Massively Parallel GPU Memory Compaction

Matthias Springer
Tokyo Institute of Technology, Japan
matthias.springer@acm.org

Hidehiko Masuhara
Tokyo Institute of Technology, Japan
masuhara@acm.org

Abstract

Memory fragmentation is a widely studied problem of dynamic memory allocators. It is well known that fragmentation can lead to premature out-of-memory errors and poor cache performance.

With the recent emergence of dynamic memory allocators for SIMD accelerators, memory fragmentation is becoming an increasingly important problem on such architectures. Nevertheless, it has received little attention so far. Memory-bound applications on SIMD architectures such as GPUs can experience an additional slowdown due to less efficient vector load/store instructions.

We propose CompactGpu, an incremental, fully-parallel, in-place memory defragmentation system for GPUs. CompactGpu is an extension to the DynaSOAr dynamic memory allocator and defragments the heap in a fully parallel fashion by merging partly occupied memory blocks. We developed several implementation techniques for memory defragmentation that are efficient on SIMD/GPU architectures, such as finding defragmentation block candidates and fast pointer rewriting based on bitmaps.

Benchmarks indicate that our implementation is very fast with typically higher performance gains than compaction overheads. It can also decrease the overall memory usage.

CCS Concepts **• Software and its engineering → Allocation / deallocation strategies**; **• Computer systems organization →** *Single instruction, multiple data.*

Keywords GPUs, dynamic allocation, fragmentation

ACM Reference Format:
Matthias Springer and Hidehiko Masuhara. 2019. Massively Parallel GPU Memory Compaction. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management (ISMM '19), June 23, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3315573.3329979

1 Introduction

Memory fragmentation is a challenging problem of dynamic memory allocators and has been widely studied on single-core and multi-core CPU systems (MIMD architectures). On such systems, dynamic memory allocators can achieve low memory fragmentation with good allocation policies [12] and compacting garbage collectors.

However, despite the recent popularity of massively parallel single-instruction multiple-data (SIMD) architectures, the memory fragmentation problem has not been studied thoroughly on such architectures.

This is because dynamic memory allocators for SIMD architectures such as GPUs have just been developed recently [7, 10, 22, 23] and not been around long enough yet. We believe that our research will enable more programmers to utilize dynamic memory allocation in their applications.

We need to study memory (de)fragmentation on massively parallel SIMD architectures because allocations follow different patterns on such architectures. Most allocations are small in size[1] and due to mostly regular control flow, many allocations have the same byte size. Such patterns are reflected in the design of state-of-the-art GPU allocators. For example, Halloc [2], one of the fastest GPU allocators can allocate only a few dozen predetermined byte sizes between 16 bytes and 3 KB. Such specialties must be exploited by memory defragmentation systems to achieve good performance.

1.1 SIMD-Specific Performance Characteristics

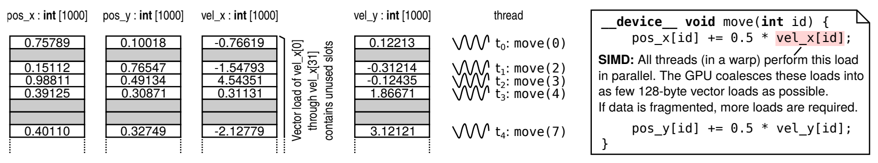
Before introducing the design of our system, we review important performance characteristics of SIMD architectures.

Effects of Fragmentation Fragmentation measures the degree of scattering of allocations across the heap and is caused by unfortunate allocate-deallocate patterns. High fragmentation leads to three main disadvantages.

- **Premature Out-of-Memory:** Large allocations cannot be accommodated even if there is enough free memory overall (*external fragmentation*).
- **Low Cache Hit Rate:** Poor data locality causes poor cache performance [8].
- **Low Vector Load/Store Efficiency:** SIMD vector load/store instructions are less efficient.

While the first two points are well-established and apply to most architectures, the specific effects on SIMD architectures have received little attention.

[1]If thousands of threads were to request large allocations, a GPU would run out of memory immediately.



```
__device__ void move(int id) {
    pos_x[id] += 0.5 * vel_x[id];
```

SIMD: All threads (in a warp) perform this load in parallel. The GPU coalesces these loads into as few 128-byte vector loads as possible. If data is fragmented, more loads are required.

```
    pos_y[id] += 0.5 * vel_y[id];
}
```

Figure 1. Example: N-body Simulation in SOA. Objects are stored in SOA arrays. Due to fragmentation, gray slots are unused. If all used cells were arranged in one consecutive chunk, data would not be fragmented. If data is fragmented, vector loads are less efficient because they include unused (gray) slots.

Effect of Fragmentation on Vectorized Access SIMD architectures achieve parallelism by executing instructions on a vector register. However, vector load/store operations are less efficient with higher fragmentation. When threads in a GPU application simultaneously access different memory addresses, the GPU *coalesces* accesses from the same SIMD work group (*warp* in CUDA, every 32 consec. threads) into one physical memory transaction if the addresses are on the same 128-byte cache line [11]. More fragmentation leads to more scattered memory addresses, resulting in poorer performance due to a higher number of memory transactions.

Memory fragmentation can greatly affect vectorized access, even if data is stored in a Structure of Arrays (SOA) data layout [24]. SOA is a best practice for more efficient vector access on SIMD architectures. In SOA, all values of a field are stored together. Previous work has demonstrated speedups of several factors when changing from a traditional *AOS* object layout to an SOA layout [9, 17].

Figure 1 shows a simplified data structure of an n-body simulation in SOA layout. Recent NVIDIA architectures coalesce simultaneous accesses of consecutive memory addresses into 128-byte vector transactions. Accessing fragmented data (`vel_x` values in the example) requires more vector transactions than accessing the same amount of dense data. This reduces the overall performance of memory-bound applications because memory bandwidth is limited [16].

1.2 Memory Defragmentation

In essence, every memory defragmentation system has to solve four basic problems.

1. Determine which parts of the heap are fragmented.
2. Based on that information, decide which objects[2] to move (relocate) and where to move them.
3. Physically relocate objects in memory.
4. Find and rewrite pointers to relocated objects. (Alternative: Ensure that objects can still be accessed through their old pointers.)

Most memory defragmentation systems are part of a garbage collector (GC). Since GCs have to scan large parts of the heap anyway, they can gather additional metainformation almost for free. This information can be used to select memory areas for compaction [13, 19] or to determine which parts of the heap contains pointers that must be rewritten [25].

[2]We use the term *object* instead of *allocation* throughout this paper because we are focusing on object-oriented systems in this work.

Background We present CompactGpu, an incremental, fully parallel, in-place memory defragmentation system for GPUs. GPUs/SIMD architectures are predominantly programmed in a C++ dialect (e.g., CUDA, OpenCL, ispc [20], Sierra [15]), so we extended an existing CUDA dynamic memory allocator. Memory management in C++ is manual, so we cannot rely on a GC to collect metainformation for us.

Our work builds on top of DynaSOAr [22], a C++/CUDA dynamic memory allocator for GPUs. DynaSOAr was designed for Single-Method Multiple-Object (SMMO) applications that express parallelism by running a method on all objects of a type. It stores objects in SOA, which allows for efficient vectorized access of allocated memory.

1.3 Contributions and Outline

CompactGpu is an efficient GPU memory defrag. system that is optimized for GPU-specific allocation patterns. It is fully parallel and in many cases the performance gain of defrag. is much larger than the defrag. overhead. This is due to careful design and engineering efforts: CompactGpu is based on parallel block merging, utilizes bitmaps to speed up pointer rewriting, exhibits mostly uniform control flow and requires no synchronization between GPU threads.

We evaluated CompactGpu with synthetic and real benchmarks. CompactGpu can improve application performance by up to 16% and reduce the overall memory consumption of an application, while incurring minimal runtime overheads.

This paper is organized as follows. Sec. 2 gives a brief overview of DynaSOAr. Sec. 3 describes the design and implementation of CompactGpu. Sec. 4 evaluates CompactGpu with synthetic and real benchmarks. Sec. 5 describes related work. Finally, Sec. 6 concludes the paper.

2 Heap Layout and Data Structures

A variety of dynamic memory allocators for GPUs have been developed in recent years. CompactGpu is implemented in DynaSOAr, but its basic ideas can be adapted to other dynamic memory allocators as long as they follow a few basic design requirements.

- The heap is divided into fixed-size **memory blocks**.
- A block contains only **objects of the same size**. This requirement is crucial. Same-size objects can be compacted much more easily than objects of different size.
- Blocks of the same object size have the **same capacity**.
- The allocator maintains **fill levels** for each block.

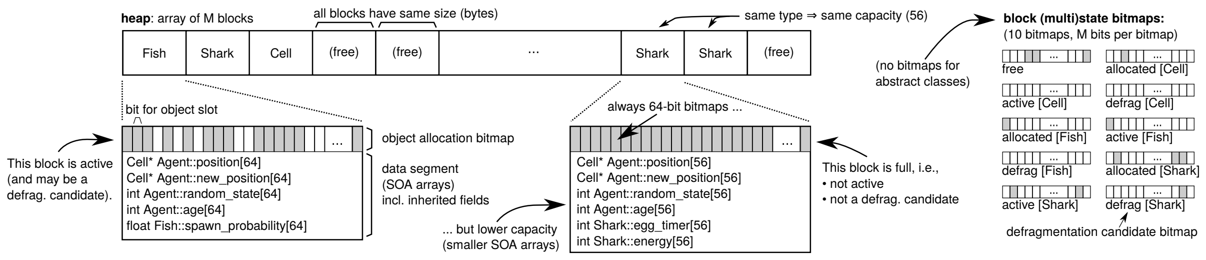

heap: array of M blocks
all blocks have same size (bytes)
same type ⇒ same capacity (56)
Fish
Shark
Cell
(free)
(free)
...
Shark
Shark
(free)
(no bitmaps for abstract classes)
block (multi)state bitmaps: (10 bitmaps, M bits per bitmap)
free
allocated [Cell]
active [Cell]
defrag [Cell]
allocated [Fish]
active [Fish]
defrag [Fish]
allocated [Shark]
active [Shark]
defrag [Shark]
defragmentation candidate bitmap
bit for object slot
always 64-bit bitmaps ...
object allocation bitmap
This block is active (and may be a defrag. candidate).
Cell* Agent::position[64]
Cell* Agent::new_position[64]
int Agent::random_state[64]
int Agent::age[64]
float Fish::spawn_probability[64]
data segment (SOA arrays) incl. inherited fields
... but lower capacity (smaller SOA arrays)
Cell* Agent::position[56]
Cell* Agent::new_position[56]
int Agent::random_state[56]
int Agent::age[56]
int Shark::egg_timer[56]
int Shark::energy[56]
This block is full, i.e.,
• not active
• not a defrag. candidate

Figure 2. Example: Heap layout for Wa-Tor. The heap consists of equally sized blocks. Up to 64 objects can be stored in a block, as indicated by the *object allocation bitmap*. A block can be in one or multiple of 10 possible states, as indicated by the state bits shown for every block. There are *allocated*, *active* and *defrag* states for the three classes `Fish`, `Shark` and `Cell`, but not for class `Agent` because it is an abstract class.

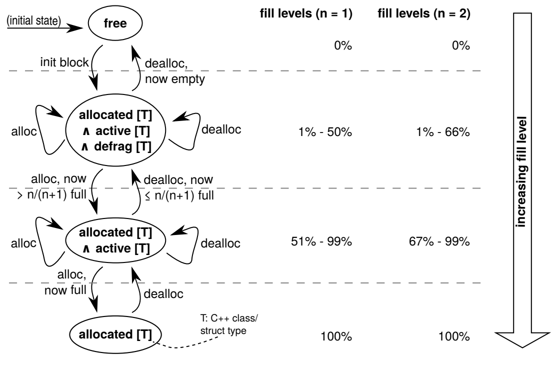


Figure 3. Block States. Initially, every block is *free*. New objects are allocated in *active* blocks of the corresponding type. We introduced a new state *defrag* to indicate defragmentation candidates. Only objects from such blocks are relocated during defragmentation.

DynaSOAr [22], Halloc [2] and UAlloc [7] are three examples of such allocators. We implemented CompactGpu in DynaSOAr because it is the only one with an SOA data layout, which allows for efficient vectorized memory access of allocated memory. While all allocators would benefit from better cache performance and more space-efficient memory usage, memory defragmentation in DynaSOAr additionally leads to more efficient vectorized memory accesses and thus better memory bandwidth utilization.

2.1 Running Example

We use a simple fish-and-sharks simulation ("Wa-Tor") as a running example to describe the data structures of CompactGpu. This application has four classes: `Cell`, `Agent`, `Fish` and `Shark`. The last two classes are subclasses of the abstract class `Agent`. Fish and sharks inhabit a 2D grid of cells in a predator-prey relationship.

This application exhibits a large number of allocations and deallocations, which lead to memory fragmentation.

2.2 Overview of the DynaSOAr Allocator

DynaSOAr is a slab allocator [4]. It divides the heap into M blocks of equal byte size, each of which can contain up to 64 objects (*capacity*) of the same C++ class/struct type, depending on the size of the type (Figure 2). A position where an object can be stored is called an *object slot*. A 64-bit *object allocation bitmap* keeps track of allocations. Objects are stored in the *data segment* in an SOA data layout: one *SOA array* per field.

A block can be in one or more *multistates*. There are $3 \times \#types + 1$ possible states: one global *free* state and three states for each type T in the system (Figure 3).

- **free:** The block is empty and does not contain any objects. No type is specified for this block.
- **allocated[T]:** The block may contain objects only of type T. No other objects can be stored in the block.
- **active[T]:** The block contains objects of type T. It is not full yet, i.e., it has space for at least one more object. *active[T] ⇒ allocated[T].*
- **defrag[T]:** The block is considered for defragmentation. We call such a block a *defragmentation candidate*. We introduced this state to support defragmentation in DynaSOAr and will describe its purpose in the next section. *defrag[T] ⇒ allocated[T] ∧ active[T].*

Allocation, deallocation and defragmentation routines frequently lookup blocks by state. For that reason, block states are indexed by bitmaps of size M; one bitmap per state.

Fragmentation Our definition of fragmentation F differs from other systems. We define it as the fraction of allocated but unused memory. If a block is in an *allocated* state, we consider all of its object slots as *allocated*. However, only object slots that actually contain an object, as indicated by the object allocation bitmap, are *used*. Fragmentation is defined as the average *free level* among all allocated blocks.

$$F = \frac{1}{\#\text{blocks}} \sum_{b \in Blocks} \frac{\#\text{free slots}(b)}{\#\text{slots}(b)}$$

Our goal is reduce F as much as possible. Zero fragmentation means that all blocks are 100% full and vectorized memory access is most efficient. Conversely, a vector load on a block that is 60% full will on average read 40% garbage. Moreover, unused memory in an allocated block is not available for objects of other types. This leads to less space-efficient memory usage.

The blocks themselves may be widely *scattered* in the heap. For example, in Figure 2, three allocated blocks are stored at the beginning of the heap and two are stored towards the end. This does not affect cache utilization or vector load/store efficiency, because with up to 64 objects per blocks, most SOA arrays are much larger than a cache line or the size of a vector load/store (128 bytes on NVIDIA GPUs). Neither can it lead to external fragmentation or premature out-of-memory errors, because all blocks have the same size.

Object Allocation To reduce fragmentation, even without active memory defragmentation, DynaSOAr allocates new objects of type T always in *active[T]* blocks. These are blocks that have space for at least one more object. Only if no active block could be found, DynaSOAr locates a free block and turns it into an *allocated[T]* and *active[T]* block (*slow path*).

DynaSOAr then reserves an object slot inside the block by atomically flipping a bit in the object allocation bitmap from 0 to 1. If this operation was successful, the block state may have to be updated in the block state bitmaps.

If the number of objects of a type drops, fragmentation can increase, because a block is deallocated only if all of its objects are deallocated. This kind of fragmentation can be eliminated with CompactGpu.

Programming Interface DynaSOAr provides an embedded C++ DSL for defining classes/fields. Through this DSL, CompactGpu can programmatically *reflect* on the classes/fields that are defined in an application, somewhat similar to the Java Reflection API or metaobject protocols [5]. This functionality is used in the pointer rewriting step to restrict heap scans to a smaller part of the heap (Sec. 3.4).

3 Defragmentation with CompactGpu

CompactGpu is a memory defragmentation system for GPUs, implemented as a DynaSOAr extension. CompactGpu is:

- **Configurable:** The desired target fragmentation rate can be tuned with parameters.
- **In-place:** No auxiliary storage is necessary and the entire heap remains usable.
- **Incremental:** A single defragmentation pass is very fast and compacts only a fraction of the heap. Compacting the entire heap requires multiple passes.
- **A stop-the-world approach:** A defragmentation pass can run only when no other GPU code is running[3].
- **Fully parallel:** Every step is implemented as a perfectly parallel CUDA kernel. No synchronization among threads is necessary for defragmentation.
- **Not order preserving:** After defragmentation, objects are likely arranged in a different order.

[3]This is because, in current GPU architectures, there is no efficient way of interrupting a kernel to run a defrag. pass, should the allocator run out of memory during the kernel. Many GPU programs are a sequence of GPU kernels [21], so there are plenty of opportunities to run a pass in-between.

Programmers initiate defragmentation manually and specify the type that should be defragmented. CompactGpu is based on three fundamental ideas.

Block Merging The heap is defragmented by moving objects from source blocks to target blocks.
Forwarding Pointers Pointers to the new object locations are placed in source blocks.
Bitmaps To speed up pointer rewriting, bitmaps are utilized to quickly filter out unaffected pointers.

We considered various alternative designs (Sec. 3.8), but the combination of forwarding pointers with bitmaps proved to be most performant on GPUs.

Block Merging We compact the heap by merging blocks. Blocks of the same type have the same capacity, so *a source block can be merged into a target block if both blocks have the same type and both blocks are no more than 50% full.*

A source block can be merged into two target blocks if none of the three blocks is more than 66% full. Or in general: A source block can be merged into n target blocks if none of the $n + 1$ blocks is more than $\frac{n}{n+1}$ full. In each case, the number of allocated blocks is reduced by one.

Defragmentation Factor We call n the *defragmentation factor*. This value is problem-specific and must be chosen by the programmer at compile time. Blocks that are no more than $\frac{n}{n+1}$ full are *defragmentation candidates*. Only those blocks are considered during defragmentation.

During defragmentation, all objects from a source block are moved to target blocks and the source block is deleted. Target blocks lose their defragmentation candidate state if they are now more than $\frac{n}{n+1}$ full. They also lose their active state if they are now entirely full.

Given a defragmentation factor of n, CompactGpu is guaranteed to bring down fragmentation to $1 - \frac{n}{n+1} = \frac{1}{n+1}$, if all defragmentation candidates are eliminated. This may require multiple defragmentation passes, as will be described later. For example, for $n = 2$, all blocks with less or equal to 66% fill level are gone after defragmentation[4]. Only blocks with a higher fill level are left over. Consequently, the fragmentation level is guaranteed to be less than $1 - 66\% = 33\%$.

Finding Defragmentation Candidates A defragmentation pass must be able to quickly find all defragmentation candidates in order to choose source and target blocks. CompactGpu extends object allocation and deallocation routines of DynaSOAr to keep track of defragmentation candidates.

CompactGpu maintains a bitmap *defrag[T]* (one per type) in which a bit is set if the corresponding block is a defragmentation candidate. This bitmap is updated based on block fill levels. An alternative implementation could scan the header of every block and generate the bitmap on demand.

[4]Since every source block must be matched with $n = 2$ target blocks, up to two defragmentation candidates may be left over.

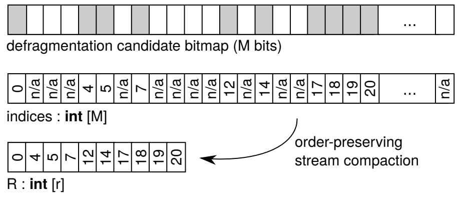

defragmentation candidate bitmap (M bits)
indices : int [M]
order-preserving stream compaction
R : int [r]

Figure 4. Example: Compacting a bitmap of defragmentation candidates. (There is such a bitmap for every type/allocation size.)

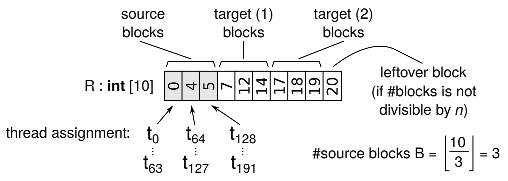

source blocks
target (1) blocks
target (2) blocks
leftover block (if #blocks is not divisible by n)
R : int [10]
thread assignment:
#source blocks B = ⌊10/3⌋ = 3

Figure 5. Example: Assigning source and target blocks ($n = 2$). E.g., objects from source block 5 are moved into blocks 14 and 19.

3.1 Defragmentation Pass

A defragmentation pass consists of four main steps. Every step is implemented as a CUDA kernel and runs in parallel.

1. Copy objects from source to target locations.
2. Store forwarding pointers.
3. Scan the heap and rewrite pointers to source locations.
4. Update block states.

3.2 Copying Objects

CompactGpu copies objects from source to target blocks. Those blocks must be defrag. candidates, to ensure that all objects of a source block fit into the corresponding target blocks.

Choosing Source/Target Blocks We utilize the defragmentation candidate bitmap to quickly find and assign target blocks to source blocks (Figure 4). We first generate an *indices* array of size M that contains i at position i if the i-th bit is set. Otherwise, we store an *invalid marker*. Now we filter/compact the array to retain only valid values, resulting in array R of size r. This *stream compaction* [3] is implemented with a parallel prefix sum operation (CUB library [18]).

Based on array R, each GPU thread can later by itself (without synchronization) efficiently determine its assigned source block and corresponding target blocks (Figure 5). Given a defragmentation factor n, the $B = \left\lfloor \frac{r}{n+1} \right\rfloor$ blocks with indices $R[0]$ through $R[B-1]$ are source blocks. Given a source block $R[s\_rid]$, its corresponding target blocks are:

$$\left\{ R[s\_rid + i \cdot B] \;\middle|\; i \in 1 \dots n \right\}$$

Copying Objects Objects are copied in parallel. We assign 64 consecutive threads to every source block (Figure 5) and every thread copies at most one object. Some threads will have no work to do, because not all (up to) 64 object slots in a source block are occupied.

Alg. 1 describes how objects are copied (Figure 6). There are 64 threads for every source block. A thread t_{tid} copies the (s_loc = *tid* % 64)th object of the source block (ID s_bid). Let s_oid be the slot ID of this object. The target slot is the s_locth free slot among all target blocks. Let t_oid and t_bid be the slot ID and block ID of that slot. To determine the target slot, we may have to examine the object allocation bitmap of multiple or all n target blocks (Line 7). This causes some *thread divergence* because the number of for-loop iterations differs among threads, but n is usually small. Furthermore, note that no synchronization is required among threads.

On GPUs, memory accesses have a much higher latency than arithmetic instructions [26]. Therefore, we have to keep the number of extra accesses in addition to the field copies (i.e., the overhead of CompactGpu) low. Since all threads in a warp copy from/to the same blocks, we require at most $1+n$ read transactions from the array R and the same number read transactions of object allocation bitmaps per warp[5].

Better Source/Target Choices? The number of object copies (and pointer rewritings) could be reduced by selecting less full defragmentation candidates as source blocks. Such an optimization does not pay off for three reasons.

First, selecting source blocks becomes much more difficult. How would a GPU thread know which block is less full? We would either have to sort the array R with a comparator function that counts the set bits in each block's object allocation bitmap (a random memory access!). Or we would have to maintain additional *defrag[T]* bitmaps for various fill levels. Both variants would greatly reduce performance.

Second, since memory is accessed in 128-byte vector transactions, reading/writing a slightly lower number of scalar values (that are likely scattered within an SOA array) is unlikely to reduce the number of memory transactions.

And third, there would be more threads without work, but since every warp must execute the same instructions (SIMD model), these threads have to nevertheless *wait* for the copying threads in the warp (*warp divergence*).

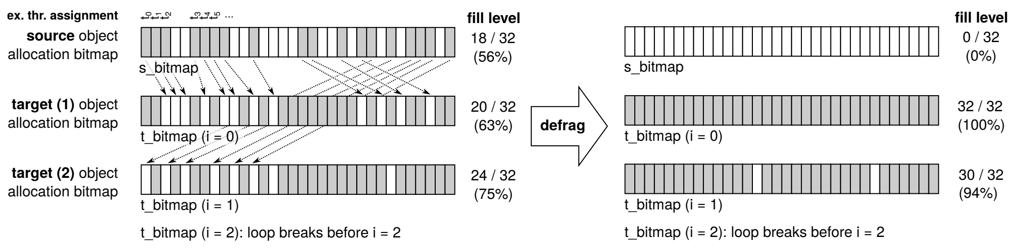
3.3 Storing Forwarding Pointers

After copying objects, pointers to the old memory location must be updated (*rewritten*). Many memory defragmentation systems do this with *forwarding pointers*: A pointer to the object's new memory location is stored at its old location.

We extended DynaSOAr to store forwarding pointers inside blocks. Every block may contain either a data segment or forwarding pointers. Listing 1 shows the data structure of a block of type `Fish` (also see lower left part of Figure 2).

Alg. 2 shows how the forwarding pointers array is populated. This algorithm is identical to Alg. 1, except for Line 20.

[5]If multiple threads of a warp access the same memory address, only one memory transaction is required (a special case of *memory coalescing*).



(a) before relocation **(b)** after relocation

Figure 6. Example: Relocating objects ($n = 3$). Assuming block size 32 instead of 64. All 18 objects fit into the first two selected target blocks, so no third block is needed. The first 12 objects fit into the first target block. The remaining 6 objects fit into the second one.

Algorithm 1: move_objects<T>() : void

```
1  for tid ← 0 to 64 · B in parallel do                      ▷ CUDA kernel
2      s_rid ← tid / 64;  s_bid ← R[s_rid];                   ▷ Source block ID
3      s_loc ← tid % 64;                          ▷ This thr. copies s_loc^th obj.
4      s_bitmap ← get_block(s_bid).bitmap;
5      if s_loc < count_set_bits(s_bitmap) then
6          t_loc ← s_loc;                ▷ This thr. copies to t_loc^th free slot
7          for i ← 0 to n do              ▷ Thr. divergence increases with n
8              t_rid ← s_rid + i · B;
9              t_bid ← R[t_rid];                              ▷ Target block ID
10             t_bitmap ← ~get_block(t_bid).bitmap;
11             t_slots ← count_set_bits(t_bitmap);
12             if t_loc < t_slots then
13                 break;                    ▷ Target block t_bid determined
14             else
15                 t_loc ← t_loc − t_slots;

16         s_oid ← nth_set_bit(s_bitmap, s_loc);
17         t_oid ← nth_set_bit(t_bitmap, t_loc);
18         s_ptr ← make_pointer(s_bid, s_oid);
19         t_ptr ← make_pointer(t_bid, t_oid);
20         *t_ptr ← *s_ptr;                                    ▷ Copy all fields
21     end                                   ▷ else: No work for this thread
```

Algorithm 2: place_forwarding_ptrs<T>() : void

```
1  for tid ← 0 to 64 · B in parallel do                      ▷ CUDA kernel
       (same as in move_objects<T>(), lines 2–19)
20         get_block(s_bid).data.forwarding_ptr[s_oid] ← t_ptr;
21     end                                   ▷ else: No work for this thread
```

The algorithms cannot run in one kernel because a forwarding pointer would then overwrite parts of the data segment[6].

3.4 Rewriting Pointers

The heap is now scanned for pointers that must be rewritten. If a pointer points to a location with a forwarding pointer, it is replaced with the forwarding pointer. We utilize the defragmentation candidate bitmap to quickly decide if a pointer must be rewritten, without reading the memory at the pointer location. *We read that location only if we are sure that it contains a forwarding pointer.* This is a key difference compared to other defragmentation systems.

[6]If the size of the first field is 8 bytes (same as a pointer), as in this example, this is harmless. Otherwise, a thread would store a forwarding pointer into a memory location that may be copied by another thread (race condition).

Algorithm 3: rewrite_pointer<T>(T* ptr) : T* GPU

```
1  s_bid ← extract_block_id(ptr);
2  if s_bid < R[B] ∧ defrag_bitmap[T][s_bid] then
3      s_oid ← extract_object_id(ptr);
4      return get_block(s_bid).data.forwarding_ptr[s_oid];
5  else
6      return n/a;
```

```
template<> struct Block<Fish> {
  uint64_t bitmap;
  union {
    struct {
      Cell* position[64];  Cell* new_position[64];  int age[64];
      int random_state[64];  float spawn_probability[64];
    } data_segment;  /* SOA arrays */
    Fish* forwarding_ptr[64];
  } data;
};
```

Listing 1. Example: Block structure for class `Fish`

Given a pointer `ptr`, Alg. 3 returns the corresponding forwarding pointer or *n/a* if no forwarding pointer exists for `ptr`. We first extract the block ID `s_bid` of the object that `ptr` points to. This block is a source block if it is a defragmentation candidate (i.e., bit set in the defragmentation candidate bitmap) and if the block ID is smaller than $R[B]$ (Line 2). Recall that the blocks with IDs $R[i]$ with $i \in [0; B - 1]$ are source blocks and R is sorted (Figure 5). Large parts of the *defrag[T]* bitmap will likely be cached by the L1/L2 caches, so we expected these bitmap lookups to be fast.

If the block is a source block, we extract the object ID `s_oid` from `ptr` and return the corresponding forwarding pointer. It is crucial that Alg. 3 is efficient because it is executed for every pointer that is found during a heap scan.

Limiting Heap Scans Recent GPUs have up to 32 GB of memory, so scanning all allocated memory for pointers to rewrite is expensive. However, since classes and fields of application code are defined with DynaSOAr's DSL, CompactGpu can reflect on application classes and determine which parts of the heap may contain pointers that must be rewritten. As such, only a small part of the heap is scanned.

For example, when defragmenting `Fish` objects, we can avoid looking into blocks of type `Fish` or `Shark`, because those classes do not have fields of type "pointer to `Fish`" or "pointer to a superclass of `Fish`". Only class `Cell` has a field of type `Agent*`, so we only scan the corresponding SOA arrays in the data segment of allocated blocks of type `Cell`. These are the pointers that are rewritten according to Alg. 3.

In general, when defragmenting objects of type T, we first determine all classes U that have at least one field of type *pointer to S*, where $S :> T$ is a supertype of T or equal to T[7]. For every such type U, we scan allocated blocks of type U. For every allocated block, we scan the SOA arrays of fields of type *pointer to S*. Only these pointers are scanned and rewritten. This can exclude more than 95% of the heap. Moreover, reading the values from an SOA array is fast because those field reads are coalesced by the GPU.

Most other allocators have limited information about the structure of their allocations. As such, they cannot restrict the search space as described here. Previous work describes alternative techniques for limiting the search space based on intermediate results from a garbage collector [25].

3.5 Updating Block States

Finally, the state of source and target blocks must be updated in the corresponding block bitmaps. Source blocks lose their *active*, *allocated* and *defrag* states and become *free*. Target blocks may lose their *active* and/or *defrag* states depending on their new fill level.

3.6 Multiple Passes

A single defragmentation pass is guaranteed to delete all source blocks, i.e., $\frac{1}{n+1}$ of all defragmentation candidates. In addition, some target blocks may lose their candidate state. However, a fragmentation level of $\frac{1}{n+1}$ can be achieved only if all candidates were eliminated (Sec. 3). This may require multiple passes.

The efficiency of passes decreases with decreasing numbers of candidates. For example, for $n = 1$, a single pass is guaranteed to eliminate 500 out of 1,000 total candidates. However, the next pass is only guaranteed to eliminate 250 out of 500 remaining candidates. Moreover, too much defragmentation can make allocations more expensive because the allocator has to (re)initialize new blocks if there are not enough active blocks. To reduce runtime overheads, defragmentation should stop before eliminating all candidates.

CompactGpu runs multiple passes until all but k_1 candidates were eliminated. The value of k_1 can be configured. Since passes are very fast, the value of k_1 matters only in cases with a large number passes or massive allocations.

Worst-case Analysis Let d be the number of defragmentation candidates. A single pass reduces d at least by a fraction of $\frac{1}{n+1}$, so no more than $\frac{n}{n+1}$ of candidates are left over. We can bound the number of passes that are necessary in the worst case to eliminate all candidates by:

$$\log_{\frac{n+1}{n}} d$$

If all but $k_1 \geq 1$ candidates should be eliminated, we can bound the number of passes by:

$$\log_{\frac{n+1}{n}} d - \log_{\frac{n+1}{n}} k_1 = \log_{\frac{n+1}{n}} \frac{d}{k_1}$$

We experimentally analyze the actual number of passes in the benchmarks section.

3.7 Defragmentation Frequency

Memory defragmentation must be initiated by the programmer explicitly. Once initiated, CompactGpu may run multiple passes depending on n, k_1 and the number of defragmentation candidates. We suggest one of the two following defragmentation policies for initiating defragmentation.

Every m Iterations Many GPU programs run a number of CUDA kernels iteratively in a loop. This policy initiates defragmentation every m iterations.

After Massive Deallocations Initiate defragmentation if there are at least k_2 many defragmentation candidates[8], where k_2 should be a large enough value. CompactGpu provides a helper method that lets programmers specify this threshold as an absolute number or as a percentage of the heap size and then initiates defragmentation if necessary. Internally, CompactGpu scales k_2 by $\frac{n}{n+1}$ to account for the fact that a larger value of n usually leads to more defragmentation candidates.

As a rule of thumb, we use the first policy for applications that experience a speedup from defragmentation, because even small compactions can lead to a performance gain. The second policy is useful for applications that mainly benefit from better space efficiency or see a slowdown from defragmentation. Future work will investigate how to automate defragmentation (choosing policies, parameters, etc.).

3.8 Pointer Rewriting Alternatives

Pointer rewriting is the most time-consuming step in applications with a large object set. In Alg. 3, reading the forwarding pointer in Line 4 is a random memory access that cannot be coalesced and thus the most expensive operation of the algorithm. To get rid of this memory access, we implemented two alternatives that recompute forwarding pointers on-the-fly.

[7]We also consider fields of type *array of pointer to S* etc. For simplicity, we mention only simple pointer types here.

[8]There is no global object counter. The number of defragmentation candidates approximates the number of allocated but unused object slots.

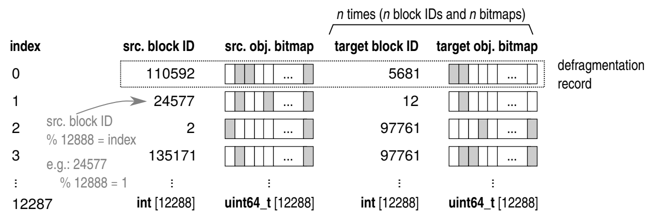


Figure 7. Example: Defragmentation records. Stored in SOA layout.

Recompute-Global*: *Recompute Forwarding Pointers
Instead of storing forwarding pointers in objects, we maintain defragmentation *records* (Figure 7). A record is a tuple of source block ID, source object bitmap (i.e., object allocation bitmap), target block IDs and target object bitmaps. Based on a record, all forwarding pointers for objects from the respective source block can be recomputed (similar to Alg. 1). Records are stored in SOA layout (4 arrays for $n = 1$).

Source block IDs are stored in shared memory[9] and the remaining 3 arrays are stored in global memory. Current NVIDIA GPUs have 48 KB of shared memory, so we can have at most 48 KB / sizeof(**int**) = 12288 records per pass.

During source/target block selection, we hash as many records to array slots as possible and proceed with object copying. The records structure is effectively a fixed-size hash table with the source block ID as key.

We do not store forwarding pointers in objects. During pointer rewriting (Alg. 3), we check in shared memory if there is a record for the block of `ptr` by hashing the block ID and using it as an array index into the source block ID array; instead of checking a bit in the defrag. candidate bitmap. We traded a (maybe cached) global memory access for a (guaranteed) fast shared memory access. However, this approach limits the number of source blocks per pass and potentially increases the number of required passes. Furthermore, we now have to recompute the forwarding pointer, which still requires a global memory accesses for reading the remaining record values in case `ptr` must be rewritten.

Recompute-Shared*: *Entire Records in Shared Memory
To further reduce the number of global memory accesses, we modified Recompute-Global to store the entire records structure in shared memory. The size of a record is $12 \cdot (n+1)$ bytes (4 byte block IDs and 8 byte bitmaps), so the shared memory can hold only 2048 records in shared memory for $n = 1$ (and even less for larger n). This further increases the number of required passes, but also reduces the number of global memory accesses during pointer rewriting.

4 Evaluation

We evaluated CompactGpu with an NVIDIA TITAN Xp GPU (12 GB device memory). We compiled the programs with nvcc (-O3) from the CUDA Toolkit 10.1 on Ubuntu 16.04.4.

[9]Shared memory is an explicitly programmable part of the L1 cache.

4.1 Defragmentation Quality

We first investigate how much fragmentation CompactGpu can eliminate. As described in Sec. 3, given a defragmentation factor n, the fragmentation level is guaranteed to be less than $\frac{1}{n+1}$ after defragmentation. However, in reality the fragmentation level is even lower.

We ran a synthetic benchmark that first allocates a very large number of objects and then randomly deallocates some objects. CompactGpu then defragments the heap with $k_1 = 0$. We measured the fragmentation level after defragmentation for different values of n. In Figure 9, the x-axis denotes the initial heap fragmentation level (i.e., the percentage of deallocated objects) and the y-axis denotes the fragmentation level after defragmentation.

CompactGpu achieves its worst defragmentation quality at an initial fragmentation level that is slightly smaller than $\frac{n}{n+1}$ (e.g, 45% for $n = 1$). In this case, many blocks are at the border of becoming defragmentation candidates. There are a few more points with bad defragmentation quality. For example, around 70% for $n = 1$. In this case, a number of defragmentation candidates were eliminated in the first pass, but the resulting blocks have unfortunate fill levels at the border of becoming a defragmentation candidate.

4.2 Number of Defragmentation Passes

We now investigate the number of defragmentation passes that are necessary to reach good fragmentation levels. The number of passes is bounded by $\log_{\frac{n+1}{n}} d$, but in reality fewer passes are needed because some target blocks lose their state as defragmentation candidates.

We ran the same synthetic benchmark, but fixed the initial fragmentation level at 60%. Figure 8 shows the number of defragmentation candidates and the fragmentation level after every defragmentation pass. Only a few passes bring down fragmentation to very low levels. Moreover, the number of required passes to eliminate all candidates is significantly lower than the theoretical upper bound.

Figure 10 shows the total number of object relocations for the synthetic benchmark (60% frag. level) at various defragmentation factors. CompactGpu runs multiple defragmentation passes, so some objects may be relocated multiple times. This overhead is indicated by the stacked bars above "1".

E.g., for $n = 5$, in 30.0% of all object relocations, an object was copied already for the second time or even more often. Even though 31 passes are required for $n = 5$, no object was relocated more than 5 times.

4.3 Benchmark Applications

We evaluated CompactGpu with four applications (with $k_1 = 16$). Since dynamic memory allocation is not widely used on GPUs yet, there are no suitable standard benchmark suites. Our benchmarks are taken from the DynaSOAr examples and exhibit varying allocation patterns.

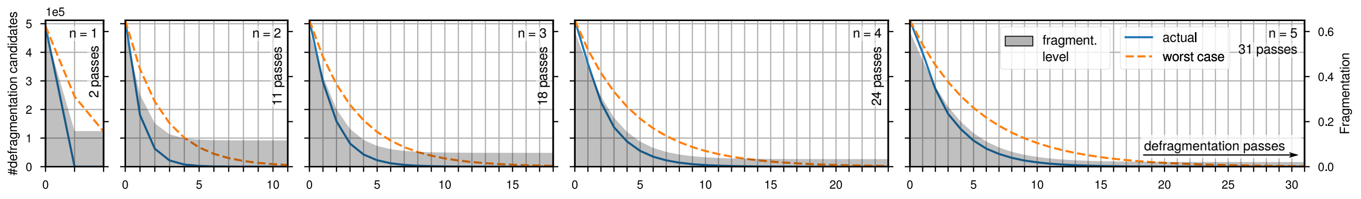


Figure 8. Number of defrag. passes required to eliminate all defrag. candidates (length of the x-axis) for various defrag. factors n. The y-axis shows the number of remaining defrag. candidates and the fragmentation level (grey area). The initial fragmentation level is 60%.

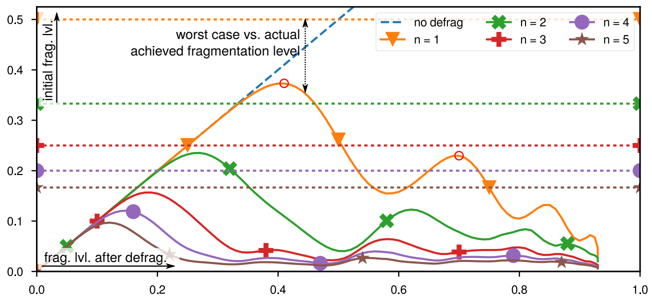


Figure 9. Fragmentation level after memory defragmentation by defragmentation factor n and initial fragmentation level. Lower is better. The dotted lines indicate worst-case fragmentation levels after defragmentation (fragmentation level $\frac{1}{n+1}$).

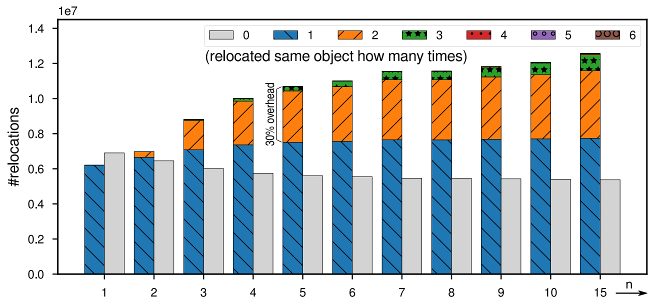


Figure 10. Number of object relocations. The x-axis denotes the defragmentation factor. The stacked bars classify relocations by the number of times an object is relocated (e.g., *0* = object not relocated, *1* = object relocated for the first time, etc.).

We measured the defragmentation quality and running time with different defragmentation factors. The defragmentation factor must be smaller than the capacity of a block, so every problem has a different maximum defragmentation factor. The dashed red lines in the running time graphs are baseline running times without defragmentation.

For every application, we also show a memory profile. The shaded area indicates the number of allocated objects (used object slots). Different colors indicate different C++ classes in the application. The lines indicate the actual memory usage (allocated object slots). The gap between the shaded area and a line is memory that is wasted due to fragmentation.

All but one application experience a speedup. One application experiences a slowdown, but space savings.

Collisions This is an n-body simulation with collisions (Figure 11). A large number of body objects is allocated at the beginning. No other objects are allocated. When two body objects collide, they are merged and one object is deallocated. The fragmentation level increases gradually with every deallocated body. The worst fragmentation is reached around iteration 5,000, when most objects were already deallocated but most blocks are still allocated due to a few remaining objects in each block.

We initiated defragmentation every 50 iterations. Defragmentation had a very small overhead and led to a performance improvement of 12.2% for $n = 36$. This is because of more efficient vector load/store instructions (more coalescing) and due to better cache utilization. Towards the end of the simulation, only few objects remain, and if they are stored in a dense way they fit into GPU caches.

Structure This is a simulation of a fracture in a composite material (Figure 12), modeled as a mesh of finite elements. The simulation exerts a force on some elements and connections between two elements break if the force between them exceeds a certain threshold. A BFS pass identifies elements that are disconnected from the remaining simulation and deallocates them.

Similar to *collisions*, this simulation exhibits only deallocations. However, this simulation has four classes. Even though many objects are already deallocated at the end of the simulation, most blocks are still allocated and overall memory consumption has barely decreased.

We initiated defragmentation every 50 iterations. Defragmentation achieved a peak speedup of 16.3% for $n = 18$.

Generational CA This is an adaptation of Game of Life (Figure 13). After a cell dies, it stays around for a few more iterations and blocks the cell. This implementation simulates only alive cells by allocating objects for alive cells and cells that may become alive in the next iteration.

This simulation contains both allocation and deallocation of objects. After iteration 2000, most cells are dead and the simulation converges into a mostly static pattern.

We initiated defragmentation every 50 iterations for a speedup of 6.3% ($n = 2$). Higher defragmentation factors led to *overfitting*: E.g., the line for $n = 5$ follows the number of allocated objects very closely, even into small local minima. This does not give any additional performance benefit.

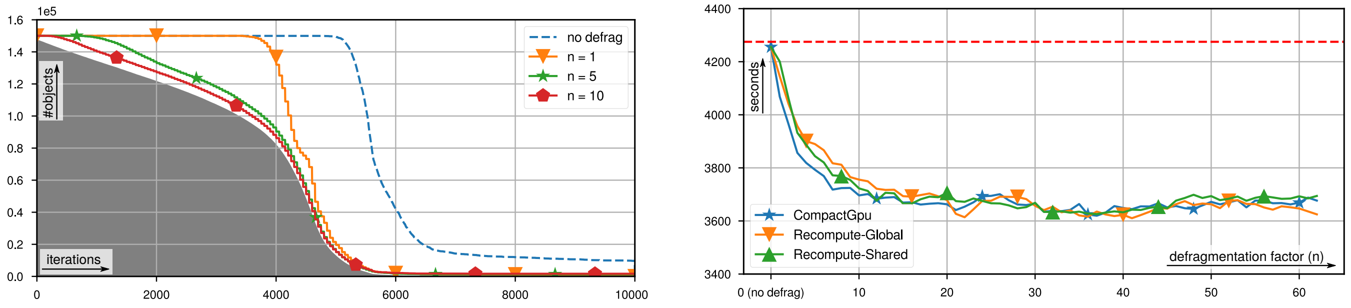


Figure 11. N-body simulation with collisions. Upon collision, two bodies are merged and the smaller object is deleted, leading to fragmentation.

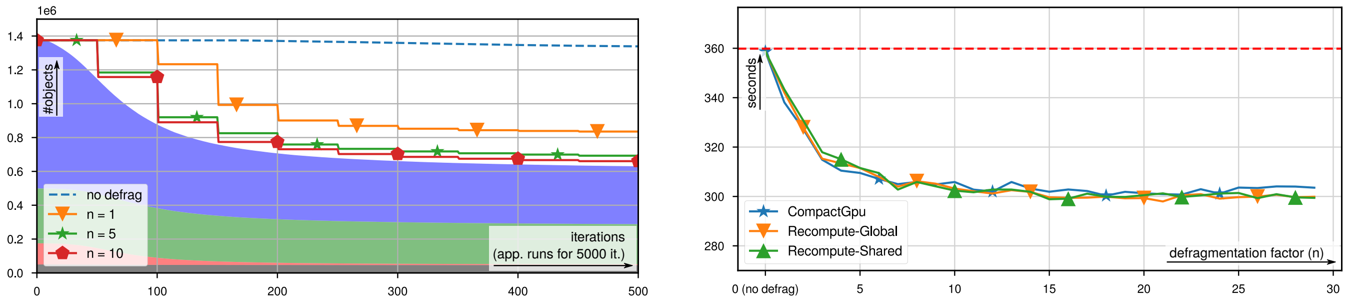


Figure 12. Structure: Simulation of a fracture in a composite material (FEM). Disconnected elements are removed from the simulation.

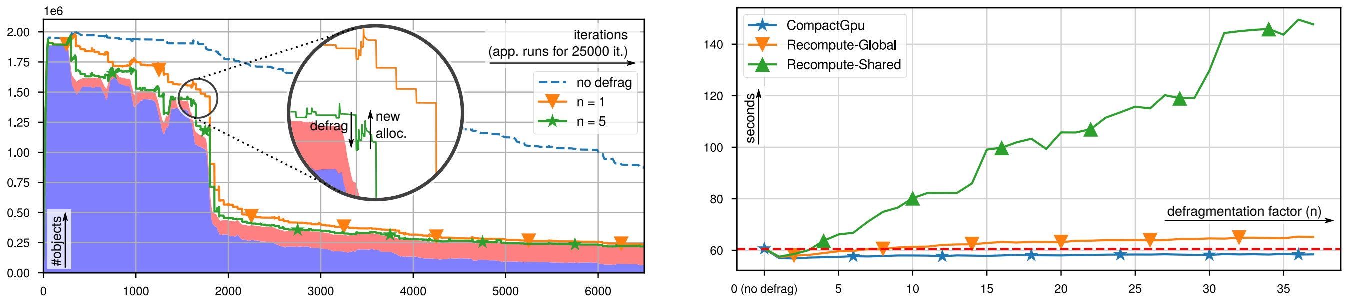


Figure 13. Generational Cellular Automaton. This benchmark is an adaption of Game of Life (rule 0235678/3468/255, similar to "Burst" [27]).

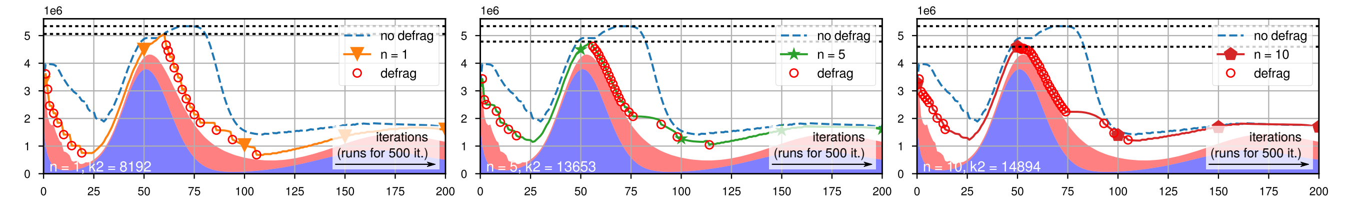
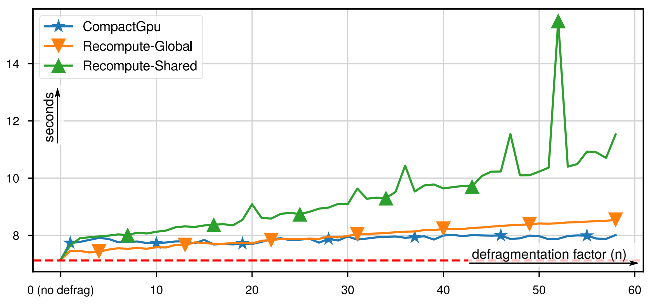


Figure 14. Wa-Tor: An agent-based fish-and-sharks simulation. Simulates population dynamics of agents in a predator-prey relationship. The dotted lines indicate maximum memory usage throughout the simulation. Circles indicate defragmentation runs.

Table 1. Benchmark characteristics and running time (right side; milliseconds) for selected defragmentation factors. If #Passes < #Defrag, the programmer initiated defragmentation but there were not enough defragmentation candidates to start a pass.

Benchmark	Alloc. Size	#Rewr. Fields	*n*	#Defrag	#Passes	Total Runtime	Defrag	Scan	Copy	Rewrite
Synthetic (60% frag.)	2,097.2 MB	1	3	1	18	n/a	44.4	4.0	6.7	33.3
Collisions	5.7 MB	1	10	200	186	3,698,945	36	17	7	8
Generational CA	57.4 MB	1	2	500	537	56,830	191	80	17	85
Structure	58.9 MB	3	10	100	368	305,846	140	54	16	65
Wa-Tor	1,107.6 MB	1	9	38	43	7,729	49	7	14	20



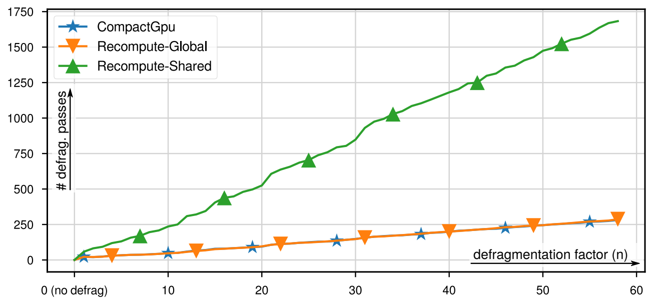


Figure 15. Running time and number of defragmentation passes for fish-and-sharks simulation.

We chose $k_1 = 16$, i.e., 16 defragmentation candidates are excluded from defragmentation. It is important to retain a few fragmented (non-full) blocks because DynaSOAr (and other allocators) first look for active (non-full) blocks during allocations (*fast path*) and have to initialize a new block if none were found (*slow path*). Too small values of k_1 led to *overcompaction*: Consider the enlarged part of the memory profile in Figure 13. At first, defragmentation lowers the overall memory usage. However, allocations in the next few iterations immediately increase the fragmentation level again due to new block initializations, bringing it almost back to the initial fragmentation level. A higher value of k_1 would likely speed up allocations and increase the performance of the overall application a little bit.

Wa-Tor This is the fish-and-sharks running example. Fish and sharks appear in waves until an equilibrium is reached [6].

This simulation experiences a slowdown from defragmentation (Figure 15). At $n = 10$, the slowdown is 8.1%. This slowdown is *not* due to defragmentation runtime overhead. The main reason is that CompactGpu is not order-preserving: Objects from a source block are scattered into multiple target blocks. This leads to less coalesced memory accesses when certain fields are accessed. Similar slowdowns have been reported on certain benchmarks in CPU systems [1]. We will further investigate this effect in future work.

The benefit of defragmentation in Wa-Tor is a lower memory footprint (Figure 14). For $n = 10$, the overall memory consumption is reduced by 14%, so that programmers can run larger problem sizes on the same hardware.

We use the *After Massive Deallocations* heuristic to initiate defragmentation. Initiating defragmentation every few iterations would incur a higher slowdown. To save memory, defragmentation is most important around iteration 50. At that time, many fish objects (red area) are deallocated and a large number of shark objects are allocated. New shark objects can reuse deallocated memory locations of fish objects as soon as the corresponding blocks are deallocated. Defragmentation eliminates many fish blocks by compaction.

Pointer Rewriting Alternatives CompactGpu is faster than Recompute variants in most cases. The high number of bitwise operations for recomputing a memory pointer (similar to Alg. 1), combined with divergent execution (some pointer are rewritten, some are not) led to a high slowdown compared to our forwarding pointer method.

Furthermore, with increasing n, Recompute-Shared requires a larger number of passes (Figure 15) because the shared memory is very small, limiting the number of defragmentation candidates per pass.

For *n-body* and *structure*, this slowdown is negible because very little time is spent on defragmentation overall, but we can see clear difference for *generation* and *wa-tor*.

4.4 Runtime Overhead

To evaluate the efficiency of our implementation, we measured the runtime overhead of CompactGpu. There are two kind of overheads.

First, CompactGpu extends (de)allocation procedures to maintain *defrag[T]* bitmaps. To measure this overhead, we compare the running time without defragmentation (red dashed line) and *no defrag* values in the running time graphs. In *no defrag*, we maintain a defragmentation candidate bitmap but never initiate defragmentation. There is almost no measurable overhead for maintaining these bitmaps.

Second, CompactGpu has three potentially expensive steps: (a) Generating/compacting an indices array R from a defragmentation candidate bitmap (*scan*), (b) copying objects and placing forwarding pointers (*copy*) and (c) scanning the heap and rewriting pointers (*rewrite*). We measure the time spent in each step, as well as the overall time spent on defragmentation (*defrag*), which includes additional overheads such as block state updates (Table 1).

In every benchmark, defragmentation takes only a very small fraction of the overall application running time. Wa-Tor has the largest overhead: The application spends 0.6% of its running time in defragmentation.

The synthetic benchmark isolates the runtime overhead for one defragmentation. We added a second class to the benchmark and made objects of both classes point to each other randomly. There are initially 32,768,000 objects of each class (object size 32 bytes). The benchmark deletes 60% of the objects of one class and initiates defragmentation.

The performance of the *scan* phases mainly depends on the efficiency of the prefix sum operations (CUB library).

The *copy* phases copy (read+write) 282.1 MB of object data and write 70.5 MB of forwarding pointers in 6.7 milliseconds (94.7 GB/s). This is 17.3% of the global memory bandwidth.

The *rewrite* step is most time consuming: CompactGpu has to check 32,768,000 pointers (262.1 MB) per pass (18 × 262.1 MB = 4,717.2 MB in total). Out of these pointers, CompactGpu rewrites only a small part (read+write 70.3 MB of forwarding pointers) because many objects were deleted. CompactGpu finishes the rewrite step in 33.3 milliseconds, resulting in a memory transfer rate of 145.9 GB/s (not taking into account other memory accesses). This is 26.6% of the global memory bandwidth of our TITAN Xp GPU. The Nvidia Profiler shows that 32% of all global memory accesses in this step hit the L1 cache (64 KB) and 63% hit the L2 cache (3,072 KB), indicating that defragmentation candidate bitmaps are largely cached.

CompactGpu achieves a high performance because most memory reads/writes have good coalescing. Overall, our benchmark results show that CompactGpu is highly optimized with little room for improvement.

5 Related Work

A vast number of memory defragmentation systems have been developed for CPU systems in the past. A main difference on GPU architectures is that it is easier to decide where to relocate objects to, because there are only a small number of object sizes. This pattern is reflected in the design of many GPU dynamic memory allocators: Many allocators maintain containers for objects of the same size [2, 7, 22]. On CPU systems, there are typically many different allocation sizes.

The only existing GPU memory defragmentation system was developed by Veldema and Philippsen [25]. Their work consists of an allocator and a defragmentation system[10]. To compact the memory, their defragmentation system selects 10% of all memory regions that are less than 75% full as source regions. They use their memory allocator to allocate a target location in another region. This is problematic because allocation is expensive and requires some sort of synchronization between threads. Runtime overheads of their defragmentation system range from 0.5% to 33%.

Veldema and Philippsen also propose a technique for limiting the search space during pointer rewriting based on additional data collected by a GC. This technique could be used in CompactGpu instead of relying on class structure metainformation of DynaSOAr.

To the best of our knowledge, there are no other defragmentation systems for GPUs. We believe that this is because of limited support for dynamic memory allocation. The default CUDA dynamic memory allocator is known to be slow and unreliable [23], so most programmers avoid dynamic memory management entirely. It is still a common practice to allocate a large chunk of memory statically and manage it manually. Out of the few custom memory allocators that exist, many (e.g., ScatterAlloc [23], Halloc [2]) use a hashing approach to scatter allocations in the heap almost randomly, in order to avoid collisions among allocating threads. Not only do they miss important opportunities for vectorization (e.g., SOA layout), but they are also known to incur the negative effects of high fragmentation [2].

Many efficient CPU memory defragmentation systems divide the heap into two areas: Objects are copied from a *from-space* to a *to-space* [13, 14]. Both spaces are swapped before every defragmentation pass. In such an approach, only half of the memory space is usable by the allocator. This is acceptable on virtual memory architectures because the virtual memory space is much larger than the physical memory space. Current GPU architectures do not have virtual memory and even the amount of physical memory is much smaller than on CPU systems. Cutting the available memory by half would be unacceptable on GPUs.

Pointer Rewriting without Forwarding Pointers Some memory defragmentation systems use data structures different from forwarding pointers [1, 13]. For example, the Compressor uses a markbit vector to recompute forwarding pointers on-the-fly during pointer rewriting [13]. A markbit vector is a bit vector where bits for the first and last heap word of an allocated object are set. Since forwarding pointers are not read from memory, only two accesses (read pointer, replace with new pointer) are required to rewrite a pointer, assuming the markbit vector is cached. We experimented with similar techniques, but they did not lead to a performance improvement (Sec. 3.8).

6 Conclusion

We presented CompactGpu, a memory defragmentation system for GPUs. CompactGpu is able to (a) speed up applications through better cache utilization and vector load/store efficiency on allocated memory and (b) lower the overall memory consumption of an application.

CompactGpu achieves low runtime overheads through careful SIMD-friendly design considerations and implementation efforts: CompactGpu utilizes bitmaps to select source/target blocks and to quickly decide if a pointer must be rewritten. Furthermore, CompactGpu exhibits mostly regular control flow, accesses memory in coalescing-friendly patterns and requires no synchronization between threads.

Our main takeaways are that (a) memory defragmentation on GPUs is feasible and able to deliver speedups, (b) too much defragmentation (overfitting and overcompaction) does not pay off and can even be detrimental to performance due to less efficient allocations, and (c) careful design considerations are necessary to achieve good performance on GPUs; many good CPU designs, such as recomputing forwarding pointers on-the-fly, are not efficient on GPUs.

[10]The source code of this system is not available, so we could not compare it with CompactGpu.

Acknowledgments

This work was supported by a JSPS Research Fellowship for Young Scientists and JSPS KAKENHI Grant Number 18J14726. We gratefully acknowledge the support of NVIDIA Corporation with the donation of the TITAN Xp GPU used for this research.

References

[1] Diab Abuaiadh, Yoav Ossia, Erez Petrank, and Uri Silbershtein. 2004. An Efficient Parallel Heap Compaction Algorithm *(OOPSLA '04)*. ACM, New York, NY, USA, 224–236.

[2] Andrew V. Adinetz and Dirk Pleiter. 2014. Halloc: A High-Throughput Dynamic Memory Allocator for GPGPU Architectures. https://github.com/canonizer/halloc. In *GPU Technology Conference 2014*.

[3] Darius Bakunas-Milanowski, Vernon Rego, Janche Sang, and Chansu Yu. 2017. Efficient Algorithms for Stream Compaction on GPUs. *International Journal of Networking and Computing* 7, 2 (2017), 208–226.

[4] Jeff Bonwick. 1994. The Slab Allocator: An Object-caching Kernel Memory Allocator *(USTC '94)*. USENIX Association, Berkeley, CA, USA, 12.

[5] Shigeru Chiba. 1995. A Metaobject Protocol for C++ *(OOPSLA '95)*. ACM, New York, NY, USA, 285–299.

[6] Alexander K. Dewdney. 1984. Computer Creations: Sharks and fish wage an ecological war on the toroidal planet Wa-Tor. *Scientific American* 251, 6 (Dec. 1984), 14–26.

[7] Isaac Gelado and Michael Garland. 2019. Throughput-oriented GPU Memory Allocation *(PPoPP '19)*. ACM, New York, NY, USA, 27–37.

[8] Dirk Grunwald, Benjamin Zorn, and Robert Henderson. 1993. Improving the Cache Locality of Memory Allocation *(PLDI '93)*. ACM, New York, NY, USA, 177–186.

[9] Holger Homann and Francois Laenen. 2018. SoAx: A generic C++ Structure of Arrays for handling particles in HPC codes. *Computer Physics Communications* 224 (2018), 325–332.

[10] Xiaohuang Huang, Christopher I. Rodrigues, Stephen Jones, Ian Buck, and Wen-Mei Hwu. 2010. XMalloc: A Scalable Lock-free Dynamic Memory Allocator for Many-core Machines *(CIT '10)*. 1134–1139.

[11] Byunghyun Jang, Dana Schaa, Perhaad Mistry, and David Kaeli. 2011. Exploiting Memory Access Patterns to Improve Memory Performance in Data-Parallel Architectures. *IEEE Transactions on Parallel and Distributed Systems* 22, 1 (Jan. 2011), 105–118.

[12] Mark S. Johnstone and Paul R. Wilson. 1998. The Memory Fragmentation Problem: Solved? *(ISMM '98)*. 26–36.

[13] Haim Kermany and Erez Petrank. 2006. The Compressor: Concurrent, Incremental, and Parallel Compaction *(PLDI '06)*. ACM, New York, NY, USA, 354–363.

[14] Bernard Lang and Francis Dupont. 1987. Incremental Incrementally Compacting Garbage Collection. In *Papers of the Symposium on Interpreters and Interpretive Techniques (SIGPLAN '87)*. ACM, New York, NY, USA, 253–263.

[15] Roland Leißa, Immanuel Haffner, and Sebastian Hack. 2014. Sierra: A SIMD Extension for C++ *(WPMVP '14)*. ACM, New York, NY, USA, 17–24.

[16] Justin Luitjens. 2011. Global Memory Usage and Strategy. https://developer.download.nvidia.com/CUDA/training/cuda_webinars_GlobalMemory.pdf. (July 2011). GPU Computing Webinar 7/12/2011, Accessed: 2019-02-28.

[17] Toni Mattis, Johannes Henning, Patrick Rein, Robert Hirschfeld, and Malte Appeltauer. 2015. Columnar Objects: Improving the Performance of Analytical Applications *(Onward! 2015)*. ACM, New York, NY, USA, 197–210.

[18] Duane Merrill and Michael Garland. 2016. *Single-pass Parallel Prefix Scan with Decoupled Look-back*. Technical Report NVR-2016-002. NVIDIA Corporation.

[19] Yoav Ossia, Ori Ben-Yitzhak, and Marc Segal. 2004. Mostly Concurrent Compaction for Mark-sweep GC *(ISMM '04)*. ACM, New York, NY, USA, 25–36.

[20] Matt Pharr and William R. Mark. 2012. ispc: A SPMD compiler for High-Performance CPU Programming *(InPar 2012)*. IEEE Computer Society, 1–13.

[21] Jie Shen, Ana Lucia Varbanescu, Xavier Martorell, and Henk Sips. 2015. *A Study of Application Kernel Structure for Data Parallel Applications*. Technical Report PDS-2015-001. Delft University of Technology.

[22] Matthias Springer and Hidehiko Masuhara. 2019. DynaSOAr: A Parallel Memory Allocator for Object-oriented Programming on GPUs with Efficient Memory Access *(ECOOP 2019)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany.

[23] Markus Steinberger, Michael Kenzel, Bernhard Kainz, and Dieter Schmalstieg. 2012. ScatterAlloc: Massively Parallel Dynamic Memory Allocation for the GPU *(InPar 2012)*. IEEE Computer Society, 1–10.

[24] Robert Strzodka. 2012. Chapter 31 - Abstraction for AoS and SoA Layout in C++. In *GPU Computing Gems Jade Edition*. Morgan Kaufmann, Boston, 429–441.

[25] Ronald Veldema and Michael Philippsen. 2012. Parallel Memory Defragmentation on a GPU *(MSPC '12)*. ACM, New York, NY, USA, 38–47.

[26] Vasily Volkov. 2016. *Understanding Latency Hiding on GPUs*. Ph.D. Dissertation. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-143.html

[27] Mirek Wójtowicz. 2002. Mirek's Cellebration: Cellular Automata Rules Lexicon. http://psoup.math.wisc.edu/mcell/rullex_gene.html. (2002). Accessed: 2019-02-22.