DMA-Assisted, Intranode Communication in GPU Accelerated Systems

Feng Ji,* Ashwin M. Aji,† James Dinan,‡ Darius Buntinas,‡ Pavan Balaji,‡
Rajeev Thakur,‡ Wu-chun Feng,† Xiaosong Ma*§

* Department of Computer Science, North Carolina State University. *fji@ncsu.edu, ma@cs.ncsu.edu*
† Department of Computer Science, Virginia Tech. {*aaji, feng*}*@cs.vt.edu*
‡ Math. and Computer Science Div., Argonne National Lab. {*dinan, buntinas, balaji, thakur*}*@mcs.anl.gov*
§ Computer Science and Mathematics Division, Oak Ridge National Laboratory

***Abstract*—Accelerator awareness has become a pressing issue in data movement models, such as MPI, because of the rapid deployment of systems that utilize accelerators. In our previous work, we developed techniques to enhance MPI with accelerator awareness, thus allowing applications to easily and efficiently communicate data between accelerator memories. In this paper, we extend this work with techniques to perform efficient data movement between accelerators within the same node using a DMA-assisted, peer-to-peer intranode communication technique that was recently introduced for NVIDIA GPUs. We present a detailed design of our new approach to intranode communication and evaluate its improvement to communication and application performance using micro-kernel benchmarks and a 2D stencil application kernel.**

I. Introduction

In recent years, graphics processing units (GPUs) have emerged as excellent low-cost, power-efficient accelerators for general-purpose, highly parallel computations. Across a broad range of computational science, engineering, and analytics domains, GPUs have shown significant advantages over traditional CPUs. These results have, in turn, led to an increasing number of supercomputer systems being designed with GPUs. In the November 2011 Top500 list [1], for example, three of the top five supercomputers in the world utilized GPUs. Systems that can accommodate two or even four GPUs per node are fairly common today, and the price versus performance benefit of GPUs indicates that such trends will become even more common over the next several years.

The processing units on current GPUs can operate only on data that is located in the on-board GPU memory. While the GPU programming libraries provide mechanisms for transferring data between host memory and GPU memory, challenges remain in efficiently scheduling and synchronizing these transfers. Managing data transfers has been previously studied [2]–[4] with respect to message-passing libraries, specifically those implementing the Message Passing Interface (MPI) standard [5]. These studies concentrated on optimizing transfers between GPU memory and host memory, when the source or the destination of the data is on a GPU device. One of the shortcomings of these previous approaches is that they did not take advantage of systems where multiple GPUs were installed on the same compute node and data had to be moved between them. In our previous work [6], we designed a shared-memory approach that utilizes a common host memory buffer that is visible to both the source and destination process, reducing the number of memory copy operations required and thus improving overall performance. However, this approach still requires intervention from the host processor and memory to "stage" data before it can be moved between the two GPU devices.

Recently, a *GPU IPC* feature was introduced on new GPU hardware from NVIDIA that allows GPU direct memory access (DMA) engines to directly move data from one GPU to another on the same node. In this paper, we present the design of an efficient intranode cross-GPU/CPU peer-to-peer communication scheme for MPI communication. We explore the use of the GPU's on-board DMA engines and take advantage of new, cross-GPU, and peer-to-peer data accessibility, provided by GPU IPC. By utilizing the latest architectural features, our scheme can bypass the extra data paths through host memory used in current intranode GPU communication mechanisms for MPI, instead performing a direct transfer between source and destinations buffers. Furthermore, this work addresses the following challenges in intranode GPU communication.

- We introduce the use of GPU DMA engines, GPUDirect [7] and CUDA IPC [8] in the design of a DMA-assisted, peer-to-peer, direct intranode MPI communication subsystem. We show that this design can be extended to optimize communication between CPU and GPU devices.
- We explore the design space of intranode communication with GPU devices, including protocol design and use of DMA engines.
- We implement our design by adapting MPICH2, a widely used MPI implementation. We evaluate the performance on two typical GPU-accelerated systems. Our results show that the DMA-assisted peer-to-peer communication is beneficial, especially when participating GPU devices are close in a system. Applying our solution to the 2D stencil benchmark from SHOC [9], we demonstrate an average 4.7% and 2.3% performance improvement for single- and double-precision runs, respectively.

The rest of this paper is organized as follows. Section II

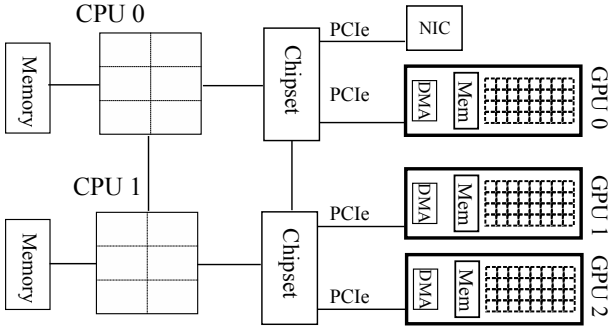


Fig. 1. GPU-accelerated computing system architecture.

presents background information on GPU computing, MPI, and MPICH2's intranode communication architecture. Section III introduces the design of our system, its integration in MPICH2, protocol designs, and a memory handle caching optimization. Section IV presents an experimental evaluation, and Section V discusses related work. We summarize our findings in Section VI.

II. Background

GPUs originally were designed for graphics rendering workloads but today are widely used as accelerators for general-purpose parallel computing. GPUs have specialized hardware optimized for parallel SIMD computations and are equipped with specialized, high-throughput device memory. Figure 1 shows an example heterogeneous system architecture, based on Keeneland cluster nodes [10] described in Section IV. In this figure, multiple GPU devices are each connected via a PCI-Express (PCIe) interconnect to a chipset (I/O hub), which connects these devices to other I/O hubs and CPUs. Different technologies can be used for connecting chipsets, CPUs, and memory, for example, the Intel Quick Path Interconnect (QPI) or AMD HyperTransport (HT) interconnects.

Several applications take advantage of such GPUs by offloading computation intensive portions of their execution to the GPU. These offloaded computation tasks are called *kernels*. When an application offloads a computation task to the GPU, the data to be used in such computation must be moved from the CPU host memory to the GPU device memory. Similarly, after the kernel execution, the resulting data must be moved back to the host memory. Data movement to and from the GPU is achieved by programming the GPU DMA engine. To do so, the CPU thread (also called the *controlling thread*) must first allocate a *GPU context*, which will be used by the controlling thread to interact with the GPU.

A. CUDA and NVIDIA GPUs

The Compute Unified Device Architecture (CUDA) is a popular, general-purpose GPU parallel programming model and is designed primarily for NVIDIA GPUs [11]. CUDA recently added support for several capabilities that can be utilized to enhance the efficiency of GPU data movement, including GPUDirect and CUDA IPC.

GPUDirect [7] enables direct, peer-to-peer GPU data transmission through GPU DMA engines, without any host processor intervention. In the past, when data needed to be moved between the device memories of two GPUs, it had to be "staged" in the host memory. With GPUDirect, the data can be transferred from one device directly to another. However, this feature is currently restricted to *peer-accessible* devices—those that are attached to the same chipset or different chipsets that are connected via AMD HT. GPUDirect does not currently support Intel QPI-connected cross-chipset GPU devices.

Another technology available only in CUDA, *CUDA IPC* [8], allows different processes to access the same buffer located in GPU device memory. With this technology, a process can share with another process a *memory handle* that references a device memory buffer. This feature is useful for parallel applications with multiple processes running on the same node, such as MPI applications.

B. MPI and MPICH2 Intranode Communication

MPI [5] is the industry standard for parallel programming on virtually all parallel computing architectures. Most popular MPI implementations provide highly optimized internode communication as well as intranode communication between cores and processors on the same node. MPICH2 [12], developed at Argonne National Laboratory, is a widely used, open-source MPI implementation. Its intranode communication is handled by the *Nemesis* [13] communication subsystem. MPICH2 has two data transmission modes: *eager* mode, optimized toward latency for shorter messages, and *rendezvous* mode, optimized toward bandwidth for large messages. The rendezvous mode is implemented through the *large message transfer* (LMT) protocol in Nemesis. Currently, this protocol has supported several transport methods that use shared-memory buffers and kernel-assisted single copy through host-side DMA. The shared-memory buffer implementation allocates buffers shared between the sender and receiver processes for them to store/remove message data. The sender and receiver processes work in parallel to pipeline the memory copies.

In our previous work, we designed an approach to allow intranode communication from GPU buffers [6]. This eliminated the need for the application to explicitly copy data from the source GPU memory to the host memory before an MPI send operation. It also eliminated explicit data copying from the host memory to the destination GPU memory after an MPI receive operation. The shared-memory LMT implementation was modified to use GPU data movement commands to directly transfer PCIe transaction data into and out of an LMT buffer. However, this method still requires copying GPU-resident data to shared buffers in host memory, requiring two DMA transfers and intervention from the host processor for the data transfer to occur.

III. Design

Each process within a node has its own virtual address space in CPU and GPU memories. A virtual address from

one process cannot be dereferenced in the address space of another, without OS support for sharing memory mappings. While peer-to-peer GPU memory copies (via GPUDirect) are possible with CUDA, they are restricted to a single process. In previous work, we addressed the intranode communicaion problem by extending MPICH2's Nemesis communication system and performing MPI communication between GPUs via host-side shared memory (`shm`).

As described in Section II-A, recent releases of CUDA (v4.1 or later) have exposed a new family of IPC functions, namely CUDA IPC [8], which provide the capability of exporting a memory handle to a GPU memory allocation from one process directly into the address space of another process within the same node. Using CUDA UPC, the MPI process driving the communication can issue an asynchronous DMA request, by calling cudaMemcpyAsync, to move data between participating GPUs directly. This feature, together with GPUDirect, can be used to perform direct, peer-to-peer data transfer. It also allows us to completely avoid pipelining through host-side shared memory buffers. We note, however, that GPUDirect is limited only to peer GPU devices connected to the same I/O hub or different I/O hubs connected via AMD HT and such "peer accessibility" must be queried from the GPU device. In our design, we use this approach for peer GPU devices and fall back to the original shared-memory-based approach for other GPU devices.

Since no static binding exists between an MPI rank and a GPU device, the process can choose any available GPU at runtime. Therefore a process cannot know whether peer accessibility is available to the pair by using its own information. To solve this problem, we use the handshake phase of the Nemesis LMT protocol (discussed in Section III-A) to exchange the peer accessibility information of devices before performing the communication. Note that the usage of LMT limits the applicability of DMA-assisted communication only to MPICH2 rendezvous mode, which is used primarily for large messages.

A. LMT Peer-GPU Protocol for Intranode Communication

In Nemesis, three LMT protocol models—PUT, GET, and COOPERATE—are provided for supporting intranode communication. PUT and GET protocols are used to implement kernel-assisted, single-copy protocols, and the COOPERATE protocol is used for the shared-memory-based intranode communication. The three protocol models are different in the process that initiates the payload transfer. In this paper, we design an additional *LMT peer-GPU* protocol, which can adaptively change into a PUT, GET or COOPERATE mode, depending on peer accessibility.

We show the control flow of the LMT Peer-GPU protocol in Figure 2. When the sender starts to participate in the handshake, it retrieves the interprocess memory handle for the sender's data buffer and then sends this memory handle along with the device number, packaged in a cookie, with the request to send (RTS) message to the receiver. When the RTS message arrives at the receiver process, it inspects the

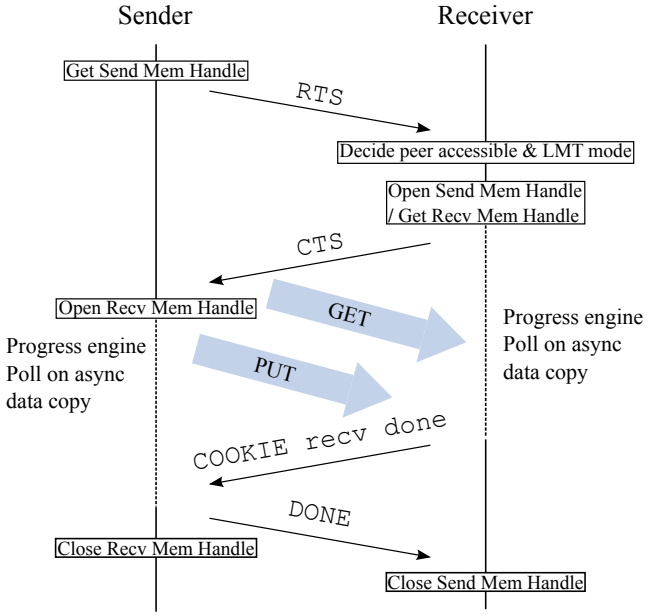


Fig. 2. LMT protocol for GPU peer-to-peer communication.

cookie and checks the peer accessibility of the two devices. If GPU peer accessibility is available, it performs peer-to-peer GPU communication using one of the three LMT protocols. Otherwise, it reverts to the `shm` approach. In another cookie created for the clear-to-send (CTS) message, the receiver embeds this decision, in order to inform the sender of the chosen protocol.

If the source and destination GPUs are peer accessible, the receiver can choose one of the three models: PUT, GET, or COOPERATE. This choice is arbitrary when GPU peers are accessible in both directions, but not in special cases (as further explained in Section III-B).

a) LMT Peer-GPU GET*:* If the receiver decides to get the data after receiving the RTS, it opens the sender's memory handle, maps it into its address space, and starts peer-GPU data movement. A progress element is inserted into MPICH2's progress engine queue by the receiver, and the DMA status is polled for completion. A DONE message is then sent to notify the sender of completion.

b) LMT Peer-GPU PUT*:* If the receiver decides to let the sender push the data, it retrieves the interprocess memory handle of the receive buffer, packages it with the device number as well as the decision in the CTS packet's cookie, and sends it back to the sender. The sender opens the receiver's memory handle, maps it into its address space, and executes peer-GPU data transfer. A progress element is inserted by the sender. The progress engine is again polled for completion, and a DONE message is sent to notify the receiver of completion.

c) LMT Peer-GPU COOPERATE*:* If the receiver decides both sides can help the data transfer, after both interprocess memory handles are exchanged in RTS and CTS messages, the payload is divided into two halves: the sender *put*s one half in the receive buffer, and the receiver *get*s the other half

from the sender's buffer. Progress elements are created on both sides. When receiver is done, a COOKIE message is sent to the sender to notify the partial completion of the data transfer; the sender then sends a DONE message to denote full completion.

The receiver's decision may seem arbitrary in the mutually accessible case; however, protocol selection decides which GPU, and hence the DMA engine, will be used for data movement—that is, the driving process will use the DMA engine on its GPU. Furthermore, DMA requests issued to the same engine will be serialized. Thus, the choice of protocol can be critical in managing DMA contention, depending on the communication pattern.

B. Intranode MPI Communication between Host and Device

In addition to GPU-to-GPU communication, we address intranode MPI communication between the host memory and the device memory. Currently, only GPU device memory can be exported to another process in CUDA IPC. Main memory buffers cannot be exported without the support of operating system kernel modules. Hence, the process with communication buffers in main memory (the *host-side process*) must be the one to initiate the payload transfer. Upon receiving the interprocess memory handle to the device memory buffer exported in the other process, the host-side process opens it, maps the memory to its address space, and then initiates the data transfer to or from the device.

The host-side process needs a valid GPU context to request the GPU DMA engine for data communication between the host and the device. However, this may not always be possible. Depending on the availability of an active GPU context, two situations might arise for host-device MPI communication, as described below.

a) Attach: If no active GPU context is available, the host-side process can *attach* to any available GPU device, by creating a new context on that GPU. A good choice here is to attach to the same device that contains the communicating device buffer. The context is then cached and can be reused for future data transfers.

b) Relay: If an active GPU context is already available, the DMA engine of the corresponding GPU can act as a *relay* to perform the data transfers with the device-side process. Although we can temporarily change the active GPU context to use another GPU device—possibly the one with the communicating device buffer—and change it back after the communication is done, this approach is not feasible. Since the active device context is a global setting in CUDA, changing it will redirect all GPU commands issued simultaneously to this communication onto the temporary active device, potentially polluting the user's program.

C. Efficient Management of Memory Handles

In all the LMT peer-GPU communication protocols, we first get the interprocess memory handle of a memory region in one process, then open it and map it into the address space of the other process. While getting the memory handle (cudaIpcGetMemHandle) is a lightweight operation [11], we find that opening the handle (cudaIpcOpenMemHandle) is expensive, probably because of interactions involving the importing and exporting of device buffer addresses in the driver run on the host side.

In a preliminary design, we open a memory handle after the RTS/CTS message exchange at the beginning of the communication and close it after the data transfer is done. Repeated opening and closing of the interprocess memory handle cause significant performance overhead.

Observing that many GPU programs have a relatively fixed memory creation pattern—for example, creating a large memory region before computation, reusing it for computation and data communication, and releasing it only after a period long enough—we choose to cache memory handles. Therefore, when a communication is done, we leave the memory handle open. During the next communication operation, when a memory handle arrives in an RTS/CTS packet's cookie, we check first to see whether the memory handle has been cached locally. If this is the case, our memory handle caching eliminates the reopening/closing of memory handles. We observe that the latency is more than halved after applying this optimization.

This design leads to a problem in closing a memory handle, however. In particular, the MPI runtime does not know when an open memory handle should be closed and closing a memory handle should happen before the memory region is freed [11]. To solve this problem, we add a two-phase GPU memory free mechanism, by providing two functions (gpuMemFree and gpuMemFree_commit). When a GPU memory free is called on a memory region, it is only recorded, with its memory handles marked in case it is ever exported, but not released immediately. When a GPU memory free commit is called, the marked memory handles are exchanged with other processes on that node. If found, a process closes the memory handle. After all processes finish closing memory handles, GPU memory regions are released.

IV. Evaluation

Our evaluation was conducted on two systems that are representative of current multi-GPU heterogeneous clusters. These systems differ significantly in cross-socket interconnect, NUMA configuration, and GPU connection topology, as summarized in Table I. The Keeneland [10] cluster is powered with NVIDIA Tesla M2070 GPUs. Each compute node is contains two Intel Xeon X5660 hex-core CPUs, 24 GB memory, and 3 GPU devices connected through 2 I/O hubs; nodes are connected via single-rail, QDR InfiniBand. The Magellan cluster is powered with NVIDIA Tesla M2070 GPUs. Nodes are configured with four AMD Opteron 6128 quad-core CPUs, 64 GB memory, and 2 GPU devices connected to 2 I/O hubs; the system interconnect is QDR InfiniBand. Both systems run the CentOS Linux operating system and utilize CUDA v4.1.

Communication performance measurements were gathered using the latency and bandwidth benchmarks from the OSU benchmark suite [14]. In addition, the impact of this work on application-level performance was measured using the Stencil2D kernel from the SHOC benchmark suite [9].

TABLE I
KEENELAND AND MAGELLAN SYSTEM ARCHITECTURES, INCLUDING GPU TOPOLOGIES.

Cluster	NUMA nodes	Interconnect	GPUs	GPU Topology	Peer Access	Distance Between Peers
Keeneland	2	Intel QPI	3	GPU 0: Node 0; GPU 1,2: Node 1	Only GPU 1 and 2	2 PCIe hops
Magellan	4	AMD HT	2	GPU 0: Node 0; GPU 1: Node 3	Yes	2 PCIe hops + 1 HT hop

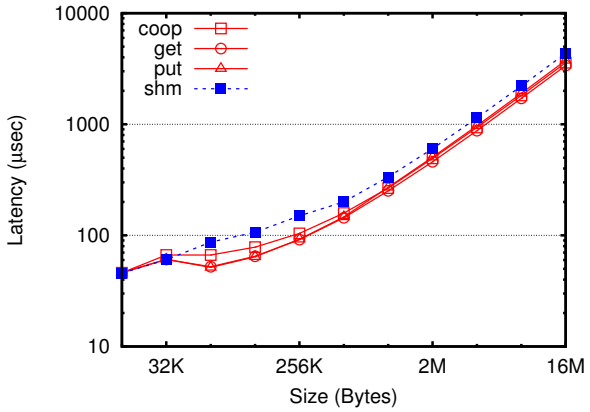
A. DMA-Assisted, Intranode GPU-GPU Communication

We first evaluate the performance of our GPU DMA-assisted peer-to-peer intranode communication. In this test, we compare its performance with that of our previous design (`shm`), the shared-memory-based data transfer approach. The latency and bandwidth test both involve two processes, using both source and destination buffers in GPU memory. On Keeneland, we use GPUs 1 and 2, connected on the same I/O hub (*near* case). On Magellan, GPUs 0 and 1 are connected to two different I/O hubs (*far* case). All our experiments in this section evaluate large message transfers (larger than 64 KB) in our current setting. We also always pin the CPU controlling process to the socket closest to the controlled GPU. Figures 3 and 4 compare the performance of these two cases.

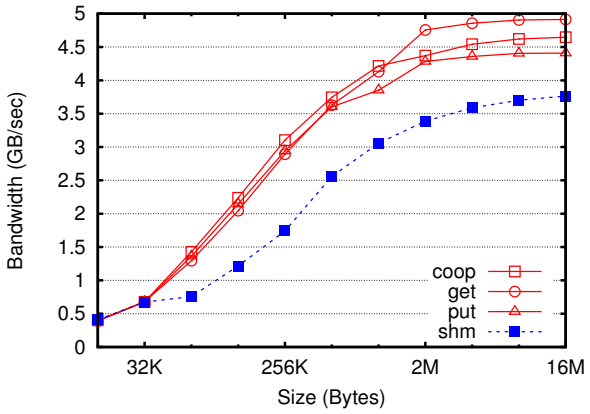
From this data, we see that the DMA-assisted communication provides lower latency and higher bandwidth than does `shm` when GPUs are connected to the same I/O hub, primarily because DMA-assisted LMT avoids data staging through host-side shared-memory buffers and reduces the contention on the shared I/O hub. When two GPUs are attached to different I/O hubs, however, as in the *far* case shown in Figure 4, we find the opposite result. The reason is that two GPUs are now connected by a longer data path consisting of three subchannels: a PCIe bus, an HT interconnect link, and another PCIe bus. Although DMA-assisted data movement avoids data staging in host-side shared memory, GPU DMA-driven transactions travel serially over the data path; at any time, only one transaction is going over the three subchannels. In contrast, the `shm` protocol uses both the sender and the receiver to write and read data into staging buffers, respectively, which partitions the data path into two relays and drives both PCIe links concurrently.

By comparing different LMT modes, we see that COOPERATE mode is never the best. This is a surprising result because, for the cooperative mode, we split the data into two halves, and both GPU DMA engines drive half of the data transfer concurrently. However, results indicate that, in practice, this method is consistently slower than one of the one-sided modes. This slower performance may be caused by the interference between two GPU devices; when a peer direct access happens, the DMA engine will talk to a remote agent on the GPU device for data location translation and memory module commands issuing. Therefore, when two DMA engines are working simultaneously, this can lead to contention in accessing these hardware resources.

We evaluate GPU-GPU communication performance when two processes are sharing one GPU device, a common case in practice because clusters typically have more CPU cores than GPU devices. Results are presented in Figure 5 for the Keeneland systeml similar results were observed on Magellan. From these results, we see that DMA-assisted data transfer is able to leverage fast data movement within the GPU device, resulting in an order of magnitude improvement in bandwidth and latency over `shm`.



(a) Latency versus Message Size

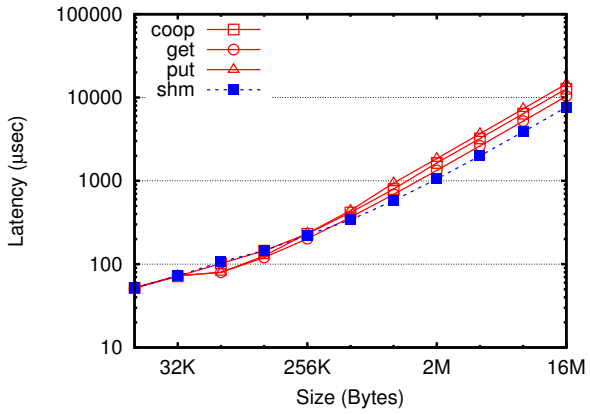


(b) Bandwidth versus Message Size

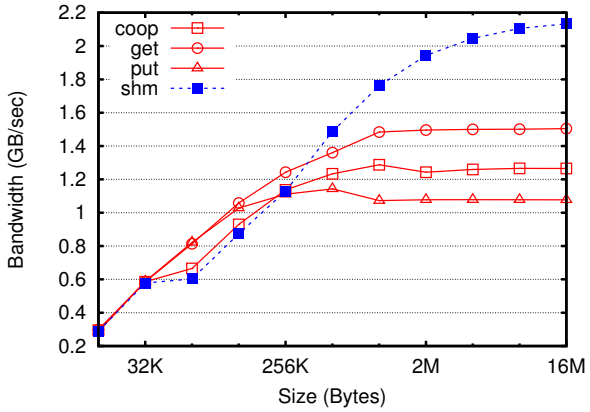
Fig. 3. Intranode GPU-GPU communication latency and bandwidth comparisons for the *near* case on Keenland. DMA-assisted communication using each LMT mode is compared with the baseline `shm` protocol.

B. DMA-Assisted, Intranode GPU-Host Communication

The DMA-assisted communication protocol can also be used for intranode communication where one buffer is located in host memory and the other is in GPU memory. Figure 6 shows results for this case on Keeneland, where two processes are both pinned to NUMA node 1, where GPUs 1 and 2 are connected. We evaluate two cases, which are distinguished

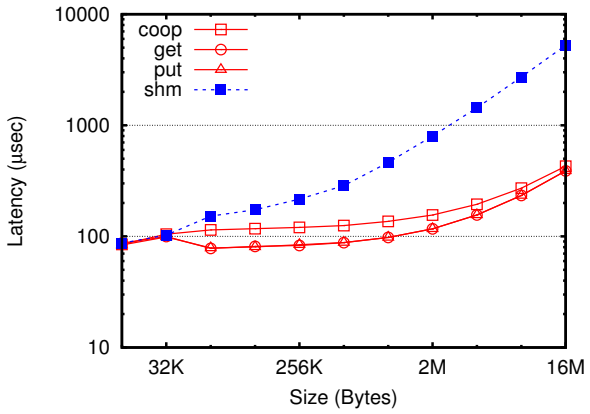


(a) Latency versus Message Size

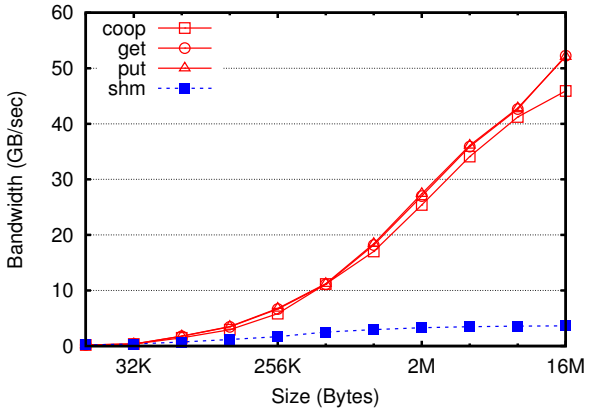


(b) Bandwidth versus Message Size

Fig. 4. Intranode GPU-GPU communication latency and bandwidth comparisons for the *far* case on Magellan. DMA-assisted communication using each LMT mode is compared with the baseline `shm` protocol.



(a) Latency versus Message Size



(b) Bandwidth versus Message Size

Fig. 5. Intranode GPU-GPU communication latency and bandwidth comparisons for the *sharing* case on Keenland. DMA-assisted communication using each LMT mode is compared with the baseline `shm` protocol.

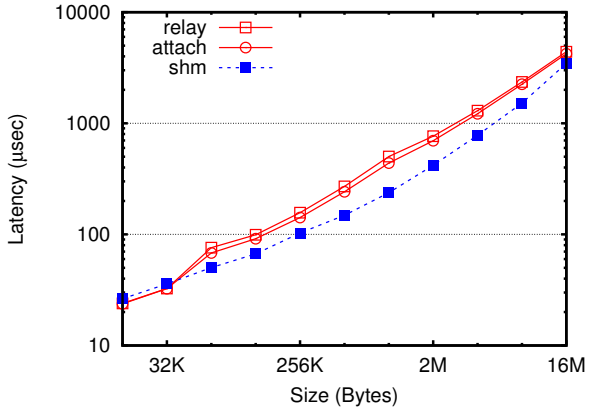
by whether the process using the host buffer is using other devices connected to the same I/O hub. Using this setup, we show results for *attached* and *relayed* transfers, as explained in Section III-B.

Our results indicate that DMA-assisted LMT does not perform as well as `shm` on Keenland, and we observed similar results on Magellan for GPU-Host communication where processes are pinned to further NUMA nodes. When comparing attached transfer performance with `shm`, we observe that although both data paths go from the GPU device to the host CPU, in `shm`, data can be copied to the shared memory buffer without considering peer accessibility. In contrast, the attached case must start a new context there to access the exported memory. As a result, the overhead of establishing peer accessibility overcomes the benefit of eliminating main memory copies—especially when the overhead of creating a new context is large. Though this overhead is amortized by later reuse, it significantly impacts performance. We expect that with the improvement of GPUs and the GPU driver, this overhead will decrease on future devices.

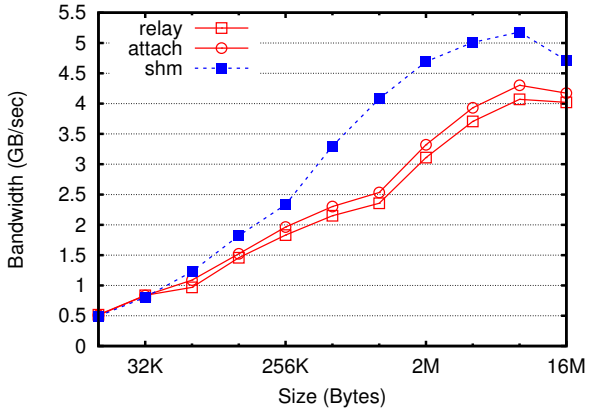
The relayed case emulates the scenario where the CPU process is using another GPU device for some computation while it performs Host-GPU communication. In this situation, the CPU process must use the DMA engine on the currently active device. This case shows that using a remote DMA engine to relay the data between a host buffer and device memory results in poor performance. Thus, in both Host-GPU transfer cases, we should fall back to `shm` to provide the best performance.

C. Application Evaluation: Stencil2D

The Stencil2D kernel from SHOC benchmark suite [9] measures the performance of a nine-point, two-dimensional stencil computation. It performs an iterative stencil computation on the GPU and requires a data exchange every haloWidth iterations. In this type of computation, processes are arranged in an N-dimensional Cartesian grid, and each process is assigned a corresponding section of an N-dimensional array. Periodically, a process must obtain the values that its neighbors have calculated for the array elements that border its patch, or its *halo*. Thus, this communication idiom, which is common across a broad range of iterative solvers, is referred to as a halo exchange.

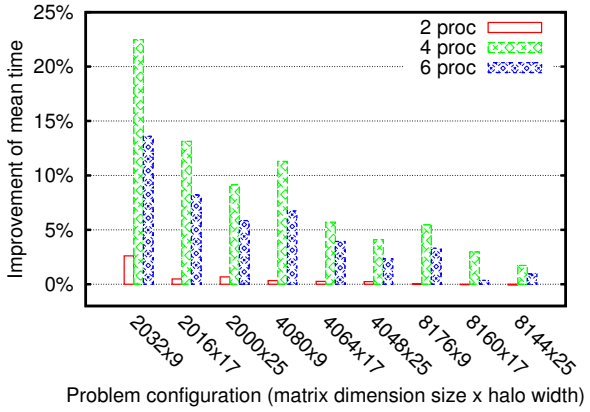


(a) Latency versus Message Size

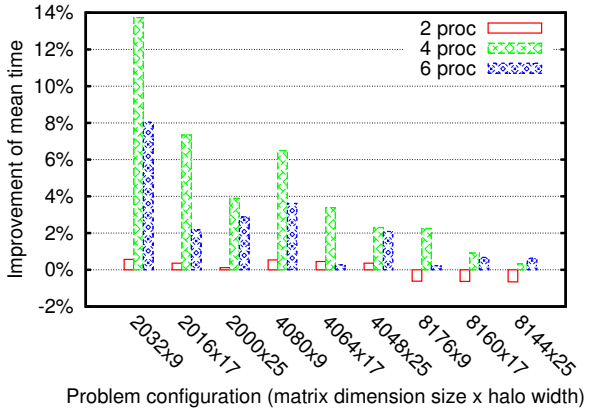


(b) Bandwidth versus Message Size

Fig. 6. Intranode Host-GPU communication latency and bandwidth comparisons on Keenland. DMA-assisted communication using *relayed* and *attached* transports is compared with the baseline `shm` protocol. When relayed transport is used, two GPUs connected to the same chipset are used.



(a) Singe precision



(b) Double precision

Fig. 7. Stencil2D performance improvement on Keeneland for 2, 4, and 6 processes. Improvement of DMA-assisted peer-to-peer communication over shared-memory buffer-based communication is shown.

Figure 7 shows the relative performance improvement of DMA-assisted communication over the original shared memory (`shm`) approach. Results were gathered on the Keenland system for both single- and double-precision versions of the calculation. Overall, DMA-assisted communication provided an average speedup of 4.7% for single precision and 2.3% for double precision. Since double-precision performance is much lower than single-precision performance on GPUs, communication accounts for a smaller portion of the total runtime, resulting in a smaller overall benefit for double-precision.

In this workload MPI ranks are assigned to GPU devices in a round-robin manner. This explains the high improvement factor seen in the case of four processes. In this case, ranks 0 and 3 are assigned to the GPU 0, and ranks 1 and 2 are assigned to GPU 1. The halo exchange happens first vertically and then horizontally; in each step, DMA-assisted peer communication occurs between one pair of processes, and the original `shm` protocol is used between the other pair. As a result, communication overlaps in a mutually beneficial pattern. This overlap also occurs in the six process case; however, additional MPI ranks sharing a GPU leads to a higher level of contention and results in a lower degree of performance improvement. The case of two processes results in a surprising decrease in performance. This may be due to PCIe bus contention, since both parties are trying to send and receive equal size messages; we plan to continue study this case.

When we analyze performance improvement relative to problem size for 2,048, 4,096, and 8,192 groups of matrix sizes (the matrix dimensions within each group are changed as needed to vary halo width), average speedups are 8.5%, 3.9%, and 1.6% for single precision and 4.3%, 2.2%, and 0.3% for double precision, respectively. We observe that the amount of computation increases quadratically with problem size, thus effectively reducing the fraction of time spent on communication and, as a result, the potential for performance improvement.

We increase the halo widths within each matrix dimension group in order to explore the change in performance as the total communication volume is the same but communication occurs less frequently. This reduces the total communication

time—for example, for the problem size of 2,048, the communication portion of total execution time decreases from 34% to 14% for single precision and 22% to 9% for double precision in the DMA-assisted case—and therefore reduces the benefit of improved data movement.

V. Related Work

Several research efforts have investigated modifications to MPI to better facilitate hybrid MPI+GPU programming. Currently, only processes running on the CPU can perform MPI calls. Stuart et al. [15] have suggested several mechanisms for extending the MPI standard to provide native support for accelerators. One significant proposal would allow GPU threads to obtain MPI ranks and participate directly in MPI communication [16]. However, due to lacking network I/O functionality on GPUs, CPU helper threads are needed, which presents challenges to performance modeling and may introduce new overheads.

Another major area of research, utilizes the current model for MPI participation in GPU-accelerated systems and extends it with transparent solutions for interacting with accelerator data [2]–[4], [6]. An advantage of this approach is that it allows existing MPI programs to more easily benefit from GPUs by reducing the amount of programmer effort that must be expended to manage distinct host and device memories. This work falls into the above category and differs from prior work in this space by developing techniques to use new accelerator features to accelerate intranode communication.

While MPI has traditionally been known as a system for internode communication, intranode communication has become equally important because of increasing core counts [17]–[21]. This work compliments existing intranode communication efforts by studying the impact of GPUs on intranode communication systems.

VI. Concluding Remarks

In this work, we explored the design of an intranode communication subsystem for a GPU-aware MPI implementation that allows the programmer to supply device buffers directly to MPI calls. Mechanisms for direct, DMA-assisted peer-to-peer data transfers involving host and GPU as well as GPU and GPU buffers were developed. Through communication benchmarking, we evaluated the performance of several alternative approaches and constructed a full system that utilizes the best protocols and parameters in each context. Our communication benchmarking revealed that DMA-assisted peer-to-peer data transfer yields greater benefits when applied to GPU devices that are nearby; in some situations, DMA-assisted transfers can hurt performance, and our implementation falls back to an efficient shared memory transport.

We evaluated the performance impact of our modified MPI implementation on a halo exchange application kernel. When compared with the baseline shared-memory data transfer method, an average speedup of 4.7% and 2.3% was observed for the stencil kernel for single- and double-precision computations, respectively.

Acknowledgments

This work was supported in part by the National Science Foundation through Career Award CNS-0546301; awards CNS-0915861, MRI-0960081, and CSR-0916719; and award I/UCRC IIP-0804155 via the NSF Center for High-Performance Recongurable Computing. This work was also supported by the U.S. Department of Energy under Contract DE-AC02-06CH11357 and X. Ma's joint appointment between NCSU and ORNL. The authors also used resources provided by the Keeneland Computing Facility at the Georgia Institute of Technology, which is supported by the National Science Foundation under contract OCI-0910735.

References

[1] "TOP500," http://www.top500.org/lists/2011/11/highlights.
[2] H. Wang, S. Potluri, M. Luo, A. Singh, S. Sur, and D. Panda, "MVAPICH2-GPU: Optimized GPU to GPU communication for InfiniBand clusters," *International Supercomputing Conference '11*, 2011.
[3] A. K. Singh, S. Potluri, H. Wang, K. Kandalla, S. Sur, and D. K. Panda, "MPI alltoall personalized exchange on GPGPU clusters: Design alternatives and benefit," in *Workshop on Parallel Programming on Accelerator Clusters '11*, 2011.
[4] H. Wang, S. Potluri, M. Luo, A. K. Singh, X. Ouyang, S. Sur, and D. K. Panda, "Optimized non-contiguous MPI datatype communication for GPU clusters: Design, implementation and evaluation with MVAPICH2." in *Proceedings of CLUSTER*. IEEE, 2011, pp. 308–316.
[5] MPI Forum, "MPI: A message-passing interface standard, version 2.2," September 4 2009.
[6] F. Ji, A. M. Aji, J. Dinan, D. Buntinas, P. Balaji, W.-C. Feng, and X. Ma, "Efficient intranode communication in GPU-accelerated systems," in *Proc. 2nd Intl. Workshop on Accelerators and Hybrid Exascale Systems (AsHES)*, 2012.
[7] "NVIDIA GPUDirect," http://developer.nvidia.com/gpudirect.
[8] "NVIDIA CUDA v4.1," http://developer.nvidia.com/cuda-toolkit-41.
[9] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (SHOC) benchmark suite," in *Proc. 3rd Workshop on General-Purpose Computation on GPUs*, ser. GPGPU '10. ACM, 2010.
[10] J. Vetter, R. Glassbrook, J. Dongarra, K. Schwan, B. Loftis, S. McNally, J. Meredith, J. Rogers, P. Roth, K. Spafford, and S. Yalamanchili, "Keeneland: Bringing heterogeneous GPU computing to the computational science community," *Computing in Science Engineering*, vol. 13, no. 5, pp. 90 –95, SEPT.-OCT. 2011.
[11] Nvidia, "NVIDIA CUDA C Programming Guide version 4.0."
[12] "MPICH2," http://www.mcs.anl.gov/research/projects/mpich2/.
[13] D. Buntinas, G. Mercier, and W. Gropp, "Implementation and shared-memory evaluation of MPICH2 over the Nemesis communication subsystem," in *Recent Adv. in PVM and MPI*, ser. Lec. Notes in Comp. Sci. Springer, 2006, vol. 4192, pp. 86–95.
[14] "OSU Micro-benchmarks 3.5," Sep. 2011, http://mvapich.cse.ohio-state.edu/benchmarks/.
[15] J. A. Stuart, P. Balaji, and J. D. Owens, "Extending MPI to accelerators," *PACT 2011 Workshop Ser.: Arch. and Systems for Big Data*, Oct. 2011.
[16] J. A. Stuart and J. D. Owens, "Message passing on data-parallel architectures," in *Proc. of the 23rd IEEE Intl. Par. and Distrib. Processing Symp.*, May 2009.
[17] D. Buntinas, G. Mercier, and W. Gropp, "Data transfers between processes in an SMP system: Performance study and application to MPI," in *ICPP 2006.*, 2006, pp. 487–496.
[18] D. Buntinas, B. Goglin, D. Goodell, G. Mercier, and S. Moreaud, "Cache-efficient, intranode, large-message MPI communication with MPICH2-Nemesis," in *ICPP 2009*, Sept. 2009, pp. 462–469.
[19] S. Moreaud, B. Goglin, D. Goodell, and R. Namyst, "Optimizing MPI communication within large multicore nodes with kernel assistance," in *The 10th Work. on Comm. Arch. for Clusters*, 2010.
[20] T. Ma, G. Bosilca, A. Bouteiller, B. Goglin, J. M. Squyres, and J. J. Dongarra, "Kernel assisted collective intra-node MPI communication among multi-core and many-core CPUs," in *ICPP 2011*, Taipei, Taiwan, 2011.
[21] T. Ma, G. Bosilca, A. Bouteiller, and J. J. Dongarra, "HierKNEM: An Adaptive Framework for Kernel-Assisted and Topology-Aware Collective Communications on Many-core Clusters," in *IPDPS 2012*, Shanghai, China, May 2012.