GPU-Aware Non-contiguous Data Movement In Open MPI

Wei Wu
University of Tennessee
Knoxville, USA
wwu12@vols.utk.edu

George Bosilca
University of Tennessee
Knoxville, USA
bosilca@icl.utk.edu

Rolf vandeVaart
NVIDIA
Santa Clara, USA
rvandevaart@nvidia.com

Sylvain Jeaugey
NVIDIA
Santa Clara, USA
sjeaugey@nvidia.com

Jack Dongarra
University of Tennessee
Knoxville, USA
Oak Ridge National Laboratory
Oak Ridge, USA
University of Manchester
Manchester, UK
dongarra@eecs.utk.edu

ABSTRACT

Due to better parallel density and power efficiency, GPUs have become more popular for use in scientific applications. Many of these applications are based on the ubiquitous Message Passing Interface (MPI) programming paradigm, and take advantage of non-contiguous memory layouts to exchange data between processes. However, support for efficient non-contiguous data movements for GPU-resident data is still in its infancy, imposing a negative impact on the overall application performance.

To address this shortcoming, we present a solution where we take advantage of the inherent parallelism in the datatype packing and unpacking operations. We developed a close integration between Open MPI's stack-based datatype engine, NVIDIA's Unified Memory Architecture and GPUDirect capabilities. In this design the datatype packing and unpacking operations are offloaded onto the GPU and handled by specialized GPU kernels, while the CPU remains the driver for data movements between nodes. By incorporating our design into the Open MPI library we have shown significantly better performance for non-contiguous GPU-resident data transfers on both shared and distributed memory machines.

CCS Concepts

•**Theory of computation** → *Distributed computing models;* •**Computer systems organization** → *Heterogeneous (hybrid) systems;* •**Computing methodologies** → Concurrent algorithms;

HPDC'16, May 31-June 04, 2016, Kyoto, Japan

ACM ISBN 978-1-4503-4314-5/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2907294.2907317

Keywords

MPI; GPU; datatype; non-contiguous data; hybrid architecture

1. INTRODUCTION

Throughput-oriented architectures, such as GPUs, are increasingly used for scientific applications because of their high efficiency in solving computationally intensive tasks at an unbeatable rate of power consumption. Evidence of this can be seen in the most recent Top500 list, where GPUs have become the most popular accelerators. GPUs are connected as peripheral devices via PCI-Express, and, for a long time, had a separate memory space than the host. Explicit memory copy directives were necessary to move data between host and GPU, before being available to CPU-based computations or communications. More recently, this memory separation has been fused with the introduction of the Unified Memory Architecture (UMA), allowing the host memory to be directly accessed from GPUs, and inversely, GPU memory to be directly accessed from CPUs – providing applications with transparent access to the entire memory, independent of the physical location of the memory.

MPI is a popular and efficient parallel programming model for distributed memory systems widely used in scientific applications. As many scientific applications operate on multi-dimensional data, manipulating parts of these data becomes complicated because the underlying memory layout is not contiguous.The MPI standard proposes a rich set of interfaces to define regular and irregular memory patterns, the so called derived datatypes (DDT). For example, the widely used linear algebra library ScaLAPACK [2] usually deals with sub-matrices and matrices with irregular shapes such as upper or lower triangular matrices. The DDTs provide a general and flexible solution to describe any collections of contiguous and non-contiguous data with a compact format. Once constructed and committed, an MPI datatype can be used as an argument for any point-to-point, collective, I/O, and one-sided functions. Internally, the MPI datatype engine will automatically pack and unpack data based on the type of operation to be realized, in an efficient way while hiding the low-level details from users. Thus, the scien-

tific application developers do not have to manually pack and unpack data in order to optimize non-contiguous data transfers, but instead they can safely rely on the MPI runtime to make such operations trivial and portable. Several studies [12, 13] have shown that, at least when handling CPU-based data, recent MPI implementations have exhibited significant performance improvement for the handling of non-contiguous datatypes. As a result, applications taking advantage of the MPI datatypes express a drastic benefit in terms of performance, code readability and maintenance compared with codes that prefer a more handmade, application specific approach.

As GPUs have high computational capabilities, an increasing number of scientific applications migrate their computationally intensive parts to GPUs. Since the MPI standard [5] does not define interactions with GPU-based data, it is expected that application developers have to explicitly initiate data movements between host and device memory prior to using MPI to move data across node boundaries. Techniques such as GPUDirect [10] have been developed to enable direct GPU data movement between processes, i.e., without going through the host memory. Unfortunately, these optimizations were designed with a focus on contiguous data, leaving the most difficult operations, the packing and unpacking of non-contiguous memory patterns, in the charge of developers. To fully utilize the PCI-Express and the network, non-contiguous data must be packed into a contiguous buffer prior to wire transfer. There are effective packing/unpacking implementations for datatypes in host memory [12]. However, exposing the same level of support for a non-contiguous MPI datatype based on GPU memory remains an open challenge.

To address the lack of non-contiguous datatype support for GPUs, we present the design of a datatype engine for non-contiguous GPU-resident data, which is able to take advantage of the embarrassingly parallel nature of the pack and unpack operations and efficiently map them onto GPU threads. The GPU datatype engine is incorporated into the Open MPI [6] library, and takes advantage of the latest NVIDIA hardware capabilities, such as GPUDirect, not only to minimize the overheads but also to decrease the overall energy consumption. For contexts where GPUDirect is not available, we provide a copy-in/copy-out protocol using host memory as an intermediary buffer. All these approaches are using a light-weight pipeline protocol to allow pack and unpack operations to work simultaneously to reduce the overall communication time of non-contiguous data between MPI processes. Although this work is done using CUDA in the context of MPI, the ideas are generic and can be easily ported not only to different programming paradigms (OpenSHMEM and OpenCL), but also to other types of accelerators with computational capabilities.

The contributions of this paper are: *a*) a datatype engine designed for GPU-resident non-contiguous data, which adapts the parallelism of the pack/unpack operations to the parallelism available on GPUs; *b*) support for different communication protocols – for RDMA and copy in/out – to maximize the benefit from the capabilities available at the hardware level (GPUDirect); *c*) a light-weight pipeline mechanism to ensure all participants, the sender and the receiver, can be used simultaneously to prepare the data for transfer (pack and unpack); and *d*) to demonstrate the performance boost achieved by the techniques presented in this paper while transferring widely used non-contiguous data memory layouts.

The rest of this paper is organized as follows. Section 2 introduces the related work. Section 3 outlines the design of the GPU datatype engine. Section 4 presents how the GPU datatype engine is integrated into Open MPI. Section 5 evaluates the performance in hybrid systems with four types of benchmarks. Section 6 concludes the paper and describes potential future work.

2. RELATED WORKS

2.1 GPU-Aware MPI

Heterogeneous systems feature several CPU cores and one or more GPUs per node. Writing efficient applications for such heterogeneous systems is a challenging task as application developers need to explicitly manage two types of data movements: intra-process communications (device to host) and inter-process communications. Recent versions of well-known MPI libraries such as MVAPICH2 [17] and Open MPI already provide some levels of GPU support. With these GPU-Aware MPI libraries, application developers can use MPI constructs to transparently move data, even if the data resides in GPU memory. Similar efforts have been made to integrate GPU-awarness into other programming models. Aij et. al. propose the MPI-ACC [1], which seamlessly integrates OpenACC with the MPI library, enabling OpenACC applications to perform end-to-end data movement. Lawlor presents the cudaMPI [8] library for communication between GPUs, which provides specialized data movement calls that translate to *cudaMemcpy* followed by the corresponding MPI call. Even though the paper discusses non-contiguous data support, the current implementation only includes support for vector types. For the PGAS programming model, Potluri et. al [11] extend OpenSHMEM to GPU clusters providing a unified memory space. However, as OpenSHMEM has no support for non-contiguous types, this implementation does not provide sufficient support to communicate non-contiguous GPU data. All these works focus on providing GPU-awarness for parallel programming models, and have been demonstrated to deliver good performance for contiguous data, but none of them provide full and efficient support for non-contiguous data residing in GPU memory.

2.2 MPI Datatype for Data Residing in GPU Memory

More recent works have focused on providing non-contiguous MPI datatype functionality for GPU data. Wang et. al. have improved the MVAPICH MPI implementation to provide the ability to transparently communicate non-contiguous GPU memory that can be represented as a single vector, and therefore translated into CUDA's two-dimensional memory copy (*cudaMemcpy2D*) [16]. A subsequent paper by the same authors tries to extend this functionality to many datatypes by proposing a vectorization algorithm to convert any type of datatype into a set of vector datatypes [15]. Unfortunately, indexed datatypes such as triangular matrices, are difficult to convert into a compact vector type. Using Wang's approach, each contiguous block in such an indexed datatype is considered as a single vector type and packed/unpacked separately from other vectors by its own call to *cudaMemcpy2D*, increasing the number of synchronizations and con-

sequently decreasing the performance. Moreover, no pipelining or overlap between the different stages of the datatype conversion is provided, even further limiting the performance.

Jenkins et. al. integrated a GPU datatype extension into the MPICH library [7]. His work focuses on the packing and unpacking of GPU kernels, but without providing overlaps between data packing/unpacking and other communication steps. Both Wang and Jenkins's work require transitioning the packed GPU data through host memory, increasing the load on the memory bus and imposing a significant sequential overhead on the communications. All of these approaches are drastically different from our proposed design, as in our work we favor pipelining between GPU data packing/unpacking and data movements, and also take advantage, when possible, of GPUDirect to bypass the host memory and therefore decrease latency and improve bandwidth.

3. DESIGN OF THE GPU DATATYPE ENGINE

The datatype constructs provided by the MPI Standard [5] give one the capability to define contiguous and non-contiguous memory layouts, allowing developers to reason at a higher level of abstraction, thinking about data instead of focusing on the memory layout of the data (for the pack/unpack operations). MPI defines data layouts of varying complexity: *contiguous* a number of repetitions of the same datatype without gaps in-between; *vector* defines a non-contiguous data layout that consists of equally spaced blocks of the same datatype; *indexed* specifies a noncontiguous data layout where neither the size of each block nor the displacements between successive blocks are equal; *struct* consists of location-blocklength-datatype tuples, allowing for the most flexible type of non-contiguous datatype construction.

Many MPI-based libraries and applications are using datatypes to move the burden of handling non-contiguous data from users to the MPI library implementors. For example, in the 2D stencil application of the Scalable HeterOgeneous Computing benchmark (SHOC) [3], two of the four boundaries are contiguous, and the other two are non-contiguous, which can be defined by a *vector* type. In the LAMMPS application from the molecular dynamics domain [13], each process keeps an array of indices of local particles that need to be communicated; such an access pattern can be captured by an *indexed* type. Hence, MPI datatypes help application developers alleviate the burden of manually packing and unpacking non-contiguous data. Therefore, extending the same datatype support to GPU data is extremely important for efficient programming in heterogeneous systems.

Current networks are bandwidth-oriented instead of latency-oriented, and fewer large messages provide better bytes per second transfer rates. Thus, in the context of non-contiguous data transfers, instead of generating a network operation for each individual contiguous block from the non-contiguous type, it is more efficient to pack the non-contiguous data into a contiguous buffer, and send less – but larger – messages. The same logic can be applied when data resides in GPU memory. Considering sending/receiving non-contiguous GPU datatypes, the four solutions presented in Figure 1 are usually employed.

a) Copy the entire non-contiguous data including the gaps from device memory into host memory. Accordingly, the data in host memory retains the same memory layout as the original, and the traditional CPU datatype engine can handle the pack/unpack operations. This solution provides good performance for memory layouts with little gaps, but cannot be generalized since it wastes a large amount of host memory for the intermediary copies, and has a potential degree of parallelism bounded by the CPU parallelism instead of taking advantage of the computational power of the GPU.

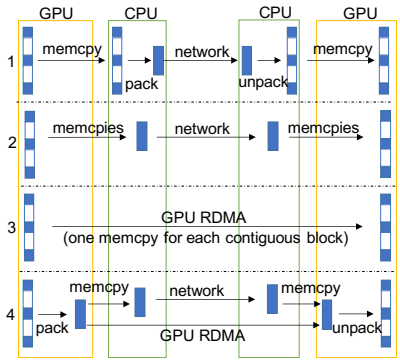


Figure 1: Four possible solutions for sending/receiving non-contiguous data residing in GPU memory.

b) The second solution is to issue one device-to-host memory copy (*cudaMemcpy*) for each piece of contiguous data, packing the data into a single, contiguous buffer. Once packed, the resulting contiguous buffer is sent using a traditional approach. The receiver will also generate the required host-to-device memory copies to scatter the temporary contiguous buffer into the expected locations in device memory. The overhead of launching lots of memory copies degrades performance. Moreover, a memory copy of each small block of contiguous data is not able to utilize the bandwidth of PCI-Express even with the help of multiple CUDA streams. Hence, the performance of this approach is limited.

c) A small improvement upon the second solution, instead of going through host memory, it issues one device-to-device memory copy for each piece of contiguous data, and directly copies data into the destination device memory. Similar to the previous solution, this alternative suffers from the overhead of launching too many memory copies and the low utilization of PCI-Express. Also, this solution only works when the peers have identical memory layouts and the hardware supports direct device-to-device copy.

d) The last solution is to utilize the GPU to pack and unpack non-contiguous data directly into/from a contiguous GPU buffer. Then the contiguous GPU-based buffer can either be moved between GPUs with hardware support, or – in the worst case – through the host memory.

Among all of the above solutions, we believe the last to be the most promising. Compared with the CPU, the GPU has many light-weight cores and significantly larger memory bandwidth, which might be beneficial for GPU packing/unpacking as these operations can be made embarrassingly parallel. Since the kernel is offloaded into the GPU

while the CPU is mostly idle (in an MPI call), it also provides the opportunity to pipeline pack/unpack with send/receive (discussed in Section 4). Moreover, this approach can be easily adapted to any hardware configuration: if GPUDirect is supported, we can bypass the host memory and use network RDMA capabilities, otherwise the copies to/from host memory can also be integrated in the pipeline, providing end-to-end overlap between pack/unpack and communications. In this paper, we present the design of a GPU datatype engine based on the 4th approach, taking advantage of CUDA *zero copy* and pipeline techniques to maximally the overlap between pack/unpack operations and communications.

In Open MPI, a datatype is described by a concise stack-based representation. Each stack element records type-specific parameters for a block, such as the number of contiguous elements in the block, the displacement of the first element from the beginning of the corresponding stack frame, and the number of blocks to be packed/unpacked. The most straightforward way to provide datatype support for GPU data would be to port the original (CPU-based) datatype engine into the GPU. However, porting the datatype stack to execute the pack/unpack operation on the GPU generates too many conditional operations, which are not GPU friendly. Thus, in order to minimize the branch operations executed by the GPU, we divided the pack/unpack operations into 2 stages. First, the host simulates the pack/unpack and generates a list of tuples <*source displacement, length, destination displacement*>. Because this list contains only relative displacements, it can be reused for a subsequent pack/unpack using the same datatype, and is therefore subject to caching optimizations. The second stage, which is represented by a kernel executing on a GPU, is using this list to execute – in parallel – as many of these pack/unpack operations as possible.

3.1 Vector Type

Other than *contiguous* datatype, *vector* is the most regular and certainly the most widely used MPI datatype constructor. A *vector* type is described by blocklengh and stride, where blocklength refers to the number of primitive datatypes that a block contains, and stride refers to the gaps between blocks. In our GPU datatype engine, we developed optimized packing/unpacking kernels specialized for a vector-like datatype. Similar to the 2 stages described above, the pack/unpack is driven by CPU. The pack kernel takes the address of the source and the destination buffers, blocklength, stride, and block count as arguments, and is launched in a dedicated CUDA stream. The operation is considered complete after a synchronization with the stream. The unpack kernel behaves similarly to the pack kernel.

While accessing global memory, a GPU device coalesces loads and stores issued by threads of a warp into as few transactions as possible to minimize DRAM bandwidth. Figure 2 shows the memory access pattern of GPU packing and unpacking kernels, forcing coalesced CUDA threads to access contiguous memory. Since device memory is accessed via 32-, 64-, or 128-byte memory-wide transactions [9], in order to minimize memory transactions, each thread theoretically should copy at least 4-bytes of data (128 bytes / 32 threads per warp). In our kernel, we force each thread to copy 8-bytes of data to reduce the number of total loops of each thread. In the case that data is not aligned with 8-bytes, the block is divided into 3 parts: the prologue and epilogue sections follow the original alignment, while the middle one follows the 8-byte alignment.

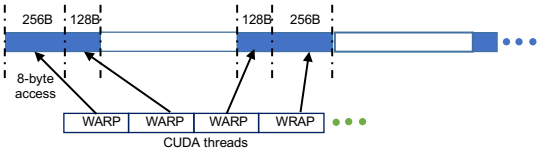


Figure 2: Access pattern of GPU pack/unpack kernels of *vector* type. The size of a CUDA block is a multiple of the warp size.

3.2 Less regular memory patterns

Datatypes other than *vector* are more complicated, and cannot be described in a concise format using only blocklengh and stride, and instead require a more detailed description including the displacement. However, one can imagine that any type can be described as a collection of vectors, even if some of the vectors have a count of a single element. Thus, it would be possible to fall back on a set of vector-based descriptions, and launch a vector kernel (similar to 3.1) for each entry. This design is unable to provide good performance as many kernels need to be launched, overwhelming the CUDA runtime.

Instead, we propose a general solution by re-encoding a representation of any complex datatype into a set of work units with similar sizes as shown in Figure 3 by picking a reasonable work unit size. As described above, each entry is identified by a tuple <*source displacement, destination displacement, length*> named *cuda_dev_dist*. Together with the source and destination buffers, these entries are independent and can be treated in parallel. When entries work on the same length they provide a good occupancy. The incomplete entries can either be delegated into another stream with a lower priority, or treated the same as all the other entries. We choose to treat them equally to the other entries, allowing us to launch a single kernel and therefore minimize launching overhead. A more detailed procedure for the pack/unpack operations is as follows:

- First, convert the representation of the datatype from stack-based into a collection of Datatype Engine Vectors (DEVs), where each DEV contains the displacement of a block from the contiguous buffer, the displacement of the corresponding block from the non-contiguous data and the corresponding blocklength (the contiguous buffer is the destination for the pack operation, and the source for the unpack).

- The second step is to compute a more balanced work distribution for each CUDA thread. Limited by the number of threads allowed per CUDA block, a contiguous block of data could be too large to use a single CUDA block, resulting in reduced parallelism. To improve parallelism, a DEV is assigned to multiple CUDA blocks. Instead of copying the entire DEV into GPU memory and letting each CUDA block compute its working range, we take advantage of the sequentiality of this operation to execute it on CPU, where each DEV is divided into several *cuda_dev_dist* (called CUDA DEV) of the same size S – plus a residue if needed – and each one is assigned to a CUDA WARP. Similar to the vector approach, each CUDA thread accesses 8-bytes of data each time; to fully

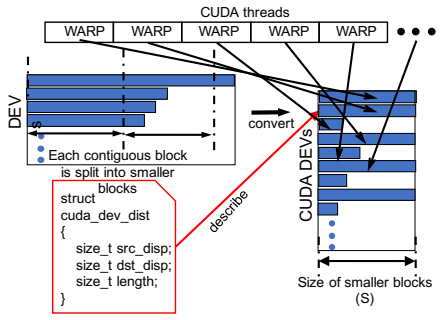


Figure 3: Access pattern of GPU pack/unpack kernels using the DEV methodology. The left *struct* describes a work unit for a CUDA WARP.

utilize all threads of a WARP, the size S must be a multiple of 8 times the CUDA WARP size (32). Thus, the lower bound of S is 256 bytes; but since CUDA provides loop unrolling capability, we set the size S to 1KB, 2KB or 4KB to reduce the branch penalties and increase opportunities for instruction level parallelism (ILP).

- Last, once the array of CUDA DEVs is generated, it is copied into device memory and the corresponding GPU kernel is launched. When a CUDA block finishes its work, it would jump N (total number of CUDA blocks) on the CUDA DEVs array to retrieve its next unit of work.

Since any datatype can be converted into DEV, this approach is capable of handling any MPI datatype. However, without a careful orchestration of the different operations, the GPU idles when the CPU is preparing the CUDA DEVs array. To improve the utilization of both GPU and CPU, we pipeline the preparation of the array and the execution of the GPU kernel: instead of traversing the entire datatype, the CPU converts only a part of the datatype, then a GPU kernel is launched to pack/unpack the converted part into a dedicated CUDA stream. The CPU can then continue converting while the GPU is executing the pack/unpack kernel. As the CUDA DEV is tied to the data representation and is independent of the location of the source and destination buffers, it can be cached, either in the main or GPU memory, thereby minimizing the overheads of future pack/unpack operations.

4. INTEGRATION WITH Open MPI

This section describes how we integrated the GPU datatype engine with the Open MPI infrastructure. The Open MPI communication framework – outside the MPI API – is divided into three layers, with each one playing a different role. At the top level, the PML (point-to-point management layer) realizes the MPI matching, fragments, and reassembles the message data from point-to-point communications. Different protocols based on the message size (short, eager, and rendezvous) and network properties are available (latency, bandwidth, RMA support), and the PML is designed to pick the best combination in order to maximize network usage. Below the PML, the BML (BTL management layer) manages different network devices, handles multi-link data transfers, and selects the most suitable BTL for a communication based on the current network device where messages go through. The lowest layer, the BTL (byte transfer layer), is used for the actual point-to-point byte movement. Each

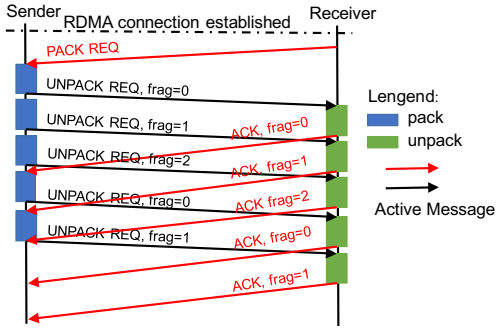


Figure 4: Pipelined RDMA protocol for send/receive of non-contiguous GPU-resident data.

BTL provides support for a particular type of network (TCP, shared memory, InfiniBand, Portals, uGNI and so on), and mainly deals with low level network communication protocols where the focus is on optimally moving blobs of bytes. As different network devices have their own optimal communication protocols, the methodology of GPU datatype engine integration is realized at the level of the network device (the BTL). In this paper, we focus on the shared memory and InfiniBand BTL, and propose support for two types of protocols: RDMA and copy in/out. Of course, these protocols are adaptable to the GPU and network capabilities, and can be easily extended to other BTLs.

4.1 RDMA Protocol

NVIDIA's GPUDirect technology improves GPU to GPU communication by allowing data movement between GPU devices without going through host memory. According to [18], PCI-E bandwidth of GPU-GPU is larger than the one of CPU-GPU, therefore, RMDA GPU-GPU communication not only provides shortest data path between processes, but also has higher PCI-E utilization. In intra-node communications, CUDA IPC allows the GPU memory of one process to be exposed to the others, and therefore provides a one sided copy mechanism similar to RDMA. In inter-node communication, GPUDirect RDMA supports data exchange directly between the GPU and the network interface controller using PCI-E, enabling direct GPU data movement between nodes. Taking advantage of GPUDirect, a basic GPU RDMA protocol can be implemented as follows: sender packs a non-contiguous GPU datatype into a contiguous GPU buffer, and then exposes this contiguous GPU buffer to the receiver process. If the synchronization is done at the level of an entire datatype packing, the receiver should not access the data until the sender has completed the pack operation. The resulting cost of this operation is therefore the cost of the pack, followed by the cost of the data movement plus the cost of the unpack. However, if a pipeline is installed between the 2 processes, the cost of the operation can be decreased, reaching the invariant (which is the cost of the data transfer) plus the cost of the most expensive operation (pack or unpack) on a single fragment, which might represent a reduction by nearly a factor of 2 if the pipeline size is correctly tuned. This approach also requires a smaller contiguous buffer on the GPU as the segments used for the pipeline can be reused once the receiver completes the unpack and notifies the sender that its operation on a segment is completed. The Open MPI's PML layer is already capable of implementing message fragmentation and can send/receive them in a

pipelined fashion. However, applying this pipelining feature directly for PML-based RDMA protocols is costly because PML is the top-level layer, and pipelining in this layer requires going through the entire Open MPI infrastructure to establish an RDMA transfer for each fragment. Starting an RDMA transfer requires the sender to send its GPU memory handle to the receiver for mapping to its own GPU memory space, which is a costly operation. With such an approach any benefits obtained from pipelining will be annihilated by the overhead of registering the RDMA fragments. To lower this cost, we implement a light-weight pipelined RDMA protocol directly at the BTL level, which only proposes a single one-time establishment of the RDMA connection (and then caching the registration).

The implementation of our pipelined RDMA protocol uses BTL-level *Active Message* [4], which is an asynchronous communication mechanism intended to expose the interconnection network's flexibility and performance. To reduce the communication overhead, each message header contains the reference of a callback handler triggered on the receiver side, allowing the sender to specify how the message will be handled on the receiver side upon message arrival.

Taking advantage of *Active Message* communications, the sender and receiver are dissociated, and they synchronize only when needed to ensure smooth progress of the pack/unpack operations. While the sender works on packing a fragment, the receiver is able to unpack the previous fragment, and then notify the sender that the fragment is now ready for reuse. Once the sender receives the notification from the receiver that a fragment can safely be reused, it will pack the next chunk of data (if any) directly inside. Figure 4 presents the steps of the pipelined RDMA protocol. Besides the address of a callback handler for invoking the remote pack or unpack functions, the header in our implementation also contains additional information providing a finer grain control of the pack/unpack functions (such as the index of the fragment to be used). In our RDMA protocol, the packing/unpacking is entirely driven by the receiver acting upon a GET protocol, providing an opportunity for a handshake prior to the beginning of the operation. During this handshake, the two participants agree on the type of datatype involved in the operation (contiguous or non-contiguous) and the best strategy to be employed. If the sender datatype is contiguous, the receiver can use the sender buffer directly for it's unpack operation, without the need for further synchronizations. Similarly, if the receiver datatype is contiguous the sender is then allowed to pack directly into the receiver buffer, without further synchronizations. Of course, based on the protocol used (PUT or GET), a final synchronization might be needed to inform the peer about the data transfer completion. The more detailed description of the pipelined RDMA protocol is as follows.

- **Sender:** detects if GPU RDMA is supported between the two MPI processes, and requests a temporary GPU-residing buffer from the datatype engine. It then retrieves the memory handle of this temporary GPU buffer, and starts the RDMA connection request providing the memory handle and the shape of the local datatype in a request message. It then waits until a pack request is received from the receiver. After finishing packing a fragment, an unpack request is sent to the receiver signaling the index of the fragment to be unpacked. In case the GPU buffer is full, or the pipeline depth has been reached, the sender waits until it receives an acknowledgment from the receiver notifying that the unpacking is finished for a particular fragment that can be reused for the next pack. This stage repeats until all the data is packed.

- **Receiver:** upon receiving an RDMA request it maps the memory handle provided by the sender into its own memory, allowing for direct access to the sender's GPU buffer. After the RDMA connection is established, the receiver signals the sender to start packing, and then waits until it receives an unpack request from the sender. After finishing the unpacking of each fragment, the receiver acknowledges the sender, allowing the fragment to be reused. In the case where the sender and the receiver are bound to different GPUs, we provide the option to allow the receiver to allocate a temporary buffer within its device memory and move the packed data from sender's device memory into its own memory before unpacking. In some configurations, going through this intermediary copy delivers better performance than accessing the data directly from remote device memory.

4.2 Copy In/Out Protocol

In some cases, due to hardware limitations or system level security restrictions, the IPC is disabled and GPU RDMA transfers are not available between different MPI processes. To compensate for the lack of RDMA transfers we provide a copy in/copy out protocol, where all data transfers go through host memory. It is worth noting that this approach is extremely similar to the case when one process uses device memory while the other only uses host memory. Open MPI handles non-contiguous datatypes on the CPU by packing them into a temporary CPU buffer prior to communication. When GPU RDMA is not available, we forced Open MPI to always consider all data as being in host memory, and therefore it always provides a CPU buffer even for datatypes residing in device memory. When the datatype engine detects that the corresponding non-contiguous data is actually in device memory, it allocates a temporary GPU buffer (with the same or smaller size than the CPU buffer) for packing. Once this GPU buffer is full, the packed data is copied into the CPU buffer for further processing. This procedure repeats until the entire data is packed. A similar mechanism applies to unpack.

Unlike the RDMA protocol, extra memory copies between device and host memory are required. To alleviate the overhead of such memory transfer, pipelining can also be used by allowing the sender to partially pack the data, fragment after fragment, and allow the receive to unpack once it receives each packed fragment. Therefore, the pipelining becomes more complex, overlapping packing/unpacking on the GPU, with device-to-host data movement and intra-node communication. Another CUDA capability, *zero copy*, can be exploited to minimize the memory copy overhead. Instead of using the CPU to explicitly drive memory movement, the CPU buffer is mapped to GPU memory with the help of CUDA UMA, and then the data movement is implicitly handled by hardware, which is able to overlap it with pack/unpack operations. Overall, as indicated in the experimental Section 5, copy in/out protocol is a general solution suitable for most platforms, and delivers good performance – especially once integrated with a pipelined protocol.

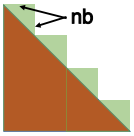


Figure 5: Triangular matrix (red one) vs Stair triangular matrix (red and green one), width and height of stair *nb* is multiple of CUDA block size

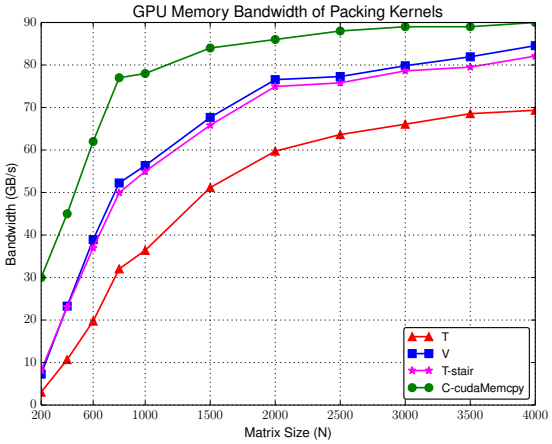


Figure 6: GPU memory bandwidth of packing kernels for sub-matrix and lower triangular matrix comparing with contiguous data of the same size. "T" represents triangular matrix, "V" represents sub-matrix, "C" represents contiguous matrix

5. EXPERIMENT AND EVALUATION

We evaluate our datatypes packing/unpacking methodology using four types of benchmarks. First, we investigate the performance of the GPU datatype engine. Second, we look at inter-process GPU-to-GPU communication through a non-contiguous data ping-pong test, and compare with MVAPICH2.1-GDR. Third, we figure out the minimal GPU resources required for GPU packing/unpacking kernels to achieve optimal overall performance when communication is engaged. Last, we analyze the impact on non-contiguous data transfer when access to the GPU resource is limited (the GPU is shared with another GPU intensive application). Experiments are carried out on an NVIDIA PSG cluster: each node is equipped with 6 NVIDIA Kepler K40 GPUs with CUDA 7.5 and 2 deca-core Intel Xeon E5-2690v2 Ivy Bridge CPUs; nodes are connected by FDR IB.

5.1 Performance Evaluation for Datatype Engine

In this section, we investigate the performance of our GPU datatype engine by using two commonly used datatypes: *vector* and *indexed*. These datatypes are representative of many dense linear algebra based applications, as they are the basic blocks of the ScaLAPACK data manipulation. More precisely, these types are represented as a sub-matrix and an (upper or lower) triangular matrix. Considering a sub-matrix with column-major format, each column is contiguous in memory, and the stride between columns is the size of the columns in the original big matrix, which follows the characteristic of a *vector* type (shown as "V" in the following figures). In the lower triangular matrix case, each column

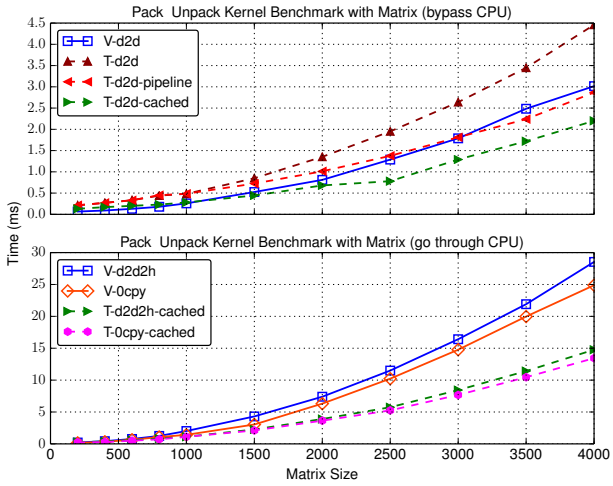


Figure 7: Performance of pack and unpack sub-matrix and lower triangular matrix varies by matrix size.

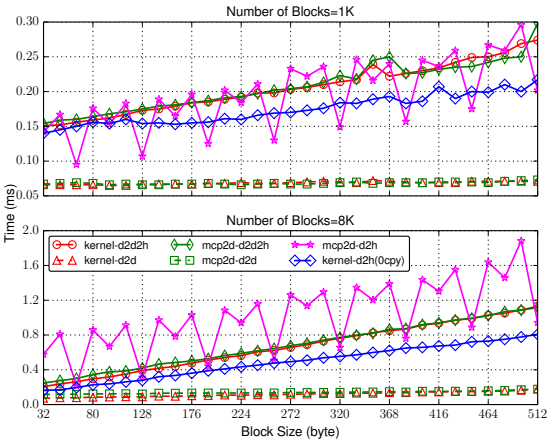


Figure 8: *vector* pack/unpack performance vs *cudaMemcpy2D*. "kernel" represents our pack/unpack kernels. "mcp2d" represents *cudaMemcpy2D*. "d2d" represents non-contiguous data packed into a GPU buffer. "d2d2h" represents "d2d" followed by a device-host data movement. "d2h" means non-contiguous GPU data moved directly into CPU buffer.

is contiguous in memory with a size smaller by one element than the size of the previous column; and the strides between consecutive columns are equal to the previous stride plus 1, which can be described by an *indexed* type (shown as "T" in the following figures). First, we evaluate the performance of our packing/unpacking kernels by measuring GPU memory bandwidth. Figure 6 presents the GPU memory bandwidth achieved from packing these two datatypes into local GPU buffer using our CUDA kernel compared with moving contiguous data of the same size using *cudaMemcpy*. *cudaMemcpy* is already the optimal implementation for moving contiguous GPU data, which can be treated as the practical peak of GPU memory bandwidth. Compared to *cudaMemcpy*, our GPU packing kernel is able to obtain 94% of the

practical peak for a *vector* type. The memory instructions in the unpacking kernel are the same as the ones in the packing kernel – but in the opposite direction – and therefore the unpacking kernel delivers the same performance as packing kernels; this is not presented in the figure. For a triangular matrix, each column has a different size, which results in inefficient occupancy of the CUDA kernels; therefore, a GPU packing kernel is only able to achieve 80% of the GPU memory's peak bandwidth. In order to prove that the bandwidth difference between the sub-matrix and the triangular matrix is indeed from the less efficient GPU occupancy, the triangular matrix is modified to a stair-like triangular matrix (Figure 5). Thus, the occupancy issue can be reduced by setting the stair size *nb* to a multiple of a CUDA block size to ensure no CUDA thread is idle. Sure enough, it is able to deliver almost the same bandwidth as the *vector* type.

After studying the performance of the packing/unpacking kernels, we measure the intra-process performance of packing non-contiguous GPU-resident data to evaluate the GPU datatype engine. Because of the current limitation of GPUDirect, using an intermediate host buffer for sending and receiving over the network is better for large messages than direct communication between remote GPUs in an InfiniBand environment [14]. Thus, studying the case of going through host memory is also necessary. In the following benchmark, one process is launched to pack the non-contiguous GPU data into a local GPU buffer, followed by a data movement to copy the packed GPU data into host memory; and then, the unpacking procedure moves the data from host memory back into the original GPU memory with the non-contiguous layout. Accordingly, the time measurement of the benchmarks in this section contains two parts: "d2d" measures the time of packing/unpacking non-contiguous data into/from a contiguous GPU buffer; and "d2d2h" measures the time of packing/unpacking plus the round trip device-host data movements. We also apply *zero copy*, shown as "0cpy," to use the CUDA UMA to map the CPU buffer to GPU memory. In this case, the GPU to CPU data movement is taken care of by hardware implicitly. Since *zero copy* involves implicit data transfer, we are only able to measure its total time without having a separate in-GPU pack/unpack time to show in figures.

Figure 7 shows the results of a double precision sub-matrix and lower triangular matrix, with respect to matrix size. From the figure, a number of interesting trends can be observed. First, the pipelining discussed in Section 3.2 overlaps the preparation of the CUDA DEVs with GPU pack/unpack kernels, almost doubling the performance. If the CUDA DEVs are cached in GPU memory (shown as "cached"), the preparation cost can be omitted; therefore, by caching the CUDA DEVs, the packing/unpacking performance is improved when working on data types of the same format. Second, even though it takes the same time (if CUDA DEVs are not cached) to pack/unpack a sub-matrix and triangular matrix of the same matrix size on a GPU, one must note that the triangular matrix is half the size of a sub-matrix; therefore, compared with a vector approach, the overhead of CUDA DEVs preparation is significant – even with pipelining – which also demonstrates the importance of caching the CUDA DEVs. Since the MPI datatype describes data layout format, not data location, by spending a few MBs of GPU memory to cache the CUDA DEVs, the packing/unpacking performance could be significantly improved when using the

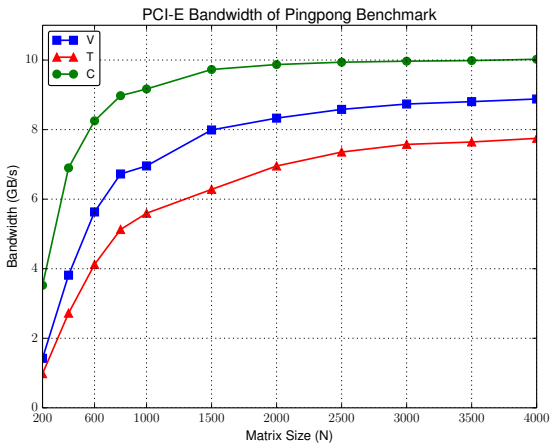


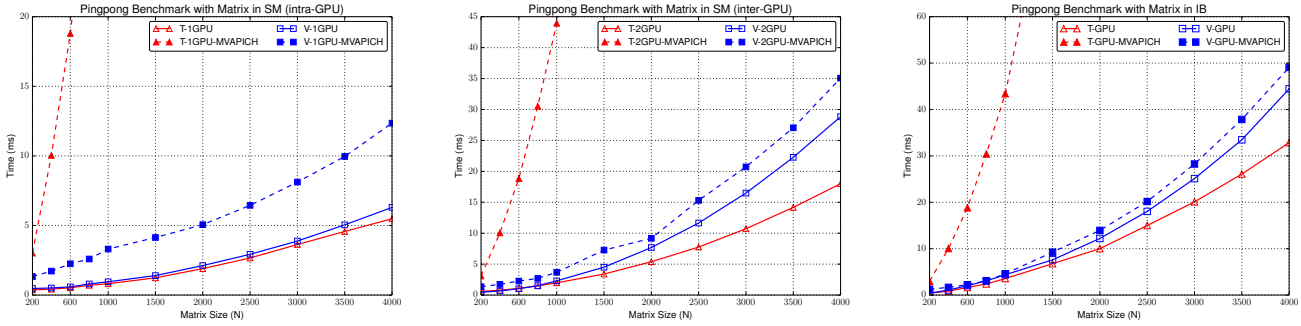
Figure 9: PCI-E bandwidth of vector and indexed data type comparing with contiguous data.

same data type repetitively. Third, since *zero copy* is able to overlap the device-host communication with the GPU kernel, it is slightly faster than explicitly moving data between device and host memory after/before pack/unpack kernels. In all remaining figures, the *zero copy* is always enabled if going through host memory is required.

Alternatively, CUDA provides a two-dimensional memory copy *cudaMemcpy2D* to move vector-like data. Figure 8 presents the comparison between our *vector* pack/unpack kernel and *cudaMemcpy2D*, when the numbers of contiguous blocks are fixed at 1000 and 8000, while block size varies covering both small and large problems. Since using our pack kernel to move vector-like non-contiguous GPU data is equivalent to initiating a device to host data movement using *cudaMemcpy2D*, we test it in three ways (device-to-device"mcp-d2d", device-to-device-to-host "mcp2d-d2d2h", and device-to-host "mcp2d-d2h"). As seen in the figure, the performance of *cudaMemcpy2D* between device and host memory highly depends on the block size: block sizes that are a multiple of 64 bytes perform better, while others experience significant performance regression – especially when the problem size increases. For non-contiguous data movement within a GPU, our kernels achieve almost the same performance as *cudaMemcpy2D*. Our DEV pack/unpack kernel is not compared with CUDA since CUDA does not provide any alternative function for irregular non-contiguous GPU data movement.

5.2 Full Evaluation: GPU-GPU Communication with MPI

In this section, we evaluate the performance of the GPU datatype engine integration with the Open MPI infrastructure. The performance is assessed using an MPI "ping-pong" benchmark. In a shared memory environment, the RDMA protocol over CUDA IPC is used to avoid extraneous memory copies between host and device. In a distributed memory setting, GPU data goes through host memory for communication. According to [14], even though the GPUDirect RDMA allows direct intra-node GPU data communication, it only delivers interesting performance for small messages (less than 30KB), which is not a typical problem size of GPU applications. Instead, when pipelining through host memory



(a) Shared Memory Intra-GPU (b) Shared Memory Inter-GPU (c) Distributed Memory

Figure 10: Ping-pong benchmark with matrices. "V" refers to sub-matrix, "T" refers to triangular matrix.

and overlapping GPU pack/unpack kernels, the GPU-CPU data movement and inter-node data transfer performs better. Therefore, in a distributed memory environment, we always pipeline through host memory. Based on such a setup, packed GPU data always goes through PCI-E for communication no matter if it is in a shared or distributed memory environment; thus, PCI-E bandwidth could be a bottleneck of overall communication in a ping-pong benchmark. Similar to last section, we first evaluate the integration of the GPU datatype engine with OpenMPI by measuring PCI-E bandwidth achieved by *vector* and *indexed* datatypes, comparing data in contiguous format of the same size, with results shown in Figure 9. Thanks to the pipeline mechanism discussed in Section 4, we achieved 90% and 78% of the PCI-E bandwidth for *vector* and *indexed* types, respectively, by selecting a proper pipeline size.

Then, in the following ping-pong benchmarks, we explore both a shared memory ("SM") and a distributed memory (using InfiniBand "IB") environment under the following configurations with several commonly used data types, and compare them with the state-of-art MVAPICH2:

- "1GPU": both sender and receiver use the same GPU.
- "2GPU": sender and receiver use different GPUs. Data is sent over network (PCI-E or InfiniBand) to the receiver process.
- "CPU": the non-contiguous data is in host memory. This benchmarks the Open MPI CPU datatype engine.

5.2.1 Vector and Indexed Type

Figure 10 presents the ping-pong benchmark with regard to the matrix size in both "SM" and "IB" environments. As discussed in Section 4.1, in the "SM" environment with CUDA IPC support, we provide two options for unpacking in the receiver side: first, the receiver unpacks directly from the packed buffer in the remote GPU memory; second, the receiver process copies the packed buffer into a local GPU buffer prior to unpacking. The first option involves a lot of small chunks of data fetching from remote device memory, generating too much traffic and under-utilizing the PCI-E. In comparison, the second option groups small data into a big data movement between GPUs, minimizing the traffic on the PCI-E and becoming faster. Based on our experiment, by using a local GPU buffer, the performance is 5-10% faster than directly accessing remote GPU memory; so limited by the space, we always use the second option in later benchmarks. The "1GPU" case omits the data movement between GPUs, being at least 2x faster than any "2GPU" case. Therefore, even though data is already packed to a contiguous format, the data transfer between GPUs over PCI-E is still the bottleneck of non-contiguous GPU data communication in an "SM" environment. Compared with MVAPICH2, our implementation is always significantly faster, independent of the datatype. Because of MVAPICH2's vectorization algorithm converting any type of datatype into a set of vector datatypes [15], each contiguous block in such an *indexed* datatype is considered as a single vector type and packed/unpacked separately, resulting in sub-optimal performance. As seen in the figure, their *indexed* implementation is slow, going outside the time range once the matrix size reached 1000.

In an "IB" environment, even though data is transitioned through host memory before being sent over the network, thanks to *zero copy*, the device-to-host transfers are handled automatically by the hardware, and this transfer is overlapped with the execution of the GPU pack/unpack kernels. In this environment we notice a significantly more desirable behavior from MVAPICH2, at least for the vector type. However, our approach achieves a roughly 10% improvement for the *vector* type. Similar to the *indexed* result of "SM" environment, the MVAPICH2 performance is quickly outside the range for matrices as small as 1500.

5.2.2 Vector-Contiguous

When using MPI datatypes, the sender and the receiver can have different datatypes as long as the datatype signatures are identical. Such features improve the application's ability to reshape data on the fly, such as in FFT and matrix transpose. In FFT, one side uses a *vector*, and the other side uses a *contiguous* type. Figure 11 shows the ping-pong performance with such datatypes of different sizes. As seen in the figure, taking the benefit of GPU RDMA and *zero copy*, our implementation performs better than MVAPICH2 in both shared and distributed memory environments.

5.2.3 Matrix Transpose

Matrix transpose is a very complex operation and a good stress-test for a datatype engine. With column-major storage, each column is contiguous in memory. A matrix can be described by a *contiguous* type or *vector* type if only accessing the sub-matrix. After the transpose, each column can be represented by a *vector* type with a block length of 1 element; consequently, the whole transposed matrix is a collection of N *vector* types. Figure 12 shows the bench-

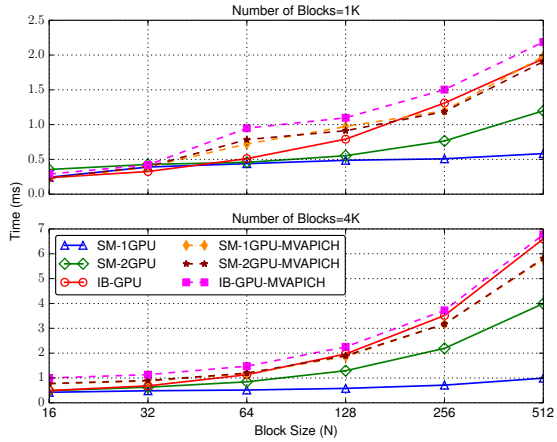


Figure 11: Ping-pong benchmark with vector and contiguous data type.

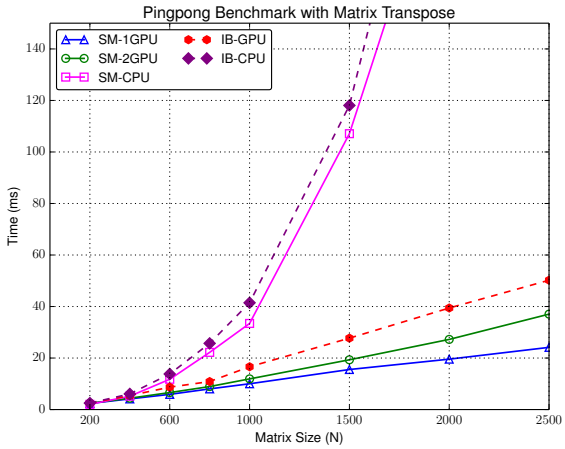


Figure 12: Ping-pong benchmark for matrix transpose in both shared and distributed memory environment.

mark for a matrix transpose depending on the matrix size. Since there is only 1 element in each block, the memory access is not following the coalesced rule, and the performance is not comparable with the regular *vector* type. However, such difficulty also occurs in the CPU implementation, benefiting from the parallel capability and high memory bandwidth, our GPU datatype implementation is at least 10x faster than the CPU version of Open MPI. Lacking stable support for such a datatype, MVAPICH2 crashed in this experiment and is not included in the figure.

5.3 GPU Resources of Packing/Unpacking Kernels

In previous benchmarks, GPU packing/unpacking kernels aggressively used CUDA's Streaming Multiprocessor (SM). Figure 6 shows that by using as many CUDA cores as possible, the kernels are able to achieve more than 80 GB/s of GPU memory bandwidth. However, in most cases, each MPI process is attached to a separate GPU; since GPUs are connected by PCI-E, then the communication bandwidth

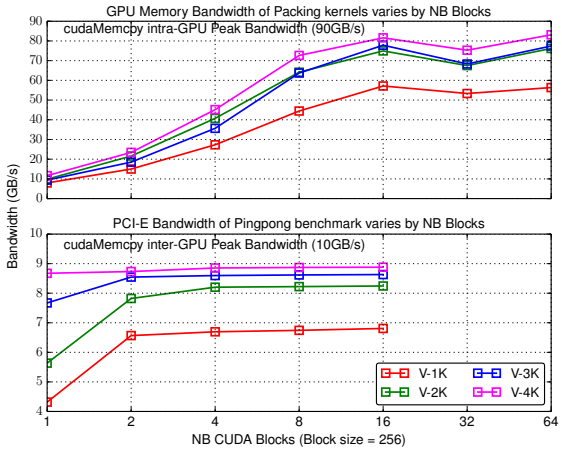


Figure 13: GPU memory and PCI-E bandwidth of pack/unpack sub-matrix "V" data types varies by number of blocks used for kernel launching. Matrix size varies from 1K to 4K.

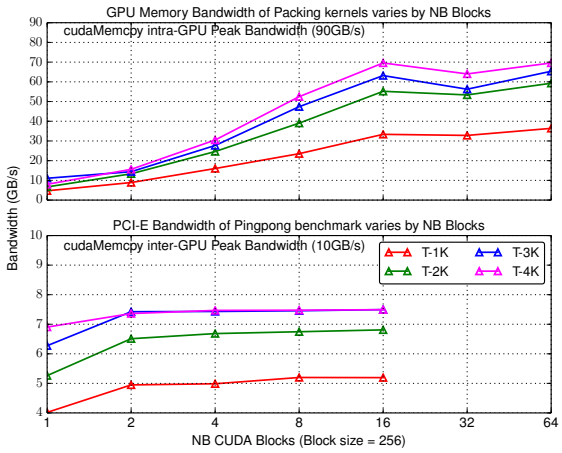


Figure 14: GPU memory and PCI-E bandwidth of pack/unpack triangular matrix "T" data types varies by number of blocks used for kernel launching. Matrix size varies from 1K to 4K.

is limited to the 10 GB/s available through PCI-E. In this section, we investigate the minimal resources required to fulfill the PCI-E bandwidth. The top figures of Figure 13 and Figure 14 present the GPU memory bandwidth of packing/unpacking kernels for sub-matrix "V" and triangular matrix "T" data types. NVIDIA's Kepler GPU has four warp schedulers per SM; therefore, in order to achieve the best GPU occupancy, the block size should be a multiple of 128 threads (32 threads per warp). In the benchmark, we use 256 threads per block. As seen in the figure, it requires 16 blocks to achieve the peak bandwidth, and achieves 10 GB/s (the peak of PCI-E bandwidth) by launching only 2 blocks in most cases. Hence, theoretically, by using no more than 2 blocks, the cost of packing/unpacking can be hidden by communication over PCI-E when pipelining is applied. Similarly, bottom figures of Figure 13 and Figure 14 illustrates that the PCI-E bandwidth of the same two data types varies

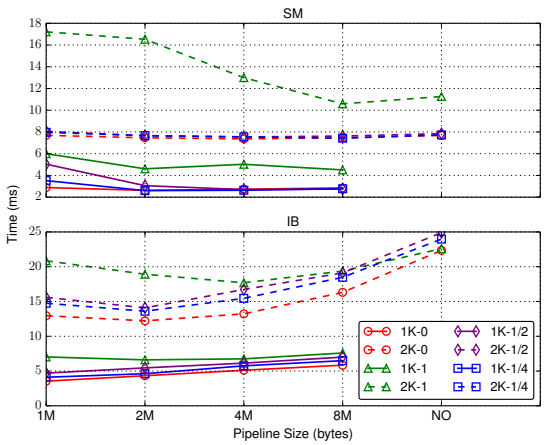


Figure 15: Ping-pong benchmark with partial GPU resources available. In the legend, the number after the matrix size is the ratio of GPU resources occupied.

by the number of blocks used for kernel launching. As seen in the figure, as we expected, the bandwidth becomes stable when using at least 2 CUDA blocks. The K40 GPU has 15 SMs, so in the worst case, one seventh of the GPU SMs are required to overlap the cost of packing/unpacking kernels with communications over PCI-E. In other cases when each MPI process is attached to the same GPU or future NVLink is introduced with higher bandwidth, our GPU datatype engine can be easily adapted by tuning CUDA blocks to fulfill bandwidth.

5.4 Pipeline and Resource Contention Effects

All previous benchmarks were executed under the assumption that the GPU resources are readily available for pack/unpack. As in some cases, overlapping communication with computation is possible, the application might be using the GPU while MPI communications with non-contiguous datatypes are ongoing. In this section, we investigate how resource contention affects the pack/unpack performance, as well as the pipelining discussed in Sec 4.

In this benchmark, we launch a special kernel to continuously occupy a fixed percentage of the GPU while executing the ping-pong benchmark. The grid size of the kernel varies to occupy full, half, or a quarter of the GPU resources; we then measure the ping-pong performance under these scenarios. The datatypes used are (*vector*) sub-matrices of size 1000 by 1000 and 2000 by 2000, since they are typical problem sizes for GPU applications in the linear algebra domain. The results are shown in Figure 15. Thanks to the pipelining methodology, a proper pipeline size improves the performance in both shared and distributed memory machines. However, as seen in the figure, with a small pipeline size the pack/unpack operations are divided into many small GPU kernels, and the scheduling of such kernels could be delayed by the CUDA runtime when the occupancy of the GPU is high. Our GPU pack/unpack kernels mainly contain memory operations without floating point operations, and they are memory bound. Therefore, as long as the GPU is not fully occupied, our pack/unpack methodology is not significantly affected. By using a proper pipeline size, we limit the loss of performance to under 10%.

6. CONCLUSIONS

As heterogeneous compute nodes become more pervasive, the need for programming paradigms capable of providing transparent access to all types of resources becomes critical. The GPU datatype engine presented in this paper takes advantage of the parallel capability of the GPUs to provide a highly efficient in-GPU datatype packing and unpacking. We integrate the GPU datatype engine into the state-of-art Open MPI library, at a level of integration such that all communications with contiguous or non-contiguous datatypes will transparently use the best packing/unpacking approach. The different protocols proposed, RDMA, copy in/out, pipeline, and the use of novel technologies, such as GPUDirect, drastically improve the performance of the non-contiguous data movements, when the source and/or the destination buffers are GPU-resident. The described principles can be extended to other accelerators, and other types of networks in a simple manner. They can also be the basic blocks for defining and implementing concepts outside the scope of point-to-point constructs, such as collective and RDMA operations.

7. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under award #1339820. We would also like to thank NVIDIA for their support and for providing access to their resources.

8. REFERENCES

[1] A. M. Aji, J. Dinan, D. Buntinas, P. Balaji, W.-c. Feng, K. R. Bisset, and R. Thakur. MPI-ACC: An Integrated and Extensible Approach to Data Movement in Accelerator-based Systems. In *HPCC'12*, pages 647–654, Washington, DC, USA, 2012.

[2] L. S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[3] 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 *GPGPU-3 Workshop*, pages 63–74, New York, NY, USA, 2010.

[4] T. Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *ISCA'92*, pages 256–266, 1992.

[5] M. Forum. MPI-2: Extensions to the message-passing interface. In *Univ. of Tennessee, Knoxville, Tech Report*, 1996.

[6] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *EuroMPI'04*, pages 97–104, Budapest, Hungary, 2004.

[7] J. Jenkins, J. Dinan, P. Balaji, T. Peterka, N. Samatova, and R. Thakur. Processing MPI Derived Datatypes on Noncontiguous GPU-Resident Data. *Parallel and Distributed Systems, IEEE Transactions on*, 25(10):2627–2637, Oct 2014.

[8] O. Lawlor. Message passing for GPGPU clusters: CudaMPI. In *CLUSTER'09.*, pages 1–8, Aug 2009.

[9] NVIDIA. NVIDIA CUDA Toolkit Documentation. http://docs.nvidia.com/cuda/index.html, 2015.

[10] NVIDIA. NVIDIA GPUDirect. https://developer.nvidia.com/gpudirect, 2015.

[11] S. Potluri, D. Bureddy, H. Wang, H. Subramoni, and D. Panda. Extending OpenSHMEM for GPU Computing. In *IPDPS'13*, pages 1001–1012, May 2013.

[12] R. Ross, N. Miller, and W. Gropp. Implementing Fast and Reusable Datatype Processing. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 2840 of *Lecture Notes in Computer Science*, pages 404–413. Springer Berlin Heidelberg, 2003.

[13] T. Schneider, R. Gerstenberger, and T. Hoefler. Micro-Applications for Communication Data Access Patterns and MPI Datatypes. In *EuroMPI'12*, pages 121–131, 2012.

[14] R. vandeVaart. Open MPI with RDMA support and CUDA. In *NVIDIA GTC'14*, 2014.

[15] H. Wang, S. Potluri, D. Bureddy, C. Rosales, and D. Panda. GPU-Aware MPI on RDMA-Enabled Clusters: Design, Implementation and Evaluation. *Parallel and Distributed Systems, IEEE Transactions on*, 25(10):2595–2605, Oct 2014.

[16] H. Wang, S. Potluri, M. Luo, A. Singh, X. Ouyang, S. Sur, and D. Panda. Optimized Non-contiguous MPI Datatype Communication for GPU Clusters: Design, Implementation and Evaluation with MVAPICH2. In *CLUSTER'11*, pages 308–316, Sept 2011.

[17] 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. *Computer Science - Research and Development*, 26(3-4):257–266, 2011.

[18] L. Wang, W. Wu, Z. Xu, J. Xiao, and Y. Yang. BLASX: A High Performance Level-3 BLAS Library for Heterogeneous Multi-GPU Computing. In *ICS'16*, Istanbul, Turkey, 2016.