Inter-APU Communication on AMD MI300A Systems via Infinity Fabric: a Deep Dive

Gabin Schieffer, Jacob Wahlgren, Ruimin Shi
{gabins,jacobwah,ruimins}@kth.se
KTH Royal Institute of Technology
Sweden

Edgar A. León, Roger Pearce, Maya Gokhale
{leon,pearce7,gokhale2}@llnl.gov
Lawrence Livermore National Laboratory, USA

Ivy Peng
ivybopeng@kth.se
KTH Royal Institute of Technology
Sweden

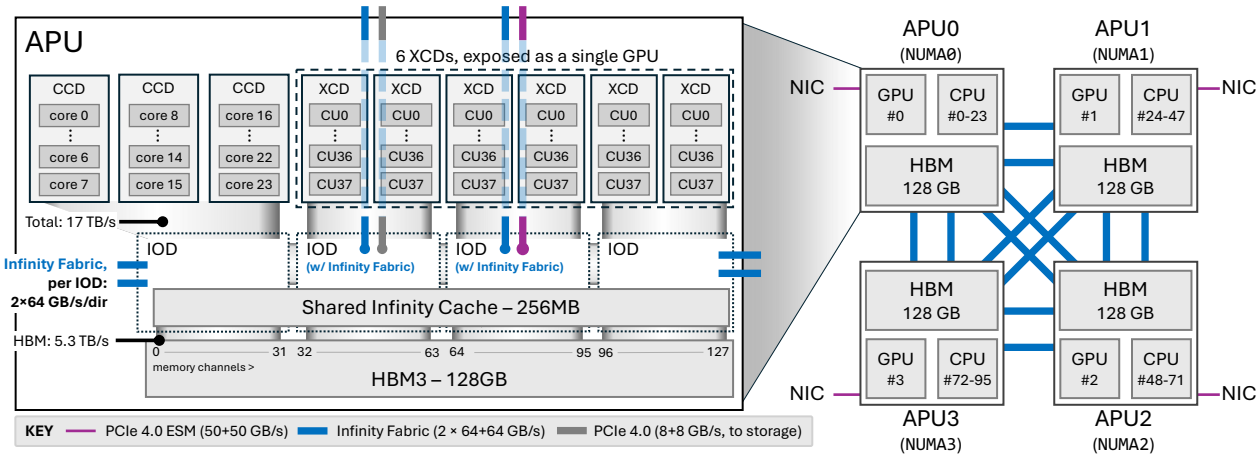


Figure 1: Node Architecture (right) with four MI300A APUs, and detailed APU architecture (left). The Infinity Fabric (in blue) interconnects the four APUs. From the user perspective, each APU is a NUMA node in this cache-coherent NUMA system.

Abstract

The ever-increasing compute performance of GPU accelerators drives up the need for efficient data movements within HPC applications to sustain performance. Proposed as a solution to alleviate CPU-GPU data movement, AMD MI300A Accelerated Processing Unit (APU) combines CPU, GPU, and high-bandwidth memory (HBM) within a single physical package. Leadership supercomputers, such as El Capitan, group four APUs within a single compute node, using Infinity Fabric Interconnect. In this work, we design specific benchmarks to evaluate direct memory access from the GPU, explicit inter-APU data movement, and collective multi-APU communication. We also compare the efficiency of HIP APIs, MPI routines, and the GPU-specialized RCCL library. Our results highlight key design choices for optimizing inter-APU communication on multi-APU AMD MI300A systems with Infinity Fabric, including programming interfaces, allocators, and data movement. Finally, we optimize two real HPC applications, Quicksilver and CloverLeaf, and evaluate them on a four MI300A APU system.

Keywords

GPU, MI300, AMD APU, Inter-GPU Communication, Infinity Fabric

1 Introduction

Multi-GPU nodes have become prevalent in leadership supercomputers and High-performance Computing (HPC) systems. Starting from pre-exascale machines, where nodes feature four to six powerful GPUs, a compute node today can be unprecedentedly powerful in providing computing capacity that used to require hundreds and even thousands of CPU-only nodes. Recently, two exascale supercomputers, El Capitan (No. 1) and Frontier (No. 2), have their compute nodes based on multiple AMD GPUs [18, 7]. In particular, ORNL's Frontier supercomputer consists of four AMD Instinct MI250X GPUs per node, and two Graphics Compute Dies (GCD) per GPU, presenting as an eight-GPU node to users. LLNL's El Capitan supercomputer features four AMD MI300A Accelerated Processing Units per node, each APU combines a CPU part and a GPU part on the same package. Although integrated CPU and GPU are not new to the mobile and laptop markets, HPC and data centers mainly use discrete GPUs. In fact, the AMD MI300A APU is the first integrated CPU and GPU that specifically targets HPC. This work provides a timely understanding of data movement mechanisms and strategies for efficient communication on multi-APU systems.

Even before the emergence of multi-APU nodes, programming and utilizing multi-GPU systems efficiently has been a challenge [7, 11, 12]. In HPC applications, two main programming paradigms are used: the first one uses a single process in the shared-memory model and leverages NUMA-aware thread binding to schedule tasks into multiple GPUs; the second one continues the MPI-based distributed-memory model to have a separate process mapped for each GPU. Efficient data movement on multi-GPU systems has been identified

as a critical optimization aspect. For that, recent vendors have introduced high-performance cache-coherent interconnects, such as Nvidia NVLink-C2C and AMD Infinity Fabric to connect GPUs [12, 30]. The complex meshes of GPUs could result in multiple data paths being usable to transfer data between two GPU endpoints, and consequently, different hardware engines along routes may be utilized for acceleration [9, 10].

This work provides an in-depth understanding of data movement and communication on the emerging multi-APU systems and optimization strategies for preparing HPC applications. Since GPU-accelerated applications are often constrained by how fast the data can be supplied to GPU for computation, efficient APU-APU communication becomes increasingly important to fully exploit the system potential. We base our study on the recently implemented multi-APU node, exemplified by El Capitan supercomputer. The large number of GPU and CPU dies connected with Infinity Fabric interconnects create a complex node-level mesh [30], as illustrated in Figure 1. To ensure a representative coverage of the study, we start with a taxonomy of communication mechanisms on multi-APU nodes, classifying them into four categories – direct GPU kernel access, explicit memory transfer, point-to-point and collective inter-process communication.

Guided by this taxonomy, we design a set of micro-benchmarks for dissecting and quantifying the performance of data movement using available software interfaces and compare the obtained performance with the peak hardware capacity. In addition to the benchmarking results on MI300A systems, we also provide a comparison with those on previous generation MI250X systems. Different from discrete GPUs, on APU, CPU and GPU share the same physical memory, and thus memory management on APU may impact communication. Therefore, we expand the study to evaluate the impact of memory allocation methods and CPU or GPU first-touch strategies on data movement. Moreover, we assess the effectiveness of specialized hardware units, namely System Direct Memory Access (SDMA) engines, the XNACK mechanism, and interaction with Linux's Heterogeneous Memory Management (HMM) system. Finally, we conduct two case studies using the Quicksilver and CloverLeaf applications. By optimizing their communication bottlenecks, we achieve up to 2.15× speedup in communication.

We made the following contributions in this work:

- We provide a taxonomy of communication strategies on multi-APU systems, including direct access, explicit transfer, point-to-point, and collective communication;
- We propose a methodology for benchmarking communication and the impact of data paths, programming interfaces, and allocators;
- We provide the first quantitative characterization of AMD MI300A-based multi-APU systems and identify key optimization insights for efficient APU-APU communication;
- We evaluate in two HPC applications Quicksilver and CloverLeaf the effectiveness of inter-APU communication optimization strategies.

2 AMD MI300A based Multi-APU Systems

Starting with El Capitan supercomputer, multi-APU nodes with integrated CPU and GPU parts become accessible to HPC applications. This section introduces APU, interconnects, and memory management on multi-APU nodes.

2.1 Accelerated Processing Unit

The AMD MI300A is an Accelerated Processing Unit (APU), which combines a CPU and a GPU on the same package, sharing a single physical memory region. More details on its unified physical memory between CPU and GPU are described in [34]. This unified physical memory design contrasts with the one taken for the Nvidia Grace Hopper Superchip, where two physical memory spaces are still used, but interconnected by cache-coherent NVLink-C2C interconnect [27]. While APUs have been widely used in consumer electronics for a long time, the introduction of such system in HPC is recent. AMD MI300A APU is built on the principle of chiplets. In the manufacturing process, chiplets are design blocks with a well-defined set of functionalities, which can be reused and combined to design more complex hardware. The inset of Figure 1 highlights the hardware characteristics at a high level.

MI300A APUs are composed of a combination of Core Complex Dies (CCDs), which implement CPU cores; Accelerator Complex Dies (XCDs), which form a GPU; memory dies, which use the HBM3 technology; and Input/Output dies, which implement memory-side caching and IO abilities for the attached processors, XCDs and CCDs. On each APU, three CCDs are used; each CCD exposes eight AMD Zen 4 CPU cores, for a total of 24 CPU cores per APU. On the GPU side, the MI300A APU features six XCDs, with 38 compute units (CU) per XCD, totaling 228 compute units over the entire APU. In the simplest configuration, the six XCDs are exposed to the user as a single GPU, with no explicit control of the mapping of GPU kernels to XCDs.

On the GPU side, the compute units implement the CDNA3 microarchitecture [30]. The L1 data cache is 32 KB per CU, with a cacheline size of 128 bits, L1 instruction cache is 64 KB, shared between pairs of two CUs. The L2 cache is shared between all CUs of a single XCD, with 4MB per XCD. All memory traffic to/from the XCD is coalesced in the L2 cache, where cache-coherence is also enforced with the rest of the APU. On the CPU-side, the CPU cores implement the Zen 4 microarchitecture. The L1 data cache is 32 KB, and the L1 instruction cache is 32 KB. Each CPU core has 1 MB of L2 cache. All cores of a CCD share a 32 MB L3 cache.

A key particularity of this system is the *Infinity Cache*, which is last-level cache (LLC), shared between all XCDs and CCDs, and implemented on the memory side. The entire LLC is 256 MB, distributed into 128 slices of 2 MB each. Each slice is paired with exactly one of the 128 memory channels [30]. This cache is implemented as part of the Input Output Die (IOD). In total, each APU features 128 GB of HBM3 memory. The physical memory is distributed across eight HBM stacks, with two stacks attached to each IOD. This configuration leads to a total of 512 GB of HBM3 memory over the entire quad-APU compute node.

2.2 Infinity Fabric Interconnect

At a higher level, MI300A HPC systems are built from a four-APU node architecture, where four MI300A are grouped onto a single board composing an HPC node. Notably, this configuration is featured on the El Capitan supercomputer and is the focus of this work. Figure 1 presents this architecture.

The key element in this system is the Infinity Fabric (IF) interconnect that connects the four APUs on each node. This interconnect implements the xGMI 3 interface (Inter-chip Global Memory Interconnect 3), also used in other categories of AMD products [4]. A single IF link is 16 bit-wide and operates at a transaction rate of 32 GT/s on the node, giving 64 GB/s per direction. Each pair of APUs is connected with two IF links, supporting 128 GB/s bandwidth per direction. This symmetric architecture is depicted in Figure 1.

For each APU, the IF links are implemented as part of the IODs. Each IOD connects to one IF link, and one configurable link, used for either Infinity Fabric or PCIe 5.0. With four IODs per APU, each APU has a total of six IF links to connect to its peer APUs, with two links dedicated for each peer. In addition, on El Capitan, on each APU, one x16 PCIe 4.0 ESM (Extended Speed Mode) link connects to the Network Interface Controller (NIC), with a bandwidth of 50 GB/s per direction. Additionally, on one of the four APUs in the system, a PCIe 4.0 connects the compute node to the near-node storage, with a bandwidth of 8 GB/s per direction.

While the Infinity Fabric interconnect on MI300A system shared similarities with the previous generation, featured on MI250X systems, the Infinity Fabric mesh on MI300A system is significantly simpler. First, on MI300A system, each APU is directly connected to all peers, whereas on MI250X system, a GPU may need up to two hops to reach other GPUs. In addition, while IF links on MI250X systems have varying bandwidth values for various pairs of GPUs [26, 11], on MI300A systems, all pairs of APUs are connected with the same link bandwidth.

2.3 Memory Management on Multi-APU Nodes

From the user perspective, a multi-APU node is a Cache Coherent NUMA (non-Uniform memory Access) system, where each APU is exposed as a NUMA node, grouping the GPU, the 24 CPU cores, and the 128 GB of HBM3 memory. *Node-level memory coherence* is managed transparently for the programmer, so that updates to one APU's memory by any processor (either CPU or GPU) are reflected in all cached copies of the data. Depending on whether the data is accessed by GPU or CPU, memory coherence may be ensured at either hardware or software level. Coherence between each CPU with the rest of the system is achieved through the use of probe filters at hardware level, while coherence between GPU and other GPUs and CPUs in the system is ensured through software support [30].

On each APU, CPU and GPU maintain their respective page tables, similar to the previous generation of AMD MI250X GPUs. The GPU page table is distinct from the CPU page table despite sharing the same physical memory space. When the GPU performs a memory operation on a virtual address that is not mapped in the GPU page table, a page fault occurs. In general, a page fault in a GPU kernel terminates the kernel with an error. On AMD MI300A, such a failed memory access will be replayed by leveraging

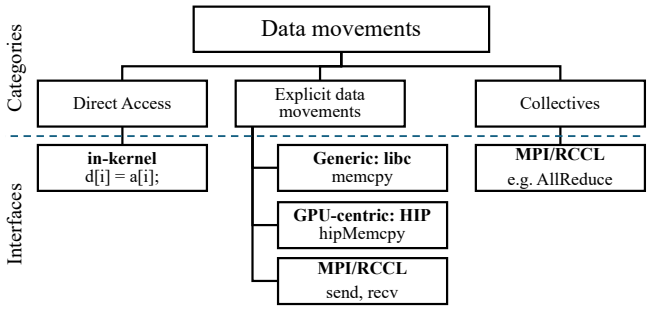


Figure 2: A taxonomy of communication on multi-APU systems, associated data movement categories, programming interfaces and libraries.

a hardware feature called *XNACK*, which can be enabled by setting the environment variable `HSA_XNACK=1`. When XNACK is enabled, together with Linux's Heterogeneous Memory Management (HMM) system, GPU kernels can access system-allocated memory allocated with, e.g., `malloc`. This approach contrasts with the one taken with the Nvidia Grace Hopper superchip [27], where two distinct physical memory regions, CPU and GPU memory, can be managed with a single system-wide page table, without the need for HMM.

3 A Taxonomy of multi-APU Communication

In this section, we present a taxonomy of communication on emerging multi-APU nodes, represented by the El Capitan supercomputer. Figure 2 presents the taxonomy, including data movement approaches and available programming interfaces.

3.1 Direct Access

In GPU applications, memory accesses are performed within GPU kernel code, using load and store instructions. Such access offers the lowest latency and highest bandwidth when performed on local GPU memory, physically residing on the GPU where it is accessed from. However, modern GPU hardware and software provide the ability to access data located remotely, either in the host's physical memory or in another GPU's physical memory. Direct access provides the highest level of granularity compared to other data movement approaches, as only the data being accessed is transferred to the accessing processor. This can be beneficial, for example, in applications with complex communication patterns, where the exact extent of the data to be accessed is unknown at runtime, e.g. in graph processing application. However, as the data remain remotely-resident, direct access is not suitable for applications with well-known communication patterns, or performing repeated accesses.

3.2 Explicit Data Movement

Explicit data movement refers to an approach where data are explicitly copied or moved to the memory attached to the processor where it is used. GPU applications are heavily reliant on this principle for several reasons. First, as previous generations of GPUs did not support direct access to remote-located memory, data were necessarily resident in local GPU memory before a GPU kernel

could be executed. Second, explicit data movement is advised for performance considerations, as direct data access to remote memory is limited by the available bandwidth of the interconnect, with a theoretical limit 128 GB/s for MI300A, which is dramatically lower than the bandwidth of the local high bandwidth GPU memory, with a theoretical value of 5.6 TB/s on MI300A. In addition, GPUs feature hardware units that are specialized in data copy and do not use the compute capabilities of the Compute Units. These hardware units, referred to as SDMA engines (System Direct Memory Access engines) in AMD's terminology, can perform copy operations in parallel with kernel execution. Therefore, explicit data movement offers the opportunity to overlap communication and computation at a high level and for large memory regions instead of relying solely on the instruction pipelining abilities of GPU compute units.

3.2.1 C standard library's memcpy. The C standard library provides a memory copy function, memcpy, which performs a copy between two buffers. The actual implementation of memcpy is compiler-dependent. While the GNU C Compiler implements memcpy as a while loop performing a series of C load-and-assign operations, Clang's implementation of memcpy is platform-dependent.

3.2.2 GPU-centric APIs. HIP [3] runtime exposes APIs to perform explicit data movement, namely hipMemcpy, originally designed for host-to-device and device-to-host data movement, but now supports any type of data movement, and hipMemcpyPeer, which is dedicated to inter-GPU peer-to-peer data movement. On AMD hardware, hipMemcpyPeer is a thin wrapper around hipMemcpy [1]. We note here that HIP APIs rely on lower-level APIs, HSA [2] memory management APIs, to execute actual copy operations. Directly using such lower-level APIs offers the opportunity for hardware-tailored performance optimizations. However, as their portability is limited and their use in real-world applications remains marginal, they fall out of the scope of this work.

3.2.3 Multi-process Point-to-Point Communication. The use of Message Passing Interface (MPI) is ubiquitous in HPC applications, to distribute computations across several processes. The MPI standard defines routines to send and receive data across processes, `MPI_Send` and `MPI_Recv`, respectively. From the application perspective, those routines are semantically equivalent to an explicit data copy operated with, e.g., memcpy. A notable difference is that instead of copying data between buffers allocated by the same process, data are transferred between buffers belonging to different processes, located on distinct processes on potentially distinct compute nodes. Naturally, such operation is more complex than for intra-process explicit data movement.

While MPI is the *de facto* standard in HPC, AMD's ROCm Collective Communication Library (RCCL), which is a collective communication library specialized for GPU communication, also provides point-to-point communication routines, similar to the ones defined in the MPI standard. For single-process applications, RCCL can be utilized by itself without any other dependency, while for multi-process applications, RCCL is used in conjunction with MPI.

3.3 Collective Communication

Collective operations are a category of communication that involves all communicating endpoints. Collective operations often consist of several communication processes. Such a communication pattern is heavily relied upon in HPC and distributed machine learning workloads. While collective communication routines are underlyingly implemented with a series of point-to-point communications, they also include computations on the collaborating processors. In HPC, the use of MPI is ubiquitous for collective communication. However, the RCCL library also appears as a strong alternative, which offers comparable capabilities, and is specialized for GPU-GPU communications, this library is notably used in distributed machine learning applications.

4 Methodology

In this section, we describe the benchmarking design for characterizing each communication category in the taxonomy in Section 3. We also introduce two real-world applications used for case study and the testbed environments.

4.1 Benchmark Design

4.1.1 Direct Access. To evaluate the performance of direct data access on quad-MI300A system, we use a GPU variant of the STREAM benchmark, where two buffers are allocated and initialized in one APU's memory, using the hipMalloc allocator. A GPU kernel is then executed on another APU. This kernel reads data from and stores data to the buffers allocated on the peer APU. By measuring the bandwidth of the copy operation, we can evaluate the achievable copy bandwidth over an APU-APU Infinity Fabric link.

To evaluate the unidirectional bandwidth of the Infinity Fabric link between two APUs, we rely on the hipMemcpy API. In the default configuration, this API uses dedicated GPU hardware units to perform a copy operation and does not rely on GPU kernels. By setting the environment variable `HSA_ENABLE_SDMA=0`, we override this behaviour and force the hipMemcpy API to use highly optimized copy kernels, referred to as "*blit*" kernels, which directly implement a copy operation using load-store instructions executed on the GPU's compute units [6], instead of relying on dedicated hardware units. We use this approach to measure the peer-to-peer copy bandwidth, achievable with direct data access. In addition, with a pointer-chasing approach we measure the latency of local memory accesses and remote memory accesses over Infinity Fabric.

4.1.2 Explicit Data Movements. We develop a bandwidth measurement benchmark to evaluate the performance of explicit data movement APIs. We construct our benchmark with Google's benchmark framework library [13]. We define three phases: allocation, first-touch, and data movement. The benchmark measures the bandwidth of the copy operation in the data movement phase; a warm-up phase is included, and measurements are repeated 10 times. The benchmark allows changing the underlying interface used in each of the three phases. For the allocation phase, four allocators can be used: the system allocator `malloc`, the HIP GPU-centric allocator `hipMalloc`, the HIP managed memory allocator `hipMallocManaged`, and the HIP host-memory allocator. The second phase, first-touch, refers to the initialization, which can be done either by a CPU thread or the GPU. For CPU first-touch, we use libc's `memset` function. The data movement phase uses either the `memcpy` function or HIP `hipMemcpy` API call. We observed that both `hipMemset` and `hipMemcpy` fail with "invalid argument" when called

on a non-HIP allocated buffer. For `hipMemset`, we work around the issue by implementing a simple GPU kernel to initialize the memory. For `hipMemcpy`, registering the malloc-allocated memory with `hipHostRegister` allows calling `hipMemcpy` on the allocation. To control the location of the source and destination buffers, we set the locality of CPU threads with `numa_run_on_node`, which constrains a CPU thread to execute on a specific NUMA node, that is, APU. For HIP-related APIs, we ensure execution on the desired GPU by using `hipSetDevice`.

4.1.3 MPI/RCCL Point-to-Point Communication. We use the OSU micro-benchmark suite (OMB) [22] to evaluate the bandwidth and latency of point-to-point send and receive operations. We compare MPI routines, widely used in HPC applications, with routines provided as part of ROCm Communication Collectives Library (RCCL), which are specialized for GPU-GPU communication. Underlyingly, these routines use system-specific APIs, similar to hipMemcpy or memcpy, to perform the actual data movement. We use the benchmarks `osu_bw` and `osu_lat` for MPI, `osu_xccl_bw` and `osu_xccl_lat` for RCCL. The latency benchmarks execute a ping-pong latency measurement. The bandwidth benchmark initiates a series of fixed-size back-to-back MPI messages with `MPI_ISend` from a sender process and receives those messages on another process using matching `MPI_Recv` operations. The wall-clock time of 10,000 `MPI_ISend` and the corresponding `MPI_Recv` operations are measured.

The MPI implementation in use, Cray MPICH, dynamically changes its underlying communication paths depending on message sizes, e.g., it uses shared memory CPU buffers for intra-node communication of messages no larger than 1024 bytes and uses SDMA-accelerated direct peer-to-peer GPU communication for larger messages. Therefore, we use two configurations for MPI. First, we enforce direct peer-to-peer GPU-GPU inter-process communication by setting `MPICH_GPU_IPC_THRESHOLD` to `0`, denoted as *GPU direct*. Second, we enable CPU staging by setting `MPICH_GPU_IPC_ENABLED` to `0`, denoted as *CPU staging*. In addition, we ensure that GPU-aware capabilities are enabled in the MPI implementation by setting `MPICH_GPU_SUPPORT_ENABLED` to `1`. We further evaluate several combinations of allocators for the source and destination buffers to evaluate the ability of the MPI implementation to map copy operations to actual hardware capabilities, in various circumstances.

4.1.4 MPI/RCCL Collective Communications. We measure the latency of common collective operations using both the RCCL and MPI benchmarks provided as part of the OSU micro-benchmark suite [22], for various message sizes and numbers of GPUs.

4.2 HPC Applications

We select two real-world GPU-accelerated HPC applications – *Quicksilver* [16] and *Cloverleaf* [20], for the use case study. Quicksilver is a dynamic Monte Carlo particle transport code that represents the Mercury workload, this workload exhibits unbalanced communication and irregular access pattern. Cloverleaf is a Lagrangian-Eulerian hydrodynamics application, which exhibits a balanced communication pattern with regular access pattern. Both applications have GPU kernels implemented in HIP and rely on MPI for inter-process communication. To demonstrate the effectiveness of communication optimization on multi-APU systems, we adapt their allocation sites and communication interfaces only, and compare them with the original version. In our evaluation, Quicksilver used the CORAL2 Problems 1 and 2 with 2M to 42M particles while CloverLeaf used the bm2028_short problem with 61440×30720 cells.

Table 1: Main node characteristics of testbeds.

	MI300A Testbed	MI250X Testbed
NUMA domains	4 (one per APU)	4 (partitioned CPU memory)
CPU	24-core AMD Zen 4	64-core AMD Trento EPYC
GPU	6 XCDs exposed as 1 GPU	2 GCDs exposed as 2 GPUs
Infinity Fabric	512 Gb/s links	288 Gb/s and 400 Gb/s links
Memory	128 GB HBM3	128 GB DDR4, 128 GB HBM2

4.3 Testbed

We use two testbeds in our study, namely an MI300A system as our main testbed and an MI250X testbed for comparison. Table 1 summarizes their main node characteristics. On the MI300A testbed, each node is equipped with four AMD MI300A APUs. We use the Cray Programming Environment 24.11 and the `hipcc` compilation toolchain from ROCm 6.2.1. For point-to-point and collective communication experiments, we use RCCL 2.20.5 and Cray MPICH 8.1.31 (based on ANL MPICH 8.4a2).

5 Multi-APU Single-Process Communication

5.1 Direct Kernel-level Access

We use a GPU-variant of the STREAM benchmark to evaluate the performance of in-kernel direct remote access. In this benchmark, the arrays are allocated on one APU, using hipMalloc, and a GPU copy kernel is executed on another APU, reading from a peer GPU, and writing back to it. Figure 4 presents the results of the copy kernel of the GPU variant of the STREAM benchmark, with a kernel executed on APU0 and data located on APU1, APU2, or APU3, for array sizes from 2 MB to 8 GB. Across all data placements, we observe a bandwidth of 103-104 GB/s, those homogeneous values are consistent with the node topology, where all APUs are directly connected to all other APUs with the same link bandwidth. The measured bandwidth represents 81% of the theoretical bandwidth of the Infinity Fabric link. We compare the results with the same benchmark executed on MI250X GPUs, with two data placements, based on the non-balanced node topology on MI250X, where Infinity Fabric bandwidth is 50 GB/s for GCD0-GCD2, and 100 GB/s for GCD0-GCD6. In this situation, we reach 82% and 81% of the theoretical link bandwidth for GCD0-GCD2 and GCD0-GCD6, respectively. The values of link utilization are similar to values obtained on the newer MI300A.

We measure the latency of direct access to local and remote memory using a pointer-chasing approach, adapted from Google's multi-chase benchmark. For local access, memory is allocated with hipMalloc on the same APU as the pointer-chasing kernel; for remote access, memory is allocated with hipMalloc on a neighbour APU to the one executing the pointer-chasing kernel. Figure 3 presents the results of this pointer-chasing approach for CPU and GPU, with increasing data size. The latency for local access to HBM memory is 240 ns for CPU, and 346 ns for GPU. For remote data

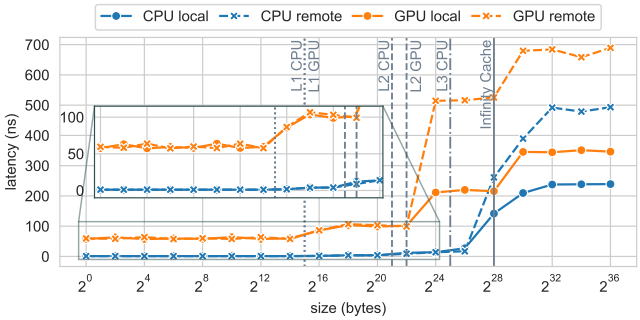


Figure 3: CPU and GPU latency of direct access to local memory, or remote memory (dashed line), located on a neighboring APU. Cache sizes are represented as vertical lines.

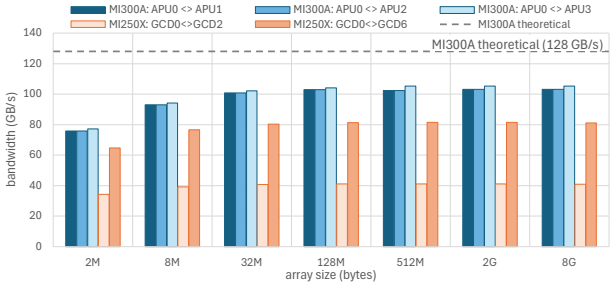


Figure 4: Bidirectional direct access bandwidth obtained with the STREAM Copy kernel, executed on APU0 with data placed on peer APUs. Results on AMD MI250X are obtained by executing on GCD0 with data placed on neighbor GCDs.

access, the latency increases to 500 ns for CPU access, and 690 ns for GPU access.

Observation 1: Direct GPU kernel access to local and remote APU's HBM has 500 ns and 690 ns latency, higher than CPU's direct access of 240 ns and 346 ns latency. GPU kernel can directly access data on remote APU at 103 GB/s.

5.2 Explicit Data Movement

In this section, we evaluate the performance of explicit data movement APIs. For this purpose, we compare hipMemcpy and memcpy operations. On the previous generation of AMD GPU, AMD MI250X, the SDMA engines were documented to be unable to fully utilize the Infinity Fabric link bandwidth due to their initial design being optimized for communication on PCIe speeds [5].

5.2.1 Low Transfer Sizes. For hipMalloc-allocated memory, Figure 5 presents the latency of memcpy and hipMemcpy operations for low transfer sizes. We observe that memcpy outperforms hipMemcpy for low transfer sizes, up to 512 KB. This is due to the nature of memcpy, which is implemented as a series of load and store instructions, which can operate on cache levels of the system. Therefore, the measured latency is below 100 ns for transfers up to 16 KB. In contrast, hipMemcpy operations are more complex, as they are delegated to the Heterogeneous System Architecture

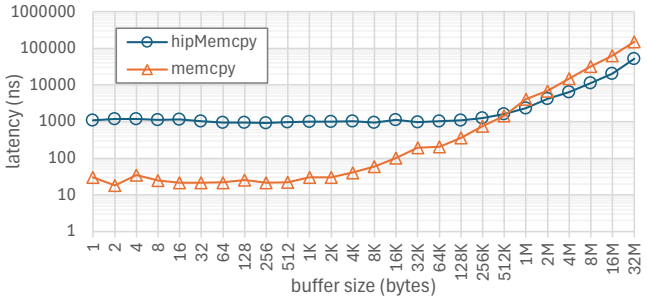


Figure 5: Latency of hipMemcpy and memcpy for an APU-APU transfer on hipMalloc-allocated buffers with CPU first-touch.

(HSA) runtime, resulting in higher latency. For transfer sizes between 1 byte and 128 KB, a hipMemcpy call represents 1 μs.

Observation 2: For transfer sizes below 512 KB, memcpy exhibit lower latency compared to hipMemcpy, due to its ability to leverage the various cache levels in the system.

5.2.2 CPU-side memcpy. We measure the bandwidth of CPU-side memcpy operation to copy large buffer from APU0 to APU1, we evaluate various allocators, both the system allocator malloc, and the HIP allocators hipMalloc, hipHostMalloc, and hipMallocManaged. We use the compiler-implemented memcpy.

In this experiment, we ensure that physical memory is allocated to both source and destination buffers. For this purpose, we initialize both buffers with an arbitrary value. For the CPU-side first-touch, we use memset, which performs initialization of each of the buffer's elements within a loop. For GPU-side initialization, the hipMemset API cannot be called on a memory region untracked by the GPU driver, e.g., allocated with malloc, resulting in an invalid argument error; instead, we use a GPU kernel for this purpose.

Figure 6 presents the achieved bandwidth for memcpy, when one thread performs a copy operation from APU0 to APU1, with a buffer size of 8 GB. For all allocators and first-touch locations, the copy bandwidth is below 20 GB/s. We suggest that this low bandwidth compared to the theoretical limit of 128 GB/s is due to the nature of the memcpy implementation, which relies on a loop to copy memory from the source buffer to the destination buffer, using load and store instructions. This implementation only leverages one CPU core and, therefore, cannot utilize the full bandwidth offered by the link between APUs. For hipMalloc and malloc allocators, with GPU first-touch, copy bandwidth is significantly lower than for other allocator/first-touch combinations, on the order of 10 GB/s.

5.2.3 GPU-centric hipMemcpy. The hipMemcpy API, provided as part of the ROCm runtime, is designed for data copy in a heterogeneous CPU-GPU system. We measure the copy bandwidth for a data copy from APU0 to APU1, performed with hipMemcpy between two 8 GB buffers. We use the same initialization strategy as for memcpy, where either CPU or GPU performs the first-touch.

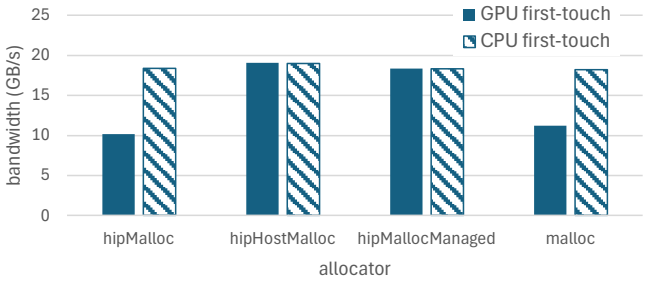


Figure 6: The impact of allocators and first-touch on the maximum bandwidth (GB/s) achieved by memcpy.

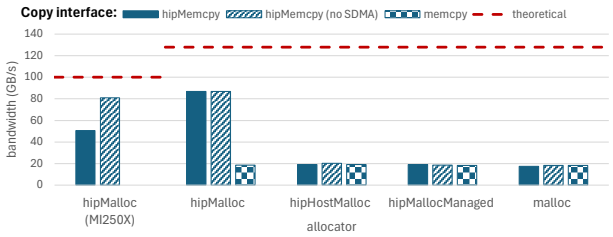


Figure 7: The achieved explicit data copy bandwidth from using hipMemcpy or memcpy for copying data between MI300A APUs.

For hipMemcpy, data movement is performed using System Direct Memory Access (SDMA) engines by default, which are hardware units for copying data across the system and bypassing compute units, enabling overlap of the copy operation with kernel execution. On MI250X GPUs systems, however, the use of SDMA engines causes under-utilization of the GPU-GPU link.

Figure 7 presents the achieved bandwidth for explicit data copy between two APUs with hipMemcpy. We observe that hipMemcpy only exhibits the highest copy bandwidth for hipMalloc-allocated buffers, with 90 GB/s. Other allocators are not able to fully leverage the bandwidth of the link and only reach values of bandwidth comparable to those obtained with memcpy. Upon inspection of the hipMemcpy implementation code [1], we suggest that these copy operations fall back on standard memcpy calls, executed as single-threaded CPU-side copies.

In addition, we do not observe any significant difference in bandwidth comparing copy operations using SDMA engines, which is the default behavior, or using direct GPU copy kernels by explicitly disabling SDMA engines. This contrasts with previous generations of AMD GPUs, embodied by the MI250X GPU, where copy operations that rely on SDMA engines are not able to fully leverage the available Infinity Fabric bandwidth [26]. This is due to the SDMA engines on this generation being tuned for PCIe speeds and, therefore, cannot leverage the full link bandwidth offered by Infinity Fabric [5]. Our results demonstrate that this limitation has been lifted on AMD MI300A, where copy operations using SDMA engines can reach the same level of bandwidth as for direct copy kernels.

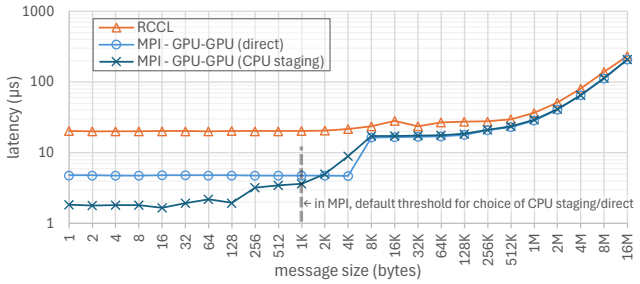


Figure 8: The latency of MPI and RCCL for point-to-point GPU-GPU communication at increased message sizes.

Observation 3: For inter-APU copy operations above 512 KB, hipMemcpy outperforms memcpy, due to its ability to offload the operation to SDMA engines or GPU copy kernels, thereby enabling the use of the full Infinity Fabric bandwidth.

6 Multi-APU Multi-Process Communication

In this section, we compare point-to-point and collective communication routines between MPI and RCCL. Moreover, we study the impact of different memory allocators on leveraging the Infinity Fabric link bandwidth.

6.1 Point-to-Point Communication

6.1.1 Latency. Our latency measurement results indicate that MPI point-to-point routines with CPU staging achieve the lowest latency. Figure 8 presents the inter-APU ping-pong latency measured on hipMalloc-allocated communication buffers at various message sizes. For small message size below 128 bytes, MPI routines with CPU staging have a latency as low as 1.9 μs, while direct peer-to-peer MPI communication exhibits a 4.8 μs latency. In contrast, the latency of RCCL is significantly higher than that of MPI for small messages, with a lowest latency of 20 μs, that is, 10× higher than MPI routines. For direct MPI GPU-GPU communication, we observe a jump in the measured latency when increasing message size from 4 KB to 8 KB. We suggest that this jump indicates a change of behavior in the MPI implementation for messages above 4 KB. However, due to the proprietary nature of the implementation, we were not able to pinpoint the exact cause. Overall, compared to the latency of direct GPU kernel access and `memcpy` in Section 5.2, point-to-point communication has a significantly higher latency for small messages. This increased latency is induced by the complex nature of a point-to-point operation, where not only data must be copied, but also expensive inter-process communication is performed.

6.1.2 Bandwidth. Figure 9 presents the bandwidth of point-to-point routines in MPI and RCCL with hipMalloc-allocated buffers between APUs. We observe that for message sizes above 8 KB, the bandwidth of RCCL matches the one obtained with direct GPU-GPU MPI communication. We observe that the CPU staging option in MPI outperforms the peer-to-peer GPU-GPU communication for message sizes of 1024 KB or smaller. This is due to the overhead of requesting transfer with SDMA engines in the case of direct peer-to-peer GPU-GPU communication, compared to the low overhead of performing a CPU-side copy between two APUs. As shown in

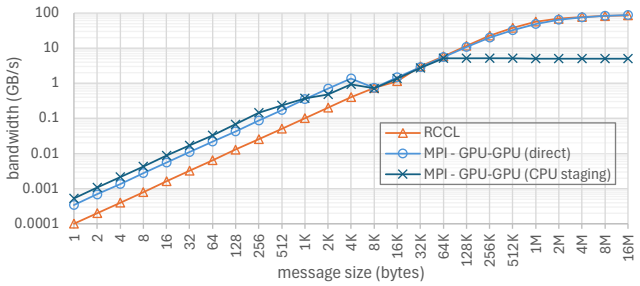
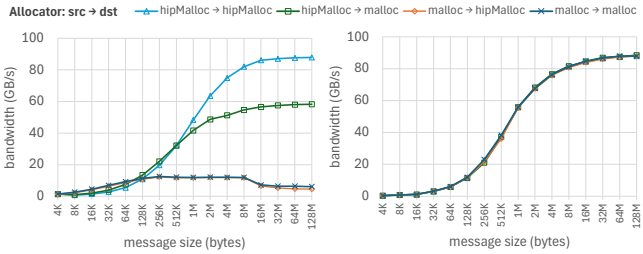


Figure 9: The measured bandwidth of MPI direct GPU-GPU communication, MPI GPU-GPU communication with CPU staging, and RCCL. Destination and source buffers are allocated with hipMalloc.



(a) MPI **(b) RCCL**

Figure 10: The impact of different allocators for the source and destination buffers on point-to-point bandwidth with SDMA enabled.

Figure 9, RCCL point-to-point communication routines achieved a maximum bandwidth of 88 GB/s, which is comparable to the bandwidth measured with hipMemcpy APIs.

6.1.3 Impact of Memory Allocators. We evaluate the impact of the allocator on point-to-point bandwidth in MPI and RCCL. For MPI, Figure 10a shows that when the source buffer is allocated with malloc, the maximum bandwidth measured for MPI send/receive operation is 11.7 GB/s. This is comparable to the values obtained with a single-threaded memcpy operation, presented in Section 5.2. When both buffers are allocated with hipMalloc, the measured bandwidth is 82 GB/s, which matches the values reported in Section 5.2, obtained with hipMemcpy. This indicates that the GPU-aware MPI implementation can efficiently leverage the available inter-APU bandwidth, when using hipMalloc for both source and destination buffers. In contrast, when the source buffer is allocated with hipMalloc and the destination buffer is allocated with malloc, the bandwidth drops to 54 GB/s. We hypothesize that in such scenario, both GPU-only and system page tables are involved in the copy operation. This causes significant overhead. For RCCL, the bandwidth measured under point-to-point routines, presented in Figure 10b, appears to be insensitive to the choice of allocator. This highlights the ability of RCCL to map the execution of point-to-point routines to the most efficient hardware interface.

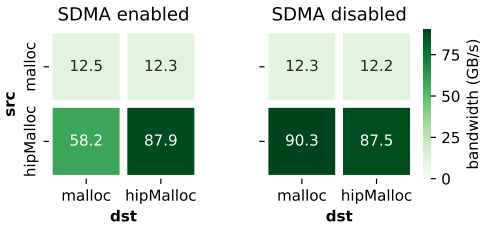


Figure 11: The peak MPI p2p bandwidth using different allocators for source and destination buffers and SDMA settings.

Observation 4: RCCL point-to-point routines can efficiently leverage the full Infinity Fabric bandwidth, independent of the choice of allocator. MPI point-to-point routines only achieve the full Infinity Fabric bandwidth when both source and destination buffers are allocated with hipMalloc.

6.1.4 The impact of SDMA engines. We further evaluate the impact of SDMA engines on the bandwidth of MPI send/receive operations between two APUs. Our results in Section 5.2 demonstrate that SDMA engines on MI300A APUs can fully utilize the Infinity Fabric link. Therefore, for MPI and RCCL, we expect similar bandwidth when disabling SDMA engines. We set the environment variable `HSA_ENABLE_SDMA=0` to measure the bandwidth with disabled SDMA engines. When SDMA is disabled, data movement relying on the HSA runtime will use direct GPU-executed copy kernels to perform data copy. We present the measured bandwidth in Figure 11, when using either hipMalloc or malloc to allocate source and destination buffers.

In MPI, when the source buffer is allocated with malloc, the state of SDMA engines does not impact the bandwidth, measured at 12 GB/s. This is expected, as the copy mechanism in this situation appears to not rely on GPU's HSA runtime but instead on CPU-side mempcy, which never relies on SDMA engines. When the source buffer is allocated with hipMalloc, and the destination buffer is allocated with malloc, the bandwidth measured with SDMA engines disabled is 90.3 GB/s. This is significantly higher than the 58.2 GB/s bandwidth measured with SDMA engines enabled. For RCCL, we conduct the identical measurements. Our results indicate that SDMA engine state has little impact on RCCL point-to-point bandwidth.

Figure 12 presents the point-to-point bandwidth measured with different SDMA settings. The results for a source buffer allocated with malloc are omitted, as they were not influenced by the state of SDMA engines in our experiments. When copying from a hipMalloc-allocated buffer to a malloc-allocated buffer, disabling SDMA engines brings a significant bandwidth improvement for all message sizes. For the largest message size, the bandwidth increases from 58.2 GB/s to 90.3 GB/s. When both source and destination buffers are allocated with hipMalloc, the bandwidth evolution exhibits a different pattern. For message sizes below 1 MB, disabling SDMA engines achieves a higher bandwidth than with SDMA enabled. In contrast, above 1 MB, disabling SDMA has a detrimental effect on bandwidth. However, at the largest message size, the same bandwidth is observed for either SDMA state.

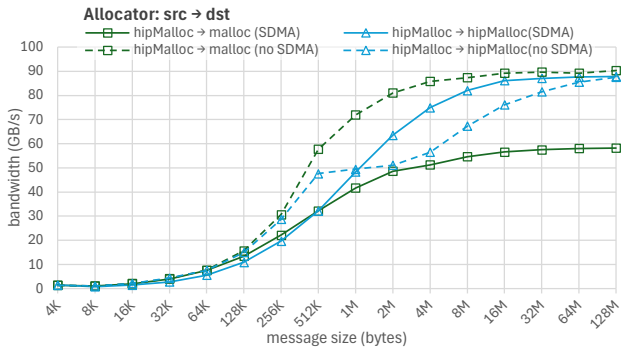


Figure 12: Bandwidth of MPI back-to-back send operations, measured with OSU micro-benchmark, for various allocators for source and destination buffers. Dashed lines indicate that SDMA engines are explicitly disabled.

Observation 5: Bandwidth of MPI point-to-point routines depends on SDMA engine status and allocator choice, with hipMalloc yielding highest bandwidth; RCCL fully utilizes Infinity Fabric link in all evaluated circumstances.

6.2 Collective Communication

We first investigate the scalability of one widely-used collective operation, AllReduce, at increased message sizes, in Figure 14. For all message sizes up to 4 KB, MPI outperforms RCCL. However, beyond 4 KB, RCCL routines start to exhibit lower latency than MPI. Moreover, RCCL latency scales linearly with the message size, as plotted in the dashed line in Figure 14. In MPI, such a linear trend is only observed above 256 KB. This indicates that the runtime of MPI is less predictable than RCCL, likely due a change in the underlying communication interface used by the MPI implementation, depending on buffer size. For instance, by default MPI uses CPU-staging for messages smaller than 1024 bytes, as discussed in Section 6.1.

Figure 13 presents the latency of RCCL and MPI collectives with two to four APUs participating in the collective. We make the same observation as for AllReduce, where for small messages (Figure 13a), MPI outperforms RCCL. This is expected, as we observed that RCCL point-to-point communication routines exhibited a baseline ~20 μs ping-pong latency. For larger messages (e.g., 16 MB in Figure 13b), RCCL collectives outperform MPI routines. As demonstrated earlier, RCCL implementations can leverage the bandwidth of Infinity Fabric links more efficiently than MPI, resulting in higher performance for bandwidth-bound communication, e.g., large messages. ReduceScatter is commonly used in distributed machine learning workloads. For large message sizes, RCCL exhibits a significant advantage of 20-38× speedup over MPI ReduceScatter.

Observation 6: For messages larger than 4 KB, RCCL collectives lead to 5 – 38× lower latency than MPI. For messages smaller than 1024 bytes, MPI collectives have the lowest latency.

7 HPC Applications

In this section, we present two case studies in real-world HPC applications to demonstrate the strategy of optimizing multi-APU communication as identified according to our characterization study.

7.1 QuickSilver

QuickSilver is a multi-process MPI application for dynamic Monte Carlo particle transport problems. As a Monte Carlo code, it uses a large number of particles in simulations, and these particles are spread across the whole domain, which is divided across MPI processes. Thus, communicating particles across processes becomes one time-consuming task of the application. For this purpose, a class `MC_Particle_Buffer` is used, to hold information on particle buffers, and to expose methods to control the exchange of particle data across processors. In the original version, these particle communication buffers are allocated using the system allocator `malloc` and are exchanged using MPI point-to-point routine `MPI_Isend`.

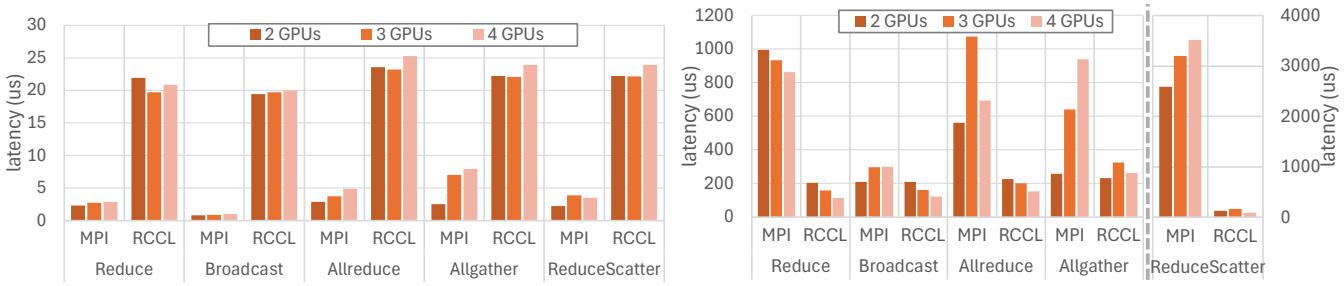
Our profiling results of QuickSilver communication pattern indicate that many small messages are used for communication. As identified in Section 6, RCCL point-to-point communication routines have higher latency than MPI routines for small messages, thus, they will not be used for point-to-point communication. Also, from the characterization, disabled SDMA has positive impacts on the bandwidth of MPI point-to-point communication on all message sizes. Therefore, we disable the SDMA engines for optimization. For experiments, we compile Quicksilver to produce a binary supporting any XNACK state, and change the environment variable `HSA_XNACK` at runtime. Finally, we adapt the allocator for communication buffers from `malloc` to `hipMalloc` because our characterization study indicated that malloc-allocated buffers do not reach the maximum link bandwidth in MPI point-to-point routines. In Quicksilver, this is achieved by adapting the `Allocate` method of `MC_Particle_Buffer`.

Figure 15 details the runtime of the six Quicksilver test cases, evaluated with XNACK enabled or disabled, using either hipMalloc or malloc for the allocation of buffers used in point-to-point MPI routines. With XNACK enabled, for any fixed problem, the runtime in Quicksilver is insensitive to the selected allocator for allocating communication buffers. However, when XNACK is disabled, we observe the speedup from 5% to 11% on the end-to-end execution time. This confirms that the selected optimization is effective for further running Quicksilver simulations on multi-APU systems.

We demonstrate in Section 6 that only hipMalloc-allocated buffers can be communicated at full link bandwidth when using MPI. However, when replacing malloc by hipMalloc, the communication time reported by Quicksilver decreases only for the largest test case, Problem 1 with 42M particles, from 6.8 s down to 5.9 s. The reason is that for low transfer sizes, the benefit of using hipMalloc-allocated buffers for communication is outweighed by the overhead of transferring data to those buffers before the actual communication.

7.2 CloverLeaf

Cloverleaf is a Lagrangian-Eulerian hydrodynamics application. Solvers in such applications heavily rely on send and receive operations. This application exhibits balanced communication, with



(a) 4 bytes (b) 16 MB

Figure 13: The measured latency of MPI and RCCL collective operations for 4 bytes and 16 MB messages, with 2 to 4 participating MI300 APUs. ReduceScatter for 16 MB messages uses a separate y-axis.

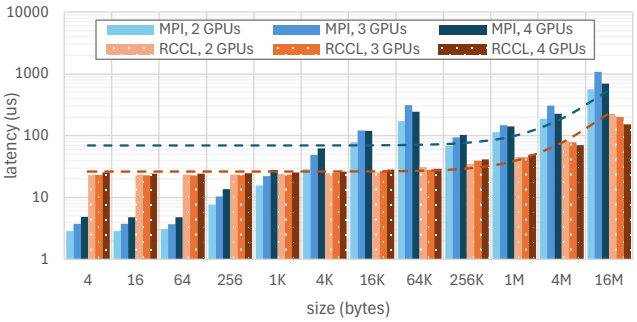


Figure 14: Latency of AllReduce collective operation, with 2 to 4 APUs in the same node participating in the collective. Linear trends are plotted as dashed lines.

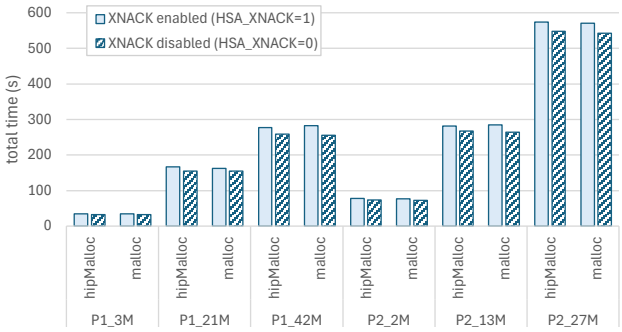
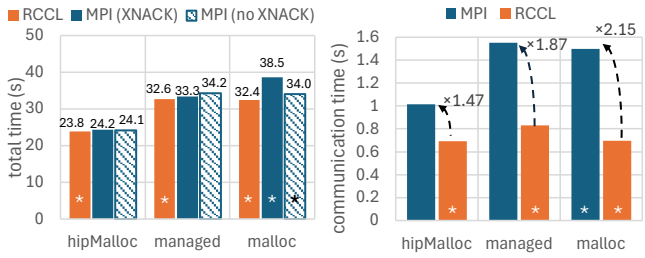


Figure 15: End-to-end runtime measured in Quicksilver for all input problems, comparing the impact of XNACK settings and allocators.

regular memory access pattern. The baseline HIP implementation supports managed memory, allocated with hipMallocManaged. We preserve MPI calls for smaller message communication operations, such as process synchronization. We add support for system-allocated memory, where the allocation with malloc must be combined with hipHostRegister, to allow hipMemcpy operations. We adapt to use RCCL point-to-point routines for send and receive operations, implemented as part of the `clover_exchange` function. The application is compiled with generic XNACK support. However, executing the RCCL implementation with XNACK disabled



(a) Total time (b) Communication time

Figure 16: The total runtime (a) and communication time (b) of CloverLeaf. "*" indicates our adapted versions, "×" indicates speedup over MPI.

caused the program to exit due to errors in the RCCL internal code. Thus, the results are obtained with XNACK enabled.

Figure 16a presents the end-to-end runtime of CloverLeaf using MPI or RCCL communication interface and various allocators. For the original implementation, the default hipMalloc-allocated version provides the lowest end-to-end runtime compared to hipMallocManaged and the malloc system allocator. This is consistent with our benchmarking results, presented in Section 5.2, which show that for MPI, only hipMalloc-allocated buffers can be exchanged between APUs at high bandwidth. We show that the original MPI version has 15% higher runtime when using malloc+hipHostRegister, compared to the hipMallocManaged version.

To understand how communication optimizations affect the end-to-end runtime, we utilize the internal CloverLeaf timers to quantify improvements in communication time. These timers measure the execution time of the MPI Halo Exchange function (`clover_exchange`), which implements the core of the application's data movement. Figure 16b presents the communication time in Cloverleaf, for the three evaluated allocators. The reported time is averaged over five trails. We observe that the communication time in the MPI implementation is highly sensitive to the allocator, with values of 1.01 s for hipMalloc, 1.50 s for malloc, and 1.55 s for hipMallocManaged. This is consistent with our characterization results for MPI explicit data transfers. However, the 0.54 s communication time difference observed between the best-performing hipMalloc and hipMallocManaged in the MPI version cannot by itself explain the 9.1 s difference in end-to-end runtime. This observation highlights that while the choice of allocators impacts the communication time

between APUs, other factors must be considered, including GPU kernel performance and performance of copy operations within a single APU's physical memory space, with e.g., hipMemcpy or memcpy. For RCCL, the communication time exhibits limited variability across the three allocators, with 0.69 s for hipMalloc and malloc, and 0.83 s for hipMallocManaged. This is also consistent with our benchmarking results and demonstrates how RCCL can achieve efficient communication, with limited impact from the allocator used for the communicated buffers.

In all tested cases in CloverLeaf, the communication-optimized implementation outperforms the original version, with a 1.5× speedup for hipMalloc, 1.9× for hipMallocManaged, and 2.2× for malloc. These results highlight the inefficiency of using non-hipMalloc buffers for MPI point-to-point communication routines. Furthermore, this demonstrates how the use of RCCL enables developers to opt for best-performing allocator for their respective use case or depending on application-specific constraints, while still achieving the highest level of bandwidth for inter-APU communication.

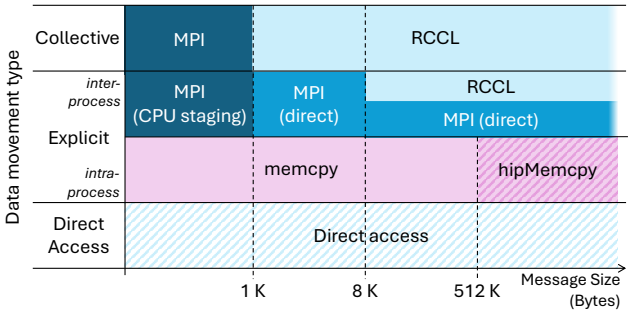


Figure 17: A summary of best-performing interface for inter-APU communication at various message sizes and data movement types, targeting to high bandwidth for explicit and direct access and low latency for collective operations. Assume buffers are allocated with hipMalloc.

8 Discussions

Allocators. Depending on the data movement interface in use, the choice of allocator might affect the performance of data movement. In our experiments, hipMalloc is the only allocator for which the highest bandwidth of the Infinity Fabric interconnect could be reached consistently across all scenarios. In details, for hipMemcpy, using hipMalloc is required and MPI point-to-point routines are only able to achieve the maximum bandwidth for buffers allocated with hipMalloc. This is due to the MPI runtime delegating the copy operations performed on hipMalloc-allocated buffers to GPU hardware, either with SDMA engines or using GPU copy kernel, therefore achieving maximum bandwidth. For other choices of allocators, RCCL is the only programming interface that can utilize the full Infinity Fabric bandwidth, independent of the allocator, appearing as a solution to operate high-bandwidth data movement when the choice of allocator is constrained. Other factors, such as allocation time, which is higher with hipMalloc than with malloc, might be taken into account.

Programming Interfaces. For each data movement scenario presented in the taxonomy Section 3, Figure 17 presents the optimal interface depending on various message sizes. In general, for explicit and collective data movement at small messages, CPU-centric interfaces, namely memcpy and MPI with CPU-staging, provide the highest performance. This is a consequence of the high latency observed for GPU-centric interfaces, which is detrimental on small message sizes, which are typically latency-bound. However, those GPU-centric interfaces are able to leverage GPU hardware to perform data movement, namely SDMA engines and GPU copy kernels. They can therefore leverage the full Infinity Fabric bandwidth, making them suitable for larger message sizes. In contrast, this is not possible with memcpy and MPI with CPU-staging, due to those interfaces utilizing solely CPU resources to perform data movement.

Communication Patterns. Our evaluation of HPC applications focuses on two applications with explicit data movement, which rely on explicit inter-process communication routines, including point-to-point and collective operations. Other applications might have unpredictable communication patterns, where the extent of the accessed data is unknown at runtime, such as graph processing applications. On the AMD MI300A tightly-coupled system, those applications can benefit from direct data access from GPU kernel, which provides granular access to remotely-located data. We demonstrated that such access strategy achieves full utilization of the Infinity Fabric bandwidth, with approximately twice the latency of local access.

9 Related Works

AMD MI250X and MI300A. Vijayaraghavan et al. [33] described the concept of APU, integrating CPU, GPU and memory within a single package, for exascale computing, later implemented as MI300A. Smith et al. [30] reported the technical details on the MI300A APU and key manufacturing insights. On this APU, the study of the unified CPU-GPU physical memory system has been conducted by Wahlgren et al. [34]. Porting and evaluation of HPC applications to one MI300A APU have been proposed through OpenMP's unified memory model [8, 32]. The previous generation of AMD GPUs, AMD MI250X, has previously been studied from various perspectives, including its Infinity Fabric interconnect [23] and Matrix Cores [25]. Our work focuses on the study of infinity Fabric Interconnect, connecting several APUs within a multi-MI300A node, which is the building block of the latest leadership HPC systems.

Evaluation of GPU-GPU Interconnects. GPU-GPU interconnects have been widely studied, including Nvidia's NVLink interconnect [17], and previous generations of Infinity Fabric interconnect, with AMD MI100 and MI250X GPUs [11, 15, 24]. De Sensi et al. [11] evaluated GPU-GPU communication at large scale on three supercomputers, using both inter- and intra-node benchmarks. The similarity and difference on several interconnect characteristics have also been explored, in particular AMD MI250X-based systems [29, 11]. Atchley et al. [7] provided a large-scale evaluation of the Frontier supercomputer, including intra-node and inter-node evaluation. Khorassani et al. [15] evaluated Slingshot-interconnected nodes based on AMD MI100 GPUs. Hidayetoglu et al. [14] focused on the multiple hierarchies in supercomputer interconnects. Schieffer et al. [26] characterized point-to-point and collective communication and memory allocation strategies on multi-GPUs MI250X-based

supercomputers. In this work, we focus on the Infinity Fabric interconnected multi-MI300A compute nodes.

Multi-GPU Optimizations. Distributed multi-GPU systems are used ubiquitously on HPC systems and Data centers to accelerate a wide range of applications, including large language models, quantum computer simulations, and database query processing. In multi-GPU applications, understanding data movement patterns and bottlenecks is critical for performance. Such analysis was conducted on, e.g, Graph Neural Networks (GNN) applications [9] and Convolutional Neural Networks (CNN) [31, 28]. To tackle the GPU-GPU communication bottleneck, several solutions have been proposed, including efficient workload partitioning using CUDA features [21] and leveraging multiple path or CPU-GPU interconnects [19]. Young et al. [35] quantified the multi-GPU interconnect bottleneck with NUMA-aware software solutions like work scheduling, page placement, page migration, page replication, and caching remote data; and proposed co-design optimization strategies. Our work, especially the characterization results on multi-APU systems, provides a strong foundation for optimizing these applications and workloads on the emerging HPC systems and data centers.

10 Conclusions

In this work, we evaluated inter-APU communication on Infinity Fabric on AMD MI300A systems. We quantified the peak hardware capacity and evaluated performance efficiency for various communication patterns, including CPU-GPU, point-to-point GPU-GPU, and GPU collectives. Our results quantified the impact of memory allocators and programming interfaces for data movement. Finally, we applied the optimization strategy on GPU-GPU communication in Quicksilver and CloverLeaf on four MI300A APUs, achieving a 2.15× speedup in communication.

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-2004814. This research is supported by the Swedish Research Council (no. 2022.03062) and LLNL LDRD project 24-ERD-047. This work has received funding from the European High Performance Computing Joint Undertaking (JU) and Sweden, Finland, Germany, Greece, France, Slovenia, Spain, and Czech Republic under grant agreement No 101093261.

References

[1] AMD. 2024. Compute language runtimes (clr) implementation, hipmemcpy implementation. https://github.com/ROCm/clr/blob/rocm-6.2.1/hipamd/src/hip_memory.cpp. (2024).

[2] AMD. 2024. Heterogeneous system architecture (hsa) documentation. https://rocm.docs.amd.com/projects/ROCR-Runtime/en/docs-6.2.1/. (2024).

[3] AMD. 2024. Heterogeneous-computing interface for portability (hip) documentation. https://rocm.docs.amd.com/projects/HIP/en/docs-6.2.1/index.html. (2024).

[4] AMD. 2024. Mi300a system optimization guide. https://instinct.docs.amd.com/projects/amdgpu-docs/en/latest/system-optimization/mi300a.html. (2024).

[5] AMD. 2024. Rocm documentation, gpu memory. https://rocm.docs.amd.com/en/docs-6.2.1/conceptual/gpu-memory.html. (2024).

[6] AMD. 2024. Rocr runtime source code. https://github.com/ROCm/ROCR-Runtime/blob/rocm-6.2.1/src/core/runtime/blit_shaders/blit_copyAligned.s. (2024).

[7] Scott Atchley et al. 2023. Frontier: exploring exascale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 1–16.

[8] Carlo Bertolli, Thorsten Blass, Lynd Stringer, Nicole Aschenbrenner, Jan-Patrick Lehr, Doru Bercea, Dhruva Chakrabarti, Lawrence Meadows, and Ron Lieberman. 2024. 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, 1420–1429.

[9] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. 2021. Dgcl: an efficient communication library for distributed gnn training. In *Proceedings of the Sixteenth European Conference on Computer Systems*, 130–144.

[10] Sangjin Choi, Taeksoo Kim, Jinwoo Jeong, Rachata Ausavarungnirun, Myeongjae Jeon, Youngjin Kwon, and Jeongseob Ahn. 2022. Memory harvesting in Multi-GPU systems with hierarchical unified virtual memory. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 625–638.

[11] Daniele De Sensi et al. 2024. Exploring gpu-to-gpu communication: insights into supercomputer interconnects. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[12] Luigi Fusco, Mikhail Khalilov, Marcin Chrapek, Giridhar Chukkapalli, Thomas Schulthess, and Torsten Hoefler. 2024. Understanding data movement in tightly coupled heterogeneous systems: a case study with the grace hopper superchip. *arXiv preprint arXiv:2408.11556*.

[13] Google. 2016. Google benchmark library. https://github.com/google/benchmark. (2016).

[14] Mert Hidayetoglu et al. 2024. Commbench: micro-benchmarking hierarchical networks with multi-gpu, multi-nic nodes. In *Proceedings of the 38th ACM International Conference on Supercomputing*, 426–436.

[15] Kawthar Shafie Khorassani, Chen-Chun Chen, Bharath Ramesh, Aamir Shafi, Hari Subramoni, and Dhabaleswar K Panda. 2023. High performance mpi over the slingshot interconnect. *Journal of Computer Science and Technology*, 38, 1, 128–145.

[16] Lawrence Livermore National Laboratory. 2017. https://asc.llnl.gov/codes/proxy-apps/quicksilver. (2017).

[17] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Xu Liu, Nathan Tallent, and Kevin Barker. 2018. Tartan: evaluating modern gpu interconnect via a multi-gpu benchmark suite. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 191–202.

[18] Gabriel H Loh et al. 2023. A research retrospective on amd's exascale computing journey. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 1–14.

[19] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Pump up the volume: processing large data on gpus with fast interconnects. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 1633–1649.

[20] Simon McIntosh-Smith, Michael Boulton, Dan Curran, and James Price. 2014. On the performance portability of structured grid codes on many-core computer architectures. In *Supercomputing: 29th International Conference, ISC 2014, Leipzig, Germany, June 22-26, 2014. Proceedings 29*. Springer, 53–75.

[21] Krzysztof M Ocetkiewicz, Cezary Czaplewski, Henryk Krawczyk, Agnieszka G Lipska, Adam Liwo, Jerzy Proficz, Adam K Sieradzan, and Paweł Czarnul. 2024. Multi-gpu unres for scalable coarse-grained simulations of very large protein systems. *Computer Physics Communications*, 298, 109112.

[22] 2001. Osu micro-benchmarks. (2001). http://mvapich.cse.ohio-state.edu/benchmarks/.

[23] Carl Pearson. 2023. Interconnect bandwidth heterogeneity on amd mi250x and infinity fabric. *arXiv preprint arXiv:2302.14827*.

[24] Carl Pearson, I-Hsin Chung, Zehra Sura, Wen-Mei Hwu, and Jinjun Xiong. 2018. Numa-aware data-transfer measurements for power/nvlink multi-gpu systems. In *International Conference on High Performance Computing*. Springer, 448–454.

[25] Gabin Schieffer, Daniel Araújo De Medeiros, Jennifer Faj, Aniruddha Marathe, and Ivy Peng. 2024. On the rise of amd matrix cores: performance, power efficiency, and programmability. In *2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 132–143.

[26] Gabin Schieffer, Ruimin Shi, Stefano Markidis, Andreas Herten, Jennifer Faj, and Ivy Peng. 2024. Understanding data movement in amd multi-gpu systems with infinity fabric. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 567–576.

[27] Gabin Schieffer, Jacob Wahlgren, Jie Ren, Jennifer Faj, and Ivy Peng. 2024. Harnessing integrated CPU-GPU system memory for HPC: a first look into Grace Hopper. In *Proceedings of the 53rd International Conference on Parallel Processing*, 199–209.

[28] Shaohuai Shi, Qiang Wang, and Xiaowen Chu. 2018. Performance modeling and evaluation of distributed deep learning frameworks on gpus. In *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress*. IEEE, 949–957.

[29] Christopher M Siefert, Carl Pearson, Stephen L Olivier, Andrey Prokopenko, Jonathan Hu, and Timothy J Fuller. 2023. Latency and bandwidth microbenchmarks of us department of energy systems in the june 2023 top 500 list. In

Proceedings of the SC'23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis, 1298–1305.

[30] Alan Smith et al. 2024. Realizing the amd exascale heterogeneous processor vision: industry product. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 876–889.

[31] Nathan R Tallent, Nitin A Gawande, Charles Siegel, Abhinav Vishnu, and Adolfy Hoisie. 2018. Evaluating on-node gpu interconnects for deep learning workloads. In *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation: 8th International Workshop, PMBS 2017, Denver, CO, USA, November 13, 2017, Proceedings 8*. Springer, 3–21.

[32] Suyash Tandon, Leopold Grinberg, Gheorghe-Teodor Bercea, Carlo Bertolli, Mark Olesen, Simone Bna, and Nicholas Malaya. 2024. Porting hpc applications to amd instinct™ mi300a using unified memory and openmp®. In *ISC High Performance 2024 Research Paper Proceedings (39th International Conference)*. Prometeus GmbH, 1–9.

[33] Thiruvengadam Vijayaraghavan et al. 2017. Design and analysis of an apu for exascale computing. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 85–96.

[34] Jacob Wahlgren, Gabin Schieffer, Ruimin Shi, Edgar Leon, Roger Pearce, Maya Gokhale, and Ivy Peng. 2025. Dissecting CPU-GPU unified physical memory on AMD MI300A APUs. In *2025 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE.

[35] Vinson Young, Aamer Jaleel, Evgeny Bolotin, Eiman Ebrahimi, David Nellans, and Oreste Villa. 2018. Combining hw/sw mechanisms to improve numa performance of multi-gpu systems. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 339–351.