# Engineering Worst-Case Inputs for Pairwise Merge Sort on GPUs

Kyle Berney
*Department of Information & Computer Sciences*
*University of Hawaii at Manoa*
Honolulu, Hawaii, USA
berneyk@hawaii.edu

Nodari Sitchinava
*Department of Information & Computer Sciences*
*University of Hawaii at Manoa*
Honolulu, Hawaii, USA
nodari@hawaii.edu

***Abstract*—Currently, the fastest comparison-based sorting implementation on GPUs is implemented using a parallel pairwise merge sort algorithm (Thrust library). To achieve fast runtimes, the number of threads $t$ to sort the input of $N$ elements is fine-tuned experimentally for each generation of Nvidia GPUs in such a way that the number of elements $E = N/t$ that each thread accesses in each merging round results in a small (empirically measured) number of shared memory contentions, known as *bank conflicts*, while balancing the number of global memory accesses and latency-hiding through thread oversubscription/occupancy.**

**In this paper, we show that for every choice of $E < w$, such that $E$ and $w$ are co-prime, there exists an input permutation on which every warp of $w$ threads of the Thrust merge sort is effectively reduced to using at most $\lceil w/E \rceil$ threads due to sequentialization of shared memory accesses due to bank conflicts. Note that this matches the trivial worst-case bound on the loss of parallelism due to memory contentions for any warp accessing $wE$ contiguous shared memory locations.**

**Our proof is constructive, i.e., we are able to automatically construct such permutation for every value of $E$. We also show in practice that such constructed inputs result in up to ~50% slowdown, compared to the performance on random inputs, on modern GPU hardware.**

***Index Terms*—GPGPU, sorting, bank conflicts, worst-case**

## I. Introduction

Graphics Processing Units (GPUs) are highly parallel, with thousands of cores, supporting hundreds of thousands of threads, with lightweight context switch. For computations that are memory bound, massive multithreading with lightweight context switching capabilities allow GPUs to hide memory latency and achieve runtimes bounded by the memory throughput rather than latency. We assume the reader is familiar with a typical GPU architecture and the standard terminology (global memory, shared memory, bank conflicts, various synchronization tools, thread organization into warps and thread blocks, occupancy, etc). For a good resource, we refer interested readers to [1], [2].

The complexity of GPU architecture (combined with the expectation of fast practical implementations) poses an extra challenge for theoretical research on GPUs. An algorithm with optimal runtime must optimize a number of interdependent parameters, such as multiple levels of memory hierarchy (global memory, shared memory, registers, specialized memories (texture, constant)), each with its own latency and bandwidth, as well as access patterns for optimal performance. Moreover, massive multithreading is managed via hierarchical thread organization, complicating algorithm design and analysis even further.

As a result, a lot of GPU research is empirical rather than theoretical in nature. Such research typically consists of implementing algorithms designed in the classical parallel models, e.g., PRAM, often forgoing theoretical analysis and instead validating via experiments on a number of *random* inputs. Occasionally, GPU implementations will implement algorithms with a focus on optimizing for one or two aspects of the architecture and typically will use heuristics and fine-grain tuning via benchmarking and experimentation to improve the overall runtime. Either way, no claims are typically made about theoretical guarantees about the runtime either in the worst-case or expected case.

Several algorithmic models for GPUs have been introduced as an overall algorithmic model for GPUs [15], [21], [23], [28], [29], yet none of them has been widely accepted. Instead majority of theoretical research on GPUs has focused on optimizing one of the optimization criteria (e.g., global memory accesses, shared memory accesses, register accesses) [4], [7], [14], [18], [19], [21], [29], [33].

While some metrics, such as global memory accesses, can significantly dominate the overall performance of an algorithm (due to its larger latency compared to other memory accesses), for algorithms that heavily utilize other levels of the memory hierarchy, the corresponding metrics cannot always be ignored nor should be hidden in the asymptotic analysis. In fact, it has been shown that for some algorithms, there is a strong correlation between the number of memory contentions in the shared memory, known as *bank conflicts*, and the overall runtime of GPU implementations [13], [19], [33].

Unfortunately, analyzing bank conflicts in shared memory can be difficult, especially for algorithms with data-dependent accesses. One approach taken in the past is to design algorithms that eliminate bank conflicts altogether, known as *bank conflict free* algorithms [4], [7], [18], [20], [33]. As a simple example, Dotsenko et. al [12] observed that for a simple parallel scan it is sufficient to pad the input in shared memory,

This material is based upon work supported by the National Science Foundation under Grants CCF-1533823 and CCF-1911245

such that the number of elements that each thread scans is co-prime with the total number of memory banks.

However, in general bank conflict free algorithms usually come at a price of increased complexity in the resulting algorithms, e.g., more overall work, higher constant factors, more complex code, etc. Instead, it would be ideal to theoretically analyze the number of bank conflicts in the existing simple algorithms, especially for those algorithms which have been shown to perform well in practice.

In general, the following result is easy to prove using the pigeonhole principle:

**Lemma 1.** *Consider a warp of $w$ threads accessing data stored in $k$ consecutive addresses of memory organized into $w$ memory banks, such that bank $i$ contains all addresses $x \equiv i \pmod{w}$. Then there is a set of $w$ (distinct) addresses, access to which will result in $\min\left\{\left\lceil \frac{k}{w}\right\rceil, w\right\}$ bank conflicts.*

Lemma 1 provides a simple worst-case bound on the number of bank conflicts for every parallel access to shared memory. However, depending on the particular algorithm's access pattern in shared memory, this bound may be too pessimistic. In particular, the above result does not consider any dependence between various accesses, which could preclude simultaneous access to the set of addresses defined by Lemma 1. Instead, a tighter analysis of specific algorithms needs to be performed to accurately analyze its performance in shared memory.

This paper takes a step in the direction of addressing this lack of theoretical analysis of bank conflicts in existing algorithms by showing that in the case of the GPU pairwise merge sort algorithm, the bound of Lemma 1 is indeed asymptotically tight.

In Section II, we provide an overview of the GPU pairwise merge sort algorithm, review the model of computation used in our analysis, and present related work. In Section III, we perform an analysis of the worst-case number of bank conflicts incurred by the GPU pairwise merge sort algorithm. Our proof is constructive: we generate the input that causes a provable number of bank conflicts for various software configuration parameters of the algorithm. Then in Section IV, we experimentally evaluate the performance of the constructed worst-case inputs. Our results show a peak slowdown of ~50% and ~40% (compared to the performance on random inputs) on 2 Nvidia GPUs: a Quadro M4000 (compute capability 5.2) and a RTX 2080 Ti (compute capability 7.5), respectively. Lastly, we discuss our conclusions and possible future work in Section V.

## II. Preliminaries

### A. Overview of Pairwise Merge Sort on GPUs

Since our analysis is specific to the details of the GPU merge sort implementation, let us review the details of the algorithm.

The GPU pairwise merge sort algorithm is based on the GPU Merge Path algorithm [14], which is a high-performance implementation of pairwise merging on a GPU.

*a) GPU Merge Path:* Let $A$ and $B$ be two sorted lists such that $|A|+|B| = n$ and let $t$ be the total number of threads. GPU Merge Path is divided into two stages: a partitioning stage and a merging stage. The idea of the partitioning stage is to identify for each thread $i \in \{1, 2, ..., t\}$ the $i$-th quantile ($i$-th group of $n/t$ smallest elements) to be merged by the $i$-th thread during the merging stage independently of other threads. By using the order-statistics of two sorted lists (via mutual binary search), each thread is able to compute the starting location of its quantile in the $A$ and $B$ lists. Then, in the merging stage, each thread performs a sequential merge of $n/t$ elements independently of other threads.

*b) GPU Pairwise Merge Sort:* Let $N$ be the number of elements and $w$ be the number of threads per warp. The implementation uses the following tuning parameters, which are chosen empirically: $b$ is the number of threads per thread block and $E$ is the number of elements that each thread will work on in each merging round, i.e., the total number of threads is chosen to be $N/E$. The parameter $b$ is chosen to be a power of two.

The algorithm starts with the base case where chunks of $bE$ consecutive elements are sorted in shared memory in parallel using $b$ threads per chunk, i.e., each thread block sorts an independent partition of $bE$ elements. In order to do this, each thread first sorts $E$ elements in registers via an odd-even sorting network [32]. Then, each thread block performs a pairwise merge sort using $\log b$ merge rounds, where in each round $i \in \{1, 2, ..., \log b\}$, $(b/2^i)$ pairs of lists, each of size $2^{i-1}E$, are merged via GPU Merge Path using $2^i$ threads.

Once each chunk of $bE$ elements is sorted, $\left\lceil \log \frac{N}{bE} \right\rceil$ pairwise merge rounds are performed, where in each round $i \in \{1, 2, ..., \left\lceil \log \frac{N}{bE} \right\rceil\}$, $2^i$ thread blocks work together to perform a pairwise merge on $2^{i-1}bE$ elements per list. Thus, each thread block needs to find its quantile of $bE$ elements in the two sorted lists. These elements will then be merged by the thread block in shared memory, independently of other thread blocks. Similar to GPU Merge Path, each thread block computes the starting addresses of its quantile in the two sorted lists via a mutual binary search in global memory. Then, the thread block proceeds by performing a single round of GPU Merge Path in shared memory on its $bE$ elements.

Karsin et. al [19] and Karsin [17] perform theoretical analysis of this algorithm by computing the number of parallel coalesced accesses in global memory, denoted $A_g$, and the number of parallel shared memory accesses (with the number of bank conflicts parameterized), denoted $A_s$. We review their results briefly.

Let $P$ be the number of physical cores on the GPU, $\beta_1$ be the average number of bank conflicts per iteration of the mutual binary search (i.e., the partitioning stage), and $\beta_2$ be the average number of bank conflicts per iteration of merging (i.e., merging stage). Then

$$A_g = O\left(\frac{Nw}{PbE}\log^2\left(\frac{N}{bE}\right) + \frac{N}{P}\log\frac{N}{bE}\right)$$

$$A_s = O\left(\frac{N}{PE}\log\left(\frac{N}{bE}\right)(\beta_1 \log bE + \beta_2 E)\right)$$

Karsin et. al [19] found empirically that for Modern GPU on random inputs, $\beta_1 = 3.1$ and $\beta_2 = 2.2$, but also showed that these numbers grow with the number of inversions in the input.

## *B. Distributed Memory Machine Model*

To analyze the number of bank conflicts incurred by each warp throughout GPU merge sort we will use the *Distributed Memory Machine (DMM)* model [24]. The DMM model consists of $w$ synchronous processors and $w$ memory modules. For a memory of size $M$, each of the $w$ memory modules contains an independent partition of size $\lceil M/w \rceil$. Each memory module $i$ contains addresses $x \equiv i \pmod{w}$. Hence, we can view memory as a 2-dimensional matrix of size $w \times \lceil M/w \rceil$, where each row represents a memory module and contiguous address space is laid out in column-major order. In each time step, each processor is able to send a memory request to any of the $w$ memory modules. However, each memory module is only able to respond to a single memory request at a time. Thus, multiple memory requests to a single memory module results in these memory requests being queued in an arbitrary order and processed sequentially. This is known as a *memory contention* and is analogous to bank conflicts in shared memory on GPUs. We consider the concurrent read exclusive write (CREW) DMM, where we allow processors to read the same memory location in the same memory module,[1] but writing to the same memory location is forbidden.

## *C. Related Work*

The DMM model was introduced as early as 1984 by Mehlhorn and Vishkin (initially called the Module Parallel Computer) [24]. Historically, the DMM has been used to study the *granularity of parallel memories* problem, which considers the simulation of PRAM algorithms on the DMM [8]–[11], [16], [24], [27], [34].

The DMM model has been mostly overlooked by the GPU community and has been reinvented with minor variations to model accesses in shared memory. Notably, Dotsenko et al. [12] visualized shared memory as a 2-dimensional matrix; and Nakano [29] formalized this approach with the Discrete Memory Machine model, which also factors in the latency of accessing memory and multi-warp scheduling. Afshani and Sitchinava [4] simplified the Discrete Memory Machine model by removing the latency and considering a single warp, which is equivalent to the DMM model. These minor variations of the DMM have been used to analyze various algorithms such as: scanning [12], sorting [4], [19], searching [18], transposition [7], and permuting [4], [20].

Over the years, various sorting implementations for GPUs have been developed such as: pairwise merge sort [3], [6], [14], [32], [33], multiway merge sort [19], [21], multiway distribution sort [22], shear sort [4], [33], bitonic sort [30], [31] and radix sort [25], [32]. Recent empirical studies have shown that the current state-of-the-art comparison-based sorting implementation on GPUs is the pairwise merge sort implementations available in the Thrust and Modern GPU libraries [19], [26].

Experimental results show that Thrust and Modern GPU perform well on random inputs [19], [26]. However, typical experiments are performed on at most a dozen random inputs with the average runtime reported (often without any mention of variance or other statistics). For the problem of comparison-based sorting, out of $n!$ possible permutations, a random sample of only a dozen inputs represents no statistical significance.

Karsin et al. [19] showed a strong correlation between the number of bank conflicts and the runtime of Modern GPU for a fixed input size of $10^8$ integer elements. Furthermore, the authors constructed so-called *conflict-heavy* inputs, which are inputs that cause a "large" number of bank conflicts, and showed that these inputs increase the runtime of Modern GPU and Thrust, compared to random inputs. Unfortunately, these conflict-heavy inputs were constructed manually for two specific software configuration parameters, the comparison of these conflict-heavy inputs with random inputs is only shown for the GTX 770 (compute capability 3.0), and theoretical analysis of the number of bank conflicts incurred was not investigated and was left as an open problem. This paper addresses this open problem.

## III. Worst-Case Bank Conflict Analysis

Observe that in the bound for the number of parallel shared memory accesses ($A_s$) in Section II-A, the number of accesses in the merging stage is larger than the number of accesses in the partitioning stage when $E \geq \log bE$. In practice, this inequality is satisfied for all values of $E$ and $b$ used in Thrust and Modern GPU [3], [6]. Therefore, we focus on constructing the worst-case input for the merging stage.

Let $w = 2^x$, for some integer $x \geq 0$, be the number of threads in a warp and the number of memory banks in shared memory; let $b = 2^y$, for some integer $y > x$, be the number of threads in a thread-block; and let $E$ be a positive integer.

We consider the *pairwise merge problem*, where two sorted lists $A$ and $B$, each of size $\frac{bE}{2}$, are being merged using $b$ threads of the same thread block. We assume that each thread knows the addresses within $A$ and $B$ from where it will start the merging process (i.e., the partitioning stage of GPU Merge Path has been performed) and it will read the $E$ elements that it is assigned in the increasing order of their values.

*a) Memory alignment:* To simplify our task, we restrict our attention to a simpler problem: maximizing bank conflicts that occur within a fixed set of $E$ *contiguous* memory banks. This is equivalent to a simpler problem of finding a permutation that maximizes the number of threads synchronously scanning elements that are located on the chosen $E$ consecutive memory banks. Since we are generating a worst-case input, this restriction only strengthens our result.

[1]One could also differentiate whether a concurrent read of the same memory location by multiple processors results in a contention. On modern GPUs, it does not. For the purposes of our analysis in this paper, this detail is irrelevant.

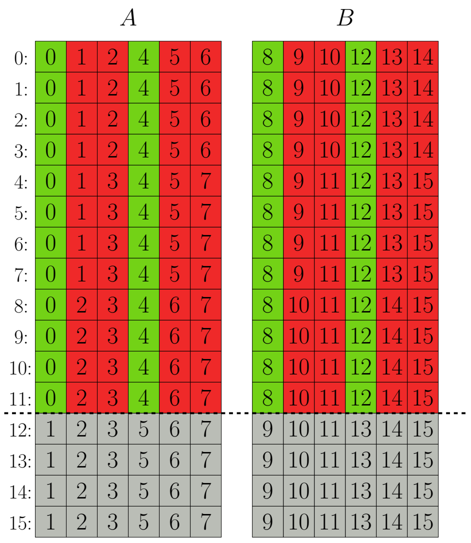

Fig. 1. A depiction of data in sorted order and the resulting access pattern for a single warp, where $w = 16$, $E = 12$, and $\text{GCD}(w, E) = 4$. Rows represent a memory bank in shared memory and elements are marked with the corresponding thread (0-indexed) which reads the particular element. In this case, elements are aligned to the first 12 memory banks, hence, elements colored green located in memory bank $i$ for $i \in \{0, 1, 2, ..., 11\}$ are accessed by its assigned thread in the $i$-th iteration (0-indexed) of the merging stage. While elements colored red indicate misaligned elements, whose assigned thread is not able to access the element in the $i$-th iteration. Elements colored gray indicate elements whose corresponding thread can perform an arbitrary scan of its elements in the $A$ and $B$ list (as they do not contribute towards the number of bank conflicts).

Let $s \in \mathbb{Z}_w$ be the starting memory bank of the chosen $E$ consecutive memory banks.[2] Since execution within a warp is performed in lock-step, we view each merging round as an execution of $E$ steps or, equivalently, $E$ accesses to shared memory. We say an input element $e$ is *aligned* (with respect to the $E$ banks), if $e$ is read in time step $j \in \mathbb{Z}_E$ and is located in bank $(s+j) \pmod{w}$. Since a thread is reading its $E$ elements in increasing order of the values, the alignment is simply a function of the relative order among the $E$ elements being merged by the thread.

*b) Considered values of $E$:* Let $\text{GCD}(w, E) = d$ for some positive integer $d$. Notice that using data in sorted order, every $d$-th chunk of $E$ elements will be aligned. Notably, if $d = E$, i.e., $E$ is a power of 2, then sorted order represents the worst-case input. Figure 1 depicts an example where $d = 4$. Thus, in this work we consider the difficult case where $\text{GCD}(w, E) = 1$, i.e., $E$ is odd.

*c) General Strategy:* Recall from Section II-A that in each merge round, each thread block finds its partition of $bE$ elements to merge across 2 sorted lists, $A$ and $B$; and each thread finds its $E$ elements to merge, within the $bE$ elements of its corresponding thread block. Thus, when constructing our worst-case input, we have the freedom to choose the number of elements in the $A$ and $B$ lists for a thread block, as long as the total number of elements within a thread block equals to $bE$ and the total number of elements in the $A$ and $B$ lists across all thread blocks is equal. Additionally, within a thread block, we have the freedom to choose the number of elements in the $A$ and $B$ lists assigned to a warp, as long as the total number of elements in the $A$ and $B$ lists across all warps in the thread block is consistent with the total number of elements given to that particular thread block.

In order to consider each thread block independently, we design our input so that each thread block is always given $\frac{bE}{2}$ elements from both the $A$ and $B$ lists. For each warp, we decide to give $(\frac{E+1}{2})w$ elements in one list and $(\frac{E-1}{2})w$ elements in the other list. This choice allows us to fix the start of the $A$ and $B$ lists for each warp to the 0-th memory bank, as well as allow us to consider each warp independently (without loss of generalization).

Formally, we partition a thread block into 2 disjoint sets $L$ and $R$, such that $L$ and $R$ both contain $\frac{b}{2w}$ warps. For every warp $l \in L$, we assign $(\frac{E+1}{2})w$ elements of the $A$ list and $(\frac{E-1}{2})w$ elements of the $B$ list. And for warps $r \in R$, we perform the symmetric assignment of $(\frac{E-1}{2})w$ elements of the $A$ list and $(\frac{E+1}{2})w$ elements of the $B$ list.

Ideally, our goal is to generate an input for each warp such that $E$ threads perform a scan of $E$ consecutive elements starting at memory bank $s$. Therefore, we design our input such that every thread performs a scan of one list then the other list, i.e., for some integer $0 \le k \le E$, the first $k$ elements merged belong to one list and the remaining $E - k$ elements belong to the other list. Furthermore, our inputs are generated with a strategy that ensures that for each thread, only elements from a single list will reside within the $E$ consecutive banks, which makes it clear which list to scan first. Thus, we can indirectly describe our input by assigning the number of elements in each list that a particular thread reads. Once again, we note that this restriction only strengthens our result.

In general, our strategy is to assign thread(s) $w - E$ elements in a particular list, which allows the next thread to do a full scan of $E$ elements that are aligned to the $E$ consecutive banks. We consider two cases: "small" $E$ where $E < \frac{w}{2}$ and "large" $E$ where $\frac{w}{2} < E < w$.

## A. "Small" $E$ ($E < \frac{w}{2}$)

As each threads first access (rank $iE$-th element, for $i = 0, 1, ..., w-1$) can reside in a different bank depending on the number of elements assigned in each list to the previous threads, we parameterize the location of the start of the $A$ and $B$ list for a particular thread. Similarly, each threads last access (rank $(iE + E - 1)$-th element) can reside in a different bank depending on the number of elements assigned in each list to the subsequent threads.

For $i = 0, 1, ..., w-1$, let $\alpha_{\uparrow}^{(i)}$ and $\beta_{\uparrow}^{(i)}$ be the number of elements at the start of the $A$ and $B$ list, respectively, that are located before the $E$ consecutive banks for the $i$-th thread in a warp. And symmetrically, let $\alpha_{\downarrow}^{(i)}$ and $\beta_{\downarrow}^{(i)}$ be the number of elements at the end of the $A$ and $B$ list, respectively, that are located after the $E$ consecutive banks. Note that $0 \le \alpha_{\uparrow}^{(i)}, \beta_{\uparrow}^{(i)}, \alpha_{\downarrow}^{(i)}, \beta_{\downarrow}^{(i)} \le w - E$. Moreover, we consider $\alpha_{\uparrow}^{(i)}$ and $\beta_{\uparrow}^{(i)}$ to be undefined if the previous threads

[2]For any integer $c \ge 1$, $\mathbb{Z}_c = \{0, 1, 2, ..., c-1\}$.

have not yet been assigned any elements or if the $i$-th thread starts within the $E$ consecutive banks; and symmetrically for $\alpha_{\downarrow}^{(i)}$ and $\beta_{\downarrow}^{(i)}$.

For some integer $m \geq 1$, we say that the $A$ list has $m$ full columns if $|A| = \alpha_{\uparrow}^{(0)} + (m-1)w + E + \alpha_{\downarrow}^{(w-1)}$, i.e., the $A$ list has $m$ columns which reside in the $E$ consecutive banks. (Similarly defined for the $B$ list.) Figure 2 shows and illustration of the defined parameters for $|A| = tE$, for some positive integer $t$, with $m$ full columns.

**Lemma 2.** *For some positive integer $t$, given $m \geq 1$ full columns of each $A$ and $B$ with $|A|+|B| = tE$, $\alpha_{\uparrow}^{(0)} + \beta_{\uparrow}^{(0)} \geq E$, and $\alpha_{\downarrow}^{(t-1)} + \beta_{\downarrow}^{(t-1)} \geq E$ we can align*

$$\begin{cases} 2mE & \text{if } \textsc{min}(\alpha_{\uparrow}^{(0)}, \beta_{\uparrow}^{(0)}) = \alpha_{\uparrow}^{(0)} \text{ and } \alpha_{\downarrow}^{(t-1)} \geq E \\ 2mE & \text{if } \textsc{min}(\alpha_{\uparrow}^{(0)}, \beta_{\uparrow}^{(0)}) = \beta_{\uparrow}^{(0)} \text{ and } \beta_{\downarrow}^{(t-1)} \geq E \\ 2mE & \text{if } \textsc{min}(\alpha_{\downarrow}^{(t-1)}, \beta_{\downarrow}^{(t-1)}) = \alpha_{\downarrow}^{(t-1)} \\ & \quad \text{and } \alpha_{\uparrow}^{(0)} \geq E \\ 2mE & \text{if } \textsc{min}(\alpha_{\downarrow}^{(t-1)}, \beta_{\downarrow}^{(t-1)}) = \beta_{\downarrow}^{(t-1)} \\ & \quad \text{and } \beta_{\uparrow}^{(0)} \geq E \\ 2mE & \text{if } \textsc{min}(\alpha_{\uparrow}^{(0)}, \beta_{\uparrow}^{(0)}) = \alpha_{\uparrow}^{(0)} \\ & \quad \text{and } \textsc{min}(\alpha_{\downarrow}^{(t-1)}, \beta_{\downarrow}^{(t-1)}) = \beta_{\downarrow}^{(t-1)} \\ 2mE & \text{if } \textsc{min}(\alpha_{\uparrow}^{(0)}, \beta_{\uparrow}^{(0)}) = \beta_{\uparrow}^{(0)} \\ & \quad \text{and } \textsc{min}(\alpha_{\downarrow}^{(t-1)}, \beta_{\downarrow}^{(t-1)}) = \alpha_{\downarrow}^{(t-1)} \\ 2mE - E & \text{otherwise} \end{cases}$$

*total elements (out of $2mE$ total possible).*

*Proof.* (Sketch) The idea of the proof is to apply what we intuitively call the "front-to-back", "back-to-front", and "outside-in" alignment strategies. The general approach in each of these strategies is to align a single column in both the $A$ list and $B$ list and use induction on the remaining $m-1$ columns. In the "front-to-back" alignment strategy, we align the first columns of the $A$ and $B$ lists; and symmetrically in the "back-to-front" strategy, we align the last columns of the $A$ and $B$ lists. The "outside-in" strategy is a synthesis of the "front-to-back" and "back-to-front" strategy, where we align the first column of one list and the last column of the other list. ☐

In all of these alignment strategies, we rely on the fact that $w-E \geq E$ because $E < w/2$, which is the number of memory banks that we do not need to align elements to. To illustrate the importance of this inequality, let us go through an example. Without loss of generality, assume that $\textsc{min}(\alpha_{\uparrow}^{(0)}, \beta_{\uparrow}^{(0)}) = \alpha_{\uparrow}^{(0)}$. In order to align the first column of the $A$ list, we assign $\alpha_{\uparrow}^{(0)}$ elements in $A$ list and $\left(\left\lceil \frac{\alpha_{\uparrow}^{(0)}}{E} \right\rceil E - \alpha_{\uparrow}^{(0)}\right)$ elements in the $B$ list to the first $t_{\uparrow} = \left\lceil \frac{\alpha_{\uparrow}^{(0)}}{E} \right\rceil$ threads (in any order). Since we know that $\alpha_{\uparrow}^{(0)} + \beta_{\uparrow}^{(0)} \geq E$ and $\alpha_{\uparrow}^{(0)} \leq \beta_{\uparrow}^{(0)}$, we are able to perform these assignments without any of the elements overflowing into the $E$ consecutive banks. We are now able to align the next $E$ elements in the $A$ list to the next thread. We now have $\beta_{\uparrow}^{(t_{\uparrow}+1)} = \beta_{\uparrow}^{(0)} - \left(\left\lceil \frac{\alpha_{\uparrow}^{(0)}}{E} \right\rceil E - \alpha_{\uparrow}^{(0)}\right)$ and $\alpha_{\uparrow}^{(t_{\uparrow}+1)} = w - E$. We apply the same strategy to align the first column of the $B$ list as we did with the first column of $A$, where we assign $\beta_{\uparrow}^{(t_{\uparrow}+1)}$ elements in $B$ and $\left(\left\lceil \frac{\beta_{\uparrow}^{(t_{\uparrow}+1)}}{E} \right\rceil E - \beta_{\uparrow}^{(t_{\uparrow}+1)}\right)$ elements in $A$. However to be able to perform these assignments, we need to know that $\alpha_{\uparrow}^{(t_{\uparrow}+1)} + \beta_{\uparrow}^{(t_{\uparrow}+1)} \geq E$ and $\beta_{\uparrow}^{(t_{\uparrow}+1)} \leq \alpha_{\uparrow}^{(t_{\uparrow}+1)}$. Since $\alpha_{\uparrow}^{(t_{\uparrow}+1)} = w - E \geq E$, we are able to guarantee that we meet these two conditions.

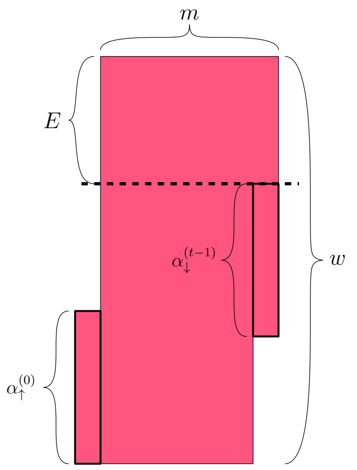

Fig. 2. An illustration of the parameter definitions for $|A| = tE$, for some positive integer $t$, with $m$ full columns. In this example, we want elements to be aligned to the first $E$ memory banks. The parameters for the $B$ list are similarly defined.

**Theorem 3.** *For $E < \frac{w}{2}$, there exists an input with $E^2$ total bank conflicts.*

*Proof.* We choose to align the elements to the first $E$ memory banks, i.e., $s = 0$.

For every warp $l \in L$, we assign $E$ elements in $A$ and $E$ elements in $B$ to the first and second thread, respectively, which aligns the first column of the $A$ and $B$ list. We then use the next $t_{\uparrow} = \left\lceil \frac{w-E}{E} \right\rceil$ threads and assign $(w-E)$ elements in $A$ and $(t_{\uparrow}E - (w-E))$ elements in $B$; and $E$ elements in $A$ to the next thread, which aligns the second column of $A$.

We now have $\alpha_{\uparrow}^{(t_{\uparrow}+3)} = w-E$, $\alpha_{\downarrow}^{(w-1)} = w-E$, $\beta_{\uparrow}^{(t_{\uparrow}+3)} = w-E-(t_{\uparrow}E-(w-E))$, $\beta_{\downarrow}^{(w-1)} = w-E$, and $\left(\frac{E-1}{2} - 1\right)$ full columns of each $A$ and $B$ remaining to align. Notice that $\alpha_{\uparrow}^{(t_{\uparrow}+3)} + \beta_{\uparrow}^{(t_{\uparrow}+3)} \geq E$ and $\alpha_{\downarrow}^{(w-1)} + \beta_{\downarrow}^{(w-1)} \geq E$, thus, we can apply Lemma 2 (using the second, third, or sixth condition) to align the remaining $m = \left(\frac{E-1}{2} - 1\right)$ full columns of each $A$ and $B$. Therefore, in total we have aligned all $3E + (E-3)E = E^2$ elements.

For every warp $r \in R$, we perform the symmetric strategy where we switch $\alpha_{\uparrow}$ & $\beta_{\uparrow}$, $\alpha_{\downarrow}$ & $\beta_{\downarrow}$, and $A$ & $B$. ☐

It follows from Theorem 3 that there does indeed exist an input which can cause the maximum number of bank conflicts

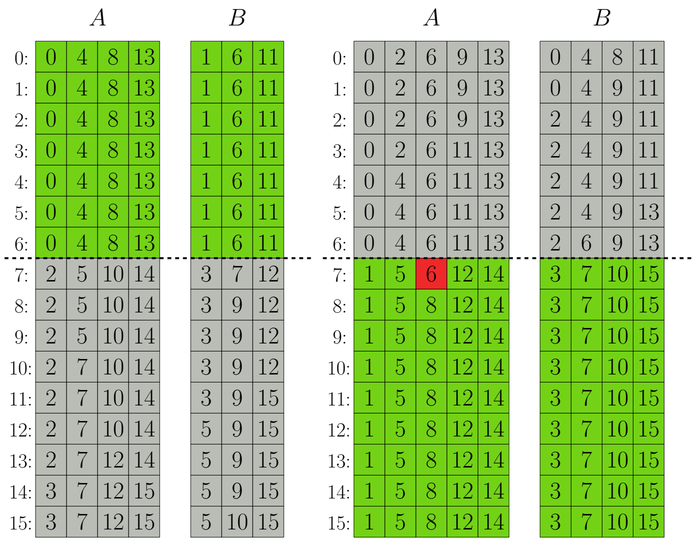

Fig. 3. A depiction of a constructed worst-case input for a single warp with $w = 16$. The left subfigure (first $A$ and $B$ pair) uses $E = 7$ (i.e., "small" $E$ case, where $E < \frac{w}{2}$) and the right subfigure (second $A$ and $B$ pair) uses $E = 9$ (i.e., "large" $E$ case, where $\frac{w}{2} < E < w$). Rows represent a memory bank in shared memory and elements are marked with the corresponding thread (0-indexed) which reads the particular element. In the left subfigure, elements are aligned to the first 7 memory banks, hence, elements colored green located in memory bank $i \in \{0, 1, 2, .., 6\}$ are accessed by its assigned thread in the $i$-th iteration (0-indexed) of the merging stage. In the right subfigure, elements are aligned to the last 9 memory banks, hence, elements colored green located in memory bank $7 + i$ for $i \in \{0, 1, 2, ..., 8\}$ are accessed by its assigned thread in the $i$-th iteration (0-indexed) of the merging stage. Elements colored gray indicate elements whose corresponding thread can perform an arbitrary scan of its elements in the $A$ and $B$ list (as they do not contribute towards the number of bank conflicts).

possible for all values of $E < \frac{w}{2}$, i.e., $\beta_2 = E$. Figure 3 (left subfigure) depicts a constructed worst-case input for a single warp.

### *B. "Large" $E$ ($\frac{w}{2} < E < w$)*

Let $r = w - E < E$ and observe that $r$ is odd, since $w$ is even and $E$ is odd (the difference of an even integer and an odd integer is always an odd integer).

**Lemma 4.** *$\text{GCD}(E, r) = 1$, i.e., $E$ and $r$ are co-prime.*

*Proof.* If $r = 1$, then it is trivial to see that $\text{GCD}(E, 1) = 1$. Thus, we focus on the case where $r \neq 1$.

Let $\text{GCD}(E, r) = d$, for some positive integer $d$. Because both $E$ and $r$ is odd, $E+r = d\left(\frac{E}{d}\right)+d\left(\frac{r}{d}\right)$, where $d$, $\frac{E}{d}$, and $\frac{r}{d}$ are all odd. Let $i$, $j$, and $k$ be integers such that $d = 2i+1$, $\frac{E}{d} = 2j + 1$, and $\frac{r}{d} = 2k + 1$.

$$\begin{aligned} d\left(\frac{E}{d}\right) + d\left(\frac{r}{d}\right) &= (2i+1)(2j+1) + (2i+1)(2k+1) \\ &= 2(2i+1)(j+k+1) = w \end{aligned}$$

Since $w$ is a power of 2, it must be true that both $d = 2i+1$ and $j+k+1$ must also be a power of 2. Therefore, $d = 2i+1$ must be equal to 1 (i.e., $i = 0$). $\square$

Lemma 4 allows us to make use of the following well-known elementary number theory results [5].

**Fact 5.** *Let $a, b, m$ be integers such that $m > 0$ and $\text{GCD}(a, m) = 1$. The linear congruence $ax \equiv b \pmod m$ has exactly 1 solution in $\mathbb{Z}_m$.*

**Fact 6.** *Let $a, m$ be integers such that $m > 0$ and $gcd(a, m) = 1$. The inverse of $a$, denoted $a^{-1}$, exists and is unique modulo $m$.*

We start by defining a sequence and proving some useful properties about it. For $i = 1, 2, ..., E-1$, let $x_i = i(E - r) \pmod E \equiv -ir \pmod E$ and $y_i = ir \pmod E \equiv -i(E - r) \pmod E$.

**Lemma 7.** *For all $i = 1, 2, ..., E - 1$,*

1) $x_i + y_i = E$
2) *For any $j \in \{1, 2, ..., E - 1\}$ such that $i \neq j$, $x_i \neq x_j$ and $y_i \neq y_j$*
3) $x_i = y_{E-i}$

*Proof.* (Proof of 1) We have that $x_i + y_i \pmod E \equiv (-ir \pmod E) + (ir \pmod E) \equiv 0$. Thus, to show that $x_i + y_i = E$, it suffices to show that $x_i + y_i \neq 0$, i.e., both $x_i$ and $y_i$ are never 0. From Lemma 4, we know that $\text{GCD}(r, E) = 1$. Thus, from Fact 5 we know that for $b = 0, 1, ..., E - 1$, $ir = b \pmod E$ has exactly 1 solution in $\mathbb{Z}_E$. Notably, for $y_i = ir \equiv 0 \pmod E$, the solution is $i = 0$ which is not included in our sequence.

(Proof of 2) Let $b_1$ and $b_2$ be integers such that such that $b_1 \not\equiv b_2 \pmod E$ and let $i_1, i_2$ be the respective solutions to the equations $ai_1 \equiv b_1 \pmod E$ and $ai_2 \equiv b_2 \pmod E$. Assume (for the sake of contradiction) that $i_1 \equiv i_2 \pmod E$. Then, from Fact 6 we have that

$$\begin{aligned} i_1 \equiv b_1 r^{-1} \pmod E &\equiv i_2 \pmod E \\ \text{and } i_2 \equiv b_2 r^{-1} \pmod E &\equiv i_1 \pmod E \ , \end{aligned}$$

which implies that $b_1 \equiv b_2 \pmod E$, a contradiction. Thus, it must be true that $i_1 \not\equiv i_2 \pmod E$. Therefore, for each $b = 1, 2, .., E - 1$ there exists exactly 1 solution in $\mathbb{Z}_E$ and the solutions are unique modulo $E$, which implies that each $y_i$ is unique. A similar argument applies to $x_i = (-i)r \equiv 0 \pmod E$.

(Proof of 3) Lastly, we have that $x_i = -ir \pmod E$ and $y_{E-i} = (E - i)r \pmod E \equiv Er - ir \pmod E \equiv -ir \pmod E$. $\square$

**Lemma 8.** *Let $n$ and $m$ be integers such that $0 < n \leq r$ and $0 < m < E - r$. For $i = 1, 2, ..., E - 2$,*

1) $x_i = r - n$ *if and only if* $y_{i+1} = n$
2) $x_i = E - m$ *if and only if* $y_{i+1} = m + r$
3) $x_i + y_{i+1} = \begin{cases} r & \text{if } x_i < r \\ w & \text{if } x_i > r \end{cases}$

*Proof.* (Proof of 1) Assume $x_i = r - n$, then from Lemma 7.1 we know that $x_i + y_i = E \implies y_i = E - r + n$. Hence, $y_{i+1} = y_i + r \pmod E \equiv E - r + n + r \pmod E = n$.

Assume $y_{i+1} = n$, then from Lemma 7.1 we know that $x_{i+1} + y_{i+1} = E \implies x_{i+1} = E - n$. Hence, $x_i = x_{i+1} + r \pmod E \equiv E - n + r \pmod E = r - n$.

(Proof of 2) Assume $x_i = E - m$, then from Lemma 7.1 we know that $x_i + y_i = E \implies y_i = E - E + m = m$. Hence, $y_{i+1} = y_i + r \pmod{E} = m + r$.

Assume $y_{i+1} = m + r$, then from Lemma 7.1 we know that $x_{i+1} + y_{i+1} = E \implies x_{i+1} = E - m - r$. Hence, $x_i = x_{i+1} + r \pmod{E} = E - m$.

(Proof of 3) We first note that $y_1 = r$ (by definition) and from Lemma 7.3 we know that $y_1 = x_{E-1} = r$. Thus, since we know that each value of $x_i$ and $y_i$ are unique (from Lemma 7.2), all values of $x_i$ for $i = 1, 2, ..., E-2$ is either greater than $r$ or less than $r$. Therefore, it follows from the first 2 points of this lemma that

$$x_i + y_{i+1} = \begin{cases} r - n + n = r \\ E - m + m + r = E + r = w \end{cases}$$

□

From Lemma 7 and Lemma 8, we know that there are exactly $(r-1)$ values of $i$ such that $x_i + y_{i+1} = r$ and $(E - r - 1)$ values of $i$ such that $x_i + y_{i+1} = w$.

Let $S = (a_1, b_1), (a_2, b_2), ..., (a_{E-1}, b_{E-1})$ such that

$$a_i = \begin{cases} x_i & \text{if } i \text{ is even} \\ y_i & \text{otherwise} \end{cases}$$

$$b_i = \begin{cases} y_i & \text{if } i \text{ is even} \\ x_i & \text{otherwise} \end{cases}$$

In order to use the sequence $S$ to assign elements to threads, we assign $E$ elements after every sum of $r$ element assigned to either the $A$ list ($a_i$'s) or $B$ list ($b_i$'s). The pairs of elements which sum up to $w$ will cause some misalignment, however, we never misalign all $E$ elements in a column.

Formally, we create a new sequence, denoted $T$, by performing the following modifications to the sequence $S$:

1) Insert $(E, 0)$ after $(a_1, b_1) = (y_1, x_1) = (r, E - r)$ and $(a_{E-1}, b_{E-1}) = (x_{E-1}, y_{E-1}) = (r, E - r)$
2) For $k = 1, 2, ..., \frac{E-1}{2} - 1$, if $a_{2k} + a_{2k+1} = x_{2k} + y_{2k+1} = r$ then we insert $(E, 0)$ after $(a_{2k+1}, b_{2k+1})$.
3) For $k = 1, 2, ..., \frac{E-1}{2}$, if $b_{2k-1} + b_{2k} = x_{2k-1} + y_{2k} = r$ then we insert $(0, E)$ after $(a_{2k}, b_{2k})$.

**Theorem 9.** *For $E > \frac{w}{2}$, we can align a total of $\frac{1}{2}\left(E^2 + E + 2Er - r^2 - r\right)$ elements.*

*Proof.* We choose to align the elements to the last $E$ memory banks, i.e., $s = r$.

For every warp $l \in L$, we use the sequence $T$ to assign the number of elements to assign to each thread.

It follows from Lemma 7.2 and Lemma 8.3 that there are exactly $(r - 1)$ pairs which sum up to $r$ and $(E - r - 1)$ pairs which sum up to $w$ in the sequence $S$. Hence, including $(a_1, b_1)$ and $(a_{E-1}, b_{E-1})$, we have inserted a total of $(r+1)$ tuples of either $(E, 0)$ or $(0, E)$ into $S$ to create $T$. Therefore, $T$ is comprised of $E$ groups of consecutive entries which sum up to $w$, with $\left(\frac{E-1}{2} + 1\right)$ groups in the $A$ list and $\left(\frac{E-1}{2}\right)$ groups in the $B$ list. Moreover, by inserting $E$ after every sum of $r$ elements, we have perfectly aligned that particular column. Hence, we have $(r + 1)$ columns that are aligned perfectly and $(E-r-1)$ columns that are partially misaligned.

To count the number of misaligned elements, we focus on the $(E - r - 1)$ values of $i \in \{1, 2, ..., E-2\}$ such that $x_i + y_{i-1} = w$. Because we know that there are $(E-r-1)$ unique values of $x_i$ such that $x_i > r$, for each of the considered $(E - r - 1)$ values of $i$ we misalign $(x_i - r)$ elements and align $y_i$ elements. As each value of $x_i$ is unique, in total $T$ misaligns $(r - 1 - r) + (r + 2 - r) + ... + (E - 1 - r) = 1+2+...+(E-r-1) = \sum_{k=1}^{E-r-1} k = \frac{1}{2}(E-r)(E-r-1) = \frac{1}{2}\left(E^2 - 2Er - E + r^2 + r\right)$ total elements.

Thus, in total we have aligned $E^2 - \frac{1}{2}\left(E^2 - 2Er - E + r^2 + r\right) = \frac{1}{2}\left(E^2 + E + 2Er - r^2 - r\right)$ total elements.

For warps $r \in R$, we use the symmetric strategy, where we use the sequence $T$ with tuple values switched. □

For $E = \frac{w}{2} + 1$ (i.e., the minimum value of $E$ for this case), we have that $r = E - 2$. Hence, $\frac{1}{2}\left(E^2 + E + 2Er - r^2 - r\right) = E^2 - 1$. Similarly, for $E = w - 1$ (i.e., the maximum value of $E$ for this case), we have that $r = 1$. Thus, $\frac{1}{2}\left(E^2 + E + 2Er - r^2 - r\right) = \frac{E^2}{2} + \frac{3}{2}E - 1$. Therefore, we have that $\frac{1}{2}\left(E^2 + E + 2Er - r^2 - r\right) = \Theta(E^2)$ and we can achieve the asymptotic worst-case number of bank conflicts, i.e., $\beta_2 = \Theta(E)$. Figure 3 (right subfigure) depicts a constructed worst-case input for a single warp.

### C. Putting it all together

We have showed that for values of $E$ such that $\text{GCD}(w, E) = 1$, $E < \frac{w}{2}$ results in $E^2$ total bank conflicts; and $\frac{w}{2} < E < w$ results in between $\frac{E^2}{2}$ and $E^2$ bank conflicts, depending on the value of $E$. Since each warp performs $wE$ work in total (per merge round), this effectively reduces the parallelism from $w$ threads down to $\left\lceil \frac{w}{E} \right\rceil$ threads, i.e., the parallel time is increased from $\Theta\left(\frac{wE}{w}\right) = \Theta(E)$ up to $\Theta\left(\frac{wE}{w/E}\right) = \Theta(E^2)$, which is the worst-case possible. Although for values close to $w$, we are a factor of 2 off from the absolute worst-case (considering leading constants), this still effectively reduces the parallelism from $w$ threads down to 2 threads per warp.

Notice that for small values of $E$ (i.e., $E < \frac{w}{2}$), while we are able to achieve the worst-case of $E^2$ total bank conflicts per warp, the total number of bank conflicts is at most $\frac{w^2}{4}$ as $E$ approaches $\frac{w}{2}$. In contrast, for large values of $E$ (i.e., $\frac{w}{2} < E < w$), the total number of bank conflicts per warp converges towards $\frac{E^2}{2} = \frac{w^2}{2}$ bank conflicts as $E$ gets closer to $w$.

Overall, it may seem that choosing a small value of $E$ is best for performance, as the worst-case does not penalize the possible parallelism as much as the worst-case of a larger $E$ value. However, decreasing $E$ increases the number of binary searches that needs to be performed in the global memory partitioning stage (i.e., partitioning elements to thread blocks). Thus, a larger $E$ value is desired to decrease the amount of work performed in global memory. Therefore, an $E$ value which balances these factors seems to be the best choice.

## IV. Experimental Results

### A. Methodology

We perform our experiments on 2 Nvidia GPUs: a Quadro M4000 (compute capability 5.2), which contains 1664 physical processors across 13 SM's, 8 GB of global memory, and 96 KiB of shared memory per SM; and an RTX 2080 Ti (compute capability 7.5), which contains 4352 physical processors across 68 SM's, 11 GB of global memory, and 96 KiB of unified L1 cache and shared memory per SM (configured at runtime to be either 32 KiB of L1 cache and 64 KiB of shared memory, or vice versa)[3].

We use the Thrust library included with the CUDA 10.1 toolkit, which defines the software parameters of $E = 15$ and $b = 512$ for the Quadro M4000. However, the software parameters are not explicitly defined for the RTX 2080 Ti and by default it uses the parameters defined for compute capability 6.0, which is $E = 17$ and $b = 256$. For these software parameters, each thread block requires 17 KiB of shared memory space, thus, 3 thread blocks (768 total threads) using a total of 51 KiB of shared memory space (13 KiB unused) can be resident on each SM. Compared to $E = 15$ and $b = 512$, each thread block uses 30 KiB of shared memory space, which results in 2 resident thread blocks (1024 total threads) using a total of 60 KiB of shared memory space (4 KiB unused). As the RTX 2080 Ti can support up to 1024 resident threads per SM, the latter parameters provides 100% theoretical occupancy while the former parameters provides only 75% theoretical occupancy. Therefore, we expect $E = 15$ and $b = 512$ to outperform $E = 17$ and $b = 256$. In our experiments we use both of these parameters for the RTX 2080 Ti.

The Modern GPU library defines $E = 15$ and $b = 128$ for the Quadro M4000 and, similar to Thrust, does not explicitly define parameters for the RTX 2080 Ti. Hence, we run experiments using the same two sets of parameters as in our Thrust experiments.

All experiments are performed on 4-byte integers with the average over 10 runs being reported. Runtimes are recorded using `cudaEventRecord` and bank conflict counts are gathered via Nvidia's provided profilers. Specifically, for the RTX 2080 Ti, `nv-nsight-cu-cli` is used to record the `l1tex__data_bank_conflicts.sum` metric. The test harness program for Thrust is compiled with the `-O3` optimization flag and the test harness program for Modern GPU is compiled with its provided Makefile.

### B. Results

Figure 4 shows both the Thrust and Modern GPU throughput results for their respectively defined software parameters on the Quadro M4000. We find that the constructed worst-case inputs cause a peak slowdown of 50.49% (occurring at 7,864,320 elements) and 33.82% (occurring at 62,914,560 elements) for Thrust and Modern GPU, respectively. Overall, we have an average slowdown of 43.53% and 27.3% for Thrust

[3]GB $= 10^9$ B and KiB $= 2^{10}$ B

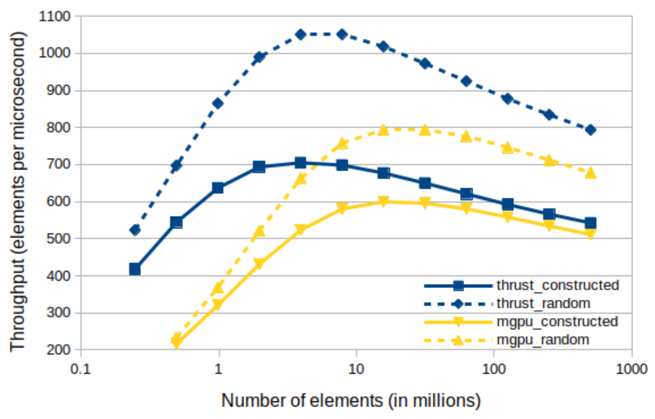

Fig. 4. Throughput results for both Thrust and Modern GPU on the Quadro M4000. Thrust results are in blue and Modern GPU results are in yellow. The solid lines represent the constructed worst-case inputs and the dashed lines represent random inputs. The $x$-axis is on a logarithmic scale.

and Modern GPU, respectively. Moreover, as expected, Thrust outperforms Modern GPU for both random and constructed worst-case inputs.

Figure 5 shows the throughput results for both software parameters in Thrust and Modern GPU on the RTX 2080 Ti. We find that for $E = 15$ and $b = 512$, the constructed worst-case inputs cause a peak slowdown of 42.43% (occurring at 31,457,280 elements) and 42.62% (occurring at 3,932,160 elements) for Thrust and Modern GPU, respectively. The average slowdown is 33.31% and 35.25% for Thrust and Modern GPU, respectively. For $E = 17$ and $b = 256$, the constructed worst-case inputs cause a peak slowdown of 22.94% (occurring at 35,651,584 elements) and 20.34% (occurring at 285,212,672 elements) for Thrust and Modern GPU, respectively. The average slowdown is 16.54% and 12.97% for Thrust and Modern GPU, respectively.

On the RTX 2080 Ti, results from both Thrust and Modern GPU confirm that for random inputs, $E = 15$ and $b = 512$ provide increased performance over $E = 17$ and $b = 256$. However, it is interesting that for the constructed worst-case inputs, the opposite is true: $E = 17$ and $b = 256$ outperforms $E = 15$ and $b = 512$. This results in a much larger slowdown for $E = 15$ and $b = 512$ compared to $E = 17$ and $b = 256$. To investigate this, we compare the runtime per element and the bank conflicts per element for both software parameters (Figure 6 shows this comparision for Thrust). We find that the relative performance of the number of bank conflicts per element predicts the relative performance of the runtime per element. In other words, there is indeed a correlation between the runtime and the number of bank conflicts. Moreover, as we expect, the number of bank conflicts per element shows logarithmic growth; and while there is some noise from the base case, we also see logarithmic growth in the runtime per element.

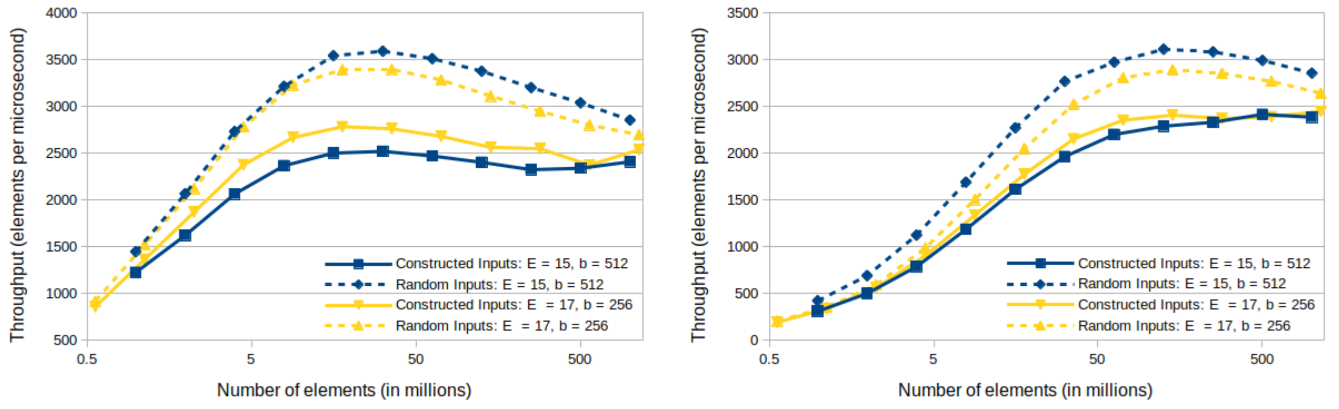

Fig. 5. Throughput results for Thrust (left) and Modern GPU (right) on the RTX 2080 Ti. The blue lines represent parameters $E = 15$ and $b = 512$ and the yellow lines represent the parameters $E = 17$ and $b = 256$. The solid lines represent the constructed worst-case inputs and the dashed lines represent random inputs. The $x$-axis is on a logarithmic scale.

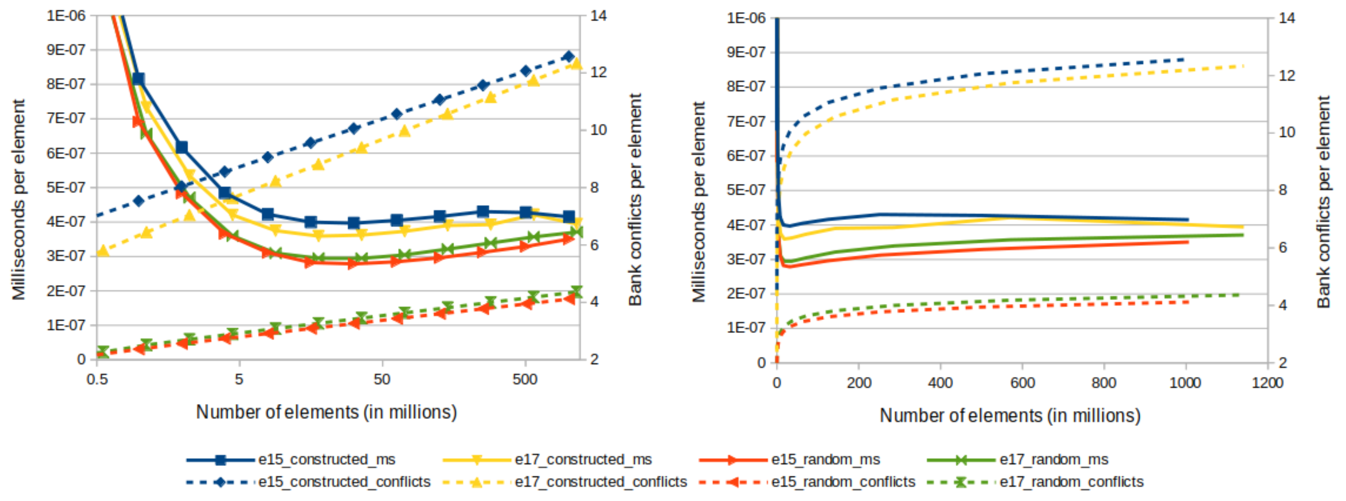

Fig. 6. Runtime (in milliseconds) per element and bank conflicts per element for Thrust on the RTX 2080 Ti. The left figure displays the data with the $x$-axis on a logarithmic scale, to clearly show each individual data point. While the right figure displays the data without data points shown and with the $x$-axis on a linear scale in order to emphasize the resulting logarithmic shape of the curves. In both figures, the solid lines represent the runtime (in milliseconds) per element and the dashed lines represent the bank conflicts per element.

## V. Conclusion

In this paper we showed that for every value of $E < w$, such that $w$ and $E$ are co-prime, there exists an input that reduces the effective parallelism of each warp on the GPU from $w$ down to $\lceil w/E \rceil$ due to memory contention in shared memory. This translates into non-trivial slowdown on such inputs in practice.

One natural question that might arise from this work is: the constructed worst-case input is a very specific permutation and, thus, is very unlikely to occur with high frequency in real world inputs. So why should we care about the worst-case performance?

This is a philosophical question that can be addressed from several aspects:

1) Every undergraduate algorithms course teaches that we should analyze algorithm runtimes on the worst-case inputs. Why should we ignore such analysis for GPU algorithms? Moreover, such analysis might lead to the discovery of better algorithmic techniques on GPUs.
2) The goal of this paper was to prove the existence of a single permutation that asymptotically matches the pessimistic bound of Lemma 1 for the parallel pairwise merge sort algorithm. However, observe that our construction can actually produce a family of permutations, as many of the elements in the non-aligned $w - E$ memory banks can be permuted without affecting the total number of bank conflicts.
3) We could relax our requirement for an absolute worst-case and produce a permutation that has slightly fewer bank conflicts than our constructed permutation. Therefore, there are many more permutations that still incur a significant number of bank conflicts.

4) Observe that the runtimes on the worst-case inputs represent an extreme end of the possible runtime variance. With the constructed inputs causing an average slowdown of ~43% and ~33% on a Quadro M4000 and a RTX 2080 Ti, respectively, the possible variance in runtime is quite significant.

A better question to ask is: can we analyze the expected number of bank conflicts for a given algorithm, for a specific input distribution? This seems to be a very difficult problem for any non-trivial data-dependent algorithm. Understanding how such data dependencies can be modeled so we can apply standard randomized analysis techniques is an interesting open problem. We hope that the analysis presented here will act as the first step in this direction.

## Acknowledgment

We would like to thank John Iacono and Aurélien Ooms for providing us access to their RTX 2080 Ti.

## References

[1] An introduction to modern GPU architecture. http://download.nvidia.com/developer/cuda/seminar/TDCI_Arch.pdf.

[2] Parallel computing with CUDA. http://download.nvidia.com/developer/cuda/seminar/TDCI_CUDA.pdf.

[3] Thrust. https://github.com/thrust/thrust.

[4] Peyman Afshani and Nodari Sitchinava. Sorting and permuting without bank conflicts on GPUs. In *Algorithms - ESA 2015 - 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings*, pages 13–24, 2015.

[5] George E. Andrews. *Number Theory*. 1971.

[6] Sean Baxter. Modern GPU. https://github.com/moderngpu/moderngpu.

[7] Bryan Catanzaro, Alexander Keller, and Michael Garland. A decomposition for in-place matrix transposition. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*, pages 193–206, 2014.

[8] Artur Czumaj, Friedhelm Meyer auf der Heide, and Volker Stemann. Improved optimal shared memory simulations, and the power of reconfiguration. In *Third Israel Symposium on Theory of Computing and Systems, ISTCS 1995, Tel Aviv, Israel, January 4-6, 1995, Proceedings*, pages 11–19, 1995.

[9] Artur Czumaj, Friedhelm Meyer auf der Heide, and Volker Stemann. Shared memory simulations with triple-logarithmic delay. In *Algorithms - ESA '95, Third Annual European Symposium, Corfu, Greece, September 25-27, 1995, Proceedings*, pages 46–59, 1995.

[10] Artur Czumaj, Friedhelm Meyer auf der Heide, and Volker Stemann. Contention resolution in hashing based shared memory simulations. *SIAM J. Comput.*, 29(5):1703–1739, 2000.

[11] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. Simple, efficient shared memory simulations. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '93, Velen, Germany, June 30 - July 2, 1993*, pages 110–119, 1993.

[12] Yuri Dotsenko, Naga K. Govindaraju, Peter-Pike J. Sloan, Charles Boyd, and John Manferdelli. Fast scan algorithms on graphics processors. In *Proceedings of the 22nd Annual International Conference on Supercomputing, ICS 2008, Island of Kos, Greece, June 7-12, 2008*, pages 205–213, 2008.

[13] Chunyang Gou and Georgi Gaydadjiev. Addressing GPU on-chip shared memory bank conflicts using elastic pipeline. *International Journal of Parallel Programming*, 41(3):400–429, 2013.

[14] Oded Green, Robert McColl, and David A. Bader. GPU merge path: a GPU merging algorithm. In *International Conference on Supercomputing, ICS'12, Venice, Italy, June 25-29, 2012*, pages 331–340, 2012.

[15] Sunpyo Hong and Hyesoon Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *36th International Symposium on Computer Architecture (ISCA 2009), June 20-24, 2009, Austin, TX, USA*, pages 152–163, 2009.

[16] Richard M. Karp, Michael Luby, and Friedhelm Meyer auf der Heide. Efficient PRAM simulation on a distributed memory machine. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing, May 4-6, 1992, Victoria, British Columbia, Canada*, pages 318–326, 1992.

[17] Ben Karsin. *A Performance Model for GPU Architectures: Analysis and Design of Fundamental Algorithms*. PhD thesis, University of Hawaii at Manoa, 2018.

[18] Ben Karsin, Henri Casanova, and Nodari Sitchinava. Efficient batched predecessor search in shared memory on GPUs. In *22nd IEEE International Conference on High Performance Computing, HiPC 2015, Bengaluru, India, December 16-19, 2015*, pages 335–344, 2015.

[19] Ben Karsin, Volker Weichert, Henri Casanova, John Iacono, and Nodari Sitchinava. Analysis-driven engineering of comparison-based sorting algorithms on GPUs. In *Proceedings of the 32nd International Conference on Supercomputing, ICS 2018, Beijing, China, June 12-15, 2018*, pages 86–95, 2018.

[20] Akihiko Kasagi, Koji Nakano, and Yasuaki Ito. An implementation of conflict-free offline permutation on the GPU. In *Third International Conference on Networking and Computing, ICNC 2012, Okinawa, Japan, December 5-7, 2012*, pages 226–232, 2012.

[21] Atsushi Koike and Kunihiko Sadakane. A novel computational model for GPUs with applications to efficient algorithms. *IJNC*, 5(1):26–60, 2015.

[22] Nikolaj Leischner, Vitaly Osipov, and Peter Sanders. GPU sample sort. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*, pages 1–10, 2010.

[23] Lin Ma, Kunal Agrawal, and Roger D. Chamberlain. A memory access model for highly-threaded many-core architectures. *Future Generation Comp. Syst.*, 30:202–215, 2014.

[24] Kurt Mehlhorn and Uzi Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Inf.*, 21:339–374, 1984.

[25] Duane Merrill. CUB. https://nvlabs.github.io/cub/.

[26] Bruce Merry. A performance comparison of sort and scan libraries for GPUs. *Parallel Processing Letters*, 25(4), 2015.

[27] Friedhelm Meyer auf der Heide, Christian Scheideler, and Volker Stemann. Exploiting storage redundancy to speed up randomized shared memory simulations. *Theor. Comput. Sci.*, 162(2):245–281, 1996.

[28] Koji Nakano. Simple memory machine models for GPUs. In *26th IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum, IPDPS 2012, Shanghai, China, May 21-25, 2012*, pages 794–803, 2012.

[29] Koji Nakano. The hierarchical memory machine model for GPUs. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum, Cambridge, MA, USA, May 20-24, 2013*, pages 591–600, 2013.

[30] Hagen Peters, Ole Schulz-Hildebrandt, and Norbert Luttenberger. Fast in-place, comparison-based sorting with CUDA: a study with bitonic sort. *Concurrency and Computation: Practice and Experience*, 23(7):681–693, 2011.

[31] Hagen Peters, Ole Schulz-Hildebrandt, and Norbert Luttenberger. A novel sorting algorithm for many-core architectures based on adaptive bitonic sort. In *26th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2012, Shanghai, China, May 21-25, 2012*, pages 227–237, 2012.

[32] Nadathur Satish, Mark J. Harris, and Michael Garland. Designing efficient sorting algorithms for manycore GPUs. In *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*, pages 1–10, 2009.

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

[34] Eli Upfal and Avi Wigderson. How to share memory in a distributed system. *J. ACM*, 34(1):116–127, 1987.