	2.1 GHz	128 GiB	Quadro GP100	3584	16 GiB	CUDA 9
PLATFORM2	2×Xeon E5-2660 v3	2×10	2.6 GHz	128 GiB	2×Tesla K40m	2 × 2880	12 GiB	CUDA 7.5

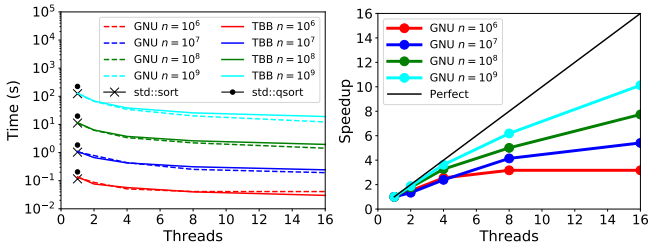
on the GPU, allowing us to use a data-oblivious sorting algorithm if needed. Therefore, we perform all experiments using uniformly distributed datasets and do not evaluate the sensitivity of our approach to the data distribution. We use the 64-bit floating point datatype. This provides a worst-case scenario for our hybrid GPU sort, since GPUs have more capacity dedicated to 32-bit than 64-bit operations [1]. Also, 64-bit elements require more data transfer per operation, further degrading our hybrid approach.

B. Experimental Methodology

We utilize two platforms for our experiments shown in Table II, where PLATFORM2 contains two GPUs. All code is compiled using the GNU GCC host compiler with the O3 optimization flag. We use OpenMP for parallelizing host operations. Results are averaged over 3 trials.

C. Reference Implementation

As a baseline comparison, we benchmark widely used parallel sorting libraries for the CPU. Since our hybrid approach uses libraries for both sorting and merging, using a state-of-the-art CPU sorting library provides a comparable baseline. We compare the performance of our approaches using the GNU library's parallel extensions, previously called the Multi-Core Standard Template Library [19], [20]. The parallel extension library uses OpenMP to specify the number of threads. Figure 4 (a) shows the scalability of the multi-core sorting algorithm (response time vs. the number of threads) on a log scale for 4 different input sizes of n, up to 10^9. We also show the scalability of sorting using the Intel Thread Building Blocks library [21]; however, we find that it is slower than the GNU parallel library for large input sizes, and thus do not use it for our baseline comparison. The sequential C++ STL std::sort and std::qsort are also plotted, with results showing that std::qsort is slower than std::sort by roughly a factor of 2. Furthermore, std::sort and the GNU parallel sort with 1 thread yield nearly identical performance. Figure 4 (b) shows the speedup of the 4 input sizes for the GNU parallel library sort. On PLATFORM1, speedups range from 3.17 ($n = 10^6$) to 10.12 ($n = 10^9$) with 16 threads. Since we sort large inputs, the speedup of the reference implementation is consistent with the larger speedup above. We repeat this experiment on PLATFORM2 with similar results. For our reference implementation, we configure each platform with 16 (PLATFORM1) or 20 (PLATFORM2) threads when executing the STL parallel sort to maximize speedup.



(a) (b)

Figure 4: (a) Sorting scalability on PLATFORM1 using 1–16 threads on log scale. Sequential std::sort and std::qsort are shown. std::qsort is roughly half as fast as std::sort. As expected, std::sort and its parallel variant have nearly identical performance. Larger input sizes yield greater scalability (b).

D. Hybrid Approach Components

We consider the performance two components of our heterogeneous sort: sorting on the GPU and pair-wise merging on the CPU. We measure end-to-end response time including all overheads (i.e., transferring data between CPU and GPU).

1) Sorting on the GPU: To determine the relative performance of sorting on the CPU and GPU, we compare our reference sort using parallel std::sort with end-to-end sorting on the GPU using Thrust. We include all overheads due to memory allocation and data transfers (CPU to GPU and back). Figure 5 shows the performance of sorting on the GPU using BLINE as a function of input size, n, where the memory requirements of sorting n do not exceed the GPU's global memory ($n_b = 1$) on PLATFORM2. Results indicate that, when we include all overheads, sorting on the GPU does not significantly outperform CPU sorting. The ratio of the response time between sorting on the CPU and GPU (red line in Figure 5) is between 1.22 and 1.32 for the input sizes shown. Thus, the GPU yields a respectable performance gain over the parallel reference implementation if batching is not required ($n_b = 1$), and no merging on the CPU is necessary. However, when sorting larger inputs, we must sort the data in batches and merge on the CPU, necessitating more data transfers and potentially degrading performance. We show similar results on PLATFORM1 in a later figure.

2) Pair-wise Merging on the CPU: PIPEMERGE uses this pairwise merge to pipeline merges while the GPU is still sorting batches (Section III-D). Figure 6 plots the (a) response time and (b) speedup up to 16 threads when merging two sorted lists using the GNU parallel mode extensions on

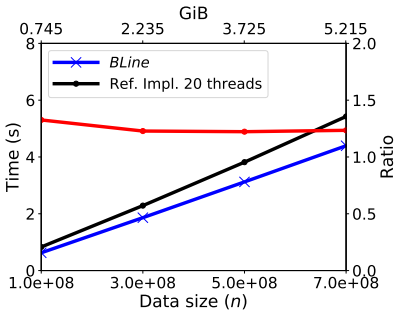
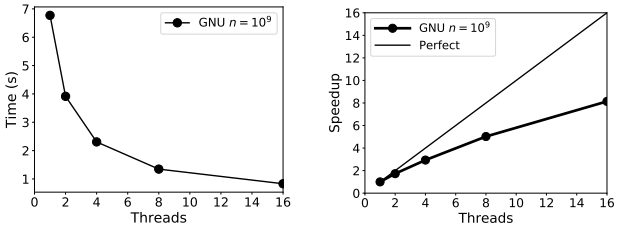


Figure 5: BLINE response time vs. n, with $n_b = 1$ on PLATFORM2. The ratio of the CPU to GPU times are plotted on the right axis.



(a) (b)

Figure 6: Merge scalability using 1-16 threads on PLATFORM1. Each sorted sublist is 0.5×10^9 elements ($n/2$).

PLATFORM1. We only show a single input size as we only merge batches on the order of $n = 10^9$. On 16 cores, the parallel merge achieves a speedup of 8.14×. A moderate speedup is expected, as merging only requires $O(n)$ work, and is memory-bound. Once all batches are sorted, we then perform a final parallel multiway merge using the parallel mode extensions (benchmarks are omitted).

E. Heterogeneous Sorting in the Literature

Recently, Stehle and Jacobsen [5] advanced a GPU radix sort that achieves significant performance gains over state-of-the-art algorithms. The authors apply their radix sort to heterogeneous sorting, whereby data larger than GPU global memory can be sorted using a hybrid approach. In summary, the authors sort sublists of the input list on the GPU and then merge the results using a multiway merge on the CPU (similarly to our approach). The heterogeneous sort in [5] focuses on overlapping data transfers to reduce the bottleneck between the CPU and GPU. While the paper shows very promising results for heterogeneous sorting, the "end-to-end" performance that they present omits several key bottlenecks in the heterogeneous sorting workflow.

In Section 5 of [5], the end-to-end time is computed using the following components: (i) the time to transfer the unsorted sublists from CPU to GPU, (ii) the sorted sublists from GPU to CPU, (iii) the time to sort on the GPU; and, (iv) the time to merge on the host. Since the authors overlap data transfers and computation, we make the following observations.

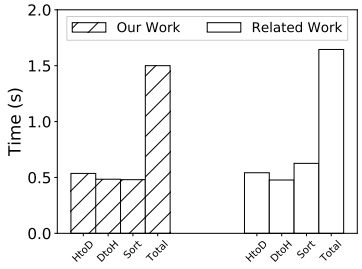


Figure 7: End-to-end performance of sorting 5.96 GiB (our work) and 6 GB (related work) showing *HtoD*, *DtoH*, and *GPUSort* on PLATFORM1. Related work values estimated from the CUB bar plot in Figure 8 of [5].

- To overlap computation and data transfer between the CPU and GPU, CUDA streams must be used which require *pinned memory* (described in Section III-D). This can degrade performance, depending on how much pinned memory is allocated [37].
- A small amount of pinned memory is typically allocated and reused multiple times to amortize the cost of allocation. This is similar to how the driver allocates temporary pinned memory when performing *cudaMalloc*. Thus, data is incrementally transferred to/from the device using this temporary staging area. For example, when copying from the GPU to CPU, a subset of the total data being copied is copied from the GPU into pinned memory on the CPU, and from there it is copied into a larger memory buffer for the CPU (that is typically pageable).
- Since memory copies are asynchronous when copying to/from the GPU using the pinned memory buffer, there is synchronization overhead for each copy.

The end-to-end response time provided in [5] does not include any of the costs and overheads associated with pinned memory. This includes (i) allocating pinned memory, (ii) transferring data from pageable memory to pinned memory (and vise-versa); and, (iii) synchronization time required when using asynchronous memory transfers. Omitting these costs may impact the performance trade-offs of the heterogeneous sort shown in [5]. Furthermore, these overheads may reduce performance, and thus how competitive their algorithm is relative to their reference implementation.

1) Comparison of End-to-End Performance Calculations: The Missing Overhead Problem: Figure 8 in [5] shows the time to perform their end-to-end sort of 6 GB of key/value pairs (375 million 64-bit keys). They show performance for naïve approaches as well as their heterogeneous sorting technique. Their platform consists of a Titan X (Pascal architecture) connected via PCIe. In their plot, Stehle and Jaconsen show that the data transfers *HtoD*, and *DtoH*,

require more time than sorting. Using our platform that is most similar to theirs (PLATFORM1), we reproduce the experiment they use to show their naïve approach: they sort 6 GB of key/value pairs using the CUB [13] sorting library. To reproduce this, we measure the time to transfer 5.96 GiB ($n = 8 \times 10^8$) of data from *HtoD*, and *DtoH*. While we do not sort key/value pairs, we measure the time to sort double precision floats with Thrust, which requires comparable time (i.e., both perform radix sort on a comparable number of elements). We allocate a pinned memory buffer of size $p_s = 10^6$ 8-byte elements for fast memory transfers to and from the GPU. Thus, we incrementally copy a chunk of the unsorted input of size p_s into the pinned memory buffer and then transfer it to the GPU, and similarly, we incrementally transfer the sorted list from the GPU into a pinned memory buffer and then into the final buffer of size n. The host-to-host copy (pageable to pinned, or pinned to pageable) is performed using std::memcpy.

Figure 7 plots the response time for the three time components that comprise the end-to-end time in [5]. Since our total data sizes are roughly equivalent (6 GB vs. 5.96 GiB), the time needed to perform data transfers are consistent with the data transfer times in the "CUB" bar in Figure 8 of [5]. Our *HtoD* and *DtoH* times are 0.536 s and 0.484 s respectively, whereas theirs are 0.542 s and 0.477 s[1]. In summary, our data transfer times are consistent. Since the sorting workloads, algorithms (Thrust vs. CUB), and GPUs (Titan X vs. Quadro GP100) differ, a precise comparison cannot be made, though our results are similar. This shows that the "end-to-end" time in [5] only includes the three runtime components shown in Figure 7.

With a pinned memory buffer of size $p_s = 10^6$, allocation requires 0.01 s; therefore, copying between the smaller pinned memory and larger pageable memory buffers is a more significant overhead. One way to avoid many host-to-host memory copies is to allocate a pinned memory buffer for the entire dataset ($p_s = n$); however, we find that this results in performance loss due to the overhead of allocating so much pinned memory. Allocating a pinned memory buffer of size $p_s = n = 8 \times 10^8$ takes 2.2 s, which is longer than the sum of the time components in Figure 7. Thus, allocating one large pinned memory buffer leads to unacceptable performance.

In Figure 8 we plot the average response time vs. n. Note that the response time in Figure 7 is when $n = 8 \times 10^8$ (≈6 GiB). The input sizes used in the experiments for Figure 8 all fit within GPU global memory on PLATFORM1, so $n_b = 1$ and we only execute one batch (i.e., no merging is required on the host). We measure response time using BLINE, because we only transfer one batch to the GPU, sort, and then return the result. This is the same methodology used by the naïve approach in [5]. Using the method used

[1]Times estimated from Figure 8 in [5].

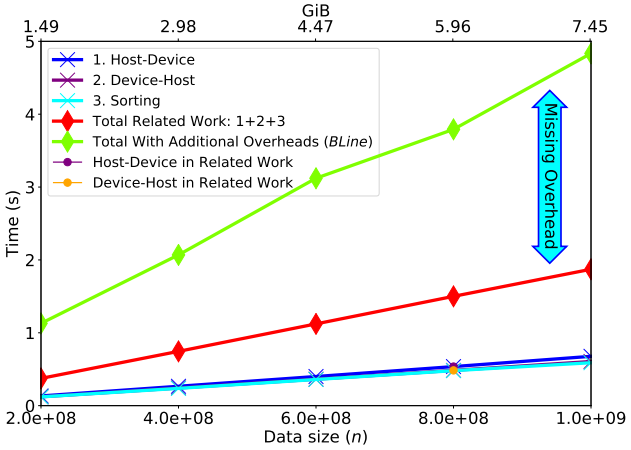


Figure 8: Average response time vs. n for various components of the BLINE sorting approach (i.e., $n_b = 1$) on PLATFORM1. Purple and yellow circles denote data transfer times from the literature [5], which are consistent with our measurements at $n = 8 \times 10^8$.

by [5] to compute end-to-end response time, we obtain the red curve in Figure 8. However, if we include all components that contribute to the runtime of BLINE, we obtain a much larger total response time (the green line in Figure 8). This indicates that *the end-to-end time in [5] omits key overheads*. Hereafter, we provide results using entire end-to-end response time, which includes overheads omitted by [5]. We show how to reduce these overheads through our techniques and optimizations.

F. Sorting Data Larger Than Global Memory

We present the results for heterogeneous sorting of data larger than GPU global memory our two platforms. We endeavor to fill the capacity of main memory on the host (128 GiB on both platforms). However, since our heterogeneous sorting approach requires approximately $3n$ space (described in Section III-C), the maximum input size we consider is roughly one third of the total main memory on the platforms ($n \approx 5 \times 10^9$). We evaluate overall performance using two sets of experiments as follows.

Experiment 1: On PLATFORM1, we employ a large batch size ($b_s = 5 \times 10^8$) and evaluate the performance of the approaches described in Section III-D4. For PIPEDATA, and PIPEMERGE, we set the number of streams $n_s = 2$ so that we can overlap sending and receiving data between the CPU and GPU. Each stream that executes a batch needs its own buffer on the GPU that stores the data to be sorted and sent back to the CPU; furthermore, recall that sorting requires a temporary buffer, thus needing $2b_s$ total space. Therefore, b_s is selected to maximize usage of GPU global memory capacity, while considering n_s and the memory requirements of sorting. Hence, the total memory required

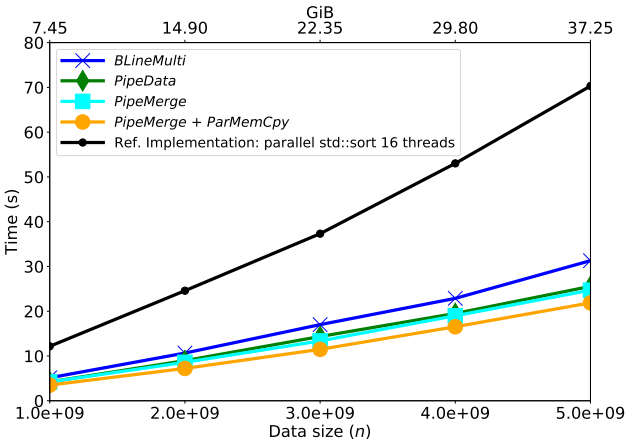


Figure 9: Response time vs. n for our approaches on PLATFORM1.

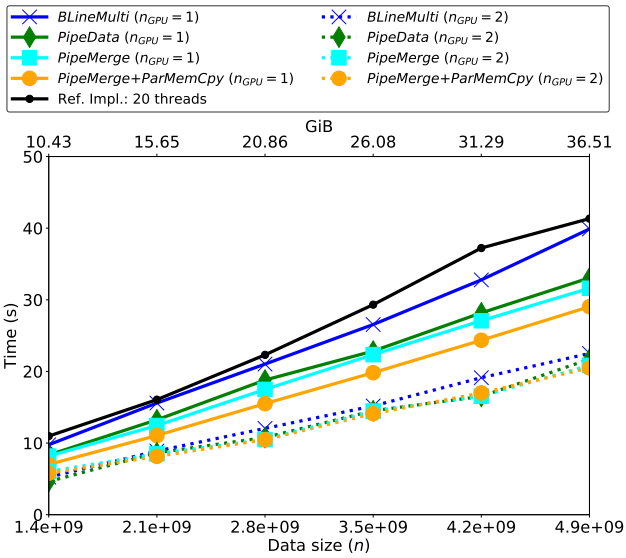


Figure 10: Response time vs. n on PLATFORM2, using 1 (solid lines) and 2 (dashed lines) GPUs.

on the GPU is $\approx 2b_s n_s$. For a fixed value of n, setting $n_s > 2$ may allow for more overlap of data transfers, but this necessitates smaller batch sizes, and thus increased the amount of merging to be done on the CPU.

Figure 9 plots the response time vs. n on PLATFORM1, which has a GP100 GPU (details in Table II). Across all input sizes, our approaches outperform the parallel CPU reference implementation, including BLINEMULTI, which does not overlap data transfers or CPU and GPU execution. At the smallest and largest input sizes $n = 10^9$, and $n = 5 \times 10^9$, we achieve speedups over the reference implementation of $3.47\times$, and $3.21\times$, respectively, using our fastest approach: PIPEMERGE with PARMEMCPY.

Comparing BLINEMULTI to PIPEDATA, we find that pipelining the data transfers improves performance. At $n = 5\times10^9$ BLINEMULTI has an average response time of 31.2 s, while PIPEDATA requires 25.55 s (22% faster).

PIPEMERGE marginally improves the performance over PIPEDATA by merging some of the batches before the final multiway merge. This is an anticipated result, as the final multiway merge requires $O(n\log n_b)$ work, so small changes in the total number of batches does not dramatically impact the performance of the multiway merge. However, if our platform had more memory (and we increase n), the number of batches, n_b, would significantly increase and we would expect the performance of the multiway merge to degrade. We expect that, in this case, pipelining merges would have a more significant impact on overall performance. Comparing PIPEDATA with and without PARMEMCPY, we observe that using PARMEMCPY reduces end-to-end response time by 13%. We attribute this to the following: (i) the host-to-host memory copy operations to/from pinned memory are a bottleneck that can be reduced by parallelizing the operation; (ii) a single core cannot saturate the memory bandwidth of the copy operation, and there is memory bandwidth available, despite performing other concurrent memory-intensive operations; and (iii) the quintessential viewpoint in the literature that *DtoH* and *HtoD* bottlenecks are responsible for poor performance in GPGPU applications may *inadvertently overlook host-side bottlenecks*. For instance, the end-to-end time calculation described in the literature in Section IV-E disregards the host as a bottleneck in heterogeneous sorting. If it was not a bottleneck, then parallelizing host-to-host memory copies would be inconsequential to performance.

Experiment 2: The GPU on PLATFORM2 has less global memory, so we perform all experiments using $b_s = 3.5\times10^8$ (not $b_s = 5 \times 10^8$). We evaluate our sorting approaches for input sizes, n, that are multiples of b_s, so the input sizes we use are not identical across platforms. All of the observations regarding performance on PLATFORM1 above are consistent with single-GPU performance on PLATFORM2, although we are able to show dual-GPU results as well. Figure 10 plots results for both 1 and 2 GPUs. We observe that using two GPUs outperforms all of the single-GPU configurations. At the smallest and largest input sizes $n = 1.4 \times 10^9$, and $n = 4.9 \times 10^9$, we achieve speedups over the parallel CPU reference implementation of $1.89\times$, and $2.02\times$, respectively, using PIPEMERGE with PARMEMCPY.

Comparing single-GPU ($n_{GPU} = 1$) and dual-GPU ($n_{GPU} = 2$) response times, we see that the relative difference between the performance of the approaches when $n_{GPU} = 2$ is smaller than when $n_{GPU} = 1$. We attribute this to the fact that the PCIe bus is shared between both GPUs. Thus, even with BLINEMULTI, when $n_{GPU} = 2$ we are able to saturate more of the PCIe bandwidth and the performance advantage of using two streams is less pronounced in comparison to using two streams when $n_{GPU} = 1$.

G. Lower Bound Performance Analysis

We develop a simple analytical model to determine the efficiency of our methods. This model yields a lower bound on the performance of our approaches. We then compare our results to this lower bound to determine implementation efficiency and identify potential performance improvements.

Lower Limit Baseline Model – 1 GPU: The simplest case is when all of the data can fit in global memory (i.e., unlimited GPU memory). Therefore, the baseline case when $b_s = 1$ provides us with a lower limit on the sorting response time (i.e., peak throughput). We model this case by using the number of sorted elements per second on BLINE where no batching occurs. We measure this lower limit on PLATFORM2 by selecting the response time from Figure 5 when n is large, but fits in GPU global memory ($n = 7 \times 10^8$). We omit modeling results on PLATFORM1.

We develop our model using the BLINE approach because all of the other techniques use multiple streams to overlap data transfers (i.e., $n_s > 1$) and require merging on the CPU. Our simple model is intended to provide us with an estimate of the peak sorting throughput for the simplest case where no merging is required, thus we elect to use a model derived from BLINE. Note, that peak throughput may be outperformed by the approaches that overlap data transfers.

Lower Limit Baseline Model – 2 GPUs: We extend our simple model to include 2 GPUs by assuming the system with 2 GPUs is connected to the host via PCIe and thus we perform a single merge on the CPU ($n_b = 2$). Each GPU sorts $n/2$ elements, so merging the two batches is unavoidable. To compute the peak sorting throughput, we execute BLINE on two GPUs with $n = 1.4 \times 10^9$, $b_s = n/2 = 7 \times 10^8$, and $n_s = 1$. Thus, each GPU sorts one batch and returns it to the host for subsequent merging. The value of n is selected such that the GPU's global memory is nearly at capacity, thus maximizing sorting throughput.

Figure 11 plots the response time vs. n for the 1- and 2-GPU models, and the PIPEDATA results from Figure 10. We compare against PIPEDATA because the other techniques make use of host-side optimizations that are not included in BLINE, from which the models are derived. In both 1 and 2 GPU cases, at $n = 1.4 \times 10^9$ we find that PIPEDATA outperforms the lower limit baseline model (blue curves). This is because overlapping the data in streams leads to a performance improvement despite the overhead of the multiway merge at the end. However, at $n > 2.1 \times 10^9$, we observe that the performance of PIPEDATA begins to degrade as a result of the cost of merging on the CPU after all of the batches have been computed on the GPU. Despite the overhead of the merge phase, at $n = 4.9 \times 10^9$, the slowdown of PIPEDATA in comparison to the model is only 0.93× and 0.88× when $n_{GPU} = 1$ and $n_{GPU} = 2$, respectively. Thus, overlapping data transfers in streams can offset some of the cost of merging on the CPU in both single- and multi-GPU configurations. Note that the slowdown is worse for the 2-GPU system. We attribute this to the PCIe bandwidth being limited and shared between both GPUs. If the bandwidth becomes saturated at various times during the sorting pipeline, per-stream sorting throughput will degrade, causing the observed slowdown.

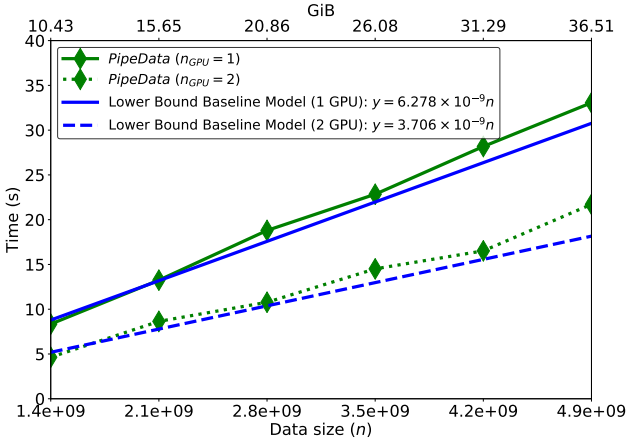


Figure 11: Lower limit baseline models compared to PIPEDATA on 1–2 GPUs on PLATFORM2.

V. DISCUSSION AND CONCLUSIONS

By examining heterogeneous sorting in the literature, we determine that the *HtoD*, and *DtoH* data transfers are not the only non-negligible overheads. There are several host-side bottlenecks that are the result of utilizing bidirectional bandwidth by overlapping data transfers. These overheads are caused by the use of *pinned memory*, which is necessary when overlapping data transfers. Using pinned memory has the added benefit of improved data transfer rates, with throughput improvements of up to a factor ∼2× over copies without pinned memory (e.g., *cudaMemcpy*). Our pinned memory data transfers occur at ∼12 GB/s, which is 75% of the peak PCIe v.3 bandwidth of 16 GB/s.

Our work alleviates host-side bottlenecks by pipelining pair-wise merges before the final multiway merge, and parallelizing the memory copies into and out of the pinned memory staging areas. While BLINE outperforms the reference implementation, the optimizations we propose yield a significant performance improvement over BLINE. On the largest input size, $n = 5 \times 10^9$, (requiring batching), our fastest approach achieves a speedup of 3.21× on PLATFORM1 (1 GPU), compared with the CPU reference implementation.

We show that host-side operations are a bottleneck when exploiting GPUs, including host-to-host memory copies, pipelined pair-wise merging, and final merging on the CPU. While NVLink will reduce the bottleneck between CPU and GPU, other bottlenecks remain or may even worsen. Increasing the GPU sorting and data transfer rates will further imbalance CPU and GPU workloads, thus increasing

the CPU merging bottleneck. Sorting in the NVLink era using multi-GPU systems needs to address the problem of merging using the GPUs, such that the CPU does not need to carry out all merging tasks.

ACKNOWLEDGMENT

We thank University of Hawaii HPC for platform use.

REFERENCES

[1] "Nvidia volta," http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, accessed: 2017-12-11.

[2] C. Kim, J. Park, N. Satish, H. Lee, P. Dubey, and J. Chhugani, "Cloudramsort: fast and efficient large-scale distributed ram sort on shared-nothing cluster," in *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 2012, pp. 841–850.

[3] H. Casanova, Y. Robert, F. Vivien, and D. Zaidouni, "On the impact of process replication on executions of large-scale parallel applications with coordinated checkpointing," *Future Generation Computer Systems*, vol. 51, pp. 7–19, 2015.

[4] S. Mittal and J. S. Vetter, "A survey of methods for analyzing and improving gpu energy efficiency," *ACM Computing Surveys*, vol. 47, no. 2, p. 19, 2015.

[5] E. Stehle and H.-A. Jacobsen, "A memory bandwidth-efficient hybrid radix sort on gpus," in *Proc. of the 2017 ACM Intl. Conf. on Management of Data*, 2017, pp. 417–432.

[6] R. Sedgewick and K. Wayne, *Algorithms, 4th Edition.* Addison-Wesley, 2011.

[7] T. Geller, "Supercomputing's exaflop target," *Commun. ACM*, vol. 54, no. 8, pp. 16–18, Aug. 2011.

[8] S. Lee and J. S. Vetter, "Early evaluation of directive-based gpu programming models for productive exascale computing," in *Proc. of the Intl. Conf. on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12, 2012, pp. 23:1–23:11.

[9] J. JáJá, *An Introduction to Parallel Algorithms*. Addison Wesley Longman Publishing Co., Inc., 1992.

[10] O. Polychroniou and K. A. Ross, "A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort," in *Proc. of the 2014 ACM Intl. Conf. on Management of Data*, 2014, pp. 755–766.

[11] M. Cho, D. Brand, R. Bordawekar, U. Finkler, V. Kulandaisamy, and R. Puri, "PARADIS: An Efficient Parallel Algorithm for In-place Radix Sort," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1518–1529, Aug. 2015.

[12] D. G. Merrill and A. S. Grimshaw, "Revisiting sorting for GPGPU stream architectures," in *Proc. of PACT*, 2010, pp. 545–546.

[13] D. Merrill, "Cub: Cuda unbound," 2015. [Online]. Available: http://nvlabs.github.io/cub/

[14] H. Casanova, J. Iacono, B. Karsin, N. Sitchinava, and V. Weichert, "An efficient multiway mergesort for GPU architectures," *CoRR*, 2017.

[15] R. Cole, "Parallel merge sort," *SIAM Journal on Computing*, vol. 17, no. 4, pp. 770–785, 1988.

[16] J. H. Reif, *Synthesis of Parallel Algorithms*, 1st ed. Morgan Kaufmann Publishers Inc., 1993.

[17] M. H. Nodine and J. S. Vitter, "Deterministic distribution sort in shared and distributed memory multiprocessors," in *Proc. of the Fifth ACM Symp. on Parallel Algorithms and Architectures*, 1993, pp. 120–129.

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

[19] J. Singler, P. Sanders, and F. Putze, *MCSTL: The Multi-core Standard Template Library*. Springer, 2007, pp. 682–694.

[20] J. Singler and B. Konsik, "The gnu libstdc++ parallel mode: Software engineering considerations," in *Proc. of the 1st Intl. Workshop on Multicore Software Engineering*, 2008, pp. 15–22.

[21] J. Reinders, *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*, 2007.

[22] N. Bell and J. Hoberock, "Thrust: a productivity-oriented library for CUDA," *GPU Computing Gems: Jade Ed.*, 2012.

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

[24] B. Schling, *The Boost C++ Libraries*. XML Press, 2011.

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

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

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

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

[29] J. Enmyren and C. W. Kessler, "Skepu: A multi-backend skeleton programming library for multi-gpu systems," in *Proc. of the Fourth Intl. Workshop on High-level Parallel Programming and Applications*, 2010, pp. 5–14.

[30] J. A. Stuart and J. D. Owens, "Multi-gpu mapreduce on gpu clusters," in *Proc. of the 2011 IEEE Intl. Parallel & Distributed Processing Symp.*, 2011, pp. 1068–1079.

[31] C. Gregg and K. Hazelwood, "Where is the data? why you cannot debate cpu vs. gpu performance without the answer," in *Proc. of the IEEE Intl. Symp. on Performance Analysis of Systems and Software*, 2011, pp. 134–144.

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

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

[34] A. Koike and K. Sadakane, "A novel computational model for GPUs with applications to efficient algorithms," *Intl. Journal of Networking and Computing*, vol. 5, no. 1, pp. 26–60, 2015.

[35] H. Peters, O. Schulz-Hildebrandt, and L. Norbert, "Fast In-Place Sorting with CUDA Based on Bitonic Sort," in *PPAM*, 2009, pp. 403–410.

[36] Y. Ye, Z. Du, D. A. Bader, Q. Yang, and W. Huo, "GPUMemSort: A high performance graphics co-processors sorting algorithm for large scale in-memory data," *GSTF Journal on Computing*, vol. 1, no. 2, 2011.

[37] N. Wilt, *The cuda handbook: A comprehensive guide to gpu programming*. Pearson Education, 2013.

[38] S. G. Akl, *Parallel Sorting Algorithms*. Academic Press, Inc., 1990.