Massively Parallel GPU Memory Compaction

Matthias Springer, Hidehiko Masuhara
Tokyo Institute of Technology

ISMM 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**: Memory defragmentation for GPUs, to make allocations more space/runtime efficient.

Outline

1. Background: GPU Architecture
2. Memory Defragmentation: Concept and Main Ideas
3. Defragmentation: Step by Step
4. Benchmarks
5. Conclusion
Background: GPU Architecture
Memory Coalescing

If the threads of a physical core access memory within the same aligned 128-byte window (L1/L2 cache line), the those accesses are combined into 1 memory transaction by the memory controller.

Source: CUDA C Programming Guide

Because the hardware really operates on 128-byte vector registers.
Worst Case: No Memory Coalescing

Threads of a physical core (warp) access memory of totally different L1/L2 cache lines.

Before attempting any other optimization, try to improve memory coalescing!

<table>
<thead>
<tr>
<th>Addresses</th>
<th>96</th>
<th>128</th>
<th>160</th>
<th>192</th>
<th>224</th>
<th>256</th>
<th>288</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads:</td>
<td>0</td>
<td>...</td>
<td>...</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Compute capability: 2.0 and later

Memory transactions: 32x 128B = 4096B
Why GPU Memory Defragmentation?

- **Space Efficiency**: Reduce overall memory consumption.
  - Avoid premature out-of-memory errors.
- **Runtime Efficiency**: Vectorized access is more efficient.
  - Accessing compact data requires fewer vector transactions (→ more memory coalescing) than accessing fragmented data.
Memory Defragmentation: Concept and Main Ideas
Dynamic Memory Allocation on GPUs

• Until recently, not supported well and not widely utilized yet
• Existing dynamic GPU memory allocators
  – CUDA allocators (new/delete): Extremely slow and unoptimized
  – Halloc [1], ScatterAlloc/mallocMC [2]: Very fast (de)allocation time
  – DynaSOAr [3]: Fast (de)allocation time, efficient access of allocations
• Memory allocation characteristics on GPUs
  – Massive number of concurrent (de)allocations
  – Most allocations are small and have the same size (due to mostly regular control flow)

Overview

- **CompactGpu**: A memory defragmentation system for the DynaSOAr memory allocator.
  - *Basic Idea*: Defragmentation by block merging.
  - *Optimization*: Fast pointer rewriting based on bitmaps.
  - Main CompactGpu techniques could be implemented in other allocators.
Main Design Choices and Requirements

- **In-place** defragmentation: To save space...
  - Defrag. by **block merging**: Combine blocks that are partly full.

- **Fully parallel** implementation
  - CompactGpu is a set of CUDA kernels.

- **Stop-the-world** approach: Run defragmentation when no other GPU code is running.

- **Manual**: Programmers initiate defragmentation manually or use a heuristic (e.g., defrag. after a large number of deallocations).
Overview: DynaSOAr Mem. Allocator

- Always allocate in active (non-full) blocks.
- Objects of same type stored in blocks in SOA data layout.

Block States

- **free**: Block is empty
- **allocated [T]**: Block contains at least 1 object of type T.
- **active [T]**: Block is allocated [T] and has at least 1 free slot.
- **defrag [T]**: Block is active [T] and is a *defragmentation candidate* (block with low fill level).

T: C++ class/struct type
new with CompactGpu

Block States

- **free**: Block is empty
- **allocated [T]**: Block contains at least 1 object of type T.
- **active [T]**: Block is allocated [T] and has at least 1 free slot.
- **defrag [T]**: Block is active [T] and is a *defragmentation candidate* (block with low fill level).
Defragmentation Factor

- $n$ is the problem-specific \textbf{defragmentation factor} that must be chosen at compile time.
  - Consider only blocks of fill level $\leq n/(n+1)$ for defragmentation (defrag. candidates).
  - Move objects \textbf{from 1 source block into $n$ target blocks}.
  - One defragmentation pass eliminates $1/(n+1)$ of all defragmentation candidates. Run \textbf{multiple passes} to eliminate all candidates.
  - Example: $n = 1$: Merge 2 blocks of fill level $\leq 50\%$.
  - Example: $n = 2$: Merge 3 blocks of fill level $\leq 66.6\%$.
  - In each case, the \textbf{source block is eliminated} by defragmentation.

- Higher $n \rightarrow$ More defragmentation
- Lower $n \rightarrow$ Less defragmentation, but faster (less work)
Block States

(initial state) → free

init block → dealloc, now empty

allocated [T] ∧ active [T] ∧ defrag [T]

alloc → dealloc, now empty

dealloc → dealloc, now empty

alloc, now > 50% full → dealloc, now ≤ 50% full

allocated [T] ∧ active [T]

alloc → dealloc

dealloc → dealloc

alloc, now full → dealloc

allocated [T]

T: C++ class/struct type

fill levels (n = 1)

0%

1% - 50%

51% - 99%

100%

increasing fill level

06/23/2019

CompactGpu - ISMM 2019
Block States

(initial state)  free

init block  dealloc, now empty

allocated [T]  ∧  active [T]  ∧  defrag [T]

alloc  dealloc  1% - 50%  1% - 66%

alloc, now > \(\frac{n}{n+1}\) full  dealloc, now \(\leq \frac{n}{n+1}\) full

allocated [T]  ∧  active [T]

alloc  dealloc  51% - 99%  67% - 99%

allocated [T]

T: C++ class/struct type

fill levels (n = 1)  fill levels (n = 2)

0%  0%

100%  100%
Block State Bitmaps

- DynaSOAr/CompactGpu indexes states in **block state bitmaps**.
- Newly introduced with CompactGpu: **defrag[T]**
Definition of Fragmentation

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

(considering only allocated blocks)
Definition of Fragmentation

**heap**: array of \( M \) blocks

All blocks have same size (bytes)

<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>

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

**Data segment** (SOA arrays)
Incl. inherited fields

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

**Object allocation bitmap**

- Guaranteed frag. level after defrag.: \( \leq \frac{1}{n+1} \)
  (Because all blocks with fill level \( \leq \frac{n}{n+1} \) are gone.)

**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**
(no bitmaps for abstract classes)

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

(considering only \( \text{allocated}[?] \) blocks)
Defragmentation: Step by Step
Choose Source/Target Blocks

- Compact defrag[T] bitmap.
  (exclusive prefix sum)

- Choose n target blocks for each source blocks.
Defragmentation by Block Merging

- Copy objects from a source block to $n$ target blocks (in parallel).
- Source block is empty (new state: **free**), reducing fragmentation.
- **In-place** defragmentation mechanism.
Rewriting Pointers to Old Locations

- Store forwarding pointers in source blocks.

- **Afterwards:** Scan heap and find pointers to relocated objects. Rewrite those pointers.
Rewriting Pointers to Old Locations

- Scan heap and look for anything that looks like a pointer.
- Rewrite if $\text{bid} < \text{R}[r/n]$ and block is a defrag. candidate.

```latex
\begin{center}
\begin{tabular}{l}
\textbf{for all} Fish*\& ptr in parallel do \\
\hspace{1em} s_bid = extract_block_id(ptr) \\
\hspace{1em} if s_bid < R[n] \&\& defrag[Fish][s_bid] then \\
\hspace{2em} s_oid = extract_object_id(ptr) \\
\hspace{2em} ptr = heap[s_bid].forwarding_ptr[s_oid] \\
\hspace{1em} end \\
\end{tabular}
\end{center}
```

**Condition 1:** $\text{bid} < 7$ (i.e., source range)

**Condition 2:** $\text{defrag[Fish][bid]}$ (i.e., defrag. cand.)
Rewriting Pointers to Old Locations

- Scan heap and look for anything that looks like a pointer.
- Rewrite if $\text{bid} < R[r/n]$ and block is a defrag. candidate.

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

Condition 1: $\text{bid} < 7$ (i.e., source range)

Condition 2: $\text{defrag}[\text{Fish}][\text{bid}]$ (i.e., defrag. cand.)

- Defrag bitmap largely cached.
- **2 mem. reads + 1 write** if pointer rewritten
- **1 mem. read** otherwise
Benchmarks
Benchmark: N-Body with Collisions

- Memory consumption drops faster.
- Performance improvement: 12%
Benchmark: Generational Cellular Automaton

- Memory consumption drops faster.
  - Too much defragmentation leads to overcompaction.
- Performance improvement: 6%
Conclusion
Conclusion

- Efficient memory defragmentation is **feasible on GPUs**.
- Besides saving memory, defragmentation makes usage of allocated memory more efficient (**better mem. coalescing**).
- GPU memory allocation patterns allow us to implement defragmentation efficiently.
- Certain CPU techniques (e.g., recomputing forwarding pointers on the fly [1]) do not pay off on GPUs.

Appendix: Microbenchmarks
Achieved Fragmentation Level

worst case vs. actual achieved fragmentation level

frag. after defrag

initial frag.
Number of Defragmentation Passes
Number of Object Copies

(relocated same object how many times)
Benchmark: N-Body with Collisions

- Memory consumption drops faster.
- Performance improvement: 12%
Benchmark: Generational Cellular Automaton

- Memory consumption drops faster.
  - Too much defragmentation leads to overcompaction.
- Performance improvement: 6%
Reducing Heap Scan Area

- Allocator has detailed information about the **structure of allocations**.
- Only **cell** has a pointer to **Agent**. Only look into **allocated[Cell]** blocks.
Background: GPU Architecture

- 20 symmetric multiprocessors (SMs)
- 128 CUDA cores per SM
- Total: $20 \times 128 = 2560$ CUDA cores
- But in reality: $20 \times 4$ physical cores, each operating on 128-byte vector registers

CUDA gives programmers the illusion of having 2560 cores.

Memory controller accesses memory in 128-byte blocks

Source: NVIDIA GeForce GTX 1080 Whitepaper