Enabling Fast, Noncontiguous GPU Data Movement in Hybrid MPI+GPU Environments

John Jenkins
Department of Computer Science
North Carolina State University
jpjenki2@ncsu.edu

Pavan Balaji
Math. and Computer Science Div.
Argonne National Laboratory
balaji@mcs.anl.gov

James Dinan
Math. and Computer Science Div.
Argonne National Laboratory
dinan@mcs.anl.gov

Nagiza F. Samatova
Department of Computer Science
North Carolina State University
Oak Ridge National Laboratory
samatova@csc.ncsu.edu

Rajeev Thakur
Math. and Computer Science Div.
Argonne National Laboratory
thakur@mcs.anl.gov

ABSTRACT

Lack of efficient and transparent interaction with GPU data in hybrid MPI+GPU environments challenges GPU-acceleration of large-scale scientific and engineering computations. A particular challenge is the efficient transfer of *noncontiguous* data to and from GPU memory. MPI supports such transfers through the use of *datatypes*, however an efficient means of utilizing datatypes for noncontiguous data in GPU memory is not currently known.

To address this gap, we present the design and implementation of efficient MPI datatypes processing system, which is capable of efficiently processing arbitrary datatypes directly on the GPU. We present a means for converting conventional datatype representations into a GPU-tractable format, which exposes parallelism. Fine-grained, element-level parallelism is then utilized by a GPU kernel to perform in-device packing and unpacking of noncontiguous elements. We demonstrate a several-fold performance improvement for noncontiguous column vectors, 3D array slices, and 4D array subvolumes over CUDA-based alternatives. Compared with optimized, layout-specific implementations, our approach incurs low overhead, while enabling the packing of datatypes that do not have a direct CUDA equivalent. These improvements are demonstrated to translate to significant improvements in end-to-end, GPU-to-GPU communication time.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Concurrent programming structures*

General Terms

Design, Performance

Keywords

MPI, GPU, Derived datatypes

1. INTRODUCTION

A great amount of interest in the HPC community has been centered on the capabilities of graphics processing units (GPUs) as inexpensive, many-core accelerators. Evidence of this is seen in recent Top500 lists of supercomputers [1], where GPU accelerators are gaining in popularity due to their effectiveness over a wide range of computational loads.

A number of technical challenges arise from the addition of a fundamentally different computing architecture to existing systems. Aside from the cost of developing, porting, and optimizing codes to run on the GPU, there is a greater concern about how their addition affects algorithms relying on point-to-point and collective communication. For the currently prevailing model of discrete graphics processing hardware with memory that is separate from the CPU's RAM, any communication operation involving data resident in GPU memory requires moving data between GPU and CPU memories, effectively adding another "hop" in the communication graph. In addition, the MPI Standard [9] does not define MPI's interaction with GPU memory managed by, for example, OpenCL [6] or CUDA [11] programming models.

Enabling MPI to interact directly with data stored in GPU memory is an important step toward providing transparent and efficient integration of GPUs into HPC applications. In addition to communication operations that transfer contiguous chunks of GPU data, a significant and challenging problem is the communication of *noncontiguous* data, which is enabled through the use of MPI *datatypes*. MPI datatypes allow the programmer to define an arbitrary layout for data which is sent or received (or input/output for MPI I/O operations) using a single MPI operation. A common use of data types in scientific computing applications is the transfer of noncontiguous array slices vectors from GPU to GPU (in applications such as stencil computations that require array boundary updates between processes [8, 10, 13]).

Communication operations like these must be done efficiently for the computational benefit of using the GPU to outweigh the cost of data transfer into CPU main memory. In order to utilize network and I/O resources effectively, noncontiguous data is first *packed* into contiguous buffers. The transfer granularity between the GPU and the CPU and over the network must be sufficiently to make ef-

ficient use of the PCIe bus. Considerations such as these, severely impact the effectiveness of conventional GPU-CPU data copy operations and present significant productivity and performance challenges to the manual packing of data in GPU memory.

In this work, we present the design of an efficient, in-GPU noncontiguous datatype processing system. We focus on NVIDIA's CUDA interface, however techniques presented are applicable broadly across accelerators and accelerator programming models. Our approach defines a datatype representation that exposes fine-grain parallelism and utilizes a GPU kernel that can efficiently utilize this parallelism to accelerate data movement. We demonstrate that our approach is efficient at packaging noncontiguous when compared with CUDA's built-in data layouts and hand-coded packing kernels. In addition, our system supports arbitrary datatypes for which no CUDA equivalent is currently provided. We evaluate our system across a range of data layouts and demonstrate an improvement of up to 700% end-to-end latency for performing large, noncontiguous vector data communication. Finally, we evaluate the impact of resource contention for GPU cores and access to the PCIe bus.

Through this work, we address three key challenges to enable the efficient processing of noncontiguous MPI datatypes in GPU memory:

1. *Datatype Representation in GPU Memory*: GPUs are optimized for a high ratio of FLOPS to memory operations, and high memory throughput can only be achieved through highly regular access patterns that load contiguous chunks of memory in bulk and memory operations that take advantage of a small user-controlled cache space. Thus, we develop a GPU-optimized serialized datatype representation for arbitrary datatypes, separated into a cacheable, constant-length parameter space, and a variable-length parameter space residing in GPU memory, as a first step towards building an efficient packing algorithm.

2. *Parallel Memory-bound Datatype Traversal*: For packing operations, it is ideal for the computational load to be minimized, spending the majority of time fetching and writing the elements specified by the type. Furthermore, an efficient datatype traversal must take advantage of GPU hardware characteristics, such as a fine degree of parallelism, and high memory bandwidth at the cost of high latency and less cache space. We identify a *fine-grain*, *dependency-free* parallel packing strategy based on canonical datum identification and a traversal algorithm based on the packing strategy and datatype representation.

3. *Packing in the Presence of Resource Contention*: The scheduling policy of GPU kernels and PCIe activity prevents meaningful resource sharing; thus, a packing operation could starve in the presence of another compute-or-bus intensive kernel. Different communication patterns may necessitate different packing strategies. We identify algorithm patterns for which the packing operation interferes with application performance and provide experimentation showing their effects.

This paper is organized as follows. In Section 2 we provide an overview of MPI datatypes and their optimized processing in CPU memory, as well as necessary concepts in efficient GPU algorithm design. Section 3.1 discusses the optimization of the datatype representation, while Section 3.2 discusses the packing algorithm, given the GPU datatype representation. The evaluation of GPU datatype processing is given in Section 4, focusing on overall performance against optimized manual implementations and CUDA

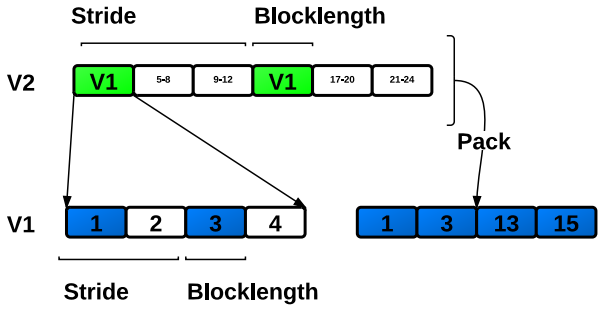

Figure 1: An example MPI `vector` of `vectors`, and the subsequent packed form.

alternatives in Section 4.2, relative performance of different components of the packing kernel and related operations in Section 4.2.1, and the effect of different packing methodologies on point-to-point communication in Section 4.3. Finally, we evaluate the effects of resource contention on packing kernels and PCIe operations in Section 4.4.

2. BACKGROUND

2.1 MPI Datatypes Specification

The Message Passing Interface (MPI) Standard [9] specifies the definition of *datatypes*, allowing users to portably communicate noncontiguous data between processes with minimal effort, increasing productivity while efficiently utilizing network resources. Perhaps the most powerful aspect of the datatypes specification is that datatypes can be layered on top of other datatypes to create complex selections of data, all within a simple and concise API. For instance, the vector of vectors shown in Figure 1, with two different gaps between elements, can be defined and communicated in a simple manner within MPI, as shown in Figure 2. *Primitive* datatypes, such as integer and floating-point variables, form the basis for *derived* datatypes, such as MPI `vectors`, which can be defined in terms of either primitive or other derived types.

The datatype encodings provided through MPI are driven primarily by the data layout of the application. For example, simulations utilizing arrays commonly use the `vector` or `subarray` types to define column vectors or subvolumes. This allows users to define subsets of their data (array slices, for example) to communicate to other processes, rather than manually selecting and preparing the data for communication. The most common datatypes used include a *strided* `vector` of *blocks*, a `subarray` defining an n-dimensional subvolume, a location-`indexed` set of blocks, and a location-indexed `struct` consisting of blocks of arbitrary datatypes. A block refers to a contiguous chunk of datatypes, and the *blocklength* refers to the number of "child" datatypes that a block contains.

The definition of datatypes is used within MPI applications to provide a simple interface for the communication of noncontiguous data either to/from I/O or across a network to other compute elements. However, initiating an I/O operation or a network transfer for each individual piece of data is expensive and poorly utilizes resources, where large, contiguous transfers are preferred. To address this challenge, MPI implementations *pack* the data into contiguous buffers prior to performing the network or I/O operation. Performing this efficiently requires optimizing a number of components.

For noncontiguous datatype processing to be useful in large-scale applications, a simple and efficient *datatype representation*

```
// Specify layout of sender's buffer, a vector of vectors,
//       with base type of double.
// The vector function signature is:
// MPI_Type_vector(count, blocklength, stride,
//       old_type, new_type).
MPI_Datatype v1, v2;
MPI_Type_vector(4, 1, 2, MPI_DOUBLE, &v1);
MPI_Type_vector(6, 1, 4, &v1, &v2);
// commit the type description
MPI_Type_commit(&v2);
// perform communication, using intermediate packing
MPI_Send(buffer, 1, v2, ...);
```

Figure 2: Defining and communicating the vector-of-vectors in Figure 1.

must be provided that allows for fast *traversal*, that is, iterating through the datatype, computing offsets in the input buffer of the encoded datatypes. The specification of datatypes, while formally described as a list of type, displacement pairs, can be encoded using a natural tree structure, where each node in the tree represents a datatype, built on top of child datatype nodes. This structure, as well as necessary parent-child relationships, is captured in the MPICH implementation of *dataloops* [12], which records type-specific parameters and propagates information about datatypes necessary for a simple traversal. Specifically, the information needed for a traversal is the *extent* and *size* of the child datatype, where the extent is the distance between successive child data types and the size is the amount of data encoded by the type, if stored contiguously.

Given an encoding of a noncontiguous datatype, the traversal of elements defined by it must be efficient. MPICH accomplishes this by representing the traversal as a buffer-filling operation. MPICH unrolls a simple depth-first search on the tree structure and uses a concise stack-based representation of the traversal. Each stack element represents type-specific parameters, such as how many `vector` blocks have been traversed. The extent and size at each level of the tree are used to compute offsets from the raw data into the contiguous buffer, and type-specific optimizations can be utilized to prevent revisiting nodes more than necessary, such as substituting specialized memory copy functions for `vector` types.

2.2 GPU Architecture and Programming Model

GPUs have become an important hardware component in HPC systems due to their optimization of massively parallel computing. In particular, Nvidia's Compute Unified Device Architecture (CUDA) defines a programming abstraction suitable for general purpose computation on GPUs (GPGPUs) [11].

CUDA presents the GPU as a CPU-driven co-processor, where the CPU issues parallel *kernels* on the GPU. Kernels and memory copies between CPU memory and separate GPU memory are performed across the PCIe bus, a high-latency, high-bandwidth operation, and DMA enables both kernel calls and memory operations to be performed asynchronously.

GPUs have multiple streaming multiprocessors (SMs), each consisting of a multiple scalar processors (SPs), giving hundreds of total available cores for computation at a given time. The threading model provided is *single instruction multiple thread*, or SIMT, which executes a group of threads (a *warp*, typically 32) in lockstep. SIMT, unlike SIMD, allows threads to *diverge* on branch instructions, where each branch is executed serially until the convergence point is reached. Thread assignment is mapped as three-dimensional grids, or *thread blocks*, where each block is scheduled on an SM up to the amount of available resources (such as statically allocated register files). The main memory in GPUs are optimized for parallel access in large chunks (typically 128B) that are *coalesced* by adjacent threads in a warp; if adjacent threads access adjacent memory, the operations are combined into a single memory transaction. While the main memory is a high-latency, high-bandwidth resource with a small L2 cache, each multiprocessor also contains a fast but small user-controlled scratch cache, called *shared memory*. Of particular importance is that GPU threads are extremely lightweight and vastly less powerful than CPU threads, but make up for it in sheer parallelism potential and extremely low context switch overhead.

Given these components, there are a number of optimization goals when devising GPU algorithms. First, PCIe bus activity should be minimized, due to high latency and transfer rates that pale in comparison to GPU hardware specifications. Second, memory access patterns on the GPU should be regular and exhibit locality with respect to threads. Third, the shared memory space should be used as much as possible to avoid multiple main memory accesses. Additionally, an algorithm must exhibit fine grain parallelism so that the hardware can utilize context switching to hide main memory access latency and stalls in the instruction pipeline.

3. IN-GPU DATATYPE PROCESSING

3.1 Representing MPI Datatypes for Efficient GPU Processing

An efficient implementation of packing on the GPU would store the type representation contiguously, loading into shared memory once upon kernel invocation. However, many datatypes have a variable-length encoding, such as the `indexed` and `struct` types. For these types, we cannot assume that the available shared memory is sufficient to store the full type representation. If we leave aside these variable-length datatype fields (such as blocklengths and displacements for the `indexed` type), then we can assume that the remaining type tree can be stored in shared memory, as each type otherwise requires a small amount of fixed-length memory to encode. The MPICH implementation, for instance, limits the depth of type trees to be 16, the fixed parameters of which can be assumed small enough to hold in shared memory. Extreme cases, such as wide trees of derived `structs`, are, to our knowledge, unseen in applications that use MPI.

Thus, to make the design friendly to both GPU access patterns and the subsequent packing algorithm, our proposed type tree representation is separated into fixed and variable length parameter spaces, each serialized in an inorder fashion. Such a separation creates *a cachable, constant-length parameter space* and *a variable-length parameter space*, both residing in GPU memory. See Table 1 for a non-exhaustive listing of datatypes with their fixed and variable length parameters.

Figure 3 shows an example type tree of arbitrary types. As illustrated in the figure, the type tree is traversed inorder, storing the fixed-length parameters (listed in Table 1) contiguously. For The variable-length parameters for types such as `indexed` and `subarray` are stored in a contiguous buffer, called the *lookaside buffer*, separate from the fixed-length parameters. For each datatype with a variable-length parameter, a pointer is added into the type's fixed-length parameters into the buffer. To control traversal and remove the explicit encoding of primitives, a bitfield is used for each type to specify the node datatype, whether or not it is a leaf

Table 1: MPI datatypes and their fixed/variable length parameters. The "Common" row contains parameters common to all datatypes in our implementation. The lookaside offset is added to point to the variable type parameters upon serialization.

Type	Fixed	Variable
Common	count size extent child elements	
`contiguous`		
`vector`	stride blocklength	
`indexed`	lookaside offset	blocklengths displacements
`blockindexed`	blocklength lookaside offset	displacements
`struct`	lookaside offset	blocklengths displacements child types
`subarray`	dimension lookaside offset	array sizes subarray sizes start offsets

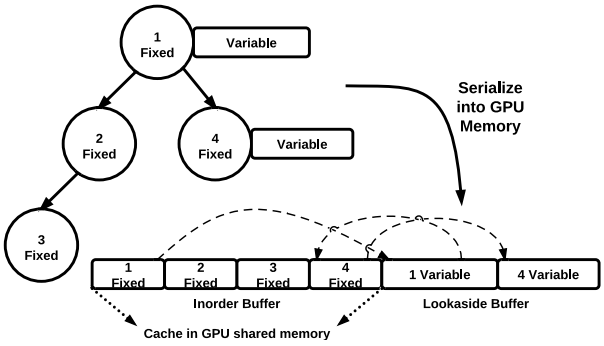

Figure 3: An example type tree in CPU memory, separated and serialized inorder into GPU memory by its fixed-and-variable-length parameters. Branches in trees only appear for `struct` types.

(derived) datatype, and if so, then the type of primitive it encodes (e.g., integer, floating-point). This is added into the fixed-length parameters of each datatype.

Since the type tree is serialized inorder, a top-down traversal to a single datum requires no additional linkage information for all but `struct` types, as the "leftmost" child types are adjacent in memory to the parent types. When there are `struct` types with multiple derived datatype children, however, additional pointers are required in the struct variable-length parameters to differentiate where in memory the children types are.

For most derived datatypes, the encoding is quite simple. For instance, the encoding for `v2` is merely the fixed parameters in rows Common and `vector` in Table 1, followed by the same parameters encoding `v1`, and requires on the order of bytes of storage. Equally simple but different from an implementation point of view, a single `indexed` type has a similarly small fixed-length storage size, followed by a potentially large list of blocklengths and displacements, requiring storage in GPU main memory.

3.2 Parallel Memory-bound Datatype Traversal

There are two technical challenges in allowing an efficient datatype traversal on the GPU and discouraging a straightforward "port" of current implementations. First and foremost, CPU-based packing implementations are based on filling buffers, leaving a possibility for the coarse-grain parallelism of filling multiple buffers. This runs contrary to best practices on the GPU, where a finer grain of parallelism is critical to performance. Second, the traversal is highly memory-bound, as small type trees can encode large amounts of data, which transforms the problem into essentially a large, non-contiguous memory copy. Section 3.2.1 addresses the mismatch in parallel packing strategies, while Section 3.2.2 discusses the algorithm itself, which attempts to minimize the memory costs of processing the datatype representation.

3.2.1 Parallelism via Point-Based Retrieval

Current MPI implementations are focused on efficiently traversing the datatype by size, that is, filling up a fixed-size buffer to be sent over the network. When considering a parallelization, such as for data residing on the GPU, the current implementation allows for parallelization in terms of size. Given a number of buffers and a buffer size, each buffer can be filled in parallel using the information encoded in the dataloops implementation. However, this will certainly not map well to the GPU, as efficient GPU computation necessitates a finer degree of parallelism.

The key insight to enable this finer degree of parallelism is that we can produce a *dependency-free* parallel traversal, given information encoded in the datatypes and minimal additional knowledge about child datatypes, making the representation a more promising candidate for use on the GPU. In addition to caching the size and extent of child datatypes, the *number of elements* can be similarly cached, allowing for fine-grained parallelism on a per-element level. By element, we refer to a single primitive datatype (e.g., integer, floating-point). When defining types (and thus building the type tree), the number of primitives encoded by a type gets propagated upwards, so that the parent type (e.g., `v2`) records the number of primitives in each instance of the child datatype (e.g., `v1`). Without this encoding, the type representation can only define the location of an element with respect to all previous points in the type, which is undesirable for the parallelism we are considering.

Using this encoding, we can base the traversal solely on which element to fetch, requiring no additional information besides the type representation. This is shown in Algorithm 1. One particular aspect to note is that adjacent threads are assigned adjacent elements, so if there is locality with respect to adjacent elements, then memory operations on them can be coalesced implicitly, so that no extra performance memory-wise is missed. Furthermore, on the most common MPI datatypes (`vector`, `subarray`, `blockindexed`), threads experience no branch divergence due to a single code path.

3.2.2 GPU Datatype Traversal Algorithm

The design of the type representation on the GPU described in Section 3.1, combined with the parallel traversal strategy in the previous section, yields a straightforward traversal for packing non-contiguous GPU data with minimal additional memory overhead. The traversal algorithm assigns a single primitive datum to a thread based on the number of primitives to be packed, traverses the type tree in a top-down fashion, updating read and write offsets, until a "leaf" derived datatype is encountered and the primitive is read from and written to the correct location. Algorithm 1 shows the general process. In Line 13, we can change packing to unpacking

by merely switching the direction of the read/write. In Line 16, pointer-jumping is only necessary for `struct` types with multiple derived children; see Section 3.1.

Algorithm 1: Point-based traversal and packing of arbitrary datatype. Refer to Table 1 for fields of the variable `type`.

```
   input  : user_buffer: buffer to pack
   input  : type: serialized datatype, pointing to root type
   input  : ID: primitive to read/write, in canonical order
   output : pack_buffer: packed buffer
 1 // in, out: location in user/packed buffer, respectively
 2 in ← 0
 3 out ← 0
 4 Load type fixed-length parameters into cache
 5 while true do
 6     // increment buffer offsets
 7     in ← in + inc_read(ID, type)
 8     out ← out + inc_write(ID, type)
 9     // compute element ID w.r.t. child type
10     ID ← ID % type.elements
11     // perform r/w if finished processing derived datatypes
12     if type is leaf then
13         pack_buffer [out] ← user_buffer [in]
14         break
15     else
16         type ← type.child
```

The functions `inc_read` and `inc_write` are type-dependent. Thankfully, they are simple to compute for the `contiguous`, `vector`, `subarray`, and `blockindexed` types, as they have a very regular structure. All but the `subarray` type are $O(1)$, and the subarray type is $O(d)$, where d is the number of dimensions. The `inc_read` and `inc_write` functions for the `vector` type computation are shown together in Algorithm 2. The general strategy is to compute the block that the element resides in, update the offsets appropriately, then "recurse" on the child type.

Algorithm 2: Read/write offset computation for the `vector` type. Refer to the Common and `vector` rows of Table 1 for the fields stored in a vector type.

```
   input  : type: vector datatype
   input  : ID: primitive to read/write, in canonical order
   output: in, out: read/write offset increments
 1 // offset w.r.t. child datatypes
 2 count_offset ← ID / type.elements
 3 // offset w.r.t. vector blocks
 4 block_offset ← count_offset / type.blocklength
 5 // for each block, advance by stride bytes
 6 // for each child datatype in block, advance by child extent
 7 in ← block_offset * type.stride + type.extent *
   (count_offset % type.blocklength)
 8 // for each child datatype, advance by child size
 9 out ← count_offset * type.size
10 return in, out
```

For the vector-of-vectors type `v2` in Figures 1 and 2, Procedure 3 shows the execution trace of a single thread traversing to its corresponding primitive. One thread is launched for each of the four primitives in the datatype. Note that the execution trace for this type is the same across all threads launched.

For the datatypes with variable-length parameters, such as `indexed`, the process is more nuanced. To avoid performing a linear scan of the type for potentially thousands of threads over potentially thousands of blocklengths, preprocessing is performed to allow a logarithmic-time binary search. Essentially, a prefix-sum is performed on the indexed type's list of blocklengths as a preprocessing step. Then, given a count of n and a list of prefix-summed blocklengths $b_0, b_1, \ldots, b_n$, the terminating condition for thread (element) i in the binary search is

$$b_h \leq i/e < b_{h+1} \tag{1}$$

where $0 \leq h < n$ and e are the number of elements in the child datatype. The additional b_n term is needed to check the condition at $h = n - 1$.

Trace 3: Execution trace of vector-of-vectors traversal for a single thread.

```
   input  : user_buffer: buffer to pack
   input  : ID: thread/datum ID
   output : pack_buffer: packed buffer
 1 in ← out ← 0
 2 Coordinated load of v2, v1 into shared memory
 3 type ← v2
 4 Increment in, out using Alg. 2, with ID, type
 5 ID ← ID % type.elements
 6 Is type a leaf type? (no)
 7 Increment type pointer by sizeof (vector type) (type ← v1)
 8 Increment in, out using Alg. 2, with ID, type
 9 ID ← ID % type.elements
10 Is type a leaf type? (yes)
11 pack_buffer [out] ← user_buffer [in]
```

There is one optimization in particular that we may perform to dramatically improve the packing operation. This optimization makes use of the fact that all writes are performed into a contiguous buffer and are thus highly coalesced by adjacent threads. Given this insight, we enable *zero-copy* memory transactions on the GPU. Essentially, instead of packing the data into GPU main memory, then performing a bulk copy on the packed buffer, recent GPUs can utilize *memory-mapping* of CPU memory into the GPU's memory space. Then, the streaming multiprocessors can write directly across the PCIe bus into CPU main memory. Since threads write exactly once and at the end of their traversal, memory mapping is a perfect opportunity to obtain additional performance for minimal effort, by avoiding the GPU main memory and implicitly pipelining the computational and PCIe loads.

There are a number of other small optimizations we can make to further increase the efficiency of the algorithm, but only under certain circumstances. We can increase data reuse by assigning multiple points to each thread, essentially looping over the traversal starting at Line 5 of Algorithm 1. However, this is useful only in circumstances with low available resources or a very large number of points to pack, as otherwise the GPU multiprocessors will be undersaturated. Also, for simple types, detecting common types and issuing custom type-specific kernels can be used that reduce the computational cost at the cost of losing generality of the packing kernel. We provide a few such type-specific kernels in Section 4.2 for comparative purposes.

3.3 Packing in the Presence of Resource Contention

In the previous sections, the methodology for packing was discussed with an underlying assumption of availability of resources and without consideration of other scenarios where packing could actually be detrimental to overall performance. For instance, what if a user initiates a send for data residing on the GPU while a fully occupant kernel is running? In the worst case, the scheduling policy of current GPUs, which schedules blocks to run to completion and only allows for architectural reasons a single kernel to be run on each multiprocessor, can easily lead to starvation of a packing kernel. This, in turn, leads to unacceptably high sending latency.

There are a number of communication patterns which could introduce resource contention, all centered around concurrent sending with other operations; global synchronizations for communication, such as in stencil codes, will not run into resource contention. Contention can occur on the computational level, when the communication is performed asynchronously to enable computational overlap. Thus, the packing operation would coincide with that computation. Furthermore, transfers could be occuring while a communication operation is being performed, such as in CPU-moderated algorithms that follow an iterative setup-compute-collect model. Thus, both PCIe directions could coincide with the packing operation and transfer of results. Of course, a combination of these can also be performed, such as when multiple users or MPI processes are accessing the same underlying hardware.

Under resource contention, the best case occurs when we are working with simple types that directly translate into CUDA calls, such as a `contiguous`, `vector`, or two-or-three-dimensional `subarray`. These can be translated into a single memory copy call, and packing/multiprocessor usage can be avoided altogether. However, this solution is only feasible for very specific datatypes, and cannot be counted on for a generic implementation.

When the datatype is nontrivial and there is resource contention preventing a packing kernel from being run, there are a number of methods that can be used to get the data onto the CPU. The two simplest ones are transferring by extent and transferring point-by-point. Both of these, of course, are highly inefficient. Transferring the entire extent of a datatype, except in cases where the extent and size are approximately equal, wastes bandwidth and still requires packing on the CPU end. Transferring point-by-point suffers from the high latency of initiating copies from the CPU. Both have the potential for interfering with user kernels that rely on host-device transfers. A more intelligent method would involve a hybrid of the two, transferring sections with a low extent-to-size ratio in bulk and transferring point-by-point otherwise. However, such a method would need more complex processing and memory management on the CPU-side and would still have the problems of both methods, albeit reduced in severity. For some types, CUDA and OpenCL allow for the transfer of regularly-strided two-and-three-dimensional subarrays. While this is very useful for the common case of array processing on the GPU, it is nevertheless a special case which be relied on for many applications. A more exotic option is to devote a *persistent kernel* for use by MPI operations and utilize signaling and polling to initiate packing, similar to Stuart *et al.*'s implementation of message-passing on the GPU [14]. However, since we show latency costs to be extremely important when performing the packing operation and their method produced an increase in these costs, we do not consider this approach (see Sections 4.2 and 4.2.1).

Unfortunately, there is currently no way within the CUDA or OpenCL interfaces to query the level of resource utilization on the GPU, complicating the act of choosing a globally efficient strategy (kernel versus memory-copies) without having application-specific knowledge. Since the overarching goal of this research is to provide transparent GPU data management from within MPI, solutions such as hijacking user kernel calls to collect statistics and infer utilization are, while interesting problems, not addressed by this paper.

4. EXPERIMENTAL EVALUATION

We evaluate our datatypes processing methodology using microbenchmarks of packing performance on numerous MPI datatypes, comparing against most-capable CUDA alternatives as well as optimized type-specific packing kernels. We additonally evaluate different components of our packing algorithm, including PCIe and memory operations to and from the input buffer, and give the full context improvement in GPU-to-GPU communication by implementing a ping-pong test on noncontiguous data. Finally, we examine the effects of GPU resource contention on packing and memory copy operations by modifying the issuing order of packing and other operations. For all tests, we used an Nvidia C2050 GPU, connected to an AMD Opteron 6128 at 800MHz. For the communication benchmarks, we used two such nodes, connected by QDR Infiniband.

4.1 Test Datatypes

While our target applications are those that communicate noncontiguous data, it is nevertheless useful to look at the absolute best case for GPU data movement of a contiguous buffer. This would allow us to measure the overhead due to the packing operation on this base case. For this purpose, we use a `contiguous` type versus a single memory copy (`cudaMemcpy`).

To benchmark two-and-three dimensional arrays such as column vectors, we use a `vector` type, of a fixed 512-byte stride between blocks. This allows us to compare the best case of the alternative, `cudaMemcpy2D`, where performance is more sensitive to data layout. The blocklength argument allows us to control the width of each block transferred. For example, if we wanted to transfer the rightmost two columns of a two-dimensional matrix, we would set the blocklength to two doubles, or 16 bytes.

To benchmark array types outside the scope of `vector` representation, we use a four-dimensional subvolume encoded as a `subarray` type. For simplicity, we use the same dimensions of $64 \times 64 \times 64 \times 64$ and iteratively increase the size of the volume selected, which for this experiment is a four-dimensional hypercube. For comparison, we use iterative calls to CUDA's three-dimensional memory copy function, equal to the width of the fourth dimension. That is, for a 4^4 subvolume, four calls to `cudaMemcpy3D` would be made, one for each 4^3 subvolume.

To benchmark an `indexed` type, for simplicity, we use the same data format as in our test `vector` type, meaning a constant blocklength and a regular displacement pattern. Other datatypes would be used in practice and be much more efficient, but this benchmark is a reasonable indicator of `indexed` performance; varying blocklengths would cause less divergence than the uniform blocklength, and a regular displacement allows us to control coalescence in a fine-grain manner. For comparison against the `indexed` type, we transfer the data block-by-block using `cudaMemcpy`.

Finally, we use a `struct` type to test the effect of thread divergence on writing. We use a simple C-style struct consisting of an 8-byte `double`, two 4-byte `ints`, and a `character`, which amounts to 24 bytes with padding. For comparison we copy the extent of each `struct` using `cudaMemcpy`.

4.2 Noncontiguous Packing Performance

For each datatype presented in Section 4.1, we evaluate the general performance of packing from GPU memory into CPU memory, with respect to the size of the packed buffer. Figure 4 shows

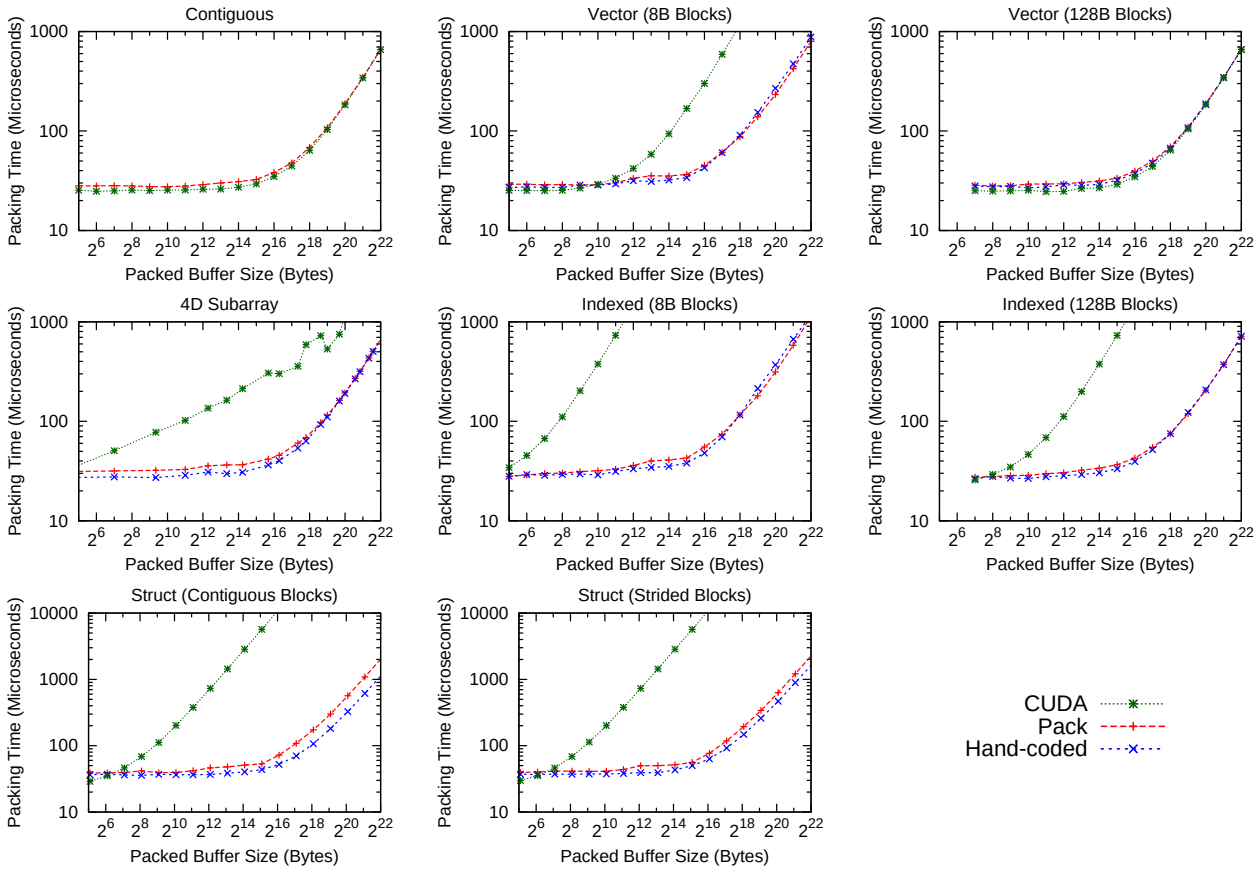

Figure 4: Packing operation time for several MPI datatypes, compared with hand-coded packing and CUDA memory copy operations.

these experiments, comparing against their respective CUDA alternatives. Furthermore, we compare against hand-coded packing routines to test the overhead of our generic packing methodology.

A number of interesting trends can be observed for the different datatypes. First, since there is a relatively large gap between the command latency and peak throughput, transfers on the lower KB level are latency-bound, and thus very small differences are seen between the CUDA API calls and the packing kernel. Given the current architecture of discrete GPUs, little can be done to improve these results, though emerging architectures that combine CPU and GPU functionality, such as AMD's Fusion [2], show promise in bridging this performance gap in the future. Furthermore, there is a difference in the latency of issuing kernels and memory operations. The launching of kernels themselves is asynchronous and return immediately, but waiting for their associated stream to indicate that they are finished is a relatively higher latency operation versus waiting on a memory copy. Thus, packing, being kernel-based, is adversely affected for smaller input sizes, performing worse than the alternative CUDA-based methods (though only on the order of microseconds).

Second, the types that do not have a CUDA-equivalent perform extremely well against CUDA, due to the bus latency in initiating each block-wise memory copy. For these types, it is clearly most beneficial to issue a packing kernel, except for data on the order of bytes. Block-wise memory copies could compete with the packing kernel only for extremely large block sizes. Extrapolating the block size to throughput ratio of the `indexed` type, it would take a block size of greater than 4 KB to match the throughput for the 128 KB type size case.

For the types that do have a CUDA-equivalent, the results are more nuanced. Here, performance is largely a function of the data layout: for two-dimensional memory copies, each block must be wide enough to saturate the bus for best performance. For single columns corresponding to a blocklength of 8 bytes, the two-dimensional memory copy performs very poorly, while the pack kernel performs approximately 20 times faster. For a larger number of contiguous columns (16 `doubles` per stride in the table), the memory copy outperforms the packing kernel in all cases, especially, for small and medium-sized inputs due to the additional kernel latency. For larger-sized inputs, both the copy and the packing kernel approach the bandwidth limit, so the relative performance difference begins to converge.

The four-dimensional `subarray` type, despite being reasonably mapped to the CUDA API, nevertheless sees major performance improvements when moving to a kernelized packing operation. Since the three-dimensional memory copies must be made iteratively to transfer the entire type, the latency is aggregated through the copies and hurts overall performance.

Against hand-coded implementations, the generic packing algorithm performs well, with little discernible difference in performance. With the exception of the `struct` type, generic packing for small packed buffer sizes sees slight overhead, while the performance of generic and hand-coded packing converges for larger packed buffer sizes. The differences in the `struct` implemen-

tations are a result of the hand-coded implementation hard-coding the location of each `struct` element, benefiting greatly from compiler optimization and greatly simplified traversal logic compared to the generalized `struct` packing algorithm.

The `vector` type is one of the more widely used MPI datatypes, and there is a significant difference in performance depending on the extent-to-size ratio, so the performance gap in the different `vectors` in Figure 4 needs to be further explored. Figure 5 fixes the size of the packed buffer and compares the completion times of the packing and the two-dimensional memory copy, varying by the blocklength (the size of contiguous chunks). Note that, for packed sizes in the lower KB range, it is preferable to use the memory copy, due to latency concerns. However, for larger sizes and a small blocklength, the packing kernel can give a 10–20 times performance improvement. Note that an intelligent MPI implementation can easily check for these cases, given minimal information about the type and hardware configuration (e.g. memory-mapped, pinned memory). Within an actual application, an example use of the `vector` type is in *halo exchange*. For multi-dimensional input matrices for a stencil computation, the border, or ghost, cells must be distributed to neighbors after each iteration. If the CUDA APIs were used for this purpose and the ghost cells had a small width, then this operation would be slow-performing, especially since GPU-based applications would experience minimal performance gain from small matrices.

For three-dimensional arrays in particular, a single `vector` type can be used to send each face of the array: the fully contiguous X-Y face, the row-wise-contiguous X-Z face, and the non-contiguous Y-Z face. Table 2 shows the transfer rate of each face for different array sizes, using the packing kernel and CUDA's two-dimensional memory copy. Here, the results largely agree with those previously presented; contiguous chunks of data are more effectively transferred using built-in CUDA copies (though there is only an approximately 10-15% difference), while packing is dramatically better for getting non-contiguous data. Note that the CUDA memory copy seems to degrade in performance for the X-Z plane transfer in the $512 \times 512 \times 512$ case. We cannot currently explain this behavior.

Table 2: Transfer of face of three dimensional matrix of double-precision values to CPU, versus cudaMemcpy2D. X-Y: fully contiguous. X-Z: z sets of x contiguous doubles. Y-Z: fully non-contiguous.

		Throughput (MB/s)	
Size	**Face**	**Pack**	**CUDA**
$64 \times 64 \times 64$	X-Y	923	**1062**
	X-Z	937	**1097**
	Y-Z	**865**	186
$128 \times 128 \times 128$	X-Y	2573	**2854**
	X-Z	2554	**2868**
	Y-Z	**2131**	209
$256 \times 256 \times 256$	X-Y	4567	**4842**
	X-Z	4553	**4845**
	Y-Z	**3728**	216
$512 \times 512 \times 512$	X-Y	5790	**5841**
	X-Z	**5792**	1645
	Y-Z	**4816**	218

4.2.1 Noncontiguous Packing Performance by Component

The performance metrics of Figure 4 give a good overview of the relative performance of the different types, but some information is still missing. For instance, what are the costs of PCIe transfers? What is the effect of memory layout on the overall performance? To answer these questions, Figure 6 shows the performance under three contexts: the full context presented in Figure 4, the completion time of packing into GPU memory (avoiding PCIe transfers), and finally datatype traversal time.

For medium and large-sized messages, the efficiency of the traversal operation is largely dependent on complexity of the type used. For instance, the `vector` and `contiguous` types, when merely traversing the type, complete quickly due to the simplicity of the traversal logic. The `subarray` type, however, suffers in performance due to the additional logic and integer computation compared to types such as `vector` necessary to represent and pack a subarray of arbitrary dimension. However, for cases such as a four-dimensional subvolume, multiple `vectors` would have to be used, which would reduce performance, so one cannot merely replace the types and get higher performance.

For types with variable-length parameters, such as `indexed` and `subarray`, the problem becomes memory-bound with respect to the input type, and thus sees lesser performance on the traversal. The indexed type, performing a binary search, still must pay the penalty of accessing GPU main memory for every point retrieved, which is a high-latency operation. Some aspects, such as coalescence between adjacent threads in the search, help reduce the severity, but the cost regardless is high enough to reduce the performance substantially. Note that the worst case for `indexed` occurs when there is a large set of approximately uniform blocklengths. Not only does this increase the size of the variable length parameter space for the type, but it also maximizes branch divergence and non-locality in the search. Similar trends are seen in the `struct` type, though to a higher degree due to an even higher reliance on the variable length parameters to perform the point retrieval; each block can be a separate datatype (see Table 1).

When the read/write stage of the packing is performed, the impact on performance is determined solely by the layout of the described non-contiguous data in memory. The best example of this is shown in the `vector` types with different blocklengths. With a small blocklength, and thus high non-contiguity, the reading of the values becomes the bottleneck of the datatype processing. With a large blocklength and thus a higher degree of contiguity, however, the reading is an efficient process due to the much higher degree of coalescence. Writing, in all cases except for the `struct`, is naturally coalesced due to writing into a contiguous buffer, so the effect of it on performance is small. The `struct` type experiences branch divergence on writing, and therefore loses much efficiency on this end compared to the other types.

Finally, adding the PCIe bus activity into the packing bottlenecks the faster packing operations, though not to a high degree due to the use of zero-copy in the packing operation. It is important to note that the `struct` type, when diverging on writing, is unable to use the bus efficiently when using zero-copy, which is highly sensitive to the read/write pattern used. Therefore, for the type, zero-copy must be disabled, otherwise performance becomes equivalent to the element-by-element memory copies presented in Figure 4. Even then, the packing and the PCIe operations are serialized and thus suffer in performance to a greater degree than the other types, in which zero-copy implicitly pipelines the PCIe and packing operations.

Because kernel latency comprises the vast majority of the running time of the packing operation for small-sized data, there is no difference between any of the types when PCIe and memory oper-

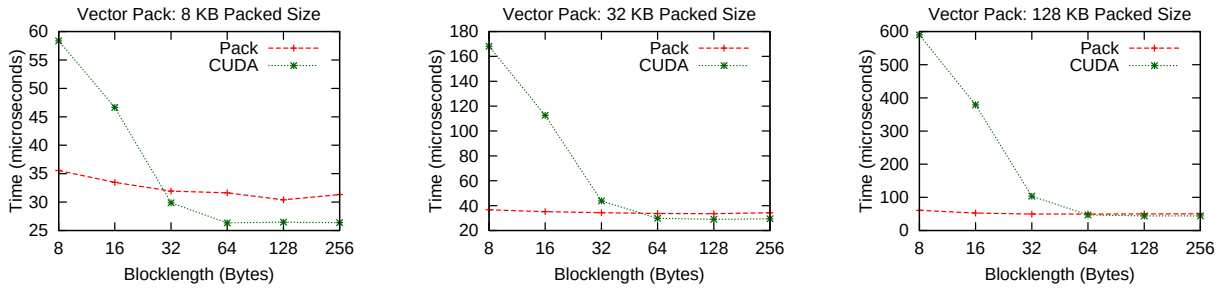

Figure 5: `vector` pack performance vs. `cudaMemcpy2D`, with varying blocklengths.

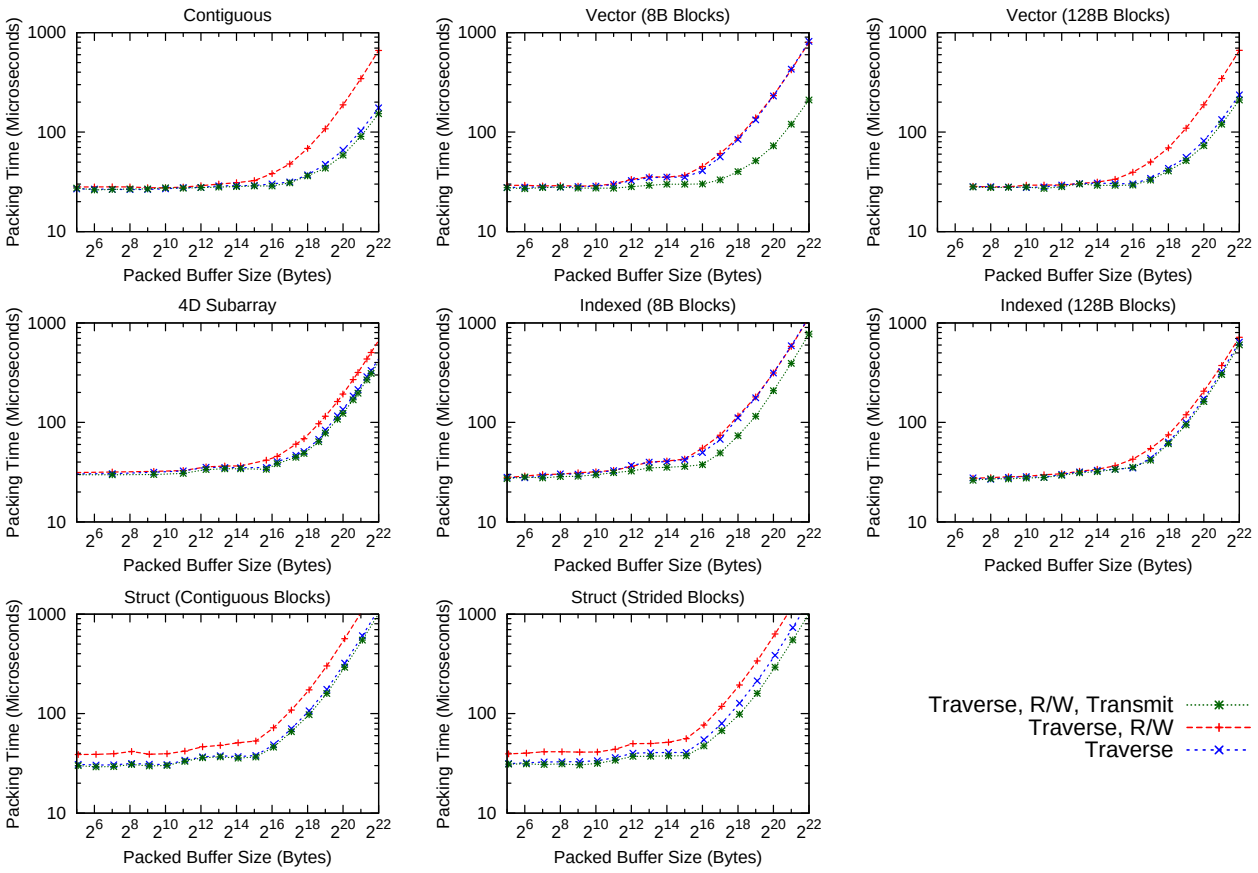

Figure 6: Packing time, by component. "Traverse" traverses the type, calculating input/output offsets. "R/W" performs the read/write operation at the end of the traversal operation. "Transmit" sends the packed data across the PCIe bus.

ations are removed.

4.3 Full Evaluation: GPU-to-GPU Communication

Now that the performance of packing on various datatypes has been studied, we consider it within the context of MPI point-to-point communication. Because of the extremely inefficient performance of CUDA-based methods on irregular data (e.g., `indexed`, `struct`), for this benchmark we only consider the packing of a `vector` type of varying blocklength; an MPI_Send where data is packed at the rate of 4 MB per second will obviously not perform well. Figure 7 shows the results of a GPU-to-GPU ping-pong test. The sender packs and sends the `vector`, while the receiver receives and unpacks the `vector`. This process is then repeated back to the original sender.

As it can be seen, the efficiency of the communication is largely dependent, as expected, on the data layout. For instance, a small blocklength, which favors the packing operation, causes a large relative performance increase versus using the two-dimensional memory copy. The larger blocklength causes the memory copy to be largely equivalent to the packing operation.

However, these results should be taken with a grain of salt. The network bandwidth for a one-way transfer that we achieved was approximately 2.0 GB/s, which is much lower than the packing throughput and memory copy for medium and large sizes. This means that the network was the bottleneck for the sending procedure for medium and large-sized messages. However, at least in our setup, the latency in sending a message was much lower, mean-

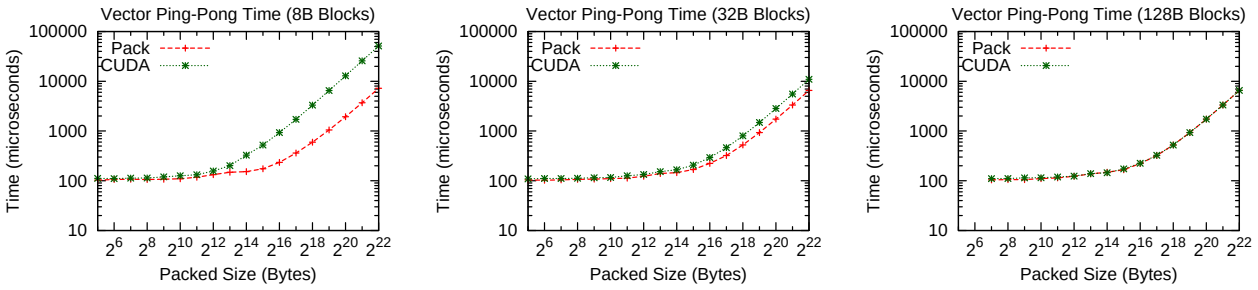

Figure 7: GPU-to-GPU ping-pong test, on the `vector` type with 8, 32, and 128 byte blocks, against `cudaMemcpy2D`.

ing that for smaller messages the GPU-to-CPU transfer was more costly. Also, the full GPU-to-GPU communication must perform packing twice, once to pack the data and again to unpack. This, in effect, adds two hops to the communication network. Therefore, on high-speed networks and for algorithms that are communication-bound, the addition of GPUs can actually hamper performance if a greater relative speedup is not achieved in the computational component, though for applications that have high-speed interconnects available, the packing operation should be able to keep up with network speeds.

4.4 Resource Contention Effects on Packing

The number of possibilities of application contexts utilizing multiple GPUs is large and growing. That said, it is important to consider the effects that issuing packing kernels would have on running applications, given the lack of "fair" scheduling present in current GPU architectures.

In Section 3.3 we identified a few algorithm patterns which could introduce resource contention on the GPU, separated into PCIe and SM (streaming multiprocessor) loads. For these types, we perform a few small representative operations that induce the particular contention scenario, then attempt to run our packing kernel, and vice versa. We call these representatives the user operations. For both directions of PCIe activity, we merely issue a memory copy. For SM contention, we utilize a simple vector add operation. The reason we do this is to tie it closely to a packing operation (using the `vector` type), where the packing time is similar to the user operation time. Furthermore, the vector add is entirely memory-bound and maximizes the amount of resource contention with respect to the memory subsystem. That is, more compute-bound operations, such as large matrix-matrix multiplications, would have a much more different run time than a corresponding packing operation and would have a larger percentage of its load within the SM, creating less resource conflicts with packing on other SMs.

For these experiments, we initialize a pair of large vectors, then perform a number of timings. First, we time each operation (the add/copy, the pack kernel, and the two-dimensional CUDA memory copy) in isolation to form our baseline. To measure contention effects on the pack/copy operation, we first initiate the user operation, then time the non-contiguous packing/transfer. To measure contention effects on the user operation, we do the same in reverse: initiate the packing kernel and then time the user operation. Regardless of the operation, we time the amount of time it takes to finish both, to see the degree of overlap occuring in the operations.

The parameter space for an experiment of this variety is enormous, so we have chosen an exemplar that is representative of the trends as a whole, and has processing times that are reasonably close for every operation. For each of the following experiments, we used a vector of total size 16 MB, and defined the `vector` datatype to have a count of 262,144, a blocklength of 8, and a stride of 64 bytes. Rather than choosing more realistic parameter sets (these cover the entire vector), we chose these values to best show the effects of resource contention due to each operation having a similar run time. In the case of GPU contention, we show that operation scheduling plays the integral role in creating contention among various operations, so we expect similar results for other datatypes and operations, though on a different scale.

Table 3 shows these exemplars. For the SM experiment, the order of initiation is critically important. When using the packing kernel, either operation, when initiated after the other, gets starved out, only starting when there are available SMs. Depending on the order of initiation, either operation can experience this degradation, though the one that is initiated first sees no penalty. The two-dimensional memory copy, avoiding the SMs entirely, does not suffer this problem, and sees no degradation in performance. In other words, the Direct Memory Access (DMA) engine handles the copy operation, leaving the GPU's SMs untouched. This means that compute-heavy applications that do double-buffering or other kinds of compute-network overlap would want to avoid performing a kernelized pack, opting instead for CUDA's memory copies.

For the PCIe experiment from GPU to CPU, both the user operation and the pack/memory copies suffer, as both must use the same lane of the bridge. However, a very interesting finding can be seen in the user-then-pack case. Since the packing operation utilizes zero-copy for all but the `struct` type (e.g. memory mapping GPU memory into CPU memory), we notice that the scheduling mechanism seems to treat the SM-issued bus transactions more favorably. Using CUDA memory copies instead of the pack does not overlap at all with the user memory copy and vice versa, since the transfers are completely serialized on the CPU end (regardless of using different CUDA streams). Therefore, if applications have this kind of algorithm pattern, if the user wants the packing to go through as quickly as possible then the packing kernel should be used or the memory copy should be issued before the user operation.

For the CPU to GPU PCIe experiment, while we would expect an insignificant degree of contention due to the operations using different PCIe lanes (PCIe is full duplex), we actually see some degradation in the time taken, though the totals for issuing both concurrently are much less than that for the completely serial case. We unfortunately cannot explain this behavior with absolute certainty, but we hypothesize it to be an artifact of the scheduler, or a small degree of contention with respect to transferring kernel parameters.

Not shown is the case when all resources are being used in some fashion. There are countless possibilities and parameterizations of this process that would lean heavily on the particular application context, though there are a few observations we can make. For

Table 3: User workloads in contention with the pack kernel and CUDA API calls, using the `vector` type, in milliseconds. The Workload column shows the order in which the operations are initiated, while the Type Proc. column shows the packing/CUDA processing time where appropriate. Section 4.4 discusses the parameters.

	SM			PCIe (CPU→GPU)			PCIe (GPU→CPU)		
Workload Order	**User**	**Type Proc.**	**Total**	**User**	**Type Proc.**	**Total**	**User**	**Type Proc.**	**Total Time**
Serialized (Pack)	1.00	2.55	3.55	3.34	2.55	5.89	2.56	2.55	5.11
Serialized (CUDA)		2.96	3.96		2.97	6.31		2.97	5.53
User→Pack	-	3.52	3.55	-	3.65	4.08	-	3.18	5.09
User→CUDA	-	3.00	3.03	-	3.66	4.06	-	5.53	5.54
Pack→User	3.53	-	3.56	4.08	-	4.11	5.08	-	5.11
CUDA→User	1.03	-	3.00	4.05	-	4.07	5.53	-	5.53

algorithm patterns that include PCIe transfers and kernels on the same CUDA stream, the scheduler is able to issue kernels immediately after PCIe transfers are issued. The same goes for memory copies issued after kernels. Therefore, there wouldn't be such a strict starvation that occurs in some of the cases in Table 3. Perhaps, in future GPU architectures, advanced schedulers would be able to enable resource sharing on a finer grain level, which would increase the fairness with respect to performance of multiple application contexts hitting on the same hardware.

5. RELATED WORK

There have been a number of efforts to integrate GPU functionality into an HPC environment, with modifications at the application, programming model, and library levels to account for a discrete GPU main memory space.

At the application level, algorithms that use both MPI and GPUs, such as Jacobsen *et al.*'s flow computation algorithm [5], are modified to allow efficient GPU computation, such as changing the problem space partitioning to benefit GPU access patterns. However, since no library support is enabled, the algorithms end up losing efficiency on the handling of memory as well as the amount of programming effort. Furthermore, MPI datatypes differ from these specialized data structures in that the datatypes efficiently encode a subset of the data structures used, for use in communication and I/O routines.

At the programming model level, Gelado *et al.* created the Asymmetric Distributed Shared Memory model (ADSM) to provide a single GPU address spaces across a cluster [4]. From the CPU standpoint, all GPU memory is in a single address space, but GPUs are only aware of their resident memory space. Their consistency model is designed for and allows operating and processing on the shared address space in contiguous chunks with memory coherence, and would have to become much more complex to enable the transfer and consistency of noncontiguous data or partial data within a contiguous buffer. Since our method is based on the message passing model with no consistency enforcements, our work does not apply here, though an interesting problem could be the combination of ADSM with noncontiguous datatypes.

Zippy [3] combines the message passing and shared-memory models (based on Global Arrays) and provides a single address space for all GPUs in the cluster, using MPI as its backend. Zippy works specifically on array-based data, as compared to our aims of supporting a generalized data representation. Since their dimensionality support extends beyond the two-and-three dimensional arrays representable by CUDA, our work is applicable to both representing an area that needs to be transferred (such as noncontiguous array boundaries) and to subsequently packaging that data efficiently.

At the library level, Distributed Computing for GPU Networks (DCGN) [14] extends MPI and partially implements the standard with a highly threaded design, utilizing signaling/polling mechanisms to allow for GPU-sourced communication. It also uses existing MPI libraries as a backend, meaning our work can directly benefit theirs. Unfortunately, given the current architectural constraints, the signaling and polling operations are cycle-consuming and lead to high latencies in GPU-sourced communication routines. This likely means that, while future architectures will allow for efficient GPU-sourced communication, the model used in today's applications will be a CPU "push-pull" model, that is, the CPU initiates the communication routine and invokes the memory management operations on the GPU.

Similarly, `cudaMPI` works on top of MPI, focusing on performance implications of different memory types, such as pinned vs. not pinned [7]. Specifically, they focus on the application of the latency/bandwidth performance model, which comes into play when doing anything GPU-related, which tends toward high-latency, high-bandwidth operations. Additionally, they briefly discuss noncontiguous memory transfer onto the CPU, but only as an application-specific column-vector transfer, and do not take into consideration MPI datatypes in general. Similar to our method, they issue a kernel to pack this data. Our work thus directly applies to their framework.

A more straightforward integration has been seen in MVAPICH2, where the authors have made their MPI implementation aware of the CUDA memory space, eliminating underlying memory copies [16]. They have provided the ability to communicate contiguous buffers, and more recently, buffers that can be represented as a single `vector` type, in communication involving buffers in GPU memory [15]. However, their methodology is based solely on existing CUDA library functions (specifically, two-dimensional memory copies) and thus cannot be extended to other datatypes; we provide a framework capable of representing and packing arbitrary datatypes built on top of each other. Our methodology can be integrated into their buffer-pool-based framework in a simple manner, however.

6. CONCLUDING REMARKS

Since GPUs are expected to continue evolving to be capable of more general purpose computations, it is extremely important to be able to integrate them into widely used libraries in the HPC community, such as MPI. We have presented one important aspect towards this end, the processing of arbitrary, non-contiguous datatypes describing data residing in GPU memory. In particular, we found that kernelizing the packing operation leads to huge performance improvements in datatypes that describe two non-exclusive data layouts: highly non-contiguous data, and irregularly located data. These cases are particularly important for future applications because there is a large degree of research into new ways of using GPU hardware to perform complex operations. With these complex operations come more complex communication patterns. Relaxing

the data layout requirements necessary for quickly getting the data from the GPU to the CPU and across nodes is helpful from an optimization standpoint: algorithms could have local access patterns that differ from global communication patterns, and if there is efficient packing available, applications could focus more towards optimizing the local patterns.

Overall, we view our method as complementary towards the goal of full, robust integration of GPU technology into the HPC community, and a strong baseline for future MPI library implementations. Obviously, it does not make sense to pack datatypes which can already be efficiently encoded and transferred using the CUDA or OpenCL libraries. Also, fully optimized implementations could do optimizations similar to what current MPI implementations do, such as substituting specialized packing operations for commonly used types. For instance, rather than going through the main memory representation for a single `vector` datatype, one could merely pass the `vector` parameters in as kernel parameters, and lower the small-message latency of the packing operation. This is analogous to writing efficient, low-level memory copying code. However, these are special case optimizations, whereas our work uses the same algorithm for every type and can handle arbitrary data layouts specified by MPI datatypes. Nevertheless, given the tree-based nature of the MPI datatypes specification, it should be fairly straightforward to detect these special cases on the fly and issue the correct optimization.

Another common optmization to make is the pipelining of data in communication codes to increase network utilization. While we did not explicitly explore this in our work, it is straighforward to provide this functionality. Given a pipeline unit of an arbitrary size, we can modify the point-to-thread mapping and the input/output offsets in Algorithm 1, signifying an *element offset* and dynamically assigning threads and thread blocks to the number of elements to read. We can also simply select the number of elements that can fit into any given buffer using the existing datatype encoding and a single-pass traversal, similar in style to Algorithm 1. However, for our test system, the pipelining would not have helped and, in fact, might have harmed the communications performance, because of the lower process-to-process bandwidth compared to the packing throughput and the additional latency costs in initializing and waiting for multiple packing operations. Nevertheless, for systems with increasingly high network capabilities, it would be important for this functionality to be available, and our design is capable of performing pipelining with little change to the underlying methods.

Furthermore, we have shown the need, through experiments on resource contention, for more complex resource scheduling on the GPU. As communication patterns get more complex and multiple threads and MPI processes access the same GPU hardware, there is nothing a user can do to prevent resource contention other than fine-tuning and organizing the code to explicitly minimize contention. Thankfully, the MPI Standard allows hints in the form of attributes to be passed to datatypes. While there may be no way to avoid resource contention, at least the user could be able to have some say in the handling of it. To enable a wider range of applications to efficiently use the GPU, providing scheduling capabilities, such as a priority-based scheduler for performance critical workloads such as packing, will become an increasingly important aspect of overall GPU adoption and use, especially, for general-purpose applications.

Acknowledgements

This work was supported in part by the U.S. Department of Energy under contract DE-AC02-06CH11357, and additionally by the National Science Foundation under Grant No. 0958311.

7. REFERENCES

[1] Top 500 supercomputing sites. http://www.top500.org.

[2] N. Brookwood. AMD fusion family of APUs: Enabling a superior, immersive PC experience. *Insight*, 64:1–8, 2010.

[3] Z. Fan, F. Qiu, and A. E. Kaufman. Zippy: A framework for computation and visualization on a GPU cluster. *Computer Graphics Forum*, 27(2):341–350, 2008.

[4] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W. mei W. Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. In *ASPLOS '10 Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, pages 347–358, 2010.

[5] 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.

[6] Khronos OpenCL Working Group. *The OpenCL Specification Version 1.1*. Khronos Group, 2011. `http://www.khronos.org/opencl/`.

[7] O. Lawlor. Message passing for GPGPU clusters: CudaMPI. In *IEEE Cluster PPAC Workshop*, pages 1–8, 2009.

[8] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka. Physis: An implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers. In *Proc. of the 2011 ACM/IEEE Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*, 2011.

[9] MPI Forum. MPI-2: Extensions to the message-passing interface. Technical report, Univ. of Tennessee, Knoxville, 1996.

[10] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 3.5-d blocking optimization for stencil computations on modern CPUs and GPUs. In *Proc. of the 2010 ACM/IEEE Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, 2010.

[11] Nvidia. Nvidia CUDA compute unified device architecture. `http://developer.nvidia.com/category/zone/cuda-zone`.

[12] R. Ross, N. Miller, and W. Gropp. Implementing fast and reusable datatype processing. In J. Dongarra, D. Laforenza, and S. Orlando, editors, *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] A. Schafer and D. Fey. High performance stencil code algorithms for GPGPUs. In *International Conference on Computational Science (ICCS)*, 2011.

[14] J. A. Stuart and J. D. Owens. Message passing on data-parallel architectures. In *Proc. of the 23rd IEEE Int's Parallel and Distributed Processing Symposium*, May 2009.

[15] H. Wang, S. Potluri, M. Luo, A. K. Singh, X. Ouyang, S. Sur, and D. K. Panda. Optimized non-contiguous MPI datatype communication for GPU clusters: Design, implementation and evaluation with MVAPICH2. In *IEEE International Conference on Cluster Computing*, Cluster '11, 2011.

[16] 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 *International Supercomputing Conference*, ISC '11, 2011.