

Dissecting CPU-GPU Unified Physical Memory on AMD MI300A APUs

Jacob Wahlgren*, Gabin Schieffer*, Ruimin Shi*, Edgar A. León† Roger Pearce† Maya Gokhale†, Ivy Peng*

*KTH Royal Institute of Technology, Sweden

{jacobwah,gabins,ruimins,ivybopeng}@kth.se

†Lawrence Livermore National Laboratory, USA

{leon,pearce7,gokhale2}@llnl.gov

Abstract—Discrete GPUs are a cornerstone of HPC and data center systems, requiring management of separate CPU and GPU memory spaces. Unified Virtual Memory (UVM) has been proposed to ease the burden of memory management; however, at a high cost in performance. The recent introduction of AMD’s MI300A Accelerated Processing Units (APUs)—as deployed in the El Capitan supercomputer—enables HPC systems featuring integrated CPU and GPU with Unified Physical Memory (UPM) for the first time. This work presents the first comprehensive characterization of the UPM architecture on MI300A. We first analyze the UPM system properties, including memory latency, bandwidth, and coherence overhead. We then assess the efficiency of the system software in memory allocation, page fault handling, TLB management, and Infinity Cache utilization. We propose a set of porting strategies for transforming applications for the UPM architecture and evaluate six applications on the MI300A APU. Our results show that applications on UPM using the unified memory model can match or outperform those in the explicitly managed model—while reducing memory costs by up to 44%.

Index Terms—Memory System Characterization, GPU Memory Management, High Performance Computing (HPC)

I. INTRODUCTION

GPUs are a critical component of leadership clusters and high-performance computing (HPC) systems for their massive parallelism and high computing power. Today, most top supercomputers are equipped with discrete GPUs [34], with separate memory spaces for the CPU and the GPU. The memory speed lags behind as the computing speed continually improves, causing efficient data access to become increasingly critical for exploiting the full potential of emerging systems [18]. Applications that require frequent data movement between the CPU and GPU memories suffer from degraded performance and increased energy usage. In the past decade, extensive works have been proposed for optimizing memory access and reducing data movements between CPU and GPU [1, 13, 25, 31].

To improve developer productivity, Nvidia introduced Unified Virtual Memory (UVM), enabling a unified memory programming model without the programmer explicitly managing data movement between CPU and GPU memory spaces [2, 14, 24]. UVM greatly simplifies code development by relying on the runtime software to transparently migrate data between CPU and GPU memories. However, as software solution, it also introduces significant performance penalties due to page

faulting and page migrations [2, 3, 20]. One study found that performance often degrades by 2–3× and sometimes as much as 14× compared with traditional explicit memory management [14]. Many works have proposed runtime and system optimizations, including batching, prefetching, preeviction, and migration mechanisms to enhance the performance of UVM [2, 3, 14, 16, 24]. Vendors have also explored architectures with more tightly integrated CPU and GPU memory, such as the Grace Hopper Superchip from Nvidia and the MI250X from AMD. Nonetheless, performance remains workload-dependent, and applications using the unified memory model enabled by UVM often result in suboptimal performance compared to the explicitly managed memory model.

In contrast to UVM, Unified Physical Memory (UPM) enables the unified memory programming model with hardware support. In a UPM system, a single physical memory space is shared by the CPU and GPU. AMD has recently introduced the first UPM architecture for HPC and Data Centers – the MI300A APU [32], which is used to implement El Capitan, the No. 1 supercomputer on the Top 500 list [34]. UPM could fundamentally eliminate the performance overhead in previous software-based unified memory solutions like UVM. However, nearly all existing HPC applications are using the explicitly managed model due to its superior performance compared to the unified memory model using UVM. With the emergence of UPM, this work aims to answer the open question of whether the unified memory model can now compete with the performance of explicit management.

In this work, we provide a timely full-stack characterization study of the UPM architecture on the AMD MI300A APU, including the system properties and system software support, as well as application-level performance. Our characterization methodology includes standard benchmarks and custom benchmarks specifically designed for the UPM architecture, as well as detailed insights from profiling tools, and a study of six HPC workloads from the Rodinia suite [12]. We highlight differences between memory allocators regarding performance and overhead, and analyze TLB management and Infinity Cache utilization on the UPM on MI300A APU. We identify a set of porting strategies for transitioning existing codes from the explicit model to the unified memory model. By analyzing six applications on MI300A APU, we find that UPM enables

the unified memory model to have performance on par with the explicitly managed model, while additionally reducing memory cost by up to 44%. This impressive memory saving on the UPM system enables much larger problems on one APU within a smaller envelope compared to traditional discrete GPUs. In summary, we made the following main contributions in this work:

- We provide an in-depth characterization of the unified physical memory architecture on the MI300A, including latency, bandwidth, and coherence overhead.
- We quantify the efficacy of system software support for memory management on UPM on MI300A, including memory allocation, page fault handling, TLB management, and Infinity Cache utilization.
- We transform six HPC workloads into the unified memory model and compare their performance with that from the explicitly managed model on UPM.
- Our results highlight that UPM enables the performance of the unified memory model to be on par with the explicitly managed model, while saving memory cost by up to 44%, when applying our porting strategies.

II. UNIFIED CPU–GPU MEMORY

GPU memory is usually explicitly managed as a separate memory space because the CPU and GPU have physically separated memories, e.g. with the GPU connected through PCIe as a peripheral device. This discrete GPU architecture naturally leads to the popularity of *explicitly managed memory model*, as exemplified in Listing 1, where separate memory allocations, e.g. via the `malloc` and `hipMalloc` allocators, are needed in the CPU and GPU memories, and data is copied between them explicitly via e.g. `hipMemcpy`. As a result, data is duplicated in both the CPU and GPU memories. Nevertheless, this explicit model is the most commonly used programming model in today’s GPU applications due to its high performance.

To improve programming productivity, the *unified memory programming model*, as exemplified by Listing 2, was introduced to support unified memory allocations in CPU and GPU memories, and avoid the need for explicitly initiated data movement. The unified memory model can be implemented by either software-based solutions such as UVM (e.g., via using `hipMallocManaged` allocator and implicit data movement triggered by page faults), or hardware-based solutions such as UPM, i.e., a single physical memory shared by CPU and GPU, eliminating the need for data movement.

A. Unified Virtual Memory

UVM enables the unified programming model by providing the illusion of a single coherent CPU–GPU memory by leveraging transparent page fault handling and page migration between CPU and GPU memories. Nvidia introduced UVM in CUDA 6 with the `cudaMallocManaged()` allocator. Since the Pascal GPU architecture, there is a dedicated hardware unit for page translation and migration that enables accessed pages in `cudaMallocManaged`-allocated memory regions to be

migrated on-demand. However, current UVM-based unified memory model can cause significant performance impact on applications due to page faults and migration costs [2, 14, 16, 24]. To mitigate performance overhead from page fault handling and page migration, several works propose optimizations for page prefetching and pre-eviction [2, 14].

Recently, more tightly connected CPU–GPU memory is supported by cache-coherent interconnects, such as the NVLink-C2C in Nvidia’s Grace Hopper Superchip and Infinity Fabric on AMD’s MI250X. These architectures support high-bandwidth low-latency data transfer between CPU and GPU memory, mitigating the page fault and migration overhead in traditional UVM by enabling the GPU to directly access CPU memory at cacheline granularity. However, the unified memory programming model on Grace Hopper and MI250X still needs to manage separate physical CPU and GPU memories.

A benefit of UVM over UPM is that it enables overcommitting memory on the GPU by utilizing host memory.

B. Unified Physical Memory in MI300A

In UPM architectures, CPUs and GPUs are integrated into the same die and share one physical memory. Such integrated CPU–GPU units are known as Accelerated Processing Units (APUs) on AMD systems. The AMD MI300A was recently released as the first APU targeting HPC systems. The current No. 1 HPC system on the Top500 list, the El Capitan supercomputer, features 44,544 MI300A APUs. UPM simplifies the system architecture as separate CPU chips and memory are not needed. Further, the architecture natively supports the unified memory programming model, and the high overhead of software management needed by UVM can be completely eliminated on the physically unified hardware. Finally, CPU–GPU data transfers, which are often the bottleneck in existing GPU codes, are no longer needed on UPM.

In this work, we focus on the UPM on the MI300A APU as it represents state-of-the-art HPC systems. The MI300A APU is enabled by chiplet integration and is based on the AMD CDNA 3 architecture [4, 32]. As illustrated in Fig. 1, the GPU part consists of six accelerator complex dies (XCDs) while the CPU part consists of three CPU complex dies (CCDs). The six XCDs are presented as a single device to the user in the standard configuration. Every two XCDs or three CCDs share an IO die (IOD). Four IODs on one APU implement cross-die communication and the HBM3 interface to eight memory stacks. Each memory stack has 16 memory channels and 16 GiB capacity. The AMD Infinity Fabric interconnects CCD and XCD chiplets and routes memory requests to memory channels. In total, each MI300A APU has 128 GiB HBM3 memory and a peak theoretical bandwidth of 5.3 TB/s.

The cache hierarchy consists of two levels in the GPU, three levels in the CPU, and a 256 MiB *Infinity Cache*. Atomic operations in the GPU are implemented with dedicated atomic units located in the shared L2 cache [5], while the CPU implements atomics by taking exclusive ownership of the data in the private L1 cache [23]. The Infinity Cache is a memory-side cache shared between the CPU and GPU, and is a new

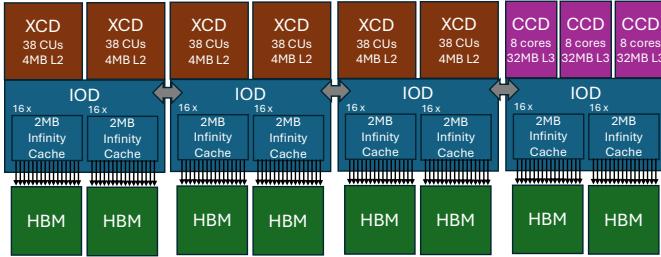


Fig. 1. An overview of the chiplet-based MI300A APU architecture including six XCD (GPU) and three CCD (CPU).

```

1 float *h = cpu_alloc(n);
2 float *d = gpu_alloc(n);
3 init_on_cpu(h);
4 copy_to_gpu(d, h, n);
5 gpu_kernel<<<...>>>(d);
6 copy_to_cpu(h, d, n);

```

Listing 1. Explicit model.

```

1 float *u = uni_alloc(n);
2 init_on_cpu(u);
3 gpu_kernel<<<...>>>(u);
4 gpu_synchronize();

```

Listing 2. Unified model.

feature of the AMD CDNA 3 architecture, aiming to increase cache bandwidth and reduce off-chip memory accesses. The peak bandwidth from the Infinity Cache can reach 17.2 TB/s, approximately $3\times$ the main memory bandwidth. The Infinity Cache does not participate in coherency and thus does not need to absorb or handle any snoop traffic, significantly improving efficiency and reducing the latency of snooping from other cache levels. It can also hold nominally uncacheable memory such as I/O buffers.

The runtime API for programming AMD GPUs is called HIP (Heterogeneous-compute Interface for Portability). It provides C++ interfaces for writing and launching GPU kernels, managing GPU memory, synchronizing CPU and GPU, etc.

C. Memory Allocation on MI300A

Two page tables are used for managing address translation on MI300A—a system page table on the CPU and a GPU page table on the GPU. Unlike Nvidia’s Grace Hopper, where the GPU can access both page tables, the GPU on MI300A can only access its own page table. Thus, page table entries (PTEs) must be propagated from the system page table to the GPU table to enable GPU access. The two copies are kept in sync using the Linux kernel’s heterogeneous memory management (HMM) subsystem.

Table I lists memory allocators on MI300A and classifies their physical memory allocation as either on-demand or up-front. *Up-front* allocators allocate all physical pages immediately when the allocator is called while *on-demand* allocators defer physical allocation until the first touch, relying on virtual memory management with page faults.

First, `malloc` is the standard `libc` function for allocating (host) memory, and serves as a representative for any standard memory allocator, including e.g. C++’s `new`. `hipHostRegister` is used to make host memory (e.g. from `malloc`) accessible on GPUs by locking the pages and mapping them in the GPU page table.

TABLE I
MEMORY ALLOCATORS ON MI300A

Allocator	GPU Access	CPU Access	Physical Allocation
<code>malloc</code>			✓ On-demand
<code>malloc (XNACK=1)</code>	✓	✓	On-demand
<code>malloc + hipHostRegister</code>	✓	✓	Up-front
<code>hipMalloc</code>	✓	✓	Up-front
<code>hipHostMalloc</code>	✓	✓	Up-front
<code>hipMallocManaged</code>	✓	✓	Up-front
<code>hipMallocManaged (XNACK=1)</code>	✓	✓	On-demand

TABLE II
OVERVIEW OF EXPERIMENTAL METHOD

Benchmarks	Memory latency	multichase [19, 21]
	Memory bandwidth	STREAM [6, 27]
	Legacy transfer	hip-bandwidth [7]
	Coherence overhead	Custom
	Allocation speed	Custom
	Page fault overhead	Custom
Profiling tools	Memory usage	libnuma
	GPU fragment size	rocprofv3
	CPU allocation size	perf
	Code generation	hipcc -save-temp
HPC workloads	backprop	
	dwt2d	
Rodinia suite [12]	heartwall	
	hotspot	
	nn	
	srad_v1	

`hipHostMalloc` directly allocates GPU-accessible page-locked memory. `hipMalloc` is the standard GPU memory allocator. Finally, `hipMallocManaged` is traditionally called to allocate UVM buffers, which are accessible from both CPU and GPU through migration (although no migration is used on UPM).

By default, the MI300A GPU does not resolve page faults, i.e., it cannot access on-demand mapped pages. To enable the GPU to resolve page faults, AMD GPUs feature a mechanism known as XNACK (“non-acknowledgment”) in the TLB, which enables page faults to be replayed [11]. With XNACK, when a page fault occurs, the TLB waits for the PTE to be updated by the fault handler before retrying the memory access, thus allowing access to on-demand mapped memory.

III. CHARACTERIZATION METHODOLOGY

We summarize our characterization method, including benchmarks, profiling tools, and HPC applications, in Table II. All of our benchmarks are open-source and available at <https://github.com/KTH-ScaLab/mi300a-benchmarks>.

We run experiments on a testbed equipped with four AMD MI300A APUs per node. Each APU has 228 GPU compute units (CUs) and 24 CPU cores, and 128 GiB HBM3 memory. The software environment includes Cray Programming Environment 24.11 and ROCm 6.3.1. We use `numactl` and `HIP_VISIBLE_DEVICES` to bind experiments to a single APU on MI300A.

A. Benchmarks

1) *Memory Latency*: We use a pointer-chasing benchmark adapted from Google’s multichase [21]. The GPU version is based on a CUDA port [19], which we modified to support HIP. We added the ability to use different memory allocators. The benchmark uses a persistent kernel that periodically increments an atomic counter, which measures the memory access time at a granularity of 200 accesses over 0.5 s per iteration from a CPU thread. We set the cache flush size to 256 MiB, the number of sample iterations to 10, and varied the buffer size from 1 KiB to 4 GiB.

2) *Memory Bandwidth*: We used a modified STREAM benchmark to measure the achievable memory bandwidth using the TRIAD kernel. For CPU, we used the standard STREAM implementation [27], and for GPU we used hipstream [6, 17]. We modified the benchmarks to support different memory allocators and data initialization on either CPU or GPU. The CPU benchmark used `OMP_PROC_BIND=true` and various number of threads from 1 to 24, selecting the best results. The CPU array size was 610 MiB, and the GPU array size was 256 MiB.

3) *Legacy CPU-GPU Data Transfer*: Many existing applications are written assuming separate memory spaces for CPU and GPU. These legacy applications can run on MI300A. However, they may incur unnecessary data transfer overhead between “host memory” and “device memory”, which no longer exist on UPM. We evaluate the cost of legacy data transfer by measuring the bandwidth of `hipMemcpy` with the `hip_bandwidth` benchmark [7].

4) *Coherence Overhead*: Many lock-free algorithms rely on high-performance atomics to resolve data races. However, parallel atomics imply coherence overhead that increases with the level of contention. The shared physical memory between CPU and GPU on UPM could further exacerbate contention and coherence overhead as data needs to be available to both the CPU and GPU.

We designed a benchmark to measure the performance of atomic operations and the coherence overhead when CPU and GPU operate on the same data structure. The benchmark computes a parallel histogram, where an array is initialized to zero, and then randomly selected elements are incremented in a loop using atomic addition. Both CPU threads and GPU threads can be used to perform this update. The throughput is measured similarly to the multichase benchmark by periodically reading an update counter from a separate CPU thread.

The CPU kernel uses `std::minstd_rand` uniform distribution to generate random numbers and is launched using `std::thread`. The compiler intrinsic `__atomic_fetch_add()` is used to implement atomic increment. The GPU kernel uses 64 threads per block and generates random numbers using XORWOW generator in the rocRAND library. Atomic increment on the GPU is implemented with `atomicAdd_system()`. (Note that the function `atomicAdd_system` is documented as “system scope” while `atomicAdd` is documented as “device scope”). They determine the `sc1` bit in the generated instruction.

However, we did not observe any difference between the two in performance or correctness.)

We performed experiments using four array sizes: 1, 1K, 1M, 1G (i.e. $2^0, 2^{10}, 2^{20}, 2^{30}$). The 1 and 1K cases fit in L1 cache, 1M fits in L2 cache, and 1G does not fit in any cache. The array contains either integer (UINT64) or floating point (FP64) elements.

5) *Allocation Speed*: Understanding the performance of different memory allocators is important for applications like adaptive mesh refinement [10] and Lagrangian hydrodynamics [22], which require allocating and deallocating memory dynamically at runtime. We design a benchmark consisting of two loops. The first loop allocates N chunks of memory of size M , while the second loop frees the chunks. We used 10 warmup iterations and set N to 100. We measure the loops using a CPU timer. There is no need for explicit device synchronization since all the allocators are inherently synchronous. We allocate 2 B to 1 GiB of data. This benchmark excludes the time to touch the allocated memory, which is studied separately in the next section on page faults.

6) *Page Fault Overhead*: On-demand memory allocators offer low latency, but come with a cost of page faults at runtime on the first touch of each page. We design a benchmark for quantifying the latency and throughput for handling different types of page faults on MI300A. The page fault overhead is the difference in runtime between accessing an already mapped page and accessing an unmapped page (causing a page fault). Our benchmark issues a single load to each page, and we measure the page fault time as the difference between running it on a newly allocated array (the faulting version) and a pre-faulted array (the non-faulting version) with a CPU timer. On the GPU, we launch a kernel to access the pages and measure the time from kernel submission to completed device sync. For memory allocation, we use `mmap` to ensure that each test is independent (`malloc` may allocate a larger chunk of address space, which is faulted in batch). The non-faulting baseline on the GPU is implemented with `hipHostRegister`, and on the CPU with `mlock`. We varied parameters such as how many pages are accessed concurrently and investigated four scenarios. In *GPU Major*, on-demand memory is allocated and directly accessed by the GPU. In *GPU Minor*, the allocated memory is touched by the CPU before measuring the fault overhead on the GPU. *1CPU* uses a single CPU core for memory access, while *12CPU* uses 12 cores. The benchmark consists of 10 warm-up iterations followed by 100 timed iterations.

B. Profiling Tools

1) *Memory Usage*: No single memory profiling interface can provide a complete picture of memory allocations on MI300A yet. Linux provides system-level memory usage via `/proc/meminfo`, and the libnuma interface reports the free memory per NUMA node, i.e., at the APU level. As expected, both reflect allocations by up-front allocators immediately, and on-demand allocators after the first touch. The HIP interface `hipMemGetInfo` and the `rocm-smi`

command report free memory "on the device", i.e. at the APU level. However, they only capture allocations by `hipMalloc`. Finally, process-level memory usage can be obtained from the `VmRSS` field in `/proc/pid/status` or the `Rss` field in `/proc/pid/smaps_rollup` (as displayed in the `top` command). However, they do not capture allocations by `hipMalloc`. We choose to profile peak memory usage by sampling from `libnuma`.

2) *GPU Adaptive Fragment Size*: AMD's GPU page table supports an adaptive scheme that uses *fragments* for improving TLB reach. A fragment is a virtually and physically contiguous range of pages with identical flags. The GPU L1 TLB can store a single entry for a whole fragment, greatly increasing its reach [8]. A larger reach means fewer TLB misses and better performance. Each PTE has a 5-bit fragment field, theoretically supporting sizes from a single page (4 KiB) to 2^{31} pages (8 TiB). The `amdgpu` driver sets the fragment field opportunistically by scanning for maximal contiguous page ranges in the fault handler.

The size of fragments in the page table cannot be read directly from userspace. Instead, we use the number of GPU TLB misses as a proxy metric. Using the `rocprofv3` GPU profiler, we measure the `TCP_UTCL1_TRANSLATION_MISS_SUM` counter to track the number of TLB misses in the GPU. We compare the number of misses in the `TRIAD` kernel of the GPU STREAM benchmark using different allocators to understand their interaction with memory fragments.

3) *CPU Allocation Granularity*: The memory allocation granularity is impacted by the used allocator and whether CPU or GPU performs first-touch. On the CPU, the number of page faults and TLB misses can imply the granularity. We track these metrics in the CPU STREAM benchmark using `perf stat`.

4) *Code Generation*: To understand which CPU and GPU instructions are generated by the compiler, we use the `-fipa-saved-temp` flag to the `hipcc/clang` compiler to output assembly code. In particular, this enables us to understand how atomic operations are implemented.

C. Porting Strategies for Unified Memory

This section outlines potential challenges arising from porting codes in the explicit model to the unified memory model and their respective porting strategies. We illustrate each challenge with a simplified example code snippet.

1) *Concurrent CPU-GPU Access*: When a data structure is accessed by CPU and GPU concurrently, simply merging them into a single buffer could result in data race. Without changing the algorithm or imposing synchronization, double buffering could be a preferred solution, i.e. swapping the buffers in each iteration instead of copying.

```

1 // gpu_kernel overlaps with next cpu_function
2 for (i = 0; i < n; i++) {
3     cpu_function(h_tmp, h_input[i]);
4     copy_to_gpu(d_tmp, h_tmp);
5     gpu_kernel<<<...>>>(d_tmp, d_sum); }
```

2) *Memory Usage Consideration*: Some applications adapt their buffering scheme based on free memory. Existing codes access memory usage counters to determine the amount of free memory capacity. However, as explained in Section III-B, previous interfaces for querying memory usage may be inaccurate to reflect all types of memory allocations on UPM. Thus, such applications must change to reliable memory usage counters, such as `meminfo` or `libnuma`. Moreover, they must adapt their calculation of free space to consider all types of memory allocations.

```

1 n = gpu_free_memory() / sizeof(element);
2 h_array = cpu_alloc(n * sizeof(element));
3 d_array = gpu_alloc(n * sizeof(element));
```

3) *Partial Memory Transfer*: Partial memory transfers arise from situations where only a partial range of a memory buffer are copied between corresponding CPU and GPU buffers. They are often used in a pipeline to overlap data movement with computation, which may become unnecessary in the unified memory model.

```

1 for (i = 0; i < n; i += chunk_size) {
2     cpu_function(h_data+i, chunk_size);
3     copy_to_gpu(d_data+i, h_data+i, chunk_size);
4     gpu_kernel<<<...>>>(d_data+i, chunk_size); }
```

4) *Stack Variables*: While UPM enables the GPU to access the host stack, the asynchronous execution model makes it challenging to analyze the lifetime of stack variables from the GPUs perspective. The host function cannot return until the GPU kernel using the host variable has completed.

```

1 x = cpu_function();
2 copy_to_gpu(d_x, x);
3 gpu_kernel<<<...>>>(d_x, d_sum);
```

5) *Static Variables*: Even with UPM, static host memory cannot be accessed from GPU code and vice versa due to linker limitations. The options for unifying static variables are using *managed variables* or modifying the code to use dynamic memory allocation (e.g., `hipMalloc`) instead. The `__managed__` storage specifier is a CUDA/HIP language extension enabling unified variables similar to `hipMallocManaged`. However, it comes with a performance penalty (as we will show in Section IV-B). On the other hand, using dynamic memory allocation requires restructuring the code.

```

1 float h_data[100];
2 __device__ float d_data[100];
```

6) *Hidden Allocator*: With libraries that allocate memory on behalf of the user (e.g. C++ containers), it can be challenging to create a high-performance unified allocation. Either a lower-performance allocator will be used (e.g. the default in C++), or the developer has to use more complex APIs or modify the library source code.

```

1 std::vector h_data;
2 while (more)
3     h_data.push_back(cpu_function());
4 d_data = gpu_alloc(h_data.size());
5 copy_to_gpu(d_data, h_data.data());
6 gpu_kernel<<<...>>>(d_data, h_data.size());
```

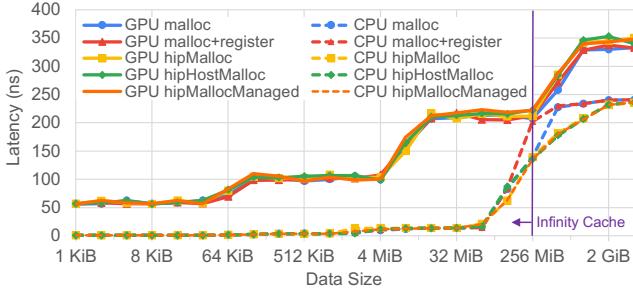


Fig. 2. Memory latency on GPU (solid lines) and CPU (dashed lines) with different allocators (semi-log).

D. Programming Model Comparison

We compare the traditional explicit programming model to the UPM-enabled unified memory model on MI300A. We select six applications (shown in Table II) covering a diverse set of coding practices from Rodinia, a widely used suite of GPU-accelerated HPC applications written in CUDA and OpenCL [12]. For each application, we create two variants. The first variant using the explicit model (corresponding to Listing 1) is a baseline version that ports the original CUDA code to HIP using hipify-perl with minor manual adjustments. The second variant uses the unified memory model (corresponding to Listing 2) by replacing duplicated CPU and GPU allocations with a single unified allocation.

We also modified the input problems to increase the memory usage and runtime. The baseline version uses from 487 MiB memory in heartwall to 43 GiB memory in nn. We measured the total execution time with `/usr/bin/time`, and inserted timers to measure the main compute phase. The total execution time ranges from 5.23 s in dwt2d to 109 s in nn.

IV. MEMORY SYSTEM CHARACTERIZATION

In this section, we provide a characterization of the UPM system properties.

A. Memory Latency

The latency results are shown in Fig. 2. GPU memory accesses reveal three cache levels – 57 ns at 1 KiB (in L1), 100-108 ns at 1 MiB (in L2), 205-218 ns at 128 MiB (Infinity Cache), and finally 333-350 ns at 4 GiB (in HBM). The CPU memory latency is lower than GPU memory latency. At 1 KiB (in L1), the CPU memory latency is only 1 ns, while at 4 GiB (in HBM), the CPU memory latency is 236-241 ns. For latency-bound tasks, the CPU has a significant advantage over the GPU. The relative difference is especially apparent for data that fits in the CPU L3 cache (96 MiB), which is missing in the GPU.

While GPU memory latency on MI300A is insensitive to the allocator in use, CPU memory latency is not. On the CPU, all allocators eventually plateau at 240 ns latency around 2 GiB. However, between L3 (96 MiB), Infinity Cache (256 MiB) and this plateau point, there is a distinction between the allocators. With 256 MiB size, the full dataset fits in IC, and at 512 MiB

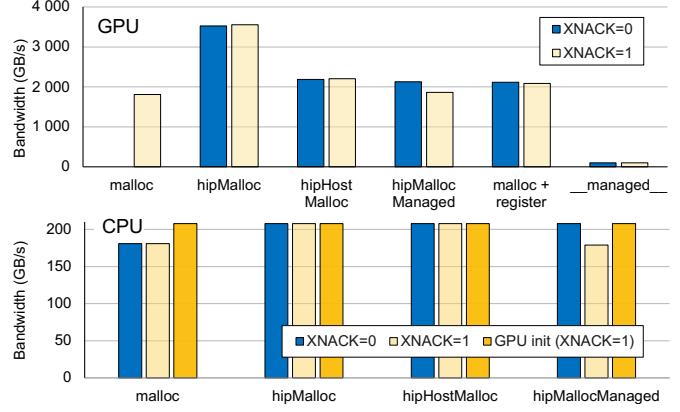


Fig. 3. The maximum measured memory bandwidth obtained from GPU (top) and CPU (bottom) using different allocators.

size, half of the accesses should hit IC. Given that, we would expect the CPU latency at 256-512 MiB to be significantly lower than 240 ns, which is observed with HIP allocators, which increase gradually. However, at 512 MiB, `malloc` and `malloc+register` already result in a latency of 230 ns. This suggests that `malloc` on the CPU cannot leverage the full power of the IC (we explore this further in Section V-D).

B. Memory Bandwidth

The memory bandwidth results are shown in Fig. 3. The GPU memory bandwidth is independent of whether the memory is first touched by the CPU or the GPU. The best GPU bandwidth is achieved with `hipMalloc` at 3.5-3.6 TB/s, while `hipHostMalloc`, `hipMallocManaged` (`xnack=0`), and `malloc+hipHostRegister` give 2.1-2.2 TB/s. The on-demand allocators `malloc` and `hipMallocManaged` (`xnack=1`) give the worst performance of the dynamic allocators at 1.8-1.9 TB/s. Finally, static unified variables with `_managed` have the lowest bandwidth at 103 GB/s.

The performance of `hipMallocManaged` depends on whether XNACK is enabled. As shown in Table I, with XNACK disabled, it allocates up-front, while with XNACK enabled, it allocates on-demand. Disabled XNACK gives higher bandwidth for both CPU and GPU.

On the CPU, the best achieved bandwidth was either 208 GB/s (case A) or around 180 GB/s (case B). The baseline bandwidth with `malloc` memory is 181 GB/s, while the bandwidth with HIP allocators is 208 GB/s. If the memory is first touched by the GPU, then `malloc` memory also achieves 208 GB/s. `hipMallocManaged` with XNACK performs similarly to `malloc` at 179 GB/s.

In case A, the peak bandwidth was reached with 24 threads (i.e. when all 24 cores were used). In contrast, in case B, the peak bandwidth was reached with only 9 threads, with performance dropping to 173-176 GB/s when using all cores (not pictured).

Regardless, the CPU is far from utilizing the full bandwidth of the memory with only 3% of the theoretical peak, compared to 67% for the GPU. For bandwidth-bound codes, the GPU

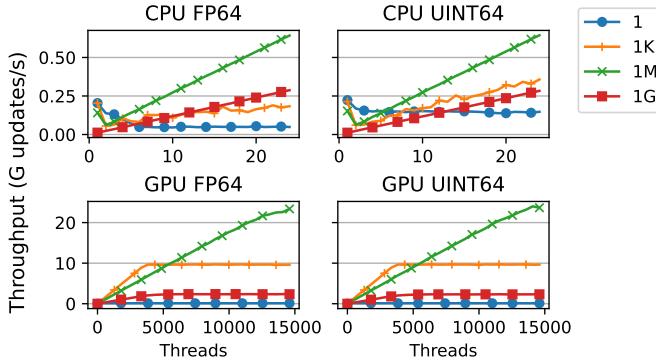


Fig. 4. Atomics throughput in billion updates/s on an array with 2^0 , 2^{10} , 2^{20} , or 2^{30} elements. Note the different axis scales for CPU and GPU.

has a clear advantage over the CPU. We found that the highest bandwidth is provided by `hipMalloc`, which is 1.6–2.0 times faster than other options on the GPU. On the CPU, on-demand allocators have a disadvantage compared to up-front allocators, unless the data is GPU-initialized, in which case all allocators provide the same bandwidth.

C. Legacy CPU–GPU Data Transfers

Using `hipMemcpy` between "host memory" (i.e. `malloc` or `hipHostMalloc`) and "GPU memory" (i.e. `hipMalloc`) is significantly slower than the achievable memory bandwidth. `hipMemcpy` only achieves a peak bandwidth of 58 GB/s, or 850 GB/s when SDMA is disabled. However, "GPU to GPU" memory transfer (i.e. `hipMalloc` to `hipMalloc`) can reach close to the GPU memory bandwidth at 1900 GB/s. A possible explanation is that `hipMemcpy` uses DMA transfers, which are more expensive when buffers are not page-locked (as in the case of `malloc`).

D. Coherence Overhead

1) *Isolated Performance*: The CPU results are shown in the first row of Fig. 4. For the three smaller array sizes, the throughput at 1 thread is higher than 2 or 3 threads, due to the introduced coherence overhead. On the 1 M array, the 1 thread case is overtaken with 6 threads and continues to scale linearly. 1 G also scales linearly but with a lower slope. These large arrays scale well since collisions between threads are relatively unlikely. 1 M is faster as it fits inside L2 cache, while 1 G requires frequent accesses to main memory. The smaller arrays 1 and 1 K have more collisions between threads. With only 1 element, performance decreases with the number of threads. The integer version (UINT64) is about 3× faster than the floating-point version (FP64). Interestingly, on the 1 K array, the FP64 version is similar or slower than 1 G, while the UINT64 version is consistently faster than 1 G.

The GPU results are shown in the second row of Fig. 4. The GPU exhibits the same performance for the FP64 and UINT64 versions and is significantly higher than the CPU performance, except when using very few threads (1 or 64) or if there is only one element. Similar to the CPU, the 1 M case has the highest throughput and scales linearly with the number of threads.

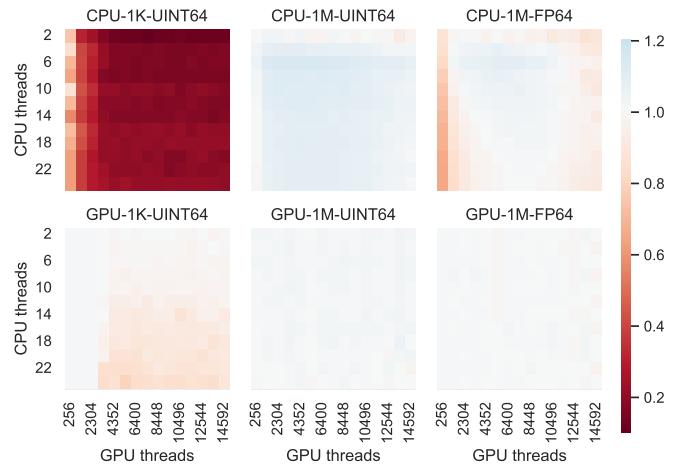


Fig. 5. Relative performance of the CPU (first row) and GPU (second row) atomics performance when co-running.

The compiler generated native `atomic_add` instructions for both integer and floating-point versions for the GPU. However, for the CPU, the compiler generated `lock incq` (atomic increment) instructions for integers, but CAS loops (using `lock cmpxchq`) for floats because the x86 instruction set does not support native atomic floating-point operations. Collisions are more expensive with the CAS loop as they lead to extra iterations through the loop. Therefore the 1 and 1 K array sizes are slower with FP64 than UINT64 on the CPU.

2) *Hybrid CPU–GPU Overhead*: Fig. 5 presents the relative performance of co-running CPU and GPU threads, as compared to the isolated performance in Fig. 4. The 1K array has the highest contention of the tested hybrid cases. This affects the CPU performance more than the GPU performance. The CPU performance is at best within 13% of the baseline, but with 3328 GPU threads or more the relative CPU performance is only between 11%–25%. Below 3328 GPU threads, the GPU performance is similar to the baseline. With an increasing number of CPU and GPU threads, the GPU performance drops off to 79%.

The 1M array has lower contention and thus higher performance. Counter-intuitively, with UINT64 the performance is slightly improved in most configurations compared with the isolated baseline. The CPU improvement is largest with 6 CPU threads and 2304–6400 GPU threads with a speedup of 1.14×. The GPU speedup is 1.02–1.03× in many cases, with a geometric mean of 1.01×. With FP64, the CPU performance is lower. There is still a region of speedup centered around the same thread configurations as for UINT64. However, with fewer than 1280 or more than 10496 GPU threads, the CPU kernel is slower than the baseline. The GPU performance is similar to the baseline with a geometric mean of 1.00.

In summary, atomics can be used to synchronize threads on both CPU and GPU. The GPU can perform more atomic operations per second than the CPU, while keys to improve atomics performance are minimizing the probability of colli-

sions and ensuring that the dataset can fit in L2 cache. The CPU FP64 performance is even more sensitive to contention since it does not support native floating-point atomics. These effects also carry over to CPU–GPU hybrid algorithms, where the CPU is more disadvantaged than the GPU by contention.

V. SYSTEM SOFTWARE FOR MEMORY MANAGEMENT

In this section, we evaluate the effectiveness of memory management for memory allocation, page fault handling, TLB management, and Infinity Cache utilization.

A. Memory Allocation

The fastest allocator is `malloc`, taking only 14 ns for allocating 32 B and 6 μ s for 1 GiB, as shown in Figure 6. This is expected since `malloc` is an on-demand allocator that does not allocate physical pages until first touch. The time for all up-front allocators is constant for allocating up to 16 KiB, indicating that this is their minimum granularity of physical memory allocation. The pattern is most revealing in `hipMalloc`, which takes 10 μ s up to 16 KiB, and then scales to 37 ms at 1 GiB. Finally, `hipHostMalloc` and `hipMallocManaged` (without XNACK) follow a similar curve, from around 15–34 μ s up to 16 KiB and then scaling to 200–400 ms at 1 GiB. Note that `hipMallocManaged` with XNACK enabled becomes an on-demand allocator, however, its execution time is constant regardless of allocation size. We believe it is caused by the overhead in the HIP implementation that is optimized for discrete GPUs.

The deallocation (figure omitted) follows a similar pattern. Interestingly, `free` is faster than `malloc` until 16 MiB. From 32 MiB, `free` takes 4–9x longer time than `malloc`. For `hipMalloc`, deallocation is faster than allocation until 2 MiB, from which deallocation becomes significantly slower than allocation by up to 22x at 256 MiB. Freeing allocation by `hipMallocManaged` with XNACK takes 3–21 μ s while freeing `hipHostMalloc` and `hipMallocManaged` (no XNACK) memory takes from 220 μ s to 67 ms at 1 GiB.

Overall, the recommended interface is `malloc` for on-demand memory and `hipMalloc` for up-front memory. For most allocations, `malloc` provides the fastest allocation and de-allocation. However, on-demand allocators pay the page fault cost at runtime, which depends on how densely or sparsely the application touches the allocated memory (see Section V-B). `hipMalloc` is the fastest up-front allocator and should be used for applications that want to avoid page fault cost at runtime.

B. Page Fault Overhead

We evaluate the overhead of page faults on GPU and CPU in terms of throughput and latency. Throughput measures the maximum number of concurrent page faults that can be handled per second while latency measures the minimum time needed for handling a single page fault.

The measured throughput of page fault handling is presented in Fig. 7. It initially increases with the number of pages until reaching a plateau at 10^5 – 10^6 pages. GPU Major reaches a

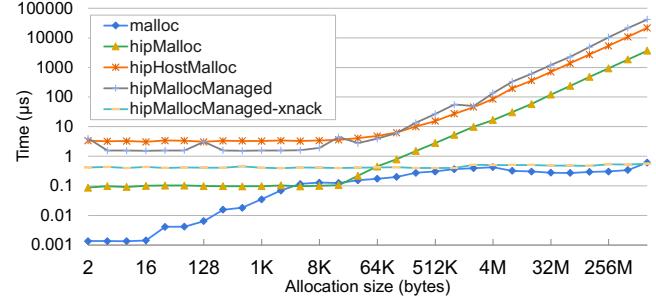


Fig. 6. The memory allocation time (μ s) using different allocators for increased allocation sizes (log-log plot).

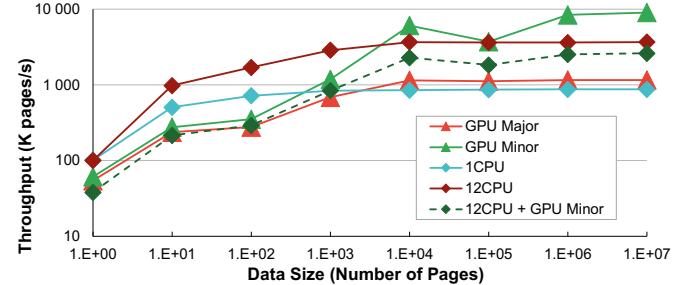


Fig. 7. The throughput of page faults in pages/s in various scenarios: first-touch fault on GPU (GPU Major); first-touch on CPU and fault on GPU (GPU Minor); fault on one CPU core (1CPU); fault on 12 CPU cores (12CPU). Log-log plot.

steady state at 10 K pages with around 1.1 M pages/s, while GPU Minor throughput increases up to 9.0 M pages/s at 10 M pages, corresponding to a third of the total memory capacity. A single CPU core saturates at 1 K pages, reaching 872 K pages/s while the 12-core CPU case saturates at 10 K pages, reaching 3.7 M pages/s. The throughput of pre-faulting on CPU and then minor faulting on the GPU (12CPU + GPU Minor), compared to the GPU Major case, achieves up to 2.2 \times improvement at 10 M pages (40 GiB).

Faults on the CPU have lower latency than GPU faults, as shown in Figure 8. The CPU single-page latency is 9 μ s on average with 11 μ s tail latency (95th percentile). The GPU latency is 1.8–2.0 times higher with 16 μ s for a minor fault and 18 μ s for a major fault. The GPU tail latency is also higher with 20 μ s for minor and 22 μ s for major faults, indicating higher variability.

The recommended strategy for applications exhibiting high concurrent page faults on GPU is to use CPU pre-faulting to transform them into GPU minor faults in advance. This is also an effective strategy for applications whose GPU runtime is dominated by GPU fault latency. However, if the time of CPU pre-faulting cannot be overlapped with GPU computation, directly faulting on GPU reduces the total latency.

C. Adaptive Memory Fragments

The fragment size in the GPU page table is related to the number of TLB misses. Fig. 9 presents the number of GPU TLB misses in STREAM. All configurations,

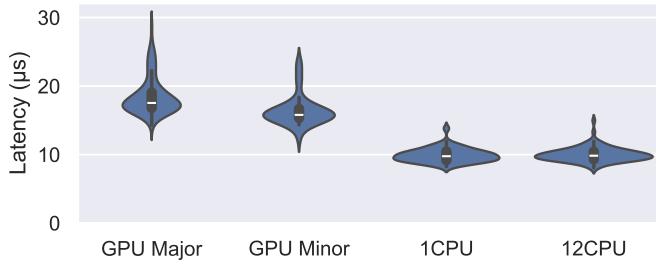


Fig. 8. The distribution of latency for resolving a single page fault on GPU and CPU.

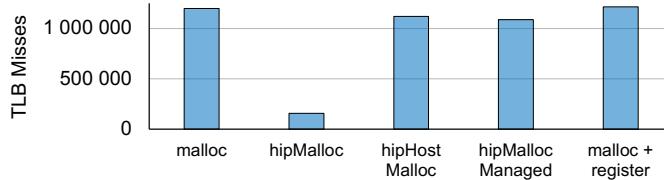


Fig. 9. The number of measured GPU TLB misses in the TRIAD kernel using five allocators.

except `hipMalloc`, have 1.0-1.2 M TLB misses, while `hipMalloc` has only 158 K misses. Since the driver sets the fragment field opportunistically based on contiguous pages, the number of GPU TLB misses on a memory range depends on the level of contiguity. On-demand allocators are naturally disadvantaged in this respect, as they allocate physical pages incrementally in a non-deterministic order. Up-front allocators can more easily ensure high contiguity by allocating all pages at once.

Our findings indicate that `hipMalloc` allocates memory with higher virtual and physical contiguity and thus uses larger fragment sizes in the GPU TLB. Consequently, memory from `hipMalloc` has fewer TLB misses, explaining the significant bandwidth advantage of `hipMalloc` shown in Section IV-B.

D. Infinity Cache Utilization

The CPU-side memory latency and bandwidth characterization in Section IV indicates that CPU-based allocators (i.e. `malloc`) and CPU initialization may result in less effective utilization of the memory-side Infinity Cache, compared with

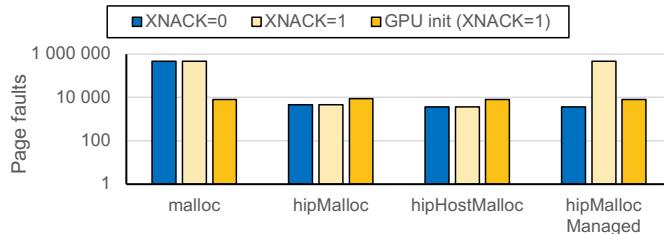


Fig. 10. The total number of page faults in the CPU STREAM benchmark, 10 iterations (log scale). Three configurations: baseline (XNACK=0), XNACK=1, and GPU init (first-touch by GPU).

HIP's up-front allocators (e.g. `hipMalloc`) and GPU initialization. It cannot be explained by different fragment sizes since memory fragments are not used in the CPU page table. Also, all allocators lead to the same number of CPU-side TLB misses.

Instead, a possible explanation for the difference in cache effectiveness lies in the mapping of data to individual memory channels, as the Infinity Cache is partitioned into slices mapped to individual memory channels [4]. Physical pages are interleaved among the eight memory stacks at a 4 KiB granularity [4]. The allocator must allocate the same number of physical pages from each corresponding physical range to evenly distribute data across channels. Any bias in the physical address mapping would result in less effective utilization of the Infinity Cache and thus higher latency and lower bandwidth for data sizes near the Infinity Cache capacity. The observed results suggest that the GPU-based allocators evenly allocate physical addresses (likely by allocating larger contiguous chunks), while CPU-initialized `malloc`-based memory has a larger bias in physical memory mapping.

Indeed, the number of CPU-side page faults (Fig. 10) varies significantly depending on the allocator. The most number of faults, around 472 K, occur with `malloc` and `hipMallocManaged` with XNACK, while `hipMalloc` and `hipHostMalloc` only have 3.7-4.6 K faults (when CPU initialized) or 8.0-8.9 K faults (when GPU initialized). The difference in page faults indicates that memory allocation granularity differs.

In summary, our findings advise developers to use up-front memory allocation (e.g. `hipMalloc`) or first-touch data on the GPU to ensure optimal physical address mapping for maximizing the utilization of the Infinity Cache.

VI. HPC APPLICATIONS ON UPM

We employed the strategies of Section III-C to port six HPC applications to the unified memory model. We summarize the implementation of each applied strategy as follows.

- Concurrent accesses arise in heartwall due to the pipelining of pre-processing on the CPU with computing on the GPU. We used double buffering with stream events synchronization in the unified version.
- In nn, `hipGetMemInfo` is used to calculate if the dataset will fit on the GPU. Our pragmatic solution was to remove the check and let the code fail if enough memory is not available.
- Partial memory transfers are used in a pipeline to overlap data movement with computation in `dwt2d` and `srad_v1`. In both cases, merging the buffers obviates the need for data movement altogether.
- A scalar flag was stored as a stack variable in `srad_v1`. The flag is set from a GPU kernel to determine the loop stop condition and is thus safe to access from the kernel.
- Static memory is used extensively for both host and device data structures in heartwall. We created two versions: `heartwall-v1` is close to the original code by using managed

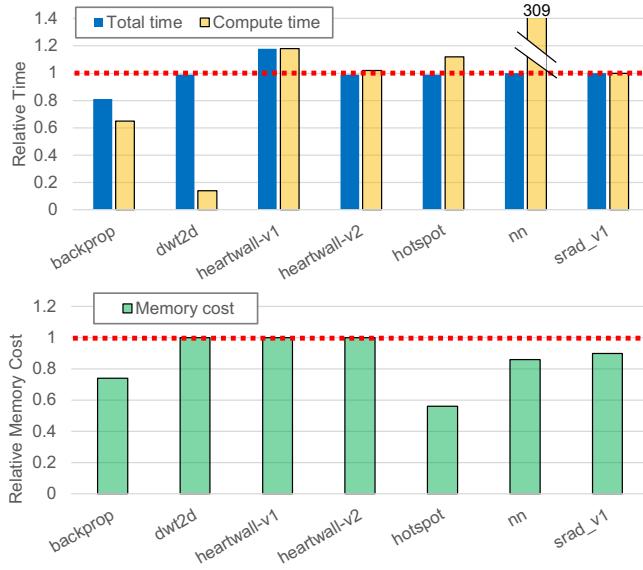


Fig. 11. The performance of six applications using the UPM-enabled unified memory model, normalized to the baseline using the explicit model. Total execution time and compute time (upper plot) and memory usage (lower plot).

static variables, while heartwall-v2 is a restructured version without static variables and repeated allocation.

- In nn, a `std::vector` is initialized on the CPU and later passed to the GPU. We decided to keep the default vector in the unified version for simplicity.

Fig. 11 presents the relative memory usage, the total execution time, and the compute time of the unified memory version, compared to the baseline, for each application. We use `hipMalloc` as the default unified allocator where possible, since it is the best performing in the characterization study.

The total execution time improved in the backprop application. By using the unified memory model, several data transfers in the main compute phase are removed, thereby reducing the compute time by 35% and total time by 19%. In dwt2d, the compute time was dominated by data transfer, and was reduced in the unified version by 86%. However, the total execution time was dominated by I/O operations outside the compute phase, and thus the two versions result in similar total runtime. In srad_v1, only a small amount of data transfer is performed in each iteration, while the runtime is dominated by kernel execution. Therefore, the compute time of srad_v1 was not significantly affected.

Static managed variables are used in heartwall-v1, leading to an 18% performance loss. In contrast, heartwall-v2 is a restructured version using dynamic memory allocation according to our porting strategy. This adaptation results in the unified memory model reaching the same performance as the explicitly managed version.

There was one performance outlier in the form of the compute time in nn, which was significantly higher than the baseline. GPU page faults on the `std::vector` significantly increased the compute time in the unified version. The magnitude of the increase is due to the relatively simple

computational kernel compared to the cost of page faults. For optimal performance, the `std::allocator` API could be applied to use `hipMalloc` instead.

We also evaluate the memory usage of the applications. In four applications (backprop, hotspot, nn, and srad_v1), the peak memory usage was reduced by 10–44% in the unified memory version, as their duplicated data in CPU and GPU buffers are merged into a unified buffer with UPM. In dwt2d and heartwall, the peak memory usage was unaffected in the unified version. In dwt2d, the peak memory usage occurs during the CPU-only IO phase, and is therefore not affected by unifying GPU data. In heartwall, the explicit host buffer plus device buffer have the same total memory usage as the UPM double buffering strategy.

In summary, the execution time of the unified memory version on UPM is competitive with the explicit model version. This is a significant step forward compared to UVM-based unified memory, which incurs high performance overhead for offering a simplified programming model [2, 3, 20]. With similar performance, UPM-enabled unified memory programming further saves up to 44% of memory usage in these applications, compared with the explicit model.

VII. RELATED WORKS

Some previous works have studied UPM on MI300A in an application-specific context. Tandon et al. [33] present OpenMP GPU offloading for UPM, porting the CFD code OpenFOAM to MI300A with OpenMP directives. Bertolli et al. [11] identify a 1.2–1.3× improved performance in QMC-Pack with direct access to the APU GPU memory, compared to the copy configuration of discrete GPUs. They also report the potential overhead of page table initialization on the APU GPU and provide the configuration of eager maps as a solution. Markidis et al. [26] ported the implicit particle-in-cell code iPIC3D to MI300A, observing only a 2% overhead from using the unified memory model while enabling simulations with a larger number of particles on up to 32,768 APUs. Schieffer et al. [30] studied inter-APU communication on MI300A systems using micro-benchmarks and the proxy applications Quicksilver and CloverLeaf, finding that `hipMalloc` buffers provide the best communication performance. Nataraja et al. [28] propose improvements to the system-level coherence for AMD APUs with a 14.4% average performance improvement on a hardware simulator. Other works characterize the memory system of earlier architectures such as Grace Hopper with various memory allocation strategies, data placement, and memory access patterns [19, 29].

Cooper et al. [16] investigate unified virtual memory in the Linux kernel’s HMM, and study the performance impacts on a diverse set of GPU workloads, revealing an aggressive prefetching strategy for demand paging. Landaverde et al. [24] investigate the performance of UVM in CUDA on synthetic and Rodinia benchmarks. They identify that UVM is limited by its high overhead and argue that the improvement in code complexity is not worthwhile. Chien et al. [14] further examine the impact of memory prefetch and hints in CUDA UVM

on application performance, showing the performance benefit from memory hints when the GPU memory is oversubscribed. In addition to UVM overhead and the impact on performance, Allen et al. [2] analyze the effectiveness of prefetch and eviction techniques in fault elimination. Choi et al. [15] focus on UVM for multi-GPU systems, providing a new approach to dynamically incorporate the spare memory of neighbor GPUs with a custom memory manager.

GPU profilers use a diverse set of methods, including API overloading, driver modification, and hardware event capture, to track the various memory behaviors like memory usage, page faults, etc. Lin et al. [25] proposed the DrGPUM profiler to automatically identify inefficient memory coding practices in GPU-accelerated applications without modification to the application, hardware, or OS. They focused on problematic memory usage, object-level and intra-object memory inefficiencies, and GPU memory optimization targets for applications. Bachkaniwala et al. [9] propose Lotus for profiling machine learning applications in PyTorch on GPUs. They focus on the preprocessing pipelines by linking the fine-grained timing of each preprocessing step to hardware-level events. Allen et al. [3] modifies the GPU driver to track events associated with servicing on-demand faults in UVM. They focus on exploiting the batch features to mitigate the high overhead in UVM.

VIII. CONCLUSIONS

In summary, this work provides the first in-depth characterization of Unified Physical Memory for CPU and GPU in AMD MI300A, including the architectural properties of the memory system, the efficiency of system software for memory management, as well as application-level performance. In six HPC applications, UPM enables the unified memory model to achieve competitive performance compared to the explicitly managed model, while saving up to 44% memory usage, when using our presented porting strategies. These results indicate that the unified memory model, once seen as a tradeoff of performance for programmability (due to software overhead in UVM), can now become the optimal choice on UPM for its high performance and significant memory saving.

ACKNOWLEDGMENTS

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-CONF-2004685. Funding from LLNL LDRD project 24-ERD-047 was used in this work. This research is supported by the Swedish Research Council (no. 2022.03062).

REFERENCES

- [1] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S. W. Keckler, “Page placement strategies for GPUs within heterogeneous memory systems,” in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015, pp. 607–618.
- [2] T. Allen and R. Ge, “Demystifying GPU UVM cost with deep runtime and workload analysis,” in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2021, pp. 141–150.
- [3] T. Allen and R. Ge, “In-depth analyses of unified virtual memory system for GPU accelerated computing,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–15.
- [4] AMD, *AMD CDNA 3 architecture*, 2023. [Online]. Available: <https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/white-papers/amd-cdna-3-white-paper.pdf>.
- [5] AMD, *AMD Instinct MI300 instruction set architecture reference guide*, 2024. [Online]. Available: <https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/instruction-set-architectures/amd-instinct-mi300-cdna3-instruction-set-architecture.pdf>.
- [6] AMD, *HIP STREAM*. [Online]. Available: <https://github.com/amd/HPCTrainingExamples/tree/main/HIP/hip-stream>.
- [7] AMD, *ROCM examples*. [Online]. Available: <https://github.com/ROCM/rocm-examples>.
- [8] AMD, *Source-code comment in amdgpu driver*. [Online]. Available: https://github.com/ROCM/ROCK-Kernel-Driver/blob/49cf5a6cfbc364b9902c20143c59302e6317ca6d/drivers/gpu/drm/amd/amdgpu/amdgpu_vm_pt.c#L760.
- [9] R. Bachkaniwala, H. Lanka, K. Rong, and A. Gavrilovska, “Lotus: Characterization of machine learning preprocessing pipelines via framework and hardware profiling,” in *2024 IEEE International Symposium on Workload Characterization (IISWC)*, IEEE, 2024, pp. 30–43.
- [10] M. J. Berger and P. Colella, “Local adaptive mesh refinement for shock hydrodynamics,” *Journal of computational Physics*, vol. 82, no. 1, pp. 64–84, 1989.
- [11] C. Bertolli, T. Blass, L. Stringer, N. Aschenbrenner, J.-P. Lehr, D. Bercea, D. Chakrabarti, L. Meadows, and R. Lieberman, “Performance analysis of runtime handling of zero-copy for OpenMP programs on MI300A APUs,” in *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, 2024, pp. 1420–1429.
- [12] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, “Rodinia: A benchmark suite for heterogeneous computing,” in *2009 IEEE international symposium on workload characterization (IISWC)*, Ieee, 2009, pp. 44–54.
- [13] G. Chen, B. Wu, D. Li, and X. Shen, “PORPLE: An extensible optimizer for portable data placement on GPU,” in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE, 2014, pp. 88–100.
- [14] S. Chien, I. Peng, and S. Markidis, “Performance evaluation of advanced features in CUDA unified mem-

ory,” in *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, IEEE, 2019, pp. 50–57.

[15] S. Choi, T. Kim, J. Jeong, R. Ausavarungnirun, M. Jeon, Y. Kwon, and J. Ahn, “Memory harvesting in multi-GPU systems with hierarchical unified virtual memory,” in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 625–638.

[16] B. Cooper, T. R. Scogland, and R. Ge, “Shared virtual memory: Its design and performance implications for diverse applications,” in *Proceedings of the 38th ACM International Conference on Supercomputing*, 2024, pp. 26–37.

[17] B. Cumming and M. Fatica, *CUDA STREAM*. [Online]. Available: <https://github.com/bcumming/cuda-stream>.

[18] J. Dongarra, “A not so simple matter of software,” 2021 ACM A.M. Turing Award Lecture, Dallas, USA, 2022.

[19] L. Fusco, M. Khalilov, M. Chrapak, G. Chukkapalli, T. Schulthess, and T. Hoefer, “Understanding data movement in tightly coupled heterogeneous systems: A case study with the Grace Hopper superchip,” *arXiv preprint arXiv:2408.11556*, 2024.

[20] D. Ganguly, Z. Zhang, J. Yang, and R. Melhem, “Interplay between hardware prefetcher and page eviction policy in CPU-GPU unified virtual memory,” in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 224–235.

[21] Google, *Multichase – a pointer chasing benchmark*. [Online]. Available: <https://github.com/google/multichase>.

[22] R. D. Hornung, J. A. Keasler, and M. B. Gokhale, “Hydrodynamics challenge problem, Lawrence Livermore National Laboratory,” Lawrence Livermore National Laboratory, Livermore, CA, Tech. Rep. LLNL-TR-490254, 2011, pp. 1–17.

[23] C. Lam, *Inside the AMD Instinct MI300A’s giant memory subsystem*, 2025. [Online]. Available: <https://chipsandcheese.com/p/inside-the-amd-radeon-instinct-mi300as>.

[24] R. Landaverde, T. Zhang, A. K. Coskun, and M. Herbordt, “An investigation of unified memory access performance in CUDA,” in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, IEEE, 2014, pp. 1–6.

[25] M. Lin, K. Zhou, and P. Su, “DrGPUM: Guiding memory optimization for GPU-accelerated applications,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023, pp. 164–178.

[26] S. Markidis, A. Hu, I. Peng, L. Pennati, I. Lumsden, D. Yokelson, S. Brink, O. Pearce, T. R. Scogland, B. R. de Supinski, G. L. Delzanno, and M. Taufer, “Exascale implicit kinetic plasma simulations on El Capitan for solving the micro-macro coupling in magnetospheric physics,” *arXiv preprint arXiv:2507.20719*, 2025.

[27] J. D. McCalpin, *STREAM: Sustainable memory bandwidth in high performance computers*. [Online]. Available: <https://www.cs.virginia.edu/stream/>.

[28] A. M. Nataraja, R. Fernández-Pascual, and A. Ros, “Enhanced system-level coherence for heterogeneous unified memory architectures,” in *2024 IEEE International Symposium on Workload Characterization (IISWC)*, IEEE, 2024, pp. 273–283.

[29] G. Schieffer, J. Wahlgren, J. Ren, J. Faj, and I. Peng, “Harnessing integrated CPU-GPU system memory for HPC: A first look into Grace Hopper,” in *Proceedings of the 53rd International Conference on Parallel Processing*, 2024, pp. 199–209.

[30] G. Schieffer, J. Wahlgren, R. Shi, E. Leon, R. Pearce, M. Gokhale, and I. Peng, “Inter-APU communication on AMD MI300A systems via Infinity Fabric: A deep dive,” in *Proceedings of the International Symposium on Memory Systems*, 2025, pp. 1–14.

[31] D. Shen, S. L. Song, A. Li, and X. Liu, “CUDA Advisor: LLVM-based runtime profiling for modern GPUs,” in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, 2018, pp. 214–227.

[32] A. Smith, G. H. Loh, M. J. Schulte, M. Ignatowski, S. Naffziger, M. Mantor, M. F. N. Kalyanasundharam, V. Alla, N. Malaya, J. L. Greathouse, *et al.*, “Realizing the AMD exascale heterogeneous processor vision: Industry product,” in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2024, pp. 876–889.

[33] S. Tandon, L. Grinberg, G.-T. Bercea, C. Bertolli, M. Olesen, S. Bna, and N. Malaya, “Porting HPC applications to AMD Instinct™ MI300A using unified memory and OpenMP,” in *ISC High Performance 2024 Research Paper Proceedings (39th International Conference)*, Prometheus GmbH, 2024, pp. 1–9.

[34] Top500.org, *June 2025 list*, 2025. [Online]. Available: <https://top500.org/lists/top500/2025/06/>.

APPENDIX

A. Abstract

The source code for the benchmarks used in this work is available at <https://github.com/KTH-ScaLab/mi300a-benchmarks>. Each benchmark is contained within a separate directory, with an accompanying makefile or build script.

B. Artifact check-list

- **Compilation:** Makefile or build.sh script.
- **Run-time environment:** ROCm 6.3.1, Cray Programming Environment 24.11
- **Hardware:** AMD MI300A APU
- **Metrics:** Atomics throughput, coherence overhead, page fault overhead, memory allocation cost, memory bandwidth, memory latency
- **How much time is needed to complete experiments (approximately)?:** < 1 h for atomics and faults, < 10 min for others
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** MIT, Apache 2.0, STREAM

C. Description

1) *How to access:* <https://github.com/KTH-ScaLab/mi300a-benchmarks> or <https://doi.org/10.5281/zenodo.16900964>.

2) *Hardware dependencies:* We use the benchmarks to study the AMD MI300A APU, but they will work on any GPU system supported by HIP/ROCM. Atomics and memory latency benchmarks require CPU-GPU coherent atomics. CPU benchmarks can run without a GPU.

3) *Software dependencies:* ROCm software environment, GPU drivers, numactl.

D. Installation

All benchmarks can be compiled using a makefile or build.sh script.

E. Experiment workflow

All of the experiments are executed on a single APU. On nodes with multiple APUs, use the following setup to bind to the first APU on the system: `HIP_VISIBLE_DEVICES=0 numactl -N0 -m0`. Each benchmark has scripts or makefile rules to help with running using the correct parameters.

F. Evaluation and expected results

The results from each benchmark are written to standard output or log files. Latency, bandwidth, and allocation speed are directly reported. The fault benchmark reports the latency of each trial; these have to be summarized as a post-processing step. The atomics benchmark reports the raw throughput; coherence overhead is obtained by calculating the ratio compared with the isolated case. The results of each benchmark should be similar to those presented in the paper.

G. Experiment customization

Most benchmarks support changing a variety of parameters, e.g. the memory allocator.