Using Umpire's Coalescing Heuristics to Improve Memory Performance

Kristi Belcher, Marty McFadden, David Beckingsale
Lawrence Livermore National Laboratory
{belcher6, mcfadden8, beckingsale1}@llnl.gov

***Abstract*—Because GPU memory is a highly constrained resource, Umpire, a data and memory management API, was created at Lawrence Livermore National Laboratory (LLNL). Umpire provides memory pools which enable less expensive ways to allocate very large amounts of memory in HPC environments. However, developers must ensure that the blocks of memory contained within the pool are properly managed. Otherwise, the pool can perform poorly or even run out of memory prematurely. The Umpire team conducted several experiments using a high explosive chemistry application to study different ways to manage blocks of memory within Umpire memory pools. We found that with the right heuristic, we can see memory savings up to 64% which, for this application, translated to an 8-16x speedup. Thus, memory pools must be effectively managed in order to be both performant and memory efficient.**

***Index Terms*—memory management, memory pools, coalescing, high performance computing**

I. Introduction and Background

Lawrence Livermore National Laboratory (LLNL) develops many large scale, multi-physics algorithms which are constantly being ported to the GPU and other HPC hardware [1]. Although the CPU parallel version of these algorithms may have worked well under the memory constraints of a CPU, when ported to the GPU, those memory resources must be reexamined [2]. In addition to limited memory resources, technology specific APIs can force application developers to tie their memory management code to a single type of device. Thus, LLNL created Umpire [3], an open source library that provides a unified, portable memory management API to accommodate modern HPC platforms with complex combinations of memory resources. Many of the large, multi-physics applications at LLNL use Umpire for memory management.

To address the limited memory resources on HPC devices, Umpire provides memory pools which allow developers to allocate all needed memory at once instead of making multiple, smaller memory allocations which can become quite expensive, particularly with device specific APIs. However, memory pool performance can suffer if the pool can't utilize unused (i.e. free) blocks of memory as it accommodates new allocations. Coalescing is Umpire's solution to managing memory blocks properly within a pool. A coalesce function will deallocate any unused blocks, replacing them with one large block to handle new allocations. A coalescing heuristic is used to figure out when to call the coalesce function. If the coalescing heuristic is not tuned properly to the application, it will not call the coalescing function often enough and the pool will continue to grow. A successful coalescing heuristic allows the memory pool to readjust for new allocations throughout the duration of the program, thereby avoiding the potential to grow too large and prematurely run out of memory. The success of the coalesce function is impacted by the size and duration of memory allocations, as well as the order in which they occur. A mixture of either large and small allocation sizes or temporary and permanent allocation durations increases the chances for unused memory blocks to occur within the pool. An effective coalescing heuristic will be able to handle these more complex allocation patterns by triggering the coalesce function as appropriate so the memory pool can readjust.

Because of the limited scope of this study, we focus on just one multi-physics application for our experiments. The application in this study includes a mixture of temporary and permanent memory allocations within the same pool. Additionally, it has a very high number of small, temporary allocations. It is because of this complex allocation pattern that the coalescing heuristic we initially used, Percent-Releasable, caused the pool to grow too large, too quickly and inevitably crash. Thus, we conducted this study to examine other coalescing heuristics and how they impact memory pool performance. This extended abstract describes our experiments to test two types of coalescing heuristics along with two kinds of heuristic tunings. We describe the resulting performance, showing how each experiment impacted the overall performance and memory overhead of the application. Finally, we provide conclusions and future work of our study.

II. Experiments and Methodology

Our experiments were done using Replay, a debugging tool provided by Umpire that tracks memory usage and collects memory pool statistics. Using Replay, we can see how the memory pool is being used within an application by tracking the current size (*amount of memory that the user has allocated from the pool*), high-watermark size (*peak value of the current size*), and actual size (*the size of the pool that the user is doing allocations from*) of the memory pool. With this information, we can track the memory usage throughout the duration of the application and how the performance varies across different coalescing heuristics.

We conducted several experiments[1] to test how different co-

[1]The byte percentage and block value ranges used in our Percent-Releasable and Blocks-Releasable experiments were selected based upon preliminary tests.

alescing heuristics affect Umpire's memory pool performance including:

- **Bytes-Based *(Percent-Releasable)* heuristic**: once a certain percentage of bytes is releasable, coalesce the pool. Test for 50%, 75%, 90%, and 100% of bytes.
- **Blocks-Based *(Blocks-Releasable)* heuristic**: once a certain number of blocks is releasable, coalesce the pool. Test for 2, 3, 5, 7, and 12 blocks.
- **High-WaterMark *(HWM)* heuristic tuning**: when a coalesce is needed, coalesce to the high-watermark instead of the actual size of the pool.
- **Coalesce-Before-Growing *(CBG)* heuristic tuning**: check to see if a coalesce is needed as the pool grows instead of after deallocation.

III. Results

As shown in Fig. 1, only one of the Percent-Releasable experiments ran to completion without timing out. Moreover, that Percent-Releasable experiment was worst at 73% memory overhead because it is not able to call the coalesce function often enough, as it requires 100% of bytes to be releasable (i.e. the pool must be empty). Because of the complex allocation pattern from this application, this criteria is not satisfied often enough and the coalesce function is therefore not triggered as often as it should be. So, we instead focused on trying to determine the best Blocks-Releasable heuristic for this application. We discovered a consistent trend toward a *trade-off*: as the number of blocks required to coalesce increases from 2 to 12, the number of total coalesce calls decreases. At the same time, the peak memory usage (and therefore memory overhead) increases (Fig. 2). Too many coalesce calls will slow down the overall runtime too much because a coalesce involves freeing and reallocating memory. At the same time, not enough coalesce calls will cause the memory pool to run out of memory too quickly. Thus, a decision must be made to find the *sweet spot* while tuning a heuristic.

The sweet spot is a range of values that is not too expensive in terms of both the number of coalescing calls and total memory used. The sweet spot for our experiments is indicated with a box in Fig. 2. Although some of the Blocks-Releasable experiments with the High-WaterMark tuning resulted in some of the lowest memory overhead, these experiments also required 30 or more coalesce calls. Therefore, we determined that the sweet spot range contained both relatively few coalescing calls and lower peak memory usage. The sweet spot range can vary depending on the application, as well as developer priorities. Since most of the points contained within our given sweet spot range are from the Blocks-Releasable heuristic with the Coalesce-Before-Grow tuning, those experiments represented the best results for this application. Overall, our Blocks-Releasable experiments resulted in a 36-64% reduction in total memory usage. After the application code team applied these changes to their memory pools, they saw a 8-16x speedup.

Heuristic Name	Coalesce Calls	Memory Overhead
Percent(50)	*Timed Out*	*Timed Out*
Percent(75)	*Timed Out*	*Timed Out*
Percent(90)	*Timed Out*	*Timed Out*
Percent(100)	2	73%
Blocks(2)	44	8%
Blocks(3)	30	**6%**
Blocks(5)	18	15%
Blocks(7)	11	11%
Blocks(12)	**7**	17%
Blocks(2) HWM	47	7%
Blocks(3) HWM	34	**5%**
Blocks(5) HWM	17	8%
Blocks(7) HWM	12	11%
Blocks(12) HWM	**9**	18%
Blocks(2) CBG	10	**10%**
Blocks(3) CBG	8	15%
Blocks(5) CBG	7	19%
Blocks(7) CBG	5	13%
Blocks(12) CBG	**4**	26%

Fig. 1. Number of coalesce calls compared to the *percentage of memory overhead* shows a trade-off for a variety of Blocks-Releasable heuristic functions. Most Percent-Releasable heuristics resulted in time outs.

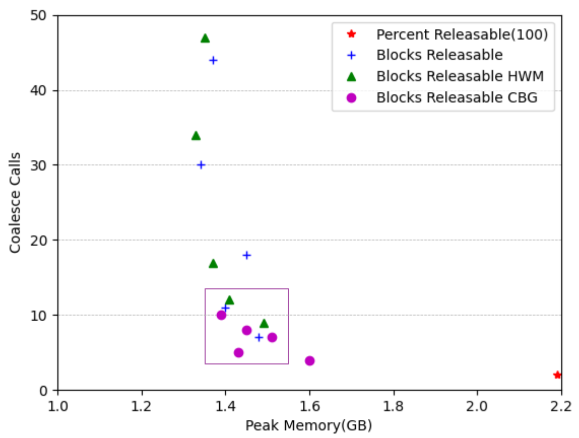

Fig. 2. The overall trends with the number of coalesce calls plotted against the *peak memory usage* remains consistent throughout all Blocks-Releasable experiments. The box represents a sweet spot range.

IV. Conclusions and Future Work

After applying the Blocks-Releasable heuristic to our chemistry application, we saw a 36-64% reduction in total memory usage and a 8-16x speedup. Our study demonstrates how effectively managing blocks of memory within a memory pool with Umpire's coalescing heuristics can dramatically improve overall memory pool performance. Since complex allocation patterns make unused memory blocks more likely to occur, separating different types of allocations into distinct memory pools can also help improve memory pool performance. In fact, the Umpire team is currently studying ways to automate the separation of temporary and permanent allocations into separate pools using Machine Learning based decision models. Future work will also include further study into how coalescing heuristics impact other applications.

V. Acknowledgements

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.
LLNL-ABS-838468

References

[1] B. Barney, "Using LC's Siera Systems," Livermore Computing Documentation, Lawrence Livermore National Laboratory.

[2] I. Arefyeva, D. Broneske, G. Campero, M. Pinnecke, and G. Saake. "Memory management strategies in CPU/GPU database systems: A survey." In International Conference: Beyond Databases, Architectures and Structures, pp. 128-142. Springer, Cham, 2018.

[3] D. A. Beckingsale, M. J. McFadden, J. P. S. Dahm, R. Pankajakshan and R. D. Hornung, "Umpire: Application-focused management and coordination of complex hierarchical memory," in IBM Journal of Research and Development, vol. 64, no. 3/4, pp. 00:1-00:10, 1 May-July 2020, doi: 10.1147/JRD.2019.2954403.