A Device Memory Pool Implementation for Omega_h Applications with Kokkos

Matthew McCall   Cameron W. Smith

September 13, 2024

1 Introduction

A memory pool is a technique for managing memory allocation in a computer program. It consists of a pre-allocated block of memory from which the program can request and release memory from the pool as needed, without invoking the system's memory allocator. This can improve the performance, reliability and portability of the program. Some of the benefits of memory pools are:

- They reduce memory fragmentation, which can cause inefficient use of memory and slow down the program.
- They reduce the overhead of system memory allocation and deallocation, which can consume a significant amount of CPU time and introduce latency.
- They allow the programmer to control the size and layout of the memory blocks, which can optimize the memory access patterns and cache efficiency.

Preliminary work done using the CUDA library for unstructured mesh adaptation in Omega_h on NVIDIA GPUs shows a significant performance increase when using a memory pool as opposed to traditional memory management strategies. However, CUDA is a proprietary library developed by NVIDIA for NVIDIA devices. The use of vendor-specific libraries such as CUDA for GPU computing can limit the portability and flexibility of applications. As such, this research aims to achieve comparable performance gains on AMD and Intel devices using the cross-platform library, Kokkos, to implement the memory pool.

2 Background and Related Works

Omega_h [1] is a software library written in C++ that provides mesh adaptivity for tetrahedron and triangle meshes, with an emphasis on high-performance computing. It is designed to add adaptive capabilities to existing simulation software. Mesh adaptivity allows for the reduction of both discretization error and the number of degrees of freedom during a simulation, as well as enabling simulations with moving objects and changing geometries. Omega_h achieves this in a manner that is fast, memory-efficient, and portable across a variety of architectures.

There exists a memory pool that works well with the CUDA backend in Omega_h [2]. However, this implementation can only be used on NVIDIA devices. On the other hand, this implementation uses Kokkos [3], a cross-platform library, to obtain device memory. The design for this pool was inspired by Boost's Simple Segregated Storage [4], a fixed-size chunk memory-pool implementation targeting host-sided memory. A fixed-sized chunk design often allows for faster allocation, whereas variably-sized chunk designs, such as those found in Umpire [5], allow for more memory efficiency. An important distinction between Boost's Simple Segregated Storage and other similar host-bound implementations and ours is that the free-list is interweaved into the chunks themselves. While this reduces memory overhead, storing a free-list in device memory would require copying the free-list to the host, manipulating it, and then copying it back to the device each time an allocation or deallocation is performed. As such, this scheme for managing chunks was impractical. It was much more reasonable to store the free-list in host memory. However, now that we have separated the free-list from the underlying chunks pool, we realized it does not have to be a list, which can result in linear allocation time. Thus, we opted to use free-sets instead. Furthermore, since memory is only requested and returned in host-side code, the large parallelism of GPUs does not interfere with traditional set-searching algorithms. As such, we were able to adapt an existing strategy, known to work well for managing host memory, to device memory where it brings significant performance improvements.

3 Technical Details

In Omega_h, a `StaticKokkosPool` is a non-resizable pool of memory from which an allocation can be made. Upon instantiation of each `StaticKokkosPool`, we call `kokkos_malloc` to get a contiguous block of memory. When we destroy the `StaticKokkosPool`, we call `kokkos_free` to release the memory back to the system. The memory pool is divided into an array of contiguous fixed-size 1 kiB chunks. Each chunk is ordered and indexed by their position.

3.1 Allocation

Instead of using a free-list, which may result in linear allocation time, we use a free-mulitset, `freeSetBySize`, which results in logarithmic allocation time. When an allocation n bytes is requested, we search through `freeSetBySize` to find the smallest free region that can accommodate the number of requested chunks. The number of requested chunks is calculated by $r = \lceil n \div chunkSizeInBytes \rceil$. In the case of a new or empty `StaticKokkosPools`, `freeSetBySize` would contain only one free region representing the entire pool. Figure 1 shows how `freeSetBySize` relates to the memory in the pool. A free region is defined as a set of contiguous free blocks between allocated regions or ends of the pool. If the found free region has more chunks than the number of requested chunks, then we split the region and only allocate the chunks requested. The remaining region remains in the free list.

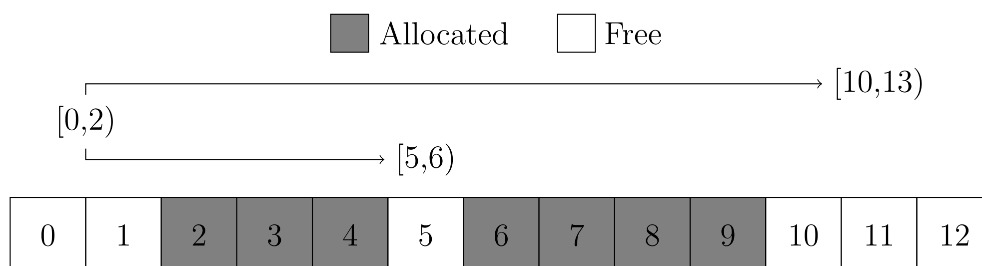

Figure 1: A fragmented pool with `freeSetBySize` showing how free regions are organized in a tree structure. Index pairs are sorted top to bottom by the size of the free region they represent in decreasing order.

3.2 Deallocation and Defragmentation

In addition to `freeSetBySize`, `freeSetByIndex` is used to achieve defragmentation in logarithmic time. As the name suggests, index pairs stored `freeSetByIndex` in sorted in numerical order of the first index itself as depicted in Figure 2. It is not possible for two index pairs to "overlap" or share the same start or end index.

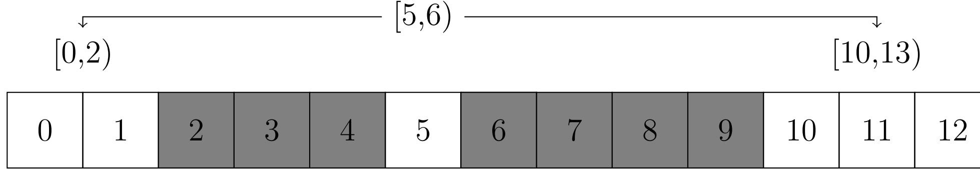

Figure 2: A fragmented pool with `freeSetByIndex` showing how free regions are organized in a tree structure. Index pairs are sorted left to right by the indices of the free region they represent in increasing order.

When memory is returned to the pool, the region is temporarily inserted into the tree as shown in Figure 3.

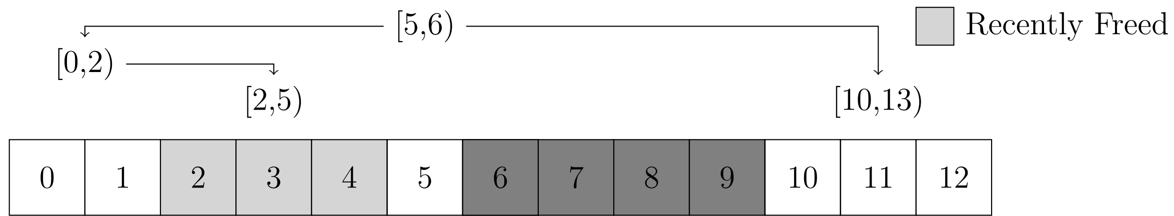

Figure 3: A fragmented pool with `freeSetByIndex` showing with a recently returned region.

We then check for free regions adjacent to the recently returned region, remove the two or three index pairs if any adjacent regions were found, and then insert one index pair encompassing the defragmented region. For example, in Figure 3 the allocation spanning blocks [2,5) is freed. This new free region is surrounded by pre-existing free regions [0,2) and [5,6). Thus we remove the three index pairs and replace them with [0,6). The result is visualized in Figure 4.

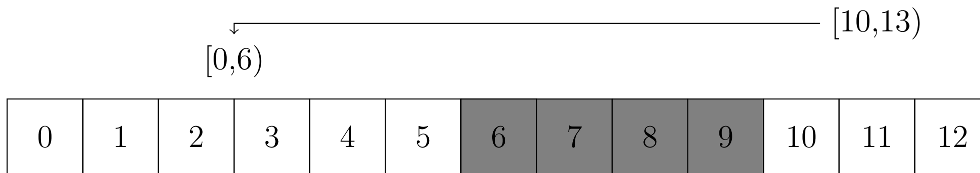

Figure 4: A less fragmented pool with `freeSetByIndex` shown after defragmentation

3.3 Resizing

In the event that we cannot find a suitable free region large enough to satisfy the allocation requested, we make a new `StaticKokkosPool`. The `KokkosPool` class performs this for us by maintaining a list of `StaticKokkosPool`s. In Omega_h, `KokkosPool` is a singleton object that is lazy initialized upon the first allocation request. Upon instantiation, it creates one `StaticKokkosPool` and adds the pool to its list. Thus, when you allocate through `KokkosPool`, it begins at the first `StaticKokkosPool` in the list and tries to allocate from it. If the allocation fails, it moves on to the next `StaticKokkosPool` and tries to allocated from that. We repeat this for every `StaticKokkosPool` in the list until we achieve a successful allocation. If we have reached the end of the list and were not able to find a large enough free region in any of the `StaticKokkosPool`s, we allocate a new `StaticKokkosPool` with the size of $\max(r, \mathit{mostChunks} * g)$, where r is the number of chunks requested as defined above, $\mathit{mostChunks}$ is the number of chunks in the largest `StaticKokkosPool`, and g is the growth factor. In Omega_h, the default growth factor is two. This strategy ensures an allocation will be successful until the machine runs out of physical memory. Ideally, the total size of the initial `StaticKokkosPool` and size of the chunks should be fine tuned to avoid this process.

4 Testing and Performance

4.1 Testing

The implementation discussed here resides in Omega_h, a mesh adaptivity library. The library was benchmarked with and without the memory pool against the FUN3D delta wing case from the Unstructured Grid Adaptive Working Group adaptation benchmarks [6] on the Oak Ridge Leadership Computing Facility's Frontier. First, we built Kokkos, libMeshb, and Omega_h as described in the Omega_h wiki [7]. By default, the memory pool is disabled and must be explicitly enabled with the `--osh-pool` command line flag. In addition, we built and used the Kokkos Tools MemoryEvents tool as per the Kokkos Tools wiki [8]. We ran the 500k case, which comprised of 581,196 tetrahedrons before adaptation and 5,283,878 tetrahedrons after adaptation, with the pool disabled and with the MemoryEvents tool enabled to get a sense of the allocation patterns for this particular set of benchmarks. We found that it uses at most 670 MiB during the lifetime of the benchmark. Thus we chose an initial pool size of 700 MiB. In addition, we found that allocation requests were greater than 1 kiB more often than not, thus we decided to set the fixed-chunk size

to 1 kiB.

4.2 Performance

After we determined these parameters for the pool, we ran all subsequent benchmarks without the Kokkos Tools enabled. In the 50k case without pooling, adapting 581,196 tetrahedrons to 533,937 tetrahedrons took 0.89 seconds total. With the pool engaged, this time was reduced to 0.49 seconds, a 45% time reduction. In the 500k case without pooling, adapting 581,196 tetrahedrons to 5,283,878 tetrahedrons5 took 18.28 seconds total while, with the pool enabled, adaptation only took 11.57 seconds on average, a 37% time reduction. We then reduced the initial size of the pool to 100 KiB. This is significantly less than what we determined the benchmarks need and should require the pool to resize more often during the runtime. We found that difference in time reduction brought about by the additional resizing behavior of the pool was less than 1%.

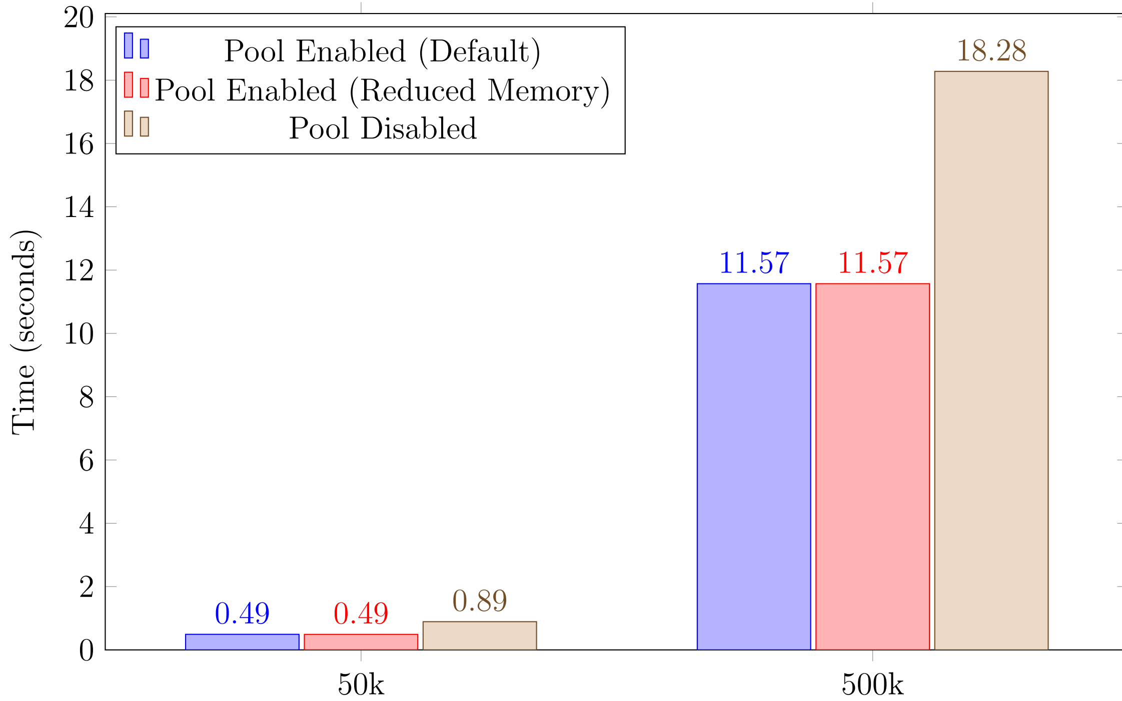

Figure 5: A bar plot showing the performance improvements in the delta wing case brought by the implementation of a memory pool. Less time is better.

5 Closing Remarks and Future Work

This implementation only splits and coalesces free regions, not individual chunks. Thus, it is possible for small allocations, or allocations that don't roughly align with multiples of the chunk size to result in excessive memory waste as the vast majority of an end chunk may not be used, especially if chunks are large. Alternative implementations such as Umpire on the other hand split and coalesce individual chunks, resulting in lesser memory usage. Regardless, this implementation manages to bring substantial performance improvements in a cross-platform manner. Thus far, this implementation has been tested on AMD and NVIDIA devices. Future work aims to test this implementation on Intel devices as well. Furthermore, this implementation has only been tested on systems with traditional, separate discrete host and device memory. With the advent of modern unified memory architectures, we intend to test this implementation on newer systems such those that utilize NVIDIA's Grace-Hopper APU and AMD and Intel equivalents. Here, we determined that fixed-size chunk memory pools are just as suitable for use in device memory spaces as host memory spaces. We also determined that such a pool is suitable for use across different devices and vendors. Testing has also confirmed that pooling device memory significantly reduces the running time of an application by upwards of up to 45%. In conclusion, fixed-size chunk memory pools are a viable method of managing device memory that brings substantial performance improvements in a cross-platform manner.

References

[1] D. A. Ibanez, *Conformal mesh adaptation on heterogeneous supercomputers.* Rensselaer Polytechnic Institute, Troy, NY, 2016.

[2] *Omega_h GitHub source* [Online]. Available: https://github.com/SCOREC/omega_h/blob/reducedThrust/src/Omega_h_malloc.cpp.

[3] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood *et al.*, "Kokkos 3: Programming model extensions for the exascale era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2022.

[4] S. Cleary and P. A. Bristow. *Boost.Pool* [Online]. Available: https://www.boost.org/doc/libs/1_83_0/libs/pool/doc/html/boost_pool/pool/pooling.html#boost_p

[5] D. Beckingsale, M. McFadden, J. Dahm, R. Pankajakshan, and R. Hornung, "Umpire: Application-focused management and coordination of complex hierarchical memory," May 2020.

[6] *Parallel Adapt Results GitHub repository* [Online]. Available: https://github.com/UGAWG/parallel-adapt-results.

[7] *Omega_h GitHub wiki* [Online]. Available: https://github.com/SCOREC/omega_h/wiki/Build-and-Run-on-OLCF-Frontier/562e93cb6fbcf2910ba45b39b2ec5a5051dd0ab6.

[8] *Kokkos Tools GitHub wiki* [Online]. Available: https://github.com/kokkos/kokkos-tools/wiki/MemoryEvents/eb3755d8b3fcec543d0a16a5ff0992f2cb505ce4.