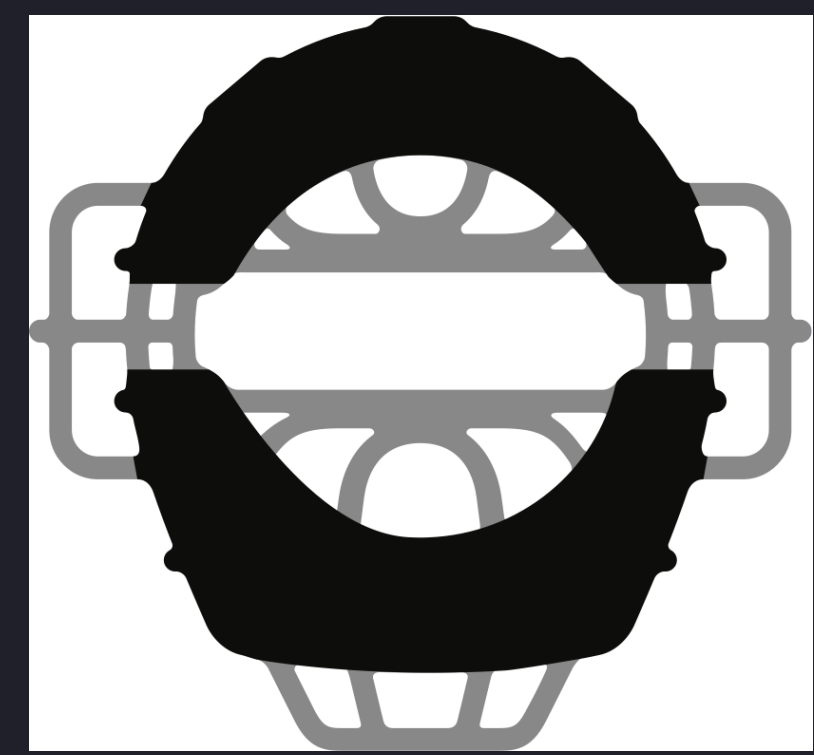




Umpire: Portable Memory Management for High-Performance Computing Applications

Kristi Belcher, David Beckingsale

Lawrence Livermore National Laboratory, Livermore, CA, USA



Modern HPC systems present application developers with complex memory hierarchies that include multiple types of memory with varying access patterns, capacities, and performance characteristics. To address these challenges, Umpire was developed at Lawrence Livermore National Laboratory (LLNL) as an open-source C++ memory management library for modern HPC platforms. This poster introduces Umpire's design principles, outlines Umpire's primary performance advantages and limitations, and fosters discussion about practical strategies for managing complex memory hierarchies in modern C++ HPC applications.

Background

- Umpire is a C++ open-source library that provides provides a **unified, portable memory management API** for modern HPC platforms
- Umpire abstracts away vendor-specific details to expose essential memory characteristics
- Umpire enables developers to write applications that **efficiently utilize heterogeneous memory systems while remaining portable** across different hardware platforms

Key Technical Features

Memory Resources represent the different types of memory available on a system
Allocators are the primary interface through which applications interact with Umpire

```
auto& rm = umpire::ResourceManager::getInstance();  
auto allocator = rm.getAllocator("HOST");  
double* data = static_cast<double*>(allocator.allocate(100 * sizeof(double)));  
// do calculations  
allocator.deallocate(data);
```

1



Allocation Strategies enable sophisticated memory allocation algorithms

```
auto& rm = umpire::ResourceManager::getInstance();  
auto pool = rm.makeAllocator<umpire::strategy::QuickPool>("pool",  
rm.getAllocator("DEVICE"));  
auto th_safe_pool = rm.makeAllocator<umpire::strategy::ThreadSafeAllocator>("pool2",  
pool);  
double* data = static_cast<double*>(th_safe_pool.allocate(100 * sizeof(double)));  
// do calculations  
th_safe_pool.deallocate(data);
```

1



Operations provide a unified interface for memory manipulation tasks

```
auto& rm = umpire::ResourceManager::getInstance();  
auto cpu_allocator = rm.getAllocator("HOST");  
auto gpu_allocator = rm.getAllocator("DEVICE");  
double* src = static_cast<double*>(cpu_allocator.allocate(100 * sizeof(double)));  
double* dest = static_cast<double*>(gpu_allocator.allocate(100 * sizeof(double)));  
// Copy data from HOST to DEVICE  
rm.copy(dest, src);
```

1



Umpire also includes multi-language support with **C and Fortran bindings** and **debugging tools** to assist developers (i.e. Replay, Caliper Service, etc.)

Performance and Discussion

- Umpire's memory pool strategies allow for a **less expensive way** to allocate all needed memory for HPC applications, compared to device specific APIs
- Umpire provides DynamicPoolList, MixedPool, QuickPool, and other memory pool allocation strategies which are optimized for different allocation patterns
- Figure 1 shows that for random allocations between 4 and 256K bytes, **QuickPool is most performant** by many orders of magnitude
- The comparison in Figure 2 between QuickPool allocations and deallocations and native hipMalloc or hipFree calls shows the **effectiveness of memory pools**

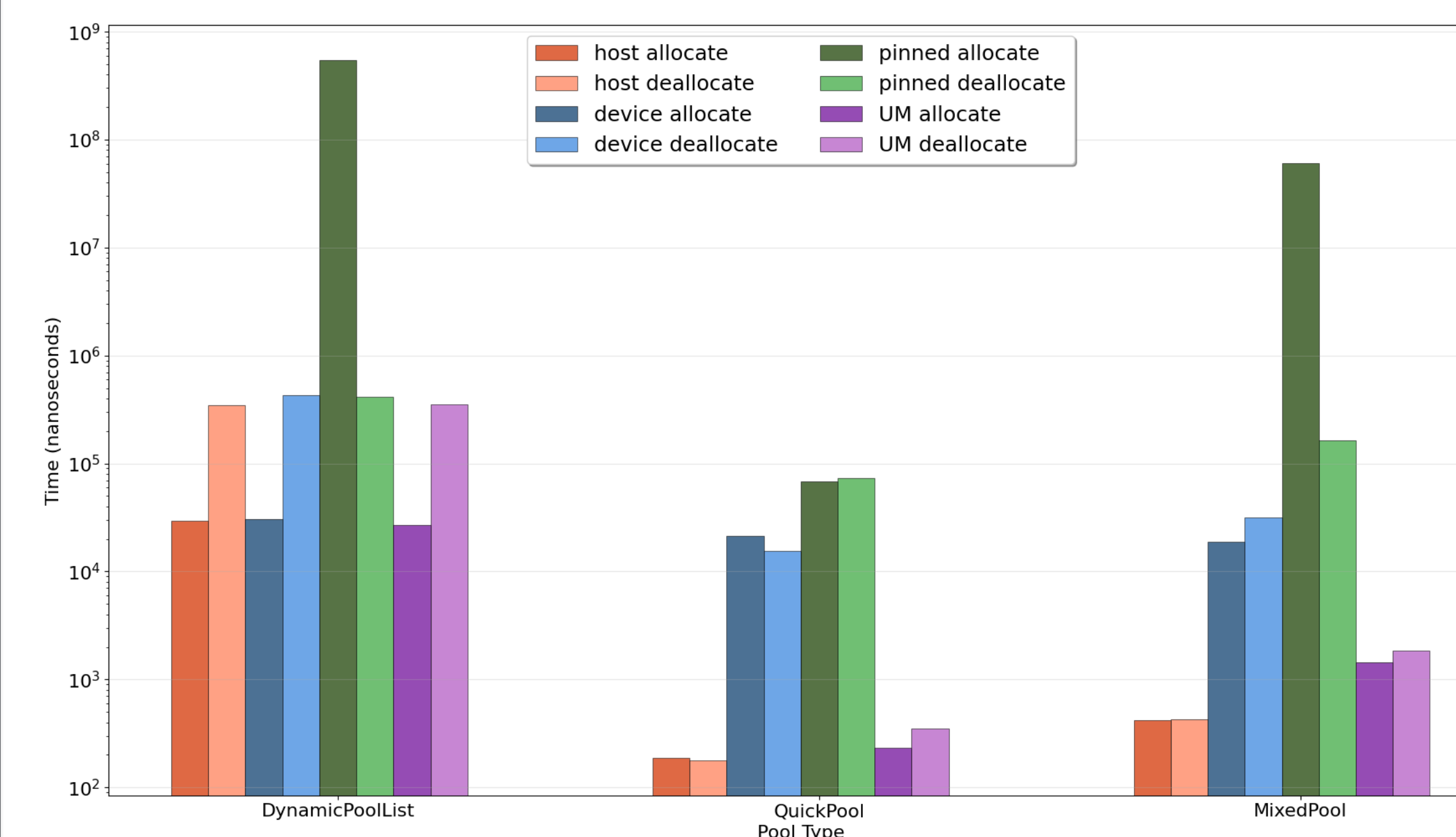


Figure 1: A comparison of QuickPool, DynamicPoolList, and MixedPool with Device, Pinned, and Unified Memory (UM) shows that QuickPool is more performant for a randomized allocation pattern. Measurements from LLNL's RZAdams, which includes 4 AMD Instinct MI300A accelerators per node.

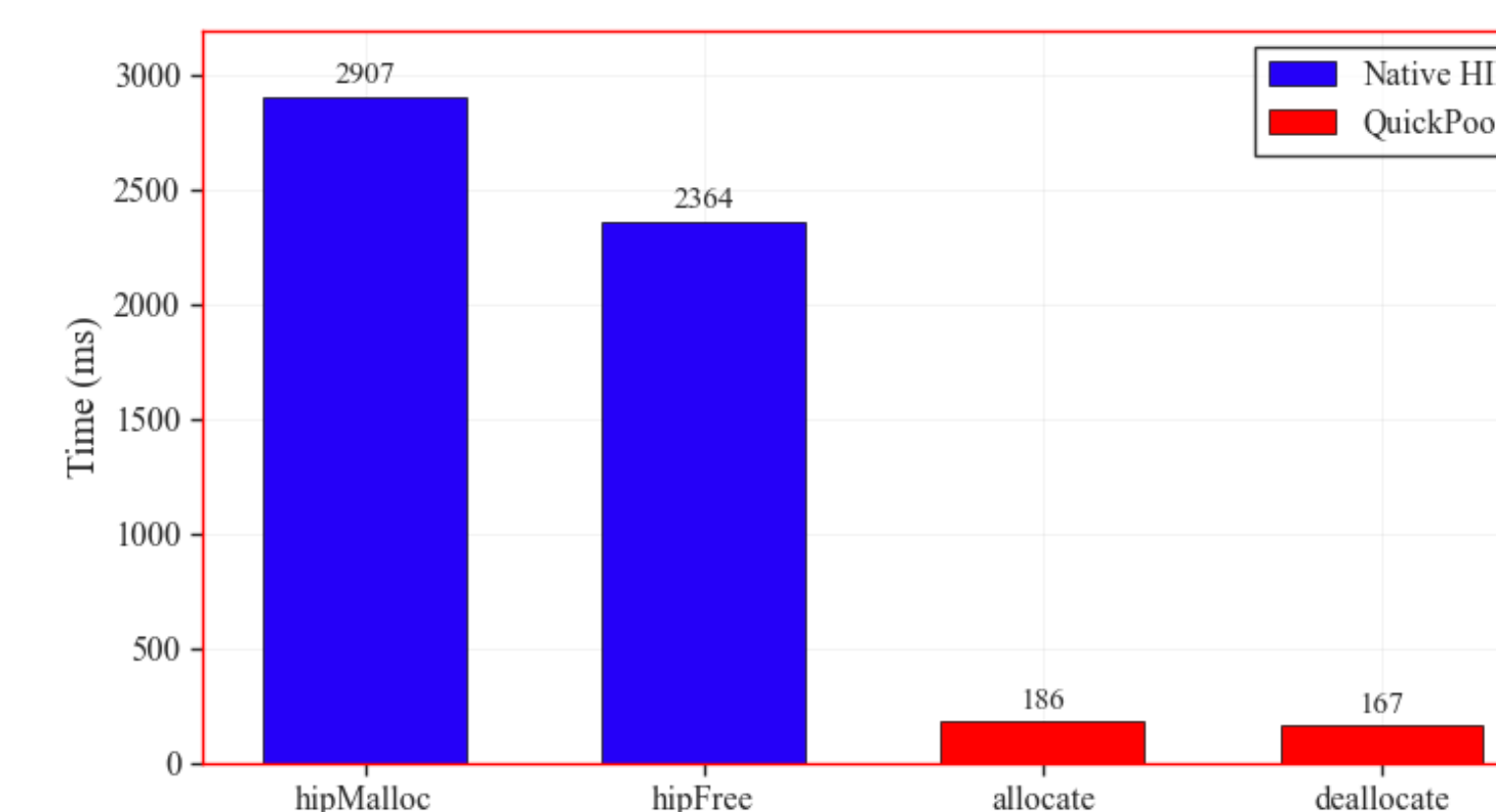
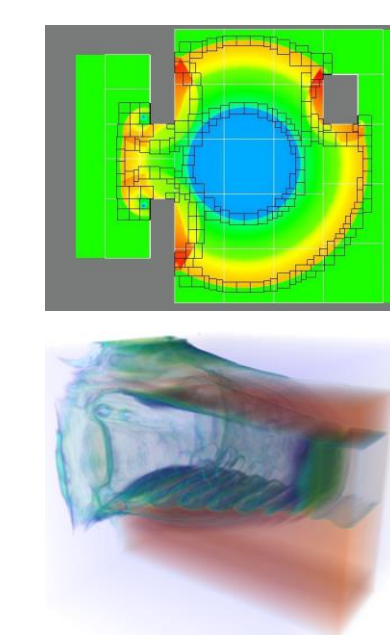


Figure 2: A comparison of QuickPool vs. native HIP shows that **QuickPool is over 15x more performant than native HIP**.

Umpire's memory pools are used extensively in LLNL Production Codes such as:

- SAMRAI, a C++ library for adaptive mesh refinement (AMR) applications
- MARBL, a multi-physics code for simulating high energy density physics



Next, the Umpire team plans to make use of C++17 features to optimize the new ResourceAwarePool

- Update the PendingList to be a PendingMap instead with std::unordered_multimap, std::optional, and std::reference_wrapper

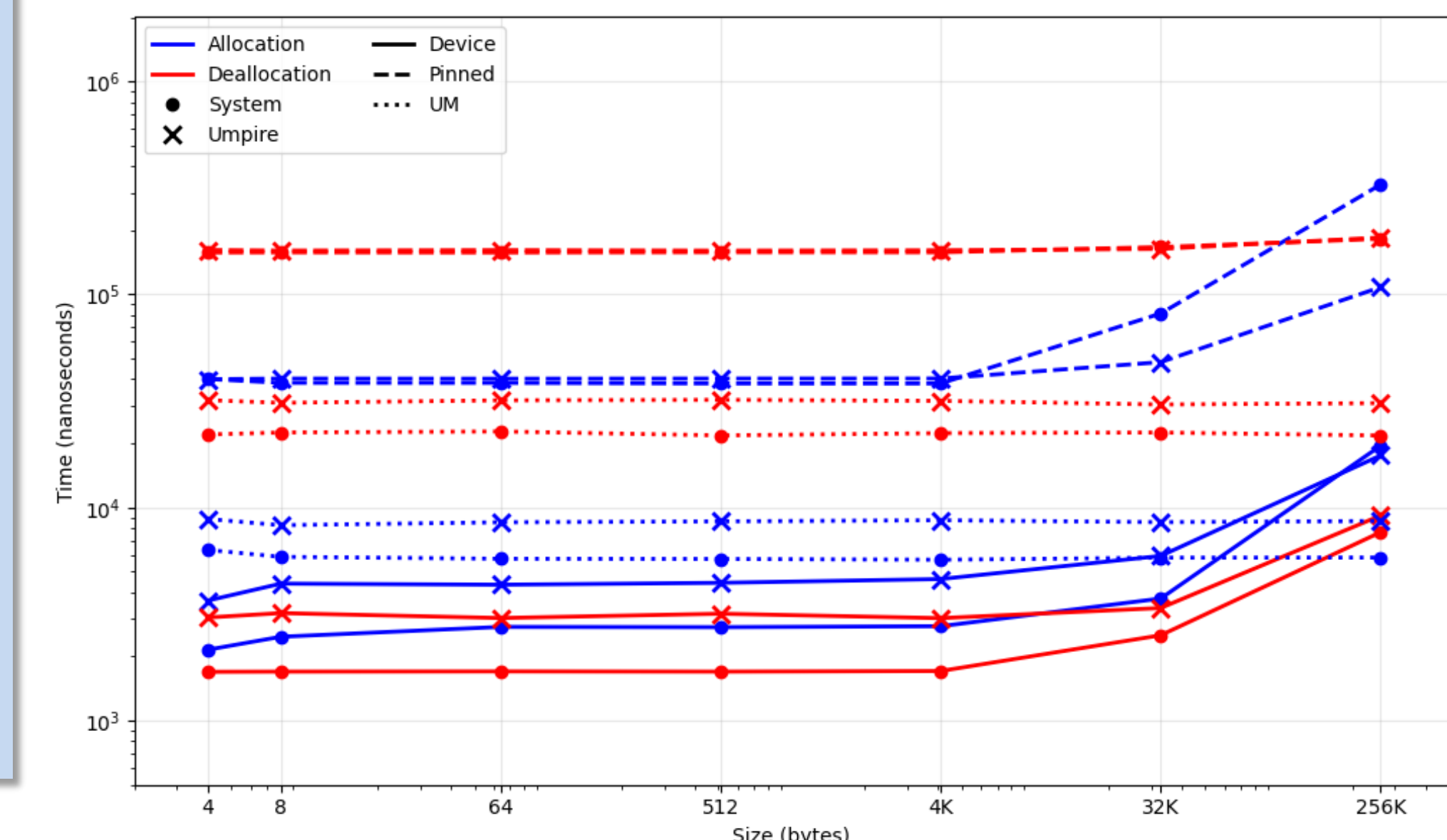
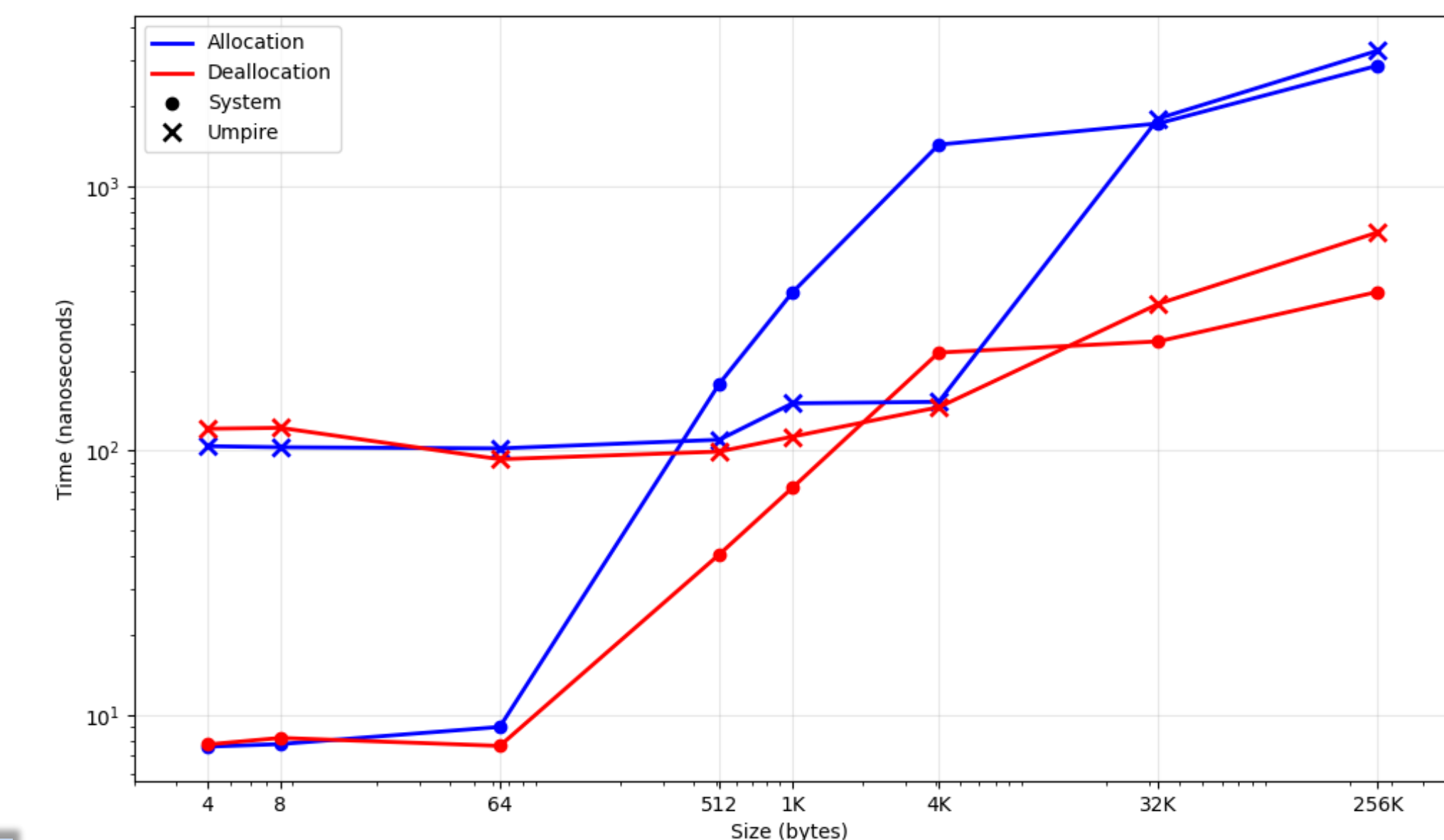
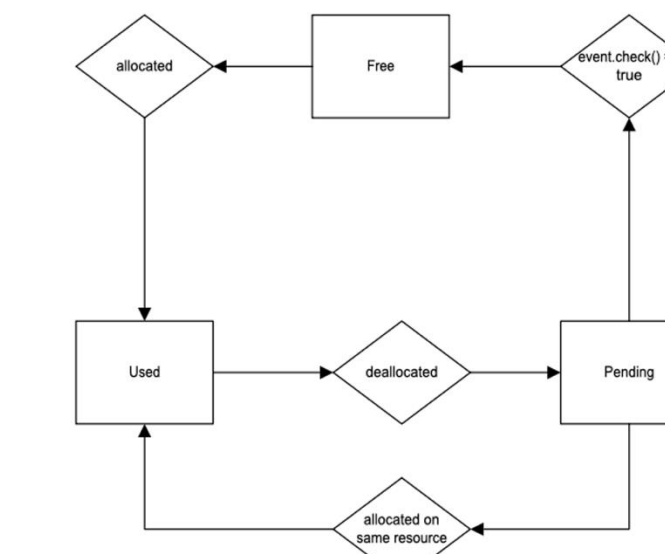


Figure 3: A comparison of the system library calls and Umpire resources for Host memory (top) and Device memory (bottom) for sizes ranging from 4 to 256K bytes.

- Fig. 3 shows there is a slowdown of 1.58 to 1.78 for the Umpire device allocator compared to the native hipMalloc calls, but as the **allocation size grows larger, this overhead becomes less and less**
- For the 256K allocation size, Umpire outperforms hipMalloc and gives a 10% speedup. There is a similar trend for deallocations
- Additionally, CPU allocations in the top chart show minimal benefit due to about **290μs overhead from metadata collection** (which can be turned off)
- Again, the overhead becomes less significant as the allocation size grows, and Umpire starts to be more on-par with system allocations and deallocations

Additional upcoming Umpire plans:

- Optimize and profile our new GPU-accessible Inter-Process Communication (IPC) allocator
- Find users for our "Device Allocator", an allocator specifically meant to be used within a GPU kernel
- Update to C++20* and improve Fortran interface
- Revamp Umpire API to use more C++ concepts