A Hybrid CPU/GPU Approach for Optimizing Sorting Throughput

Michael Gowanlock[a,*], Ben Karsin[b,c]

[a] *School of Informatics, Computing, & Cyber Systems, Northern Arizona University, Flagstaff, AZ, U.S.A.*
[b] *Department of Information and Computer Sciences, University of Hawaii at Manoa, Honolulu, HI, U.S.A.*
[c] *Computer Science Department, Université Libre de Bruxelles, Brussels, Belgium*

Abstract

The GPU is an effective architecture for sorting due to its massive parallelism and high memory bandwidth. However, for input datasets that exceed global memory capacity, the communication overhead between host (CPU) and GPU may degrade the overall performance of heterogeneous approaches. Thus, to achieve performance gains over multi-core parallel CPU algorithms, heterogeneous sorting using the GPU needs to obviate communication overheads. We provide a detailed overview of current host-GPU data transfer mechanisms and advance several methods of mitigating the associated performance bottlenecks. Using these methods, we develop a heterogeneous CPU/GPU sorting algorithm that effectively exploits the architecture. Furthermore, we demonstrate that, while out-of-place GPU sorting achieves the best performance, an in-place sort has the potential to further reduce some host-side bottlenecks, which encourages several future research priorities. Our approaches mitigate several bottlenecks, as demonstrated on single- and dual-GPU platforms, achieving speedups up to 3.47× over the parallel reference implementation on the CPU. We discuss future research for heterogeneous sorting in the multi-GPU NVLink era.

Keywords:
GPGPU, Heterogeneous Architectures, Sorting

1. Introduction

Graphics processing units (GPUs) provide a high degree of computational throughput for relatively low cost and power consumption. However, to effectively use the resources of a GPU, data must first be transferred from the host (CPU), and this data transfer is frequently a performance bottleneck for General Purpose Computing on Graphics Processing Units (GPGPU). Currently, this data transfer is performed over PCIe v.3, with a peak aggregate data transfer rate of 32 GB/s. The new NVLink technology improves this to a maximum of 300 GB/s [1]. Nevertheless, this communication overhead is likely to remain a bottleneck for many applications that have large input or output sets, and this bottleneck will be exacerbated on multi-GPU systems, where the GPUs compete for bandwidth.

The high computational throughput and energy efficiency makes GPUs attractive resources for clusters. As such, many large-scale clusters are able to reduce the number of nodes by increasing the computational throughput of each node by using GPUs. This strategy provides three primary benefits. First, it reduces the network communication overhead between nodes. Second, reducing the number of nodes reduces the occurrence of expensive node failures [2]. Third, the power consumption per unit performance metric (e.g., FLOPS) can be significantly reduced [3]. Overall, the use of GPUs improves scalability and efficiency, enabling the prospect of near-future *exascale* systems.

In this work, we examine the problem of sorting on heterogeneous CPU/GPU systems. Given an unsorted list, A, of n elements in main memory, we wish to generate a sorted output list, B, using the resources

*Corresponding author. Michael Gowanlock, School of Informatics, Computing, and Cyber Systems (Building #90) 1295 S. Knoles Dr., Flagstaff, AZ, 86011, U.S.A. E-mail: michael.gowanlock@nau.edu

of a system containing CPUs and GPU(s). Issues such as achieving a balanced workload between CPUs and GPU(s) and minimizing data transfer overhead must be considered to achieve good performance. Additionally, it can be difficult to efficiently make use of the high computational throughput and massive parallelism of GPUs. For example, to achieve good performance on GPUs, the global memory system must be accessed in blocks (or *coalesced*) [4].[1] Additionally, data transfers between host (CPU) and device (GPU) must be optimized, as data movement is a frequently cited performance bottleneck [6, 7].

In recent years, CUDA has provided mechanisms that simplify the host-GPU data transfer process. However, these mechanisms are not well understood, with recent works making incorrect assumptions [8, 7] or overlooking some associated overheads [9, 10]. As such, it is not clear that any strategy using GPU resources will outperform a multi-core CPU-only approach. We perform an in-depth investigation of sorting on heterogeneous CPU/GPU systems and we find that there is a significant, unavoidable host-GPU data transfer cost that is overlooked by previous works [9]. Our prior work [11] makes the following contributions, which we elaborate on in this paper:

- We advance a heterogeneous sort that utilizes both multi-core CPUs and many-core GPU(s) and 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 critical data transfer bottlenecks cannot be omitted when evaluating the performance of heterogeneous sorting, as they can dominate the system response time.

This comprehensive article extends the prior work with the following contributions:

- We discuss a range of host-GPU data transfer methods and determine the performance impact of each.
- We investigate the memory management mechanism, "unified memory", and demonstrate how, for some workloads, it can significantly reduce performance.
- Out-of-place sorting requires twice as much space as sorting in-place. Thus, we implement and optimize an in-place GPU sorting algorithm to determine how it impacts overall heterogeneous sorting time. We find that, while in-place sorting can potentially reduce the number of batches computed on the GPU (and thus the number of batches to be merged on the CPU), the benefits of current state-of-the-art out-of-place sorting algorithms outweigh the impact of reducing global memory usage. With a more efficient in-place sorting algorithm, however, this approach has the potential to further reduce heterogeneous sorting bottlenecks. Designing a GPU-efficient in-place sorting algorithm, however, remains an open problem.
- We demonstrate that pipelining merging tasks on the host while the GPU is sorting can help mitigate some of the final multiway merge overhead. Pipelining merging tasks becomes increasingly important as the number of sorted batches increase (e.g., when sorting on systems with large main memory capacities).

The paper is organized as follows: Section 2 describes related work on CPU and GPU sorting. In Section 3 we discuss the performance impacts of host-GPU data transfer methods. Section 4 advances the algorithm to sort datasets exceeding global memory on heterogeneous CPU/GPU systems. Section 5 evaluates the proposed algorithm as well as our in-place sorting algorithm. In Section 6 we discuss the future of heterogeneous sorting and conclude the paper.

2. Related Work

Sorting is a well-studied problem due to its utility as an algorithm subroutine [12, 13]. We focus on sorting large datasets using a heterogeneous platform consisting of CPUs and GPUs. We discuss related work focused on parallel sorting algorithms, GPU-efficient sorting algorithms, and in-place sorting algorithms, as the algorithms presented in this work rely on algorithms with these features to achieve high performance on CPU/GPU systems. We discuss work related to host-GPU data transfers in Section 3.

[1] See standard resources [5] for more details on modern GPU architectures.

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 [14] and has been shown to provide the best performance on a range of platforms [15, 16, 4, 17, 18]. When sorting based on an arbitrary comparator function, however, parallel algorithms based on Mergesort [19], Quicksort [20], and Distribution sort [21] are commonly used. Each of these algorithms has relative advantages depending on the application and hardware platform. See standard references [14, 13] for an overview of these algorithms.

Optimized parallel sorting algorithms are available for both CPU [22, 23, 24] and GPU [25, 26, 17]. The Intel Thread Building Blocks (TBB) [24] and MCSTL [22] parallel libraries provide highly optimized parallel Mergesort algorithms for CPUs. Several libraries also provide optimized radix sort algorithms [27, 23]. PARADIS [16] is the fastest on the CPU, but we were unable to obtain the code for evaluation.

In-Place Sorting Algorithms – The restriction that an algorithm operate "in-place" (i.e., without additional space) may add significant difficulty when attempting to achieve peak performance on a specific architecture [13]. As such, the best-performing parallel sorting algorithms (e.g., Mergesort, radix sort) make use of a copy of the input list during the sorting process [14]. Parallel Quicksort [13], however, can be implemented as an in-place algorithm and has been shown to achieve good performance in practice on a range of architectures [28]. Quicksort, however, relies on randomization and exhibits irregular memory access patterns, potentially degrading performance on GPUs [4]. The in-place MSD radix sort has been shown to perform better when sorting primitive datatypes [29]. However, like Quicksort, in-place MSD radix sort is not I/O-efficient[2]. Another class of in-place, parallel sorting algorithms are sorting networks, which have a high degree of parallelism and regular memory accesses [31]. However, to sort n elements, sorting networks typically require a factor $O(\log n)$ more work than more efficient algorithms [3]. To our knowledge, finding a practical parallel sorting algorithm that has a high degree of parallelism, is I/O-efficient, operates in-place, and is work optimal remains an important open problem. Despite this, several works have developed fast in-place sorting algorithms for GPU architectures [33, 34, 35, 36]. No attempts, however, have been made to optimize these algorithms for recent GPU architectures. Furthermore, we were only able to get the code for the "GPU-arraysort" algorithm of Awan and Saeed [33], which is designed to sort a large number of small arrays. Thus, we implement and optimize our own in-place sorting algorithm for the GPU.

GPUs and Hybrid Architectures – GPUs have been shown to yield performance gains over CPUs on many general-purpose applications [37, 38, 39]. Applications that can leverage large amounts of parallelism can realize remarkable performance gains using GPUs [40, 41]. However, memory limitations and data transfers can degrade performance [42].

While sorting on the GPU has received a lot of attention [39, 43, 44, 45, 9, 46], most papers assume that inputs fit in GPU memory and disregard the cost of data transfers to the GPU[4]. In the GPU sorting community, this is reasonable as the overhead of host-GPU data transfers is independent of the sorting technique used on the GPU. The Thrust library [25] provides radix and Mergesort implementations that are highly optimized for GPUs and are frequently used as baselines [44]. Several works outperform Thrust [9, 45, 18], but performance gains are limited and the Thrust library is continually improved. We use Thrust for on-GPU out-of-place sorting, as the impact of using other sorting algorithms will be minimal when including host-GPU transfer costs.

Fully utilizing the CPUs and GPU(s) is a significant challenge due to the data transfer bottleneck [7, 42, 6, 48], which we discuss in detail in Section 3. Consequently, sorting using a hybrid approach has been mostly overlooked. Recently, Stehle and Jacobsen [9] proposed a hybrid CPU/GPU radix sort, showing that it outperforms existing CPU-only sorts. However, we find that this work ignores a portion of the overheads associated with data transfers. In this work and our previous related work [11] we consider all potential performance loss due to data transfers to determine the efficacy of CPU/GPU sorting. Shamoto et al. [6] also consider the issue of sorting on hybrid architectures. However, they focus on distributed-

[2] I/O efficiency is formally defined in the external memory model [30].

[3] The well-known AKS sorting network [32] is work-efficient, but is not practical due to very large constant factors.

[4] Arkhipov et al. [47] provide an extensive survey of sorting algorithms designed for modern GPUs.

memory systems, requiring different performance considerations, including the significant cost of network communication overhead. Nevertheless, similar to the results in our previous work [11], they find that pinned memory allocation can cause significant performance degradation, so they do not explicitly allocate pinned memory buffers.

3. Host-GPU Data Transfer Methods

Transferring data between main memory and the GPU represents a significant challenge when attempting to achieve peak performance on heterogeneous CPU/GPU systems. Modern GPUs have significantly less global memory than RAM available to CPUs, so repeated data transfers are necessary to solve large problems using the GPU. In recent years, this has become more pronounced, with commercial CPU systems having access to large main memory (several TiB), while GPU memory sizes remain much smaller (< 50 GiB).

Despite the importance of fast host-GPU memory transfers, the mechanism on modern GPUs is not well understood. To our knowledge, the only prior work to extensively study host-GPU data transfers is that of Fujii et al. [48]. They provide a thorough overview of the technology associated with host-GPU memory transfers and present custom data transfer methods. While their custom transfer methods provide promising results, it is not clear that they can be easily applied to more recent GPU architectures. As a result of this lack of study, several recent works either rely on incorrect assumptions [8, 7] or ignore the memory transfer cost entirely [9, 10]. Che et al. [8] present a thorough set of CPU/GPU benchmarks, though they overlook data transfer costs by assuming overlapped transfers without considering pinned memory overheads. Ye et al. [7] show that pipelining asynchronous data transfers improves throughput. However, while asynchronous transfers require pinned memory buffers [5], they do not discuss the overheads associated with allocation or host-to-host transfers involving pinned memory. In Section 5 we show that these overheads are significant.

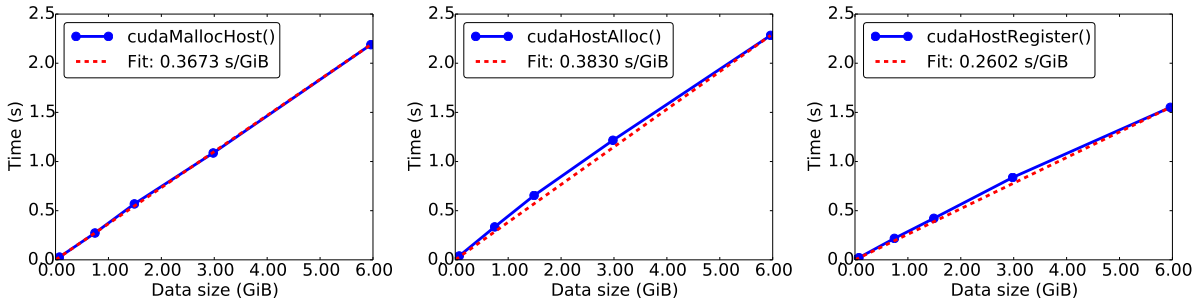
The confusion around host-GPU data transfer performance is compounded by the many mechanisms that are available. While each mechanism has different performance implications, the underlying hardware primitives remain. One primary feature is *pinned* (or page-locked) memory, i.e., memory in RAM that is defined to a locked address that cannot be swapped to disk by the operating system when using virtual memory. Any data that is transferred between main memory and GPU global memory must be in pinned memory. This feature is hidden by many data transfer mechanisms, although it is still a necessity. We discuss the various data transfer mechanisms available in the CUDA API [49].

3.1. Data Transfers and Pinned Memory

The simplest method to perform host-GPU memory transfer is with the `cudaMemcpy()` function. This function is called on the CPU with pointers to an address in main memory and global memory on the GPU, respectively. If the data in main memory is not *pinned*, the function allocates a temporary pinned memory buffer, copies the data to it, and then transfers it to the GPU. Therefore, if pinned memory can be reused, it is beneficial to allocate a pinned memory buffer once and reuse it multiple times, rather than have `cudaMemcpy()` allocate a temporary buffer for each data transfer.

The original method for pinned memory allocation in the CUDA API is `cudaMallocHost()`. The `cudaHostAlloc()` function improves the functionality of `cudaMallocHost()` by allowing allocations for different purposes (e.g., pinned memory shared between CUDA contexts, or mapped to the CUDA address space). Both `cudaMallocHost()` and `cudaHostAlloc()` can be employed when allocating pinned memory as a staging area for copying between the host and GPU. Additionally, there is a zero-copy option, where pageable memory already allocated on the host can be page-locked after allocation (e.g., using `malloc()` or `new`) using `cudaHostRegister()`. This option is useful when the data to be transferred to the GPU has already been allocated on the host in pageable memory, and therefore, avoids a superfluous copy from a pageable to a page-locked buffer.

Figure 1 shows the time to allocate pinned memory using three different methods on Platform1 (detailed in Table 2). We observe that, for all three methods, allocation time increases linearly with the buffer size. As expected, `cudaMallocHost()` and `cudaHostAlloc()` (Figure 1 (a)–(b)) yield the same performance. `cudaHostRegister()` on the system is faster than the previous methods. However, `cudaHostRegister()`



(a) cudaMallocHost() (b) cudaHostAlloc() (c) cudaHostRegister()

Figure 1: Pinned memory allocation overheads using three different allocation functions of the CUDA API [49] on Platform1.

must be performed on an existing dataset, which, in the context of sorting, would involve page-locking the large, unsorted input list, A, which would lead to an unacceptable performance overhead. For instance, when A is 6 GiB, using `cudaHostRegister()` takes 1.56s which was shown in previous work [11] to be longer in duration than the time to sort A on the GPU. Therefore, rather than using `cudaHostRegister()`, we allocate a small pinned memory buffer to be reused multiple times by incrementally transferring the data to/from the GPU. Regardless of the method used, *page-locking memory has an unavoidable allocation cost*, which implies that for a large A, the *entire buffer cannot be allocated as pinned memory*. Therefore, it is important to carefully allocate pinned memory, particularly for applications where there is a low ratio of compute to data transfer time, such as sorting large datasets. Indeed, Shamoto et al. [6] find that allocating large pinned memory buffers reduces the performance of distributed memory sorting on a large CPU/GPU cluster; consequently, they do not allocate large pinned memory buffers.

3.2. Unified Memory

With CUDA version 6, NVIDIA introduced a streamlined memory management system called *Unified Memory*. The Pascal architecture expanded this system with hardware support for page faulting [50] Thus, on Pascal GPUs, managed or unified memory can be allocated at a capacity much larger than the GPU's global memory capacity, and then paged in dynamically when the pages are not resident in global memory on the GPU. This capability streamlines GPU programming, as the user does not need to manually page data between the host and GPU. The user can allocate unified memory using `cudaMallocManaged()`, which is accessible to both the host and GPU using a single pointer. Because unified memory abstracts away the issue of host-GPU memory transfers, it seems to solve the data transfer problem. However, like virtual memory sometimes causing disk accesses [51], some algorithms may result in many host-GPU data transfers.

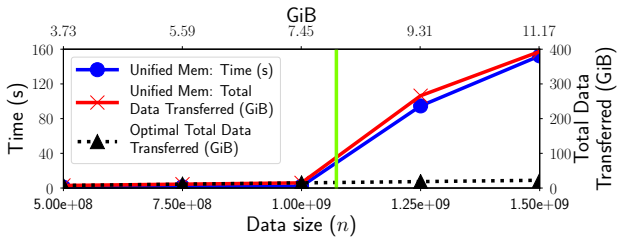


Figure 2: Response time vs. n when using unified memory. The vertical line demarcates where the total memory footprint fits within global memory capacity (to the left of the vertical line). Unified memory performance degrades when the memory footprint exceeds global memory capacity, and causes a significant amount of data to be transferred between host and GPU (right vertical axis). The optimal curve shows the minimum data that is needed to be transferred if the CPU performs merging (n elements to and from the GPU).

As an example, consider sorting a large unsorted list, A, using GPU hardware. One way to accomplish this is to define the memory storing A as unified memory and simply call Thrust radix sort [25] (discussed in Section 2). The page fault system will automatically perform memory transfers as needed and the entire list will be sorted. Figure 2 plots the response time vs. input size when using unified memory on Platform1 (Pascal architecture with 16 GB global memory, detailed in Table 2). Sorting out-of-place (as is done by

Thrust radix sort) requires storing $2n$ elements in global memory. As expected, using unified memory is efficient when $2n$ requires less memory than global memory capacity (the vertical line demarcates the input size that fits within global memory). However, when $n \geq 1.25 \times 10^9$, Thrust radix sort requires more memory than global memory capacity, and performance degrades significantly. This is because only a portion of the data can remain on the GPU, causing many page faults to occur while sorting, as the same data elements must be transferred to and from the GPU many times. We confirm this behavior by measuring the total amount of data transferred to/from the host using the nvprof profiler [5] (the blue curve in Figure 2). If, however, we can perform all sorting on the GPU and all merging on the CPU, we can sort batches that fit in global memory, requiring only $2n$ elements be transferred (black dotted line in Figure 2). We conclude that unified memory may not be practical for all algorithms, such as sorting large datasets and, in Section 4, we present our heterogeneous sorting approach that transfers a minimal amount of data.

4. Heterogeneous Sorting

We present our approach to hybrid CPU/GPU sorting, and use CUDA [5] terminology. We sort on the GPU and merge on the CPU. We divide our unsorted input list A into n_b batches of size b_s to be sorted on the GPU. 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 1 outlines the parameters and notation we use in this work.

Table 1: Table of notation.

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

4.1. Parallel Merging on the CPU

Our general approach is to sort batches on the GPU and merge them all on the CPU. After all n_b batches have been sorted on the GPU and transferred to the host, we use the GNU parallel mode extensions multiway merge. Merging the 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 [52] using a pairwise or multiway merging approach. We find, however, that multiway merge performs better than pairwise merging due to cache-efficiency, so we use it for merging after all batches have been sorted. We also use pairwise merging in one of our pipelining optimizations.

The CPU multiway mergesort is performed out-of-place, so the main memory stores: the unsorted input list, A, the working memory for the sorted batches computed on the GPU, W, and the sorted output list, B, totaling $\sim 3n$ space. The only way to reduce this space complexity on the CPU/host 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 [53, 46], as threads need their own working memory for merging the segments of the sublists. Thus, we employ out-of-place merging to achieve good performance.

4.2. Sorting on the GPU

Thrust (out-of-place) – We use the Thrust library [25] to sort batches on the GPU. High-performance sorting implementations, including Thrust, sort out-of-place, requiring twice the memory of the input list. This doubles both the memory required on the GPU for each batch to be sorted and 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 phase, as it will increase the amount of merging on the CPU (merging takes $O(n \log n_b)$ work).

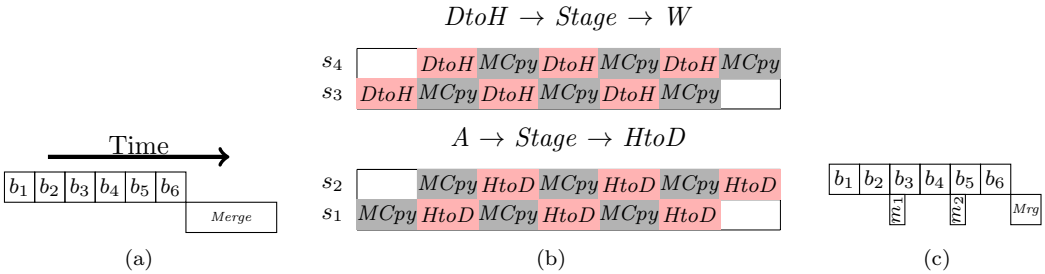


Figure 3: Illustrative examples for our approaches. (a) Example of generating $n_b = 6$ on the GPU and then merging the result with multiway merge on the CPU, if multiway merge requires 3× the time to sort one batch. (b) Pipelining the data transfers in streams, denoted s_1, s_2, s_3, s_4. The pipeline 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. (c) Pipelining merging pairs of sublists on the CPU while the GPU sorts batches. For illustrative purposes, we assume that the time needed to pairwise 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.

Bitonic Mergesort (in-place) – To reduce the amount of merging work executed on the CPU, we explore the use of an algorithm to sort batches *in-place* on the GPU. Current state-of-the-art GPU sorting implementations, such as Thrust, are highly optimized for current architectures and developing an in-place sorting algorithm that is competitive with these libraries remains a difficult open problem. Our goal is to investigate the potential performance benefits of in-place sorting in the context of heterogeneous sorting.

When designing algorithms for the GPU, aspects of the architecture must be taken into consideration. For example, a GPU-efficient algorithm requires a high degree of parallelism as well as coalesced memory accesses [5] (memory must be accessed in blocks). However, thus far there is no practical sorting algorithm that is: (*i*) highly parallel, (*ii*) in-place, (*iii*) has blocked memory accesses (i.e., is I/O-efficient [30]), and (*iv*) is optimal. To determine the impact of in-place GPU sorting on total response time, we design and implement a GPU-efficient sorting algorithm that has the first three of the above features (i.e., is not optimal). One such in-place sorting algorithm is the *Bitonic sorting network* [46]. It is a simple algorithm with a large amount of parallelism and performs blocked memory accesses, though it takes a factor $O(\log n)$ more time than the out-of-place algorithm to sort n elements. In Section 5.5, we provide details of our in-place sorting algorithm and evaluate its impact on performance. We refer to this approach as BITONIC.

4.3. Approaches and Optimizations

Our baseline approach, which we refer to as BLINE, assumes that A fits in memory on a single GPU. We sort A on the GPU by copying the data from the host to the GPU, sorting it, and then transferring 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. See [5] for details on memory transfers and streams.

4.3.1. Baseline With Multiple Batches

Multiple batches need to be sorted on the GPU when $n_b > 1$. This occurs when A exceeds half of global memory on the GPU when sorting out-of-place. Figure 3 (a) shows an example with $n_b = 6$ being sorted on the GPU and then merged on the CPU (using multiway merge). Since merging occurs after all batches have been sorted, there is load imbalance between the CPU and GPU. Figure 3 (a) illustrates a hypothetical example where merging all batches requires 3× the time required to sort a batch on the GPU. While, in the PRAM model, merging on the host requires $O(n \log n_b)$ work (using either pairwise or multiway merge), 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. The baseline approach that uses multiple batches is denoted as BLINEMULTI. It uses blocking data transfers, and the default CUDA stream. The workflow is as follows: $A \rightarrow HtoD \rightarrow GPUSort \rightarrow DtoH \rightarrow W \rightarrow Merge \rightarrow B$.

4.3.2. Pipelining Data Transfers

When sorting data larger than global memory ($n_b > 1$), we can overlap host-GPU data transfers with an optimization we denote as PipeData. 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* that is allocated with *cudaMallocHost* or another method (Section 3.1). Pinned memory buffers are used as a staging area to incrementally copy data from and to the unsorted list, A, and the working memory W, respectively. 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). On multi-GPU systems, the number of batches to sort per stream is $n_b/(n_s n_{GPU})$. The workflow when $n_b > 1$ and PipeData is used is: $A \rightarrow$ *Stage* $\rightarrow$ *HtoD* $\rightarrow$ *GPUSort* $\rightarrow$ *DtoH* $\rightarrow$ *Stage* $\rightarrow W \rightarrow$ *Merge* $\rightarrow B$.

Figure 3 (b) shows pipelining for: (*i*) host-only operations; and (*ii*) data transfers. First, four streams, s_1–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 3 (b), the pinned memory buffer size is $p_s = b_s/3$. Thus, we copy data in either direction using pinned memory as a staging area, allowing for the interleaving of data transfers.

The time to execute the different operations in the pipeline will vary, unlike those in Figure 3 (b). The time spent copying data in/out of the pinned memory staging area on the host (*MCpy*) and data transfers (*HtoD* or *DtoH*) may be unbalanced, affecting the number of streams needed to efficiently overlap these operations. If memory copies in/out of pinned memory are a bottleneck, then more streams can be used for simultaneous data transfers; or, if the data transfers are a bottleneck, then fewer streams should be used. 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.

4.3.3. Pipelining The Merge Phase

While the GPU is sorting batches, the host exclusively performs memory 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, we can decrease the load imbalance shown in Figure 3 (a) 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 of CPU merging work increases. Once two batches are sorted and transferred back to the host, we can perform pairwise merges on the CPU, reducing the amount of work done by the final multiway merge. For instance, in Figure 3 (c), 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.

An important observation is that after the last batch is sorted on the GPU, all pairwise merging should be finished, such that the pairwise merges do not delay the final multiway merge. As such, we utilize two heuristics to compute the number of pairwise pipelined merges to perform: (1) 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 pairwise merged, and when n_b is odd, the last batch is not merged. (2) 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 pairwise merging on the host. Thus, fewer merges can occur before the multiway merge phase, as a function of n_{GPU}.

We do not pipeline merging of any sublists that are the product of a previous pipelined merge (e.g., the output of m_1 in Figure 3 (c)). We find that attempting to pipeline additional merges results in delaying the multiway merging procedure, and thus degrades performance. We refer to our approach that pipelines both pairwise merges and data transfers (Section 4.3.2) as PipeMerge.

4.3.4. Summary of Approaches and Optimizations

BLine is the baseline for sorting a single batch ($n_b = 1$) on the GPU using blocking for data transfers. BLineMulti is the baseline approach when sorting multiple batches on the GPU and performs a single multiway merging on the CPU once all batches are sorted. PipeData uses pinned memory and CUDA

streams to pipeline the data transfers to/from the device to overlap transfers and utilize more bidirectional bandwidth over PCIe. PipeMerge extends PipeData by concurrently sorting on the GPU and merging on the CPU to reduce the overhead of the multiway merge at the end. ParMemCpy additionally parallelizes memory copies on the host to and from the pinned memory staging buffers.

5. Experimental Evaluation

Experimental Methodology – We utilize two platforms for our experiments shown in Table 2, 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. We consider the performance of two components of our heterogeneous sort: sorting on the GPU and pairwise merging on the CPU. We measure the full end-to-end response time including all overheads (e.g., transferring data between CPU and GPU, copying data between buffers). Some of our approaches (described in the previous section) make use of an explicitly allocated pinned memory buffer. For all experiments, unless otherwise noted, we use a pinned memory buffer size $p_s = 10^6$ elements. In Sections 5.2 and 5.3 we show that this buffer size has a negligible allocation cost and is sufficiently large to enable efficient memory transfers.

In prior work that focuses on GPU-only sorting, performance is evaluated for a range of input distributions due to the sensitivities of sorting algorithms to the input characteristics [16]. However, the performance of our hybrid sorting is dominated by memory transfers, which is independent of input distribution. Also, our approach can use any sorting algorithm on the GPU, allowing us to use a data-oblivious sorting algorithm if needed. Therefore, we perform all experiments using uniformly distributed data and do not evaluate the sensitivity of our approach to the data distribution. We use the 64-bit floating point datatype in all experiments. This provides a conservative scenario for our hybrid GPU sort, since modern 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 the performance of our hybrid approach.

Table 2: Details of hardware platforms.

	CPU				**GPU**		
Platform	Model	Cores	Clock	Memory	Model	Memory	Software
Platform1	2×Xeon E5-2620 v4	2×8	2.1 GHz	128 GiB	Quadro GP100	16 GiB	CUDA 9
Platform2	2×Xeon E5-2660 v3	2×10	2.6 GHz	128 GiB	2×Tesla K40m	12 GiB	CUDA 9

5.1. Reference Implementation, Sorting, and Pair-wise Merging Benchmarks

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 (except when using our in-place GPU sorting implementation), using a state-of-the-art CPU sorting library provides a comparable baseline. We compare the performance of our approaches to the GNU library's parallel extension algorithms [22, 23]. The library uses OpenMP to specify the number of threads, and is simply called with the following syntax `__gnu_parallel::sort`$(A, A + n)$. We configure each platform with 16 (Platform1) or 20 (Platform2) threads when executing the parallel sort. For additional detail, see our previous work [11].

CPU vs. GPU Sorting – To determine the relative performance of sorting on the CPU and GPU, we compare the CPU reference implementation to sorting on the GPU with Thrust. We include all overheads due to memory allocation and data transfers. Figure 4 plots the average response time of BLine and the reference implementation on Platform2 and indicates that, when we include all overheads, sorting on the GPU does not significantly outperform the CPU. The response time ratio between sorting on the CPU and GPU is between 1.22 and 1.32 for the input sizes shown. However, when sorting larger inputs we must sort batches on the GPU and merge on the CPU.

Cost of Pairwise Merging – The PipeMerge optimization uses a pairwise merge to pipeline merges while the GPU is still sorting batches (Section 4.3.3). We use the GNU parallel mode extensions to merge two sorted lists using the following syntax `__gnu_parallel::merge`$(b_1, b_1+b_s, b_2, b_2+b_s, m_1)$, where b_1 and

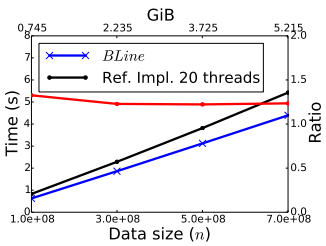


Figure 4: BLine and CPU sorting response time vs. n on Platform2, with ratio plotted on the right axis.

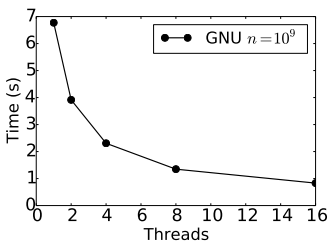


Figure 5: Merge scalability using 1-16 threads on Platform1 with each sorted sublist containing 0.5×10^9 elts. $(n/2)$.

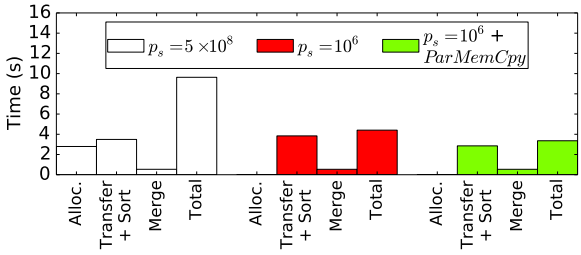


Figure 6: Response time breakdown for $n = 10^9$ and $n_b = 2$ using PipeData showing the time components for: pinned memory allocation, data transfers + sorting, *Merge*, and the total response time on Platform1.

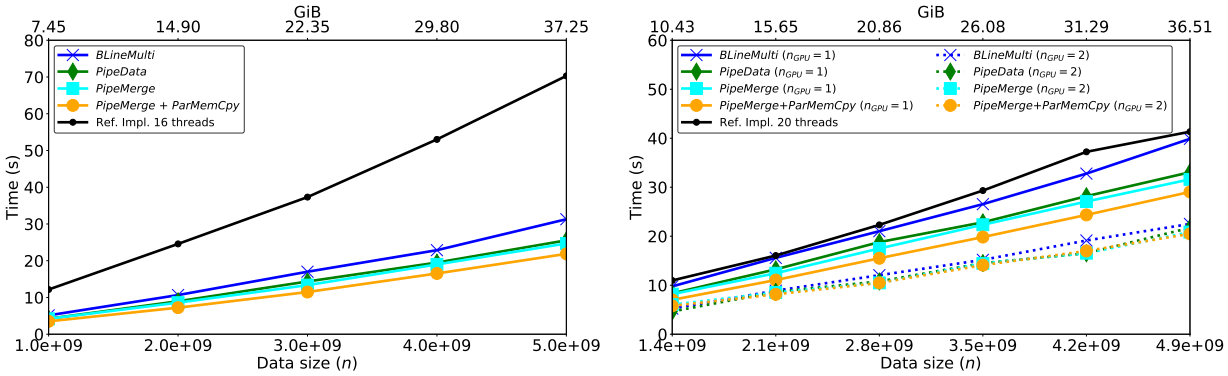
b_2 are sorted batches, b_s is the batch size, and m_1 is an output buffer for the merged result. Figure 5 plots the response time when merging on Platform1. These results show the speedup on up to 16 threads/cores when merging two sorted lists achieves a speedup of 8.14× on 16 cores. A moderate speedup is expected, as merging only requires $O(n)$ work, and is memory-bound. Once all batches are sorted, we perform a final parallel multiway merge using the parallel mode extensions (benchmarks omitted).

5.2. Heterogeneous Sorting in the Literature

Stehle and Jacobsen [9] 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. The authors sort batches 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 [9] focuses on overlapping data transfers to reduce the bottleneck between the CPU and GPU. While the paper shows promising results for heterogeneous sorting, the "end-to-end" performance that they present omits several key bottlenecks. In Section 5 of [9], the end-to-end time is computed using the following components: (*i*) the time to transfer the unsorted batches from CPU to GPU, (*ii*) the sorted batches from GPU to CPU, (*iii*) the time to sort on the GPU; and, (*iv*) the time to merge on the host.

The end-to-end response time provided in [9] 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 vice-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 [9], and thus how competitive their algorithm is relative to their reference implementation. As demonstrated in Section 3, allocating a large pinned memory buffer takes a significant amount of time. Specifically, allocating a buffer of size $p_s = n = 8 \times 10^8$ (5.96 GiB) with `cudaMallocHost()` takes 2.19 s (Figure 1 (a)), which is longer than the sum of the time components in the end-to-end response time of sorting 6 GB of key/value pairs in Figure 8 of [9].

To demonstrate the impact of pinned memory allocation on performance, Figure 6 plots three response time components: pinned memory allocation, transfer + sort, and *Merge*, along with the total time, when using PipeData for sorting $n = 10^9$, where $b_s = n/2 = 5 \times 10^8$, thus $n_b = 2$, and $n_s = 2$. Since the data transfers and computation are overlapped, we cannot report them individually, so they are reported together as "transfer + sort" (specifically, this includes *GPUSort*, *HtoH*, *HtoD* and, *DtoH*). When we allocate a single large buffer ($p_s = n/2 = 5 \times 10^8$, left white bars in Figure 6), we find that the total time to allocate pinned memory is 2.79 s, which is almost as much time as needed to perform data transfers and sort. With a smaller pinned memory buffer ($p_s = 10^6$, middle red bars), the allocation overhead is negligible, though the transfer + sort increases slightly. Applying the ParMemCpy optimization (right green bars) decreases data transfer overheads, making the transfer + sort time even faster than when using a large buffer. Consequently, using a small pinned memory buffer, and performing parallel memory copies is able to obviate the overhead of both pinned memory allocation and the overhead associated with many host-to-host copy operations.



(a) Platform1 (b) Platform2

Figure 7: Response time vs. n for our approaches on (a) Platform1 and (b) Platform2, where 1 GPU (2 GPUs) are shown with solid (dashed) lines.

5.3. Sorting Data Larger Than Global Memory

We present the results for heterogeneous sorting of data larger than GPU global memory on two platforms. We endeavor to fill the capacity of main memory on the host (128 GiB on both platforms in this subsection). Since our heterogeneous sorting approach requires approximately $3n$ space (described in Section 4.1), the maximum input size is roughly one third of the total main memory ($n \approx 5 \times 10^9$).

Single GPU Performance – On Platform1, we employ a large batch size ($b_s = 5 \times 10^8$) and evaluate the performance of the approaches described in Section 4.3.4. For PipeData, and PipeMerge, we set the number of streams $n_s = 2$ so that we can overlap sending and receiving data between the host 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, the total global memory required is $2b_sn_s$. For a fixed value of n, setting $n_s > 2$ may allow for more overlap of data transfers, but this reduces the batch size, increasing the amount of merging to be performed by the CPU.

Figure 7 (a) plots the response time vs. n on Platform1. 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×, and 3.21×, respectively, using our fastest approach: PipeMerge with ParMemCpy. Comparing BLineMulti to PipeData, we find that pipelining the data transfers improves performance. At $n = 5 \times 10^9$ 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. 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. 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 bandwidth available to main memory, 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 3 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.

Dual-GPU Performance – The GPU on Platform2 has less global memory, so we perform all experiments using $b_s = 3.5 \times 10^8$ (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 observations regarding

single-GPU performance on PLATFORM1 are consistent with PLATFORM2. Figure 7 (b) 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×, and 2.02×, 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 $n_{GPU} = 1$.

5.4. Pairwise Merging a Larger Number of Batches

On systems with larger main memory capacities, larger datasets can be sorted, requiring more batches. As the number of batches increase, this also increases the time needed to execute the final multiway merge. In such cases, mitigating some of the multiway merge overhead by pipelining pairwise merges on the host has a more significant impact on overall response time than observed in Figure 7. To verify this, we measure the time to perform the multiway merge phase with PIPEDATA and PIPEMERGE, as we increase the number of batches, n_b. We decrease the size of each batch, b_s, for a fixed input size of $n = 4 \times 10^9$ which increases the number of batches, n_b. The multiway merge phase using PIPEDATA requires 5.07 s ($n_b = 8$) and 15.21 s ($n_b = 400$), whereas these times are 4.56 s and 13.79 s with PIPEMERGE, respectively. Therefore, across a varied number of batches, PIPEMERGE can be used to decrease the final multiway merge overhead.

5.5. In-place GPU Sorting

As mentioned in Section 4.2, high performance sorting libraries sort out-of-place, and thus, on the GPU, we need 2× the space to sort each batch. Consequently, we double the total number of batches that we need to merge in comparison to using an in-place sort. One way to reduce the number of batches is to sort large batches on the GPU using an in-place sort. We design and implement an in-place sorting algorithm for the GPU to understand a potential trade-off between additional work on the GPU and fewer total batches (less CPU work). However, current state-of-the-art GPU sorting libraries are highly optimized and designing a GPU-efficient in-place sorting algorithm remains a difficult open problem. Thus, while we expect existing sorting libraries (e.g., Thrust) to significantly outperform our in-place algorithm, we use our in-place sort to evaluate potential benefits of in-place sorting for our heterogeneous sorting algorithm. The goal is to determine both the relative performance between BITONIC and existing state-of-the-art GPU sorting algorithms, as well as the impact of in-place sorting on our hybrid sort for large input sizes.

Algorithm Overview – Given a batch of size n_b on the GPU, our in-place sorting algorithm, BITONIC, operates as follows. Lists of size $M = 1024$ are sorted using a fast shared-memory sorting algorithm (we use the *shearsort* implementation used by Karsin et al. [18]). Pairs of lists are then merged using a *Bitonic merge network* [31]. Bitonic merging is performed using only compare-and-swap (CAS) operations and is thus completely in-place. For details on the Bitonic mergesort algorithm, see standard references [13, 31]. We note that the bitonic mergesort algorithm is not work-efficient, as it requires a factor $O(\log n)$ more work than optimal. This additional work may significantly degrade GPU sorting performance. However, using this in-place sort, we can sort larger batches, reducing the amount of merging performed on the CPU.

Optimizations – The bitonic mergesort network has some properties that make a large number of GPU-specific optimizations unnecessary.[5] For example, bitonic mergesort is *data oblivious*: the memory access pattern is independent of the input permutation. This results in few branches and low divergence. Additionally, aside from the base case (merging small lists), memory accesses are in contiguous memory, making global memory accesses coalesced. Thus, when merging large lists, standard GPU-specific optimizations will not improve performance. We do, however, optimize the step of sorting or merging small lists. As described above, we sort blocks of size M using the *shearsort* implementation of Karsin et al. [18]. This shearsort operates completely in shared memory and sorts by considering the list of $M = 1024$ elements as a 32×32

[5]See documentation [5] for details on optimization techniques.

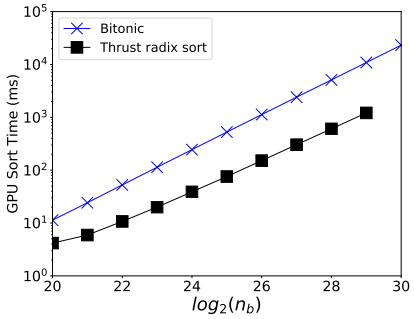


Figure 8: Average time to sort 64-bit floats on PLATFORM2 using in-place and out-of-place sorting algorithms. Only on-GPU sorting time included.

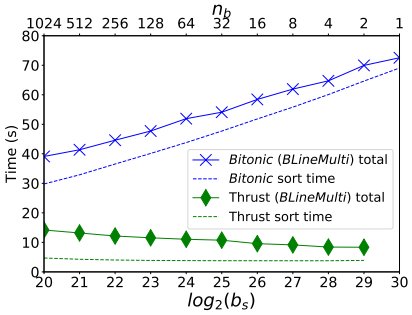


Figure 9: Total response time sorting, along with the time spent just sorting batches on the GPU, for varying batch sizes. Results shown for $n = 2^{30}$ 64-bit floats on PLATFORM2.

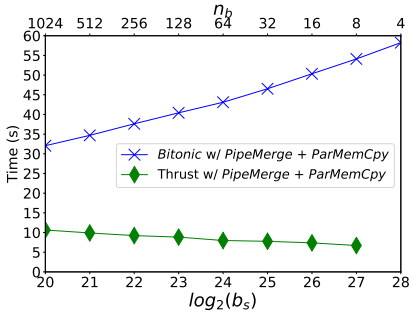


Figure 10: Total response time to sort $n = 2^{30}$ 64-bit floats on PLATFORM2 using BITONIC and Thrust, with both PIPEMERGE and PARMEMCPY.

matrix. Each row of the matrix is sorted in alternating order, and then each column is sorted in ascending order. This process is repeated $\log(32) = 5$ times, and, after a final sorting of rows in ascending order, the list is sorted. For more details on the shearsort algorithm, see [54]. We use several optimizations to improve the performance of shearsort in GPU shared memory. First, each thread sorts a row by loading it into registers and sorting it entirely in registers. Second, to avoid bank conflicts, we transpose the matrix before and after sorting columns. Finally, while each warp of 32 threads sorts a 32×32 matrix, we define thread-blocks of 128 threads and logically divide shared memory to improve GPU occupancy.

GPU Sorting Performance – To determine the practical implications of the work-inefficiency of our in-place GPU sorting algorithm, we compare its performance to that of current state-of-the-art out-of-place GPU sorting algorithms for a range of input sizes. For these experiments, we ignore the overheads associated with data transfers or merging on the CPU, and focus only on the time spent by the GPU to sort a single batch. Due to the inefficiencies discussed above, we expect our in-place sort to be significantly slower than existing out-of-place implementations.

The results in Figure 8 indicate that Thrust significantly outperforms our in-place Bitonic sorting network, BITONIC. This is expected, as Thrust is highly optimized for each NVIDIA GPU architecture. Furthermore, BITONIC is not work-efficient, performing an extra $\log(n)$ work to sort n elements. Thus, overall, on PLATFORM2, BITONIC is between 2.71× and 8.97× slower than the Thrust out-of-place radix sort. However, if the data transfer and CPU merging costs are large enough to mitigate the cost of sorting on the GPU, it may be worthwhile to use a slower in-place sort to reduce the number of batches to be merged on the CPU. Thus, to determine the overall performance of BITONIC, we consider how its in-place nature impacts the total heterogeneous sorting time.

Heterogeneous Sorting Performance – While BITONIC is significantly slower than the sort available in the Thrust library, it is completely in-place, allowing us to sort larger inputs on the GPU. This has the potential to improve the end-to-end performance of heterogeneous CPU/GPU sorting. Thus, we compare end-to-end response time when sorting large inputs using Thrust and BITONIC to sort batches on the GPU, for varying batch sizes. Figure 9 plots both the end-to-end response time, as well as the time spent sorting, for $n = 2^{30}$, on PLATFORM2 for BLINEMULTI. These results indicate that, because BITONIC is slower than Thrust, the end-to-end response time when using BITONIC becomes dominated by time spent sorting on the GPU. Although BITONIC can operate with a larger maximum batch size ($b_s = 2^{30}$) and decrease the time spent merging, the time spent sorting with BITONIC increases at a faster rate. We attribute this to the extra $O(\log n)$ work performed by BITONIC. Thrust is more efficient than BITONIC; therefore, the total response time when sorting with Thrust decreases as we increase the batch size.

As observed in Figure 7, the PIPEMERGE and PARMEMCPY optimizations significantly impact overall performance when sorting large inputs. Therefore, we also compare end-to-end performance when sorting with Thrust and BITONIC while using these optimizations. Figure 10 plots the overall response time when sorting $n = 2^{30}$ elements on PLATFORM2 with PIPEMERGE and PARMEMCPY, for varying batch sizes. The

results are similar to Figure 9, where Thrust outperforms BITONIC. Thus, we conclude that without a faster and more efficient in-place sort for the GPU, we cannot realize the benefits of using larger batch sizes.

6. Discussion and Conclusions

Data transfers between the host and the GPU require the use of a pinned memory address space. Creating and managing this buffer can cause significant overheads, even when it is performed implicitly with unified memory. Explicitly managing pinned memory buffers provides several benefits, such as reduced allocation overhead, overlapping data transfers, and improved transfer rates (a factor $\sim 2\times$ throughput increase over naive transfer methods). In the context of heterogeneous CPU/GPU sorting, we find that effective use of pinned memory allows us to significantly improve sorting performance over a CPU-only approach. While our baseline GPU-only approach, BLINE, outperforms the CPU-only reference implementation, our other optimizations (e.g., pipelining pairwise merges, parallelizing host-host memory copies) significantly improve performance. On the largest input size, $n = 5 \times 10^9$, our fastest approach achieves a speedup of $3.21\times$ on PLATFORM1 (1 GPU), compared with the CPU-only reference implementation.

Merging the sorted batches on the CPU remains a bottleneck. To further mitigate this bottleneck, a sorting algorithm that has a lower memory footprint is needed. While our in-place sorting algorithm, BITONIC, is both highly parallel and I/O efficient, it is not work-efficient. Therefore, out-of-place sorting still yields the best performance, despite the overhead associated with doubling the number of batches that need to be merged. An interesting direction of future work is to develop practical GPU sorting algorithms that achieve a better trade-off between space complexity and work efficiency. Currently, this is an open question, particularly in the parallel algorithms community.

New interconnects such as NVLink [55] may worsen the load imbalance between CPU and GPU tasks. Increasing on-GPU sorting performance and host-GPU data transfer rates will increase the CPU merging bottleneck, as the GPUs will remain idle for a larger fraction of the total end-to-end time. Sorting using multi-GPU systems with newer host-GPU interconnects will need to address merging using the GPUs.

Acknowledgment

We thank the University of Hawai'i for the use of their cluster, UHHPC. This material is based upon work supported by the National Science Foundation under Grants 1533823 and 1745331 and Fonds de la Recherche Scientifique-FNRS under Grant no MISU F 6001 1.

[1] Nvidia volta, `http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`, accessed: Oct. 20, 2018.

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

[3] S. Mittal, J. S. Vetter, A survey of methods for analyzing and improving gpu energy efficiency, ACM Computing Surveys 47 (2) (2015) 19.

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

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

[6] H. Shamoto, K. Shirahata, A. Drozd, H. Sato, S. Matsuoka, Gpu-accelerated large-scale distributed sorting coping with device memory capacity, IEEE Transactions on Big Data 2 (1) (2016) 57–69. `doi:10.1109/TBDATA.2015.2511001`.

[7] Y. Ye, Z. Du, D. A. Bader, Q. Yang, W. Huo, GPUMemSort: A high performance graphics co-processors sorting algorithm for large scale in-memory data, GSTF Journal on Computing 1 (2).

[8] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, K. Skadron, Rodinia: A benchmark suite for heterogeneous computing, in: Proc. of the 2009 IEEE Intl. Symposium on Workload Characterization (IISWC), IISWC '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 44–54. `doi:10.1109/IISWC.2009.5306797`.

[9] E. Stehle, 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.

[10] S. Mittal, J. S. Vetter, A survey of cpu-gpu heterogeneous computing techniques, ACM Comput. Surv. 47 (4) (2015) 69:1–69:35.

[11] M. Gowanlock, B. Karsin, Sorting Large Datasets with Heterogeneous CPU/GPU Architectures, in: 2018 IEEE Intl. Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2018, pp. 560–569.

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

[13] D. E. Knuth, The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching, Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[14] J. JáJá, An Introduction to Parallel Algorithms, Addison Wesley Longman Publishing Co., Inc., 1992.
[15] O. Polychroniou, 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.
[16] M. Cho, D. Brand, R. Bordawekar, U. Finkler, V. Kulandaisamy, R. Puri, PARADIS: An Efficient Parallel Algorithm for In-place Radix Sort, Proc. VLDB Endow. 8 (12) (2015) 1518–1529.
[17] D. Merrill, Cub: Cuda unbound, `http://nvlabs.github.io/cub/` (2015).
[18] B. Karsin, V. Weichert, H. Casanova, J. Iacono, N. Sitchinava, Analysis-driven engineering of comparison-based sorting algorithms on GPUs, in: Proc. of the Intl. Conf. on Supercomputing, ACM, 2018.
[19] R. Cole, Parallel merge sort, SIAM Journal on Computing 17 (4) (1988) 770–785.
[20] J. H. Reif, Synthesis of Parallel Algorithms, 1st Edition, Morgan Kaufmann Publishers Inc., 1993.
[21] M. H. Nodine, 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.
[22] J. Singler, P. Sanders, F. Putze, MCSTL: The Multi-core Standard Template Library, Springer, 2007, pp. 682–694.
[23] J. Singler, 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.
[24] J. Reinders, Intel threading building blocks: outfitting C++ for multi-core processor parallelism, 2007.
[25] N. Bell, J. Hoberock, Thrust: a productivity-oriented library for CUDA, GPU Computing Gems: Jade Ed.
[26] S. Baxter, Modern GPU, `http://nvlabs.github.io/moderngpu/` (2013).
[27] B. Schling, The Boost C++ Libraries, XML Press, 2011.
[28] S. Gueron, V. Krasnov, Fast quicksort implementation using AVX instructions, Comput. J. 59 (1) (2016) 83–90.
[29] F. El-Aker, Fast in-place integer radix sorting, in: Proc. of the Intl. Conf. on Computation Science, 2005, pp. 788–791.
[30] A. Aggarwal, S. Vitter, Jeffrey, The input/output complexity of sorting and related problems, Commun. ACM 31 (9) (1988) 1116–1127.
[31] K. E. Batcher, Sorting networks and their applications, in: Proc. of the Spring Joint Computer Conf., AFIPS '68 (Spring), 1968, pp. 307–314.
[32] M. Ajtai, J. Komlós, E. Szemerédi, An 0(n log n) sorting network, in: Proc. of the Fifteenth Annual ACM Symposium on Theory of Computing, STOC '83, ACM, New York, NY, USA, 1983, pp. 1–9.
[33] M. G. Awan, F. Saeed, Gpu-arraysort: A parallel, in-place algorithm for sorting large number of arrays, in: 2016 45th Intl. Conf. on Parallel Processing Workshops (ICPPW), Vol. 00, 2016, pp. 78–87.
[34] N. Satish, M. Harris, M. Garland, Designing Efficient Sorting Algorithms for Manycore GPUs, in: Proc. of the 2009 IEEE Intl. Parallel & Distributed Processing Symposium, 2009, pp. 1–10.
[35] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, P. Dubey, Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort, in: Proc. of the 2010 ACM SIGMOD Intl. Conf. on Management of Data, 2010, pp. 351–362.
[36] H. Peters, O. Schulz-Hildebrandt, N. Luttenberger, Fast in-place sorting with cuda based on bitonic sort, in: Proc. of the Intl. Conf. on Parallel Processing and Applied Mathematics, 2009, pp. 403–410.
[37] K. Kaczmarski, Experimental B^+-tree for GPU, in: Proc. of ADBIS, Vol. 2, 2011, pp. 232–241.
[38] J. Soman, K. Kothapalli, P. J. Narayanan, Discrete range searching primitive for the GPU and its applications, J. Exp. Algorithmics 17 (2012) 4.5:4.1–4.5:4.17.
[39] O. Green, R. McColl, D. A. Bader, GPU merge path: a GPU merging algorithm, in: Proc. of ICS, 2012, pp. 331–340.
[40] J. Enmyren, 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.
[41] J. A. Stuart, J. D. Owens, Multi-gpu mapreduce on gpu clusters, in: Proc. of the 2011 IEEE Intl. Parallel & Distributed Processing Symp., 2011, pp. 1068–1079.
[42] C. Gregg, 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.
[43] N. Leischner, V. Osipov, P. Sanders, GPU sample sort, in: Proc. of IPDPS, 2010, pp. 1–10.
[44] B. Merry, A performance comparison of sort and scan libraries for GPUs, Parallel Processing Letters 4.
[45] A. Koike, K. Sadakane, A novel computational model for GPUs with applications to efficient algorithms, Intl. Journal of Networking and Computing 5 (1) (2015) 26–60.
[46] H. Peters, O. Schulz-Hildebrandt, L. Norbert, Fast In-Place Sorting with CUDA Based on Bitonic Sort, in: PPAM, 2009, pp. 403–410.
[47] D. I. Arkhipov, D. Wu, K. Li, A. C. Regan, Sorting with gpus: A survey`arXiv:1709.02520`.
[48] Y. Fujii, T. Azumi, N. Nishio, S. Kato, M. Edahiro, Data transfer matters for gpu computing, in: Proc. of the 2013 Intl. Conf. on Parallel and Distributed Systems, ICPADS '13, IEEE Computer Society, Washington, DC, USA, 2013, pp. 275–282.
[49] CUDA API, `https://docs.nvidia.com/cuda/cuda-runtime-api/`, accessed: Oct. 20, 2018.
[50] Nvidia Pascal, `https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf`, accessed: Oct. 20, 2018.
[51] J. L. Hennessy, D. A. Patterson, Computer Architecture, Fifth Edition: A Quantitative Approach, 5th Edition, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011.
[52] R. M. Karp, A survey of parallel algorithms for shared-memory machines, Tech. rep., Berkeley, CA, USA (1988).
[53] S. G. Akl, Parallel Sorting Algorithms, Academic Press, Inc., 1990.
[54] P. Afshani, N. Sitchinava, Sorting and permuting without bank conflicts on GPUs, in: Proc. of ESA, 2015, pp. 13–25.
[55] D. Foley, J. Danskin, Ultra-Performance Pascal GPU and NVLink Interconnect, IEEE Micro 37 (2) (2017) 7–17.