CompactGpu: Massively Parallel Memory Defragmentation on GPUs

Matthias Springer (Tokyo Institute of Technology)
東京工業大学 Tokyo Institute of Technology

https://github.com/prg-titech/dynasoar

Why Defragment GPU Memory?

- Space efficiency: Reduce memory usage
- Improve runtime performance:
  Accessing compact data requires fewer vector accesses
  → Better memory coalescing

Design Requirements

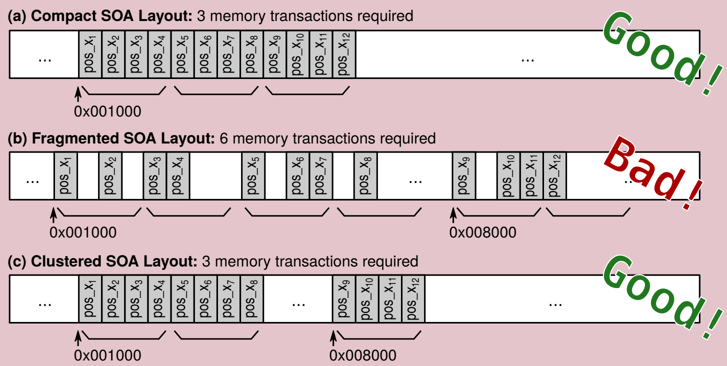
- Extension to the DynaSOAr dynamic GPU memory allocator
- Parallel, in-place, stop-the-world defragmentation approach
- To reduce defragmentation overhead: Uniform control flow, little synchronization, efficient memory access

Related Work

- R. Veldema, M. Philippsen. Parallel Memory Defragmentation on GPUs. MSPC '12
  Assumes many different allocation sizes, not in-place, large runtime overhead
- M. Springer, H. Masuhara. DynaSOAr: A Parallel Memory Allocator for Object-oriented Programming on GPUs with Efficient Memory Access. ECOOP '19
- H. Boehm. Space Efficient Conservative Garbage Collection. PLDI '93
  Similar problem: How to find all pointers to moved objects that must be rewritten?

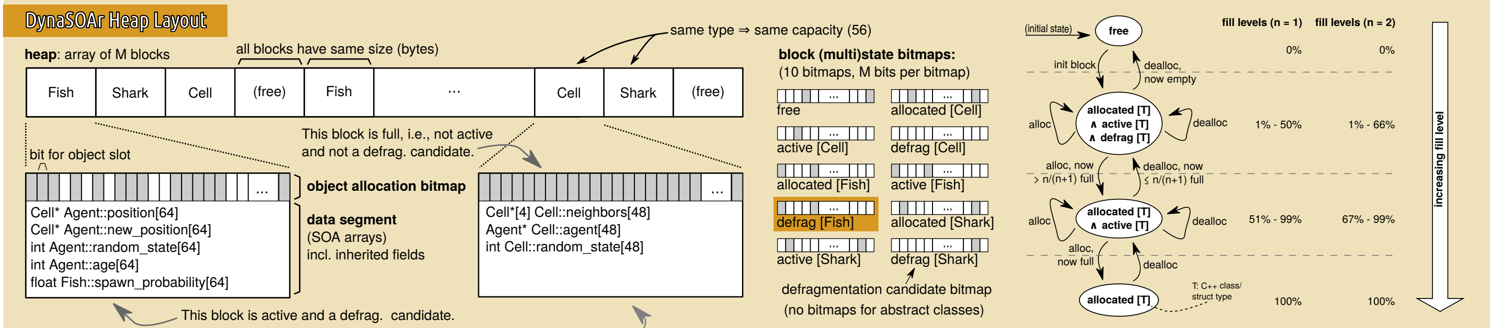
Background: GPU Architecture and Dyn. Memory Allocation

- Pattern: Many small allocations, mostly same size
- For good mem. access performance: Structure of Arrays (SOA) data layout
- Recent NVIDIA GPUs have 128-byte vector registers
  → Memory access in aligned, 128-byte transactions



For illustration purposes: Vector length 32 byte (4 scalars) instead of 128 byte (32 scalars). N-body sim.

DynaSOAr Heap Layout



Running example:
Fish-and-Sharks simulation

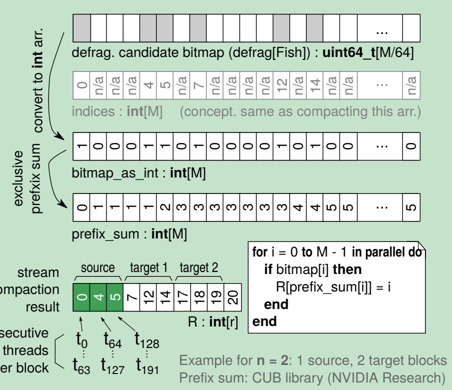
Defragmentation by Block Merging: parallel_defrag<Fish>()

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

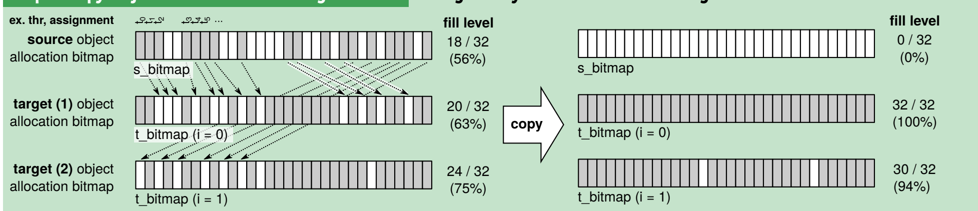
Definition of Defragmentation Candidates: Depends on defrag. factor n (problem-spec., compile-time parameter)

n = 1:	n = 2:	Arbitrary n:	Guaranteed target frag.: 1/(n+1)
≤ 50% full	≤ 66.6% full	≤ n/(n+1) full	

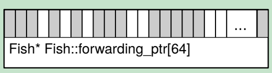
Step 1: Choose Src./Target Blocks



Step 2: Copy Objects From Source to Target Blocks: Merge every source block into n target blocks.



Step 3: Store Forwarding Ptrs. in Source



Overwrite data segment with pointers.

Step 4: Rewrite Pointers to Relocated Objects with Bitmap

```
for all Fish*& ptr in parallel do
    s_bid = extract_block_id(ptr)
    if defrag[Fish][s_bid] then
        s_oid = extract_object_id(ptr)
        ptr = heap[s_bid].forwarding_ptr[s_oid]
    end
end
```

How to find all Fish*/Agent* values on the heap?

- Option 1: Scan heap, look for anything that could be a pointer. *Slow!*
- Option 2: Utilize DynaSOAr's data layout DSL. Scan only mem. locations of SOA arrays with base type Fish*/Agent*. *Fast!*
  Fast: defrag[T] bitmap largely cached in L2

Memory transactions: 2 memory reads + 1 write for relocated objects, 1 memory read for all others

Step 5: Update Block State Bitmaps

Blocks may now be empty, full and/or no longer defrag. candidates.

Step 6: If there are > n defrag. candidates left, go to Step 1.

Generalization: Other Allocators?

Many other GPU allocators (Halloc, ScatterAlloc) use hashing (very high frag.) and do not utilize SOA.