Efficient Intranode Communication in GPU-Accelerated Systems

Feng Ji*, Ashwin M. Aji†, James Dinan‡, Darius Buntinas‡, Pavan Balaji‡, 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
‡ *Mathematics and Computer Science Division, Argonne National Laboratory*
{dinan,buntinas,balaji}@mcs.anl.gov
§ *Computer Science and Mathematics Division, Oak Ridge National Laboratory*

***Abstract*—Current implementations of MPI are unaware of accelerator memory (i.e., GPU device memory) and require programmers to explicitly move data between memory spaces. This approach is inefficient, especially for intranode communication where it can result in several extra copy operations. In this work, we integrate GPU-awareness into a popular MPI runtime system and develop techniques to significantly reduce the cost of intranode communication involving one or more GPUs. Experiment results show an up to 2x increase in bandwidth, resulting in an average of 4.3% improvement to the total execution time of a halo exchange benchmark.**

I. Introduction

Graphics processing units (GPUs) have undergone significant architectural generalization over the past several years and have transformed into general purpose, highly parallel processors. As low-cost, power-efficient accelerators, GPUs have provided significant speedup across a broad range of computational science, engineering, and analytics domains. In the November 2011 Top500 list [1], 3 out of the top 5 supercomputers in the world—and a total of 39 out of 500 systems—utilize GPUs.

The Message Passing Interface (MPI) [2] is the industry standard for parallel programming and is used on virtually every high performance computing system. Introduced in 1984, the MPI standard has evolved to complement changes in CPU architecture and high-performance networks. One of the current challenges faced by the MPI community is the evolution of this popular parallel programming model to interoperate with and exploit GPU accelerators.

Current implementations of MPI continue to assume that all communication buffers are located in the main memory. Hence, developers must explicitly move data between device and host memories in order to perform MPI operations. With current GPUs, this strategy results in high-latency data movement across the PCIe bus and through several temporary buffers, even when performing communication between processes located on the same node. To facilitate these transfers, developers today typically maintain duplicate buffers in host and GPU memory manually, an approach that not only introduces code complexity but also is wasteful of system resources. Furthermore, one of MPI's strengths is its ability to hide the performance details of data movement in parallel computations. This capability is significantly diminished when manual GPU data movement is performed outside the MPI library. In an experiment conducted with a pair of MPI processes on the same physical machine, we find that the bandwidth achieved is only half the theoretical peak when manual data movement is used.

We address these challenges by extending MPI and integrating direct support for the GPU memory space into MPICH2, a popular open-source MPI implementation. This interface allows programmers to pass GPU buffers directly to MPI routines without the need for explicit intermediate copies. As an initial step toward efficient MPI communication for GPU-accelerated parallel applications, we present an approach to perform efficient intranode communication. By integrating GPU data movement into MPI, multiple optimizations become possible. Temporary copies can be eliminated, freeing additional resources to the application and significantly improving performance. In addition, by pipelining GPU and inter-process data movement, PCIe and main memory concurrency can be leveraged to increase transfer efficiency.

We evaluate our system on several microbenchmarks and on a nine-point two-dimensional stencil benchmark that performs halo-type neighboring exchange communication. Results indicate up to twofold improvement in bandwidth for large messages and roughly 10% improvement in latency for small messages. These improvements in raw communication performance translated into an average improvement of 4.3% to the total execution time of the 2-D stencil benchmark across a range of process counts and problem sizes.

The rest of this paper is organized as follows. Section II presents background information on GPU computing, MPI, and MPICH2's intranode communication architecture. Section III discusses current challenges in mixed GPU+MPI programming. Section IV introduces the design of our system, its integration with the Nemesis communication

subsystem in MPICH2, and several performance optimizations. Section V presents an experimental evaluation, and Section VI discusses related work. Section VII summarizes our conclusions.

II. Background

This work focuses on enhancing the performance of intranode communication where the source, target, or both buffers reside in separate accelerator memory. In this section, we provide background on computing with GPUs and MPI.

A. GPU Computing and CUDA

Graphics processing units were originally designed for rendering workloads. In recent years, the highly parallel GPU architecture was found to be extremely useful as a computation accelerator across a broad range of high-performance computing workloads.

Current GPU devices fall into two broad categories: integrated and discrete. Integrated GPUs are built into the same hardware components as the host processor and share memory and other resources. Discrete GPUs, on the other hand, are distinct from the host processor and tend to utilize a distinct memory subsystem. The larger chip area, more powerful parallel processing units, and high-throughput dedicated memory on discrete GPUs provide great performance potential and have made discrete GPUs a preferred design in current high-performance computing (HPC) systems. Current discrete GPUs are expansion cards that connect to the host processor and memory through the PCIe bus. Thus, data in host and device memories must be explicitly copied by using special commands.

The Compute Unified Device Architecture (CUDA) is one of the most popular general-purpose parallel programming models on GPU today and is designed primarily for NVIDIA GPUs [3]. Our work here focuses on CUDA because of its importance to current HPC applications; however, our design is general and can be readily adapted to other GPU programming models. In the CUDA model, data between host and device memories is transferred by using the cudaMemcpy command. The programmer annotates functions in the program source code to enable them to run on the GPU. When invoked, these functions are launched as GPU *kernels*.

B. The Message Passing Interface

MPI [2] is the industry standard for parallel programming. MPI defines point-to-point send/receive, one-sided, and collective communication operations and allows the programmer to construct parallel programs that are portable across virtually all parallel computing architectures. While MPI is best known for high-performance internode communication, most popular MPI implementations also provide highly optimized intranode communication between cores and processors on the same node.

C. MPICH2 Intranode Communication Architecture

MPICH2 is an open-source implementation of MPI standard. Designed for high-performance communication, MPICH2 is built on the Nemesis communication subsystem [4] and supports intranode message-passing through shared memory and internode communication by using network modules designed for specific networks. The Nemesis network module API efficiently supports high-performance RDMA-style networks as well as TCP. MPICH2 abstracts connections between two processes with a *virtual channel* data structure that stores state information associated with the connection. In this work, we focus on intranode GPU native communication and therefore will make modifications primarily to Nemesis.

MPICH2 has two data transmission modes: *eager* mode and *rendezvous* mode. Eager mode is intended for shorter messages and is optimized for latency; rendezvous mode is intended for large messages and is optimized for bandwidth. In eager mode, a message is sent to the other party through shared-memory message queues. The data is copied from the user buffer into one or more available elements in a free queue, which are then inserted to the receiver's receive queue. The message queue is implemented with atomic memory operations so that multiple processes can enqueue elements on the queue concurrently without locking. This strategy means that only one receive queue is needed per process, rather than one queue per pair of processes.

Nemesis uses the *large message transfer* (LMT) protocol to implement rendezvous mode. In the LMT protocol, shared-memory copy buffers are created between pairs of processes the first time they communicate with each other. The copy buffers are arranged as a ring buffer so that the sender can copy into one buffer while the receiver is copying out of another buffer in a pipelined manner. Nemesis supports other implementations of LMT that use *vmsplice* or *I/OAT* with kernel module support; however, these methods are not considered in this paper. The Nemesis progress engine is responsible for driving the LMT protocol.

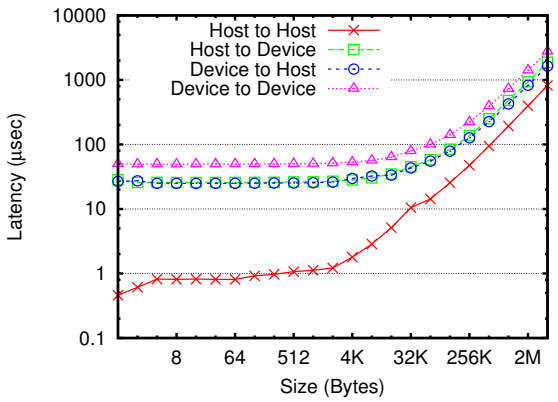
III. Challenges in CUDA+MPI Programming

The mixed CUDA+MPI parallel programming model has gained widespread adoption for programming clusters with GPU accelerators today. In this model, intranode parallelism is expressed in the CUDA model, and internode parallelism is managed through MPI. Because MPI implementations are not aware of the distinct accelerator memory space, the programmer must manually copy data between device and host buffers, as shown in Listing 1. In this simple example, the only purpose of the host_buf buffer is to facilitate MPI communication of data stored in device memory. As the number of accelerators (and hence distinct memories) per node increases, manual data movement poses significant productivity problems [5].

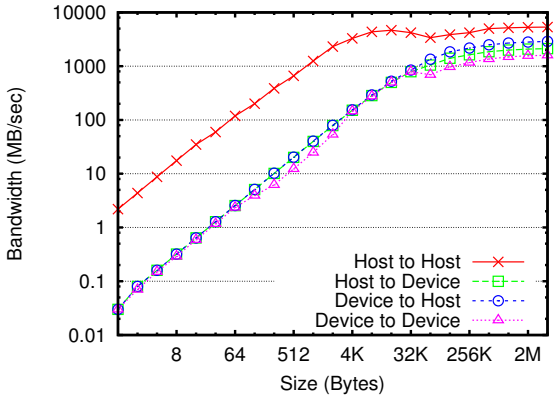
```
double *dev_buf, *host_buf;
cudaMalloc(&dev_buf, size);
cudaMallocHost(&host_buf, size);

if (my_rank == sender) { /* sender */
  computation_on_GPU(dev_buf);
  cudaMemcpy(host_buf, dev_buf, size, ...);
  MPI_Send(host_buf, size, ...);
} else {                 /* receiver */
  MPI_Recv(host_buf, size, ...);
  cudaMemcpy(dev_buf, host_buf, size, ...);
  computation_on_GPU(dev_buf);
}
```

Listing 1. Example MPI program with manual data movement.



(a) Latency Versus Transfer Size



(b) Bandwidth Versus Transfer Size

Figure 1. Intranode latency and bandwidth between two MPI processes, with different source/destination buffer location choices. When a GPU device buffer is used, its associated host buffer is pinned.

Beyond the problem of programming productivity, the performance of such a mixed programming scheme is also troublesome. In Figure 1, we measured the latency and the bandwidth of intranode communication between two MPI processes running on a single machine, using the latency and bandwidth tests from the OSU MPI micro-benchmark suite [6]. In this experiment, we vary the location of source and destination buffers between main memory and GPU device memory. When a buffer is located in device memory, a manual cudaMemcpy is performed between a temporary host buffer and the source/target device buffer.

From these results, we see that latency is increased and bandwidth decreased by more than an order of magnitude for small messages when either buffer is located in device memory. The bandwidth achieved when a GPU buffer is used peaks at between 1.6 and 2.8 GB/sec. This is significantly lower than the theoretical peak of this system, 6 GB/sec, which is bounded by the smaller of the PCIe and memory bandwidths. This gap of roughly 50% in performance is incurred because the naïve two-copy method ignores parallelism present in the hardware that allows bidirectional memory copies between host and device and host-to-host copies to proceed in parallel. As we describe in Section IV, techniques such as pipelining of data transfers can greatly improve communication performance by leveraging hardware data movement parallelism. Such techniques can be implemented in an application by the programmer; however, they introduce significant complexity and, as we will demonstrate, can deliver greater performance benefit when integrated with the existing intranode communication infrastructure.

IV. Design

To address the challenges of mixed CUDA+MPI programming, we integrate support for device-resident buffers into the MPI implementation. This approach not only improves productivity but also greatly improves performance by more efficiently communicating data within the node. The GPU memory buffer support is added to MPICH2 by making MPI communication routines aware of addresses in the device memory, enabling MPI to internally perform efficient intranode data movement. DMA-assisted asynchronous PCIe data movement is leveraged and built into the current LMT pipeline of Nemesis in order to increase throughput. Data movement between host and device also plays an important role in achieving a high performance implementation, and several schemes are explored.

A. Integrating GPU-Awareness in the MPI Interface

Several options are possible to integrate support passing GPU buffer arguments to MPI routines. CUDA provides a unified virtual address space (UVA), which maps device buffers into the virtual address space and provides routines that can be used to query whether a given address corresponds to host or device memory. This way, device buffers can be passed directly to MPI, and MPI can decide whether to start a PCIe or a main memory transfer by internally querying the buffer's location. However, other accelerator programming models of interest, such as OpenCL [7], do not provide a unified virtual address space and require several

handles (e.g., context handle) to access the device.

To provide a fully generic interface that can support a variety of accelerator models, we have extended the MPI interface with an *MPIGPU* interface that adds a buffer type parameter for each buffer passed to an MPI routine. The buffer type parameter allows the programmer to specify how the corresponding void * argument should be interpreted and in which memory space the argument resides. For example, in an OpenCL program, a struct or record argument with relevant resource handles would be passed, which provides MPI with all the information needed to access the given buffer.

This new interface is demonstrated in the following example, where a nonblocking receive is posted to receive into a host buffer and a send is performed from a GPU buffer.

```
MPIGPU_Irecv(host_buf, MPIGPU_HOST_CPU, count,
  datatype, left_neighbor, tag, comm, &request);

MPIGPU_Send(dev_buf, MPIGPU_BUF_GPU, count,
  datatype, right_neighbor, tag, comm);
```

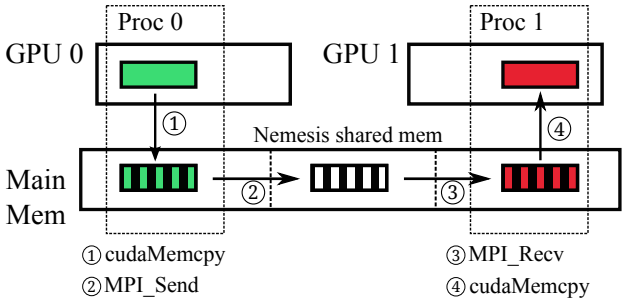
B. Eliminating Unnecessary Copies

Integrating support for GPU buffers into the MPI interface not only enhances productivity but also can significantly improve performance by allowing the implementation to eliminate unnecessary copying of data into temporary buffers. In Figure 2(a) we show the data movement corresponding to the manual mixed CUDA+MPI code from Listing 1. In this example, which is currently how all CUDA+MPI codes are written, a user-managed host-side buffer is used to help transfer data buffers residing in GPU memory. Data is first transferred from the device to a temporary buffer; next Nemesis copies this data into an internal buffer that is accessible to both processes; it is copied again into a temporary user buffer; and finally it is transferred to the device. Thus, when both the source and destination buffers reside on the GPU, two memory copies and two PCIe data transfers are involved.

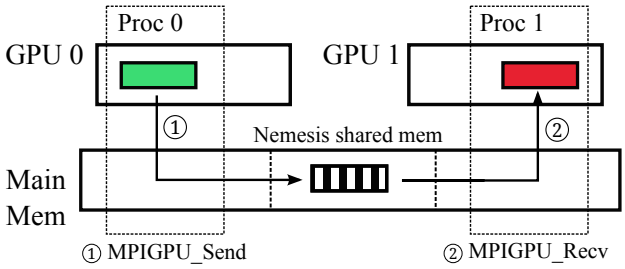
By integrating support for GPU buffers into MPI, we can eliminate the unnecessary user-level staging of data between MPI and the device, as shown in Figure 2(b). Now MPI takes the responsibility for moving data from device memory. It will first check the buffer type, and then transport the given data directly to the shared buffer used for Nemesis interprocess communication. As we demonstrate in Section V-A, this optimization significantly reduces the latency of communication operations, especially for latency-sensitive small messages.

C. Increasing Throughput with Pipelining

While eliminating memory copies is extremely beneficial for small messages transported by using the eager mode protocol, optimizing the rendezvous mode protocol is the key to providing good bandwidth for large messages. The LMT



(a) CUDA+MPI mixed programming: through a host-side buffer.



(b) MPIGPU: Eliminating unnecessary copies.

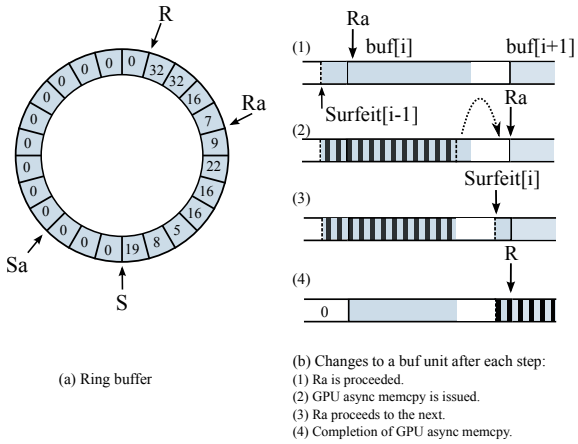
Figure 2. Memory copies in (a) CUDA+MPI and (b) MPIGPU.

protocol for rendezvous mode in Nemesis is implemented through a shared-memory copy buffer, which is created in a virtual channel between every pair of MPI processes in a given node. The copy buffer is managed as a ring buffer, as shown in Figure 3(a), where a copy buffer length for each buffer unit represents its full/empty state.

When a message is about to be transported through this ring buffer, the sender packs the message data into multiple segments and tries to copy them into as many consecutive available ring buffer units as possible; the receiver serially unpacks the consecutive buffer units out of the ring buffer and into the user-designated receive buffer. The data segmentation in Nemesis is done according to the maximum length of a copy buffer unit and the MPI data type. In MPICH2, a progress engine drives this in a nonblocking manner—the packing/unpacking progress can be paused, if the next buffer unit is unavailable, and resumed later; the progress engine can take the chance to drive other pending actions.

Modern GPUs are equipped with one or more DMA engines, which can be used to either synchronously or asynchronously transmit data between GPU device and page-locked main memory. Asynchronous DMA transfer has been shown to maximize the performance of the PCIe bus and achieve the highest bandwidth [3]. Thus, the Nemesis LMT shared buffers are page-locked to enable asynchronous DMA copies.

However, a challenge brought by asynchronous GPU memory copy in a nonblocking progress-engine-driven loop is, for both sender and receiver, to poll the ring buffer and



(a) Ring buffer

(b) Changes to a buf unit after each step:
(1) Ra is proceeded.
(2) GPU async memcpy is issued.
(3) Ra proceeds to the next.
(4) Completion of GPU async memcpy.

Figure 3. LMT shared copy buffer.

decide whether to pause the loop. Originally, a CPU-side memory copy is a synchronous action and returns with data ready in the destination. But now, we need to poll the status of both (1) availability of the buffer units, which are changed by the other party in a communication pair, and (2) completion of previously submitted asynchronous DMA memory copy. The two types of polling activity are interdependent: failing to poll the completion of DMA copy will delay the notification of availability of buffer units to the other party, and failing to check the newly available buffer units will delay starting new asynchronous DMA copy.

Under these requirements, we modify the algorithm (in the loop of sender and receiver both) by including two new $ahead$ pointer (Ra and Sa in Figure 3(a)) and start DMA asynchronous copies as eagerly as possible. That is, whenever there is an available buffer (empty for sender, or nonempty for receiver), a new DMA copy is issued, and the $ahead$ pointer (Sa for sender, Ra for receiver) proceeds; but the length of that buffer unit is not changed until the completion of the DMA command. Here we put priority on availability of buffer units over timely checking DMA completion. This strategy keeps the buffering throughput efficient, since the PCIe latency is much larger than that of main memory operations, and floods the DMA engine, which typically can combine DMA requests or at least reduce idle time by pipelining operations. The pseudo-code for the receive progress loop is shown in Algorithm 1; the send-side loop is similar. Note this is a multilevel nested loop, and the loading/saving-state activities (line 1 and 5) are designed for the re-entrant progress driven by the progress engine. The unpack function (line 14) eagerly starts the DMA asynchronous memory copy and handles data layout according to MPI types. The inner loop (line 3–11) holds the progression of Ra under either of the following circumstances: (1) there is more data to receive, but the sender has not put data in the buffer, or the pending DMA commands has reached a maximum threshold; or (2) there is no more data to receive, and there are pending DMA commands. The DMA maximum pending threshold is a parameter to decide the level of compromising, for the eagerness of initiating DMA copies, the timeliness of checking DMA completion, and, in turn, the timeliness of notifying the availability of buffer units to the other party.

Algorithm 1: Pseudo-code of receive progress loop

```
1  load state;
2  repeat
3      while (more data AND (copybuf_len[Ra] == 0 OR
       pending dma reaches max)) OR (no more data
       AND pending dma) do
4          if failed for a certain number of polls then
5              save state and exit;
6          while query dma of R finish do
7              if surfeit[R-1] != 0 then
8                  copybuf_len[R-1] := 0;
9              if surfeit[R] = 0 then
10                 copybuf_len[R] := 0;
11             proceed R;
12     if more data then
13         proceed Ra;
14         unpack(buf[Ra] - surfeit, &surfeit);
15         surfeit[Ra] := surfeit;
16         if surfeit != 0 then
17             copy surfeit buf before buf[Ra+1];
18
19 until no more data AND no pending DMA;
```

With a specific MPI data type, a single element may cross two neighboring units. However, data segment pack/unpack functions work only at the boundaries of fundamental data types. Thus, surfeit data, which belongs to the next data element, may not be copied out of the shared memory buffer. The surfeit is decided upon return of unpack, even if its asynchronous DMA request is proceeding. When detected, it can be immediately moved to the end of this unit, making it a contiguous buffer with the available data in the next buffer unit. Thus, Ra can proceed proactively. Later, when a DMA command is found complete, the previous buffer unit's length can be set to 0. This is shown in Figure 3(b). Also note that this surfeit occurring in the buffer unit at $R-1$ can make us lose one unit, because the postponed DMA polling for R, due to proactive DMA initiation, may not clear its buffer length in time, if the DMA maximum pending threshold is set to the number of buffer units.

D. Efficient Host-Device Data Movement

GPU memory, the on-board device memory, is separated from the main memory on the host. In spite of the recent

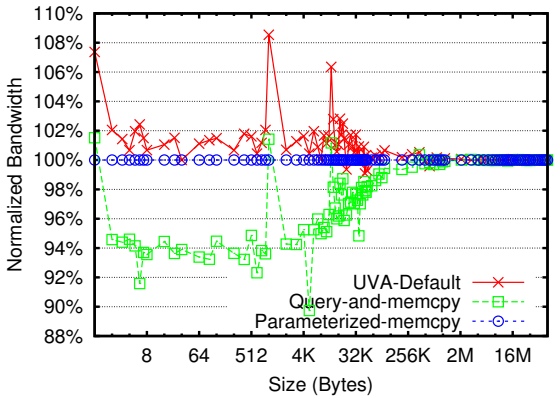
CUDA UVA and GPUDirect technologies [3], special GPU memory copy commands, for example, cudaMemcpy (which are different from memcpy on the host side), must be used to handle the memory copies involving GPU memory. These commands utilize the DMA engine on the GPU through interaction with the GPU driver. Typically these GPU memory copy commands ask for the location of source and destination buffers, in main memory or GPU device memory, as function arguments. The different memory copying methods make it challenging to introduce GPU memory awareness to MPICH2, which assumes a single memory space. When a memory copy needs to be initiated, the locations of source and destination buffers must be known beforehand to determine whether normal memcpy or GPU memory copy should be used.

The recent CUDA UVA technology provides programmers with a logical universal virtual address space to make programming easier. It maps GPU device memory buffers to a part of the virtual address space of a process so that the driver and the runtime library can tell where a buffer is located using its address. This provides developers with a pointer query API and an easier version of cudaMemcpy, the latter of which, asking for only a default argument, can determine the location of a buffer by itself and perform a transparent memory copy. Though the proprietary driver is a black box, we speculate that the default version of cudaMemcpy API embeds something similar to that done for the pointer query API before starting the actual cudaMemcpy action with correct buffer location parameters.

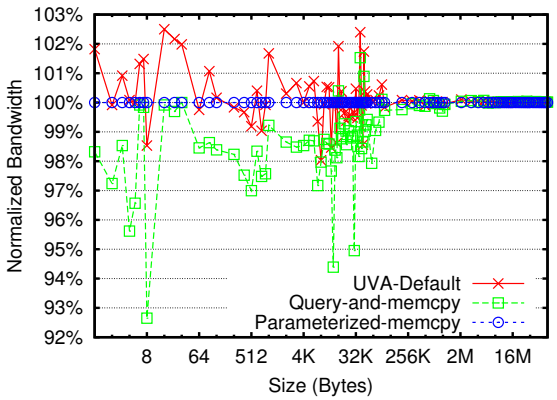
Therefore, adding support for GPU memory copy into MPICH2 can be done in three ways:

- **UVA-Default:** This method simply replaces all memcpy with cudaMemcpy with the default parameter, relying on UVA and the GPU driver to perform device-host and host-host copy operations. This is the easiest method for extending MPICH2.
- **Query-and-memcpy:** This method queries the buffer location with the driver at run time. After a query, it can be decided whether to use normal memcpy or a GPU memory copy command.
- **Parameterized-memcpy:** This method stores the location of a buffer pointer and passes it around with the pointer within MPICH2. Any memory copy that may involve a GPU-resident pointer will be modified to check this location first and decide to use normal memcpy or a GPU memory copy command.

The first two approaches require less modification to MPICH2's internals. However, they may introduce extra overhead of interaction with the GPU driver. In Figure 4, we measured the bandwidth of moving data across host and a GPU device with all three methods (using the same system setup as in Section V). While the overhead is less important when the copied data sizes are increased, for data



(a) Host to Device



(b) Device to Host

Figure 4. Bandwidth of different GPU memory copy methods between host and GPU device. Adapted from from CUDA SDK bandwidth test [8]. Host buffer is pinned always.

sizes smaller than 64 KB, Query-and-memcpy shows this overhead, more clearly in the direction of "Host to Device." Interestingly, the UVA-Default method does not show any overhead. The difference here is that the Query-and-memcpy method enters the driver twice, for pointer querying and the GPU memory copy command, whereas UVA-Default and Parameterized-memcpy do so only once and probably cause less operating system overhead due to privilege level change.

Although UVA-Default seems to be a good option, a factor that one cannot neglect is the performance on the host-side memory copy, if it were used to replace all memcpy in MPICH2. The CPU-side memcpy in MPICH2, namely, MPIU_Memcpy, is actually an efficient implementation [9] that tries to optimize both memory copy performance and the impact on cache behavior of the computation. When an MPI communication primitive is called, it may evict the data stored in CPU cache, which in turn affects the performance of computation after the copy completes. Therefore, the less such cache disruption an MPI implementation does, the more easily an MPI program developer can analyze the

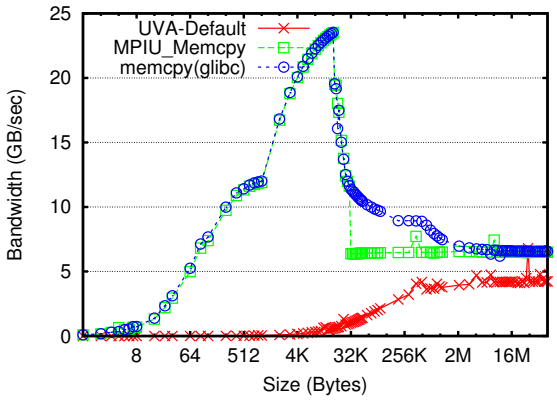


Figure 5. Bandwidth of different memory copy methods on the host side. Adapted from bw_mem bcopy from lmbench [10]. (MPIU_Memcpy is turned on for x86 but not x86-64 by default.)

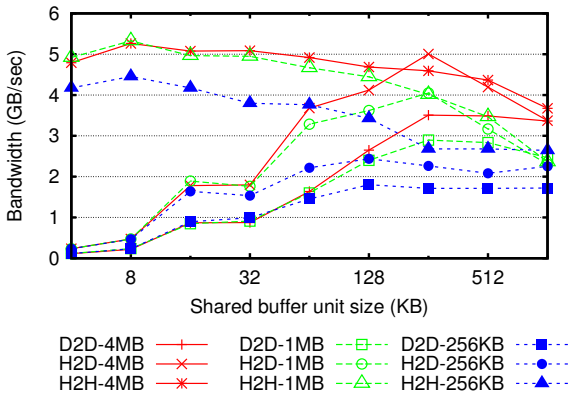


Figure 6. Bandwidth of two intranode MPI processes, with different shared buffer unit sizes, using messages at 4 MB, 1 MB, and 256 KB. Source and destination buffers both in GPU memory("D2D"), both in main memory ("H2H"), or source in main memory and destination in GPU memory ("H2D"); the result of "D2H" is similar to "H2D."

performance of his own code. In MPICH2, MPIU_Memcpy uses nontemporal instructions to alleviate the impact for large enough copy data sizes (i.e., 64 KB and beyond).

In Figure 5, we show the bandwidth of host-side memory copy using UVA-Default, MPIU_Memcpy, and memcpy (from glibc), on the same system as above. As MPIU_Memcpy falls back to memcpy before 64KB, they overlap until non-temporal version is used. The bandwidth of memcpy in the mid-range is larger than the actual memory bandwidth, which is measured in larger sizes, and is due to the hardware cache behavior of the microbenchamrk. The true memory bandwidth is shown when MPIU_Memcpy starts the nontemporal implementation but appears more gradually in memcpy. However, UVA-Default does not behave as well as the other two. The reason is probably that it does too much work in checking the location of destination and source buffers, and this work disturbs hardware cache a lot even for small data copy sizes. This overhead is not shown in the previous experiment, however, because the PCIe latency dominates it in that case but here memory and cache latency is smaller.

In addition to performance considerations, we decided to adopt Parameterized-memcpy for portability reasons to support GPU programming models that do not provide UVA.

V. Evaluation

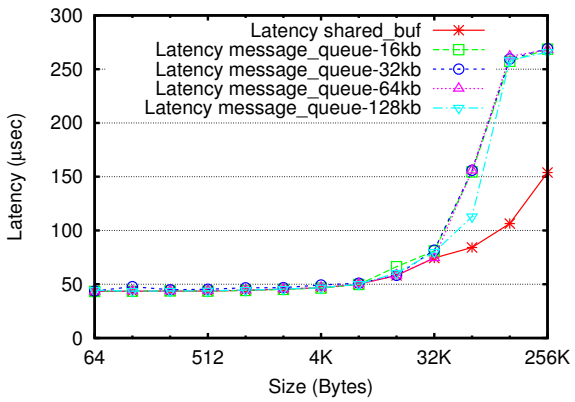
Our evaluation in this section and the previous sections is conducted on Keeneland [11] cluster, a National Science Foundation Track2D Experimental System based on the HP SL390 powered with Nvidia Tesla M2070 GPUs in Oak Ridge National Laboratory. Each compute node in Keeneland has two Intel Xeon X5660 CPUs, 24 GB main memory, 3 GPU devices connected through 2 IO hubs; nodes are connected via single rail, QDR Infiniband. The software environment is CentOS release 5.5 (Final) with Linux kernel 2.6.18-194.el5.perfctr, CUDA driver/runtime v4.0.

A. Exploration of Buffering and Message Queue Parameters

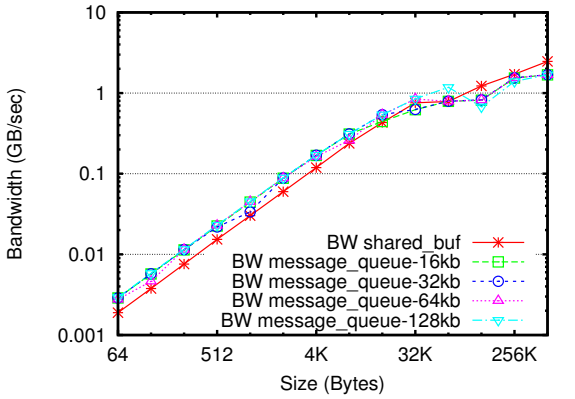
We present an exploration of the parameter space for the shared buffer and message queue. Using the ideal parameters for our experimental platform, we measure performance and compare it with a baseline manual code similar to that presented in Listing 1. Performance measurements were taken using the latency and bandwidth tests from OSU microbenchmark suite [6].

First, we determine the shared buffer unit size for LMT shared buffer using bandwidth test. The results are shown in Figure 6 for 4 MB, 1 MB, and 256 KB messages, which are points of interest for MPI messaging and are representative of trend. First, we notice that different trends are shown when GPU memory is used or not (as we compare "H2H" to "D2D"/"H2D"); this result suggests that different parameters should be used for host-only and host-GPU transfers. Second, as long as GPU memory is used, on one side or both, the best bandwidth can be attained only with a 128 KB or larger shared buffer unit size, since small-sized data transmission on PCIe bus is inefficient. The optimal point occurs at 256 KB, with the exception of 128 KB for 256 KB messages. Considering the possibility of even larger messages, we chose 256 KB for this parameter. Third, for host-only messages, the peak bandwidth is reached with smaller unit sizes (32 KB by default now in Nemesis). The reason is that because main memory has much lower access latency than does the PCIe bus, moving fine-grained chunks in shared buffer can increase parallelism of the pipeline. Therefore, we should let the shared buffer fall back to its original parameter value when no GPU memory is involved. And this information regarding the location of buffer of the other side can be exchanged in the LMT's handshaking step.

Using this shared buffer parameter setting, we try to cap the performance curves of eager message queues, in order to



(a) Latency versus Message Size



(b) Bandwidth versus Message Size

Figure 7. Latency/bandwidth of two intranode MPI processes, source and destination buffers both in GPU memory, using different message queue element sizes. The optimal shared buffer is shown for comparison. The results of "Device-to-Host" and "Host-to-Device" are similar.

find the best message queue element sizes and the threshold of switching modes. In Figure 7, latency and bandwidth of message queues at 16 KB, 32 KB, 64 KB, and 128 KB are shown together with that of the shared buffer. The latency of the message queue has no significant advantage over that of the shared buffer when the GPU buffer is used, because the PCIe bus latency is dominant. However, the message queue can have a larger number of cells than the number of shared buffer unit sizes, because of its scalable linear-complexity nature. Therefore it shows better bandwidth, simply because it can hold more simultaneous requests. With regard to latency or bandwidth, the message queue loses to the shared buffer beyond the point of 64 KB, which we choose as the threshold of switching modes. When the message queue is used for sizes below that, we choose 64 KB as the message queue element size, which is also the original value now in the system for host-side messages.

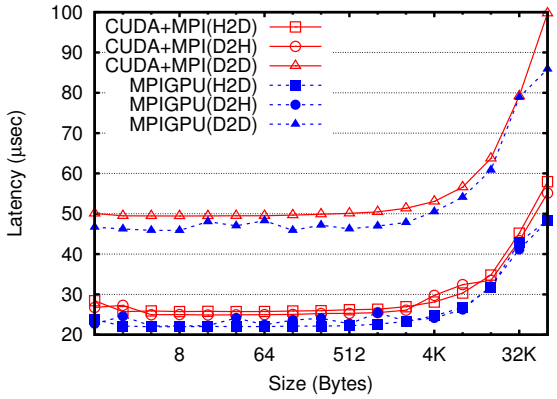
B. Performance Comparison with Manual CUDA+MPI

Using this set of parameters, we compare the latency and bandwidth of MPIGPU with manual CUDA+MPI where the user must manually move data between host and device in Figure 8. The benefit of latency for small messages, from eliminating one or two main-memory copy, is bounded by the memory latency. On average, for messages smaller than or equal to 64 KB, latency improvement is 6.4%, 15.7%, and 10.9% for "D2D," "H2D," and "D2H," respectively. With full range of size till 4 MB, the improvement is increased to 24.5%, 31.9%, and 23.9%, respectively. The benefit of bandwidth increases with messages sizes and achieves up to 2x speedup over manual mixing CUDA with MPI for all three cases shown. On average, for messages between 64 KB and 4 MB, the improvement is 56.5%, 48.7%, and 27.9% for "D2D," "H2D," and "D2H," respectively; with full range of size between 1 byte and 4 MB, the improvement is 18.1%, 27.5%, and 10.2%, respectively. While the peak "D2D" bandwidth is around 60% of the theoretical one—that is, the smaller of bandwidth on PCIe and main memory assuming a fully pipelined conduit—the "D2H" bandwidth efficiency is nearly 90%, almost saturating the theoretical bandwidth. The efficiency in the reverse direction is not as large because, on this system, measurement shows larger device-to-host PCIe bandwidth than the other direction.

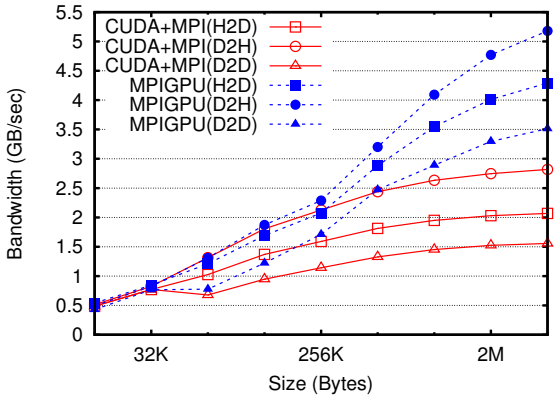
C. Evaluation with 2-D Stencil Benchmark

Stencil2D from SHOC benchmark suite [12] measures the performance of a halo-type, nine-point, two-dimensional stencil computation. It performs an iterative stencil computation on the GPU and requires a halo 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 a N-d array. Periodically, a process must obtain the values that its neighbors have calculated for the array elements that neighbor 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.

In Figure 9, the mean time speedup of using MPIGPU against the original manual version is shown. Also shown is the percentage of execution time spent on communication. In general, we have seen an average 4.3% improvement in the total execution time of stencil2d. Grouped by different number of processes, an averaged improvement of 5.8%, 5.6%, and 5.0% are shown for 4, 6, and 8 processes respectively, except for the case of two processes, where only a 0.69% improvement is shown. This is probably because Stencil2D exchanges a vector-typed data along one dimension, which leads to a number of small PCIe requests. Thus, more processes leads to more asynchronous data transmission commands along this dimension and hence a larger potential for DMA engines to combine these small requests than in the case of only two processes. Also note that the percentage



(a) Latency versus Message Size



(b) Bandwidth versus Message Size

Figure 8. Comparing MPIGPU with CUDA+MPI manual, on latency/bandwidth of two intranode MPI processes. Source and destination buffers both in GPU memory ("D2D"), source in main memory and destination in GPU memory ("H2D"), and source in GPU memory and destination in host memory ("D2H").

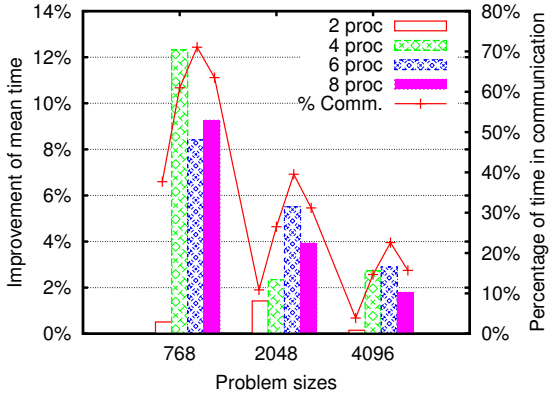


Figure 9. Stencil2D performance improvement of MPIGPU over CUDA+MPI. The problem size is fixed when the number of processes are varied.

of the execution time spent in communication decreases when we increase the problem size, because in Stencil2D the computation grows quadratically with the problem size and the communication grows linearly. This limits the MPIGPU's potential benefit when we increase the problem size. Since a node has only three GPU devices, when the number of processes is larger than that of the devices, the tasks are assigned in a round-robin order. And a context-sharing overhead is introduced when multiple processes are using the same GPU device. This is the reason the 8-process cases have a smaller percentage of communication than do the others (although they have more data to exchange) and hence less improvement, shown in the medium and large problem size. The trend is not shown in the small problem size, where the tiny (6144 bytes) exchanges data size and the smaller number of noncontiguous PCIe data transfers cause other latency-induced overhead.

VI. Related Work

Because of the increasing popularity of using GPU or other accelerators for HPC programs, MPI, as an evolving standard, has a natural demand to make it accelerator-friendly. Recently, Stuart et al. proposed several potential directions for extending the MPI standard to provide native support of these accelerators [13]. One significant propsed extension is to allow accelerators to obtain MPI ranks and participate directly in MPI operations. In comparison, our work operates within the current MPI standard and proposes a method of integration that greatly improves intranode communication performance and productivity for current GPU-accelerated applications.

Similar work to ours has recently been done to add support for CUDA devices to MVAPICH2, which is based on MPICH2 and optimized for InfiniBand networks. This work has focused on MPI point-to-point communication for internode GPU communication [5], all-to-all communication [14], and noncontiguous-type communication [15]. Similar work has also been proposed in the context of OpenMPI [16]. However, performance optimization for intranode MPI communication with GPU native buffer support has not been addressed and is the focus of this paper.

GPUDirect [17] is a set of techniques for performance improvement on NVIDIA GPUs. It includes eliminating one-extra copy between two system memories for RDMA communication of GPU pinned host buffers, peer-to-peer memory access and peer-to-peer data transmission between GPU devices within one process. cudaIPC, a new feature in CUDA 4.1 [18], can improve device-to-device MPI communication when present; yet this feature is not available in other GPU programing models and may not be helpful for device-to-host and host-to-device MPI communication.

VII. Concluding Remarks

GPU and other accelerators are emerging computation hardware in HPC systems. In the past few years we have seen rapid evolution of GPUs, with new architectural functionality and software features every season or half year. With increasing interest in these accelerators, we expect systems softwares, such as the MPI communication system, to have a more pressing need to extend native support of GPU and other accelerators for application development.

In this work, we propose a uniform MPI communication interface for a GPU-accelerated system, where user-specified buffers can reside in traditional main memory or in GPU memory. We present the design and implementation of integrating transparent GPU-awareness into MPICH2, a popular MPI implementation. With a focus on intranode communication, we further improve the throughput by taking advantage of the DMA engines on GPU and adapting the ring buffer pipeline of Nemesis communication subsystem. We also highlight the importance of choosing an efficient memory copy method. The augmented MPICH2-GPU system is evaluated through both microbenchmarking and a halo exchange benchmark and shows 23.9–31.9% latency improvement and 10.2–27.5% bandwidth improvement, for different source and destination buffer locations, and 4.3% average performance improvement in the halo exchange benchmark.

Acknowledgment

This work has been sponsored in part by an NSF CAREER Award (CNS-0546301), an NSF award (CNS-0915861), Xiaosong Mas joint appointment between NCSU and ORNL, and the U.S. Department of Energy under Contract DE-AC02-06CH11357. This research has also used resources of 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] *MPI: A Message-Passing Interface Standard Version 2.2*. Message Passing Interface Forum, 2009.

[3] Nvidia, "NVIDIA CUDA C Programming Guide version 4.0," May 2011.

[4] D. Buntinas, G. Mercier, and W. Gropp, "Implementation and shared-memory evaluation of MPICH2 over the Nemesis communication subsystem," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, B. Mohr, J. Trff, J. Worringen, and J. Dongarra, Eds. Springer Berlin / Heidelberg, 2006, vol. 4192, pp. 86–95, doi 10.1007/11846802_19.

[5] 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 (ISC) '11*, 2011.

[6] "OSU Micro-benchmarks 3.5," 2011, http://mvapich.cse.ohio-state.edu/benchmarks/.

[7] Khronos Group, "OpenCL 1.2," http://www.khronos.org/opencl/.

[8] "CUDA SDK version 4.0," May 2011, http://developer.nvidia.com/cuda-toolkit-40.

[9] D. Buntinas, G. Mercier, and W. Gropp, "Data transfers between processes in an SMP system: Performance study and application to MPI," in *International Conference on Parallel Processing, 2006. ICPP 2006.*, August 2006, pp. 487–496.

[10] L. McVoy and C. Staelin, "lmbench: portable tools for performance analysis," in *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 1996, pp. 23–23.

[11] 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.

[12] 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 *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU '10. New York: ACM, 2010, pp. 63–74.

[13] J. A. Stuart, P. Balaji, and J. D. Owens, "Extending MPI to accelerators," *PACT 2011 Workshop Series: Architectures and Systems for Big Data*, Oct. 2011.

[14] 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 (PPAC '11), held in conjunction with Cluster '11*, Sept. 2011.

[15] 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.

[16] E. Gabriel, G. Fagg, G. Bosilca, T. Angskun, J. Dongarra, J. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. Castain, D. Daniel, R. Graham, and T. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, D. Kranzlmller, P. Kacsuk, and J. Dongarra, Eds. Springer Berlin / Heidelberg, 2004, vol. 3241, pp. 353–377.

[17] "NVIDIA GPUDirect," http://developer.nvidia.com/gpudirect.

[18] "NVIDIA CUDA toolkit 4.1 release candidate 2," http://developer.nvidia.com/cuda-toolkit-41.