

A Device Memory Pool Implementation for Omega_h Applications with Kokkos

Smith, Cameron McCall, Matthew

August 10, 2023

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 that is divided into smaller chunks of fixed size. 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 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.

2 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 pool 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 kB chunks. Each chunk is ordered and indexed by their position.

2.1 Allocation

Instead of using a free-list, which may result in linear allocation time, we use a free-multiset, `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. 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. It is worth noting that we only split regions of chunks and not the chunks themselves.

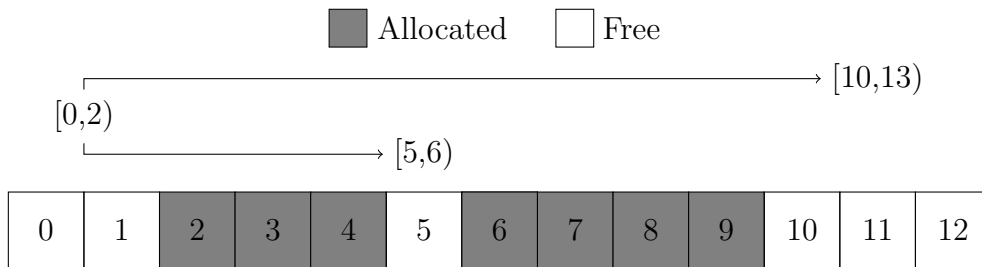


Figure 1: A fragmented pool with `freeSetBySize` showing how free regions are organized in a tree structure. Index pairs are sorted top to bottom by the size of the free region they represent in decreasing order.

2.2 Deallocation and Defragmentation

In addition to `freeSetBySize`, `freeSetByIndex` is used to achieve defragmentation in logarithmic time.

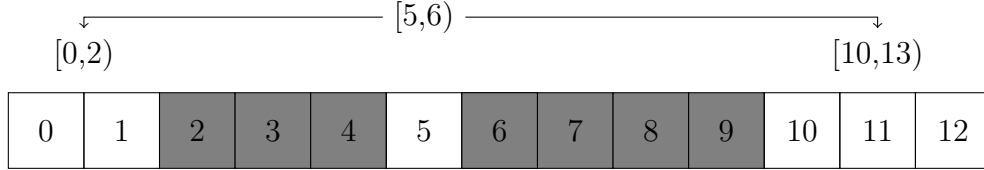


Figure 2: A fragmented pool with `freeSetByIndex` showing how free regions are organized in a tree structure. Index pairs are sorted left to right lexicographically by the indices of the free region they represent in increasing order.

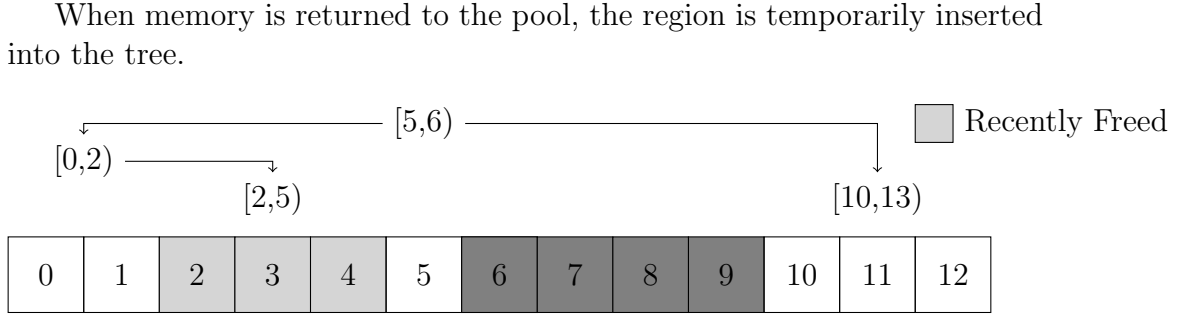


Figure 3: A fragmented pool with `freeSetByIndex` showing with a recently returned region.

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.

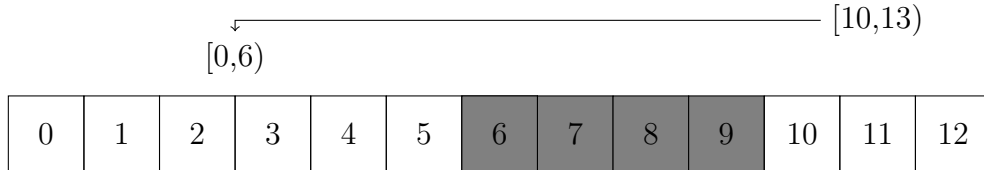


Figure 4: A less fragmented pool with `freeSetByIndex` shown after defragmentation

2.3 Resizing

Aforementioned, a `StaticKokkosPool` is a non-resizable pool. In the event that we cannot find 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 `StaticKokkosPools`. In `Omega_h`, `KokkosPool` is a singleton object that is lazy initialized upon the first allocation request. When the `KokkosPool` is instantiated, it creates one `StaticKokkosPool` with 700 MB of memory and adds the pool to its list. Thus, when you allocate through `KokkosPool`, it iterates over its list of `StaticKokkosPools` and tries to allocate from all until it achieves a successful allocation. If we have reached the end of the list, we allocate a new `StaticKokkosPool` with the size of $\max(r, mostChunks * g)$, where r is the number of chunks requested as defined above, $mostChunks$ is the number of chunks in the largest `StaticKokkosPool`, and g is the growth factor. In `Omega_h`, the default growth factor is 2. This strategy guarantees an allocation will be successful up until the machine runs out of physical memory.

3 Performance Results

4 Related Works