CompactGpu: Massively Parallel Memory Defragmentation on GPUs

Matthias Springer
Tokyo Institute of Technology

ACM SRC @ PLDI 2019
Introduction / Motivation

- **Goal:** Make GPU programming easier to use.
- **Focus:** Object-oriented programming on GPUs/CUDA.
  - Many OOP applications in high-performance computing.
  - DynaSOAr [1]: Dynamic memory allocator for GPUs.
  - **CompactGpu:** Make allocations more space/runtime efficient with memory defragmentation.

Why Defragment GPU Memory?

- **Space Efficiency:** Reduce overall memory consumption (and prevent premature out of memory errors).

- **Runtime Efficiency:** Accessing compact data requires fewer vector transactions and benefits cache utilization.

```
(a) Compact SOA Layout: 3 memory transactions required

(b) Fragmented SOA Layout: 6 memory transactions required

(c) Clustered SOA Layout: 3 memory transactions required
```
GPU Allocation Characteristics

- **Massive number** of concurrent allocations.
- Most allocations are small and have the **same size** (due to mostly uniform control flow).
- Allows us to optimize defragmentation more than on CPUs.
Related Work / State of the Art

● Dynamic GPU Memory Allocation
  - Not well supported until recently, so not widely utilized yet.
  - Default CUDA allocator (malloc/free): Unoptimized and extremely slow.
  - Halloc [2], ScatterAlloc/mallocMC [3]: Fast (de)allocation time, but high fragmentation (hashing).
  - DynaSOAr: My own allocator, with additional optimizations for structured data (objects).

● GPU Memory Defragmentation [4]
  - High runtime overhead (up to 50%).
  - Different assumptions about allocation pattern.
  - Uses a memory allocator for moving allocations in memory.

DynaSOAr Heap Layout

**Heap:** array of M blocks

- Fish
- Shark
- Cell
- (free)
- Fish
- ...
- Cell
- Shark
- (free)

- **Object Allocation Bitmap**
- **Data Segment**
  - (SOA arrays)
  - incl. inherited fields

- Contains 48 objects in **Structure of Arrays** (SOA) data layout

- This block is full, i.e., not active and not a defrag. candidate.
- This block is active and a defrag. candidate.

- Running example:
  - Fish-and-Sharks simulation

- Cell*: Agent::position[64]
- Cell*: Agent::new_position[64]
- int Agent::random_state[64]
- int Agent::age[64]
- float Fish::spawn_probability[64]
DynaSOAr Heap Layout

**heap**: array of M blocks

<table>
<thead>
<tr>
<th>Fish</th>
<th>Shark</th>
<th>Cell</th>
<th>(free)</th>
<th>Fish</th>
<th>...</th>
<th>Cell</th>
<th>Shark</th>
<th>(free)</th>
</tr>
</thead>
</table>

- **All blocks have same size (bytes)**
- **No fragmentation. GOOD!**
- **This block is full, i.e., not active and not a defrag. candidate.**

**object allocation bitmap**

- **data segment** (SOA arrays) incl. inherited fields
  - Cell* Agent::position[64]
  - Cell* Agent::new_position[64]
  - int Agent::random_state[64]
  - int Agent::age[64]
  - float Fish::spawn_probability[64]
- **Contributes to fragmentation. BAD!**

**Running example:**
Fish-and-Sharks simulation

- **No fragmentation. GOOD!**
Block Merging: $1 + 1 = 1$

Do this in parallel for all eligible blocks:

- Take 2 blocks
- ≤ 50% full
- ≤ 50% full

06/25/19

@ PLDI 2019
Block Merging: $1 + 2 = 2$

Do this in parallel for all eligible blocks:

```
≤ 66% full + ≤ 66% full + ≤ 66% full
```

This block is active and a defrag candidate.

```
heap: array of M blocks
all blocks have same size (bytes)
```

```
Fish Shark Cell (free) Fish ... Cell Shark (free)
```

```
bit for object slot
```

```
Cell* Agent::position[64]
Cell* Agent::new_position[64]
t Agent::random_state[64]
t Agent::age[64]
float Fish::spawn_probability[64]
```

```
data segment
(SOA arrays)
incl. inherited fields
```

```
06/25/19
CompactGpu - ACM SRC @ PLDI 2019
```
Block Merging: $1 + n = n$

- Higher $n$: Better defragmentation guarantees.
- Lower $n$: A bit faster, fewer passes.
- $n$ is can be configured by the programmer.
Pointer Rewriting

- Rewrite pointers to objects that were moved.

- Basic Ideas:
  - Store **forwarding pointers** in source blocks.
  - Allocator has knowledge about the structure (fields, classes) of the data it is allocating. **No need to scan the entire heap.**
  - Quickly decide if a pointer must be rewritten with **bitmaps** that fit in the L2 cache.
Benchmark Results: n-body

This benchmark:
Defragmentation is about 0.5% of the total running time.
Conclusion

- Efficient memory defragmentation is feasible on GPUs.
- Besides saving memory, defragmentation makes usage of allocated memory more efficient: Better cache utilization and better vectorized access.
- GPU allocation patterns allow us to implement defragmentation very efficiently.
  - Choosing source/target blocks: Parallel prefix sum.
  - Copying objects: Very efficient due to SOA layout.
  - Rewriting pointers: Fast due to many optimizations that reduce #memory accesses (bitmaps, restricting heap scan areas).
References


