**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.

**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 techniques (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].
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 eliminate 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 \( \frac{1}{n+1} \) 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 \( R\left\lfloor \frac{r}{n+1} - 1 \right\rfloor \) are source blocks. Given a source block \( R[rid] \), its corresponding target blocks are:

\[
\left\{ R[rid + i \cdot B] \mid i \in 1...n \right\}
\]

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].
References


Acknowledgments

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