Optimized Non-contiguous MPI Datatype Communication for GPU Clusters: Design, Implementation and Evaluation with MVAPICH2

Hao Wang, Sreeram Potluri, Miao Luo, Ashish Kumar Singh, Xiangyong Ouyang,
Sayantan Sur, Dhabaleswar K. Panda
Department of Computer Science and Engineering,
The Ohio State University,
{wangh, potluri, luom, singhas, ouyangx, surs, panda}@cse.ohio-state.edu

***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 remains the biggest hurdle to overall performance and programmer productivity. Real scientific applications utilize multi-dimensional data. Data in higher dimensions may not be contiguous in memory. In order to improve programmer productivity and to enable communication libraries to optimize non-contiguous data communication, the MPI interface provides MPI datatypes. Currently, state of the art MPI libraries do not provide native datatype support for data that resides in GPU memory. The management of non-contiguous GPU data is a source of productivity and performance loss, because GPU application developers have to manually move the data out of and in to GPUs.**

In this paper, we present our design for enabling high-performance communication support between GPUs for non-contiguous datatypes. We describe our innovative approach to improve performance by "offloading" datatype packing and unpacking on to a GPU device, and "pipelining" all data transfer stages between two GPUs. Our design is integrated into the popular MVAPICH2 MPI library for InfiniBand, iWARP and RoCE clusters. We perform a detailed evaluation of our design on a GPU cluster with the latest NVIDIA Fermi GPU adapters. The evaluation reveals that the proposed designs can achieve up to 88% latency improvement for vector datatype at 4 MB size with micro benchmarks. For Stencil2D application from the SHOC benchmark suite, our design can simplify the data communication in its main loop, reducing the lines of code by 36%. Further, our method can improve the performance of Stencil2D by up to 42% for single precision data set, and 39% for double precision data set. To the best of our knowledge, this is the first such design, implementation and evaluation of non-contiguous MPI data communication for GPU clusters.

I. 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). Three of the five top systems in the June, 2011 Top500 list [1] leverage GPU technology to gain excellent performance. The Compute Unified Device Architecture (CUDA) is one of the most popular programming models for NVIDIA GPUs. CUDA provides methods to create 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 fast, (x16 PCIe 2.0 provides 8 GB/s bandwidth in each direction) it is still slow compared to the compute capabilities of GPUs. One of the main factors limiting GPU enabled applications is the latency of data movement between CPU and GPU memories.

Scientific applications often manipulate multi-dimensional data. The most commonly used finite element methods employ either 2-D or 3-D data. Dealing with parts of multi-dimensional structures data is more complex because of their non-contiguous nature. In order to simplify the task of sending and receiving non-contiguous data, the Message Passing Interface (MPI), which is one of the most popular parallel programming models, provides datatype support. Using this, multi-dimensional data can be described as a datatype and can be directly used in MPI send, receive and collective operations. The MPI implementation can internally transform non-contiguous data into contiguous data and transfer it over the network. Advanced MPI libraries may also use specialized non-contiguous data transfer support from network adapters to optimize communication for non-contiguous MPI datatypes. Over time, developers of scientific applications have come to rely on the MPI datatype features for writing real scientific applications.

A. Motivation

As GPUs are increasingly being used for general purpose computation, more and more complex real-world scientific applications that are written in MPI are being modified to tap into GPU performance benefits. Since many of these applications involve multi-dimensional data, programmers have to deal with moving these structures between GPU and main memory. Multi-dimensional data is contiguous only in one dimension, and non-contiguous in other dimensions. An example is, the 2D stencil where the north and south boundaries are contiguous, and the east and west boundaries are non-contiguous in a row major organization. Currently, the following methods can be used to move non-contiguous two-dimensional data from/into GPU memory. They are shown in Figure 1.

978-0-7695-4516-5/11 $26.00 © 2011 IEEE
DOI 10.1109/CLUSTER.2011.42

(a) Use `cudaMemcpy2D` to instruct GPU device to transfer data from GPU to Host memory. The resulting data in host memory is also non-contiguous. This is shown Figure 1(a).
(b) Use `cudaMemcpy2D` directly to transfer non-contiguous data from GPU to Host, where the resulting data is contiguous. This is shown in Figure 1(b).
(c) Use `cudaMemcpy2D` to flatten the two-dimensional data into contiguous region in GPU device memory and then subsequently use another `cudaMemcpy` to bring the data into host memory. This is shown in Figure 1(c).

The optimal choice of these options is not immediately obvious. Which one should the programmer follow? What are the performance advantages of each option over the other? In Section IV-A, we provide a detailed performance analysis of these choices. For a non-contiguous vector of size 4 KB, the cost of option (a) is 200 μs on a Tesla 2050 GPU. The cost for option (b) is 281 μs and the cost for option (c) is only 35 μs. There is a **factor of eight** difference between options (b) and (c).

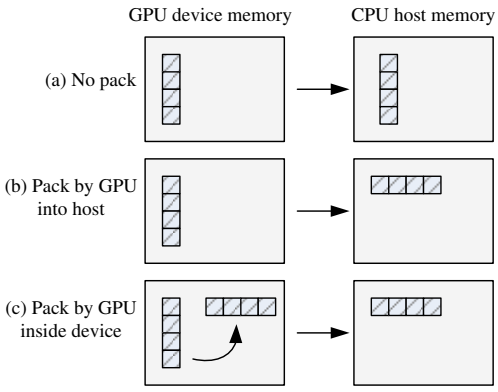


Figure 1. Data packing options for GPU based systems

It is clear that the programming difficulty of dealing with non-contiguous data coupled with the wide performance differences between the design choices will pose a serious challenge to programming these GPUs effectively for scientific parallel applications. This provides a strong motivation for supporting non-contiguous MPI datatypes to *transparently* operate on data in the GPU. Not only do MPI datatypes provide ease of programming, but they also automatically provide the performance of an optimized MPI library to the application without exposing low-level details to the programmer.

B. Contributions

This paper makes several important contributions. They are as follows:

1) Our work enables high-performance GPU to GPU communication for non-contiguous data. MPI datatype support significantly reduces programmer effort in communicating data between GPUs.
2) Our work partially offloads datatype processing from host CPU to GPU. Using this offloading mechanism, non-contiguous buffers can be packed and unpacked more efficiently than existing CPU based methods.
3) Our innovative protocol sub-divides data transfer into multiple stages and applies pipelining to get high performance.
4) We demonstrate improvements in performance and ease of programming with established GPU benchmarks for parallel computing, like SHOC [2].

To the best of our knowledge, this is the first research paper that describes a high-performance approach towards non-contiguous datatype communication between GPUs. The design proposed in this paper has been integrated into MVAPICH2. MVAPICH2 natively supports direct GPU to GPU communication using NVIDIA CUDA 4.0. In a previous paper [3], the design of GPU support in MVAPICH2 was discussed. However, the previous design only considered contiguous data communication. In this paper, we design and implement the high-performance support for non-contiguous MPI datatype communication. Evaluation reveals that the proposed extensions can achieve up to 88% improvement in latency for vector data of 4 MB. For the Stencil2D application benchmark, our design can simplify the data communication in its main loop, reducing the lines of code by 36%. Moreover, with our method, performance of Stencil2D can be improved by up to 42% and 39% for single precision and double precision data sets, respectively.

The rest of the paper is organized as follows. In Section II we provide the necessary background about the topics discussed in this paper. In Section III we distinguish our work from existing research in this area. In Section IV we describe our design in detail. In Section V we present detailed evaluation of the proposed designs. The paper concludes in Section VI.

II. Background

In this section, we provide the appropriate background for this paper. We will discuss the GPU architecture and the support for GPU-to-GPU communication in MVAPICH2.

A. GPU Architecture and Programming Model

Graphics Processing Units (GPUs) are increasingly being used for general purpose computing. They are often called General Purpose GPUs, or GPGPUs. GPGPUs can be viewed as a data-parallel multi-core system. In this paper, we focus on the NVIDIA GPU architecture. In this architecture, the GPU is connected as a peripheral device on the I/O bus (PCI express). The latest architectural revision of NVIDIA GPUs for High-performance computing is called Fermi. The Fermi [4] GPU architecture consists of 16 streaming multiprocessors (SMs), each of which consists of 32 cores.

Each SM includes local memory and uses it as L1 cache; and all SMs share L2 cache and the directly connected DRAM. NVIDIA also provides a software framework for programming its GPUs. It is called the Compute Unified Device Architecture (CUDA) [5]. Code that runs on the GPU is often called the GPU kernel. This code is compiled by NVIDIA compilers and executes in a multi-threaded fashion with SIMD data model.

In a cluster environment, GPUs are connected to the local nodes via PCI express and the nodes themselves are connected to each other via a high-performance network, such as InfiniBand. GPUs can read/write memory that is attached on the local node. They cannot access remote memory directly, and need intervention from the CPU to do so. The MPI library, for example, sits on the CPU and is responsible for communication.

Before GPU kernel executes, data must be moved from the main memory into the device memory for GPU kernel execution. During the execution, data that needs to be exchanged between nodes is first moved into the main memory and then transferred using any standard communication interface. The data received in the exchange has to be moved into the device memory for computation. Eventually, the results have to be moved back into the host memory. The transfers between device and main memory can be done with CUDA interfaces, such as `cudaMemcpy()` for the contiguous data and `cudaMemcpy2D()` for strided data.

B. MVAPICH2 on GPU Clusters

The Message Passing Interface (MPI) is the de-facto standard for parallel applications. MVAPICH and MVAPICH2 [6] are popular open-source implementations of MPI on InfiniBand, 10Gigabit Ethernet/iWARP and the emerging RDMA over Converged Enhanced Ethernet (RoCE). In this paper, we only focus on InfiniBand interconnect, but our mechanism is valid on any advanced interconnects providing Remote Direct Memory Access (RDMA) capability, with which communication can be performed without any host processor involvement.

MVAPICH2 unifies data movement in CUDA and InfiniBand while providing the standard MPI 2.2 semantics. This design was first proposed by Wang et. al in [3]. MVAPICH2 leverages the Unified Virtual Addressing (UVA) feature that is provided by the new CUDA 4.0 release. In addition to providing unified addressing, MVAPICH2 can further optimize the performance of GPU to GPU communication. This is achieved by pipelining transfers from GPU to host memory, host memory to remote host memory via InfiniBand and finally from remote host to destination GPU memory. The three levels of pipelining result in a significant performance boost. Currently, MVAPICH2 only supports transfer of contiguous datatypes. In this paper, we present the design and implementation of MVAPICH2 for optimizing transfer of non-contiguous MPI datatypes between GPUs.

III. Relation to Previous Work

Scientists and application developers using heterogeneous computing systems equipped with GPUs need to manage two kinds of data transfers: between GPU and Host, and between different Hosts. This brings about several programming challenges. In [3], Wang et. al. show that using MVAPICH2, the data movement steps can be simplified when the MPI library is CUDA "aware". Using MVAPICH2, an application programmer can directly issue an `MPI_Send` (or `MPI_Isend`) from memory that resides in GPU. This alleviates the need for the programmer to first stage memory in host and then issue send or receive operations. Further, it can pipeline all the various stages of transfer (as discussed in Section II-B) and achieve better performance than user optimized approaches. Currently, MVAPICH2 only supports contiguous datatype communication between GPUs. In this paper, we focus on communication with non-contiguous MPI datatypes.

Gelado et.al. propose the Asymmetric Distributed Shared Memory model (ADSM) to unify host memory and device memory on the heterogeneous system with GPGPUs [7]. With newly introduced interfaces in ADSM, the device memory becomes transparent to programmers. ADSM run time system handles the memory management and the communication between device memory and host memory through the memory coherence protocol. In this paper, we focus on the MPI programming model that is more scalable than DSM model in a HEC cluster and our approaches work completely in user-space. Using our proposed methods, users can write straight MPI code and the MPI library is responsible for staging any data.

Other programming models for a HEC cluster with GPGPUs, like Zippy [8], DCGN [9], and cudaMPI [10] depend on MPI as the underlying communication method between nodes. These works can directly benefit from our work as MVAPICH2 pipelines messages internally.

Pipelining is a widely used technique by application developers to reduce the communication latency between host memory and GPU memory. Many researchers have attempted overlapping memory copy from GPU to host memory with kernel execution on GPU [11], [12] or MPI data transfer between nodes [13], [14]. These approaches improve performance at the cost of productivity, since careful evaluation and tuning is necessary for each target platform. MVAPICH2 provides a simplified method for application developers to perform GPU to GPU communication with highest performance. Instead of tuning each application, only the MVAPICH2 library needs to be tuned for the platform and all applications using MPI can directly benefit from it.

There are several researchers optimizing MPI datatype processing [15], [16], [17], and researchers optimizing data movement on the advanced network like InfiniBand and 10Gigabit Ethernet [18], [19]. However, none of them have

considered GPU platforms. Our work shows that when certain parts of the datatype processing are offloaded on the GPU device, significant performance improvements can be gained. To the best of our knowledge, our work is the first one of its kind to support non-contiguous data processing and communication from GPU device memory.

IV. Design

Our work in [3] has enabled MVAPICH2 to support GPU to GPU communication through the standard MPI interface. In this work we extend it by providing support for non-contiguous MPI datatype transfers. In the rest of this paper, we refer to the existing work as MVAPICH2 and the work proposed in this paper as MV2-GPU-NC.

A. Datatype Processing Offload to GPU

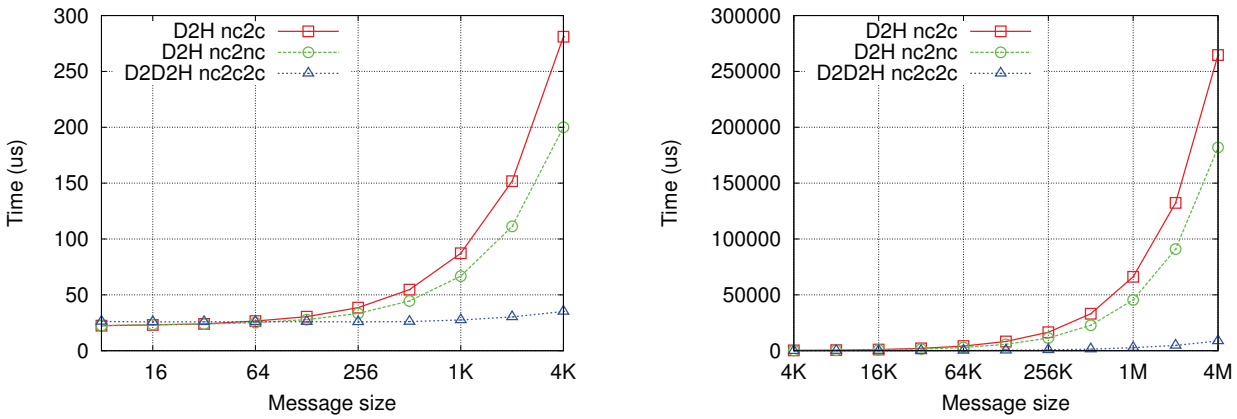
The existing design for GPU to GPU communication in MVAPICH2 occurs as follows: 1. Application programmer issues standard MPI communication calls with buffers in GPU device memory as parameters; 2. MVAPICH2 library detects if the buffers provided in the MPI call are in the device memory or in the host memory; 3. As the buffer is in device memory, the library gets a staging buffer from a pre-allocated pool and initiates data movement from device to host, asynchronously (large messages are chunked); 4. As the data movement to the host completes, transfers over the network are initiated; 5. The receiver side will move data into the device as the transfers complete. For large messages, steps of moving data from device to host, transferring data over the network and moving data from host to device are overlapped through a pipeline.

For non-contiguous transfers, the data has to be packed before it can be sent over the network. This avoids multiple transfers over the network that can be prohibitively expensive. Currently, application developers move non-contiguous data from device to the host using cudaMemcpy2D, create an MPI datatype and then issue an MPI communication call over the datatype. The MPI library internally packs and sends the data over the network. On the receiver side, the MPI library unpacks the data and give it to the user who then moves the data from host to device using another cudaMemcpy2D operation. As mentioned in Section I-A the movement of data from device to host forms a large part of the total communication time. While designing GPU-to-GPU transfer of non-contiguous data we have explored various schemes of packing non-contiguous data from device memory into host memory. We present a comparison of these schemes in Figure 2. "D2H nc2nc" mimics the generally used scheme of moving non-contiguous data from device memory to non-contiguous buffer in the host memory with cudaMemcpy2D. "D2H nc2c" moves data from a non-contiguous buffer in device memory to a contiguous buffer in the host memory. This does the job of packing the data within the cudaMemcpy2D and is achieved by using appropriate stride, width and height parameters. "D2D2H nc2c2c" uses an intermediate device buffer to pack data in the device memory before moving it to the host. This involves a cudaMemcpy2D call from device to device and then a cudaMemcpy call from device to host. The last two schemes can offload datatype packing to the GPU by using asynchronous copies (cudaMemcpy2DAsync). As shown in the Figure 2, "D2D2H nc2c2c" provides much better latencies compared to the other schemes for all messages sizes greater than 64 bytes. For 4M Bytes, latency of "D2D2H nc2c2c" is only 4.8% of the latency from "D2H nc2nc". We use the "D2D2H nc2c2c" with asynchronous copies in our implementation.

B. Communication Pipeline Design

Data movement between GPU device memory and host memory happens through a DMA which allows for asynchronous progress. Similarly, data transfer over the network can progress asynchronously using RDMA. In an ideal scenario, it is possible to implement data transfer from one GPU to another remote GPU in a completely asynchronous fashion. Our previous research [3] has shown that for messages beyond a given threshold, movement of data between device memory and host memory can be completely overlapped with the RDMA data transfer over the network. When the data size is smaller than the threshold, data movement from/into the GPU dominates the overall latency. As a result, we use the pipeline data transfer for messages larger than the threshold. However, for non-contiguous data, latency of packing data in the GPU is always larger than the RDMA data transfer latency or time for contiguous data movement between device memory and main memory. So, data packing and unpacking latency will determine the pipeline performance.

There are five steps involved in moving data from one GPU to another GPU on a remote node: "D2D nc2c" to pack non-contiguous data into contiguous buffer inside GPU device memory; "D2H c2c" to move the contiguous data from GPU device memory to local host memory; "RDMA" to do the RDMA data transfer between nodes; "H2D c2c" to move the contiguous data from host memory to GPU device memory on the receiver side; "D2D c2nc" to unpack contiguous data to the non-contiguous destination buffer inside GPU device memory. The latency to transfer N bytes of data in this process is: $T\_d2d\_nc2c(N) + T\_d2h\_c2c(N) + T\_rdma(N) + T\_h2d(N)\_c2c + T\_d2d\_c2nc(N)$. We observe similar latency for $T\_d2d\_nc2c(N)$ and $T\_d2d\_c2nc(N)$. So, the following expression can be used to model the latency of pipelined non-contiguous data transfer, where the data is divided into n block: $(n+2)*T\_d2d\_nc2c(N/n)$. Based on our experimentation, we found 64KB to be the optimal block size in our experimental environment. This unit is presented as a configurable parameter to the MPI library and can



(a) Small Message

(b) Large Message

Figure 2. Non-contiguous Data Pack Performance

be tuned once by the system administrator during the time of installation by using OSU micro-benchmarks. Once the optimal block size for the cluster is detected, the number can be placed in a configuration file. This configuration approach is transparent to the end user.

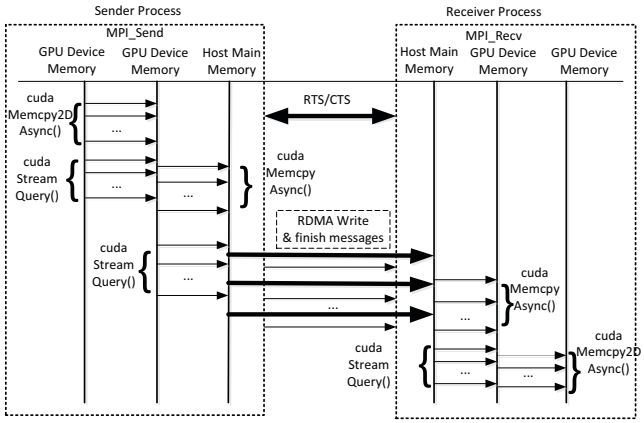


Figure 3. Non-contiguous Data Pipeline Design

Figure 3 illustrates the architecture of our pipeline design for the vector data type. When the memory from parameters in the MPI call is detected in the device memory, the sender grabs a temporary buffer called *tbuf* in the device memory to pack the non-contiguous source data; and triggers multiple asynchronous memory copy with `cudamemcpy2DAsync()` function, each of which does a chunk size non-contiguous data pack. Then the sender returns back to MPI progress engine and sends a Request To Send (RTS) message to the receiver to get the RDMA Put address. During the handshaking process, if MPI progress engine finds any pack operation finished, the sender will get a chunk sized buffer called *vbuf* from host memory buffer pool, and starts the asynchronous memory copy from device to host (from tbuf to vbuf) with `cudamemcpyAsync()` function. When the receiver receives the RTS, it will reply back a Clear To Send (CTS) message based on MPI message matching semantics. In the CTS message, the receiver will encode the remote buffer address and size of the message to be received. The address encoded in the CTS message is that of a list of vbufs. Once the CTS message is received and one of the asynchronous memory copies finishes, the sender will call InfiniBand verbs interface to perform the RDMA write. After each RDMA write finishes, the sender will send 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 the corresponding place of tbuf. Finally, when any CUDA memory copy from host to device is finished, the corresponding unpack operation will be started to do the data unpack in the device memory.

C. Programming model

With the proposed methods in MV2-GPU-NC, application programmers can easily send/receive non-contiguous data in GPU device memory and get much better performance. Figure 4 provides a comparison of code complexity for GPU to GPU vector transfers using existing methods and using the proposed deisgn of MV2-GPU-NC. Figure 4(a) shows a naive implementation using blocking CUDA 2D memory copies and blocking MPI communication calls. It

```
MPI_Type_vector();
MPI_Type_commit();
...
if (haveEastNeighbor) {
    // copy noncontiguous data from device to host
    cudaMemcpy2D(...);
    // send data with vector type to east neighbor
    MPI_Send(...);
    // receive data with vector type from east neighbor
    MPI_Recv(...);
    // copy noncontiguous data from host to device
    cudaMemcpy2D(...);
}
...
```

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

```
MPI_Type_vector();
MPI_Type_commit();
...
if (haveEastNeighbor) {
  for (i = 0; i < pipeline_length; i++) {
    // pack each block from non contiguous to contiguous in GPU
    cudaMemcpy2DAsync(…);
  }
  while (active_pack_stream || active_d2h_stream ) {
    if (active_pack_stream > 0) {
      if (cudaStreamQuery() == cudaSuccess) {
        // copy each block from device memory to host memory
        cudaMemcpyAsync(…);
      }
    }
    if (active_d2h_stream > 0) {
      if (cudaStreamQuery() == cudaSuccess) {
        // send each block to east neighbor from host memory
        MPI_Isend(...);
      }
    }
  }
  MPI_Waitall(...);
  for (j=0; j < pipeline_length; j++) {
    // receive each block from east neighbor to host memory
    MPI_Irecv(...);
  }
  while (active_recv > 0 || active_h2d_stream > 0) {
    if (active_recv > 0) {
      MPI_Test (...);
      // copy each block from host memory to device memory
      cudaMemcpyAsync (...);
    }
    if (active_h2d_stream > 0) {
      if (cudaStreamQuery()== cudaSuccess) {
        // unpack each block from contiguous to non contiguous in GPU
        cudaMemcpy2DAsync(...);
      }
    }
  }
}
…
```

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

```
MPI_Type_vector();
MPI_Type_commit();
...
if (haveEastNeighbor) {
    // send data with vector type from device memory to east neighbor
    MPI_Send(...);
    // receive data with vector type to device memory from east neighbor
    MPI_Recv(...);
}
…
```

(c) MV2-GPU-NC (highest performance and productivity)

Figure 4. Pseudo-code comparing existing approaches and MV2-GPU-NC non-contiguous data communication between GPUs

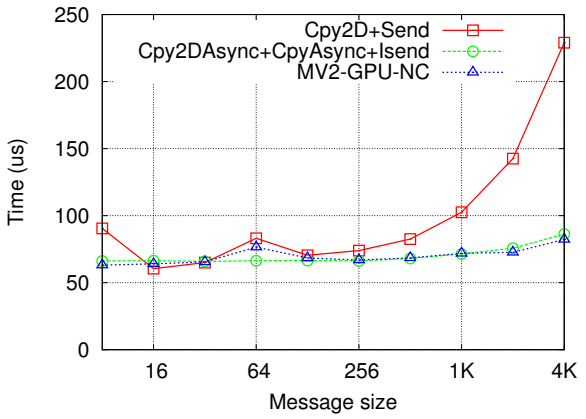
is straightforward for application developers but does not provide good performance. Figure 4(b) shows how an application developer can improve the performance by carefully interleaving non-blocking memory copy and MPI communication calls. This method follows the aforementioned idea in Section IV-A to pack and unpack non-contiguous data with GPU. Although it can provide good performance, this method is complicated for programmers and the platform specific performance tuning is a big challenge. Figure 4(c) shows the ease with which application developers can exploit overlap using MV2-GPU-NC: the standard MPI interfaces are used; and the underlying library takes care of the optimizations.

V. Experiment and Evaluation

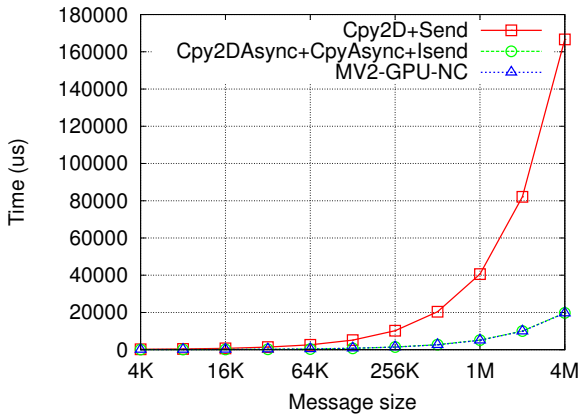
We used a cluster with eight nodes in our experimental evaluation. Each node is equipped with dual Intel Xeon Quad-core Westmere CPUs operating at 2.53 GHz, 12 GB host memory, and Nvidia Tesla C2050 GPUs with 3 GB DRAM. The InfiniBand HCAs used on this cluster are Mellanox QDR MT26428. Each node has Red Hat Linux 5.4, OFED 1.5.1, MVAPICH2-1.6RC2, and CUDA Toolkit 4.0. The MPI level evaluation is based on OSU Micro Benchmarks [20]. We modified Stencil2D application in SHOC 1.0.1 with MV2-GPU-NC to send and receive both contiguous and non-contiguous data in GPU device memory. We run one process per node and use one GPU per process for all experiments.

A. Performance Evaluation for Vector Data

In this section, we compare the performance of vector type non-contiguous data transfer using benchmarks based on the three designs shown in Figure 4. "Cpy2D+Send" uses the blocking design shown in Figure 4(a). It uses `cudaMemcpy2D()` to move non-contiguous data between device and host, and `MPI_Send()` and `MPI_Recv()` to transfer data between the nodes. "Cpy2DAsync+CpyAsync+Isend" follows the design shown in Figure 4(b) that uses `cudaMemcpy2DAsync()` to pack and unpack data inside the device memory and uses `cudaMemcpyAsync()` to move data between device memory and host memory. Through asynchronous 2D copies, this method offloads the packing and unpacking operations to the GPU, and it uses non-blocking MPI calls. This method provides good performance for vector data transfer from GPU to GPU without MV2-GPU-NC through a carefully pipelined implementation. "MV2-GPU-NC" represents the method proposed in this paper: non-contiguous datatype processing offloaded to GPU and asynchronous pipeline used inside MVAPICH2 library. This design is shown in Figure 4(c). These benchmarks are run on a 1x2 process grid for varying non-contiguous message sizes and a constant chunk size of 4 bytes (float).



(a) Small Message



(b) Large Message

Figure 5. Vector Communication Latency

The results for small message and large message transfers are shown in Figure 5(a) and Figure 5(b), respectively. "MV2-GPU-NC" can achieve up to 88% latency improvement over "Cpy2D+Send" for 4M byte message. There are two key reasons for the performance improvement observed here: 1. The use of GPU for data packing and unpacking is better for most of the data sizes as illustrated in Figure 2; 2. The use of the complete asynchronous pipeline is better by overlapping data packing and unpacking, data movement between device and host, and MPI transfer between nodes. "Cpy2DAsync+CpyAsync+Isend" and "MV2-GPU-NC" show similar performance as they follow a similar implementation at the benchmark and library levels, respectively. "MV2-GPU-NC" hides the complexity of pipelining and the overhead of platform specific tuning from application developers.

B. Performance evaluation for Stencil2D

Scalable HeterOgeneous Computing benchmark (SHOC) [2] is a suite of benchmarks to test the performance and stability of heterogeneous computing systems built with GPUs and multi-core processors. The application benchmark of interest in this work is the Stencil2D which is designed to measure the performance of a two-dimensional nine point stencil calculation. It has a halo communication exchange.

1) Communication Breakdown: Figure 6 shows a breakdown of the communication time in Stencil2D. The experiment was carried out on a 2x4 process grid with a 8Kx8K single precision data set per process. The time shown in the figure is at rank 1, which has neighbors in three dimensions: south, west and east. South_mpi, west_mpi and east_mpi represent the time spent in mpi. South_cuda, west_cuda and east_cuda represent the time spent in moving data between device and main memory. We see that non-contiguous data movement between the device memory and the main memory dominates the communication time. In the following sections, we present results with Stencil2D modified to use MV2-GPU-NC.

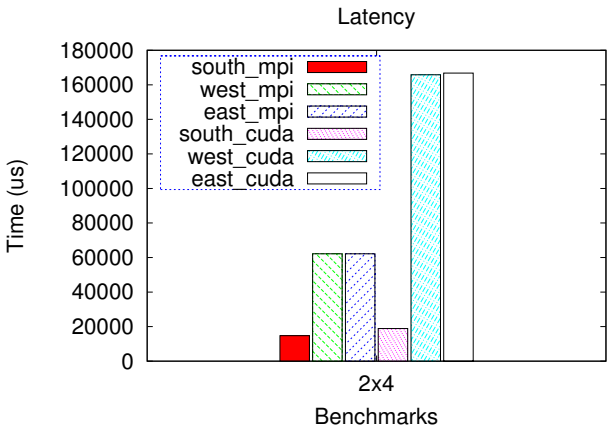


Figure 6. Dimension-wise Communication Breakdown in Stencil2D

2) Code Complexity Comparison: The existing communication of Stencil2D is similar to the one shown in Figure 4(a) except that it uses non-blocking `MPI_Irecv`. We have used MV2-GPU-NC to modify the communication exchange in Stencil2D. We call the original version as "Stencil2D-Def" and the modified version as "Stencil2D-

	Stencil2D-Def	Stencil2D-MV2-GPU-NC
Function calls	MPI_Irecv: 4 MPI_Send 4 MPI_Waitall: 2 cudaMemcpy: 4 cudaMemcpy2D: 4	MPI_Irecv: 4 MPI_Send: 4 MPI_Waitall: 2 cudaMemcpy: 0 cudaMemcpy2D: 0
Lines of Code	245	158

Table I
COMPARING COMPLEXITY OF EXISTING STENCIL2D CODE WITH MODIFIED CODE USING MV2-GPU-NC

Process Grid (Matrix Size/Process)	Stencil2D-Def	Stencil2D-MV2-GPU-NC	Improvement
1x8 (64k x 1k)	0.547788	0.314085	42%
8x1 (1k x 64k)	0.33474	0.272082	19%
2x4 (8k x 8k)	0.36016	0.261888	27%
4x2 (8k x 8k)	0.33183	0.258249	22%

Table II
COMPARING MEDIAN EXECUTION TIMES OF STENCIL2D - SINGLE PRECISION (SEC)

MV2-GPU-NC". Table I compares these two implementations for code complexity. "Function Calls" is the number of function calls in the main loop of Stencil2D and "Lines of Code" is the number of lines of code in the main loop. In "Stencil2D-Def" the programmer has to call `cudaMemcpy()` and `cudaMemcpy2d()` four times each to move data out of and into the device memory for north/source and east/west directions respectively, while in "Stencil2D-MV2-GPU-NC" the programmer can directly provide the buffers in device memory as parameters in MPI communication calls. As a result, we observe that using MV2-GPU-NC, the code used in this loop is decreased by 36% (from 245 lines to 158 lines).

Process Grid (Matrix Size/Process)	Stencil2D-Def	Stencil2D-MV2-GPU-NC	Improvement
1x8 (64k x 1k)	0.780297	0.474613	39%
8x1 (1k x 64k)	0.563038	0.438698	22%
2x4 (8k x 8k)	0.57544	0.424826	26%
4x2 (8k x 8k)	0.546968	0.431908	21%

Table III
COMPARING MEDIAN EXECUTION TIMES OF STENCIL2D - DOUBLE PRECISION (SEC)

3) Performance Comparison: In this section, we compare the performance of the two versions of Stencil2D: "Stencil2D-Def" and "Stencil2D-MV2-GPU-NC". Table II presents the median execution times of each Stencil2D iteration on five different process grids and single precision data sets. Table III presents results on the same set of configurations but with double precision data. In a "1x8" process grid, the data exchange happens only in the east-west dimension and involves non-contiguous buffers. In a "8x1" process grid, data exchange happens only in the north-south dimension and hence involves contiguous buffers. Due to device memory requirements of SHOC benchmark suite, we had to limit our matrix size per process to 256M Bytes to run Stencil2D. To evaluate the proposed pipelining schemes, which get activated beyond 64K Bytes, we used a larger matrix size in the communicating dimension in these two cases. We used a 64K x 1K matrix per process for the "1x8" process grid and a 1K x 64K matrix per process for the "8x1" process grid. The other process grids "2x4" and "4x2" have communication in both east-west and north-south dimensions. We used an 8K x 8K matrix per process for these two cases. The time reported here includes time for GPU kernel execution and data exchange. For the single precision data set, "Stencil2D-MV2-GPU-NC" can get 42%, 19%, 27% and 22% improvement in latency on the "1x8", "8x1", "2x4" and "4x2" process grids, respectively. We clearly see the highest improvement on the "1x8" grid because of the larger non-contiguous data movement. This case gets benefits from GPU offloaded datatype processing and the use of pipelining. We see the least improvement on the "8x1" process grid where the transfers are contiguous. The benefits for this case are from pipelining alone. For "2x4" and "4x2" cases, the percentage of non-contiguous data exchange are 60% and 40%, respectively and hence we see greater improvement on the "2x4" grid than on the "4x2" grid though they use the same data set per process. For the double precision data set, "Stencil2D-MV2-GPU-NC" can get 39%, 22%, 26% and 21% improvement, respectively.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we design, implement, and evaluate a high performance GPU to GPU communication method for non-contiguous data in InfiniBand clusters. Our design not only eases programmers' effort on non-contiguous data communication from GPU to GPU, but also provides the high performance through offloading datatype processing from CPU to GPU and pipelining all stages in data transfer. To the best of our knowledge, it is the first research paper to describe a high performance approach for non-contiguous datatype communication between GPUs.

The investigation reveals that our approach can achieve up to 88% latency improvement for vector datatype at 4 MB data size with micro benchmarks. For Stencil2D benchmark, our approach can simplify the data communication in its

main loop, reducing the lines of code by 36%; at the same time, its performance can be improved by up to 42% and 39% for single precision and double precision data set, respectively.

We intend to continue working in this direction. The approach developed in this paper will be integrated into future public MVAPICH2 releases. We also plan to evaluate the impact of our approach with more applications.

VII. Acknowledgements

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-0621484, #CCF-0702675, #CCF-0833169, #CCF-0916302 and #OCI-0926691; grant from Wright Center for Innovation #WCI04-010-OSU-0; grants from Intel, Mellanox, Cisco, QLogic, and Sun Microsystems; Equipment donations from Intel, Mellanox, AMD, Obsidian, Advanced Clustering, Appro, QLogic, and Sun Microsystems.

References

[1] TOP500 Supercomputing Sites. http://www.top500.org/.

[2] 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 Processing on Graphics Processing Units (GPGPU'10)*, 2010.

[3] H. Wang, S. Potluri, M. Luo, A. K. Singh, S. Sur, and D. K. Panda, "MVAPICH2-GPU: Optimized GPU to GPU Communication for InfiniBand Clusters," in *Computer Science - R&D 26(3-4)*, pp. 257–266, 2011.

[4] NVIDIA, "NVIDIA's Fermi: The First Complete GPU Computing Architecture." http://www.nvidia.com/content/PDF/fermi_white_papers/P.Glaskowsky_NVIDIA%27s_Fermi-The_First_Complete_GPU_Architecture.pdf.

[5] NVIDIA, "NVIDIA CUDA Compute Unified Device Architecture." http://developer.download.nvidia.com/compute/cuda/2_0/docs/CudaReferenceManual_2.0.pdf.

[6] MVAPICH2: High Performance MPI over InfiniBand, 10GigE/iWARP and RoCE. http://mvapich.cse.ohio-state.edu/.

[7] I. Gelado, J. Cabezas, N. Navarro, J. E. Stone, S. J. Patel, and W. mei W. Hwu, "An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems," in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'10)*, pp. 347–358, 2010.

[8] Z. Fan, F. Qiu, and A. E. Kaufman, "Zippy: A Framework for Computation and Visualization on a GPU Cluster," in *Computer Graphics Forum (Eurographics)*, pp. 27(2):341–350, 2008.

[9] J. A. Stuart and J. D. Owens, "Message Passing on Data-Parallel Architectures," in *Proceedings of the 23th IEEE International Parallel and Distributed Processing Symposium (IPDPS'09)*, 2009.

[10] O. S. Lawlor, "Message Passing for GPGPU Clusters: cudaMPI," in *Proceedings of the 2009 IEEE International Conference on Cluster Computing (Cluster'09)*, 2009.

[11] W. Ma, S. Krishnamoorthy, O. Villa, and K. Kowalski, "Acceleration of Streamed Tensor Contraction Expressions on GPGPU-based Clusters," in *Proceedings of the 2010 IEEE International Conference on Cluster Computing (Cluster'10)*, 2010.

[12] E. H. Phillips and M. Fatica, "Implementing the Himeno Benchmark with CUDA on GPU Clusters," in *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS'10)*, 2010.

[13] D. A. Jacobsen, J. C. Thibault, and I. Senocak, "An MPI-CUDA Implementation for Massively Parallel Incompressible Flow Computations on Multi-GPU Clusters," in *Proceedings of the 48th AIAA Aerospace Sciences Meeting*, 2010.

[14] J. W. Choi, A. Singh, and R. W. Vuduc, "Model-driven Autotuning of Sparse Matrix-Vector Multiply on GPUs," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parrallel Programming (PPoPP'10)*, pp. 115–126, 2010.

[15] R. Ross, N. Miller, and W. Gropp, "Implementing Fast and Reusable Datatype Processing," in *Proceedings of Recent Advances in Parallel Virtual Machine and Message Passing Interface, 10th European PVM/MPI Users' Group Meeting (EuroPVM/MPI2003)*, 2003.

[16] J. Wu, P. Wyckoff, and D. K. Panda, "High Performance Implementation of MPI Derived Datatype Communication over InfiniBand," in *Proceedings of 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, 2004.

[17] P. Balaji, D. Buntinas, S. Balay, B. F. Smith, R. Thakur, and W. Gropp, "Nonuniformly Communicating Noncontiguous Data: A Case Study with PETSc and MPI," in *Proceedings of the 21th International Parallel and Distributed Processing Symposium (IPDPS'07)*, 2007.

[18] X. Ouyang, R. Rajachandrasekar, X. Besseron, and D. K. Panda, "High Performance Pipelined Process Migration with RDMA," in *Proceedings of the 11th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid'11)*, 2011.

[19] G. Liao, X. Zhu, and L. N. Bhuyan, "A New Server I/O Architecture for High Speed Networks," in *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA-17)*, 2011.

[20] OSU Micro Benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/.