Redzone stream compaction: removing k items from a list in parallel O(k) time

JOHAN BONTES, Computer science, University of Cape Town, Rondebosch, South Africa
JAMES GAIN, Computer science, University of Cape Town, Rondebosch, South Africa

Stream compaction, the parallel removal of selected items from a list, is a fundamental building block in parallel algorithms. It is extensively used, both in computer graphics, for shading, collision detection, and ray tracing, as well as in general computing, such as for tree traversal and database selection.

In this article, we present Redzone stream compaction, the first parallel stream compaction algorithm to remove k items from a list with $n \geq k$ elements in $O(k)$ rather than $O(n)$ time. Based on our benchmark experiments on both GPU and CPU, if k is proportionally small ($k \ll n$), Redzone outperforms existing parallel stream compaction by orders of magnitude, while if k is close to n, it underperforms by a constant factor. Redzone removes items in-place and needs only $O(1)$ auxiliary space. However, unlike current $O(n)$ algorithms, it is unstable (i.e., the order of elements is not preserved) and it needs a list of the items to be removed.

CCS Concepts: • **Theory of computation** → **Massively parallel algorithms**;

Additional Key Words and Phrases: Stream compaction, list removal

ACM Reference Format:
Johan Bontes and James Gain. 2024. Redzone stream compaction: removing k items from a list in parallel O(k) time. *ACM Trans. Parallel Comput.* 11, 3, Article 14 (August 2024), 16 pages. https://doi.org/10.1145/3675782

1 Introduction

Stream compaction, removing multiple elements from a list in parallel, is a fundamental primitive in many parallel algorithms. In sequential algorithms, this is known as stable list removal. It is used when filtering data, for example, in collision detection [6] or when culling data elements that are no longer needed, such as in Kd tree construction [27]. As a stable algorithm it operates on a sequential array, where items to be deleted are marked, either in the array itself (as shown in Figure 1), using an auxiliary stencil array, or by a remove predicate.

Stable stream compaction needs one [23] or multiple [4, 9, 18] sequential read passes. Stability has the benefit of preserving the order of elements, but comes at a cost. If the output is a contiguous array—and not, say, a linked list— then it must process at least $n - k$ elements, because all these elements need to be moved. If not using a *keep* list, then $O(n)$ reads are required to distinguish "keep" from "remove" items, implying $O(\frac{n}{p})$ parallel time on a machine with p processors.

This work was financially supported by the Hasso Plattner Institute for Digital Engineering, through the HPI Research School at UCT.

Authors' Contact Information: Johan Bontes, Computer Science, University of Cape Town, Rondebosch, South Africa; e-mail: johan@digitsolutions.nl; James Gain, Computer Science, University of Cape Town, Rondebosch, South Africa; e-mail: james.gain@uct.ac.za.

ACM 2329-4949/2024/08-ART14
https://doi.org/10.1145/3675782

Contribution. Our Redzone algorithm is the first parallel stream compaction method to delete k items from a list $\mathcal{A}$ of size n in $O(k)$ time rather than $O(n)$. Operations need only $O(1)$ space. The input is $\mathcal{A}$ and a list $\mathcal{R}$ containing k indices to be removed. It updates $\mathcal{A}$ in-place. However, it is unstable in that it does not preserve the order of elements. Redzone cannot perform out-of-place compaction. If $k \ll n$, then it outperforms $O(n)$ compaction by orders of magnitude.

Reference implementations in C++ for both CPUs and NVidia's CUDA, as well as the source code for our benchmarking experiments, are available on Github under an MIT license.[1]

Article Structure. After the Introduction, Section 2 explains stream compaction, and Section 3 discusses previous work on stream compaction. Section 4 details our proposed Redzone algorithm, its time and space complexity, synchronization issues, proof for its correctness, as well as how to extend the algorithm to add and remove items concurrently. The Experimental results in Section 5 benchmark Redzone against the current state-of-the-art using both GPU and CPU algorithms. The Conclusion summarizes, highlights some observed opportunities to improve performance, and lists areas for future research.

2 Concept

Assuming a *keep* predicate indicating elements to be retained, stable stream compaction [19], running in $O(n)$ sequential or $O(n/p)$ parallel time can be coded as follows (illustrated in Figure 1, top):

```
dest = 0
for a in A: if keep(a): A'[dest++] = a
```
(1)

However, when $k \ll n$, traversing all n items is wasteful. Instead, a "remove" list can be used. Using such a list, the element indexed by r is deleted as follows:

```
A[r] = A[--n]
```
(2)

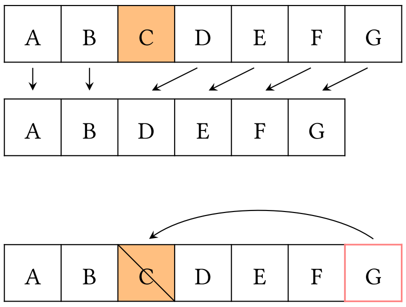

Fig. 1. Stable stream compaction (top) visits all items and copies valid ones to the output preserving list order. Unstable stream compaction (bottom) uses items at the end of the list to replace deletions elsewhere.

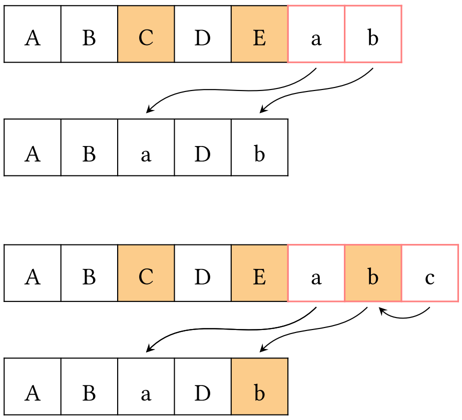

Fig. 2. Items in the red zone can be used to fill holes due to deletions outside the red zone (top). Naïvely filling holes this way fails if the red zone itself contains an item (b) to be deleted (bottom).

This deletes the element at index r in $O(1)$ time by overwriting destination $\mathcal{A}[r]$ with source $\mathcal{A}[n-1]$ (see Figure 1, bottom). Given a list $\mathcal{R}$ containing k indices to be removed, all items in $\mathcal{R}$ can be removed from $\mathcal{A}$ in $O(k)$ time by using k tail items to fill in holes. We call this tail the *red zone* ($\mathcal{Z}$).

The drawbacks are that removal is unstable, causing elements to be reordered (see Figure 2), and that writes are scattered, which may be computationally expensive.

Naïvely filling holes caused by deletions using items in the red zone will work only so long as no items in the red zone are deleted (Figure 2, top). This scheme fails when deleting items in the red zone itself (shown in Figure 2, bottom). In this example, two errors occur, because red zone item

[1]https://github.com/JBontes/redzone.html

(b) is incorrectly used as a source, causing keep item (c) to be discarded and remove item (b) to be kept. Our proposed algorithm solves this issue with extra bookkeeping.

3 Background and Related Work

Parallel stream compaction is a common primitive operation, widely used in contexts such as graphics [6, 8, 24], simulation [17], and data stores [15, 25]. It removes, in parallel, k elements from a contiguous list $\mathcal{A}$ containing $|\mathcal{A}| = n$ items. The items to be removed can be marked within $\mathcal{A}$ itself, using a Boolean stencil array of length n containing `true` for entries to be removed or by a predicate function. In all these cases $O(n)$, keep/remove queries are required. Stream compaction outputs a list $\mathcal{A}'$ of length $n - k$ with elements in the same order as $\mathcal{A}$ if the method is stable and in an arbitrary order if it is unstable. The primitive is so common that Segura et al. [20] even proposed an implementation in GPU hardware.

Stream compaction is trivial to perform on a sequential machine (see Listing (1)). In parallel code, the main problem is sequencing to ensure writes have no gaps or overlaps due to race conditions. This can be done using atomic counters, but stability is easier to achieve with *prefix sums* [7, 11, 22], also known as *scans*: running totals used to parallelize operations. Blelloch [5] shows how prefix sums can be used to parallelize tasks that seem inherently sequential.

Horn first published a stable compaction algorithm in 2005 for early GPUs without support for scattered writes that operates in $O(n \log n)$ time for both prefix sum calculation and writes. Sengupta et al. [21] reduced the time needed for prefix sums to $O(n)$, leaving write time unchanged. Roger et al. [18] used GPU hardware support for scattered writes to improve write time and thus overall time to $O(n)$. This is the approach used by the `remove` stream compaction function in NVidia's Thrust library [2]. The above improvements were aided by advances in GPU architecture in the early 2000s. GPUs now have feature parity with CPUs so many parallel algorithms can run on both [12, 14].

One important implementation detail of stream compaction is that multiple reads are required to achieve global synchronization if parallel components do not have access to a low latency communication channel. This happens when multiple distinct processors or machines are used. In such a case, one read is needed to gather data for the prefix sum and another to collect valid elements and store them in the correct locations. Synchronization can be done by splitting execution phases into subprograms (known as *kernels*), each launched in sequence. However, launching kernels incurs overhead. Hughes et al. [9] eliminate overhead in their InK-Compact algorithm by performing both stages in a single kernel. Our experiments show that their approach performs well when deleting the majority of items—on so-called *sparse streams*—but underperforms otherwise.

Moreira et al.'s Jumping Jack [13] is an unstable $O(n)$ algorithm that scans the list, creates a prefix sum, and calculates the maximum allowable size to find elements that can be used to fill in contiguous sections marked for deletion. They report that, unfortunately, this does not work well on sparse streams. Sun et al. [23] engineered a version that only needs a single read of the stream. This approach is easy to apply if all parallel threads can efficiently communicate but requires computationally expensive global synchronization if not. Billeter et al. [4] supplement the prefix sum with generation of a bitmask followed by a population count, thus reducing the number of sums needed. Shortly after their publication, GPUs supporting these operations in hardware became available. Bakunas-Milanowski et al. [1] combine prefixes with atomic operations, which do not sequence threads, resulting in fast unstable $O(n)$ compaction. Bernabé et al. [3] investigate the performance and power-efficiency of various $O(n)$ stream compaction algorithms on different CPU architectures and find that multiple low-cost computers perform better per watt than expensive multi-core CPUs.

ALGORITHM 1: Redzone stream compaction. Code on the same line runs in parallel. Dotted lines ···· mark synchronization points. This code needs low latency thread communication, such as shared memory, requiring all threads to run on the same (multi)processor on a GPU or CPU.

```
Input:  List $\mathcal{A}$, Removals $\mathcal{R}$
Output:  $\mathcal{A}' \leftarrow \mathcal{A} - \mathcal{R}$
 1: if $|\mathcal{R}| = 0$ or $|\mathcal{R}| = |\mathcal{A}|$ then return
 2: $z \leftarrow |\mathcal{A}| - |\mathcal{R}|$; $j \leftarrow 0$; $k \leftarrow 0$                ▷ red zone boundary
 3: for $r \in \mathcal{R}$ do                                   ▷ phase 1: mark deleted items in $\mathcal{Z}$
 4:     if $r \geq z$ then MarkAsDeleted($\mathcal{A}[r]$)
    ..................................................................................
 5: for $i \in \{0 \dots |\mathcal{R}| - 1\}$ do                     ▷ phase 2: fill holes, record orphans
 6:     $a \leftarrow \mathcal{A}[z + i]$; $r \leftarrow \mathcal{R}[i]$
 7:     if $r \geq z$ and isDeleted($a$) then do nothing          ▷ $r \in \mathcal{Z}$ and $a$ is invalid
 8:     else if $r \geq z$ and not isDeleted($a$) then           ▷ $r \in \mathcal{Z}$, but $a$ is valid
 9:         $j \leftarrow j$ + PrefixSumToIndex($i$)
    ...............................................
10:         $\mathcal{Z}'[j] \leftarrow a$                                  ▷ keep orphaned $a$
11:     else if $r < z$ and isDeleted(a) then                  ▷ $r$ is valid, but $a$ is not
12:         $k \leftarrow k$ + PrefixSumToIndex($i$)
    ...............................................
13:         $\mathcal{R}'[k] \leftarrow r$                                  ▷ keep orphaned $r$
14:     else                                                 ▷ both $r$ and $a$ are valid
15:         $\mathcal{A}'[r] \leftarrow a$                                  ▷ delete $\mathcal{A}[r]$
    ..................................................................................
16: for $i \in \{0 \dots j - 1\}$ do              ▷ phase 3: process orphans, note: $|\mathcal{R}'| = |\mathcal{Z}'|$
17:     $\mathcal{A}'[\mathcal{R}'[i]] \leftarrow \mathcal{Z}'[i]$
```

Interestingly, all the above papers view stream compaction in isolation without taking into account the preparation work of collecting removal items. Considering that this can take more time than the actual compaction, this leaves avenues for optimization unexplored.

Currently, no method for stream compaction is known that can remove items in $O(k)$ time.

4 Proposed Algorithm

Redzone compaction achieves $O(k)$ runtime, using three phases to: (1) mark elements as invalid sources; (2) perform easy deletions immediately and schedule hard ones for later; and (3) clean up.

Algorithm 1 shows an overview of the code, which is explained below. Circled numbers, such as ②, refer to line numbers in the algorithm. Dotted horizontal lines ···· show synchronization points.

Redzone takes as input a list of n items $\mathcal{A}$ and a list $\mathcal{R}$ of k indices into $\mathcal{A}$ to be deleted and outputs the result in-place into $\mathcal{A}'$: the first $n - k$ elements of $\mathcal{A}$. The use of $\mathcal{R}$ differs from stable compaction algorithms that mark $\mathcal{A}$ itself or use a stencil array of Booleans (of length n) to denote keep/remove items.

The algorithm deletes items from $\mathcal{A}$ by using k items from the tail of list $\mathcal{A}$. This tail is called *the red zone* ($\mathcal{Z}$) and contains source elements to overwrite the items slated for removal in $\mathcal{R}$ (15, 17). After taking care of the trivial case where all or no items are to be deleted ①, the start of the red zone is stored in variable z ②. $\mathcal{R}$ is iterated over ③ and any items $\mathcal{A}[r]$ pointing into the red zone are marked as deleted ④. This marking of red zone items is identical to the marking of deleted items in stable $O(n)$ stream compaction and can be done in-place, provided that unused state space is available in the data elements of $\mathcal{A}$. This ensures that deleted items are not used as sources to replace deletions elsewhere, preventing the issue illustrated in Figure 2. Because phase 1 performs scattered writes, our experiments (see Figure 6) show this to be computationally expensive. Ideally, this phase should be mixed in with work that generates the list of removal items $\mathcal{R}$ so the latency incurred by scattered writes can be hidden by other processing.

Table 1. Possible States for Destination r and Source a and the Implications Thereof

case	r	a	implications
A	$r \notin \mathcal{Z}$	valid	both a and r are valid, remove 1 item from both $\mathcal{R}_v$ and $\mathcal{Z}_v$; delete $\mathcal{A}[r] \leftarrow a$ ⑮.
B	$r \in \mathcal{Z}$	deleted	both a and r are invalid, remove 1 item from both $\mathcal{R}_d$ and $\mathcal{Z}_d$; quietly discard a and r ⑦.
C	$r \notin \mathcal{Z}$	deleted	r indexes a valid destination in $\mathcal{A}$, but a is not a valid source, delete 1 item from $\mathcal{R}_v$ and another from $\mathcal{Z}_d$; add orphaned r to $\mathcal{R}'$ and discard a ⑬.
D	$r \in \mathcal{Z}$	valid	r indexes the red zone and is not a valid destination, but a is a valid source, remove one item from $\mathcal{R}_d$ and another from $\mathcal{Z}_v$; add orphaned a to $\mathcal{Z}'$ and discard r ⑩.

Next, the algorithm moves to phase 2. For every item in $\mathcal{R}$ ⑤, a pairing is created between source $a : \mathcal{Z}[i]$ and destination $r : \mathcal{R}[i]$ ⑥. This is the last point where we read from input lists $\mathcal{R}$ and $\mathcal{Z}$, which allows us to reuse this space to store temporary data for orphans $\mathcal{R}'$ and $\mathcal{Z}'$ in-place, making Redzone an $O(1)$ space algorithm.

The source a and destination r items can individually be either valid or invalid, giving rise to four possible cases (7, 8, 11, 14) (detailed in Table 1).

If both are invalid, because (a) $r \in \mathcal{Z}$ $(r \geq z)$ is in the red zone and thus invalid as a destination, and (b) a is marked for deletion and thus invalid as a source, then no action is needed and both are quietly discarded ⑦.

If, however, $r \in \mathcal{Z}$ is not a valid destination, but a has not been marked for deletion ⑧, then a is a valid source without a destination: a *source orphan*. It is added to the in-place source orphan list $\mathcal{Z}'$ ⑩ for later processing in phase 3. These stores can be run fully in parallel on a GPU by first creating a bitmask of all threads where the if-predicate is satisfied (`ballot`), then performing an exclusive scan of the population count (Hamming weight) of these masks per warp[2] across a block ⑨ [4]. The starting index j per warp is the prefix sum supplemented by the preceding number of lanes in the bitmask; this is calculated as `popcount(bitmask & (1 << laneId) - 1)`, where `laneId` is 0 for the first and 31 for the last thread in a warp. Counters j and k (9, 12) might also be updated using atomic increments, but prefix sums are typically more efficient.

The symmetric case where r is not in the red zone and thus a valid destination, but a is not a valid source because it is marked as deleted, gives rise to a *destination orphan* and is stored in-place in $\mathcal{R}'$ for later processing (12, 13). Note that counts j and k run fully independently; the outer loop (5..15) can run parallel batches of any width. Because every read of a, r frees one item in $\mathcal{Z}, \mathcal{R}$, the write indices j, k into $\mathcal{Z}', \mathcal{R}'$, respectively, never run ahead of read index i. In fact, because if statements ⑧ and ⑪ are mutually exclusive, $j + k \leq i$, we could conceivably intersperse $\mathcal{Z}'$ and $\mathcal{R}'$. However, to simplify processing in phase 3, these two lists are stored separately.

If none of the three former cases apply, then both source a and destination r must be valid ⑬ and a is moved to $\mathcal{A}[r]$ ⑭.

Finally, in phase 3, every orphaned source a in $\mathcal{Z}'$ is moved to destination r in $\mathcal{R}'$. As per the proof in Section 4.5, the number of source orphans equals the destination count: $|\mathcal{Z}'| = |\mathcal{R}'|$. Lines (10, 13) ensure that orphan lists contain only valid items, hence the default processing from line ⑮ can be repeated with the orphans ⑰. No attempt is made to preserve the order of items in $\mathcal{A}'$. In

[2]A *warp* (also known as a *wavefront*) is a group of 32 or 64 threads on a GPU with hardware support for data sharing, akin to SIMD units in a CPU.

ALGORITHM 2: Redzone compaction for high-latency IPC.

```
Input:  List $\mathcal{A}$, Removals $\mathcal{R}$, Processors P
Output: $\mathcal{A}' \leftarrow \mathcal{A} - \mathcal{R}$
0:  $s \leftarrow \left\lceil \frac{|\mathcal{R}|}{|P|} \right\rceil; b_p \leftarrow s; b_{|P|-1} \leftarrow |R| - s(|\mathcal{R}| - 1); \text{start}_p \leftarrow (p)s; j_p \leftarrow k_p \leftarrow 0$    ▷ $s$ = batch size
                                            ▷ phase 1 is unchanged, note that lists start at zero
----------------------------------------------------------------------------------------------
5:  for $p, i_p \in P, \{\text{start}_p \,..\, \text{start}_p + b_p\}$ do     ▷ phase 2: data is segmented per block p
6:      $a \leftarrow \mathcal{A}[z + i_p]; r \leftarrow \mathcal{R}[i_p]$
7:      if $r \geq z$ and isDeleted($a$) then do nothing        ▷ $r \in \mathcal{Z}$ and $a$ is invalid
8:      else if $r \geq z$ and not isDeleted($a$) then          ▷ $r \in \mathcal{Z}$, but $a$ is valid
9:          $j_p \leftarrow j_p$ + PrefixSumToIndex($i_p, p$)
            .....................................................
10:         $\mathcal{Z}'[j_p] \leftarrow a$                             ▷ keep orphaned $a$
11:     else if $r < z$ and isDeleted(a) then                   ▷ r is valid, but a is not
12:         $k_p \leftarrow k_p$ + PrefixSumToIndex($i_p, p$)
            .....................................................
13:         $\mathcal{R}'[k_p] \leftarrow r$                             ▷ keep orphaned r
14:     else                                                    ▷ both r and a are valid
15:         $\mathcal{A}'[r] \leftarrow a$                               ▷ delete $\mathcal{A}[r]$
----------------------------------------------------------------------------------------------
16: $C \leftarrow \left\lceil \frac{j_{|P|}}{|P|} \right\rceil; t \leftarrow$ threadId()          ▷ phase 3: C = number of runs
17: $J \leftarrow$ PrefixSum($j_p$); $K \leftarrow$ PrefixSum($k_p$)
    .........................................................................
18: for $c \in \{0 \,..\, C - 1\}$ do    ▷ phase 3: process orphans, note: $|\mathcal{R}'| = |\mathcal{Z}'|$ on aggregate but not per block
19:     $y_p \leftarrow$ LoadBalance($J, t, s$)                          ▷ source address
20:     $x_p \leftarrow$ LoadBalance($K, t, s$)                          ▷ destination address
21:     $\mathcal{A}'[\mathcal{R}'[x_p]] \leftarrow \mathcal{Z}'[y_p]$                          ▷ delete orphans
22:     $t \leftarrow t + |P|$
```

fact, due to thread scheduling, the sequencing of output elements will likely differ between runs executed with identical input.

4.1 Redzone Algorithm for High-latency IPC

Algorithm 1 assumes fast **interprocess communication (IPC)**, such as shared memory, where all threads run on the same (multi-)processor. With a few adjustments, Redzone can run efficiently in high-latency IPC environments.

Let us define a *block* as a collection of threads able to communicate over a low-latency IPC channel, whereas inter-block communication happens over a high-latency IPC channel. To minimize the need for intra-block communication, phases 2 and 3 switch to batched processing. At the start of phase 3, the running totals of orphan counts per block are collated into two prefix sums, which are used to pair up orphans and complete the deletions. Phase 1 is unchanged.

Phase 1 is unchanged (1..4). Phase 2 switches to batched processing. Each block b_p owns its own slice of $\mathcal{R}$ and $\mathcal{Z}$ of length $s = \lceil \frac{|R|}{|P|} \rceil$. The final block $b_{|P|-1}$ contains trailing elements (0). Otherwise, phase 2 matches Algorithm 1 (5..15).

Due to the batched processing per block, phase 3 receives two orphan counts per block, j_p and k_p. Using these counts, each block redundantly generates prefix sums (J, K) (17), so all blocks share the same view but do not communicate. Given $|P|$ blocks, we need $C = \lceil \frac{j_{|P|}}{|P|} \rceil$ runs (16) to process all orphans. In each run c (18), load balancing calculates destination (x_p) and source indices (y_p) using prefix J, thread counter t, and batch size s (19). Because j and k do not line up, K is processed

ALGORITHM 3: Load balancing.

```
Input:  Prefix sum of start indices per block $\mathcal{S}$, thread counter $t$, block offset $b$
Output: Destination $d$
1: Start ← $t/b$
2: if Start > $|\mathcal{S}|$ then return out of bounds                ▷ all items have been processed
3: Index ← ReduceMin($\mathcal{S}$[Start], $\geq t$)                   ▷ get the smallest index $\geq t$
4: for ever do                                                         ▷ find the correct block index
5:     if ($t \geq \mathcal{S}$[Index]) then Index ← Index + 1
6:     else break                                                      ▷ if $k \gg |t|$, loop will likely iterate only once
7: SubIndex ← $t - \mathcal{S}$[Index]
8: return Index $\times$ $b$ + SubIndex
```

separately from J ⑳. The orphans are paired up and $\mathcal{A}[r]$ is deleted ㉑. Thread counter t tracks the batches ㉒. Note that loop ⑱ does not need any synchronization—even between threads. Global syncs between phases allow memory writes to settle.

A simple load balancer (see Algorithm 3) assigns threads to source and destination orphans. This is needed, because each batch has its own list of orphans and the counts *per batch* of destination and source orphans do not agree, although they do match on aggregate. Load balancing does not affect asymptotic runtime. If need be, then load balancing can be moved out of loop ⑱ so it is only performed once. Our CPU reference implementation of Redzone demonstrates this.

4.2 Time Complexity

Phase 1 performs k reads from $\mathcal{R}$ ③ and writes at best 0, on average $\frac{k^2}{n}$, and at worst k entries to $\mathcal{Z}$ ④, marking deleted items in the red zone. Phase 2 always reads $2k$ items from $\mathcal{R}$ and $\mathcal{Z}$ combined ⑥, and writes at best $x = 0$, on average $x = \frac{\min(k^3,(n-k)^3)}{n^2}$ and at worst $x = \frac{1}{2}k$ to both orphan lists ⑩, ⑬, as well as up to $k - x$ writes to overwrite deleted items in $\mathcal{A}$ ⑮. Finally, phase 3 reads no more than $\frac{1}{2}k$ orphans from each list to delete items in $\mathcal{A}$ ⑰. Redzone compaction performs $1 + 2 + 2(\frac{1}{2}) = 4k$ reads and $1 + 2(\frac{1}{2}) + 2(\frac{1}{2}) = 3k$ writes in the worst case and $1 + 2 + 0 = 3k$ reads and $0 + 1 + 0 = k$ writes in the best case. Other operations track reads and writes at $O(1)$ cost, resulting in $O(k)$ runtime. Writes to $\mathcal{A}/\mathcal{A}'$ are scattered, other reads and writes are sequential, greatly aiding efficiency.

When more than half of the list gets deleted ($\frac{k}{n} > \frac{1}{2}$), we have observed in our tests (see Figure 6) that scattered writes to mark deletions in phase 1 ④ take up the majority of the runtime. The GPUs used in our experiments (as well as many other modern GPUs and CPUs) can only write to a single cache line (e.g., a section of 256 contiguous bytes) per clock cycle. If scattered writes touch multiple cache lines, then writes are serialized, meaning that threads writing to different cache lines are paused until earlier writes retire.

However, our marking of deleted items in only $\mathcal{Z}$ is identical to deletion marking required in all of $\mathcal{A}$ by $O(n)$ compaction algorithms. These algorithms need to track deleted items in $\mathcal{A}$ itself or use a stencil array. A fair comparison should count this cost for all or none of the competing algorithms. If so, then the generation of removal list $\mathcal{R}$ must likewise be included in performance comparisons. However, this list is built up using contiguous writes and occupies a tiny fraction of the time spent performing scattered writes. Marking deleted items in $\mathcal{Z}$ (or even all of $\mathcal{A}$) may incur a smaller penalty if the preparation features sufficient non-write operations to hide latency due to scattered writes. Note that Redzone writes to $\mathcal{A}'$ in phases 2 and 3 are still scattered, whereas stable stream compaction need only perform contiguous writes in these phases. Depending on the implementation of scattered writes in hardware, this may bias performance for or against Redzone.

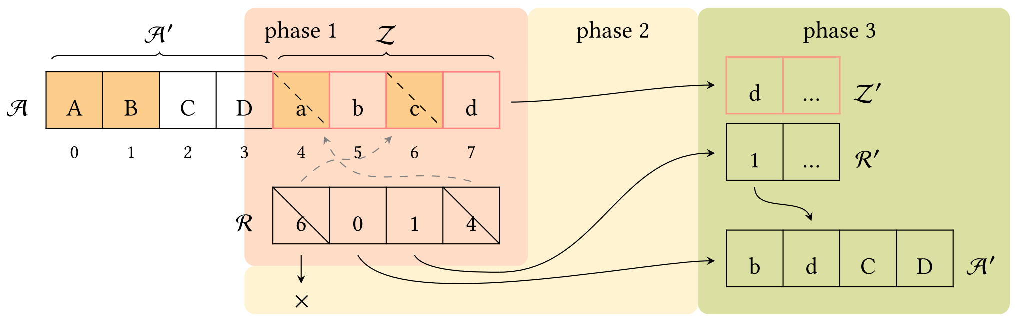

Fig. 3. Redzone compaction works in three phases. Phase 1 scans removals ($\mathcal{R}$) and marks deletions ($\cancel{6}$, $\cancel{4}$) in the red zone ($\mathcal{Z}$) as invalid sources. Phase 2 pairs every source element $a \in \mathcal{Z}$ with its destination sibling $r \in \mathcal{R}$. A destination that is invalid, because it points to the red zone ($\cancel{6}$) paired with a deleted source ($\cancel{a}$), is quietly discarded. Valid destinations (0) paired with unmarked sources (b) fill holes in $\mathcal{A}'$. An orphaned destination (1) paired to an invalid source ($\cancel{c}$) is collected in $\mathcal{R}'$; an orphaned source (d) paired to an invalid destination ($\cancel{4}$) is collected in $\mathcal{Z}'$. Phase 3 pairs source orphans $\mathcal{Z}'$ with destination orphans $\mathcal{R}'$ (d, 1) so each $\mathcal{A}'[r'] \leftarrow z'$.

4.3 Space Complexity

Redzone takes $\mathcal{A}$ and $\mathcal{R}$ as input. This requires $n \leftarrow |\mathcal{A}|, k \leftarrow |\mathcal{R}| = n + k$ space. $O(n)$ time algorithms take $\mathcal{A}$ and either use a predicate function, mark deletion items in place, or use a Boolean stencil. They need n space without or $2n$ space with a stencil. Redzone's intermediate lists ($\mathcal{Z}$, $\mathcal{Z}'$, and $\mathcal{R}'$) take $O(1)$ space. Input $\mathcal{Z}$ is an alias for the k tail items of $\mathcal{A}$, output $\mathcal{A}'$ is an alias for the front $n - k$ items of $\mathcal{A}$ (see Figure 3). Orphan lists $\mathcal{Z}'$ and $\mathcal{R}'$ are stored in-place in $\mathcal{Z}$ and $\mathcal{R}$, respectively, as Redzone processes both lists. Ergo, apart from its input, Redzone has $O(1)$ space complexity.

4.4 Synchronization

Thus far, we have glossed over synchronization needs. Algorithm 1 requires local synchronization (denoted by $\cdots$) between phases $\textcircled{5, 16}$, *local* meaning inside the same (multi-)processor. Algorithm 2 needs global synchronization (—), because writes must settle before entering a new phase. Inside phase 2, both algorithms need local synchronization ($\cdots$) to calculate the orphan indices j and k. This synchronization *cannot* be skipped, even if prefix sums $\textcircled{9, 12}$ are replaced by atomic counters! The reason for this is subtle. The reads of a and r in line $\textcircled{6}$ of both algorithms free up a slot in $\mathcal{Z}$ and $\mathcal{R}$. However, without synchronization before writes $\textcircled{10, 13}$ to the orphan lists $\mathcal{Z}'$ and $\mathcal{R}'$, a thread with index $i = x$ may not yet have read its data before another thread with index $j = x$ (or $k = x$) writes an orphan to that same entry. This race condition is prevented by waiting after reads $\textcircled{6}$ before writes $\textcircled{10, 13}$. In Algorithm 1, the sync of the prefix sums $\textcircled{9, 12}$ performs this function implicitly.

If these sync points are removed, then a wait after $\textcircled{6}$ must be added. This can be a local sync if each block reads and writes only from and to its own section—as in Algorithm 1. Alternatively, no synchronization is needed in phase 2 if $\frac{1}{2}k$ auxiliary storage is used for each of the two orphan lists, turning Redzone into an $O(k)$ space algorithm.

4.5 Proof $|\mathcal{Z}'| = |\mathcal{R}'|$

For Redzone to function correctly, both source and destination orphan lists must be the same size at the end of phase 2: $|\mathcal{Z}'| = |\mathcal{R}'| = k = j$ $\textcircled{16}$. Lines $\textcircled{10, 13}$ ensure that these source and destination lists contain only valid items. This allows the final loop $\textcircled{16..17}$ to process orphans correctly. The proof for this requirement follows:

In phase 1, $\mathcal{R}$ is traversed and if destination $r \in \mathcal{Z}$ (meaning $r \geq z$) points into the red zone, then source $\mathcal{Z}[r-z]$ will be deleted and is thus not valid. We mark these sources as deleted: $\mathcal{Z}[r-z] \leftarrow$ isDeleted ④. At the start of phase 2, we thus have subsets of $\mathcal{R}, \mathcal{Z}$ with deleted items: $\mathcal{R}_d \leftarrow \{d_r \in \mathcal{R} \mid d_r \geq z\}$ ⑬ and $\mathcal{Z}_d \leftarrow \{d_z \in \{\mathcal{Z} \cap \mathcal{R}\}\}$ ⑩.

Line ④ marks one element of $\mathcal{Z}$ as invalid for every $r_d \in \mathcal{R}_d$ that points into the red zone. Thus, $|\mathcal{R}_d| = |\mathcal{Z}_d|$ and because $|\mathcal{Z}| = |\mathcal{R}|$ ②, it follows that $|\mathcal{Z}_v| = |\mathcal{R}_v|$ for the valid subsets of both lists $\mathcal{Z}_v \leftarrow \mathcal{Z} - \mathcal{Z}_d$ and $\mathcal{R}_v \leftarrow \mathcal{R} - \mathcal{R}_d$.

In other words, every invalid destination in $\mathcal{R}_d$ has exactly one invalid source twin in $\mathcal{Z}_d$ and, conversely, every valid source in $\mathcal{Z}_v$ has exactly one valid destination twin in $\mathcal{R}_v$. We thus obtain the following result:

LEMMA 4.1. *$|\mathcal{R}_d| = |\mathcal{Z}_d|$ and $|\mathcal{R}_v| = |\mathcal{Z}_v|$.*

Note that $r \leftarrow \mathcal{R}[i]$ where i increases monotonically from 0 to $k-1$ ③, but r is an arbitrary index ④. This means that the red zone indices in $\mathcal{R}$ do not typically align with deleted items in $\mathcal{Z}$. Due to this misalignment, when both lists are traversed in phase 2 ⑥, the source/destination pairs $(\mathcal{Z}[i], \mathcal{R}[i])$ can be in one of four states as shown in Table 1.

Given two Boolean variables, four states (A to D) are possible as listed in Table 1. Lemma 4.1 implies that for every case C that pairs a valid destination with an invalid source (r_v, z_d) there exists an opposite pairing D that matches (r_d, z_v). This can be proven as follows:

A draws equally from $\mathcal{R}_v$ and $\mathcal{Z}_v$; likewise, B draws equally from $\mathcal{R}_d$ and $\mathcal{Z}_d$, neither violates Lemma 4.1.

C draws from $\mathcal{R}_v$ and $\mathcal{Z}_d$, violating both parts of Lemma 4.1, a single case C causes $|\mathcal{Z}_v| = |\mathcal{R}_v| + 1$ and $|\mathcal{R}_d| = |\mathcal{Z}_d| + 1$.

D symmetrically gives: $|\mathcal{Z}_v| = |\mathcal{R}_v| - 1$ and $|\mathcal{R}_d| = |\mathcal{Z}_d| - 1$.

If Lemma 4.1 is to be maintained, then imbalances due to over-application of C can only be restored if an equal number of cases D apply, and vice versa, ergo |D| = |C|: the number of destination orphans in $\mathcal{R}'$ due to case C equal source orphans in $\mathcal{Z}'$ due to case D. This leads to our final result, concluding the proof.

LEMMA 4.2. $|\mathcal{Z}'| = |\mathcal{R}'|$ ⑩,⑬

4.6 Adding as Well as Removing Items

Users may wish to concurrently add and remove items from list $\mathcal{A}$. For example, when modelling a todo list where tasks are added and resolved, but—because Redzone is unstable—the order of todo items does not matter. In this case, addition can be efficiently handled as follows: Add all additions (gains) $\mathcal{G}$ to the end of $\mathcal{A}$, wait for writes to resolve and perform Redzone on the newly expanded list $\mathcal{A} + \mathcal{G}$. If $|\mathcal{G}|$ is small, then it can be buffered and fed to Redzone separately. This order optimizes memory access patterns and reduces the number of orphans—compared to first deleting and then adding—although it does not change the $O(j+k)$ asymptotic time to add j and remove k items.

5 Experimental Results

To evaluate the performance of Redzone, we benchmark GPU implementations of Algorithms 1 and 2 as well as a CPU version of Algorithm 2. Our GPU implementations of Redzone 1 and 2 are coded in pure CUDA C++ without any framework. The CPU version of Redzone 2 uses Intel's TBB [10] library.

Our GPU code is compared against InK compaction [9] and NVidia's `Thrust::remove` [2] function on an NVidia RTX 3070 GPU with 8 GiB of RAM running at 1.815 GHz, as well as on an NVidia

A100 running at 1.41 Ghz in the MIG 4g-20gb configuration, which offers $4 \times 14 = 56$ multiprocessors, 20 MiB of L2 cache, and 20 GiB RAM.

On the CPU, Redzone is compared against Stencil compaction [3] and `std::remove` from C++20 using dual Intel Xeon Gold 6330 CPUs running at 2 GHz with $2 \times 28 = 56$ cores. To aid comparison, all CPU algorithms use Intel's TBB [10] library for thread scheduling. CPU code was compiled with gcc 11.2 using `-Ofast -march=native` settings, GPU code with CUDA 12.3 using `-O3` flags, with compute/sm flags set to 80 for the A100 and 86 for the RTX 3070.

To test a range of inputs, we vary array size from 128 KiB to 2 GiB and fill arrays with random data of which 2% to 90% is marked for removal. This random data is generated using the MT19937 random generator built into C++20. Duplicates are prevented by drawing from list $\mathcal{A}$ without replacement. The overall wall clock time tc is measured using the CPU's high-resolution timer. Clock cycles per phase on the GPU tg_i are measured using the GPU's `clock64` function; these are transformed to wall clock time using the CPU timer: $t_{\text{new}} = t_{\text{old}} \times \frac{tc}{\Sigma tg_i}$. A fixed random seed ensures all algorithms use the same data. Runs are repeated 20× with 20 different random seeds. Figures 4 to 7 show the mean times per phase as well as the maximum observed total time. For an apples-to-apples comparison, a phase 1 is added to all competing algorithms in our experiments. The vertically stacked bar graphs display cumulative logarithmic time to make it easy to mentally remove phase 1 time if needed. Horizontally stacked bar graphs show relative time per phase for $n = 2$ GiB to visualize what percentage of its total time each phase occupies.

5.1 Comparison of Algorithms on the GPU

Figure 4 compares Redzone algorithm 2 against `thrust::remove` and InK compaction. InK compaction is downloaded from Github[3] and written in plain CUDA C++, `thrust::remove` and its related functions use NVidia's `cub` library [16]. Runs are done on two Ampere GPUs: an A100 using 57,344 threads and an RTX 3070 GPU using 47,104 threads. The A100's memory bus is 20× wider than the RTX 3070, speeding up contiguous, but not scattered, reads and writes.

InK is consistently slower than Thrust, because the latter groups reads and writes in cache-friendly small batches, whereas InK reads all of $\mathcal{A}$ once, rereads it again, and only then writes. In phase 1, InK and Thrust perform k scattered writes; here, Redzone only performs $\frac{k^2}{n}$ writes, on average (see Section 4.2 for details, Section 5.2 for analysis). Later, Redzone phases reverse this advantage. On average, phase 2 does $\leq k - \frac{k^2}{n}$ and phase 3 $\leq \frac{k^2}{n}$ scattered writes; all writes in phase 2 of InK and Thrust are contiguous. Note that all algorithms do at least k scattered writes. As can be seen in Figure 4, such writes dominate runtime when $k \geq 50\%$. In these cases, InK and Thrust spend the majority of time in phase 1 running the simple for loop: `parallel for r in R: A[r] = deleted`. Redzone is faster for $k \leq 50\%$, because, on average, it performs $3\frac{1}{2}k$ contiguous reads, whereas Thrust and InK perform $2n$ such reads (contiguous writes are fire-and-forget and take negligible time). Redzone is only slightly slower than Thrust at $k = 90\%$, because scattered writes dominate runtime and both algorithms do k such writes (as explained in Section 5.2).

However, if an application is compute-rather than memory-bound, it may speed up stream compaction by intermixing phase 1 marking with compute operations to hide scattered write latency. If, say, all of the runtime of phase 1 can thus be hidden, then the phase 1 time segments of Figures 4 to 7 would disappear, and Redzone would underperform for $k \geq 10\%$.

For $n < 8$ MiB, constant factors dominate runtime; for this reason, we repeat the experiment on a single GPU block.

[3]Downloaded from https://github.com/knotman90/cuStreamComp

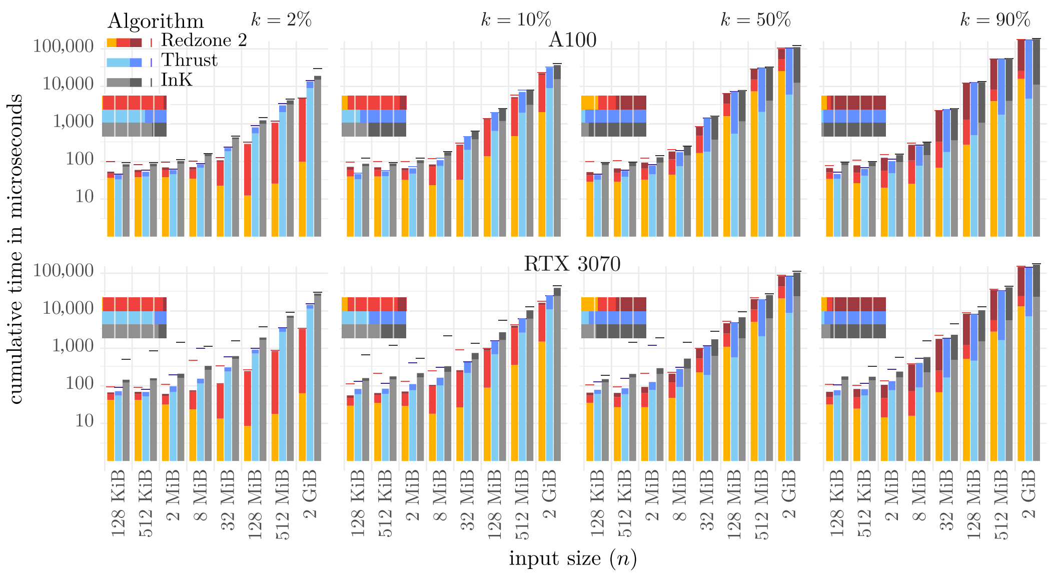

Fig. 4. Time in microseconds for three compaction algorithms run on two GPUs. Input size (n) varies from 128 KiB to 2 GiB, removals (k) range from 2%, 10%, 50%, to 90%. Runs on the A100 (top) use 56 blocks of 1,024 (57,344) threads, those on the RTX 3070 (bottom) use 46 × 1,024 (47,104) threads. Logarithmic mean times per phase are shown using stacked bars with Phase 3 🟧 (Redzone only) at the bottom, followed by phase 2 🟥🟦⬜, and phase 1 on top 🟥🟦⬛. Lines –−− atop each stack shows the maximum observed runtime. Horizontal bars ▨ show relative time per phase at n = 2 GiB, with a divider at 20% intervals.

Figure 5 compares Redzone algorithm 1 with $O(n)$ compaction on a single GPU core running 1,024 threads. This setup puts less strain on the GPU's memory subsystem, resulting in relatively faster scattered writes; phase 1 takes up a smaller percentage of the time ~60%–50% (at $k = 90\%$) rather than ~90% of the time in Figure 4. Due to reduced overhead, inputs < 8 MiB are processed faster. The horizontal bars displaying relative time per phase at n = 2 GiB show that relative time for Redzone's phase 3 is maximized at $k = 50\%$, as is explained in Section 5.2.

Unfortunately, cub and thus Thrust only support compaction GPU-wide, not on a single block. InK *can* be run on a single block, but then it superfluously reads $\mathcal{A}$ to calculate a prefix sum to schedule multiple blocks. Removing this extra work yields the Single pass algorithm. Reading $\mathcal{A}$ only once results in a ~1.8× speedup of InK's phase 2. Because this change to Thrust yields the same code, Redzone 1 is only compared against Single pass. This optimization can also be applied to InK for multiple GPU blocks. InK splits work into batches [9]. Instead of counting removals per batch in phase 2, we can do this while marking removals in phase 1 using atomic counters. This adds negligible time to phase 1, but nearly halves InK's phase 2 time. Figure 4 uses unaltered InK, but our Github code includes this optimization.

5.2 Breakdown of Runtime per Phase

The relative time Redzone spends per phase depends on the number of elements to remove (k), the percentage of elements deleted ($\frac{k}{n}$), the percentage of orphans in the red zone $r = \frac{|\mathcal{Z}'|}{k}$, and to a small extent the version (1 or 2) of the algorithm used. Figure 6 gives an indication of the runtime per phase for different values of k and r. It shows the average time spent per phase as a percentage of overall time for Redzone algorithm 2.

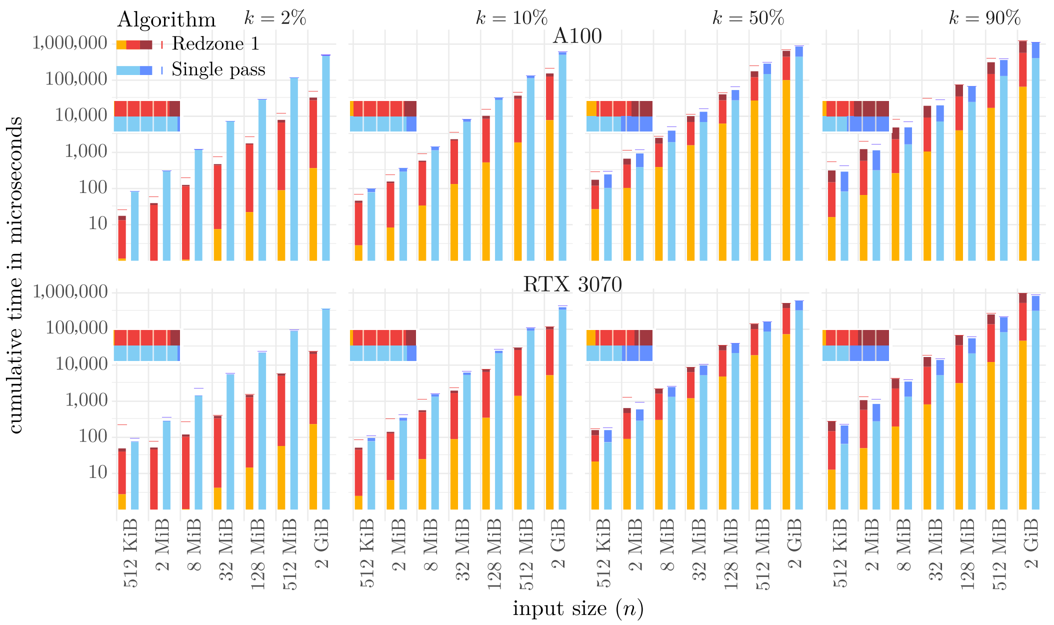

Fig. 5. Time in microseconds for Redzone 1 vs. Single pass on a single block of 1,024 threads. The top half shows times on the A100 GPU, the bottom half the same on the RTX 3070. Input size (n) varies from 512 KiB to 2 GiB, removals (k) range from 2%, 10%, 50%, to 90% of n. Logarithmic mean times per phases are shown using stacked vertical bars 🟥🟦, lines −− show the max time. Phase 3 (Redzone only) 🟧 is below, followed by phase 2 🟥 🟦, and phase 1 🟥 🟦 on top. Horizontal bars 🟧🟥🟦 show relative time per phase at n = 2 GiB.

Figure 6 displays the observed workload per phase for non-uniformly distributed removals. In this experiment, we run Redzone algorithm 2 using 57,344 threads on the A100 GPU over three parameters. The size parameter n ranges from 512 KiB to 2 GiB and the removals parameter $\frac{k}{n}$ includes $k = 2\%, 10\%$, and 50%. The last parameter is r: the percentage of items to be removed from the red zone itself. It increments in steps of 25% from $r = 0\%$ to $r = 100\%$.

Some modest spikes occur in relative runtimes, e.g., at the intersection of $k = 10\%$, $r = 50\%$, and $n = 32$ MiB. This is due to phase data no longer fitting into the 20 MiB cache. Cache eviction in phase 2 penalizes phase 3. For large n, the cache is exceeded throughout, restoring balance.

Interesting things happen for different values of r. At $r = 0\%$, no red zone items need marking in phase 1, and hence no orphans will be created. Phase 2 scatters k red zone items into $\mathcal{A}'$. Note that phase 1 will still *read* all of $\mathcal{R}$ and phase 3 will synchronize, incurring a small fixed cost. At the other extreme, $r = 100\%$, all items in the red zone will be marked as orphans. Due to line ⑦ (case B in the proof), phase 2 will quietly discard all these orphans and no writes take place here. This sounds very efficient, but alas, because Redzone cannot detect this case, phase 1 still does k scattered writes to mark red zone items, phase 2 still reads all of $\mathcal{R}$ and $\mathcal{Z}$, and phase 3 runs bookkeeping and syncs. For $k = 50\%$, overall performance varies little between these extremes. At $k = 10\%$ and below, the effect is more noticeable: $r = 0\%$ takes the most overall time, as r increases, the total time decreases. The opposite holds for the time taken in phase 1; phase 1 time is minimal at $r = 0\%$ and increases with higher percentages of r.

An application can potentially hide—some of—phase 1 time (as discussed in Section 5.1). Later phases depend on phase 1 and thus cannot easily be mixed with compute work. For this reason, only phase 1 and total time are included in the left panel of Figure 6. This shows that the potential for phase 1 savings grows as r and k increase.

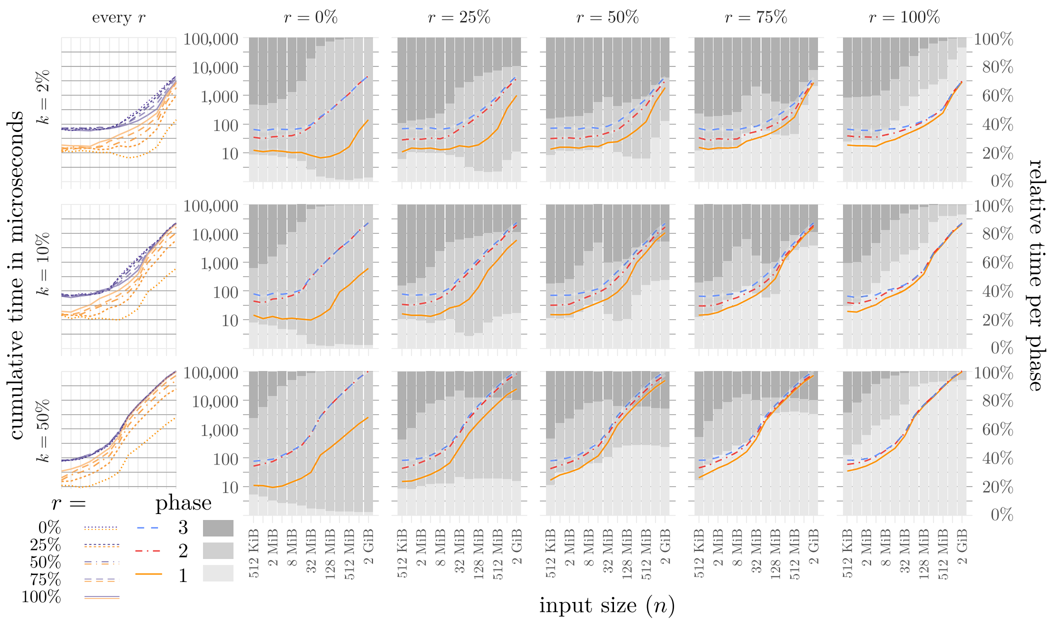

Fig. 6. Comparison of relative time per phase due to k: the percentage of $\mathcal{A}$ that is removed and r: the percentage of red zone items to be removed. This is run on an A100 GPU for Algorithm 2 using 57,344 threads. In the large right panel, the absolute times per phase in microseconds are displayed as colored lines. Shaded gray regions show the percentage of time occupied by each phase. The small left panel shows the total time with different line types as well as the phase 1 time for the various values of r, phase 2 is omitted. Not shown is $k = 90\%$, because $r > 88\%$ for that value of k, meaning that $r = 75\%$ and below cannot occur.

5.3 Comparison of Algorithms on the CPU

Figure 7 compares Redzone 2 against Algorithm 2 (Stencil) from Bernabé et al.'s CPU compaction paper [3] as well as `std::remove` from the standard C++ algorithms library on a dual Intel Xeon 6330 CPU with $2 \times 28 = 56$ threads. To aid comparison, all three algorithms use Intel's TBB [10] library. Stencil compacts $\mathcal{A}$ in three phases. Phase 1 marks off deleted elements, not in $\mathcal{A}$, but in an auxiliary Boolean stencil. Phase 2 reads the stencil and generates a prefix sum with counts per thread for scheduling. In phase 3, Stencil uses this data to compact the list. Unlike Redzone, InK, and `std/thrust::remove`, Stencil is an out-of-place algorithm. It also becomes more efficient as $\frac{k}{n}$ increases, which is useful for sparse lists. Having multiple threads write to a 1-bit Boolean stencil is known to perform poorly due to false sharing, however, implementing the stencil using 8- or 32-bit integers did not meaningfully affect Stencil's performance. As per our remark on InK at the end of Section 5.1, Stencil's phase 2 can be fully eliminated, in which case, we expect it to outperform `std::remove`.

6 Conclusion

Redzone stream compaction is the first parallel $O(k)$ algorithm for stream compaction. It outperforms existing $O(n)$ algorithms if a number of factors align: Unstable but in-place stream compaction is required (Redzone cannot perform out-of-place compaction without performing $O(n - k)$ extra work); a list of items to be deleted, $\mathcal{R}$, can be generated ($\mathcal{R}$ must not have duplicates) and, on average, fewer than half of the items are deleted: $\frac{k}{n} < 50\%$.

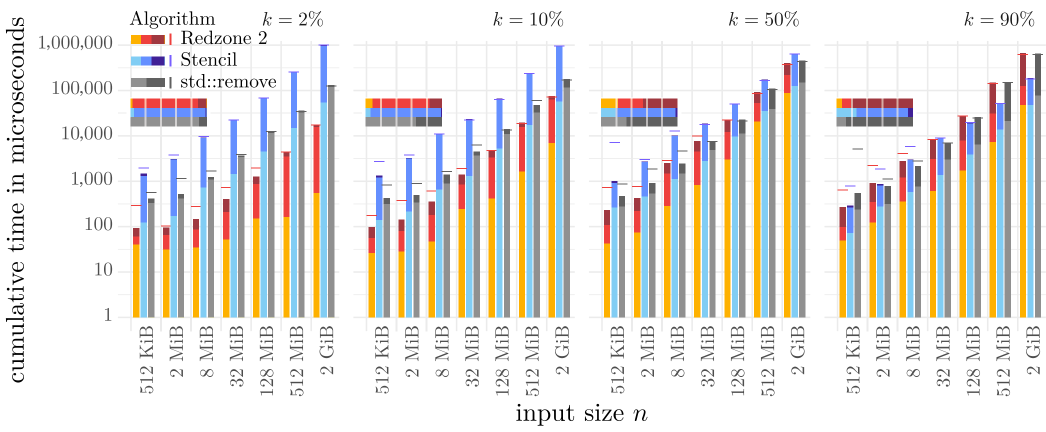

Fig. 7. Time in microseconds for three CPU-based algorithms. Input size varies from 512 KiB to 2 GiB, removals (k) range from 2%, 10%, 50%, to 90% of n. Runs are performed on a dual Intel Xeon 6330 CPU. Logarithmic means for the times per phase are shown using stacked bars with Phase 3 🟧🟦 (Redzone and Stencil) at the bottom, followed by phase 2 🟥🟦⬜, and phase 1 on top 🟥🟪⬛. A line –– atop each stack shows the maximum observed runtime. Horizontal bars ▨ show relative time per phase at n = 2 GiB.

In such circumstances, we have empirically determined that Redzone performs on par with competing $O(n)$ algorithms on the GPU and CPU when $\frac{k}{n}$ = 50% and outperforms them by roughly an order of magnitude when $\frac{k}{n}$ = 2%.

All code used in this article is available under an MIT license at: https://github.com/JBontes/redzone.html

Possible optimizations. Redzone has optimal time complexity $O(k)$, but constant factor improvements are possible.

- Write latency in phase 1 can be hidden by combining it with compute work in earlier stages of the application. If, at this time, deletion count is unknown, then this does not imply all k removals must be marked. The running count of keep items $(n - k)$ marks the start of the red zone. Only items beyond this bound need to be marked.
- If k is small, then processing can be sped up using fast memory (such as shared memory on a GPU). Rather than storing orphans in-place in $\mathcal{R}'$ and $\mathcal{Z}'$, these can be stored using $2(\frac{1}{2})k$ fast memory. This reduces latency in lines (4, 10, 13, 17), performs only a single read from $\mathcal{R}$ rather than two (3, 6), and allows reads from $\mathcal{Z}$ and $\mathcal{R}$ to be annotated as "do not store in cache."
- Data can be prefetched from main into fast memory before it is needed. This is especially advantageous if the hardware features asynchronous memory transfer [26]. If care is taken to have each thread prefetch its own elements, i.e., only those the same thread will use later on, then no synchronization is required, but a per-thread barrier suffices. Experiments show up to a 30% reduction in runtime in our GPU tests.
- Rather than storing all orphans (10, 13) early, this can be delayed until counts for both are known at (13). We can then pair up $m = \min(j, k)$ orphans, delete the items associated with these m paired orphans, and store any unpaired leftover orphans in either $\mathcal{R}'$ or $\mathcal{Z}'$. This reduces the number of orphans written in phase 2 and re-read in phase 3 (17), and processes most of them early in phase 2 instead, reducing cache contention. In our system, phase 3—without load balancing—occupies between 1% and 14% of total runtime (see Figure 6). Experiments show that about half this time can be saved by processing orphans early.

If $\frac{k}{n} > \frac{1}{2}$, then Redzone is a poor choice, due to extra I/O needed to achieve $O(k)$ runtime. Redzone performs $3.5k$ reads on average, meaning that an $O(n)$ algorithm reading $2n$ elements will outperform it in such cases.

Future work. For sparse compaction, the algorithmic inverse to Redzone may generate a new list $\mathcal{A}'$ given a *preserve* list $\mathcal{P}$ containing p items to preserve. This would effectively remove all non-preserve items from list $\mathcal{A}$, taking only $O(p)$ time, rather than $O(n - p)$ as Redzone does. An unstable out-of-place algorithm, where the output $\mathcal{A}'$ does not overlap with input list $\mathcal{A}$, is trivial:

```
for i in {0..p - 1}: A'[i] = A[P[i]].                    (3)
```

However, no *in-place* algorithm is currently known. Ideally, it would need only $O(1)$ space during operation and run in $O(p)$ time for an unstable algorithm. A stable $O(p \log p)$ in-place algorithm can be obtained by sorting list $\mathcal{P}$ first.

Acknowledgments

Computations were performed using facilities provided by the University of Cape Town's ICTS High Performance Computing team: https://ucthpc.uct.ac.za. Prof. Bernabé kindly made the source code used in his paper [3] available. Finally, the authors wish to thank the reviewers for their constructive comments.

References

[1] Darius Bakunas-Milanowski, Vernon Rego, Janche Sang, and Chansu Yu. 2017. Efficient algorithms for stream compaction on GPUs. *Int. J. Netw. Comput.* 7, 2 (2017), 208–226.

[2] Nathan Bell and Jared Hoberock. 2012. Thrust: A productivity-oriented library for CUDA. In *GPU Computing Gems Jade Edition*. Elsevier, 359–371. DOI: https://doi.org/10.1016/B978-0-12-385963-1.00026-5

[3] Gregorio Bernabé, Manuel E. Acacio, and Harald Köstler. 2018. On the parallelization of stream compaction on a low-cost SDC cluster. *Sci. Program.* 2018 (Jan. 2018), 1–10. DOI: https://doi.org/10.1155/2018/2037272

[4] Markus Billeter, Ola Olsson, and Ulf Assarsson. 2009. Efficient stream compaction on wide SIMD many-core architectures. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on High Performance Graphics*, Stephen N. Spencer, David K. McAllister, Matt Pharr, Ingo Wald, David P. Luebke, and Philipp Slusallek (Eds.). Eurographics Association, 159–166. DOI: https://doi.org/10.1145/1572769.1572795

[5] Guy E. Blelloch. 1990. *Prefix Sums and Their Applications.* Technical Report. School of Computer Science, Carnegie Mellon University Pittsburgh, PA, USA.

[6] Alexander Greß, Michael Guthe, and Reinhard Klein. 2006. GPU-based collision detection for deformable parameterized surfaces. *Comput. Graph. Forum* 25, 3 (2006), 497–506. DOI: https://doi.org/10.1111/j.1467-8659.2006.00969.x

[7] Mark Harris, Shubhabrata Sengupta, and John D. Owens. 2007. Parallel prefix sum (scan) with CUDA. *GPU Gems* 3, 39 (2007), 851–876.

[8] Jared Hoberock, Victor Lu, Yuntao Jia, and John C. Hart. 2009. Stream compaction for deferred shading. In *Proceedings of the Conference on High Performance Graphics (HPG'09).* Association for Computing Machinery, New York, NY, USA, 173–180. DOI: https://doi.org/10.1145/1572769.1572797

[9] David Meirion Hughes, Ik Soo Lim, Mark W. Jones, Aaron Knoll, and Ben Spencer. 2013. InK-compact: In-kernel stream compaction and its application to multi-kernel data visualization on general-purpose GPUs. *Comput. Graph. Forum* 32, 6 (2013), 178–188. DOI: https://doi.org/10.1111/cgf.12083

[10] Intel Corporation. 2024. oneAPI Threading Building Blocks. Retrieved from: https://github.com/oneapi-src/oneTBB

[11] Peter M. Kogge and Harold S. Stone. 1973. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Trans. Comput.* 22, 8 (1973), 786–793. DOI: https://doi.org/10.1109/TC.1973.5009159

[12] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. 2010. Debunking the 100X GPU vs. CPU Myth: An evaluation of throughput computing on CPU and GPU. *SIGARCH Comput. Archit. News* 38, 3 (June 2010), 451–460. DOI: https://doi.org/10.1145/1816038.1816021

[13] Pedro Miguel Moreira, Luís Paulo Reis, and A. Augusto de Sousa. 2009. JUMPING JACK-A parallel algorithm for non-monotonic stream compaction. In *Proceedings of the International Conference on Computer Graphics Theory and Applications*, Vol. 1. SCITEPRESS, 137–146. DOI: https://doi.org/10.5220/0001785001370146

[14] Goran S. Nikolić, Bojan R. Dimitrijević, Tatjana R. Nikolić, and Mile K. Stojcev. 2022. A survey of three types of processing units: CPU, GPU and TPU. In *Proceedings of the 57th International Scientific Conference on Information, Communication and Energy Systems and Technologies (ICEST'22)*. IEEE, 1–6.

[15] Corey J. Nolet, Divye Gala, Alexandre Fender, Mahesh Doijade, Joe Eaton, Edward Raff, John Zedlewski, Brad Rees, and Tim Oates. 2023. cuSLINK: Single-linkage agglomerative clustering on the GPU. In *Machine Learning and Knowledge Discovery in Databases: Research Track - European Conference, ECML PKDD 2023, Turin, Italy, September 18–22, 2023, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 14169)*, Danai Koutra, Claudia Plant, Manuel Gomez-Rodriguez, Elena Baralis, and Francesco Bonchi (Eds.). Springer, 711–726. DOI: https://doi.org/10.1007/978-3-031-43412-9_42

[16] NVIDIA Corporation. 2024. CUB - Cuda UnBound. Retrieved from: https://nvidia.github.io/cccl/cub/

[17] Paul Richmond, Simon Coakley, and Daniela M. Romano. 2009. A high performance agent based modelling framework on graphics card hardware with CUDA. In *Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 2 (AAMAS'09)*. International Foundation for Autonomous Agents and Multiagent Systems, 1125–1126. DOI: https://doi.org/10.5555/1558109.1558172

[18] David Roger, Ulf Assarsson, and Nicolas Holzschuch. 2007. Efficient stream reduction on the GPU. In *Proceedings of the Workshop on General Purpose Processing on Graphics Processing Units*.

[19] Mohsen Safari and Marieke Huisman. 2022. Formal verification of parallel prefix sum and stream compaction algorithms in CUDA. *Theoret. Comput. Sci.* 912 (2022), 81–98. DOI: https://doi.org/10.1016/j.tcs.2022.02.027

[20] Albert Segura, Jose-Maria Arnau, and Antonio González. 2019. SCU: A GPU stream compaction unit for graph processing. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA'19)*. Association for Computing Machinery, New York, NY, USA, 424–435. DOI: https://doi.org/10.1145/3307650.3322254

[21] Shubhabrata Sengupta, Aaron Lefohn, and John D. Owens. 2006. A work-efficient step-efficient prefix-sum algorithm. In *Proceedings of the Workshop on Edge Computing Using New Commodity Architectures*.

[22] Harold S. Stone. 1975. Parallel tridiagonal equation solvers. *ACM Trans. Math. Softw.* 1, 4 (1975), 289–307. DOI: https://doi.org/10.1145/355656.355657

[23] Qiao Sun, Chao Yang, Changmao Wu, Leisheng Li, and Fangfang Liu. 2016. Fast parallel stream compaction for IA-based multi/many-core processors. In *Proceedings of the IEEE/ACM 16th International Symposium on Cluster, Cloud and Grid Computing (CCGrid'16)*. IEEE, 736–745. DOI: https://doi.org/10.1109/CCGrid.2016.112

[24] Ingo Wald, Christiaan P. Gribble, Solomon Boulos, and Andrew Kensler. 2007. *SIMD Ray Stream Tracing - SIMD Ray Traversal with Generalized Ray Packets and On-the-fly Re-Ordering*. Technical Report. Informe Técnico, SCI Institute.

[25] Haicheng Wu, Gregory Diamos, Srihari Cadambi, and Sudhakar Yalamanchili. 2012. Kernel weaver: Automatically fusing database primitives for efficient GPU computation. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'12)*. IEEE Computer Society, 107–118. DOI: https://doi.org/10.1109/MICRO.2012.19

[26] Xinyao Yi, David Stokes, Yonghong Yan, and Chunhua Liao. 2021. CUDAMicroBench: Microbenchmarks to assist CUDA performance programming. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPS Workshops'21)*. IEEE, 397–406. DOI: https://doi.org/10.1109/IPDPSW52791.2021.00068

[27] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. 2008. Real-time KD-tree construction on graphics hardware. *ACM Trans. Graph.* 27, 5, Article 126 (Dec. 2008), 11 pages. DOI: https://doi.org/10.1145/1409060.1409079

Received 29 November 2023; revised 13 April 2024; accepted 18 June 2024