Eliminating Bank Conflicts in GPU Mergesort

Kyle Berney
University of Hawaii at Mānoa
Honolulu, HI, USA
berneyk@hawaii.edu

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

ABSTRACT

Graphics Processing Units (GPUs) have emerged as popular architectures for high-performance computing due to their parallelism and low latency context switching capabilities. However, optimizing GPU implementations can be challenging due to the complexity of the architecture, such as the diverse characteristics of memory units. While most optimization efforts focus on parallelism and global memory access, for some algorithms memory conflicts in shared memory, known as bank conflicts, can significantly impact performance. This affects the accuracy of theoretical runtime analysis of GPU algorithms.

In this paper, we present a number-theoretic solution for eliminating all bank conflicts for Thrust library's mergesort implementation – the fastest comparison-based sorting implementation on GPUs. Our experiments demonstrate that the modified mergesort takes virtually the same time to run on the worst-case inputs as it does on random inputs (the worst-case inputs have been shown in the past to cause up to 50% slowdown).

CCS CONCEPTS

• **Theory of computation → Shared memory algorithms**; **Sorting and searching**.

KEYWORDS

GPU algorithms, bank conflicts, worst-case analysis

ACM Reference Format:
Kyle Berney and Nodari Sitchinava. 2025. Eliminating Bank Conflicts in GPU Mergesort. In *37th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '25), July 28-August 1, 2025, Portland, OR, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3694906.3743337

1 INTRODUCTION

Over the past decade, Graphics Processing Units (GPUs) have become a popular architecture for high performance computing. GPUs are highly parallel architectures, featuring thousands of physical cores and low latency context switching capabilities, thereby allowing the utilization of hundreds of thousands of threads. However, due to the hierarchical design of its memory system, with each level of the memory hierarchy having its own latency, bandwidth, and optimal access requirements, achieving fast GPU implementations can be a challenging task.[1]

As a consequence, most implementations on GPUs focus on optimizing a few (typically one or two) performance criteria, such as parallelism [7, 16, 24, 39], global memory accesses [5, 7, 10, 29, 32, 39], shared memory accesses [1, 8, 11, 18, 21, 23, 28, 30, 44], register utilization [5, 6, 19, 29, 41], or synchronization [20, 42, 43]. Typically, high-performance GPU implementations are optimized to expose a high degree of parallelism and to reduce the number of accesses to *global memory* – the largest memory unit with the highest latency. Both of these aspects can be modeled using well-known computational models that provide a wealth of algorithms and techniques that can be adopted to GPUs. For example, the techniques developed in the classical *Parallel Random Access Machine (PRAM)* model [26] carry over well for maximizing parallelism needed for the large number of GPU threads. On the other hand, since optimal accesses to global memory are essentially performed in blocks of contiguous data, known as *coalesced accesses*, such accesses to global memory can be modeled and analyzed using the *External Memory (EM)* model [2] (or its parallel variant – the *PEM* model [4]), where the global memory is modeled as the external memory, and the faster, but smaller *shared memory* acts as the internal memory.

The EM and PEM models treat internal memory as a uniform random access memory and many GPU implementations treat shared memory as such. However, that is not actually the case in practice and to take a full advantage of the GPU performance potential, it is often useful to consider the special features of shared memory when designing algorithms.

Design of shared memory. First of all, shared memory is shared among all threads of a *thread block* – a program-defined, but globally fixed, subset of threads. Threads within each thread block are scheduled by the hardware in groups of w threads, called *warps*, and similar to the PRAM, the threads within a warp execute synchronously.[2] On current NVIDIA GPUs, $w = 32$.

Work supported by National Science Foundation grants CCF-1911245 and CCF-2432018.

SPAA '25, July 28-August 1, 2025, Portland, OR, USA

ACM ISBN 979-8-4007-1258-6/2025/07
https://doi.org/10.1145/3694906.3743337

[1] We provide a brief overview of relevant GPU terminology and concepts. For more details, we refer interested readers to [35, 36].
[2] Starting with NVIDIA GPUs with Compute Capability 7.0, the threads within a warp no longer execute synchronously. However, if there are no delays due to bank conflicts, the threads have no reason to diverge, and will continue to execute in lock-step. Hence, synchronous execution is a reasonable assumptions when designing bank conflict free algorithms even for the newer hardware.

Shared memory is organized into w memory modules, known as *memory banks*,[3] and accessing the same memory bank simultaneously by multiple threads in the same warp incurs a delay.[4] This congestion on memory banks is known as *bank conflicts* in the GPU terminology. More specifically, in shared memory that consists of w banks, the i-th memory bank is formed by addresses $j \equiv i \pmod{w}$, i.e., items in contiguous array are stored strided across the memory banks. Hence, when coarsening parallelism, the natural approach of having each thread process contiguous subarray in shared memory (which indeed is often used in practice) may lead to bank conflicts. It has been shown that in GPU implementations, which heavily utilize shared memory, bank conflicts can significantly increase the overall runtime in the worst case [8, 23, 29, 44].

Formal analysis of the delay due to bank conflicts in an algorithm is a challenging task, especially if accesses to shared memory are data-dependent. For this reason, most GPU implementations forego bank conflict analysis and instead evaluate GPU implementations empirically, utilizing various profiling tools to measure bank conflicts at runtime and applying ad-hoc heuristics in an attempt to reduce the number of bank conflicts experimentally.

In this paper, we show that we can eliminate all bank conflicts in the Thrust library's implementation of the mergesort algorithm [38]. Our surprisingly simple approach results in runtimes that are virtually the same as the runtime of the highly optimized Thrust mergesort on random inputs, but ours holds in the worst-case.

Thrust's mergesort implementation. Thrust library implements the classical mergesort by parallelizing the merging of two sorted arrays $\mathcal{A}$ and $\mathcal{B}$, each of size $n/2$, as follows. To implement the merge using t threads, it identifies t pairs of contiguous subarrays A_i and B_i, $|A_i| + |B_i| = n/t$, such that the sorted sequence of $A_i \cup B_i$ forms a contiguous subarray in the sorted output of $\mathcal{A} \cup \mathcal{B}$. The identification of the i-th pair of subsequences A_i and B_i is an order statistic that is found by the i-th thread independently of other threads via a mutual binary search on $\mathcal{A}$ and $\mathcal{B}$ in $O(\log n)$ time. (This is often given as an exercise in classical textbooks on algorithms, e.g., [12, Exercise 9.3-10].) This approach is known as the *merge path algorithm*, named so by Green et al. [24], who were the first to implement it on GPUs and, which was since incorporated in the Thrust library [38].

To reduce the number of global memory accesses, the identification of pairs A_i and B_i is performed hierarchically in 2 stages: first in global memory, where subsequences for each thread block are identified and loaded into (contiguous) shared memory space; then in shared memory, where each thread identifies A_i and B_i. However, because the subarrays A_i and B_i are kept contiguous in shared memory and, therefore, occupy various memory banks, the naive (sequential) processing of A_i and B_i by the i-th thread directly in shared memory may lead to bank conflicts. Eliminating bank conflicts is extra challenging if the access pattern to the items of A_i and B_i at each step is data dependent. In fact, analytically determining the number of bank conflicts even for the classical problem of merging sorted sequences of A_i and B_i on a random input is an open problem. Therefore, the current state-of-the-art implementation of mergesort on GPUs [24, 38] instead uses the following heuristic, which was determined to perform better in practice: choose the number of threads t such that $|A_i| + |B_i| = n/t$ is coprime with w. Karsin et al. [29] empirically measured that on randomly chosen inputs, the average number of bank conflicts per step is a small constant (between 2 and 3). In contrast, Berney and Sitchinava [8] proved the existence of inputs that cause $n/t - o(n/t)$ bank conflicts per step, when n/t and w are coprime. Note that n/t is the trivial upper bound on the number of bank conflicts. They also showed that these inputs caused a peak slowdown of $\approx 50\%$ in practice, compared to the runtime on random inputs.

Our contributions. In this work, we show that each pair of subarrays A_i and B_i can be loaded from shared memory into a register file of the i-th thread without incurring *any* bank conflicts. We refer to this procedure as the *load-balanced dual subsequence gather*.[5] Once the subarrays A_i and B_i are in the registers of a thread, they can be merged by the thread sequentially without any additional access to shared memory, i.e., without any bank conflicts. Having no bank conflicts not only improves practical performance, but also significantly simplifies theoretical analysis of the mergesort on GPUs, because without the delay due to bank conflicts, the runtime analysis becomes equivalent to the analysis in the PRAM model. We experimentally evaluate our approach by incorporating our load-balanced dual subsequence gather procedure into Thrust's highly tuned mergesort implementation. Our experiments demonstrate that the runtime using our bank conflict free load-balanced dual subsequence gather is essentially the same as of the original Thrust Mergesort on random inputs. Since it had been shown in the past that random inputs cause 2-3 bank conflicts on average per element, in practice, the overhead of our approach is equivalent to 2-3 extra accesses to shared memory. However, this holds for all inputs, including in the worst case.

Finally, we use our observations to generalize the worst case input for Thrust mergesort by Berney and Sitchinava [8] to values of n/t that are not coprime with w. In practice, Thrust mergesort performs significantly worse when n/t is not coprime with w. Moreover, since $w = 32$ on all modern GPUs, it is easy to find parameter t, which makes n/t be coprime with w. However, generalizing worst-case input construction to arbitrary parameter values is of theoretical interest, and was left as an open problem by the prior work [8]. Here we close this question by providing a method to produce worst case input for arbitrary t.

2 PRELIMINARIES

Memory organization into memory modules is not unique to GPUs, emerging as early as the 1980s [22]. Such memory design has been modeled using the *Distributed Memory Machine (DMM)* [31] and the analysis of the delay, due to congestion on memory modules, of an arbitrary PRAM algorithm is known as the *granularity of parallel memories* problem [13–15, 17, 27, 31, 33, 40]. The DMM

[3] While, technically, the number of shared memory banks could be different from the number of threads in a warp, it is natural that these numbers are the same and is actually the case on all modern NVIDIA GPUs, so we parameterize them by the same variable w.

[4] More precisely, multiple threads of the same warp accessing *distinct* memory locations in the same memory bank simultaneously incur a delay, i.e., access to the same element by multiple threads does not incur a delay. However, this feature is irrelevant for the problems we consider here because all threads access distinct elements.

[5] The inverse procedure can be used to write elements from registers into shared memory in a bank conflict free manner, i.e., a load-balanced dual subsequence scatter.

0:	0	12	24	36	48
1:	1	13	25	37	49
2:	2	14	26	38	50
3:	3	15	27	39	51
4:	4	16	28	40	52
5:	5	17	29	41	53
6:	6	18	30	42	54
7:	7	19	31	43	55
8:	8	20	32	44	56
9:	9	21	33	45	57
10:	10	22	34	46	58
11:	11	23	35	47	59

0:	0	12	24	36	48	60
1:	1	13	25	37	49	61
2:	2	14	26	38	50	62
3:	3	15	27	39	51	63
4:	4	16	28	40	52	64
5:	5	17	29	41	53	65
6:	6	18	30	42	54	66
7:	7	19	31	43	55	67
8:	8	20	32	44	56	68
9:	9	21	33	45	57	69
10:	10	22	34	46	58	70
11:	11	23	35	47	59	71

Figure 1: Visualization of strided accesses in shared memory with $w = 12$ (number of shared memory banks and number of threads in a warp). The left figure depicts bank conflict free accesses: the w threads of a warp access green entries concurrently. In comparison, the right figure depicts the worst case number of bank conflicts resulting from the w threads of a warp accessing the red entries concurrently. Cells are labeled with their shared memory index/address.

can be used to model shared memory accesses on a GPU, because of the natural mapping of shared memory banks to memory modules of the DMM and threads of a warp to DMM processors. For a DMM consisting of w memory modules and w synchronous processors, Czumaj et al. [14] present an access schedule that results in a $O(\log\log\log w \log^* w)$ factor delay, with high probability. In contrast, a naive PRAM implementation can incur a $O(w)$ factor slowdown in the worst case.

Unfortunately in practice, the overheads associated with the techniques used in these general approaches, such as universal hashing, randomization, and data replication, make it impractical for high performance implementations. Alternatively, one can design *bank conflict free* algorithms – dedicated algorithms for specific problems that guarantee no bank conflicts – directly in the DMM model [1, 11, 18, 21, 28, 30, 34, 44]. Without any bank conflicts, the runtime analysis becomes much simpler, as it becomes equivalent to PRAM analysis. Compared to standard PRAM approaches, however, bank conflict free algorithms usually come at a cost of increased overhead, e.g., auxiliary memory usage [1, 11, 18, 21], increased code complexity [21, 44], higher constant factors [1, 11], or more overall work [28, 30, 44].

The crux of designing bank conflict free algorithms is understanding the mapping of accesses performed to each of the w shared memory banks. On NVIDIA GPUs, memory bank $i \in \{0, 1, \ldots, w-1\}$ contains every i-th element; i.e., the j-th element in shared memory resides in memory bank $i \equiv j \pmod{w}$. Thus, a 2-dimensional matrix consisting of w rows is a natural visualization, where the i-th row corresponds to the i-th memory bank and data is laid out in the column-major order. It has been observed in previous work [8, 11, 18, 28] that bank conflicts do not occur when accesses by threads of a warp are separated by a distance that is coprime with w (i.e., does not share a common divisor with w). Conversely, bank conflicts do occur when the access stride instead shares a common divisor with w (i.e., not coprime). Figure 1 illustrates this behavior on shared memory with $w = 12$, using accesses strided by a distance of 5 (coprime) and 6 (not coprime). Leveraging this observation, researchers have designed bank conflict free algorithms via padding data, staggering accesses, and/or permuting elements into an alternate layout. However, in spite of this common design approach, insight into this behavior has not been fully understood and formalized, leaving researchers to reprove bank conflict free accesses for each problem considered. In this work, we apply number theory—a perfect tool to discover insights into integer mappings—to clarify and codify this phenomena. Our analysis in Sections 3 and 4 relies on various number theory results, specifically, we utilize results related to congruences, the greatest common divisor of two integers, and complete residue systems. For a full review of definitions and relevant number theory results, we refer readers to Appendix A. Formal proofs of these results can by found in introductory textbooks on number theory, e.g., [3] with the exception of Corollary 17 and 18, which we provide for completeness, although they are typically given as an exercise in undergraduate number theory courses.

3 LOAD-BALANCED DUAL SUBSEQUENCE GATHER

The load-balanced dual subsequence gather is a bank conflict free algorithm for loading subsequences from at most two sequences, from shared memory into register space. Intuitively, it is a simple algorithm consisting of a permutation and a scan of A_i in ascending order and B_i in descending order (see Algorithm 1). As mentioned in Section 2, we use number theory to prove that such access results in no bank conflicts, namely we construct *complete residue systems* modulo w (see Definition 13 in the Appendix) and use various results related to the greatest common divisor (GCD) of two integers in our proofs.

To explain the basic ideas of our algorithm, in Sections 3.1 and 3.2, we consider a single warp with its elements from A and B stored in contiguous shared memory locations. We start in Section 3.1 by considering values of w and E that are coprime and show that reversing B and dynamically staggering the scan results in bank conflict free accesses. When w and E are not coprime, additional bank conflicts arise from strided accesses to shared memory. In Section 3.2, we show that these bank conflicts can be eliminated by performing an additional circular shift of the input in shared memory. Finally, in Section 3.3 we extend our approach to a full thread block, leading to a practical implementation.

Table 1 describes the main parameters used throughout this section. We assume that the number of threads per thread block, u, is a multiple of w, so that there are u/w complete warps in a thread block. We refer to the subsequences of $\mathcal{A}$ and $\mathcal{B}$ for a thread block as A and B; and the offsets of A_i and B_i in A and B, i.e., indices where the i-th thread starts) as a_i and b_i, respectively.

Algorithm 1 Reads $|A_i|$ elements from the A list and $|B_i| = E - |A_i|$ elements from the B list from shared memory into registers. Let π be a permutation that reverses the order of elements (described in Section 3.1) and ρ performs a circular shift (described in Section 3.2).

```
1: DUAL-SUBSEQUENCE-GATHER(shmem, items, A, B, a_i, b_i, |A_i|, |B_i|)
2:     shmem = ρ(A ∪ π(B))                          ▷ Permute elements in shared memory
3:     k = a_i (mod E)
4:     for j = 0 to E − 1
5:         if j − k (mod E) < |A_i|
6:             idx = ρ(j − k (mod E) + a_i)          ▷ Read (j − k (mod E))-th element of A_i
7:         else
8:             idx = ρ(π(k − j − 1 (mod E) + b_i))   ▷ Read (k − j − 1 (mod E))-th element of B_i
9:         items[j] = shmem[idx]
```

Table 1: Descriptions of the main parameters for the load-balanced dual subsequence gather.

Parameter	Description
A	Subarray of $\mathcal{A}$ for a thread block
B	Subarray of $\mathcal{B}$ for a thread block
A_i	Subarray of A for the i-th thread
B_i	Subarray of B for the i-th thread
a_i	Offset of A_i in A
b_i	Offset of B_i in B
u	Number of threads per thread block
w	Number of banks in shared memory and the number of threads per warp
E	Number of elements per thread
d	Greatest common divisor of w and E

3.1 Coprime w and E

Accessing elements in shared memory with a stride distance that is coprime, relative to the number of banks, has been commonly used to reduce the number of bank conflicts. We start by formalizing this pattern in the context of number theory and show that using a coprime stride distance results in a complete residue system.

LEMMA 1. *Let $j \in \mathbb{Z}$. If $d = \text{GCD}(w, E) = 1$, then $R_j = \{j + kE : k \in \mathbb{Z} \text{ and } 0 \le k < w\}$ is a complete residue system.*

PROOF. Since $|R_j| = w$, it suffices to show that for all $r_a, r_b \in R_j$, if $r_a \neq r_b$ then $r_a \not\equiv r_b \pmod w$. Assume for the sake of contradiction that $r_a \equiv r_b \pmod w$. It follows from Corollary 16, that $r_a \equiv r_b \pmod w \implies j + aE \equiv j + bE \pmod w \implies a \equiv b \pmod w \implies a = b$, since $0 \le a, b < w$, which is a contradiction. □

Consider an arbitrary warp with its subsequences of A and B stored in contiguous shared memory locations (subsequences of A stored first and subsequences B stored immediately afterwards). For ease of exposition, we refer to the elements by its local index in shared memory in this memory layout (e.g., the local index of the first element is 0). Our approach performs E rounds of shared memory accesses, where in round $j \in \{0, 1, \ldots, E-1\}$, R_j defines the set of w elements that are read into the register space of threads. It follows from Lemma 1, that for every j, all elements of R_j are located in distinct shared memory banks. To ensure bank conflict free access, it remains to show that every round j can be performed with a single parallel access by the w threads of the warp, i.e., each thread of the warp reads exactly one element of R_j.

Without loss of generality, consider the elements in A. Since the elements in R_j are separated by E positions and the number of elements that will be accessed by any single thread in A is at most E (i.e., $|A_i| \le E$), the elements in R_j that reside in the A list will be read by unique threads. Accounting for both A and B, at most 2 elements will be read by any single thread in each round (see Figure 7 for an example). To resolve this conflict between A and B, we reverse the order of the elements in B. Recall that $a_i + b_i = iE$ and $|A_i| + |B_i| = E$. The order of elements in the A list remains unchanged, hence, elements of A_i are read in ascending order in rounds:

$$a_i \pmod E, a_i + 1 \pmod E, \ldots, a_i + |A_i| - 1 \pmod E .$$

After reversing the order of elements in the B list, the elements of B_i are now located at indices $\{(wE-1) - b_i, (wE-1) - (b_i + 1), \ldots, (wE-1) - (b_i + |B_i| - 1)\}$. Hence, the first element of B_i is read in round $wE - 1 - b_i \equiv a_i - 1 \pmod E$ and the last element of B_i is read in round $wE - 1 - (b_i + |B_i| - 1) \equiv a_i + |A_i| \pmod E$. Overall, B_i is read in descending order in rounds:

$$a_i + |A_i| \pmod E, \ldots, a_i - 2, a_i - 1 \pmod E .$$

Therefore, exactly a single element is read in each round by every thread. Figure 2 illustrates an example of the accesses performed.

3.2 Non-coprime w and E

In Section 3.1 (coprime w and E), Lemma 1 shows that for $j \in \mathbb{Z}$, $R_j = \{j + kE : k \in \mathbb{Z} \text{ and } 0 \le k < w\}$ is a complete residue system modulo w. However, if $d = \text{GCD}(w, E) > 1$ (w and E are not coprime), then $w/d \in \mathbb{Z}$ and every (w/d)-th element in R_j is congruent to each other modulo w. Let $r_a, r_{a+\frac{w}{d}} \in R_j$, $r_a = j + aE \equiv j + aE \pmod w \equiv j + \left(a + \frac{w}{d}\right) E = r_{a+\frac{w}{d}}$. Therefore, if E and w are not coprime, then R_j is not a complete residue system modulo w. To solve this issue, we partition R_j into d disjoint subsets, each consisting of w/d elements. For $j \in \{0, 1, \ldots, E-1\}$ and $\ell \in \{0, 1, \ldots, d-1\}$, let

$$R_j^{(\ell)} = \left\{ j + \left(\frac{\ell w}{d} + k \right) E : k \in \mathbb{Z} \text{ and } 0 \le k < \frac{w}{d} \right\}$$

$$\text{and } D_\ell = \left\{ \ell + kd : k \in \mathbb{Z} \text{ and } 0 \le k < \frac{w}{d} \right\} .$$

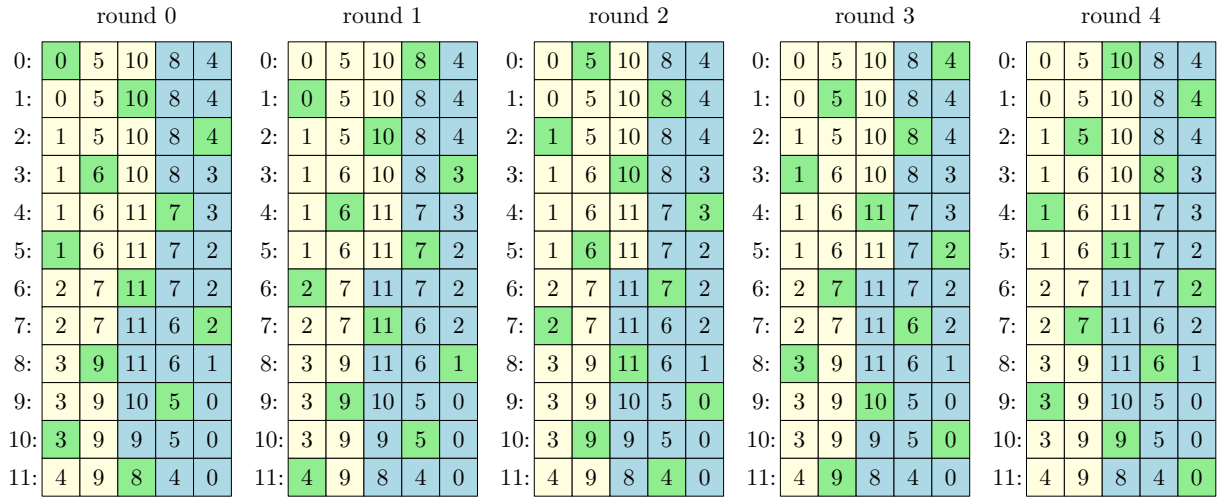

Figure 2: An example of shared memory accesses performed by a warp in `CF-Merge` for $w = 12$, $E = 5$, and $d = 1$ (i.e., coprime) on an arbitrary input. Elements belonging to the A list (B list) are colored yellow (blue). Cell values correspond to the thread IDs that access that cell. Cells colored green are the cells accessed by the w threads in a particular round, demonstrating no bank conflicts in that round.

We show that the elements in each subset $R_j^{(\ell)}$ are congruent to the elements in $D_{j \pmod d}$. First observe that $D = \bigcup_{\ell=0}^{d-1} D_\ell$ is a complete residue system modulo w. Hence, to construct a complete residue system modulo w, we shift subsets so that each resulting set R'_j contains a single partition that is congruent to a unique D_ℓ.

Lemma 2. *Let $j' \in \{0, 1, \ldots, d-1\}$ such that $j \equiv j' \pmod d$ (i.e., $j = qd + j'$ for some $q \in \mathbb{Z}$). Consider the sets $R_j^{(\ell)}$ and $D_{j'}$.*

1. *For all $r_a \in R_j^{(\ell)}$, there exists $d_b \in D_{j'}$ such that, $r_a \equiv d_b \pmod w$.*
2. *For all $r_a, r_b \in R_j^{(\ell)}$ such that $r_a \neq r_b$, $r_a \not\equiv r_b \pmod w$.*

Proof. *Proof of (1):* By the definition of the greatest common divisor, $\frac{E}{d} \in \mathbb{Z}$. Hence, $r_a = j + (\frac{\ell w}{d} + a)E = j + \frac{\ell wE}{d} + a \cdot \frac{E}{d} \cdot d \equiv j + a \cdot \frac{E}{d} \cdot d \pmod w$. Thus, there exists $a' \in \mathbb{Z}_w$ such that $0 \leq a' < \frac{w}{d}$ and $a \cdot \frac{E}{d} \equiv a' \pmod w$. If $j < d$, then $r_a \equiv j + a'd \pmod w \equiv d_{a'} \pmod w$. Otherwise, $j \geq d$ and there exists $q \in \mathbb{Z}$ such that $j = qd + j'$, therefore, $r_a \equiv qd + j' + a'd \pmod w \equiv d_{q+a' \pmod{w/d}} \pmod w$.

Proof of (2): We have that $r_a = j + (\frac{\ell w}{d} + a)E \equiv j + a \cdot \frac{E}{d} \cdot d \pmod w \equiv j + a'd \pmod w$ and $r_b = j + (\frac{\ell w}{d} + b)E \equiv j + b \cdot \frac{E}{d} \cdot d \pmod w \equiv j + b'd \pmod w$, for some $a', b' \in \mathbb{Z}_{\frac{w}{d}}$. Therefore, it suffices to show that $a' \not\equiv b' \pmod{w/d}$. Assume for the sake of contradiction, that $a' \equiv b' \pmod{w/d}$. It follows from Corollary 18, that $\gcd(\frac{w}{d}, \frac{E}{d}) = 1$, and hence from Corollary 16, $a' \equiv b' \pmod{w/d} \implies a\left(\frac{E}{d}\right) \equiv b\left(\frac{E}{d}\right) \pmod{w/d} \implies a \equiv b \pmod{w/d} \implies a \not\equiv b \pmod{w/d}$, since $0 \leq a, b < \frac{w}{d}$, which is a contradiction. □

Corollary 3. *Let $R'_j = R_j^{(0)} + R_{j+1 \pmod E}^{(1)} + R_{j+2 \pmod E}^{(2)} + \ldots + R_{j+d-1 \pmod E}^{(d-1)}$. R'_j is a complete residue system modulo w.*

Proof. It follows from Lemma 2 that each partition of R'_j is congruent to $D_{j'}$. Since R'_j is the union of consecutive indexed partitions (in a circular manner), each partition is congruent to a unique $D_{j'}$. Therefore, R'_j is a complete residue system modulo w. □

Lemma 4. *Consider the last element in $R_j^{(\ell)}$, denoted a, and the first element in $R_{j+1 \pmod E}^{(\ell+1)}$, denoted b. The difference $(b - a)$ is $(E + 1)$, if $j < (E - 1)$, and 1 otherwise.*

Proof. *Case 1:* $j < E - 1 \implies b - a = \left(j + 1 + \left(\frac{(\ell+1)w}{d}\right)E\right) - \left(j + \left(\frac{(\ell+1)w}{d} - 1\right)E\right) = E + 1$.

Case 2: $j = E - 1 \implies b - a = \left(\left(\frac{(\ell+1)w}{d}\right)E\right) - \left((E-1) + \left(\frac{(\ell+1)w}{d} - 1\right)E\right) = 1$. □

Lemma 4 shows that for $0 \leq j \leq E-d$, the distance between all elements in R'_j is at least E; and for $E-d < j < E$, the distance between all elements in R'_j is at least E except for a single pair of neighboring elements (i.e., a distance of 1). Ideally, we want the access pattern to match the one used in Section 3.1, so that the elements in R'_j are separated by a distance of exactly E. By construction of R'_j, any element indexed at location $k \in \{0, 1, \ldots, wE-1\}$ will be read in round $j \equiv k - \left\lfloor \frac{kd}{wE} \right\rfloor \pmod E$. Notice that for $k = \ell \cdot \frac{wE}{d}$, the element indexed at k is read in round $\ell \cdot \frac{wE}{d} - \left\lfloor \ell \cdot \frac{wE}{d} \cdot \frac{d}{wE} \right\rfloor = \ell \cdot \frac{wE}{d} - \ell \equiv -\ell \pmod E$. Furthermore for $x \in \{0, 1, \ldots, \frac{wE}{d} - 1\}$, the element at index $(k + x)$ is read in round $x - \ell \pmod E$. Intuitively, each partition of $\left(\frac{wE}{d}\right)$ elements has an access pattern that is circular shifted by ℓ rounds relative to the access pattern of the 0-th partition (elements indexed $\{0, 1, \ldots, \frac{wE}{d} - 1\}$). Therefore, we align the accesses to elements in the ℓ-th partition by performing a circular shift of ℓ

locations. After shifting elements, any element originally indexed at location k is read in round $j \equiv k \pmod{E}$. As in Section 3.1, to resolve read conflicts between lists, we additionally reverse the order of the elements in B. Figure 3 illustrates an example of accesses for values of w and E that are not coprime.

3.3 Thread Block

In this section we extend our algorithm to processing a thread block that consists of u threads (organized into u/w warps).

Consider an arbitrary warp $v \in \{0, 1, \ldots, \frac{u}{w} - 1\}$ in a thread block and let α_v and β_v be the indices of the first elements in the A list and B list for the warp, respectively. Since each warp processes wE elements, there are at most vwE elements from the A list assigned to the previous $(v-1)$ warps. Hence, $\alpha_v \in \{0, 1, \ldots, vwE - 1\}$. We extend the permutation used in Section 3.1 to reverse all elements in the B list for the full thread block. After this reversal, $\beta_v = (uwE - 1) - (vwE - \alpha_v) = (u - v)wE + \alpha_v - 1$. Let $|A_v|, |B_v| \in \mathbb{Z}^+$ be the number of elements in the A and B list assigned to warp v, i.e., $|A_v| + |B_v| = wE$, or equivalently, $|B_v| = wE - |A_v|$. For warp v, the elements of the A list start in memory bank $\alpha_v \pmod{w}$ and end in memory bank $\alpha_v + |A_v| - 1 \pmod{w}$. And, the elements of the B list end in memory bank $(u - v)wE + \alpha_v - 1 - (|B_v| - 1) \equiv \alpha_v + |A_v| \pmod{w}$ and start in memory bank $(u - v)wE + \alpha_v - 1 \equiv \alpha_v - 1 \pmod{w}$.

Therefore, each warp can view the resulting memory layout as wE elements stored in contiguous memory locations (starting in an arbitrary memory bank). For values of E such that $\gcd(w, E) > 1$ (i.e., not coprime), the permutation ρ is similarly extended where each partition $\ell \in \{0, 1, \ldots, \frac{ud}{w} - 1\}$ of $\left(\frac{wE}{d}\right)$ elements are circular shifted by $\ell \pmod{d}$ positions, with accesses in each partition shifted accordingly. Figure 8 illustrates an example of accesses for a full thread block.

4 WORST-CASE INPUTS

Although the mergesort implementation in the state-of-the-art Thrust library [38] has been shown empirically to perform only 2 to 3 bank conflicts on average on random inputs [29], merging of two sorted lists A_i and B_i is a deterministic process. Hence, a careful design of the input permutation could result in a large number of bank conflicts in the worst case. In our previous work [8] we presented an algorithm that generated such a worst-case permutation. However, our algorithm was restricted to the values of w and E, such that w is a power of 2, $d = \gcd(w, E) = 1$, and $\frac{w}{2} < E < w$. The task of generating worst-case inputs for other values of w and E was left as an open problem.

In this section, building on our observations in Section 3.2, we generalize this strategy to an arbitrary $d = \gcd(w, E) > 1$, extend the parameter w to an arbitrary integer, and extend the range of E to $1 < E \leq w$. For a visualization of our inputs, see Figure 4.

Without loss of generality, consider an arbitrary warp with $|A| = \left\lceil \frac{E}{2} \right\rceil w$ and $|B| = \left\lfloor \frac{E}{2} \right\rfloor w$. Our approach is to divide the wE elements into d subproblems of $\frac{wE}{d}$ elements and $\frac{w}{d}$ threads, and construct worst-case inputs for each subproblem independently. Without loss of generality, we consider $\left\lceil \frac{E}{2d} \right\rceil w$ elements of A and $\left\lfloor \frac{E}{2d} \right\rfloor w$ elements of B. By Euclid's Division Lemma, there exist unique positive integers q and r such that $0 \leq r < E$ and $w = qE + r$. For $i = \{1, 2, \ldots, \frac{E}{d} - 1\}$, we define $s_i = i\left(\frac{r}{d}\right) \pmod{E/d}$, $x_i = \left(\frac{E}{d} - s_i\right) d$, $y_i = s_i \cdot d$, and $S = (a_i, b_i)$ to be a sequence, such that

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

Lemma 5. *$s_i \neq s_j$ for any $i, j \in \{1, 2, \ldots, \frac{E}{d} - 1\}$ such that $i \neq j$.*

Proof. Assume, for the sake of contradiction, that $s_i = s_j$.

From Lemma 17, $d = \gcd(w, E) = \gcd(E, r)$; and from Lemma 18, $\gcd\left(\frac{w}{d}, \frac{E}{d}\right) = 1$ and $\gcd\left(\frac{E}{d}, \frac{r}{d}\right) = 1$. Then, it follows from Lemma 16 that $s_i = s_j \implies i\left(\frac{r}{d}\right) \equiv j\left(\frac{r}{d}\right) \pmod{E/d} \implies i \equiv j \pmod{E/d} \implies i = j$, which is a contradiction. □

Lemma 6. *$\frac{E}{d} - s_i = s_{\frac{E}{d} - i}$ for all $i \in \{1, 2, \ldots, \frac{E}{d} - 1\}$.*

Proof. $\frac{E}{d} - s_i = \frac{E}{d} - \left(i\left(\frac{r}{d}\right) \pmod{E/d}\right) \equiv \left(\frac{E}{d} - i\right)\left(\frac{r}{d}\right) \pmod{E/d} = s_{\frac{E}{d} - i}$. □

Lemma 7. *For all $i \in \{1, 2, \ldots, \frac{E}{d} - 2\}$:*

$$x_i + y_{i+1} = \begin{cases} r & \text{if } x_i < r \iff s_i > \frac{E}{d} - \frac{r}{d} \\ E + r & \text{otherwise.} \end{cases}$$

Proof. (Note that $x_i = r \iff i = \frac{E}{d} - 1$, therefore, there are only two cases.)

Case 1: Assume $x_i < r \iff s_i > \frac{E}{d} - \frac{r}{d} \implies s_{i+1} = s_i + \frac{r}{d} - \frac{E}{d} \therefore x_i + y_{i+1} = \left(\frac{E}{d} - s_i\right) d + s_{i+1} \cdot d = \left(\frac{E}{d} - s_i + s_i + \frac{r}{d} - \frac{E}{d}\right) d = r$.

Case 2: Assume $x_i > r \iff s_i < \frac{E}{d} - \frac{r}{d} \implies s_{i+1} = s_i + \frac{r}{d} \therefore x_i + y_{i+1} = \left(\frac{E}{d} - s_i\right) d + s_{i+1} \cdot d = \left(\frac{E}{d} - s_i + s_i + \frac{r}{d}\right) d = E + r$. □

At a high level, our worst-case construction algorithm will construct a sequence T from S, consisting of tuples, such that the i-th tuple $(a_i, b_i) \in T$, will correspond to thread i reading a_i items from list A and b_i items from list B. In fact, a large number of threads will be reading items only from list A or list B, i.e., the corresponding tuples will be of the form $(E, 0)$ or $(0, E)$. If we visualize shared memory as a $w \times \frac{N}{w}$ matrix, with w rows representing the w memory banks, the items corresponding to the tuples of the form $(E, 0)$ or $(0, E)$ will be aligned at the bottom E rows. Thus, the threads reading these items will be forced to perform a sequential scan of the bottom E banks, causing bank conflicts with each access. The remaining tuples, which are exactly the tuples of S, will be used to help this alignment.

Formally, we construct a new sequence T from S as follows:

(1) Insert the tuple $(a_1, b_1) = (y_1, x_1) = (r, E - r)$ of S, followed by q tuples of $(E, 0)$;
(2) For $i = \{1, 2, \ldots, \frac{E}{d} - 2\}$, insert $(a_{i+1}, b_{i+1}) \in S$ followed by:
   - If $x_i + y_{i+1} = r$, insert q tuples of $(E, 0)$ if i is even or $(0, E)$ otherwise.
   - If $x_i + y_{i+1} = E + r$, insert $(q - 1)$ tuples of $(E, 0)$ if i is even or $(0, E)$ otherwise.
(3) Insert q tuples of $(E, 0)$ if $\left(\frac{E}{d} - 1\right)$ is even or q tuples of $(0, E)$ otherwise.

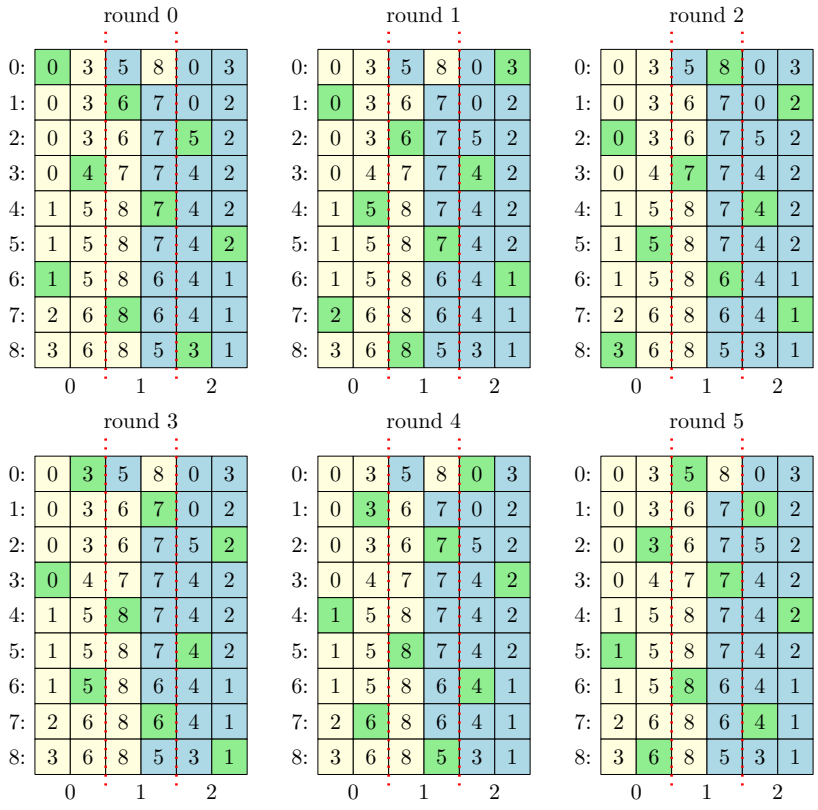

Figure 3: An example of shared memory accesses performed by a warp in `CF-Merge` for $w = 9$, $E = 6$, and $d = 3$ (i.e., not coprime) on an arbitrary input. Elements belonging to the A list (B list) are colored yellow (blue). The red dotted lines separate partitions of $wE/d = 16$ elements, that have been circular shifted by 0, 1, and 2 positions, respectively. Cell values correspond to the thread IDs that access that cell. Cells colored green are the cells accessed by the w threads in a particular round, demonstrating no bank conflicts in that round.

In total, we have inserted $2q + q\left(\frac{r}{d} - 1\right) + (q-1)\left(\frac{E}{d} - \frac{r}{d} - 1\right)$ tuples in addition to the $\left(\frac{E}{d} - 1\right)$ tuples of S. Hence, $|T| = \left(\frac{E}{d} - 1\right) + q\left(\frac{r}{d} + 1\right) + (q-1)\left(\frac{E}{d} - \frac{r}{d} - 1\right) = \frac{r}{d} + q\left(\frac{E}{d}\right) = \frac{w}{d}$.

Theorem 8. *Using the sequence T to assign elements to the $\frac{w}{d}$ threads in the subproblem of $\frac{wE}{d}$ elements, we can align accesses that result in*

$$\begin{cases} \frac{E^2}{d} & \text{if } E \le \frac{w}{2} \\ \frac{1}{2}\left(\frac{E^2}{d} + \frac{2Er}{d} + E - \frac{r^2}{d} - r\right) & \text{otherwise} \end{cases}$$

total bank conflicts.

Proof. In both cases, we count the number of bank conflicts in the last E shared memory banks, $\{w - E, w - E + 1, \ldots, w - 1\}$.

Case 1: Assume $1 < E \le \frac{w}{2} \iff q > 1$. Since $q > 1$, every column ends with (at least) a single scan of E elements, thus, resulting in a total of $E\left(\frac{E}{d}\right) = \frac{E^2}{d}$ bank conflicts.

Case 2: Assume $\frac{w}{2} < E \le w \iff q = 1$. From Lemma 7, we know that there are $\left(\frac{E}{d} - 1 - \left(\frac{E}{d} - \frac{r}{d}\right)\right) = \left(\frac{r}{d} - 1\right)$ pairs that sum up to r; and $\left(\frac{E}{d} - \frac{r}{d} - 1\right)$ pairs that sum up to $E + r = w$. Hence, $\left(\frac{r}{d} - 1\right)$ columns end with a single scan of E elements; and $\left(\frac{E}{d} - \frac{r}{d} - 1\right)$ columns end with a partial scan of elements.

Recall that for $i \in \{1, 2, \ldots, \frac{E}{d} - 2\}$, $x_i + y_{i+1} = E + r = w$ if $x_i > r \iff s_i < \frac{E}{d} - \frac{r}{d}$. Thus, $(x_i - r)$ elements are misaligned in the corresponding column. In total, there are

$$\begin{aligned} \sum_{i=1}^{\frac{E}{d} - \frac{r}{d} - 1} x_i - r &= \sum_{i=1}^{\frac{E}{d} - \frac{r}{d} - 1} \left(\left(\frac{E}{d} - s_i\right) d - r\right) \\ &= \sum_{i=1}^{\frac{E}{d} - \frac{r}{d} - 1} \left(\left(\frac{E}{d} - \frac{r}{d} - s_i\right) d\right) \\ &= d + 2d + 3d + \ldots + \left(\frac{E}{d} - \frac{r}{d} - 1\right) d \\ &= d \sum_{i=1}^{\frac{E}{d} - \frac{r}{d} - 1} i \\ &= \frac{1}{2}\left(\frac{E^2}{d} - \frac{2Er}{d} - E + \frac{r^2}{d} + r\right) \end{aligned}$$

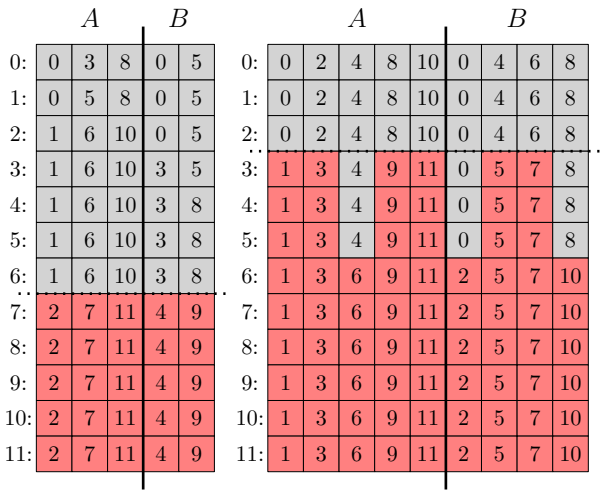

Figure 4: Visualization of the worst case inputs for Thrust mergesort (using our generalized strategy described in Section 4) for $w = 12$. The left figure shows the inputs for $E = 5$ (i.e., coprime) and the right shows $E = 9$ (i.e., not coprime). Cells are labeled with the thread ID that accesses that cell. In both figures, the elements residing in the last E memory banks are aligned to cause bank conflicts when threads perform sequential merge in shared memory. Red cells correspond to the accesses that contribute to the worst-case number of bank conflicts.

misaligned elements. Therefore, there are a total of $\frac{1}{2}\left(\frac{E^2}{d} + \frac{2Er}{d} + E - \frac{r^2}{d} - r\right)$ bank conflicts. Note that $d = E \implies r = 0$ and, consequently, $\frac{1}{2}\left(\frac{E^2}{d} - \frac{2Er}{d} - E + \frac{r^2}{d} + r\right) = \frac{1}{2}(E - E) = 0$ elements are misaligned. □

For the symmetric case, where the subproblem contains $\left\lfloor \frac{E}{2d} \right\rfloor w$ elements of A and $\left\lceil \frac{E}{2d} \right\rceil w$ elements of B, the tuple values of T are switched. Therefore, combining all d subproblems together results in a total of

$$\begin{cases} E^2 & \text{if } 1 < E \leq \frac{w}{2} \\ \frac{1}{2}\left(E^2 + 2Er + Ed - r^2 - rd\right) & \text{otherwise} \end{cases}$$

bank conflicts.

5 EXPERIMENTS

We evaluate the performance of our bank conflict free load-balanced dual subsequence gather by incorporating the algorithm into the implementation of pairwise merge sort provided in Thrust [38], which we refer to as `CF-Merge`. In our experiments, we compare `CF-Merge` to the unmodified Thrust mergesort implementation on both uniform random inputs and the constructed worst-case inputs from Section 4.

Berney and Sitchinava [8] observed that Thrust uses the software parameters $E = 17$ and $u = 256$, while the parameters $E = 15$ and $u = 512$ provide better performance, on random inputs. This performance difference is attributed to the corresponding occupancy,[6] with $E = 15$ and $u = 512$ providing the optimal 100% theoretical occupancy. Likewise, we compare the performance of these software parameters. Note that for values of E that are not coprime with $w = 32$, the performance of Thrust is much worse, while the runtime of `CF-Merge` will not be affected. Therefore, we are presenting experiments for the values of E that make Thrust run the fastest. Other values of E would make `CF-Merge` look comparably even better.

Our implementation of `CF-Merge` [9] uses the thread block approach described in Section 3.3. As the permutations performed only rely on information on the total size of each list, each thread block reorders elements during the initial transfer from global memory into shared memory. Furthermore, because both $E = 15$ and $E = 17$ are coprime with $w = 32$, only the coprime variant is implemented. Once elements have been read into register space via the load-balanced dual subsequence gather, threads process elements internally. In practice, on NVIDIA GPUs using the CUDA compiler, register memory requires static access as dynamic access to internal data are instead compiled into local memory space. One solution is to use data-oblivious algorithm to implement the merge in the registers and in our implementation we adopt the odd-even transposition sort [25] provided in Thrust.

We conduct experiments using $n = \{2^i E : 16 \leq i \leq 26\}$ 4-byte integers on a NVIDIA RTX 2080 Ti featuring 4,352 total physical cores, 11 GB of global memory, and 96 KiB of unified L1 cache and shared memory (configured to be 32 KiB of L1 cache and 64 KiB of shared memory, or vice versa) per streaming multiprocessor (SM).[7] All code is written using CUDA 11 [37] and compiled with the `-O3` and `-use_fast_math` optimization flags. Runtimes are recorded via `cudaEventRecord`, with the average across 10 runs being reported.

5.1 Results

Figure 5 shows the throughput results for both software parameters on the constructed worst-case inputs. Results show that on these inputs, `CF-Merge` provides an average, mean, and maximum speedup of 1.37, 1.45, and 1.47 for $E = 15$ and $u = 512$; and 1.17, 1.23, and 1.25 for $E = 17$ and $u = 256$. This highlights the performance benefits of `CF-Merge`, which uses the bank conflict free load-balanced dual subsequence gather, compared to the unmodified Thrust implementation, which on these inputs incurs the asymptotic worst-case number of bank conflicts. In contrast, on random inputs `CF-Merge` achieves performance comparable to the unmodified Thrust, which has been empirically shown previously to incur a small constant number of bank conflicts (2 to 3) [29]. This illustrates that the runtime overhead associated with performing the load-balanced dual subsequence gather is only 2-3 additional memory accesses per element. Using NVIDIA's `nvprof` profiler we confirmed that our implementation produces no bank conflicts during merging.

Overall, these results validate that `CF-Merge` eliminates the observed slowdown incurred by bank conflicts in shared memory, thereby providing fast runtimes on all possible inputs. Results for

[6] Ratio of active warps to the maximum number of active warps, per SM.
[7] GB = 10^9 bytes and KiB = 2^{10} bytes.

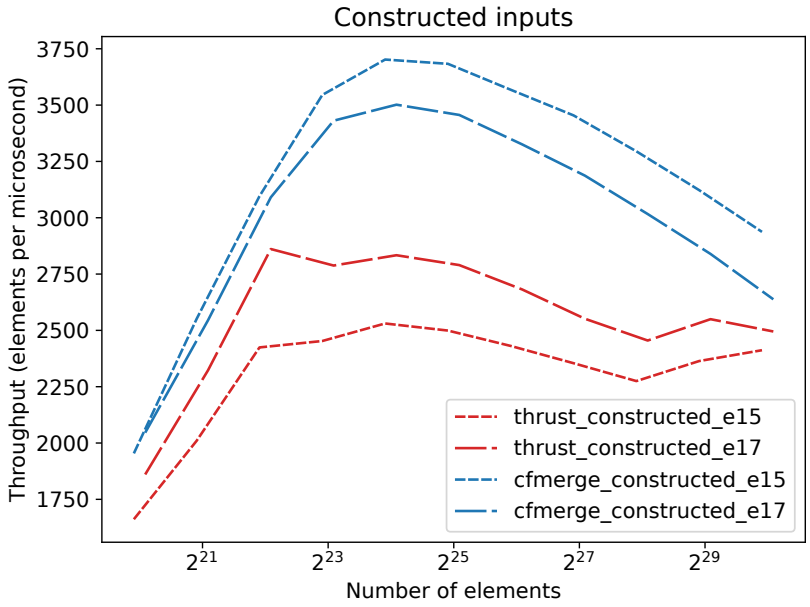

Figure 5: Throughput results (in elements per microsecond) for Thrust and `CF-Merge` on a NVIDIA RTX 2080 Ti using the constructed worst-case inputs. Thrust results are in red and `CF-Merge` results are in blue. The short dashed lines represent software parameters $E = 15$ and $u = 512$; and the long dashed lines represent software parameters $E = 17$ and $u = 256$. The x-axis is displayed on a logarithmic scale.

both the constructed worst-case inputs and random inputs are shown for each software parameter in Figure 6.

6 CONCLUSION

In this paper, we address the challenges associated with shared memory performance and the analysis of worst-case scenarios caused by bank conflicts in GPU algorithms. Leveraging the Distributed Memory Machine and principles from number theory we provide a framework for designing and analyzing bank conflict free algorithms on GPUs. In particular, we show that by using a simple technique we are able to eliminate all bank conflicts in Thrust library's implementation of mergesort.

Often in practice, algorithms designed to be efficient in the worst case perform significantly worse than randomized algorithm. So it is even more surprising that our technique results in virtually the same running time as a heuristic approach in a highly-tuned library implementation on a random input.

Our proposed approach, the load-balanced dual subsequence gather, eliminates bank conflicts by efficiently loading contiguous subarrays from shared memory into register files of individual threads and once the data is in the registers, it can be processed sequentially by individual threads without any bank conflicts. Observe that while the subarrays are merged in case of the mergesort, once they are in registers, they can also be processed in some other way, depending on the needs of a specific problem. Thus, our approach can be used to convert any algorithm that involves a parallel scan of a pair of arrays into a bank conflict free algorithm.

A natural question to ask is whether we should care about designing bank conflict free algorithms or focus on performance of the implementations on the worst case inputs, if simple heuristics (like selecting values of parameters that are coprime with the number of memory banks) already perform well on random or "real world" inputs? While this can be viewed as a philosophical question, every undergraduate algorithms course emphasizes worst case analysis of sequential algorithms. So why should it be ignored for algorithms for GPUs? Moreover, bank conflict free algorithms are much easier to analyze because without bank conflicts shared memory analysis becomes equivalent to the analysis of PRAM algorithms. Thus, bank conflict free algorithms allow us to better predict how the existing collection of parallel algorithms would perform on GPUs.

REFERENCES

[1] Peyman Afshani and Nodari Sitchinava. 2015. Sorting and Permuting without Bank Conflicts on GPUs. In *European Symposium on Algorithms (Lecture Notes in Computer Science, Vol. 9294)*. Springer, 13–24.
[2] Alok Aggarwal and Jeffrey Scott Vitter. 1988. The Input/Output Complexity of Sorting and Related Problems. *Commun. ACM* 31, 9 (1988), 1116–1127.
[3] George E. Andrews. 1994. *Number Theory*. Dover Publications, Inc.
[4] Lars Arge, Michael T. Goodrich, Michael J. Nelson, and Nodari Sitchinava. 2008. Fundamental Parallel Algorithms for Private-cache Chip Multiprocessors. In *Symposium on Parallelism in Algorithms and Architectures*. ACM, 197–206.
[5] Saman Ashkiani, Andrew A. Davidson, Ulrich Meyer, and John D. Owens. 2017. GPU Multisplit: An Extended Study of a Parallel Algorithm. *ACM Trans. Parallel Comput.* 4, 1 (2017), 2:1–2:44.
[6] Eli Ben-Sasson, Matan Hamilis, Mark Silberstein, and Eran Tromer. 2016. Fast Multiplication in Binary Fields on GPUs via Register Cache. In *International Conference on Supercomputing*. ACM, 35:1–35:12.
[7] Kyle Berney, Henri Casanova, Ben Karsin, and Nodari Sitchinava. 2022. Beyond Binary Search: Parallel In-Place Construction of Implicit Search Tree Layouts. *IEEE Trans. Computers* 71, 5 (2022), 1104–1116.
[8] Kyle Berney and Nodari Sitchinava. 2020. Engineering Worst-Case Inputs for Pairwise Merge Sort on GPUs. In *International Parallel and Distributed Processing Symposium*. IEEE Computer Society, 1133–1142.

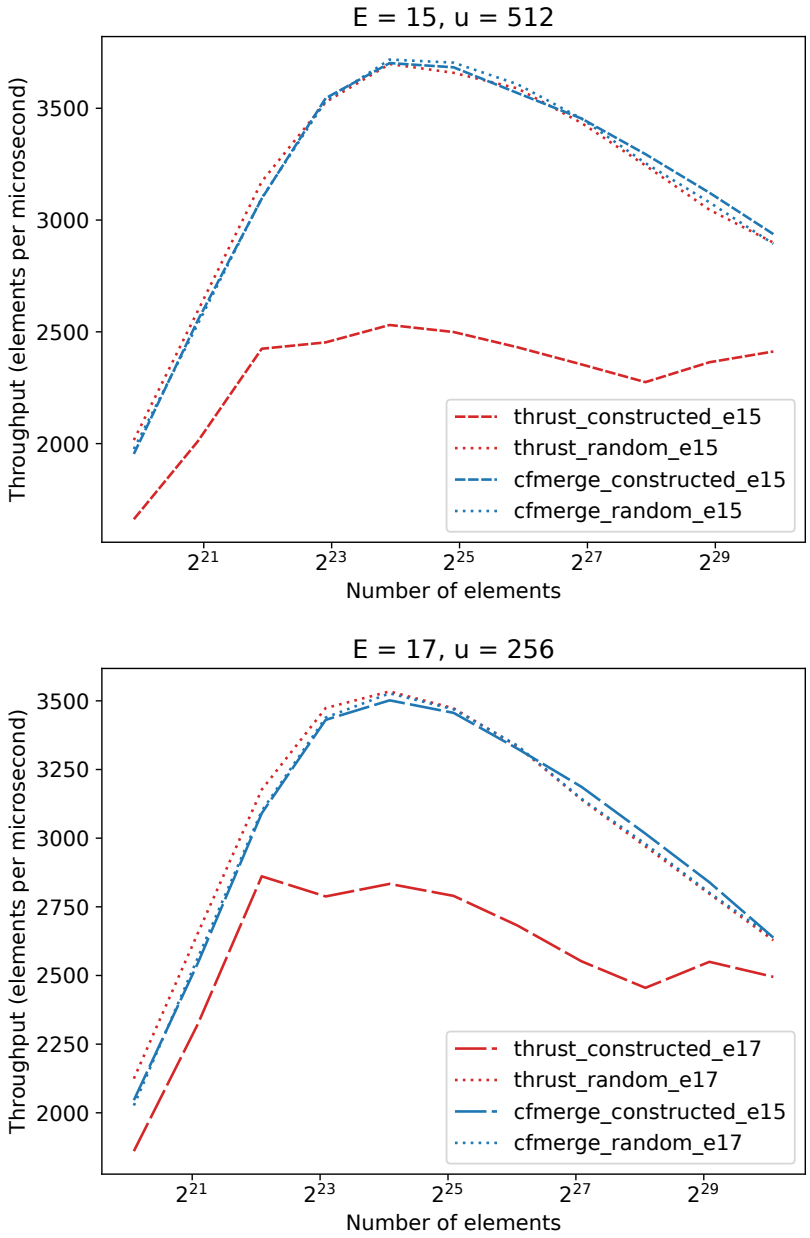

Figure 6: Throughput results (in elements per microsecond) for Thrust and `CF-Merge` on a NVIDIA RTX 2080 Ti using parameters $E = 15$ and $u = 512$ (top), and $E = 17$ and $u = 256$ (bottom). Thrust results are in red and `CF-Merge` results are in blue. The dashed lines represent the constructed worst-case inputs and the dotted lines represent uniform random inputs. The x-axis is displayed on a logarithmic scale.

[9] Kyle Berney and Nodari Sitchinava. 2025. CF-Merge: Bank Conflict Free GPU Mergesort. https://github.com/algoparc/GPU-CFMerge/.

[10] Federico Busato and Nicola Bombieri. 2016. An Efficient Implementation of the Bellman-Ford Algorithm for Kepler GPU Architectures. *IEEE Trans. Parallel Distributed Syst.* 27, 8 (2016), 2222–2233.

[11] Bryan Catanzaro, Alexander Keller, and Michael Garland. 2014. A Decomposition for In-place Matrix Transposition. In *Symposium on Principles and Practice of Parallel Programming*. ACM, 193–206.

[12] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. 2022. *Introduction to Algorithms* (4nd ed.). McGraw-Hill Higher Education.

[13] Artur Czumaj, Friedhelm Meyer auf der Heide, and Volker Stemann. 1995. Improved Optimal Shared Memory Simulations, and the Power of Reconfiguration. In *Israel Symposium on Theory of Computing and Systems*. IEEE Computer Society, 11–19.

[14] Artur Czumaj, Friedhelm Meyer auf der Heide, and Volker Stemann. 1995. Shared Memory Simulations with Triple-Logarithmic Delay. In *European Symposium on Algorithms (Lecture Notes in Computer Science, Vol. 979)*. Springer, 46–59.

[15] Artur Czumaj, Friedhelm Meyer auf der Heide, and Volker Stemann. 2000. Contention Resolution in Hashing Based Shared Memory Simulations. *SIAM J. Comput.* 29, 5 (2000), 1703–1739.

[16] Andrew A. Davidson, Sean Baxter, Michael Garland, and John D. Owens. 2014. Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths. In *International Parallel and Distributed Processing Symposium*. IEEE Computer Society, 349–359.
[17] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. 1993. Simple, Efficient Shared Memory Simulations. In *Symposium on Parallel Algorithms and Architectures*. ACM, 110–119.
[18] Yuri Dotsenko, Naga K. Govindaraju, Peter-Pike J. Sloan, Charles Boyd, and John Manferdelli. 2008. Fast Scan Algorithms on Graphics Processors. In *International Conference on Supercomputing*. ACM, 205–213.
[19] Thomas L. Falch and Anne C. Elster. 2014. Register Caching for Stencil Computations on GPUs. In *Symbolic and Numeric Algorithms for Scientific Computing*. IEEE, 479–486.
[20] Isaac Gelado and Michael Garland. 2019. Throughput-oriented GPU Memory Allocation. In *Symposium on Principles and Practice of Parallel Programming*. ACM, 27–37.
[21] Dominik Göddeke and Robert Strzodka. 2011. Cyclic Reduction Tridiagonal Solvers on GPUs Applied to Mixed-Precision Multigrid. *IEEE Trans. Parallel Distributed Syst.* 22, 1 (2011), 22–32.
[22] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir. 1983. The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer. *IEEE Trans. Computers* 32, 2 (1983), 175–189.
[23] Chunyang Gou and Georgi Gaydadjiev. 2013. Addressing GPU On-Chip Shared Memory Bank Conflicts Using Elastic Pipeline. *International Journal of Parallel Programming* 41, 3 (2013), 400–429.
[24] Oded Green, Robert McColl, and David A. Bader. 2012. GPU Merge Path: A GPU Merging Algorithm. In *International Conference on Supercomputing*. ACM, 331–340.
[25] A. Nico Habermann. 1972. *Parallel Neighbor-Sort (or the Glory of the Induction Principle)*. Technical Report AD-759 248. Carnegie Mellon University.
[26] Joseph F. JáJá. 1992. *An Introduction to Parallel Algorithms*. Addison-Wesley.
[27] Richard M. Karp, Michael Luby, and Friedhelm Meyer auf der Heide. 1992. Efficient PRAM Simulation on a Distributed Memory Machine. In *Symposium on Theory of Computing*. ACM, 318–326.
[28] Ben Karsin, Henri Casanova, and Nodari Sitchinava. 2015. Efficient Batched Predecessor Search in Shared Memory on GPUs. In *High Performance Computing*. IEEE Computer Society, 335–344.
[29] Ben Karsin, Volker Weichert, Henri Casanova, John Iacono, and Nodari Sitchinava. 2018. Analysis-driven Engineering of Comparison-based Sorting Algorithms on GPUs. In *International Conference on Supercomputing*. ACM, 86–95.
[30] Akihiko Kasagi, Koji Nakano, and Yasuaki Ito. 2012. An Implementation of Conflict-Free Offline Permutation on the GPU. In *International Conference on Networking and Computing*. IEEE Computer Society, 226–232.
[31] Kurt Mehlhorn and Uzi Vishkin. 1984. Randomized and Deterministic Simulations of PRAMs by Parallel Machines with Restricted Granularity of Parallel Memories. *Acta Inf.* 21 (1984), 339–374.
[32] Duane Merrill and Andrew S. Grimshaw. 2011. High Performance and Scalable Radix Sorting: a Case Study of Implementing Dynamic Parallelism for GPU Computing. *Parallel Processing Letters* 21, 2 (2011), 245–272.
[33] Friedhelm Meyer auf der Heide, Christian Scheideler, and Volker Stemann. 1996. Exploiting Storage Redundancy to Speed up Randomized Shared Memory Simulations. *Theor. Comput. Sci.* 162, 2 (1996), 245–281.
[34] Koji Nakano. 2012. Simple Memory Machine Models for GPUs. In *International Parallel and Distributed Processing Symposium, Workshops and PhD Forum*. IEEE Computer Society, 794–803.
[35] NVIDIA. 2022. CUDA C Best Practices Guide. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html.
[36] NVIDIA. 2022. CUDA C Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.
[37] NVIDIA. 2022. CUDA Toolkit Documentation v11. https://docs.nvidia.com/cuda/index.html.
[38] NVIDIA. 2022. Thrust: The C++ Parallel Algorithms Library. https://github.com/thrust/thrust.
[39] Nadathur Satish, Mark J. Harris, and Michael Garland. 2009. Designing Efficient Sorting Algorithms for Manycore GPUs. In *International Symposium on Parallel and Distributed Processing*. IEEE, 1–10.
[40] Eli Upfal and Avi Wigderson. 1987. How to Share Memory in a Distributed System. *J. ACM* 34, 1 (1987), 116–127.
[41] Jie Wang, Xinfeng Xie, and Jason Cong. 2017. Communication Optimization on GPU: a Case Study of Sequence Alignment Algorithms. In *International Parallel and Distributed Processing Symposium*. IEEE Computer Society, 72–81.
[42] Shucai Xiao and Wu-chun Feng. 2010. Inter-block GPU Communication via Fast Barrier Synchronization. In *International Symposium on Parallel and Distributed Processing*. IEEE Computer Society, 1–12.
[43] Yunlong Xu, Lan Gao, Rui Wang, Zhongzhi Luan, Weiguo Wu, and Depei Qian. 2016. Lock-based Synchronization for GPU Architectures. In *Computing Frontiers*. ACM, 205–213.
[44] Yao Zhang, Jonathan Cohen, and John D. Owens. 2010. Fast Tridiagonal Solvers on the GPU. In *Symposium on Principles and Practice of Parallel Programming*, R. Govindarajan, David A. Padua, and Mary W. Hall (Eds.). ACM, 127–136.

A NUMBER THEORY

Lemma 9. *(Euclid's Division Lemma) For any integers a and b such that $b > 0$, there exists unique integers q and r such that $0 \leq r < b$ and $a = qb + r$.*

Definition 10. *Let $a, b \in \mathbb{Z}$ and $d \in \mathbb{Z}^+$, such that $a \neq b \neq 0$, We say d is the* greatest common divisor *of a and b, denoted $\text{GCD}(a, b)$, if*

(1) *d is a common divisor of a and b; and*
(2) *any integer c that is a common divisor of a and b is also a divisor of d.*

Theorem 11. *Let $a, b \in \mathbb{Z}$, such that $a \neq b \neq 0$, $\text{GCD}(a, b)$ exists and is unique.*

Definition 12. *Let $a, b \in \mathbb{Z}$, such that $a \neq b \neq 0$. If $\text{GCD}(a, b) = 1$, then a and b are* coprime *(also known as* relatively prime *or* mutually prime*).*

Definition 13. *Let $m \in \mathbb{Z}^+$. The set $R = \{r_0, r_1, \ldots, r_{m-1}\}$ is a* complete residue system *modulo m if the following are satisfied:*

(1) *For each $i, j \in \{0, 1, \ldots, m-1\}$ such that $i \neq j$, $r_i \not\equiv r_j \pmod{m}$.*
(2) *For each $n \in \mathbb{Z}$, there exists $r_i \in R$ such that $n \equiv r_i \pmod{m}$.*

Corollary 14. *Let $m \in \mathbb{Z}^+$. The set $\mathbb{Z}_m = \{0, 1, 2, \ldots, m-1\}$ is a complete residue system.*

Definition 15. *Let $a, b \in \mathbb{Z}$ and $m \in \mathbb{Z}^+$. If $ab \equiv 1 \pmod{m}$, then b is an* inverse *of a modulo m.*

Corollary 16. *Let $n \in \mathbb{Z}$ and $m \in \mathbb{Z}^+$. If $\text{GCD}(n, m) = 1$, then n has a single unique inverse modulo m.*

Corollary 17. *Let $a, b \in \mathbb{Z}^+$, such that $a \geq b$, and let $q, r \in \mathbb{Z}$ such that, $a = qb + r$. $\text{GCD}(a, b) = \text{GCD}(b, r)$.*

Proof. For ease of notation, let $d = \text{GCD}(a, b)$. By definition, $d \mid a$ and $d \mid b$, hence, there exists positive integers x, y such that $a = dx$ and $b = dy \implies r = a - qb = d(x - qy)$. Thus, $d = \text{GCD}(a, b) \mid r$ and $\text{GCD}(a, b) \leq \text{GCD}(b, r)$. We use a similar argument to show that $\text{GCD}(b, r) \mid a$ and $\text{GCD}(b, r) \leq \text{GCD}(a, b)$. Therefore, $\text{GCD}(a, b) = \text{GCD}(b, r)$. □

Corollary 18. *Let $a, b \in \mathbb{Z}$ and $d = \text{GCD}(a, b)$, $\text{GCD}\left(\frac{a}{d}, \frac{b}{d}\right) = 1$.*

Proof. Assume c is a positive common divisor of a and b such that $c \mid \frac{a}{d}$ and $c \mid \frac{b}{d}$. In other words, $\frac{a/d}{c} \in \mathbb{Z}^+$ and $\frac{b/d}{c} \in \mathbb{Z}^+$. Hence, there exists $x, y \in \mathbb{Z}^+$ such that $\frac{a}{d} = cx \implies a = cdx$ and $\frac{b}{d} = cy \implies b = cdy$. Thus, cd is a positive common divisor of a and b. Since d is the greatest common divisor of a and b, c must be equal to 1. Therefore, $c = 1$ is the greatest common divisor of $\frac{a}{d}$ and $\frac{b}{d}$. □

B SUPPLEMENTAL FIGURES

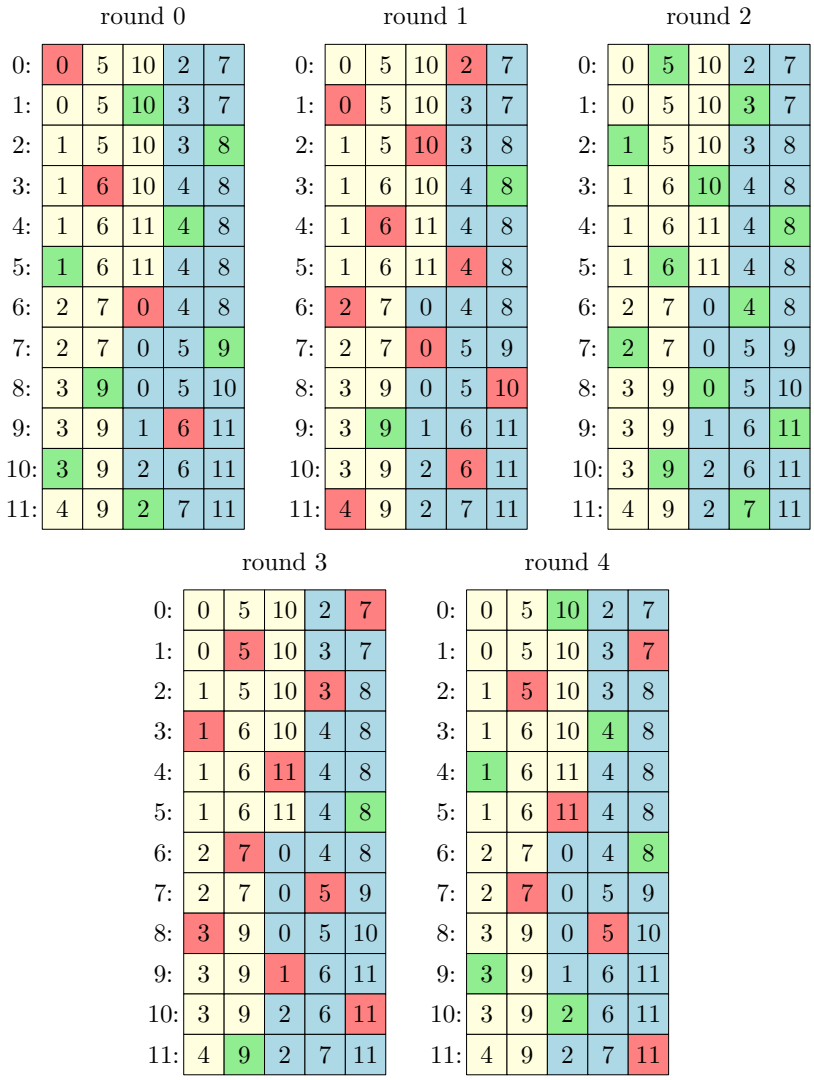

Figure 7: Depiction of the read stalls caused by threads in a warp accessing up to 2 elements per round for $w = 12$, $E = 5$, and $d = 1$ (i.e., coprime) on arbitrary input. Cell numbers correspond to the thread that performs the access. Elements colored red cause a stall due to threads needing to access 2 elements in each round.

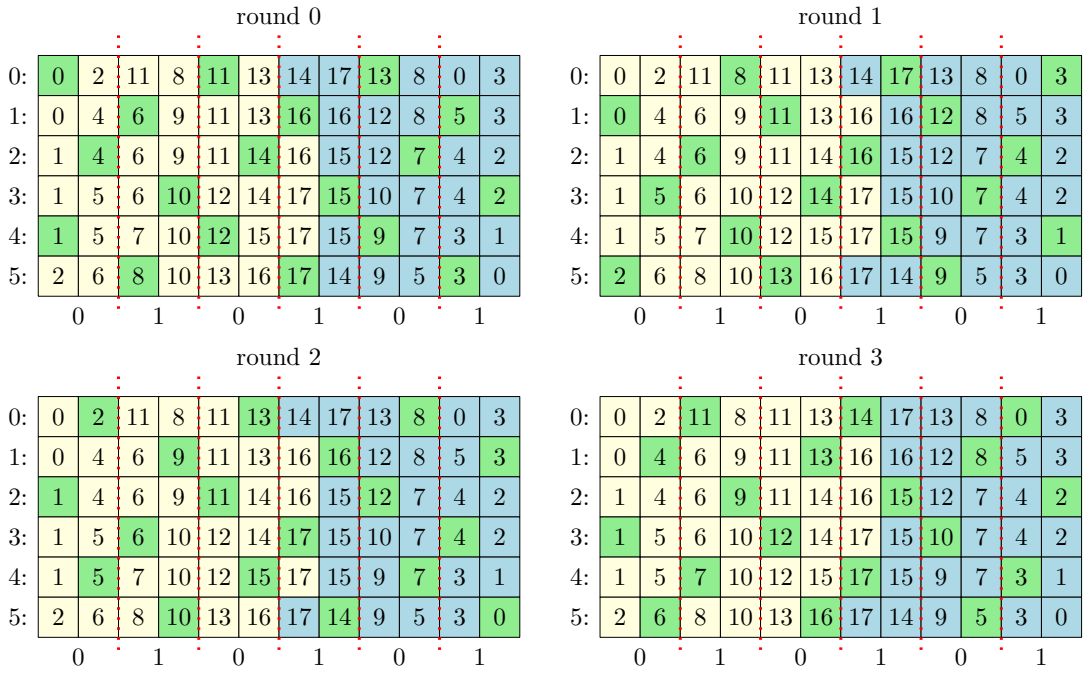

Figure 8: An example of shared memory accesses performed by a thread block in `CF-Merge` for $u = 18$, $w = 6$, $E = 4$, and $d = 2$ (i.e., not coprime) on an arbitrary input. Elements belonging to the A list (B list) are colored yellow (blue). The red dotted lines separate partitions of $wE/d = 12$ elements, that have been circular shifted by 0 and 1 positions, respectively. Cell values correspond to the thread IDs that access that cell. Cells colored green are the cells accessed by the threads in a particular round, demonstrating no bank conflicts in that round. Recall that bank conflicts potentially occur only by accesses by the threads of the same warp, i.e., $\{0, \ldots, 5\}$, $\{6, \ldots, 11\}$, $\{12, \ldots, 17\}$.