Comput Sci Res Dev (2011) 26: 257–266
DOI 10.1007/s00450-011-0171-3

SPECIAL ISSUE PAPER

MVAPICH2-GPU: optimized GPU to GPU communication for InfiniBand clusters

Hao Wang · Sreeram Potluri · Miao Luo · Ashish Kumar Singh · Sayantan Sur · Dhabaleswar K. Panda

Published online: 12 April 2011

Abstract Data parallel architectures, such as General Purpose Graphics Units (GPGPUs) have seen a tremendous rise in their application for High End Computing. However, data movement in and out of GPGPUs remain the biggest hurdle to overall performance and programmer productivity. Applications executing on a cluster with GPUs have to manage data movement using CUDA in addition to MPI, the de-facto parallel programming standard. Currently, data movement with CUDA and MPI libraries is not integrated and it is not as efficient as possible. In addition, MPI-2 one sided communication does not work for windows in GPU memory, as there is no way to remotely get or put data from GPU memory in a one-sided manner.

In this paper, we propose a novel MPI design that integrates CUDA data movement transparently with MPI. The programmer is presented with one MPI interface that can communicate to and from GPUs. Data movement from GPU and network can now be overlapped. The proposed design is incorporated into the MVAPICH2 library. To the best of our knowledge, this is the first work of its kind to enable advanced MPI features and optimized pipelining in a widely used MPI library. We observe up to 45% improvement in one-way latency. In addition, we show that collective communication performance can be improved significantly: 32%, 37% and 30% improvement for Scatter, Gather and Allotall collective operations, respectively. Further, we enable MPI-2 one sided communication with GPUs. We observe up to 45% improvement for Put and Get operations.

Keywords MPI · Clusters · GPGPU · CUDA · InfiniBand

This research is supported in part by U.S. Department of Energy grants #DE-FC02-06ER25749 and #DE-FC02-06ER25755; National Science Foundation grants #CCF-0833169, #CCF-0916302, #OCI-0926691 and #CCF-0937842; grant from Wright Center for Innovation #WCI04-010-OSU-0; grants from Intel, Mellanox, Cisco, QLogic, and Sun Microsytems.

H. Wang (✉) · S. Potluri · M. Luo · A.K. Singh · S. Sur · D.K. Panda
Department of Computer Science and Engineering, The Ohio State University, Columbus, USA
e-mail: wangh@cse.ohio-state.edu

S. Potluri
e-mail: potluri@cse.ohio-state.edu

M. Luo
e-mail: luom@cse.ohio-state.edu

A.K. Singh
e-mail: singhas@cse.ohio-state.edu

S. Sur
e-mail: surs@cse.ohio-state.edu

D.K. Panda
e-mail: panda@cse.ohio-state.edu

1 Introduction

We have witnessed a dramatic increase of data parallel architectures, such as Graphics Processing Units (GPUs) in the field of High End Computing (HEC). The #1 system in the Top500 list [1] leverages GPU technology for record-breaking performance. The Compute Unified Device Architecture (CUDA) is one of the most popular programming models for NVIDIA GPUs. CUDA provides methods to invoke computation functions on GPUs, move data between CPU and GPU and synchronize threads on GPU. Typically, GPUs are connected as peripheral devices on PCI Express. Even though PCI Express is very fast (x16 PCIe 2.0 provides 8 GB/s bandwidth in each direction) it is still slow compared to the compute capabilities of GPUs. In particular, one of the main limiting factors of GPU enabled applications is the latency between CPU and GPU memory transfers. Data movement between GPU device memory and host main memory must be performed manually using CUDA library.

The Message Passing Interface (MPI) is one of the most popular parallel programming models. Parallel applications written in MPI can leverage CUDA to tap into GPU performance advantages. However, in such a situation, the programmer needs to manage data movement between GPU device memory and host memory. If data in GPU device memory needs to be communicated out of the node, it needs to be brought into the host memory. This is because current state-of-the-art MPI libraries can only access host memory to send and receive messages. Since the CUDA and MPI libraries are separate, there is a lack of co-ordination among the two. This results in two issues: (i) higher latency due to explicit staging requirements of memory buffers for point-to-point and collective communication, and (ii) lack of availability of advanced MPI features, such as one-sided communication.

There have been several efforts to tackle the first issue by explicitly pipelining data movement and CUDA kernel execution at the application-level [2–7]. However, these approaches can only improve the point-to-point performance to a certain extent. MPI collective operations are often implemented using complex platform specific algorithms that are highly tuned and only known to the MPI library. Since the collective communication routines are internal to MPI, application level pipelining for point-to-point operations does not apply to them directly. Thus, the only option to pipeline data movement at the application level would require a complete re-implementation of collective operations at the application-level. This is a prohibitive cost and not feasible. Also, advanced MPI-2 features, such as one-sided communication cannot be used currently, as there is no way to remotely get or put data from GPU memory in a one-sided manner. Further, application-level techniques increase code complexity for the user of MPI and CUDA libraries. Application-level techniques are not of a lasting nature, as the speed of data movement is expected to vary widely from platform to platform. Therefore, it is likely that the application optimized on one platform becomes non-optimal when the platform is upgraded or when the application is executed on another platform. We believe that one of the solutions to this problem is to integrate data movement to and from GPU device memory with the MPI library.

The MPI library has the opportunity to optimize the GPU data movement on one node and data transfer among multiple nodes at the same time. There are several open research questions which need to be answered before one can support GPU data movement inside MPI library in an efficient manner:

1. Can the programming be simplified when providing an MPI-level interface that supports GPU data movement inside MPI library directly?
2. Can the design within MPI library achieve its maximum potential and even outperform application-level pipelining techniques?
3. Can the benefits be observed at the MPI level with a set of benchmarks?

In this paper, we propose a set of new MPI-level designs to support GPU data movement directly with MPI interfaces and overlap GPU data movement and RDMA data transfer inside MPI library. The design and study is carried out with the popular MVAPICH2 [8] library for InfiniBand. To the best of our knowledge, MVAPICH2-GPU is the first work to support GPU to GPU communication using MPI. We compare a program snippet of using MVAPICH2-GPU to that of using MPI and CUDA separately to achieve pipelining. Through such comparison, we illustrate how MVAPICH2-GPU can ease complexity of GPU to GPU transfers. We present performance investigation on a GPU cluster with 8 nodes with MPI point-to-point, one-sided and collective operations. Our investigation reveals that MVAPICH2-GPU can significantly improve the performance when compared with overlap designs using the traditional MPI and CUDA interfaces: up to 45% latency improvement can be observed for one-way latency for message size of 4 MB along with 45% improvement for one sided Put and Get for message size of 4 MB. In addition, we observe 32%, 37% and 30% improvement for Scatter, Gather and Alltoall collective operations respectively. These improvements are observed with latest NVIDIA and Mellanox GPU-Direct technology. Similar benefits are also observed even when GPU-Direct is not installed on the cluster.

The rest of the paper is organized as follows. In Sect. 2, we provide background information about this topic. In Sect. 3, we describe the detailed design of MVAPICH2-GPU and its optimizations. Experimental results and discussions are presented in Sect. 4. We discuss related work in Sect. 5. Finally, we conclude the paper in Sect. 6.

2 Background

2.1 InfiniBand architecture

InfiniBand [9] is an industry standard switched fabric that is designed for interconnecting nodes in HEC clusters. The recently released TOP-500 rankings in November 2010 reveal that more than 40% of the computing systems use InfiniBand as their primary interconnect. Remote Direct Memory Access (RDMA) is one of the powerful features of InfiniBand. Using RDMA, communication can be performed without any host processor involvement. InfiniBand requires communication buffers to be registered. The HCA keeps the translation of virtual addresses to physical page locations.

2.2 GPU architecture and programming model

GPU is usually viewed as a data-parallel multi-core system. The Compute Unified Device Architecture (CUDA) [10] is a proprietary framework from NVIDIA to develop applications on GPU. The computational elements of algorithms written with CUDA are known as kernels consisting of many threads to execute the same program in parallel.

In a cluster environment GPUs are connected to individual nodes as PCIe devices. The nodes themselves are connected to each other via a high-performance network, such as InfiniBand. GPUs can only read/write memory attached with the host. The GPU pins down a block of main memory with CUDA interfaces, such as `cudaMallocHost()`. Before GPU kernel execution, data must be moved from this main memory into the device memory; and after the execution, results need to be moved out to the memory. This is done using `cudaMemcpy()`.

As mentioned in the previous sub-section, InfiniBand requires communication memory to be registered. Due to a limitation in the Linux kernel, it is not possible for two PCI devices to register the same page. Therefore, if GPU device memory needs to be communicated to a remote node, the following sequence is required: GPU memory $\rightarrow$ Host memory for GPU (using `cudaMemcpy()`) $\rightarrow$ Host memory for InfiniBand (using `memcpy()`)$\rightarrow$ InfiniBand Network. GPU Direct [11] is a Linux kernel patch that overcomes this limitation. Using GPU Direct, both network adapter and GPU can pin down the same buffer. Therefore, using GPU Direct, the following sequence is sufficient for network communication: GPU memory $\rightarrow$ Host memory (using `cudaMemcpy()`) $\rightarrow$ InfiniBand Network. GPU Direct cuts down one step in the communication process.

The latest release of CUDA, version 4.0, supports a new feature called Unified Virtual Addressing (UVA). UVA provides a single address space for all CPU and GPU memory that is allocated via the CUDA library. In addition, UVA can also be used to find out if a particular buffer was allocated in the GPU memory or in the host memory. This is very useful for MPI libraries, since the MPI standard does not require the user to obtain memory through any specific interface. Using UVA, MPI library can find out whether the buffer was allocated in the GPU memory.

2.3 MPI on InfiniBand

The Message Passing Interface (MPI) is the de-facto standard for parallel applications. Using MPI, parallel applications can send and receive messages and perform collective operations. MVAPICH and MVAPICH2 [8] are popular open-source implementations of MPI on InfiniBand, 10 Gigabit Ethernet/iWARP and the emerging RDMA over Converged Enhanced Ethernet (RoCE). In this paper, we use the MVAPICH2 software stack.

Point-to-point Communication: Using point-to-point communication in MPI, parallel applications can send arbitrarily sized messages. MVAPICH2 implements point-to-point messaging using RDMA. There are two protocols - RDMA Put and RDMA Get. In RDMA Put mechanism, the two processes perform a handshake using Request To Send (RTS) and Clear To Send (CTS) messages. When the sender receives CTS, it is able to issue RDMA write operation. Finally, the sender will send RDMA finish message to notify the receiver the RDMA write finish and the data is ready in the receive side.

Collective Communication: Using collective communication in MPI, parallel applications can perform group communication. Reduce, Alltoall, Gather are some of the heavily used operations. These operations are optimized by MPI library vendors by adapting to the specific system architecture.

One Sided Communication: Remote memory access was introduced by the MPI-2 specification. Using one-sided communication, applications can directly access remote memory. Before remote memory can be accessed, the application needs to create a "memory window" using MPI. In order to read the memory window after several one-sided operations, the application needs to call MPI synchronization functions. These functions will ensure that all incoming and outgoing network operations have completed. In the case the memory window is in GPU device memory, an MPI+CUDA application cannot directly access the memory on the device. This is because the MPI library cannot access the GPU memory. The only alternative is to have the application programmer manually move the window to host memory, while MPI one-sided operations take place. This is prohibitively expensive and in general not a feasible solution. In this paper, we present our novel design of an integrated MPI and CUDA library, using which one-sided GPU memory windows access is now possible.

3 Design

3.1 MVAPICH2-GPU interface

Figure 1 illustrates the data flow from a GPU device on one node to a GPU device on a remote node. Firstly, the source process copies data from its GPU device memory to the main memory (1). Then, it sends it to the remote process over the network (2). Finally, the target process copies the data from its main memory to its GPU device memory (3). The application programmer has to independently handle the copies from GPU memory to main memory using CUDA and the inter-process transfers through a communication model like MPI. Internally, most MPI implementations use the rendezvous protocol to handle large message

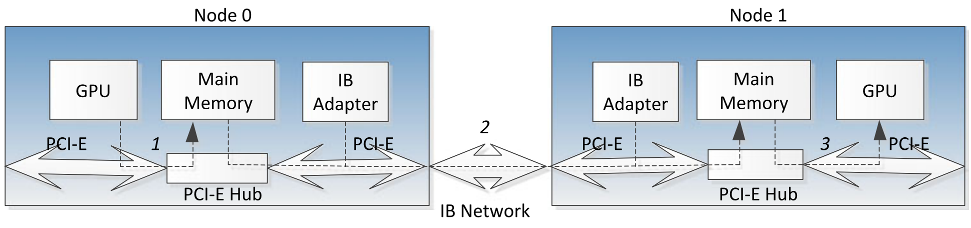


Fig. 1 Data flow from GPU memory to remote GPU memory

transfers. The target process provides the source with destination address information through a handshake. Then the source process directly writes to this address using RDMA and notifies the target process of the completion.

We have redesigned MPI to handle communication that involve GPU device memory with the standard MPI interfaces. We refer to this work as MVAPICH2-GPU. In order to directly use GPU memory address as parameters in standard MPI interfaces, we use UVA internally to detect a given memory address in the host memory or in the device memory. The three data transfers: GPU memory to host memory, network, host memory to GPU memory (on remote side) are all pipelined. This pipelining is internal to MVAPICH2-GPU. The application programmer remains unaware of this pipelining. For them, it is simply a normal MPI call.

Figure 2 illustrates pseudocode that compare the existing methods of GPU to GPU communication with that offered by MVAPICH2-GPU. Figure 2(a) shows a naive implementation using blocking CUDA memory copies and blocking MPI communication calls. Figure 2(b) shows how an application developer can improve the performance by carefully interleaving non-blocking memory copy and communication calls. Application and platform specific tuning is crucial to achieve maximum performance in this scenario. Figure 2(c) shows the ease in which a user can exploit overlap using MVAPICH2-GPU: the standard MPI_Send and MPI_Recv interfaces are used; and the underlying library takes care of optimally pipelining the memory copies and network transfers. MVAPICH2-GPU not only simplifies programming, but also has the potential for better performance as discussed in the following sections.

```
if(rank == 0) {
   cudaMemcpy(s_buf, s_device, size, cudaMemcpyDeviceToHost);
   MPI_Send(s_buf, size, MPI_CHAR, 1, 1, MPI_COMM_WORLD);
   ...
} else if(rank == 1) {
   MPI_Recv(r_buf, size, MPI_CHAR, 0, 1, MPI_COMM_WORLD, &reqstat);
   cudaMemcpy(r_device, r_buf, size, cudaMemcpyHostToDevice);
   ...
}
```

(a) Naïve MPI and CUDA without pipelining (good productivity, bad performance)

```
if(rank == 0) {
   for(j=0; j<pipeline_length; j++) {
      cudaMemcpyAsync(s_buf + j*block_size, s_device + j*block_size,
                block_size, cudaMemcpyDeviceToHost, cuda_stream[j]);
   }
   for(j=0; j<pipeline_length; j++) {
      while (result != cudaSuccess) {
         result = cudaStreamQuery(cuda_stream[j]);
         if(j>0) MPI_Test(sreq[j-1], &flag, &status[j-1]);
      }
      MPI_Isend(s_buf + j*block_size, block_size, MPI_CHAR, 1, 1,
                MPI_COMM_WORLD, &sreq[j]);
   }
   MPI_Waitall(pipeline_length, sreq, sstat);
   ...
} else if (rank == 1) {
   for(j=0; j<pipeline_length; j++) {
      MPI_Irecv(r_buf + j*block_size, block_size, MPI_CHAR, 0, 1,
                MPI_COMM_WORLD, &rreq[j]);
   }
   for(j=0; j<pipeline_length; j++) {
      MPI_Wait(&rreq[j], &rstat[j]);
      cudaMemcpyAsync(r_device + j*block_size, r_buf + j*block_size,
                block_size, cudaMemcpyHostToDevice, cuda_stream[j]);
   }
   for(j=0; j<pipeline_length; j++) {
      cudaStreamSynchronize(cuda_stream[j]);
   }
   ...
}
```

(b) Advanced MPI and CUDA with pipelining (low producitivity, good performance)

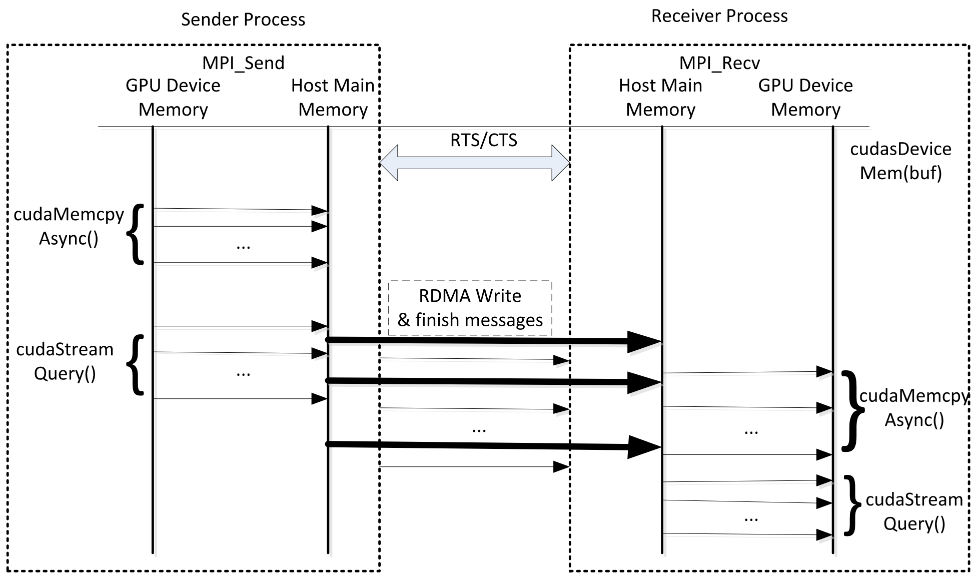
```
if(rank == 0) {
   MPI_Send(s_device, size, MPI_CHAR, 1, 1, MPI_COMM_WORLD);
   ...
} else if(rank == 1) {
   MPI_Recv(r_device, size, MPI_CHAR, 0, 1, MPI_COMM_WORLD, &reqstat);
   ...
}
```

(c) MVAPICH2-GPU (higest performance and productivity)

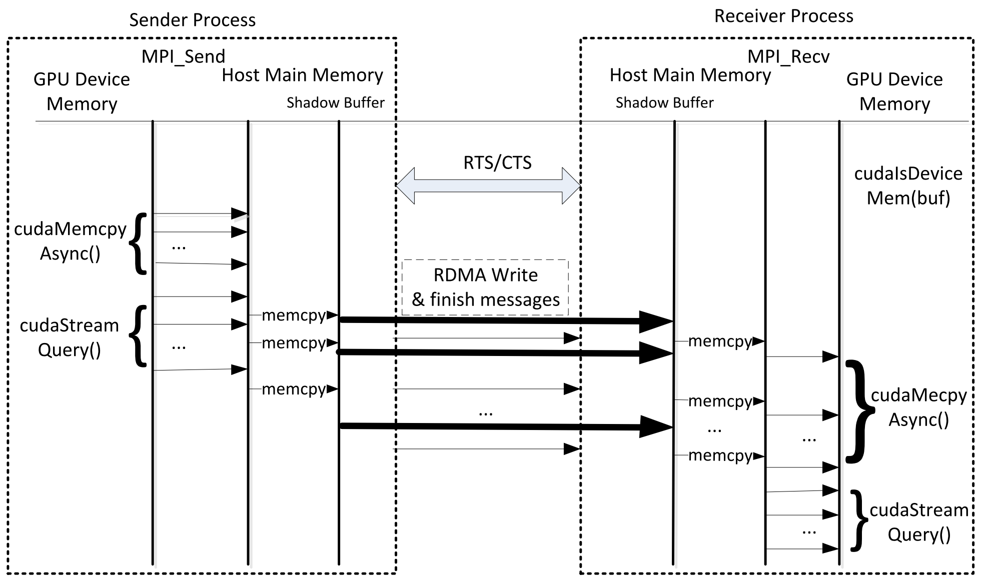
Fig. 2 Pseudocode comparing existing approaches and improved productivity using MVAPICH2-GPU

3.2 Pipelined design

We describe the pipelined design of MVAPICH2-GPU library in this section. Firstly, we consider the case in which data in one GPU device memory has to be sent to remote GPU device memory. The steps are illustrated in Fig. 3(a). The sender sends a RTS message to initialize the data transfer. After the receiver receives the RTS, it replies back with a CTS (based on MPI message matching semantics). The receiver also encodes the remote buffer address and size of the message to be received within the CTS message. The data is internally buffered at the sender and receiver host memory. These buffers are called *vbufs*, and are managed in a FIFO pool. The address encoded in the CTS message is that of a list of vbufs. When the CTS message is received by the sender, it starts asynchronous CUDA mem-



(a) MVAPICH2-GPU with GPU-Direct



(b) MVAPICH2-GPU without GPU-Direct

Fig. 3 MVAPICH2-GPU with GPU-Direct and without GPU-Direct

ory copy from device to host for each block (pipeline unit). The MPI library uses CUDA stream query function to check the status of each asynchronous memory copy. Once one of the asynchronous memory copies finishes, the sender calls InfiniBand verbs interface to perform the RDMA write. Finally, after each RDMA write finishes, the sender sends out a RDMA write finish message. When the receiver gets the RDMA write finish message, it starts the asynchronous CUDA memory copy to copy data from a vbuf to GPU device memory.

The above description is for the case where GPU direct is enabled. If GPU-Direct is not enabled, we use an additional *shadow_vbuf* to do one more memory copy as illustrated in Fig. 3(b). This additional memory copy is required since buffers cannot be shared between InfiniBand and GPU devices.

In the internal pipelined design, consideration of two factors is important for performance: the first factor is vbuf organization and its usage. With the traditional MPI interfaces and CUDA interfaces, some portion in main memory is allocated to communicate with GPU device's memory. The allocation, management and use of this memory is directly in the programmer's control. In our design, MVAPICH2-GPU is responsible for the allocation and management of this memory. The vbuf is allocated and freed in `MPI_Init()` and `MPI_Finalize()`, respectively. Once vbuf is allocated, it will be divided into blocks and organized as a buffer pool. During the handling of RTS, one vbuf block will be taken from the buffer pool. In order to keep total memory usage in control, one buffer is kept for each process ($O(N)$) rather than keeping one buffer for each connection between various node pairs ($O(N^2)$) involved in communication. The size of each buffer block may be smaller than the size of data to be sent as vbuf block. Additionally, vbufs are managed as a FIFO pool for RDMA.

The second factor is the usage of CUDA API for asynchronous memory copies. We use `cudaStreamQuery()` to do busy checking instead of blocking `cudaStream-Synchronize()` call. The CUDA stream query interface is a non-blocking operation to check whether pending DMA transactions are over. It returns quickly, regardless of whether a copy is complete or not. In contrast, the CUDA stream synchronize call blocks for particular copy to finish. In our design, the control returns back to MPI progress engine to handle other messages, which essentially avoids blocking of pipeline.

Design for One-Sided Communication: The existing MPI interface and implementations do not provide a feasible way of using one-sided communication for data in GPU device memory. The only way to achieve this is to create windows in the host memory and copy the complete window data out of and into the GPU device memory before and after each synchronization epoch. This is cost prohibitive. Our work directly addresses this limitation and enables full access to one-sided communication on GPU device memory.

In one-sided communication, the origin process (the process issuing the operation) provides the information about both the source and destination buffers. This is enabled by the window creation step during which all the processes in the MPI communicator initially exchange the addresses and size of their memory regions. When the buffers are in GPU memory, they can be directly used as a parameter in `MPI_Put` and `MPI_Get`. The Put operation is implemented using point-to-point protocol described in Sect. 3.2, with the exception that the source process provides the information about the target buffer in the RTS message. For Get operation, the origin process acquires a vbuf locally and sends its address information along with the target address information in the RTS message. The remote process on receiving the RTS message initiates the pipeline of copies and RDMA writes from the requested GPU buffer to the remote vbuf. Once a RDMA write completes, a "finish" message is sent. The origin process pipelines the copy from the vbuf to GPU memory as it receives the finish messages. This design involves copying only the required data out and into GPU device memory rather than the whole window.

Design for Collective Communication: MPI provides a collective interface which enables libraries to efficiently optimize commonly used communication patterns in applications. However, the collective interface provided by MPI-2 is blocking which makes it impossible to overlap GPU to host memory copies with communication. Since all MPI implementations build collectives on top of point-to-point transfers, our optimizations described in Sect. 3.2 can immediately benefit collectives. The MPI Forum is likely to include non-blocking communication in the upcoming MPI-3 specification. Even using non-blocking collectives, latency of communication and effective usage of RDMA will remain critical for performance. Our design is general, and applicable to non-blocking collectives as well.

4 Experiment and evaluation

We used two clusters for our experimental evaluation. The first cluster has eight nodes equipped with dual Intel Xeon Quad-core Westmere CPUs operating at 2.53 GHz and 12 GB of host memory. These nodes have Tesla C2050 GPUs with 3 GB DRAM. The InfiniBand HCAs used on this cluster are Mellanox QDR MT26428. Each node has Red Hat Linux 5.4, OFED 1.5.1, MVAPICH2-1.6RC2, and CUDA Toolkit 4.0. All the experiments with GPU-Direct were run on this cluster. The second cluster is Longhorn at Texas Advanced Computing Center. This cluster is used for all the experiments without GPU-Direct. The Longhorn cluster contains 256 compute nodes, 14.5 TB aggregate memory, 512 GPUs and a QDR Infiniband interconnect. Our MPI-level evaluation is based on OSU Micro Benchmarks [12]. We run one process per node and use one GPU per process for all experiments. We run the collective tests on eight nodes.

4.1 Choosing optimal pipeline unit for MVAPICH2-GPU

In the naïve approach where pipelining is not employed, the latency to transfer data of size N from GPU device memory to the remote GPU device memory is: $T\_cuda(N) + T\_mpi(N) + T\_cuda(N)$. T_cuda represents time spent in CUDA memory copies and T_mpi represents time spent in MPI library for the network transfer. Pipelining achieves best performance when latency of network transfer and latency of CUDA memory copies are overlapped by one another. The expression to model the latency of pipelined data transfer, where the data is divided into n blocks is: $T\_cuda(N/n) + n * T\_mpi(N/n) + T\_cuda(N/n)$. Figure 4 illustrates the latency for network transfer and the CUDA memory copy on the cluster with GPU Direct. We observe that, when the data size is equal to 128 KB, the network (RDMA write) latency is equal to CUDA memory copy latency on this platform. It can be expected that MVAPICH2-GPU will get better performance when the data size is larger than 128 KB where CUDA memory copy latency can be overlapped by RDMA write latency. Based on

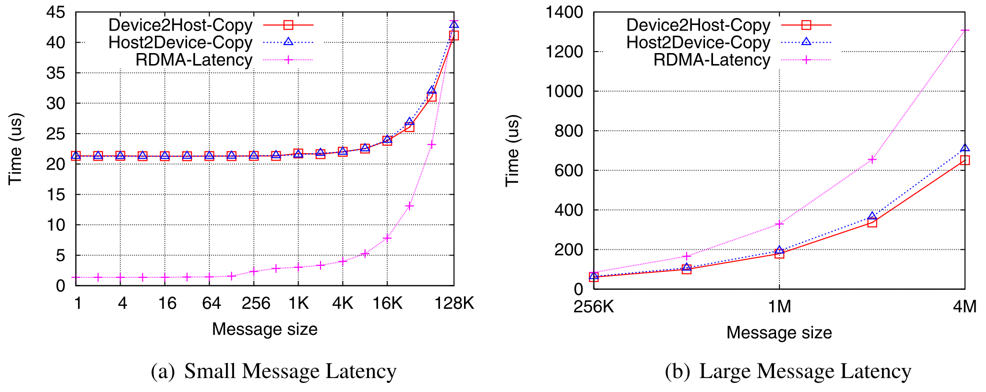


Fig. 4 Copy vs RDMA latency performance

(a) Small Message Latency (b) Large Message Latency

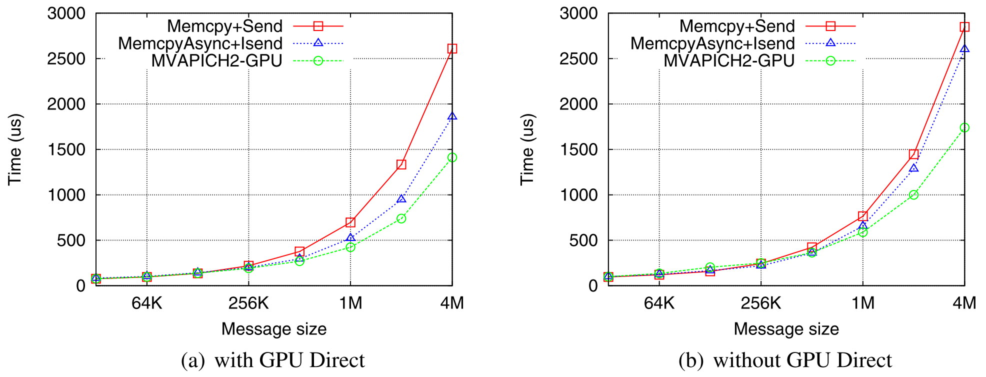


Fig. 5 Two-sided latency performance

(a) with GPU Direct (b) without GPU Direct

our experimentation, we found 128 KB and 256 KB to be the optimal block size for the OSU cluster and TACC Longhorn cluster, respectively. In the near future, when the newer PCIe 3.0 is used and CUDA memory copy cost is reduced, MVAPICH2-GPU is expected to get better performance for messages smaller than these sizes. The pipeline unit is presented as a configurable parameter of the MVAPICH2 library. This can be tuned once by the system administrator during the time of installation of the library by using OSU micro-benchmarks. Once the optimal value for the cluster is found, it can be placed in a configuration file (MVAPICH2 supports this), and end-users will transparently use this setting.

4.2 Two sided point-to-point communication evaluation

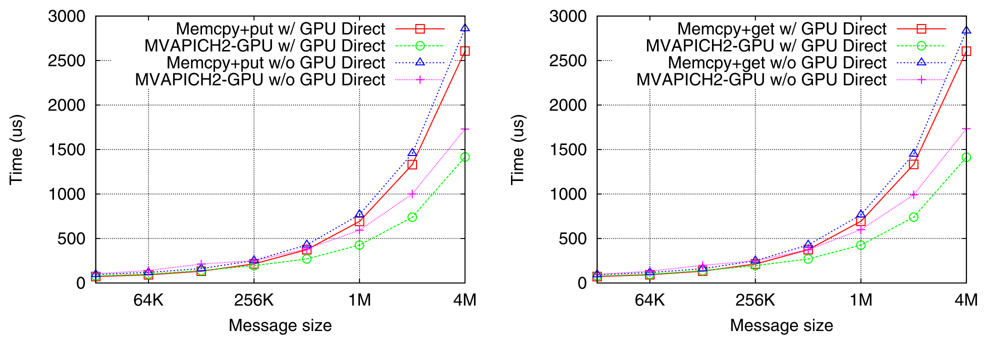
In this section, we present the evaluation for two-sided communication. We evaluate several combinations. Memcpy+Send is the method illustrated in Fig. 2(a) which uses blocking calls: `cudaMemcpy`, `MPI_Send`, and `MPI_Recv`. MemcpyAsync+Isend is the method shown in Fig. 2(b) that uses the corresponding non-blocking calls to implement the pipeline and achieve overlap. Finally, MVAPICH2-GPU represents the method using the integrated MPI and CUDA design proposed in this paper.

The results are presented in Fig. 5. Figure 5(a) compares GPU to GPU send receive latency with GPU-Direct. Figure 5(b) compares performance without GPU-Direct. These figures show that MVAPICH2-GPU consistently performs better than the optimized MPI-level implementation (i.e. optimizations are done by the application developer), MemcpyAsync+Isend. We achieve up to 24% and 33% improvement in latency with GPU-Direct and without GPU-Direct respectively for message size of 4 MB. Comparing with Memcpy+Send, the benefit is 45% and 38% respectively. MVAPICH2-GPU performs better than the best achievable performance at MPI-level as it avoids the overhead of MPI-level rendezvous handshakes, message matching and message processing for each of the pipeline messages. Moreover, implementing the pipeline internally enables us to issue non-blocking CUDA memory copy and RDMA write in an overlapped and fine-grained fashion. This improves the achievable overlap.

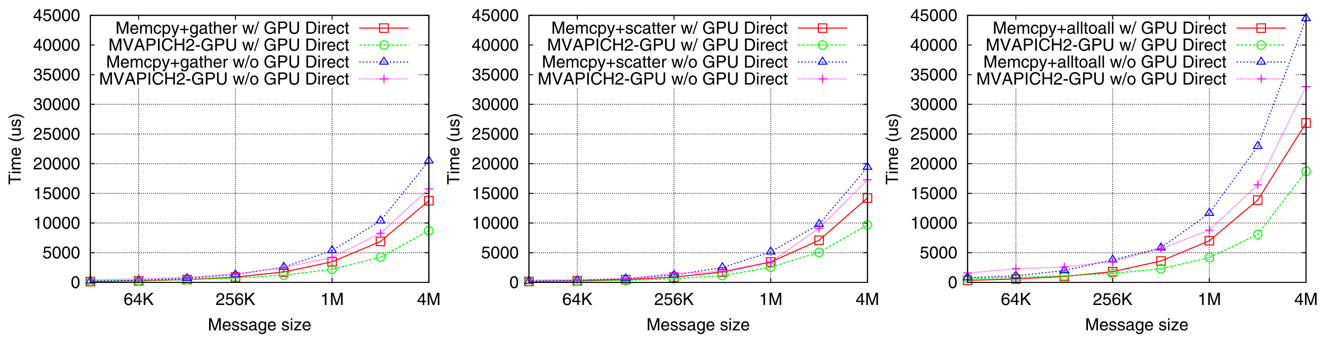
4.3 One-sided communication evaluation

As described earlier, one-sided communication cannot be effectively used for GPU-GPU transfers without extended support from the MPI integrated with CUDA. Without such support, one is required to copy the complete memory window out and into the GPU for each synchronization epoch, which is prohibitively expensive. In our benchmarks, we use window size equal to the size of the data being exchanged. MVAPICH2-GPU also supports the case where accesses are not of the entire window size, with similar per-

Fig. 6 One-sided latency performance



(a) MPI_Put latency (b) MPI_Get latency



(a) MPI_Gather latency (b) MPI_Scatter latency (c) MPI_Alltoall latency

Fig. 7 Collectives latency performance

formance benefits. Results in Fig. 6 show that for a message size of 4 MB MVAPICH2-GPU can achieve upto 45% improvement in Put and Get latency performance through overlap, when using GPU Direct. It achieves 39% improvement in performance without GPU Direct for message size of 4 MB. MVAPICH2-GPU performs better than Memcpy+put/get since it can overlap network transfers with local DMAs from GPU memory to Host memory.

4.4 Collective communication evaluation

Figure 7 compares the performance of three commonly used collective operations: Scatter, Gather, and Alltoall. Because of the blocking nature of collectives and their complex data exchange patterns, it is hard to pipeline the communication with the asynchronous CUDA memory copy by just using MPI interface (without CUDA integration). Often, the most optimal collective algorithm is platform dependent, since MPI library designers have spent a lot of effort to optimize it. For example, for Alltoall alone, MVAPICH2 uses three different algorithms based on the message size and the number of processes. In these figures, cudaMemcpy+collective is the method to copy the data in the GPU memory into the main memory, do the collective, and then copy the data back to the GPU memory. With GPU Direct we can observe a performance improvement of up to 32% for Scatter, 37% for Gather, 30% for Alltoall at message sizes of 4 MB, 1 MB and 4 MB, respectively. We see and improvement of 23%, 33%, 26% for the same cases without GPU Direct respectively.

5 Related work

Communication latency between host memory and GPU memory is one of the major bottlenecks in a GPU accelerator based system. Several researchers have attempted overlapping memory copy from GPU to host memory with kernel execution on GPU. One such example is the work by Ma et al. in [2]. Jacobsen et al. [3] have utilized CUDA and MPI interfaces to overlap GPU data transfer and MPI communication with computation to accelerate the computational fluid dynamics simulations. The similar method has also been used in [4, 5]. These approaches improve performance at the cost of productivity. Additionally, it requires careful evaluation and tuning. Moreover, the performance gains are not lasting, i.e. a platform upgrade may change the cost parameters. MVAPICH2-GPU provides a simplified method for programmers to perform GPU to GPU communication with highest performance.

Many researchers have proposed new programming models for GPU clusters. Fan et al. [6] have proposed Zippy framework to do computation and visualization on GPU cluster. Stuart et al. [7] have proposed multi-thread framework DGGN on GPU. However, they still depends on MPI to do the underlying inter node communication. These efforts will benefit from our work as we take care of pipelining internally.

AMD Fusion Family of Accelerated Processing Units [13] is an innovative architecture, where x86 cores, vector (SIMD) engines, and a Unified Video Decoder (UVD) are integrated in the same die. They are connected by a high performance bus, and shares coherent memory. Communication latency between host memory and GPU memory existing in current NVIDIA architecture will be alleviated. However, the issues related to integration of accelerator in applications still remains a challenge. In the future, we will work on integrating our work for AMD APU platforms as well.

6 Conclusions and future work

In this paper, we proposed MVAPICH2-GPU to support efficient GPU to GPU communication using MPI. MVAPICH2-GPU pipelines to overlap RDMA data transfer and CUDA memory copy inside MPI library. This achieves better performance than the application-level pipelining. Without MVAPICH2-GPU support, optimizing collective operations is a huge challenge for the application programmer. Further, we have enabled advanced MPI features, such as one-sided communication on GPUs. To the best of our knowledge, our work is the first to support efficient GPU to GPU communication using MPI.

Our investigation reveals that MVAPICH2-GPU can significantly improve performance. When GPU-Direct is deployed, we observe up to 45% improvement in point-to-point latency for message size of 4 MB. We also observe similar 45% improvement for one-sided communication with message size of 4 MB. Further, we show performance improvements of up to 32%, 37% and 30% for Scatter, Gather and Alltoall collective operations, respectively.

We intend to continue working in this direction. MVAPICH2-GPU developed in this paper will be integrated into future public MVAPICH2 releases. We also plan to evaluate the impact of MVAPICH2-GPU at the application level.

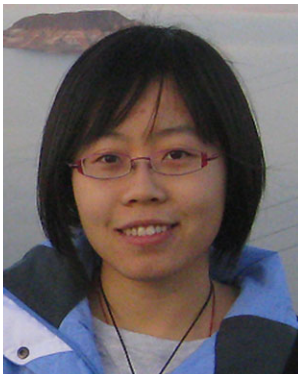
References

1. TOP500 Supercomputing Sites. http://www.top500.org/
2. Ma W, Krishnamoorthy S, Villa O, Kowalski K (2010) Acceleration of streamed tensor contraction expressions on GPGPU-based clusters. In: Proceedings of the 2010 IEEE international conference on cluster computing (Cluster'10)
3. Jacobsen DA, Thibault JC, Senocak I (2010) An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters. In: Proceedings of the 48th AIAA aerospace sciences meeting
4. Phillips EH, Fatica M (2010) Implementing the Himeno benchmark with CUDA on GPU clusters. In: Proceedings of the 24th IEEE international parallel and distributed processing symposium (IPDPS'10)
5. Choi JW, Singh A, Vuduc RW (2010) Model-driven autotuning of sparse matrix-vector multiply on GPUs. In: Proceedings of the 15th ACM SIGPLAN symposium on principles and practice of parallel programming (PPoPP'10), pp 115–126
6. Fan Z, Qiu F, Kaufman AE (2008) Zippy: a framework for computation and visualization on a GPU cluster. Comput. Graph. Forum 27(2):341–350
7. Stuart JA, Owens JD (2009) Message passing on data-parallel architectures. In: Proceedings of the 23th IEEE international parallel and distributed processing symposium (IPDPS'09)
8. MVAPICH2: High performance MPI over InfiniBand/10GigE/iWARP and RoCE. http://mvapich.cse.ohio-state.edu/
9. InfiniBand Trade Association. http://www.infinibandta.com
10. NVIDIA: NVIDIA CUDA compute unified device architecture. http://developer.download.nvidia.com/compute/cuda/2_0/docs/CudaReferenceManual_2.0.pdf
11. Mellanox: NVIDIA GPUDirect technology—accelerating GPU-based systems. http://www.mellanox.com/pdf/whitepapers/TB_GPU_Direct.pdf
12. OSU Micro Benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/
13. AMD: AMD fusion family of APUs: enabling a superior, immersive PC experience. http://sites.amd.com/us/Documents/48423B_fusion_whitepaper_WEB.pdf

Hao Wang is a Post-doctoral Researcher at the Department of Computer Science at The Ohio State University. His research interests include high performance computing, parallel computing architecture, and cloud computing. He is a member of the Network-Based Computing Laboratory lead by Dr. D.K. Panda. He is currently involved to design and development of MVAPICH2 (High Performance MPI over InfiniBand, 10GigE/iWARP and RoCE) being used by more than 1,500 organizations worldwide (in 60 countries). Dr. Wang received his Ph.D. degree from Institute of Computing Technology, Chinese Academy of Sciences (ICT, CAS) in 2008.

Sreeram Potluri is a Ph.D. Student in the Department of Computer Science and Engineering at The Ohio State University. His research interests include high speed interconnects, parallel programming models and highend computing applications. His recent work includes optimizing AWP-ODC, a widely used seismic modeling application, using MPI-1 Non-blocking and MPI-2 RMA semantics on large scale InfiniBand clusters. This work was published at ICS'10 and is part of the application's entry as a finalist for the 2010 Gordon Bell Prize. He is a member of the Network-Based Computing Laboratory lead by Dr. D.K. Panda. Sreeram is involved in the design and development of MVAPICH2, an open-source high-

performance MPI over InfiniBand, 10GigE/iWARP and RoCE. This software is currently used by over 1,500 organizations in 60 countries.



Miao Luo is a Ph.D. Student in the Department of Computer Science and Engineering at The Ohio State University. Her research interests include high performance computing, high performance communication over modern interconnects, and hybrid programming models. She has published about 5 papers in conferences related to these research areas. She is a member of the Network-Based Computing Laboratory lead by Dr. D.K. Panda. She is involved in the design and development of MVAPICH2, an open-source highperformance MPI over InfiniBand, 10GigE/iWARP and RoCE. This software is currently used by over 1,500 organizations in 60 countries.

Ashish Kumar Singh is a Ph.D. Student in the Department of Computer Science and Engineering at The Ohio State University. His research interests include network based computing, high performance computing and GPGPU computing. He joined for his Ph.D. Program after working with Samsung Electronics for a year. He is a member of the Network-Based Computing Laboratory lead by Dr. D.K. Panda.

Sayantan Sur is a Research Scientist at the Department of Computer Science at The Ohio State University. His research interests include high speed interconnection networks, high performance computing, fault tolerance and parallel computer architecture. He has published more than 20 papers in major conferences and journals related to these research areas. He is a member of the Network-Based Computing Laboratory lead by Prof. D.K. Panda. He is currently collaborating with National Laboratories, Supercomputer Centers, and leading InfiniBand companies on designing various subsystems of next generation high performance computing platforms. He has contributed significantly to the MVAPICH/MVAPICH2 (High Performance MPI over InfiniBand and 10GigE/iWARP) open-source software packages. The software developed as a part of this effort is currently used by over 1,500 organizations in 60 countries. In the past, he has held the position of Post-doctoral researcher at IBM T. J. Watson Research Center, Hawthorne and Member Technical Staff at Sun Microsystems. Dr. Sur received his Ph.D. degree from The Ohio State University in 2007. More details are available at: http://www.cse.ohiostate.edu/~surs.

Dhabaleswar K. Panda is a Professor of Computer Science at The Ohio State University. His research interests include parallel computer architecture, high performance networking, InfiniBand, exascale computing, programming models, high performance file systems and storage, accelerators, virtualization and cloud computing. He has published over 285 papers (including multiple Best Paper Awards) in major journals and international conferences related to these research areas. Dr. Panda and his research group members have been doing extensive research on modern networking technologies including InfiniBand, 10GigE/iWARP and RDMA over Converged Enhanced Ethernet (RoCE). His research group is currently collaborating with National Laboratories and leading companies on designing various subsystems of next generation high-end systems. The MVAPICH/MVAPICH2 (High Performance MPI over InfiniBand, 10GigE/iWARP and RoCE) open-source software packages, developed by his research group (http://mvapich.cse.ohio-state.edu), are currently being used by more than 1,500 organizations worldwide (in 60 countries). This software has enabled many InfiniBand clusters to get into the latest TOP500 ranking. These software packages are also available with the Open Fabrics stack for network vendors (InfiniBand, iWARP and RoCE), server vendors and Linux distributors. Dr. Panda's research is supported by funding from US National Science Foundation, US Department of Energy, and several industry including Intel, Cisco, SUN, Mellanox, and QLogic. Dr. Panda is an IEEE Fellow and a member of ACM.