LC-MEMENTO: A Memory Model for Accelerated Architectures

Kiran Ranganath[1], Jesun Firoz[2] Joshua Suetterlein[2], Joseph Manzano[2], Andres Marquez[2], Mark Raugas[2], and Daniel Wong[1]

[1] University of California Riverside, Riverside, California, 92521
{krang006, danwong}@ucr.edu
[2] Pacific Northwest National Laboratory, Richland, Washington, 99352
{jesun.firoz, joshua.suetterlein, joseph.manzano, andres.marquez, mark.raugas}@pnnl.gov

Abstract. With the advent of heterogeneous architectures, in particular, with the ubiquity of multi-GPU systems, it is becoming increasingly important to manage device memory efficiently in order to reap the benefits of the additional core count. To date, such responsibility mainly falls on the programmer where device-to-host data communication (and vice versa), if not done properly, may incur costly memory transfer operations and synchronization. The problem may be compounded by additional requirement to maintain system-wide memory consistency that may involve expensive synchronization overhead. In this paper, we present **L**ocation **C**onsistency **Me**mory **M**odel for **En**hanced **T**ransfer **O**perations (LC-MEMENTO). This framework considers incorporating runtime techniques for multi-GPU memory management to support relaxed synchronization semantics and memory transfer operations automatically. Specifically, we implement a relaxed form of a memory consistency model based on the Location Consistency (LC) in an Asynchronous Many-Task Runtime (ARTS) and demonstrate that, this memory model enables additional optimization opportunities for the three representative applications encompassing different computational patterns (scientific computation, graphs, data streaming, etc.).

1 Introduction

In recent years, the widespread adoption and integration of accelerators as part of the High-Performance Computing (HPC) ecosystems have presented several unique challenges to the runtimes and software stacks with regards to effectively managing massive heterogeneous computational resources and interacting with the hierarchical memory subsystems. Among these accelerators, General Purpose Graphic Processing Units (GPGPUs) are the most prevalent ones that are either currently in use or in the roadmaps of the current and future generation of the most powerful supercomputers [29].This trend has sparked a new wave of research in software stacks and runtime libraries to fully utilize the current heterogeneity of these designs, e.g., [27]. Some of the most important challenges in this area include the coordination among computational and memory resources, efficient communication between parallel components, and productive use of additional resources e.g., high performance networks.

Traditionally, the burden of managing all these challenges have fallen on the application programmer or the user due to the lack of proper runtime support. Hence, manually managing these resources have made the development of novel, high-performance implementations of interesting applications a very onerous process. However, one of the most promising programming execution paradigms to effectively support programmers to exploit massive parallelism available on recent accelerator-based systems are the Asynchronous Many-Task (AMT) runtimes. These frameworks are designed to utilize the underlying computational substrates by dividing the computation into well-defined tasks which can be executed asynchronously. Nonetheless, support for effective runtime scheduling for memory management and fine-grained synchronization can be improved.

To support more flexible frameworks that can more effectively map resources without programmer intervention, we consider a previously proposed memory consistency model, Location Consistency [10], and re-purpose this consistency model for multi-GPU systems. Thus, this paper introduces LC-MEMENTO: a new runtime extension to an AMT runtime named ARTS [25] to support accelerated architectures. These extensions include a runtime-managed transparent scheduler for multi-GPUs systems designed to exploit new multi-GPU designs; and a novel memory model extension, designed to increase the productivity of programmers and the performance of current data analytics workflows. The contributions of this paper are as follows: (1) A novel GPU cache-based memory model based on the Location Consistency Model [10] with extensions for polymorphic[3] (relaxed) synchronization/collective operations; (2) a scheduler framework that automatically exploits multi-accelerator environments for task scheduling and memory management without user intervention; and (3) a performance analysis of three important data analytics kernels (STREAM and Random Access benchmark from HPC Challenge benchmark suite [17], as well as the well-known Breadth-first Search (BFS) from graph analytics) that demonstrate the advantages of our framework. In addition, we also compare our approach with current technologies such as Unified Virtual Memory (UVM).

This paper is organized as follows. Section 2 explains the necessary concepts that permeates this paper such memory consistency and the basics of ARTS. Section 3 introduces and explains the LC-MEMENTO implementation under ARTS. Section 4 demonstrate the effectiveness of our techniques via experimental results, obtained on a Summit supercomputer Node and on a DGX-1 Volta GPU based nodes. In Section 5, we showcase select state-of-the-art research that might complement or enhance our line of research. Finally, Section 6 discusses our conclusions and future directions.

2 Background

To fully explain the design and impact of LC-MEMENTO, this section briefly discusses the concepts behind memory models and location consistency, as well

[3] Their ancillary functionality can be set and reset across different program's phases

as introduces the overall design of the Abstract Runtime System (ARTS), a runtime in which the concepts of LC-MEMENTO are implemented.

2.1 Memory Consistency Models

As computer architectures are progressively becoming more complex, the exposition of the hierarchical and hybrid memory subsystem requires giving consideration to unique challenges. The introduction of reorder buffers, bypass memory queues, and the availability of multi-core and many-core processors with different cache levels introduced many new complex issues related to the ordering and the visibility of memory operations in these systems. The challenges associated with the questions of when operations on memory locations can be reordered or when two distinct parallel actors will observe an updated value on a fixed memory address, became the crux of many optimization-related research in architecture and system software (e.g., [16]). The rules of the ordering and visibility of these memory operations is roughly collected under a **memory consistency model**[3]. Under this paradigm, the strength of a model is inversely dictated by the allowed cardinality of the set of possible orderings. In other words, the more possible chains of valid memory operators that a model allows, the weaker it is. Weaker models can allow for the same variable to have different values across executions if not correctly synchronized. The most common and one of the strongest models is called **sequential consistency**. Under this model, the ordering of the memory operations and their visibility are *as if* it follows the program order across all parallel actors in the system. This may imply a highly synchronized underlying network (like a bus) for which this type of coordination is possible. Such stringent model may prevent certain re-ordering operations that may be beneficial from the perspective of the hardware. For example, a store and a load can be reordered if they do not pose any dependency or the values of one store that overwrites the other may not need to be updated outside the local memory buffer or cache line. For this purpose, several weaker models have been proposed such as CDAG[13] for distributed systems and lazy release consistency[18]. Memory models are also very useful to understand the memory behavior in parallel computation. A notable one is *Location Consistency (LC).*

Location Consistency Location Consistency [10] is, arguably, the weakest "practical"[4] memory model. This model weakens the visibility requirements of the classical execution models by allowing a single variable to have multiple values at the same time in different parts of the system, typically by residing in a local piece of memory (cache for example), unless collapsed **explicitly** by the user. The aggregated values of a memory location is called its Partial Ordered Set or POMSET. The model has three operations: a program read, a program write and the synchronized acquire and release pair. For normal program reads and writes, the location will keep the written values and return a value from it when

[4] A practical memory model is one that can be used by application developers to write non-chaotic codes since all of its non-determinism can be contained by special operators.

a read is requested. The ordering of these operations is not respected when no acquire and release pairs are presented. In addition, two consecutive reads might return different values from the POMSET in the same execution. For example, if T_0 follows the instruction sequence: $R_1 = X; X = 1$ and T_1 executes the following instruction stream: $R_2 = X; X = 2$, the result $R_1 = 2$ and $R_2 = 1$ is not allowed under other memory models (both sequential and coherence memory models would disallowed this result), but it is considered a legal case under LC since the writes can be reordered (see [15] for a further discussion about the original LC properties and proofs). The semantics of the acquire/release pairs operations follow the classical view of entry consistency[5]. In this model, the acquire and release pair establishes a region of code for which its constituting instructions must be contained inside their boundaries in terms of execution and completion. However, instructions originating before and after the pair can be reordered, started or completed at any point of the execution (even inside the acquire/release pair code block). LC enhances this concept by ensuring that the POMSET is collapsed to a single value after the release operation so that any consequent reads will return that value and only that value. The enhanced semantics of the acquire and release pairs plus the relaxed constrains for the other memory operations allow reordering optimizations to take place in both the software stack and in the hardware. Such freedom in reordering allows fine-grained asynchronous frameworks to take advantage of latency hiding techniques as long as the application can support some degree of error such as in certain data and graph workflows.

2.2 The Abstract Runtime System: ARTS

ARTS is designed as a best of breed runtime from our previous experience with OCR[9], GMT[7], and AM++[30] leveraging key features from each runtime. ARTS supports parallelism via asynchronous tasking, active messages, and lightweight multithreading. To enable synchronization, ARTS provides a global address space which can be used to access segments of memory called data blocks. In order to support data analytic workloads, ARTS provides methods for termination detection (i.e. quiescence), asynchronous memory operations (e.g. get/put), and remote atomics. As a dataflow inspired runtime, applications are expressed as a Directed Acyclic Graph (DAG) of fine-grain tasks, with each vertex representing a task and an edge between two vertices depicting dependencies between these two tasks. The DAG is constructed dynamically; it is both created and evaluated at runtime. Data is passed between tasks via typed segmented memory chunks a.k.a data blocks. Besides basic typing information, the data blocks under ARTS have no inherent semantic meaning and follow an enhanced version of Entry Consistency plus DAG consistency called CDAG. Under this model, acquire and release pairs are defined at the beginning and end of the computational tasks and the system ordering is imposed by the DAG structure of the computational graph with the extra caveat that possible concurrent writes must be ordered by the DAG explicitly. Dependencies between tasks are expressed using a signaling API. This API is the vehicle for the movement of data blocks around the underlying cluster. Both tasks and data blocks are identified with a globally unique ID (GUID) facilitating access throughout the cluster via the runtime. To evaluate the DAG, the runtime employs five layers, our tasking/scheduling layer, an out-of-order engine that allows reservations of resources dynamically even when dependencies are not ready, the global address space/memory model presented above, several network layers, and an introspection framework. Each layer is informed by the ARTS abstract machine model which is used to describe the underlying computational substrate. While the ARTS execution model was originally developed for traditional CPU architectures, the following sections discuss its extension to a multi-GPU substrate.

2.3 NVIDIA CUDA Programming and Execution Environment

CUDA has rapidly become the de-facto language of GPGPU accelerators which speaks to the fact that NVIDIA GPUs are the dominant force behind the accelerator based supercomputers (for the current generation). This notoriety has amassed a vast of collection of highly optimized libraries for different domains, from scientific, data processing, and machine learning. Moreover, the multi-GPU environment has become a staple on the high performance field. Such situations presents challenges as how to effectively use all available computational resources and how to coordinate data movement across them (challenges familiar to distributed computing). The current available solutions include the concepts of kernel streams [24] that allow the scheduling of individual kernels to separate GPUs concurrently as well as ordering them in case of dependencies (i.e. kernels within a stream are executed in order). In the case of memory, current NVIDIA hardware provide the concept of Unified Memory (UVM) which allows several GPUs and the host to share a memory space which is controlled at a page level. These solutions, although very helpful, have severe shortcomings. The stream based approach would leave the entire scheduling and managing process to the application users creating a very onerous and error-prone development process. The UVM-based approach has the disadvantage that accesses and reuse happen at page boundaries which might be too big for certain applications. Taking all these concepts together, the reordering capabilities of weak memory models, the flexibility of asynchronous runtime systems and the building blocks provided by the current state of the art accelerators, we can build a framework to explore enhanced scheduler, and memory model extensions that will benefit accelerators running on the most powerful computers available.

3 LC-MEMENTO Design and Implementation

To create a productive and performance oriented programming framework for multi accelerator environments, we enhance the existing memory model with new semantics to combine synchronization and computation in a single operation (polymorphic collective / synchronization ops) such that the cost of synchronization can be amortized with these collective like operators. This memory model is implemented through the use of an accelerator cache which at the same time can exploit locality when available. Moreover, this is built upon a scheduler framework that distributes the work across multiple GPUs while providing policies to manage work and storage if the user requires. Each enhancement is explained below.

3.1 Asynchronous Runtime Scheduler for Accelerators

To indicate which tasks should execute on a GPU, we provide a specialized LC-MEMENTO task. This task also allows the user to enumerate the required data blocks via their GUIDs. The Data block GUIDs can be provided both at task creation or during the runtime prior to task execution. These GPU tasks are not scheduled until all task dependencies are met, and are not executed on a device until the required data blocks are present in the GPU's data block cache. In addition to user written GPU functions, we leverage the vast library of optimized domain specific libraries by providing concurrency abstractions to execute existing libraries easing LC-MEMENTO / ARTS GPU development.

When a GPU task's dependencies have been met, the task is scheduled based on user configurable predefined load-balancing schemes designed to explore the trade-off between parallelism and memory consumption. To run parallel kernels and to facilitate data movement between the host and GPUs, we leverage the parallel NVIDIA stream constructs. Prior to running a kernel, the appropriate data blocks are moved to the selected GPU (assuming that the data block is not already present) by the runtime. Further, the runtime maintains a directory of data block GUIDs on the host to manage the memory of a given GPU's cache. By tracking this information at the runtime level on the host, we are able to schedule work according to heuristics to promote parallelism, locality, or memory efficiency. Once a kernel is completed, the appropriate tasks are signaled and dirty (modified) data blocks are moved back to the host. Copies of valid data blocks remain on the device, until a garbage collection process is executed by an idle host thread in order to promote temporal locality.

There are several benefits of the runtime maintaining both memory and scheduling. First, work can be load balanced across GPUs transparent to the user since the correctness is being guaranteed by the application DAG. The second benefit comes from the runtime managing memory. By specifying the data block requirements, the user can ensure there is sufficient memory to run a given kernel prior to execution working to eliminate out-of-memory errors during runtime. In addition, the runtime ensures the transfer of appropriate data block to the device prior to the execution of the kernels. The final benefit is the virtualization of memory at a sub-page level which can have significant impact on irregular applications.

3.2 Memory Models for Accelerators

One of the key limitations to location consistency is the mechanism used to reduce the POMSET. Particularly, how a user can indicate which value the POMSET should be reduced to. For some applications, the most recently written or random value is acceptable. While this is reasonable from a hardware implementation perspective, we find this less useful to the application programmer.

Instead we propose performing reduction operations on the POMSET. Appropriate operations should be associative and preferably idempotent[5]. Since we are implementing our solution in software, operations are not required to be atomic. Examples of these operations include addition, subtraction, multiplication, min, max, AND, OR, and XOR.

Rather than implementing LC for each level of access granularity (memory location, cache line, data structure, etc.), we maintain consistency at the ARTS's memory abstraction: the data block. Under LC-MEMENTO, data blocks are placed inside a software cache on each accelerator, with the CPU maintaining a global directory of their GUIDs similar to a directory based distributed memory cache [14]. Besides each data block being typed as either read or write, we extend the data block with an LC-MEMENTO based type which we denote as LC data block or LCDAB. On a cache miss when accessing a LCDAB, the host transfers a copy to the accelerator. In response, the accelerator is free to access the LCDAB without any synchronization with other accelerators or other CPU's data block (LCDAB or otherwise) copies. When a LCDAB is evicted, the host pulls the accelerator's copy and perform a merge operation (which can be user-defined) between the host and devices copies. If the user requires a consistent state across all devices, they can use a synchronization API call to ensure that the global view of the data is consistent. This operation usually translates to merge operations to be performed on all of the data block copies across the system using both the ARTS CDAG protocol for internode consistency and LC-MEMENTO for intranode one. Currently, LC-MEMENTO has two different merge techniques. The first one is a reduction which copies the LCDABs from each accelerators to the host and performs the merge on the host. While this implementation provides semantically correct results, it is less efficient as the reduction is performed serially on the host. The second reduction mode implements a dynamic reduction tree across accelerators using their optimized collective operator libraries. As the tree is created, we identify which accelerators have a valid copy of the LCDAB. Next, we form a reduction tree based on the host/accelerator topology, ensuring that a high bandwidth is maintained across all participating accelerators. The data block is then sent to its peer accelerator, where it performs the merge operation in a kernel based on the size of the LCDAB (e.g., leveraging SIMT parallelism for NVIDIA GPUs). Once all the accelerator connected through the links have reduced their data, the results are transferred and merged with the host's copy.

In some cases, the operations used to perform are not idempotent (i.e. add, XOR, etc.). In these cases, we provide an API to initialize zero copy LCDABs on the accelerators. We have found this sufficient for implementing benchmarks such as HPC Challenge's Random Access.

[5] A concept in computer science and mathematics in which operators can be applied multiple times without changing the results / state of the computation after the first application

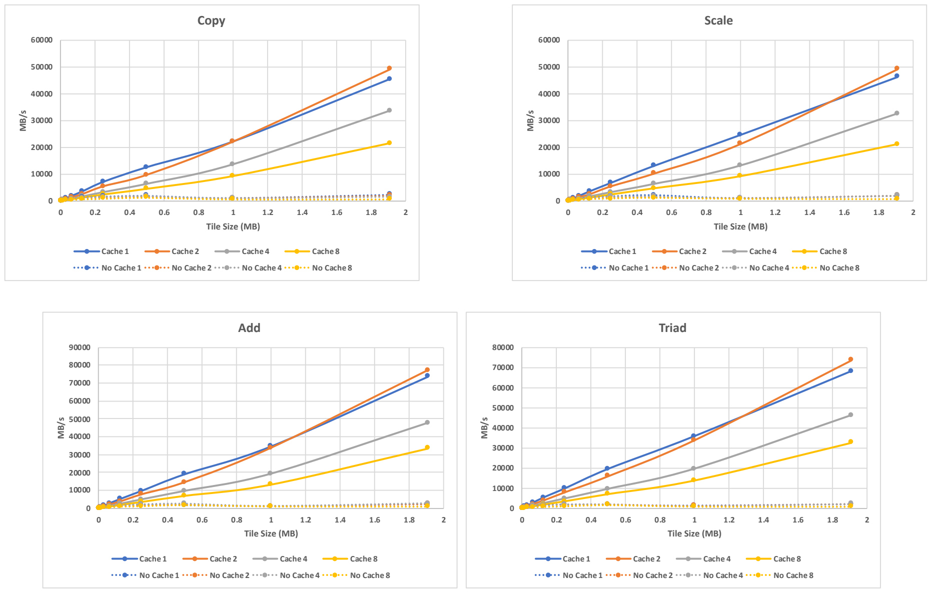


Fig. 1: STREAM benchmark results, demonstrating the usefulness of caching and larger tile size.

4 Evaluation

We evaluate our proposed framework using three of the cornerstone kernels in data and graph analytics. We conduct our experiments on a DGX-1 Volta-based platform. We compile our code with CUDA 9 and GCC 8 to evaluate our framework. The DGX-1 system has 8 Volta GPUs connected to a x86 host with 16 CPU cores. The DGX-1 GPUs are connected in two grids of four and then connected across by their closest GPUs.

4.1 STREAM Benchmark

The STREAM benchmark is a part of the HPC Challenge benchmark suite [17]. It is designed to test the sustainable bandwidth of a memory subsystem focusing on the performance of its cache. The benchmark consists of four kernels, copy, scale, add, and triad. Each kernel operates on two to three operands in a loop with a stride of one. The kernels themselves show little reuse within a kernel, but data can be cached across kernel invocations. The operation intensity for copy/scale and add/triad is 16 bytes to 1 op and 24 bytes to two ops respectively. For LC-MEMENTO, we tile the input arrays so they can be distributed by our scheduler. This benchmark is written for a single node, and we tested it on the DGX-1 system with a problem size of 15 MBs. Figure 1 presents our results across multiple GPUs as we scale the tile size. In the subsequent figures, “X” in the legend “Cache X” or “UMA X” denotes the number of GPU(s) used.

By tiling the data into manageable data blocks (to avoid out-of-memory error), by transferring the tiled data transparently to the device with the help of

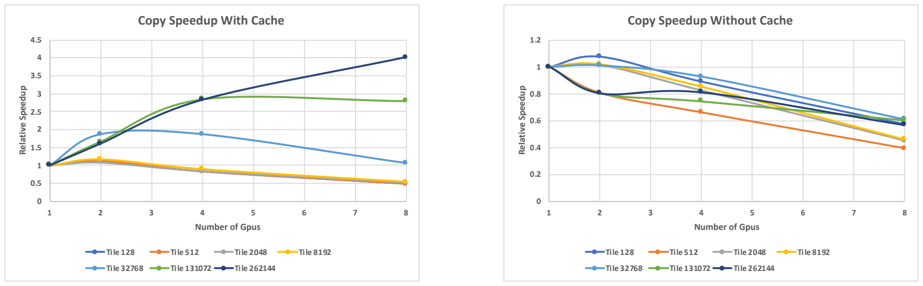


Fig. 2: Copy with and without cache

the runtime scheduler as the kernels become ready to execute, and by enabling reuse of data across different kernels to take advantage of spatial locality, LC-MEMENTO can achieve better bandwidth utilization, compared to the case with no available caching (Figure 1). Another insight is that small tile sizes do not scale well since the cost of launching multiple kernels (tasks) cannot be efficiently amortized. Thus, the best performance is achieved with the largest tile sizes. Finally, it is worthwhile to point out the difference in operational intensity and performance between the different kernels. In these cases, both add and triad outperform copy and scale in their performance.

Figures 2 and 3 present the relative speedup of the copy and add benchmarks. In the case of larger tile sizes, we can achieve a peak speed up of 4x running on 8 GPUs. This is because at this size, we achieve a higher bandwidth when moving data onto the GPU.

In Figure 4, we scale the tile size proportionally with the problem size to find the maximum speedup achievable on a single node using 8 GPUs. We see that the performance levels stabilize around 5x, as we saturate the bandwidth available to the GPUs. These benchmarks illustrate the importance of data movement. To scale, we require tasks to either have high reuse, or are large enough to saturate the bandwidth to the GPUs. If the reuse is low, we will always be bound by data movement.

Before delving into more kernel analysis, we should revisit the synergies and discords of LC-MEMENTO and the Unified Virtual Memory (UVM) technology since the STREAM benchmark specializes in memory behavior.

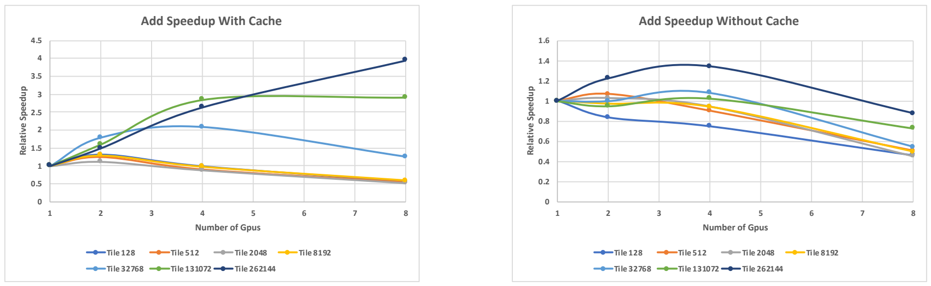


Fig. 3: Add with and without cache

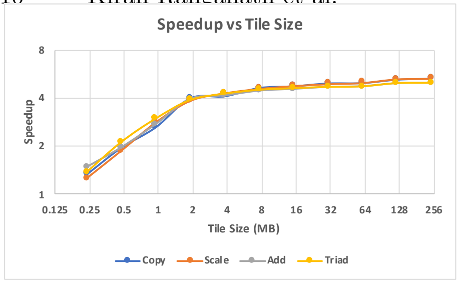


Fig. 4: Speedup while varying tile size

Figure 5 shows the STREAM benchmark with implementation on both LC-MEMENTO and using Unified Memory technologies. This implementation uses the first version of the UVM implemented in the NVIDIA Volta architectures. We scale the problem size and tile size proportionally in each of the eight GPUs. This figure shows that the maximum speedup achievable in these experiments is related to the max bandwidth and kernel invocation rate. This translates to around of 4x improvement over UMA at the largest tile size.

Figure 6 showcase that the copy and add kernels as we increase the tile size with more GPUs. Although both frameworks decreases as bigger tile sizes and more GPUs are introduces, we observe that LC-MEMENTO scales up to the size of one MB tile while UVM does not scale in any situation. This seems to be a side effect of congestion and saturation of bandwidth as we increase the GPUs. UVM can overcome LC-MEMENTO in the largest cases when using only one GPU due to its hardware support. However, LC-MEMENTO helps with concurrency in the other cases.

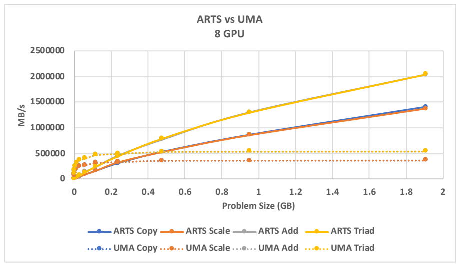


Fig. 5: STREAM ARTS vs UVM

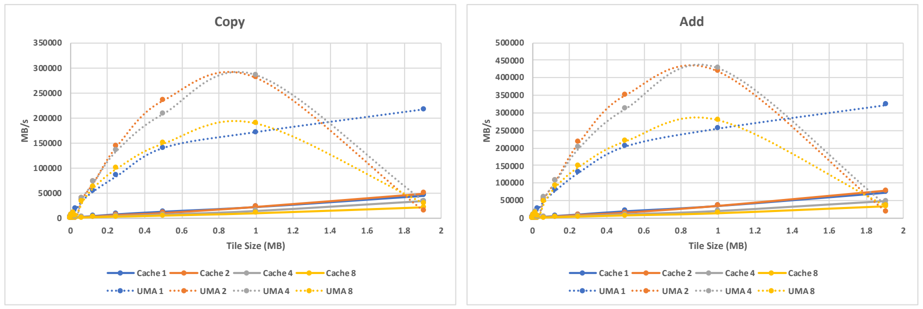


Fig. 6: LC-MEMENTO versus UVM as GPU and Tile size increases.

4.2 Random Access Benchmark

The Random Access benchmark is also part of the HPC Challenge benchmark suite[17]. This benchmark generates random updates to a large table that is designed to stress the memory subsystem. The benchmark metric is defined as Giga Updates per Seconds (GUPS). During each table access, a bit-wise XOR operation is performed to value with a random number. Afterwards, the table can be check for consistency in case that weak synchronization is used. We implement this benchmark to first generate the 1024 updates per SIMT thread in the system. Next, we pass this update frontier to a task with a tile of the table and perform all the updates that correspond to that tile. At the end, we synchronize the results using the LC extensions. The benchmark states that a process should only look ahead by 1024 updates (designed with MPI ranks in mind). We have relaxed this constraint to 1024 per thread since this better utilizes the GPU while still maintaining the benchmark's objective. We run this benchmark on a single node of the DGX-1 system using 16 host threads and scaling the number of GPUs from one to eight. We use a table size of 1.25 GB and divide our tiles evenly across the GPUs. We perform 2684354560 updates. As a baseline we provide a Unified Memory version for comparison.

From Figure 7, we observe that LC-MEMENTO's performance achieve a relative speedup of 2.8. While we are not suffering the effects of synchronization, we still observe the network bottleneck of data movement and kernel invocation. Without work to amortize this cost, scaling the kernel's performance is difficult. The Unified Memory baseline exhibits good scalability for one to five GPUs. We believe that at six GPUs the cost of on-demand paging (at a size of 4K) becomes too onerous and grinds its performance to a halt. In the case of using above six GPUs, LC-MEMENTO maintains its performance while UVM is overwhelmed.

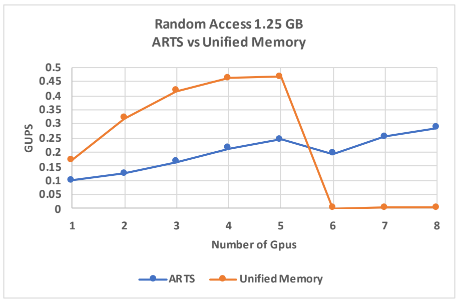
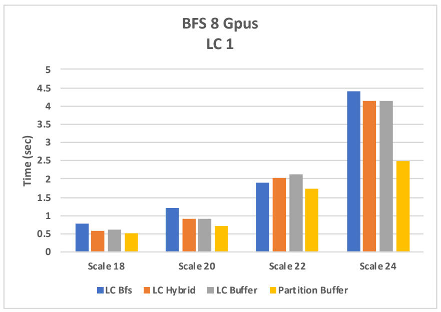


Fig. 7: Random Access benchmark on ARTS vs using Unified Virtual Memory.

4.3 Breadth-first Search



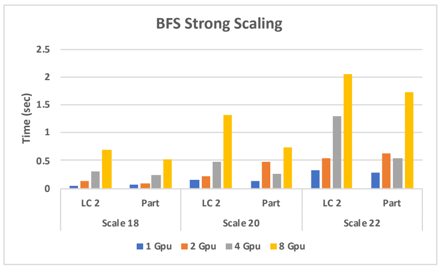


Fig. 8: Experimental results with the BFS kernel. On the left, we report the execution time of different variants of the BFS kernel on 8 GPUs. On the right, we report strong scaling results. Here, Scale X denotes a graph input with 2^X vertices, generated with the RMAT synthetic graph generator from the Graph500 benchmark.

Breadth-first search is one of the most important and common graph kernels that is widely used to evaluate the performance of HPC systems for irregular workloads. We implemented the k-level asynchronous Breadth-first search algorithm presented in [8]. In this algorithm, the level of asynchronous is controlled with the k parameter (i.e., when k is equal to one, the algorithm is fully synchronous; otherwise, the algorithm is fully asynchronous for k steps with redundant work for straggler computation) which is taken advantage by the LC-MEMENTO framework. This is thanks to LC-MEMENTO's flexibility with global reduction operations and their semantics. This basic variant based on the extended location consistency is denoted as *LC BFS* in Figure 8. However, notice that, graph algorithms are highly irregular and thus can generate uneven workload in each iteration. in particular, the frontier list (i.e. the next set of vertices to explore) assigned to the GPUs may not be big enough to require GPU execution. To address this issue, we either engage the CPU or a GPU to explore a frontier, based on the size of the frontier. This helps to avoid the data transfer and kernel launching overhead to the device. We denote this version of our algorithm as *LC Hybrid*. We also implemented two additional versions of BFS, where we allocate fixed-size buffers on the GPUs as a buffer pool, before the start of the algorithm. As the frontiers being generated can be of variable size, pre-allocating these buffers can help by avoiding the cost of re-allocation, in every iteration. We refer to these versions as *LC Buffer* and *Partition Buffer (partitioned buffer without LC).* Setting the value of k to 1 will also result in a level-synchronous or label-setting BFS algorithm, similar to Graph500[28] benchmark, redundant work is avoided and vertices are assigned at the final level.

We evaluate the performance of these algorithms running on a single node of the DGX-1 system using 16 host CPU threads and eight GPUs ad report the results in Figure 8. The input graphs are generated with Graph500's RMAT generator. Each vertex has a uniform degree of 16 edges. We generated graphs of scale 18, 20, 22, and 24. Here, scale X denotes a graph with 2^X vertices. We observe that "partition buffer" variant of the algorithm performs best, while the LC buffered and hybrid versions demonstrate similar performance. This is not surprising, since the input graphs have uniform degree distribution, hence the load is well-balanced and have little chance to benefiting from the extended LC memory model. However, we anticipate that, graphs with power-law degree distribution, in which a few high-degree vertices exist, will benefit from the extended LC memory model. Our strong scaling plot (Figure 8) shows that scalability suffers due to the overhead of small search frontiers per tile and no re-use.

We also vary the frequency of synchronization when applying LC-MEMENTO and report the results in Figure 9. As with many graph applications, we observe that the result tends to be data-dependent (size of the frontier being generated) and not particularly related to the size of the graphs. For example, for graphs with scales of 18 and 24, we see that synchronizing in every two iterations performs best. The plot to the right compares the best LC-MEMENTO synchronization with a partition approach. The partition approach performs better, since at each iteration, it performs no extra data movement compared to the LC variant which updates the data on the CPU by flushing its cache.

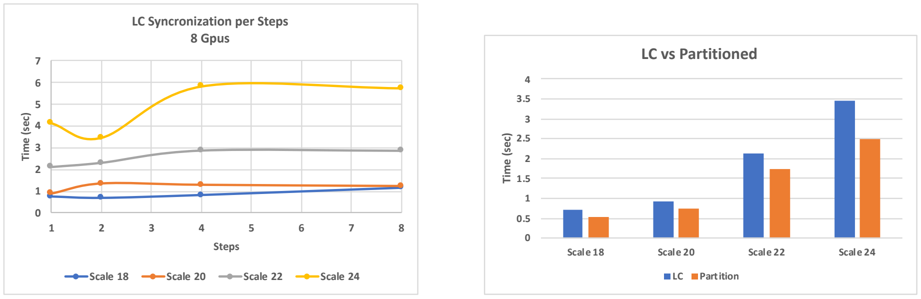


Fig. 9: LC-MEMENTO versus Partitioned BFS implementations

We anticipate, LC-MEMENTO could perform better with unbalanced graphs as utilization becomes an issue with fixed partitions.

5 Related Work

Memory consistency and cache coherency protocols. [22] proposed a *hardware-assisted* cache coherency protocol for multi-GPU systems. In contrast, our approach is based on runtime software, with possibilities for expansion and integration of other consistency techniques. Recent proposals for sequential consistency include [21, 11, 23]. Although these works demonstrate that relaxed consistency protocols are no better than sequential consistency, our work shows that there can be considerable benefit for relaxed-synchronization based memory consistency models for select data analytics applications.

Scheduling techniques. Recent works like MAPA[20], WOTIR[19], Gandiva[31] and Philly[12] explored placement optimizations of DNN workloads on multi-node multi-GPU environments. In Effisha [6], the authors proposed a preemptive scheduler for kernels to better support priority-based scheduling. Furthermore, Works like [26, 1, 2], explored hardware optimizations and require compiler-assisted code transformation and explicit insertion of runtime API calls in the original code. In contrast, our approach is transparent to runtime-assisted scheduling.

Programming Models. Groute [4] and Kokkos [27] proposed asynchronous multi-GPU programming models. However, memory consistency and ownership is required to be managed by the programmer in the aforementioned models. LC-MEMENTO, on the other hand, implements the location consistency memory model in the runtime and supports asynchronous execution in the runtime without programmer intervention.

6 Conclusions and Future Work

This paper presents the LC-MEMENTO extensions for an AMT runtime system. It allows the exploitation of current multi GPU designs with a transparent scheduler and a weaker memory models, instantiated as GPU cache with polymorphic synchronization / collective operators. These extensions were used to implement three kernels and showcased their performance gains. We found out that for certain kernels (with high locality) the framework could scale up to 4x while leaving all the resource allocation decisions to the internal runtime (i.e., completely transparent to the programmer). Moreover, we compare against UVM solutions and found that under certain conditions the cache / scheduler based solution could compete and even beat the hardware/driver based one.

These extensions are promising but they have a much larger optimization space that remains for improving efficiency of AMT runtime systems. For example, new enhancements to unlock new capabilities on the framework (e.g., lower kernel launch time can produce better execution profiles for the scheduler and faster throughput) can offer synergistic benefits with LC-MEMENTO and AMT runtime systems in general.

References

1. Abdolrashidi, A., et al.: Wireframe: Supporting data-dependent parallelism through dependency graph execution in gpus. In: Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture. pp. 600–611 (2017)
2. Abdolrashidi, A., et al.: Blockmaestro: Enabling programmer-transparent task-based execution in gpu systems. In: 2021 48th Annual IEEE/ACM International Symposium on Computer Architecture (ISCA). IEEE (2021)
3. Adve, S.V., Gharachorloo, K.: Shared memory consistency models: A tutorial. Computer **29**(12), 66–76 (Dec 1996)
4. Ben-Nun, T., et al.: Groute: An asynchronous multi-gpu programming model for irregular computations. ACM SIGPLAN Notices **52**(8), 235–248 (2017)
5. Bershad, B.N., Zekauskas, M.J.: Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Tech. rep. (1991)
6. Chen, G., et al.: Effisha: A software framework for enabling effficient preemptive scheduling of gpu. In: Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 3–16 (2017)
7. Droco, M., et al.: Global Memory and Threading (GMT), `https://github.com/pnnl/gmt`
8. Firoz, J.S., Zalewski, M., Kanewala, T., Lumsdaine, A.: Synchronization-avoiding graph algorithms. In: 2018 IEEE 25th International Conference on High Performance Computing (HiPC). pp. 52–61. IEEE (2018)
9. Modelado Foundation: Open Community Runtime. `https://xstackwiki.modelado.org/Open_Community_Runtime`
10. Gao, G.R., Sarkar, V.: Location consistency-a new memory model and cache consistency protocol. IEEE Transactions on Computers **49**(8), 798–813 (2000)
11. Hechtman, B.A., Sorin, D.J.: Exploring memory consistency for massively-threaded throughput-oriented processors. In: Proceedings of the 40th Annual International Symposium on Computer Architecture. pp. 201–212 (2013)
12. Jeon, M., et al.: Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In: 2019 USENIX Annual Technical Conference (USENIX ATC 19). pp. 947–960 (2019)
13. Landwehr, J., et al.: Designing scalable distributed memory models: A case study. In: Proceedings of the Computing Frontiers Conference. p. 174–182. CF'17, Association for Computing Machinery, New York, NY, USA (2017)
14. Lenoski, D., et al.: The Directory-based Cache Coherence Protocol for the DASH Multiprocessor. In: Proceedings of the 17th Annual International Symposium on Computer Architecture. pp. 148–159. ISCA '90, ACM, New York, NY, USA (1990)

15. Long, G., et al.: Location consistency model revisited: Problem, solution and prospects. In: 2008 Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies. pp. 91–98 (2008)
16. Lustig, D., et al.: A formal analysis of the nvidia ptx memory consistency model. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. p. 257–270. ASPLOS '19, Association for Computing Machinery, New York, NY, USA (2019)
17. Luszczek, P.R., et al.: The HPC Challenge (HPCC) Benchmark Suite. In: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing. p. 213–es. SC '06, Association for Computing Machinery, New York, NY, USA (2006)
18. Protiae, J., Milutinoviae, V.: Entry consistency versus lazy release consistency in dsm systems: analytical comparison and a new hybrid solution. In: Distributed Computing Systems, 1997., Proceedings of the Sixth IEEE Computer Society Workshop on Future Trends of. pp. 78–83 (Oct 1997)
19. Ranganath, K., et al.: Speeding up collective communications through inter-gpu re-routing. IEEE Computer Architecture Letters **18**(2), 128–131 (2019)
20. Ranganath, K., et al.: Mapa: Multi-accelerator pattern allocation policy for multi-tenant gpu servers. In: SC21: International Conference for High Performance Computing, Networking, Storage and Analysis. ACM (2021)
21. Ren, X., Lis, M.: Efficient sequential consistency in gpus via relativistic cache coherence. In: 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA). pp. 625–636. IEEE (2017)
22. Ren, X., Lustig, D., Bolotin, E., Jaleel, A., Villa, O., Nellans, D.: Hmg: Extending cache coherence protocols across modern hierarchical multi-gpu systems. In: 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA). pp. 582–595. IEEE (2020)
23. Singh, A., Aga, S., Narayanasamy, S.: Efficiently enforcing strong memory ordering in gpus. In: Proceedings of the 48th International Symposium on Microarchitecture. pp. 699–712 (2015)
24. Steve Rennich: Streams and Concurrency, `https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf`
25. Suetterlein, J., et al.: The Abstract Runtime System: ARTS, `https://github.com/pnnl/ARTS`
26. Tripathy, D., et al.: Localityguru: A ptx analyzer for extracting thread block-level locality in gpgpus. Proceedings of the 15th IEEE/ACM International Conference on Networking, Architecture, and Storage (2021 (To appear))
27. Trott, C.R., Edwards, H.C.: Kokkos: The c++ performance portability programming model. Tech. rep., Sandia National Lab.(SNL-NM), Albuquerque, NM (United States) (2017)
28. Ueno, K., Suzumura, T.: Highly Scalable Graph Search for the Graph500 Benchmark. In: Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing. p. 149–160. HPDC '12, Association for Computing Machinery, New York, NY, USA (2012)
29. Vergara, M., et al.: Scaling the Summit: Deploying the World's Fastest Supercomputer. In: Intl Workshop on OpenPOWER for HPC (IWOPH'19) (6 2019)
30. Willcock, J.J., et al.: AM++: A Generalized Active Message Framework. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques. p. 401–410. PACT '10, Association for Computing Machinery, New York, NY, USA (2010). https://doi.org/10.1145/1854273.1854323
31. Xiao, W., et al.: Gandiva: Introspective cluster scheduling for deep learning. In: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). pp. 595–610 (2018)