CompactGpu: Massively Parallel Memory Defragmentation on GPUs

PLDI'19 Student Research Competition – Extended Abstract

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

Abstract

We propose CompactGpu, an incremental, fully-parallel, in-place memory defragmentation system for GPUs. CompactGpu 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.

Keywords CUDA, Memory Defragmentation

Extended Abstract

Memory fragmentation is a challenging problem of dynamic memory allocators and has been widely studied on single-core and multi-core CPU systems. However, despite the recent popularity of massively parallel SIMD architectures such as GPUs, the memory fragmentation problem has not been studied thoroughly on such architectures.

We have to study memory (de)fragmentation on such architectures because allocations follow different patterns: Most allocations are small in size and due to mostly regular control flow, many allocations have the same byte size. Such specialties must be exploited by memory defragmentation systems to achieve good performance.

Effects of Fragmentation 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 [4].
- *Low Vector Load/Store Efficiency:* SIMD vector/load store instructions are less efficient.

Background and Contributions We developed CompactGpu, an incremental, fully-parallel, in-place memory defragmentation systems for GPUs on top of the DynaSOAr dynamic memory allocator [8] for CUDA. While the basic concept of CompactGpu is applicable to many other GPU allocators, we chose DynaSOAr because it allocates objects in a Structure of Arrays (SOA) data layout and lets us explore the effect of fragmentation on vector load/store instructions.

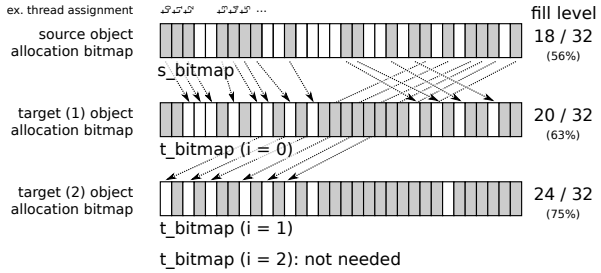

Figure 1. Relocating objects from a source block to 3 target blocks ($n = 3$). All blocks have a fill level of no more than $\frac{n}{n+1} = 75\%$. Only 2 target blocks are required in this example. The third target block remains untouched.

CompactGpu is built upon ideas from other systems. Similar to the only other GPU memory defragmentation system by Veldema and Philippsen [9], CompactGpu merges memory blocks that have a low fill level. As many other systems, CompactGpu stores *forwarding pointers* at old memory locations [2, 6] and performs a subsequent heap scan to rewrite pointers to relocated objects; other techiques (e.g., recomputing pointers on-the-fly [5]) did not pay off. CompactGpu improves upon other systems with three main ideas.

Bitmaps CompactGpu uses bitmaps to choose source/target blocks and to quickly decide if a pointer must be rewritten.

Configurability The desired fragmentation rate after defragmentation can be configured.

Efficient Implementation CompactGpu exhibits mostly regular control flow, accesses memory in GPU-friendly patterns and requires no synchronization between threads.

Requirements To apply the CompactGpu technique to an existing GPU allocator, we require that...

- ... the allocator is block based.
- ... a block contains allocations of a fixed byte size.
- ... every block has the same byte size, so all blocks of a given allocation size have the same number of objects.

Many state-of-the-art GPU allocators fulfill these requirements [1, 3, 8].

*Academic Advisor: Hidehiko Masuhara, Tokyo Institute of Technology, masuhara@acm.org

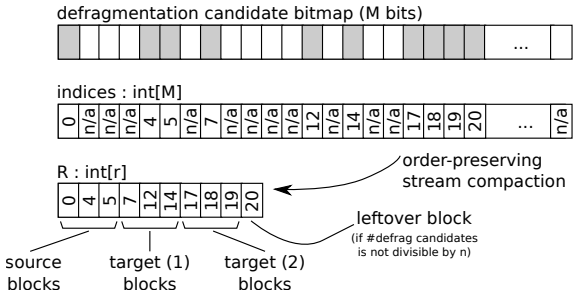

Figure 2. Computing a compact indices array from a defragmentation candidate bitmap and determining source/target blocks ($n = 2$).

We define fragmentation as the *average free level* of all blocks that contain allocations. E.g., if blocks are on average 40% full, then the fragmentation level is 60%. This means that the memory consumption could be lowered by 60% if blocks were compacted.

Design of CompactGpu CompactGpu compacts the heap by merging a source block into n target blocks, where n is a configurable parameter, assuming that both blocks have the same allocation size.

A source block can be merged into n target blocks if all $n+1$ blocks are no more than $\frac{n}{n+1}$ full (Figure 1). We call non-empty blocks with a fill level of $\leq \frac{n}{n+1}$ *defragmentation candidates*. A single defragmentation pass is guaranteed to elimate all source blocks, i.e., $\frac{1}{n+1}$ of all defragmentation candidates. In addition, some target blocks may become so full that they are no longer defragmentation candidates. CompactGpu compacts the heap by running multiple of defragmentation passes. After multiple passes, when all defragmentation candidates are eliminated, the fragmentation level is guaranteed to be lower than $1 - \frac{n}{n+1} = \frac{1}{n+1}$, because only blocks with a fill level of more than $\frac{n}{n+1}$ or higher are left over.

CompactGpu maintains a defragmentation candidate bitmap for every possible allocation size by extending (de)allocation procedures of DynaSOAr. To run a defragmentation pass for a certain allocation size, CompactGpu converts the bitmap into an indices array and compacts it using a parallel prefix sum pass[1] (Figure 2).

If there are r defragmentation candidates, then the first $B = \left\lfloor \frac{r}{n+1} \right\rfloor$ blocks with indices $R[0]$ through $R[\left\lfloor \frac{r}{n+1} \right\rfloor - 1]$ are source blocks. Given a source block $R[rid]$, its corresponding target blocks are:

$$\left\{ R[rid + i \cdot B] \;\middle|\; i \in 1 \ldots n \right\}$$

Objects are moved from source blocks to target blocks in parallel. No synchronization is required among threads. Afterwards, forwarding pointers are stored in the source blocks and the heap is scanned for pointers that should be rewritten. We utilize the defragmentation candidate bitmap and the array R to quickly decide if a pointer ptr must be rewritten.

[1]We use the prefix sum implementation of the CUB library [7].

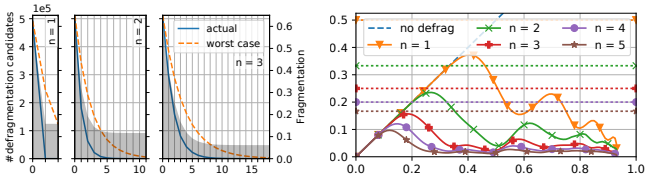

Figure 3. Synthetic benchmark. (left) Number of defragmentation passes. (right) x-axis: initial fragmentation, y-axis: fragmentation after defragmentation.

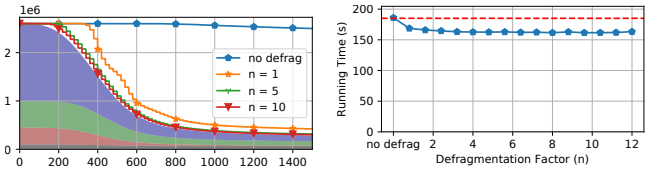

Figure 4. Benchmark: Simulation of a fracture in composite material (finite element method).

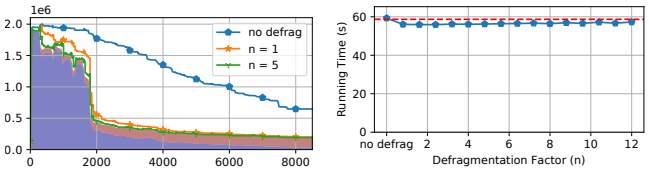

Figure 5. Benchmark: Generational cellular automaton.

Assuming ptr points to an object in block *bid*, we retrieve the forwarding pointer from that block only if $bid < R[B]$ and the bit *bid* in the defragmentation candidate bitmap is set. Defragmentation candidate bitmaps are small and fit into L1/L2 caches, and $R[B]$ is constant during a defragmentation pass, so pointer rewriting is very fast.

Preliminary Evaluation We evaluted CompactGpu on an NVIDIA Titan Xp with a synthetic benchmark that first allocates many objects and then deallocates 60% of them at random (leading to 60% initial fragmentation). Figure 3 (left) shows the number of defragmentation passes required to eliminate all defragmentation candidates for different values of n, along with the achieved fragmentation level. Just 1–3 passes are enough to eliminate most fragmentation. In the right graph, the initial fragmentation level is changed (x-axis) and the achieved target fragmentation rate is shown on the y-axis. The achieved fragmentation rate is much lower than the theoretical upper bound of $\frac{1}{n+1}$ (dashed lines).

Figures 4 and 5 show a memory profile (left) and application running time (right) of two DynaSOAr benchmarks, for various values of n. The memory profile shows the overall memory consumption (lines) and the actual sum of all allocations broken down by allocation size (shaded area). CompactGpu can significantly lower memory usage and speed up applications by up to 14.5% due to better cache utilization and more efficient vector access. Memory access performance gains are higher than CompactGpu's overhead.

References

[1] 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*.

[2] Christine H. Flood, David Detlefs, Nir Shavit, and Xiaolan Zhang. 2001. Parallel Garbage Collection for Shared Memory Multiprocessors. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1 (JVM'01)*. USENIX Association, Berkeley, CA, USA, 21–21.

[3] Isaac Gelado and Michael Garland. 2019. Throughput-oriented GPU Memory Allocation. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*. ACM, New York, NY, USA, 27–37. https://doi.org/10.1145/3293883.3295727

[4] Dirk Grunwald, Benjamin Zorn, and Robert Henderson. 1993. Improving the Cache Locality of Memory Allocation. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation (PLDI '93)*. ACM, New York, NY, USA, 177–186. https://doi.org/10.1145/155090.155107

[5] Haim Kermany and Erez Petrank. 2006. The Compressor: Concurrent, Incremental, and Parallel Compaction. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, New York, NY, USA, 354–363. https://doi.org/10.1145/1133981.1134023

[6] Simon Marlow, Tim Harris, Roshan P. James, and Simon Peyton Jones. 2008. Parallel Generational-copying Garbage Collection with a Block-structured Heap. In *Proceedings of the 7th International Symposium on Memory Management (ISMM '08)*. ACM, New York, NY, USA, 11–20. https://doi.org/10.1145/1375634.1375637

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

[8] Matthias Springer and Hidehiko Masuhara. 2019. DynaSOAr: A Parallel Memory Allocator for Object-oriented Programming on GPUs with Efficient Memory Access. *CoRR* abs/1810.11765 (Jan. 2019). arXiv:1810.11765

[9] Ronald Veldema and Michael Philippsen. 2012. Parallel Memory Defragmentation on a GPU. In *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC '12)*. ACM, New York, NY, USA, 38–47. https://doi.org/10.1145/2247684.2247693

Acknowledgments

We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Titan Xp GPU used for this research.