Performance Analysis of Runtime Handling of Zero-Copy for OpenMP Programs on MI300A APUs

Carlo Bertolli*, Thorsten Blass‡, Lynd Stringer†, Nicole Aschenbrenner‡, Jan-Patrick Lehr‡,
Doru Bercea*, Dhruva Chakrabarti†, Lawrence Meadows†, and Ron Lieberman*
*Advanced Micro Devices (AMD), Austin, USA,
Email: {Carlo.Bertolli, Doru.Bercea, Ron.Lieberman}@amd.com
†Advanced Micro Devices (AMD), Santa Clara, USA,
Email: {Lynd.Stringer, Dhruva.Chakrabarti, Lawrence.Meadows}@amd.com
‡Advanced Micro Devices (AMD), Munich, Germany,
Email: {Thorsten.Blass, Nicole.Aschenbrenner, JanPatrick.Lehr}@amd.com

***Abstract*—In current discrete GPU systems, the penalty of data movement between host and device memory is inevitable, forcing many large-scale applications to include optimizations that amortize this cost. On systems like the AMD Instinct™ MI300A series accelerators, based on the accelerated processing unit (APU) architecture, host and device memories are unified into a single physical storage. On an APU, the GPU can access memory in the same way the CPU does, thus avoiding the need for additional data movement (zero-copy). To inform developers of MI300A on expected advantages and potential overheads, we follow an experimental approach to study our OpenMP implementation that leverages MI300A zero-copy. Performance results show that zero-copy is faster than the legacy "copy" implementation by a ratio of 1.2X-2.3X for a production-ready application, but that incurs up to 11% penalty for one SPECaccel 2023 benchmark.**

I. Introduction

Most GPU-accelerated large-scale systems used in high performance computing (HPC) applications [1], [2] are based on a *discrete memory architecture*, where CPUs and GPUs each have their own physical memory storage. The historical impact of this is that most applications deployed since the advent of GPU-systems use latency hiding strategies in their code. Separate storage between CPU and GPU influences node and operation price, as well as application performance, due to the need of moving data near the computation. AMD Instinct MI300A series accelerators [3], standing for accelerated processing unit (APU), approaches these problems by having the CPU and GPU device on the same socket share the same physical (main memory) storage. Data transfers needed by previous discrete GPUs are no longer required. The APU architecture offers an opportunity to continue improving exascale computing, but requires software changes to fully capture the advantage.

A major concern for HPC users is that incorporating software changes and new optimizations into their existing applications leads to high maintenance costs. Past application porting efforts, to support the end of CPU-only nodes and the introduction of heterogeneous CPU-GPU nodes alongside grid-programming languages like CUDA® [4] and HIP [5], are a testament of how high the application re-engineering costs can be. Today, these costs are mitigated by mature high-level programming abstractions (OpenMP [6], [7], Kokkos [8], SYCL® [9], Raja [10], OpenACC [11]) and domain-specific frameworks [12], [13]. These solutions are widely adopted by HPC production-grade applications that rely on efficient compilers and runtimes to provide performance portability.

In this paper, we describe our work that is based on the existing OpenMP LLVM implementation and that leverages its unified shared memory *zero-copy* semantics to model the APU's single physical memory. In zero-copy, there is no extra memory allocation due to GPU offloading and GPU threads rely on unified memory to access CPU-allocated memory. We implemented two theoretically simple - albeit technically sophisticated - extensions of LLVM's zero-copy implementation to be able to execute in zero-copy *any* OpenMP application [14], [15], not just those that have been designed to use the `unified_shared_memory` construct, and to optimize unified memory overheads by prefetching the GPU page table. The resulting zero-copy runtime *configurations* expose an optimization space that we study in this paper, to drive the next iteration of OpenMP compiler and runtime for MI300A.

For applications ported from discrete GPU systems to MI300A without modifications, we study the performance impact of two commonly used memory management optimizations: data prefetching and data streaming. Data prefetching consists of a bulk CPU-to-GPU memory data transfer at the start of the application followed by a long-running GPU computation with minimal or no data transfer. In a multi-threaded setup, data streaming is used to hide the data transfer cost of one thread behind the kernel execution of another. We show that our implementation offers the best performance possible for the QMCPack [16] production-grade OpenMP application without removing these optimizations for discrete GPUs. We further study zero-copy performance on MI300A by

979-8-3503-5554-3/24/$31.00 ©2024 IEEE
DOI 10.1109/SCW63240.2024.00183

reporting on experiments using SPECaccel® 2023 [17] C/C++ benchmarks, which exercise OpenMP application programming patterns different from QMCPack.

This paper makes the following contributions:

- LLVM-based compiler and runtime optimizations that enable transparent zero-copy to obtain best performance on MI300A for applications optimized for discrete GPUs.
- An experimental study of the effects of data transfer optimization strategies in a production-grade application when ported to an APU system using zero-copy.
- An experimental study on how zero-copy performs on MI300A with various OpenMP programming patterns exercised by SPECaccel 2023 C/C++ benchmarks.

Section II describes related work. Section III gives background information on APU architecture and the OpenMP programming model. Section IV describes the LLVM extensions we implemented for MI300A. Section V presents our experiments. Finally, Section VI synthesizes the lesson learned from the experiments and Section VII concludes.

II. Related Work

GPU programming languages such as CUDA and HIP require applications to manage host and device memory directly in their program without an intervening abstraction, using host and device memory allocation routines (e.g., `malloc` for host memory and `cuda/hipMalloc` for device memory). Porting a CUDA or HIP application from a discrete GPU system to an APU requires a re-implementation of how an application manages its data in memory, potentially completely eliminating device memory management in the program.

There are HPC languages and frameworks whose programming constructs include abstract data management that is similar to OpenMP and are relevant to ease of porting to APU systems. Examples are Kokkos [8] and SYCL [9]. Kokkos includes the concept of *Memory Space* that is used for generic programming and can be instantiated to a unified memory implementation. Porting an application from a discrete GPU system to an APU requires instantiating the abstract Memory Space used by an application with an implementation that uses Unified Memory. Switching an abstract class implementation between different subclasses is straightforward in C++.

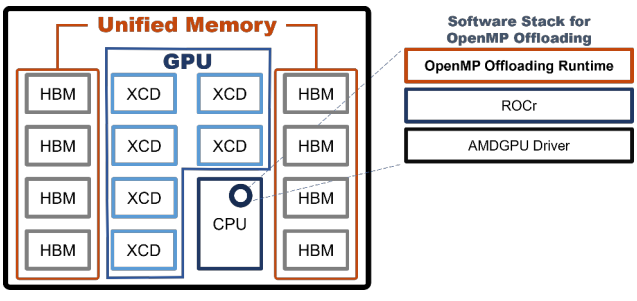

Fig. 1. Compiler view of AMD Instinct MI300A series accelerator architecture and the OpenMP offloading software stack.

SYCL includes a similar concept to OpenMP, called *Unified Shared Memory*. Allocating memory in Unified Shared Memory provides the program with a pointer that can be accessed on both CPU and GPUs without translation, further allocations or memory copies. Allocations in Unified Shared Memory are characterized by their *kind*, which can be `host`, `device`, or `shared`. Porting a SYCL application from a discrete GPU system to an APU requires changing the allocation kind used in the program. We expect Kokkos, SYCL, and other similar languages and frameworks to offer APU implementations whose performance characteristics will reflect that of OpenMP, and that will include similar optimization mechanisms as described in this paper.

The performance characteristics of unified memory systems have been studied for discrete GPU systems in the past. In an earlier study of OpenMP offloading with unified memory capability, it was found that performance was considerably lowered when the GPU memory was oversubscribed [18]. Another study [19] reports the behavior of HIP applications using unified memory on discrete GPUs. Authors found that using unified memory may improve programmer productivity, but that it has non-trivial performance characteristics and impact.

To target one of the key performance factors of unified memory, the authors propose a compiler-driven hybrid execution for explicit and implicit data management [20]. In [21] the authors present a detailed analysis of NVIDIAs unified virtual memory system and reveal insights into the behavior of page fault generation and servicing.

Other approaches exist to help developers with software-defined unified memory schemes for specific application domains, such as graph processing [22], oversubscribing GPU memory for very large deep neural networks [23], or using multiple GPU devices [24].

III. Background

A. Abstract APU Architecture

Fig. 1 shows the architecture of a single APU socket. Each socket features: an array of high bandwidth memory (HBM) units, which are configured as a single logical *memory* from the perspective of the CPU and GPU, a set of *CPU* cores, and a set of accelerated compute die(s) (*XCD*), which are configured as a single logical GPU device.

APU sockets can be composed together in a multi-socket accelerator card, where either CPU or GPU threads on a socket can access memory located in a different socket. GPUs in different sockets are seen by OpenMP as multiple devices. Programmers of a multi-socket APU card can either: program multiple sockets using a single OpenMP program, by carefully selecting CPU and GPU thread affinity (e.g., CPU thread running on a socket offloads to the GPU device on the same socket) or use one MPI process per socket, with multiple OpenMP threads per process, to offload to the GPU device on the same socket. The experiments we analyze in this paper are on a single socket APU.

```
#pragma omp declare target (alpha)
double alpha;
int main() {
  int N = ...; // problem dependent
  double * a = new double[N];
  double * b = new double[N];
  // initialize a and b and alpha from I/O
  FileInput(N, a, b, &alpha);

  #pragma omp target teams loop map(tofrom: a[:N]) \
      map(to: b[:N]) map(always, to: alpha)
  for(size_t i = 0; i < N; i++)
      a[i] += b[i] * alpha;
}
```

Fig. 2. Example program for OpenMP offloading with data environments.

B. System Software

The ability for threads on CPU and GPU to access the same memory using same addresses, may it be on a discrete GPU system or an APU, is obtained using Unified Memory support, which includes the XNACK capability. We give an abstract description of the role of XNACK on an APU as it is relevant to optimizations described later.

Suppose a GPU thread is provided, via kernel arguments, an address `addr` that was obtained by a CPU thread using an OS library call, such as `malloc`. The first time the GPU thread accesses a page pointed to by `addr`, the GPU page table does not contain a translation for `addr`. The XNACK-replay protocol implements a search in the CPU page table and the insertion of the page table entry in the GPU page table, to enable logical-to-physical address translation. GPU threads are stalled while the GPU Translation Lookaside Buffer (TLB) implementation is executing the XNACK-replay protocol. This cost is one-off per page during the course of application execution.

The OpenMP software stack (see Fig. 1) is relevant to the implementation of certain functionalities on APUs, such as device memory allocation. ROCr [25] is an implementation of the HSA™ [26] API with extensions for AMD GPUs. OpenMP uses ROCr to implement data management and kernel execution on AMD GPUs. The OpenMP offloading runtime, ROCr, and the driver run on the CPU.

When a program invokes a device memory allocation routine (e.g., `hipMalloc`) on an APU socket, as there is no separate device memory, the driver invokes the OS memory allocator to fulfill the request. Legacy code using `hipMalloc` is still supported, but effectively results in unnecessary memory duplication and transfers.

C. OpenMP

The reader is encouraged to refer to the OpenMP specification [6] and OpenMP GPU offloading programming guides [7] for full details on OpenMP offloading. In this section, we describe a simple program that gives essential information needed for the understanding of the paper.

Fig. 2 shows a program that performs a numerical computation on two arrays and a global variable. Pragma `target teams loop` is used to offload from host thread to a GPU device using multiple workgroups (known as threadblocks in HIP and CUDA) and threads in each workgroup to execute the for-loop iterations in parallel.

OpenMP abstracts CPU and GPU memory using *data environments*. A CPU or GPU thread can access data present in their own data environments. For instance, pointers `a` and `b` and the memory they point to are available in the CPU data environment when declared and allocated in the `main` function. The construct `map` for `a` and `b` adds the pointers and their pointed memory to the GPU data environment, so that GPU threads can access them. Global variables can be made available on GPU data environments by declaring them using pragma `declare target`, as shown for global `alpha` in the example program.

It is important to understand that data environments are not the same as physical storage: the host and device data environments in the program example can be implemented by the same physical storage or by separate ones. Data environments are an abstraction for programmers to specify memory consistency and visibility from CPU and GPU threads. In a different perspective, programmers cannot assume that programming multiple host and device data environments necessarily means using multiple physical storage and that mapping memory means GPU memory allocation and CPU-GPU data transfer.

Mapping memory is not necessary when programmers use the *requirement* pragma `unified_shared_memory`, where a pointer "[..] will always refer to the same location in memory from all devices [..]" [6] and "Host pointers may be passed as device pointer arguments to device memory routines [..]" [6].

On a discrete GPU, data mapping is implemented with device memory allocations and host-device data transfers. In the program of Fig. 2, the `map` constructs are implemented as follows: device memory allocations for `a` and `b` arrays before the kernel launch; host-to-device (`to`) data transfer for arrays `a` and `b` before kernel launch; device-to-host (`from`) data transfer for array `a` after the kernel has completed executing. On an APU, we refer to this implementation as *copy*.

IV. APU Programming in OpenMP

As described in Section III, the OpenMP abstract data model is designed to enable implementations to map host and device data environments either to different physical storage or to the same one. The "native" way to implement the OpenMP runtime on APUs is to use *zero-copy*: the OpenMP runtime does not perform extra unnecessary memory allocations and data transfers to implement map constructs; CPU and GPU threads access the same physical pages for mapped data using the same logical addresses. We describe different runtime *configurations* and how they handle the creation and modification of data environments. From an OpenMP semantics viewpoint, they are all equivalent.

A. "Legacy" Copy

The "Legacy" Copy configuration maps GPU device memory allocations to the single APU memory. When memory is mapped, the OpenMP runtime allocates "device" memory via ROCr and the GPU driver. The driver allocates HBM space, in the same way OS-allocators (e.g., `mmap`) do. As the GPU runtime is transparent to the language runtime (see Section III), the OpenMP runtime can operate in the default copy mode that is normally used on discrete GPU systems without any changes.

In the example of Fig. 2, mapping `a` and `b` means that the OpenMP runtime matches them with the pointers and memory returned by ROCr when allocating "device" memory (let's call them `a_prime` and `b_prime`). Consequently, while the CPU thread executing the `main` function accesses `a` and `b`, the GPU threads executing in the target region access the matching `a_prime` and `b_prime` pointers and their memory. Memory copies between CPU and GPU are implemented as HBM-to-HBM copies.

B. Unified Shared Memory

Unified Shared Memory is obtained by adding `#pragma omp requires unified_shared_memory` to all translation units of an application. The implementation of Unified Shared Memory configuration predates the work done for this paper and it is available in LLVM trunk [27] as well as ROCm 6.1.1 [28]. It is the main mechanism underlying the OpenFOAM MI300A porting results described in [29]. In this paper, we briefly review its implementation details as a comparison point for our extensions.

In Unified Shared Memory, when memory is mapped, no storage operation is performed. Target regions that result in kernel launches are passed host memory pointers by value. This makes the Unified Shared Memory configuration an implementation of the zero-copy pattern.

Using the requirement pragma affects the behavior of both compiler and runtime for global variables. Consider the program in Fig. 2. The compiler produces two code objects, one for the GPU and one for the CPU. One difference between the two code objects is the implementation of the global `alpha`. In the generated CPU code, it is a global variable in the module, with the same type as in the source code. In the generated GPU code, it is a global variable, but the type is *pointer to* the type of the original global variable. At initialization time, the pointer in the GPU code object is assigned the address of the global in host memory. Code in target regions is generated using double indirection to access the global `alpha`.

Applications built with `#pragma omp requires unified_shared_memory` can only be deployed on GPUs that support Unified Memory. They cannot be switched between Copy and Unified Shared Memory implementations based on deployment environment. As such, they are less portable than the remaining solutions.

C. Implicit Zero-Copy

Implicit Zero-Copy (or Implicit Z-C in short) extends the zero-copy behavior for local variables to programs that do not use pragma `unified_shared_memory`. The OpenMP runtime detects that the system it is running on is an APU and that XNACK (Unified Memory support) is enabled in the current run environment and automatically toggles the same runtime behavior used for Unified Shared Memory described above. Unlike Unified Shared Memory, an application running as Implicit Z-C on MI300A (or supported APUs) is run as Copy configuration on discrete GPUs, potentially optimizing CPU-GPU memory transfer operations using carefully placed mapping operations. This makes Implicit Z-C the performance portable solution for applications that are optimized for discrete GPUs.

Special care is needed for global variables. Referring back to the example in the previous section, when building an application without `unified_shared_memory`, the compiler generates a copy for `alpha` in both CPU and GPU code objects. That is, the CPU and each GPU have their own copies of the global variable. Kernel code that accesses the global does not use the double indirection mechanism described in Unified Shared Memory, but it directly accesses the global location in the GPU code object.

When operating in Implicit Zero-Copy, the OpenMP runtime knows that the application was not built with `unified_shared_memory` and it switches handling of globals as if operating in Copy mode. When a global is mapped, system-to-system memory transfers are issued. In this way, the host and device copies are kept consistent according to OpenMP mapping semantics.

Implicit Zero-Copy is a solution for programmers that do not want to change their applications but still want to obtain zero-copy implementation. This configuration is available both in LLVM "trunk" (see patch [14]) and in ROCm 6.1.1 [28][1].

D. Eager Maps

As described in Section III, when a GPU thread accesses a memory page allocated via an OS-allocator (e.g., `mmap`) for the first time, the driver loads its page table entry into the GPU page table (XNACK mechanism). Subsequent uses of the same page do not require loading the same entry again, but first access cost is generally an expensive operation. OpenMP data management abstractions can be used to optimize this process. The assumption is that programmers must map all memory that is used in target regions, which is true when the program does not use the `requires` clause `unified_shared_memory`. Any memory location that a kernel accesses needs to be mapped before the kernel is launched. Upon mapping, the OpenMP runtime triggers GPU page table prefaulting using a ROCr call. This call does not allocate memory, nor replaces

[1]ROCm provides Implicit Zero-Copy as an opt-in run configuration also on discrete GPU systems, by setting the environment variable OMPX_APU_MAPS to value 1 in an environment where XNACK support is enabled (e.g. HSA_XNACK environment variable is set to value 1).

the original memory allocated by the CPU thread that is being mapped, but it adds to the GPU page table all CPU page table entries corresponding to the mapped memory. This eager ahead-of-time prefaulting operation is expensive the first time a page is added to the GPU page table. Similar to what happens in the XNACK algorithm, the host page table is walked to identify the entry to be added. Unlike XNACK, this operation is issued from the host side and requires supervisor privilege to modify page tables, using a system call.

Any subsequent prefault issued from the host for a page that has already been prefaulted is a simple access to the GPU page table to verify that the page is present. The cost is higher than in the case of the XNACK protocol, because accessing the GPU page table on the host requires a system call, similar to when modifying it.

When using this Eager Maps configuration, the GPU does not need to run with XNACK support. The OpenMP runtime behaves in zero-copy mode, the same way as for the Implicit Zero-Copy configuration. Section V describes quantitative trade-offs between Implicit Zero-Copy and Eager Maps configurations. Eager Maps is available in ROCm 6.1.1 [28].

V. Experiments

In this section we present experimental results that aim at answering the following two questions:

- What are the performance consequences of using APU programming mechanisms described in previous sections?
- Do I have to rewrite or re-optimize/tune my application when moving to an APU?

We test QMCPack [16], a production-grade application that uses OpenMP offloading for its main user-programmed kernels, and C/C++ benchmarks in SPECaccel 2023. SPECaccel 2023 benchmarks exercise multiple programming patterns for OpenMP GPU offloading, which enabled us to study corner cases for the zero-copy configurations. Results are obtained running with *base* tuning and *ref* (reference workload) size options (see [30] for more information about SPEC accel run configurations).

Experiments were performed on an AMD Instinct MI300A series accelerator with a single socket, with one CPU and one GPU, with AMDGPU driver version 6.3.5.

Each SPECaccel 2023 experiment is run 8 times. QMCPack experiments are run 4 times each, to limit the total execution time for the experiments. The median value is used to compute ratios and we report the Coefficient of Variation (CoV) to support statistical robustness of our claims. To match Copy and zero-copy configurations, we turn on Transparent Huge Pages (THP) so that both configurations work with 2MB page sizes.

A. Experiments with QMCPack

QMCPack features more than fifty `target` constructs for OpenMP offloading and roughly the same number for data, task, and parallelism management. QMCPack is especially relevant to study zero-copy performance on MI300A because it employs two main optimization patterns tailored for discrete GPUs:

- Ahead-of-time Data Transfer: this is implemented as a bulk data transfer at the beginning of the application, followed by a long-running GPU computation phase with minimal data transfers.
- Data Transfer Latency Hiding: this is implemented by using multiple OpenMP host threads offloading to the same device, where a data transfer issued by a thread can be masked by profitable computation in a kernel from another thread.

We show that these optimizations do not hurt performance when using zero-copy programming mechanisms and that they are effective at minimizing overheads when running in Legacy Copy mode on MI300A.

Results shown in this paper are for running QMCPack "NiO" performance tests [31], from small (S2) to large (S128) problem sizes. Results for the smallest available problem size (S1) is not included in this paper, as it is not representative of GPU application performance: it spends all execution in the offloading runtime and minimal time in GPU kernels, resulting in zero-copy configurations disproportionately winning over Copy.

Small problem sizes are used to study the overheads incurred by a specific OpenMP implementation, while large problem sizes reflect the performance of real production experiments. We build QMCPack using the following configuration:

```
-DQMC_COMPLEX=OFF
-DQMC_MIXED_PRECISION=OFF
-DQMC_MPI=OFF, -DENABLE_CUDA=OFF,
-DQMC_CUDA2HIP=OFF,
```

We do not use HIP device library in QMCPack because that results in memory allocation being performed using HIP device runtime, which nullifies the effects of OpenMP zero-copy implementations.

Full details on QMCPack build configuration and related instructions can be found at [32]. We run QMCPack using a single MPI process with 1, 2, 4, and 8 OpenMP CPU threads offloading to the same device. The version of QMCPack that we used is obtained from the main development branch (`develop`) at: https://github.com/QMCPACK/qmcpack with latest git commit SHA `78340fd6`.

Library dependencies for QMCPack are as following: libopenblas-dev 0.3.20, hdf5-1.12.0, BOOST 1.74.0.3, libfftw3-3. This configuration is sufficient for single node performance runs.

1) Raw Data Graphs: All experimental data provided in this paper compare three run configurations against the baseline Legacy Copy implementation. The three configurations are: Unified Shared Memory, Implicit Zero Copy, and Eager Maps. Each graph contains three lines, each corresponding to the ratios between the execution time of Copy and Unified Shared Memory, Implicit Zero Copy, and Eager Maps.

The first set of graphs (see Fig. 3) shows the values for the described ratios for a fixed problem size when using 1, 2, 4,

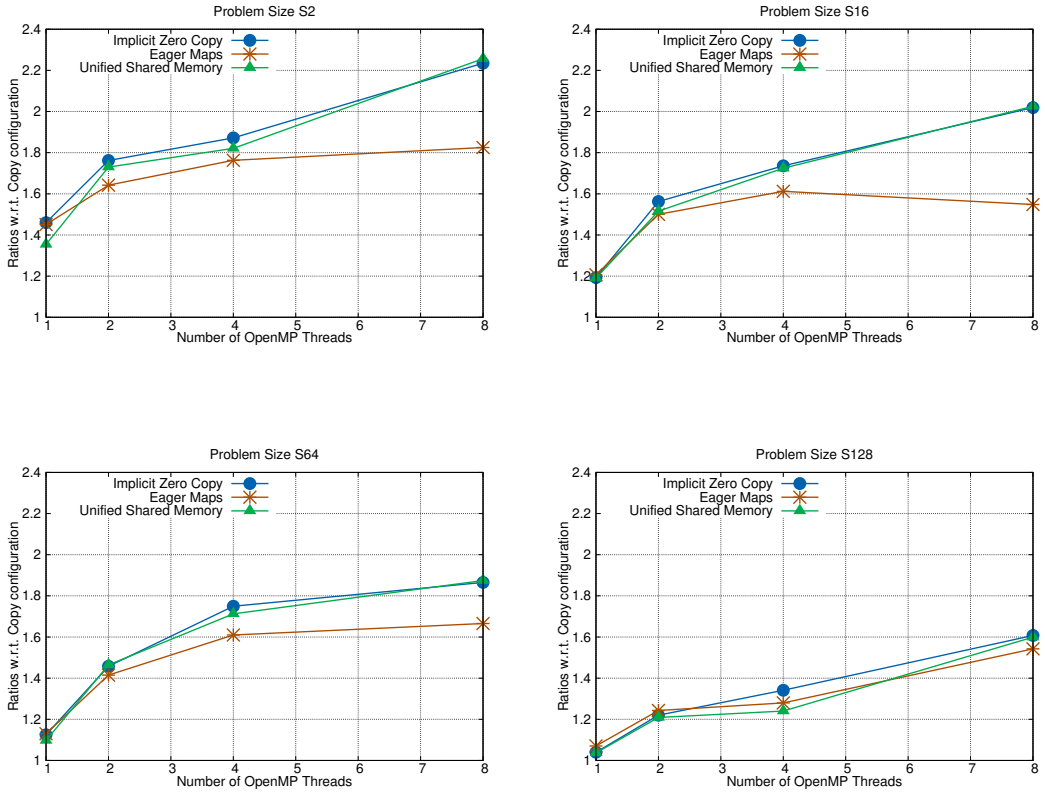

Fig. 3. Ratios between the execution times of Copy and zero-copy configurations for different problem sizes by varying number of OpenMP threads.

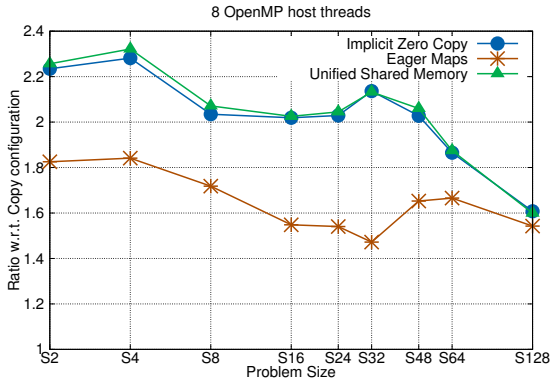

Fig. 4. Ratios between the execution times of Copy and zero-copy configurations when using 8 OpenMP host threads by varying problem size.

and 8 OpenMP threads on the host to offload to the GPU. Different graphs refer to different problem sizes.

The graph in Fig. 4 is sourced from the same data as Fig. 3, but it highlights the ratio behavior with 8 OpenMP host threads when varying the problem size.

Experiments show the following Coefficient of Variation (CoV) values: Copy has 0.03, Implicit Zero Copy has 0.10, Unified Shared Memory has 0.08. Eager Maps has two major outliers: S32 and 8 threads has one data point that is one order of magnitude larger than the rest of the data and has a CoV of 4.2. Eager maps uses a system call to prefault TLB entries and the anomalous data point may be due to random interference by the operating system. S128 has a maximum CoV of 2.9, and above 1 for any number of OpenMP threads. We believe this is due to TLB thrashing. For all other problem sizes, Eager Maps shows 0.03 CoV. In general, these CoV values indicate that the reported ratios are robust w.r.t. statistical variations in the performance measurements.

2) Performance Comparison of Implicit Zero-Copy and Unified Shared Memory with Copy: Overall, the graphs in Fig. 3 show that Implicit Zero-Copy and Unified Shared Memory configurations for APUs perform similarly and always better than the "Legacy" Copy configuration. As a reminder, the two APU configurations only differ at runtime in the way they handle global variables. QMCPack does not use global variables, which justifies the two configurations having

identical results.

To understand these results, we analyzed the case of S2 with one OpenMP thread using rocprof HSA call tracing. Table I, left side, lists some of the most significant, in terms of latency, calls to HSA performed when running in Copy and Implicit Zero-Copy configurations. Latency ratio is computed by dividing the total time spent in Copy configuration executing the ROCr call, by the same metric for Implicit Z-C, as reported by rocprof.

Copy configuration uses the first signal-related HSA call for kernel dispatch and memory copies and Implicit Zero-Copy almost exclusively for kernel dispatch. The rest of the HSA calls are used for memory allocation and management. Implicit Zero-Copy uses these calls for loading the GPU code and data, and other support structures, during initialization. Copy additionally uses these calls to implement OpenMP data mapping during the whole execution.

Latency ratios reported in Table I include operations performed during initialization, whereas Fig. 3 only includes steady-state computation ratios. While the ratios reported in the table cannot be immediately used to compute the ratios in the graphs, they indicate the source of difference between Copy and zero-copy configurations in general reported in Fig. 3: Copy spends thousands more time than Implicit Zero-Copy in copying memory between two HBM locations.

Going back to the graphs in Fig. 3 we see that with 8 OpenMP threads results in improved ratios between Implicit Zero-Copy and Unified Shared Memory against the "Legacy" Copy configuration. This is because more OpenMP threads will share the same runtime stack, including components such as the OpenMP host and offloading runtimes, ROCr, and the driver and GPU firmware. When running in Legacy Copy configuration, OpenMP threads on the host need to: allocate memory for the GPU, transfer memory between CPU and GPU, and launch kernels and synchronize all operations between themselves using HSA signals. APU configurations only need to launch kernels and synchronize their execution, as there is no extra memory to manage. This means a smaller number of calls to the runtime, which pays off when increasing the number of threads.

To justify this conclusion, we study the HSA calls performed by Copy and Implicit Zero-Copy configurations for S2 input with 8 OpenMP threads (see Table I, right side). Similar to the case of 1 OpenMP thread, Implicit Zero-Copy performs a significant smaller amount of calls to HSA. However, we notice that the number of calls to the HSA API increases for both Copy and Implicit Zero-Copy when using 8 OpenMP threads compared to using 1 OpenMP thread. Specifically, in Copy, HSA call _memory_async_copy is called 1,124,258 for Copy with 8 OpenMP threads, and 307,607 for 1 OpenMP thread. As Implicit Zero-Copy does not call that function, the latency ratio between the two configurations is significantly higher compared to the case of one OpenMP thread. This explains why increasing the number of OpenMP threads used to offload to the GPU results in increased difference between Copy and Implicit Zero-Copy.

3) Effects of Data Transfer Optimizations: Results in Fig. 4 show that performance advantages obtained with APU configurations diminish when the problem size increases. This is because memory management overheads incurred in the Legacy Copy configuration are minimized (in fact, hidden) due to the optimizations that QMCPack includes for discrete GPU systems. As noted elsewhere, on an APU, data transfers are between different locations in the same storage and it has generally higher performance than a host-device memory transfer in discrete GPUs. As there is less to hide, less host multi-threading is needed on an APU than a discrete GPU for data transfer hiding purposes. This supports our claim that portability from discrete GPU systems and APUs does not require a rewriting of data management optimizations, but that they can help improving performance of the Legacy Copy configuration.

To understand this trend, we analyzed two ratios between S2 and S24 problem sizes, where S2 represents the highest ratio point and S24 a local minimum. Total kernel execution times reported by rocprof for Copy and Implicit Zero-Copy configurations increases 10 times between S2 and S24. Total HSA call execution time increases 5X for Copy and 10X for Implicit Zero-Copy, although the latter has a much smaller total compared to the former. As expected, going from S2 to S24, QMCPack generally uses larger data structures rather than ten times their number, which can be transferred by roughly the same amount of memory copy operations, resulting in a smaller increase of Copy compared to Implicit Zero-Copy. From these results, we infer that increases in problem size reflects in memory copy overheads (for Copy) about at half rate than kernel execution time. This means that kernel time starts dominating total execution time and it becomes the first order overhead for large realistic problem sizes.

4) Eager Maps: We notice that the Eager Maps configuration diverges from Implicit Zero Copy and Unified Shared Memory configurations in some of the runs. The implicit cost of Eager Maps is that of a system call every time a memory page is mapped from a CPU thread. The first time some memory is mapped, the cost is of the order of the number of pages to be added to the GPU page table. When the same memory is mapped again, the cost is that of a system call, as there is nothing to add. Unlike Eager Maps configuration, Implicit Zero-Copy and Unified Shared Memory use page fault handling when a GPU thread touches a page for the first time. Successive accesses to the same page are free.

Graphs of Fig. 3 and highlighted in Fig. 4 show that for smaller problem sizes than S128, the performance advantage of Eager Maps over the Copy configuration scales at a lower rate than the other two APU configurations. This disadvantage is not present for S128, as the problem size is large enough that the overhead incurred due to GPU TLB prefaulting is hidden by the average time it takes to execute kernels. In a sense, this is similar to data management overheads and their optimization. There is an inherent cost per thread that is paid to prefault the GPU page table, which is of the order of the latency of a system call. That cost is hidden by memory

TABLE I
HSA API CALL STATISTICS FOR COPY AND IMPLICIT ZERO-COPY FOR QMCPACK PROBLEM SIZE S2 WITH 1 AND 8 OPENMP THREADS.

		1 OpenMP Thread			8 OpenMP Threads		
		Copy	**Implicit Z-C**	**Copy/***	**Copy**	**Implicit Z-C**	**Copy/***
ROCr/HSA Call	**Used for**	**#Calls**	**#Calls**	**Latency Ratio**	**#Calls**	**#Calls**	**Latency Ratio**
signal_wait_scacquire	Kernel Completion	351,653	99,627	2.07	1,360,088	738,483	2.71
memory_pool_allocate	Allocate device memory	23,277	19	7.41	20,848	90	3.68
memory_async_copy	Memory copy	307,607	3	3,190	1,124,258	3	1.11×10^4
signal_async_handler	Memory copy	194,848	0	N/A	491,492	0	N/A

management optimizations for discrete GPU systems and we have already noted above that increasing the problem size makes such optimizations more effective.

To justify these statements we analyzed two metrics in comparison between Implicit Zero-Copy and Eager Maps for S2 input with one OpenMP thread. First, the advantage of Eager Maps over Implicit Zero-Copy is due to increased TLB hits when host allocated memory is first touched by the GPU. We found that, for the first hundred kernel launches, the difference between the two configurations is in the order of tens of milliseconds. After the initial phase, the difference lowers to milliseconds and lower, persisting due to the use of host-allocated arrays to perform cross-team (block) reductions on the host. The advantage of Eager Maps sums to less than a second, in the order of a tenth of a second.

Second, the advantage of Implicit Zero-Copy over Eager maps is due to the absence of calls, on the host, to the TLB prefaulting ROCr/HSA API call, named `svm_attributes_set`. Using rocprof tracing, this call is called over a million and a half times and it costs a few seconds over the entire application execution. If considering whole application execution time, we can see that Eager Maps saves less than a second, but pays a few seconds to perform prefaulting.

B. Experiments with SPECaccel 2023

TABLE II
RATIOS BETWEEN COPY AND EACH ONE OF THE ZERO-COPY CONFIGURATIONS FOR SPECACCEL 2023. RATIO HIGHER THAN 1 MEANS ZERO-COPY CONFIGURATION PERFORMS BETTER THAN COPY.

Benchmark	**stencil**	**lbm**	**ep**	**spC**	**bt**
Implicit Z-C	0.99	1.05	0.89	7.80	4.88
Unified Shared Memory	0.99	1.043	0.89	7.61	4.77
Eager Maps	0.98	1.025	0.99	8.10	5.10

SPECaccel 2023 benchmarks include OpenMP programming patterns that, unlike QMCPack, are not specifically optimized for discrete GPUs and exercise relevant corner cases when comparing Copy with the three zero-copy configurations. We only consider C/C++ tests on a single-socket MI300A node (SPECaccel does not use MPI). We report the ratio between Copy configuration and the other zero-copy configurations (see Table II). For all configurations and runs, the highest detected Coefficient of Variation was 0.03. This indicates robust numerical results in our tests.

TABLE III
OVERHEAD ANALYSIS FOR 403.STENCIL AND 452.EP. MM IS THE OVERHEAD INDUCED BY MEMORY MANAGEMENT, INCLUDING GPU-SPECIFIC MEMORY ALLOCATION AND CPU-GPU MEMORY COPIES. GPU-SPECIFIC MEMORY MEANS MEMORY ALLOCATED VIA ROCR AND ASSOCIATED WITH ITS GPU AGENT, PHYSICALLY LOCATED IN THE SAME STORAGE AS HOST-ALLOCATED MEMORY. MI IS OVERHEAD INDUCED BY FIRST TOUCH (OR INITIALIZATION) OF MEMORY ON THE GPU, WHICH INCLUDES EXECUTING THE XNACK PROTOCOL FOR EVERY PAGE OF THE TOUCHED MEMORY. TIME QUANTITIES WERE OBTAINED BY SPECIFYING `LIBOMPTARGET_KERNEL_TRACE=3` WHEN RUNNING THE APPLICATION AND ANALYZING THE RESULTING TRACE.

Base unit: microsec.	**stencil**		**ep**	
	MM	MI	MM	MI
Copy	O(10^5)	O(0)	O(10^5)	O(0)
Implicit Z-C or USM	O(0)	O(10^6)	O(0)	O(10^6)
Eager Maps	O(10^4)	O(0)	O(10^5)	O(0)

Benchmarks 403.stencil and 452.ep report lower performance in zero-copy configurations, compared to copy. In Copy configuration, 403.stencil performs two data copies, between host thread allocated memory and ROCr allocated memory, at the beginning and at the end of the simulation. 452.ep allocates GPU memory using ROCr but does not perform memory copies. Steady-state computations of both kernels access memory exclusively from the GPU - with an exception for scalar reduction variables. Let's denote the **memory management** overhead for Copy as MM_{copy}. Implicit Zero-Copy and Unified Shared Memory fold the memory management operations for these two benchmarks resulting in $MM_{zcopy} = 0$.

Looking at individual kernel performance, we noticed that 452.ep initializes memory in a target region, which performs worse if the memory being initialized was obtained using an OS-allocator (e.g., malloc, mmap) compared to using a ROCr memory allocator, as in Copy configuration. This is due to different management of TLB with XNACK-enabled and XNACK-disabled. When initializing memory that was originally allocated using an OS-allocator, in XNACK-enabled state, GPU TLB page faults are performed while the kernel is running, upon touch of a memory page and page-by-page. When initializing memory that was originally allocated using ROCr (GPU memory pool allocator), in XNACK-disabled state, GPU TLB page faults are performed in bulk upon allocation. This is evident when running with driver debug prints turned on: XNACK-enabled (Implicit Zero-Copy, Unified Shared Memory) shows TLB fault handling events upon first touch. XNACK-disabled (Copy) shows none. Let's denote

the cost of **memory initialization** (MI) for zero-copy (Implicit Zero-Copy and Unified Shared Memory) as MI_{zcopy} and for Copy as MI_{copy}. From the description above, $MI_{copy} = 0$. 403.stencil suffers less from the first-touch overhead due to the fact that it initializes a much smaller array compared to 452.ep.

For 403.stencil, MM_{copy} is of the order of hundreds of thousands of microseconds. $MM_{zcopy} = 0$. $MI_{copy} = 0$ and MI_{zcopy} is of the order of a few million microseconds. The difference between the overhead incurred by Copy (MM_{copy}) and that incurred by zero-copy (MM_{zcopy}) is negative in value and of the order of tens of microseconds. This explains why zero-copy configurations perform slightly worse than Copy.

For 452.ep, $MM_{copy} = MM_{zcopy} = 0$. $MI_{copy} = 0$ and MI_{zcopy} is of the order of a few million microseconds. That makes total execution time of Copy configuration faster by a few million microseconds than Implicit Zero-Copy and Unified Shared Memory, justifying the 0.89 ratio reported in Table II.

We note that Eager Maps seems to alleviate part of the slowdown for 452.ep. When using Eager Maps, OpenMP memory mapping translates to prefaulting the GPU page table, which translates to $MI_{eager} = 0$, effectively matching Eager Maps performance to that of Copy. Table III summarizes the overheads in terms of their orders of magnitude for the 403.stencil and 452.ep benchmarks.

404.lbm performs a large data transfer at the beginning of the application, when running in Copy configuration. This is not executed for the zero-copy configurations, which consequently perform slightly better. 457.spC and 470.bt show significant performance improvements when using any zero-copy configurations. 457.spC performs data allocations and data deletions every 13 kernel launches, and the memory being allocated is in the order of GBs. Data allocations are synchronous w.r.t. subsequent kernel launches, due to data dependency, and are not parallelized in the runtime due to the individual copy sizes. Kernel executions inside the data allocation and data deletion sequence may take up to 6% the time it takes to perform a single allocation. Consequently, the Copy configuration is significantly slowed down compared to zero-copy configurations. 470.bt is similar, except that the largest data allocation is above 2GBs, 10 kernels are executed between the data allocation and data deletion sequence, and the most time consuming kernel is approximately 30% of the time it takes to execute the largest data allocation. Both 457.spC and 470.bt return best results when using eager maps. Similar to other benchmarks, this is due to the fact that implicit zero-copy and unified_shared_memory suffer from the first-touch initialization issue for the data being otherwise allocated in Copy configuration. Host data is allocated on the program stack at each of the containing host function invocation, and is first-touched on the GPU every time the function is called. Eager maps overcomes that issue by prefaulting the GPU page table and avoiding GPU-issued page faults and related kernel handling overhead.

VI. Lessons Learned

We identified the following two MI300A optimizations as most effective.

Folding Memory Copies: for Copy configuration, the most significant source of overhead is data transfers. Frequent and large data transfers cancel any positive one-time effect on kernel execution enabled by using ROCr allocators for data. All zero-copy configurations overcome this overhead by folding copy operations. This overhead is significant in QMCPack and SPECaccel 2023 benchmarks 457.spC and 470.bt.

GPU TLB Bulk Page Faulting: for Implicit Zero-Copy and Unified Shared Memory configurations, the most significant source of overhead is GPU page table faulting. When in Copy mode, ROCr allocator triggers bulk GPU page table prefaults and first-touch memory initialization kernels are not affected by page table faults while running. In Implicit Zero-Copy and Unified Shared Memory page faults are handled during kernel execution, page-by-page, resulting in worse performance compared to Copy. Eager Maps overcomes first-touch overhead of zero-copy by prefaulting host-allocated memory upon mapping, and before it is touched on the GPU, resulting in TLB page fault free initialization. This overhead is the most significant and 452.ep and has a minor role in 403.stencil.

Our initial analysis of SPECaccel 2023 shows that TLB prefaulting of Eager Maps is profitable when a large amount of memory is first touched on the GPU. For smaller sizes and multiple arrays, each prefaulting request introduces an overhead that should be considered against the cost of page-by-page GPU fault handling. This is evident in QMCPack where we see frequent prefaulting requests to the driver that are issued concurrently by multiple OpenMP host threads, resulting in lowered Eager Maps performance compared to the other zero-copy configurations.

VII. Conclusion

This paper shows the results of our study on porting applications optimized for discrete GPUs to the MI300A APU GPU. We extended the LLVM OpenMP compiler and runtime to support zero-copy for all OpenMP applications on APUs and we applied our implementation to the QMCPack production application and to SPECaccel 2023 C/C++ benchmarks to study its effects. Experiments show that implementations based on zero-copy always achieve superior performance compared to the Copy configuration, with few corner cases that we clearly identify as being caused by GPU page table initialization. For those corner cases, the eager maps configuration is a solution that enables running with zero-copy but without the overhead of Implicit Zero-Copy and unified_shared_memory. Furthermore, data transfer optimizations implemented for discrete-GPU systems do not slow down zero-copy based implementations, but they improve performance of the copy configuration. As a result, we expect all production grade applications to move to zero-copy without major limitations.

REFERENCES

[1] (November 13, 2023) Frontier remains No. 1 in the TOP500 but Aurora with Intel's Sapphire Rapids chips enters with a half-scale system at No. 2. 11/17/2023. [Online]. Available: https://www.top500.org/news/frontier-remains-no-1-in-the-top500-but-aurora-with-intels-sapphire-rapids-chips-enters-with-a-half-scale-system-at-no-2/

[2] O. R. N. Laboratory. (May 30, 2022) Frontier supercomputer debuts as world's fastest, breaking exascale barrier. 11/14/2023. [Online]. Available: https://www.ornl.gov/news/frontier-supercomputer-debuts-worlds-fastest-breaking-exascale-barrier

[3] AMD Instinct™ MI300A Accelerators. 12/8/2023. [Online]. Available: https://www.amd.com/en/products/accelerators/instinct/mi300/mi300a.html

[4] CUDA C++ Programming Guide. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[5] HIP Programming Manual. [Online]. Available: https://rocm.docs.amd.com/projects/HIP/en/latest/user_guide/programming_manual.html

[6] *OpenMP Technical Report 12: Version 6.0 Preview 2*, 2023, available at https://www.openmp.org/wp-content/uploads/openmp-TR12.pdf.

[7] T. Deakin and T. G. Mattson, *Programming Your GPU with OpenMP: Performance Portability for GPUs*. The MIT Press, 11 2023. [Online]. Available: https://doi.org/10.7551/mitpress/14866.003.0015

[8] Kokkos Documentation: Memory Spaces. [Online]. Available: https://kokkos.github.io/kokkos-core-wiki/API/core/memory_spaces.html#memory-spaces

[9] Heterogeneous Programming with SYCL: Data Management with Unified Shared Memory. [Online]. Available: https://enccs.github.io/sycl-workshop/unified-shared-memory/#usm-memory-allocation

[10] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. W. Scogland, "RAJA: Portable Performance for Large-Scale Scientific Applications," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2019, pp. 71–82.

[11] *The OpenACC Application Programming Interface*, 2022, https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.3-final.pdf.

[12] Pytorch Webpage. [Online]. Available: https://pytorch.org/

[13] Tensorflow Webpage. [Online]. Available: https://www.tensorflow.org/

[14] (2024) [OpenMP] Enable automatic unified shared memory on MI300A. [Online]. Available: https://github.com/llvm/llvm-project/pull/77512

[15] (2023) SC23 OpenMP Booth Talk: OpenMP Target Offloading for AMD GPUs. [Online]. Available: https://www.openmp.org/wp-content/uploads/sc23-openmp-offloading-lehr.pdf

[16] P. R. C. Kent, A. Annaberdiyev, A. Benali, M. C. Bennett, E. J. Landinez Borda, P. Doak, H. Hao, K. D. Jordan, J. T. Krogel, I. Kylänpää, J. Lee, Y. Luo, F. D. Malone, C. A. Melton, L. Mitas, M. A. Morales, E. Neuscamman, F. A. Reboredo, B. Rubenstein, K. Saritas, S. Upadhyay, G. Wang, S. Zhang, and L. Zhao, "QMCPACK: Advances in the development, efficiency, and application of auxiliary field and real-space variational and diffusion quantum Monte Carlo," *The Journal of Chemical Physics*, vol. 152, no. 17, p. 174105, 05 2020. [Online]. Available: https://doi.org/10.1063/5.0004860

[17] SPECaccel ® 2023. 3/18/2024. [Online]. Available: https://www.spec.org/accel2023/

[18] A. Mishra, L. Li, M. Kong, H. Finkel, and B. Chapman, "Benchmarking and Evaluating Unified Memory for OpenMP GPU Offloading," in *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*, ser. LLVM-HPC'17. New York, NY, USA: Association for Computing Machinery, 2017.

[19] Z. Jin and J. S. Vetter, "Evaluating Unified Memory Performance in HIP," in *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2022, pp. 562–568.

[20] L. Li and B. Chapman, "Compiler assisted hybrid implicit and explicit GPU memory management under unified address space," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019.

[21] 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*, ser. SC '21. New York, NY, USA: Association for Computing Machinery, 2021.

[22] P. Wang, J. Wang, C. Li, J. Wang, H. Zhu, and M. Guo, "Grus: Toward Unified-memory-efficient High-performance Graph Processing on GPU," *ACM Trans. Archit. Code Optim.*, vol. 18, no. 2, feb 2021.

[23] J. Jung, J. Kim, and J. Lee, "DeepUM: Tensor Migration and Prefetching in Unified Memory," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 207–221.

[24] A. K. Ziabari, Y. Sun, Y. Ma, D. Schaa, J. L. Abellán, R. Ubal, J. Kim, A. Joshi, and D. Kaeli, "UMH: A Hardware-Based Unified Memory Hierarchy for Systems with Multiple Discrete GPUs," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 4, dec 2016.

[25] (2024) HSA Runtime API and runtime for ROCm. [Online]. Available: https://github.com/ROCm/ROCR-Runtime

[26] *HSA Platform System Architecture Specification*, 2018, available at http://hsafoundation.com/wp-content/uploads/2021/02/HSA-SysArch-1.2.pdf.

[27] (2024) Llvm "trunk" repository. [Online]. Available: https://github.com/llvm/llvm-project

[28] (2024) AMD ROCm 6.1.1 documentation. [Online]. Available: https://rocm.docs.amd.com/en/docs-6.1.1/

[29] S. Tandon, L. Grinberg, G.-T. Bercea, C. Bertolli, M. Olesen, S. Bnà, and N. Malaya, "Porting HPC Applications to AMD Instinct™ MI300A Using Unified Memory and OpenMP," 2024.

[30] Using SPECaccel® 2022: The 'runaccel' Command. 9/24/2024. [Online]. Available: https://www.spec.org/accel2023/Docs/runaccel.html

[31] (2024) QMCPack NiO Performance Tests: How to Obtain and Run. [Online]. Available: https://qmcpack.readthedocs.io/en/develop/installation.html#role-of-qmc-data

[32] QMCPack User's Guide and Developer's Manual. [Online]. Available: https://qmcpack.readthedocs.io/en/develop/