MPI Derived Datatypes Processing on Noncontiguous GPU-resident Data

John Jenkins,* James Dinan,† Pavan Balaji,† Tom Peterka,† Nagiza F. Samatova,* Rajeev Thakur†
*Department of Computer Science, North Carolina State University
jpjenki2@ncsu.edu, samatova@csc.ncsu.edu
†Mathematics and Computer Science Division, Argonne National Laboratory
{balaji, dinan, thakur}@mcs.anl.gov

Abstract—Driven by the goals of efficient and generic communication of noncontiguous data layouts in GPU memory, for which solutions do not currently exist, we present a parallel, noncontiguous data-processing methodology through the MPI datatypes specification. Our processing algorithm utilizes a kernel on the GPU to pack arbitrary noncontiguous GPU data by enriching the datatypes encoding to expose a fine-grained, data-point level of parallelism. Additionally, the typically tree-based datatype encoding is preprocessed to enable efficient, cached access across GPU threads.

Using CUDA, we show that the computational method outperforms DMA-based alternatives for several common data layouts as well as more complex data layouts for which reasonable DMA-based processing does not exist. Our method incurs low overhead for data layouts that closely match best-case DMA usage or that can be processed by layout-specific implementations. We additionally investigate usage scenarios for data packing that incur resource contention, identifying potential pitfalls for various packing strategies. We also demonstrate the efficacy of kernel-based packing in various communication scenarios, showing multifold improvement in point-to-point communication and evaluating packing within the context of the SHOC stencil benchmark and HACC mesh analysis.

1 Introduction

Considerable interest in the HPC community has 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 because of their effectiveness over a wide range of computational loads and a favorable FLOPs-to-power ratio.

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 integrating them into algorithms with nontrivial point-to-point and collective communication patterns. The currently prevailing GPU accelerator model consists of discrete graphics processing hardware with memory separate from the CPU's RAM. Hence, any communication operation involving data resident in GPU memory requires moving data between GPU and CPU memories, effectively adding another "hop" to the communication graph. Since the MPI standard [2] does not define MPI's interaction with GPU memory managed by, for example, OpenCL [3] or CUDA [4], the burden of managing distinct memory spaces, especially of noncontiguous communication, falls on the application developers.

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. A challenging problem within this interaction is the communication of *noncontiguous* data. MPI datatypes enable such communication for data in CPU memory, allowing the programmer to define an arbitrary layout for data to be sent or received (or input/output for MPI I/O operations) using a single MPI operation. A common use of datatypes in scientific computing is the transfer of noncontiguous array slices from GPU to GPU in applications such as stencil computations, which require array boundary updates (cell exchange) between processes [5], [6], [7].

For the computational benefit of using the GPU to outweigh the cost of data transfer into CPU main memory, these communication operations must be performed with minimal overhead. The naive solutions of transferring point by point and transferring the entire noncontiguous buffer to the CPU are unacceptable from a performance point of view, suffering from unacceptably high latencies and wasted bandwidth, respectively. To achieve a sufficiently coarse transfer granularity when working with noncontiguous data, one must *pack* the data into a contiguous buffer prior to transfer. While effective packing implementations exist for noncontiguous data residing in CPU memory [8], there exists no generalized packing methodology for data residing within GPU memory that takes advantage of GPU parallelism and memory bandwidth.

In this work, we present the design of an efficient, in-GPU noncontiguous datatype processing system. We focus on NVIDIA's CUDA interface, although the techniques presented are applicable across accelerator hardware and programming models. We develop a datatype representation that exposes fine-grained parallelism, and we utilize a GPU kernel to leverage this parallelism in order to accelerate data movement. We demonstrate comparable or better noncontiguous data packaging compared with CUDA's built-in transfer routines, with low overhead compared with hand-coded packing kernels. We demonstrate up to 700% end-to-end latency improvement for performing large, noncontiguous vector data communication. In addition, our system

supports arbitrary datatypes for which, to our knowledge, no equivalent exists. We also evaluate the impact of resource contention for GPU cores and access to the PCIe bus. To realize these design goals, we identify and address three key challenges in enabling efficient processing of MPI datatypes in GPU memory:

1) *Datatype Representation in GPU Memory*: High memory utilization on GPUs requires highly regular, contiguous access patterns exploiting register memory and the small user-controlled cache. Thus, as a first step towards building an efficient packing algorithm, we develop a GPU-optimized serialized datatype representation for arbitrary MPI datatypes in GPU memory, separated into a cacheable, constant-length parameter space, and a variable-length parameter space.
2) *Parallel GPU Packing Kernel*: An efficient datatype processing algorithm must take advantage of GPU hardware characteristics, such as a fine degree of parallelism and low context-switch overhead. We identify a *fine-grained*, *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 resource sharing to the degree operating systems and CPUs allow; a packing operation could starve in the presence of another resource-intensive kernel. Different communication patterns may necessitate different packing strategies. We identify algorithm patterns for which the packing operation interferes with application performance, and we present experiments showing the 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 discusses the optimization of the datatype representation and describes the packing algorithm, given the GPU datatype representation. A detailed evaluation of GPU datatype processing is given in Section 4. In Section 6 we review related work, and in Section 7 we provide concluding remarks and discussion.

2 Background

2.1 MPI Datatypes Specification

The Message Passing Interface (MPI) standard [2] specifies the definition of *datatypes*, allowing users to portably communicate noncontiguous data between processes with minimal effort, while efficiently utilizing network resources. For instance, a noncontiguous column vector can be defined by using a `vector` type, as shown in Figure 1. In this example, the datatype `CS` has a *stride* of five elements and a *blocklength* of two elements. The stride encodes the distance between consecutive *blocks*, while the blocklength encodes the number of datatype children per block.

The most powerful aspect of the datatypes specification is support for *composition*, layering datatypes to create

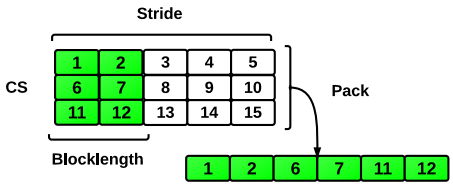


Fig. 1. Array slice with a width of two elements, an MPI vector datatype `CS` encoding it, and the slice's subsequent packed form.

```
// 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 CSvec, CS;
MPI_Type_vector(4, 1, 2, MPI_DOUBLE, &CSvec);
MPI_Type_vector(3, 2, 5, CSvec, &CS);
// commit the type description
MPI_Type_commit(&CS);
// perform communication, using intermediate packing
MPI_Send(buffer, 1, CS, ...);
```

Fig. 2. Defining and communicating a vector of vectors.

complex selections of data within a simple and concise API. For instance, the "elements" of `CS` could themselves be datatypes such as array subvolumes, and the packing operation would pack, for each "element" of `CS`, the data specified by the datatype. As another example of composition, Figure 2 shows a code snippet defining a vector of vectors, each with a unique stride, that cannot be easily defined by a single datatype. *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 by the data layout of the application. For example, simulations utilizing arrays commonly use the `vector` or `subarray` types. This allows users to define subsets of their data in order to communicate to other processes, rather than manually preparing the data for communication. The most common datatypes used include a *strided* `vector` of *blocks*, a `subarray` defining an n-dimensional subvolume, an `indexed` set of location-blocklength pairs with a homogeneous underlying datatype, and a `struct` consisting of location-blocklength-datatype tuples. 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 to/from both disk and 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. In order to address this challenge,

MPI implementations *pack* the data into contiguous buffers prior to performing the network or I/O operation.

For noncontiguous datatype processing to be useful in large-scale applications, a simple datatype representation must be provided that allows for fast *traversal* of the datatype. Datatype traversal refers to computing offsets in the input buffer for each primitive defined by the datatype. While datatypes are formally described as a list of ⟨*type*, *displacement*⟩ pairs, in practice they are encoded by using a tree structure, where each node in the tree represents a datatype. This structure, as well as necessary parent-child relationships, is captured in the MPICH implementation of *dataloops* [8], which records type-specific parameters and propagates information about datatypes necessary for a simple traversal. Specifically, the *extent* and *size* of child datatypes drive the processing algorithm, 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.

MPICH processes datatypes by unrolling a depth-first search on the tree structure, using a concise stack-based representation. Each stack element records 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 are utilized to reduce traversal overhead, such as substituting specialized memory copy functions for `vector` types.

2.2 GPU Architecture and Programming Model

NVIDIA's Compute Unified Device Architecture (CUDA) defines a programming abstraction for general-purpose computation on GPUs (GPGPUs) [4]. For this paper, we focus on CUDA and NVIDIA GPUs, although the algorithms can be easily applied to other libraries, such as OpenCL.

CUDA presents the GPU as a CPU-driven coprocessor, where the CPU issues asynchronous parallel *kernels* on the GPU. Kernel launches and memory copies between CPU memory and separate GPU memory are performed across the PCIe bus, a high-latency, high-bandwidth operation; and direct memory access (DMA) enables both kernel calls and memory operations to be performed asynchronously.

GPUs have multiple streaming multiprocessors (SMs), each consisting of 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 (single instruction, multiple data), allows threads to *diverge* on branch instructions, where each branch is executed serially until a convergence point is reached. Threads are grouped in three-dimensional grids, or *thread blocks*, where each block is statically allocated register and cache memory and scheduled on an SM. Compared with CPU threads, GPU threads are extremely lightweight and far less powerful but make up for these limitations in sheer parallelism potential and extremely low context switch overhead.

The main memory in GPUs are optimized for parallel access in large chunks (typically 128 B) 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*.

Given these components, a number of optimization goals can be defined when devising GPU algorithms. First, PCIe bus activity should be minimized, because of high latency and transfer rates that pale in comparison with 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 in order to minimize main memory accesses. Third, GPU algorithms should exhibit fine-grained parallelism so that the hardware can utilize context switching to hide main memory access latency and stalls in the instruction pipeline.

2.3 GPU-GPU Communication in MPI – MVAPICH

Recently, the MVAPICH team has utilized key developments in recent CUDA frameworks to enable the transparent MPI communication of buffers in GPU memory [9], [10]. In particular, CUDA Unified Virtual Addressing can discern whether a pointer references GPU memory, allowing MVAPICH to provide the same communication interface for both CPU and GPU buffers. Currently, MVAPICH can perform two types of communication with data in GPU memory, relying solely on existing CUDA library functions: contiguous buffers and strided buffers encodable by CUDA's two-dimensional memory copy routine (`cudaMemcpy2D`). By contrast, we provide a datatype-processing algorithm capable of representing and packing arbitrary datatypes. Our methodology can be integrated into their buffer-pool-based framework in a simple manner, however.

3 In-GPU Datatype Processing

The communication data flow driving our datatype processing is shown in Figure 3, using as an example the `CS` datatype from Figure 1. Given a datatype definition, the data is packed within GPU memory using a kernel, then transferred to CPU memory to be communicated. To optimize this flow, we organize the datatype representation to be efficiently accessed by GPU threads. Furthermore, we use a packing algorithm that fully utilizes GPU threading resources, so that each thread reads a noncontiguous element and places it into contiguous space, free of interthread dependencies. For illustrative purposes, we assume that `CS` is composed of a second `vector` type `CSvec`, shown in Figure 2. In other words, `CSvec` is a child datatype of `CS`.

3.1 MPI Datatype Encoding in GPU Memory

As opposed to the dynamic tree structure that the MPI datatypes specification would imply, GPU best practices

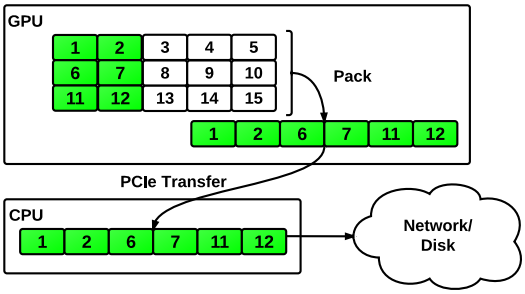


Fig. 3. Communication pattern necessitating GPU packing (unpacking if arrows are reversed).

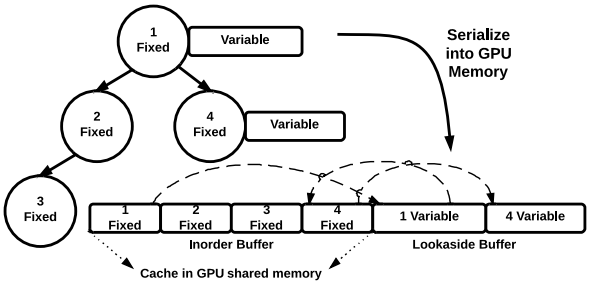


Fig. 4. Example type tree, serialized into GPU memory. Branches in trees appear only for `struct` types.

TABLE 1
MPI datatypes and their fixed- and variable-length parameters. The "Common" row contains parameters common to all datatypes in our implementation.

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

suggest storing the type representation contiguously, preferably loading into shared memory once upon kernel invocation. However, many datatypes have a variable-length encoding, such as the `indexed` and `struct` types. This presents a problem because hundreds, if not thousands, of threads may be resident on a single SM, and we cannot assume that the available shared memory is sufficient to store the full type representation for types with variable-length encoding.

Thus, we enforce a cache policy that all GPU threads can benefit from, caching only the fixed-length parameter space of the datatype(s). In order to facilitate this, the datatype representation is separated into fixed- and variable-length parameter spaces, using a serialization order corresponding to a preorder traversal of the type tree. With variable-length datatype fields (such as blocklengths and displacements for the `indexed` type) left aside, we observe 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. This separation creates a *cacheable, fixed-length parameter space* and a *variable-length parameter space*, both residing in GPU memory. See Table 1 for a listing of datatypes with their fixed- and variable-length parameters.

Figure 4 shows an example type tree of arbitrary types. The type tree is preorder-traversed, storing the fixed-length parameters contiguously. The variable-length parameters are stored in a separate contiguous buffer, called the *lookaside buffer*. For each datatype with a variable-length parameter, a pointer to the lookaside buffer is included into the type's fixed-length parameters. We call this the *lookaside offset*. In order to control traversal and remove the explicit encoding of primitives, a bitfield is used to specify the node type (leaf vs. nonleaf), encoding the primitive type if the node is a leaf (e.g., integer, floating point). This bitfield is also included in the fixed-length parameter space.

Since the type tree is preorder-serialized, a top-down traversal to a single datum requires no additional linkage information for nearly every type. The only exception is when there are `struct` types with multiple derived datatype children, requiring additional pointers in the struct variable-length parameters to differentiate where in memory the children types are.

For most derived datatypes, the encoding is simple. For example, the encoding for `CS` is the fixed parameters in rows Common and `vector` in Table 1, followed by the same parameters encoding `CSvec`. A single `indexed` type is equally simple, although different from an implementation point of view. It 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 GPU Packing Kernel

Technical challenges discourage a straightforward "port" of current CPU-based datatype processing algorithms to the GPU. In particular, CPU-side datatype processing implementations are based on serially filling fixed-size buffers from noncontiguous data in CPU memory, leaving the possibility for the coarse-grained parallelism of filling multiple buffers. This runs contrary to best practices on the GPU, where a finer grain of parallelism is critical to performance. Section 3.2.1 addresses the mismatch in parallel packing strategies, while Section 3.2.2 discusses the algorithm itself, based on the parallel processing strategy and optimized datatype representation.

3.2.1 Parallelism via Point-Based Retrieval

To enable a finer degree of parallelism than the coarse-grained method of filling multiple packing buffers, we enrich the datatypes encoding with minimal additional knowledge about child datatypes to produce a *dependency-free* parallel traversal. In addition to caching the size and extent of child datatypes, the *number of primitives* can be similarly

cached, allowing for fine-grained parallelism on a per-primitive level. When defining types (and thus building the type tree), the number of primitives encoded by a type gets propagated upward, so that the parent type (e.g., `CS`) records the number of primitives in each instance of the child datatype (e.g., `CSvec`). Using this encoding, we can base the traversal solely on which primitive to fetch, requiring no additional information besides the type representation. Without this encoding, the type representation can define the location of a primitive only with respect to all previous primitives encoded by the type, which is undesirable for the level of parallelism we are considering.

3.2.2 GPU Datatype Traversal Algorithm

The datatype representation and the parallel datatype traversal strategy on the GPU yield a straightforward packing algorithm with two favorable properties: constant per-thread storage, asides from the shared datatype representation, and no interthread dependencies.

The traversal algorithm assigns each GPU thread to a single primitive datum and traverses the type tree in a top-down fashion, using the datatype's extent, size, and number of child type primitives to update read and write offsets. After the "leaf" derived datatype is encountered, the offsets point to the locations in memory of both the element to pack and where to place it. Algorithm 1 shows the general process. Packing and unpacking can be toggled by merely switching the direction of the read/write on Line 12. On Line 17, pointer jumping is necessary only for `struct` types with multiple derived children; see Section 3.1. Note that adjacent threads are implicitly assigned adjacent primitives defined by the datatype, so locality between adjacent primitives enables coalesced memory operations on them. Furthermore, on the most common MPI datatypes (`vector`, `subarray`, `blockindexed`), threads experience no branch divergence because of a single code path.

The functions `inc_read` and `inc_write` are type-dependent. Fortunately, they are simple to compute for the `contiguous`, `vector`, `subarray`, and `blockindexed` types, as each has a very regular structure. All but the `subarray` type have an $O(1)$ complexity, and the subarray type has an $O(d)$ complexity, 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 primitive resides in, update the offsets appropriately, and then "recurse" on the child type.

For the composite types `CS` and `CSvec`, 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. In order to avoid performing a per-thread linear scan of of the blocklengths, preprocessing is performed to allow a logarithmic-time binary search. A prefix-sum is performed on the indexed type's list of blocklengths as a preprocessing step. Then,

Algorithm 1: Point-based traversal and packing of arbitrary datatype.

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

Algorithm 2: Read/write offset computation for the `vector` type.

```
input  : type: vector datatype
input  : ID: primitive to pack, in canonical order
output : in_inc, out_inc: 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 extent
7  in_inc ← block_offset * type.stride + type.extent *
   (count_offset % type.blocklength)
8  // for each child datatype, advance by child size
9  out_inc ← count_offset * type.size
10 return in_inc, out_inc
```

given a count of n and a list of prefix-summed blocklengths $b_0, b_1, \ldots, b_n$, the terminating condition for thread (primitive) 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 $h = n - 1$.

Having observed that all writes are performed into a contiguous buffer and are thus highly coalesced by adjacent GPU threads, we have applied one optimization in particular in order to dramatically improve the packing operation. Specifically, we enable *zero-copy* memory transactions on the GPU. Instead of packing the data into GPU main

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 CS, CSvec into shared memory
 3 type ← CS
 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)
 8 //   type ← CSvec
 9 Increment in, out using Alg. 2, with ID, type
10 ID ← ID % type.elements
11 Is type a leaf type? (yes)
12 pack_buffer [out] ← user_buffer [in]
```

memory and then performing a bulk copy on the packed buffer, current-generation GPUs can utilize *memory mapping* of CPU memory into the GPU's memory space. Then, the streaming multiprocessors can, in effect, 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 with minimal effort, by avoiding the GPU main memory and implicitly pipelining the computational and PCIe loads.

3.3 Packing with Resource Contention

The methodology for packing was discussed with an underlying assumption of resource availability and without consideration of 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 allows (for architectural reasons) only a single kernel to be run on each multiprocessor—can easily lead to starvation of a packing kernel. This, in turn, can lead to unacceptably high wait times.

A number of communication patterns could introduce resource contention, centered on concurrently performing communication and other operations. At the computational level, communication can be performed asynchronously in order to enable computational overlap, causing the packing operation to coincide with that computation. Furthermore, PCIe transfers can be occurring while a communication operation is being performed, such as in CPU-moderated algorithms that follow an iterative setup-compute-collect model, that clash with packed data transfer. A combination of these can also occur, such as when multiple users or MPI processes are accessing the same underlying hardware. Communication patterns utilizing global synchronization, such as in stencil codes, will not run into resource contention, however.

With resource contention, the best case occurs when we are working with types such as `vector` or two- or three-dimensional `subarray`. CUDA and OpenCL allow for the transfer of regularly strided two- and-three-dimensional subarrays, in addition to contiguous buffers, avoiding multiprocessor usage. While useful for the common case of array processing on the GPU, it is nevertheless a special case that cannot be relied on for all applications.

When the datatype is nontrivial and resource contention s preventing a packing kernel from being run, a number of methods 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 which are highly inefficient. Transferring the extent of a datatype 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. Performing some combination of the two, similar to data sieving in the ROMIO MPI-IO implementation [11], 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. Another option is to devote a *persistent kernel* for use by MPI operations and utilize signaling and polling to initiate packing, similar to Stuart and Owens's implementation of message-passing on many-core processors [12]. However, since we show latency costs to be extremely important when performing the packing operation and since their method produced an increase in these costs, we do not consider this approach (see Sections 4.2 and 4.3).

Unfortunately, no way currently exists within the CUDA or OpenCL interfaces to query the level of resource utilization on the GPU asides from high-level utilization (provided through the NVIDIA driver), complicating the selection of a globally efficient strategy without 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, not addressed by this paper.

4 Evaluation with Microbenchmarks

We evaluate our datatypes processing methodology using microbenchmarks of packing performance on numerous MPI datatypes, comparing with CUDA alternatives as well as optimized type-specific packing kernels. We break down the costs of our packing algorithm, as well as look at full context GPU-to-GPU communication through a noncontiguous ping-pong test, comparing with MVAPICH version 1.8. We also 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 North Carolina State University's ARC cluster, with nodes containing an AMD Opteron 6128 at 800 MHz and an NVIDIA C2050 GPU with version 4.1 of CUDA. Each node is connected by QDR InfiniBand. We pin CPU memory used in transfers to enable DMA, and we enable zero-copy for all datatypes but the `struct` type during packing.

4.1 Test Datatypes

To measure kernel overhead and provide an upper bound on packing performance, we perform a baseline comparison with the `contiguous` datatype, which can be satisfied with a single memory copy call (`cudaMemcpy`).

To benchmark strided arrays such as column vectors, we use a `vector` type, compared with the CUDA alternative of `cudaMemcpy2D`. We fix the stride between blocks to 512 bytes, which enables maximum performance of the CUDA operation; unaligned arrays greatly hamper CUDA's performance in this regard. Furthermore, we vary the blocklength to experiment the performance implications of block width. 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, compared with iterative calls to `cudaMemcpy3D`. We fix the containing volume to be $64 \times 64 \times 64 \times 64$ and pack/transfer a four-dimensional hypercube of increasing size.

To benchmark an `indexed` type, for simplicity, we use the same data format as in our test `vector` type. 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 would, and a regular displacement allows us to control coalescence in a fine-grained manner. For comparison, we transfer the data block by block using `cudaMemcpy`.

We additionally evaluate the `indexedblock` type (abbreviated as `idxblock` in the experiments), which is similar to the `indexed` type but has a uniform blocklength, rendering the need to perform a binary search unnecessary. For simplicity, we use the same data format as the `indexed` and `vector` types. For comparison, we transfer the data block by block using `cudaMemcpy`.

We also 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`. Furthermore, we disable the use of zero-copy for this type, as the uncoalesced write pattern induced by thread divergence leads to the issuance of a PCIe transaction for each `struct` member, causing significant performance regression.

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. Figures 5 shows these experiments compared with their respective CUDA alternatives. Furthermore, we compare with hand-coded packing routines in order to test the overhead of our generic packing methodology, shown in Figure 6.

A number of interesting trends can be observed for the different datatypes. First, since there is a relatively large gap between command latency and throughput, transfers on the lower KB level are latency-bound, and thus very small absolute 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, although combined CPU and GPU architectures, such as AMD's Fusion [13], show promise in bridging this performance gap in the future. Furthermore, the latency of issuing kernels is slightly larger than that of issuing memory copies, adversely affecting our kernelized packing for smaller inputs (though only on the order of microseconds).

Second, the packing kernel is clearly preferable for types that do not have a CUDA equivalent (e.g., `cudaMemcpy2D`), because of the latency in initiating each blockwise memory copy. Blockwise memory copies, such as for the `indexed` type, could compete with the packing kernel only for extremely large block sizes.

For the types that do have a CUDA equivalent, the results are more nuanced. Aside from latency considerations, 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 poorly, whereas the packing kernel performs approximately 20 times faster. For a larger number of contiguous columns (16 `doubles` per stride in Figure 5), the memory copy outperforms the packing kernel in all cases, especially for small and medium-sized inputs, because of 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, sees major performance improvements when moving to a kernelized packing operation. Since the three-dimensional memory copies must be made iteratively to transfer the type, the latency is aggregated through the copies and hurts overall performance.

Compared with type-specific implementations, the generic packing algorithm performs well, with little difference in performance. The differences in normalized performance between the type-specific and generic algorithms are due to the overhead of loading the type representation and instruction overhead from supporting arbitrary type representations. This overhead, however, amounts to between about two and five microseconds for most inputs. The differences in the `struct` implementations are a result of hard-coding the relative location of each `struct` primitive, benefiting from compiler optimization and greatly simplified traversal logic.

The `vector` type is one of the more widely used MPI datatypes, and performance is highly dependent on the parameterization, so the performance gap in the different `vectors` in Figure 5 needs to be further explored. Figure 7 fixes the number of blocks in the `vector` and compares the performance of the packing kernel and the two-dimensional memory copy for varying blocklengths. As seen in the figure, the performance of CUDA is highly dependent

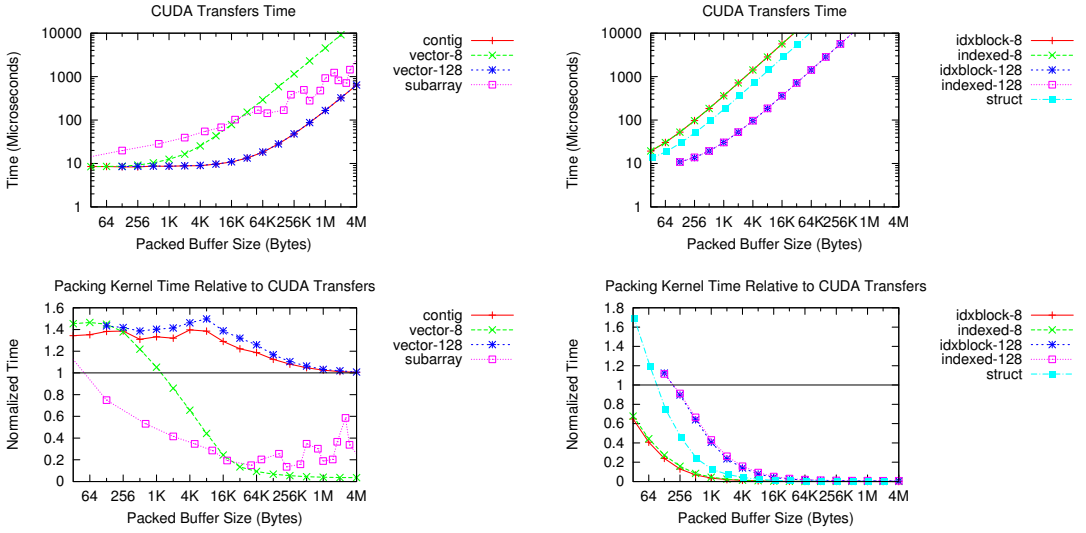


Fig. 5. Time-to-CPU packing time using the CUDA API, and corresponding relative performance of packing kernel.

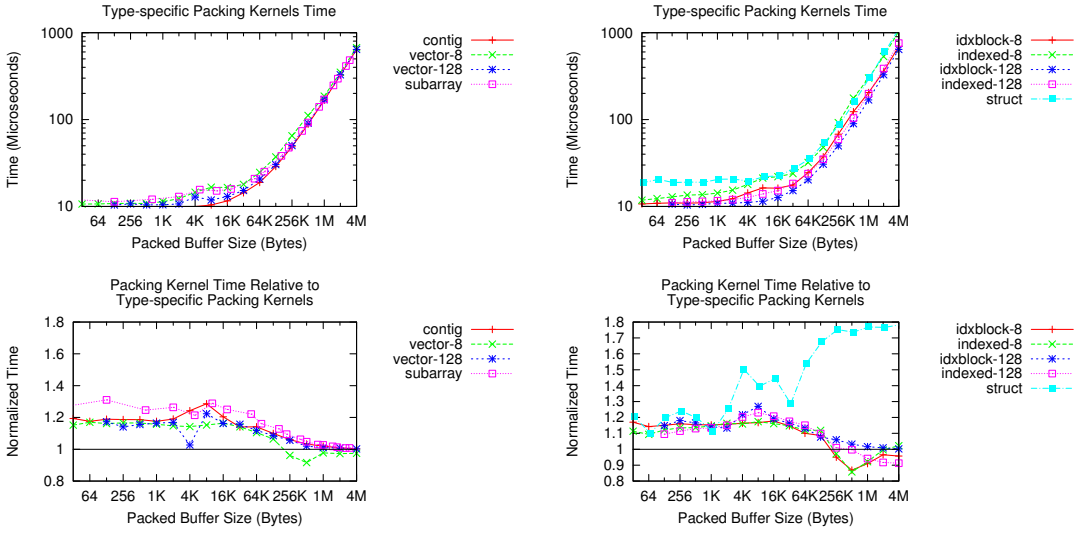


Fig. 6. Hand-coded packing kernel times and relative generalized pack performance.

on the blocklength. Blocklengths that are multiples of 32 bytes perform best, but all others experience significant performance regression. Similar performance characteristics are seen when varying the stride parameter, although this is not shown in the paper. Note that an intelligent MPI datatype processing implementation can easily check for these cases, given information about the type and hardware configuration.

For three-dimensional arrays, a single `vector` type can be used to send each face of the array: the fully contiguous X-Y face, the contiguous-per-row X-Z face, and the noncontiguous Y-Z face. Together, these operations represent the communication step of a variety of matrix algorithms, such as stencil computation. Table 2 shows the transfer rate of each face for different array sizes, using the packing kernel and CUDA's two-dimensional memory copy. The results largely agree with those previously presented; contiguous chunks of data are more effectively transferred by using built-in CUDA copies (though there is only an approximately 10–15% difference), while packing is dramatically better for getting noncontiguous 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.

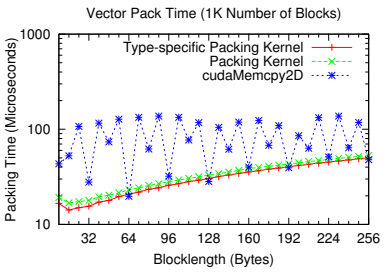


Fig. 7. `vector` pack performance vs. `cudaMemcpy2D`, with varying blocklengths.

TABLE 2
Two-dimensional plane transfer to CPU versus cudaMemcpy2D.

Size	Face	Throughput (MB/s) 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

4.3 Noncontiguous Packing Performance by Component

The performance metrics in Section 4.2 leaves out some key information about our packing methodology. 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 8 shows packing performance under three contexts: the full context as presented in Section 4.2, the completion time of packing into GPU memory (avoiding PCIe transfers), and the datatype traversal time. Note that the packing operations for small messages are latency bound, meaning the issuing of the packing kernel is the dominant cost.

For medium-sized and large-sized messages, the efficiency of the traversal operation is largely dependent on the complexity of the type used. For instance, the `vector` and `contiguous` types, when only traversing the type, complete quickly because of the simplicity of the traversal logic. The `subarray` type, however, suffers in performance due to the additional logic and integer computation compared with types such as `vector` necessary to represent and pack a subarray of arbitrary dimension. For cases such as a four-dimensional subvolume, however, 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`, the problem becomes memory-bound with respect to the input type and sees less performance on the traversal. The `indexed` type, performing a binary search, must access GPU main memory for every point retrieved, although coalescence between adjacent threads in the search helps reduce the cost. Note that the worst case for `indexed` occurs with a large set of approximately uniform blocklengths, maximizing the size of the variable-length parameter space as well as branch divergence in the search. Similar trends are seen in the `struct` type, although to a higher degree because each block is a separate datatype (see Table 1). The cost of performing the binary search for these types can be seen by comparing the `indexed` and `indexedblock` types. Specifically, the binary search implementation of `indexed` type traversal causes significant overhead, although for packed buffer sizes less than 64 KB the absolute overhead is no more than 9 microseconds.

The impact of the read/write stage of packing on performance is determined by the encoded data layout. The best example is shown in the `indexed` and `vector` types. With a small blocklength and thus high noncontiguity, reading the values is the bottleneck of the datatype processing. With a large blocklength and thus a higher degree of contiguity, the reading is an efficient process because of the much higher degree of coalescence. If the type has variable-length parameters, then the traversal is the primary cost,; but significant overhead can still be seen when packing highly noncontiguous data, such as with the `indexed` type with 8-byte blocks.

Adding the PCIe bus activity into the packing adds overhead and ultimately bottlenecks the faster packing operations for larger buffer sizes. Zero-copy keeps the overhead small for medium-sized buffers. As mentioned in Section 4.1, zero-copy is not used in the `struct` type, causing a higher relative performance degradation than seen in the other types because of the serialization of the packing and PCIe operations.

4.4 Full Evaluation: GPU-to-GPU Communication

We now assess the packing performance within the context of MPI point-to-point communication. Because of the inefficient performance of CUDA-based methods on irregular data (e.g., `indexed`, `struct`), for this benchmark we consider only 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 not perform well. Furthermore, we do not consider the use of GPUDirect for our method, a kernel patch that allows both InfiniBand and CUDA to pin the same block of memory (it is used by MVAPICH, however). This will be the focus of further research and evaluation, although the integration of it will equally benefit the packing algorithm and CUDA alternatives. Figure 9 shows the completion time of a GPU-to-GPU ping-pong benchmark. The sender packs the `vector` data from GPU memory into contiguous CPU memory, immediately followed by a send operation, while the receiver unpacks the `vector` into GPU memory. This process is then repeated back to the original sender.

The efficiency of the communication is again dependent on the data layout. A small blocklength and large buffer size, which favors the packing operation, cause a large relative performance increase compared with using the two-dimensional memory copy. A larger blocklength causes the memory copy to be largely equivalent to the packing operation. For small message sizes, GPU-to-CPU latency is the primary cost, which in this benchmark is felt four times over. Network latency, by comparison, was much lower. For medium- to large-sized messages, the measured network bandwidth of 2.0 GB/s formed the bottleneck, which is much lower than the packing and memory copy throughput.

Compared with MVAPICH, our packing methodology performs roughly equivalently for small-sized and medium-sized buffers and begins to outperform MVAPICH's vector

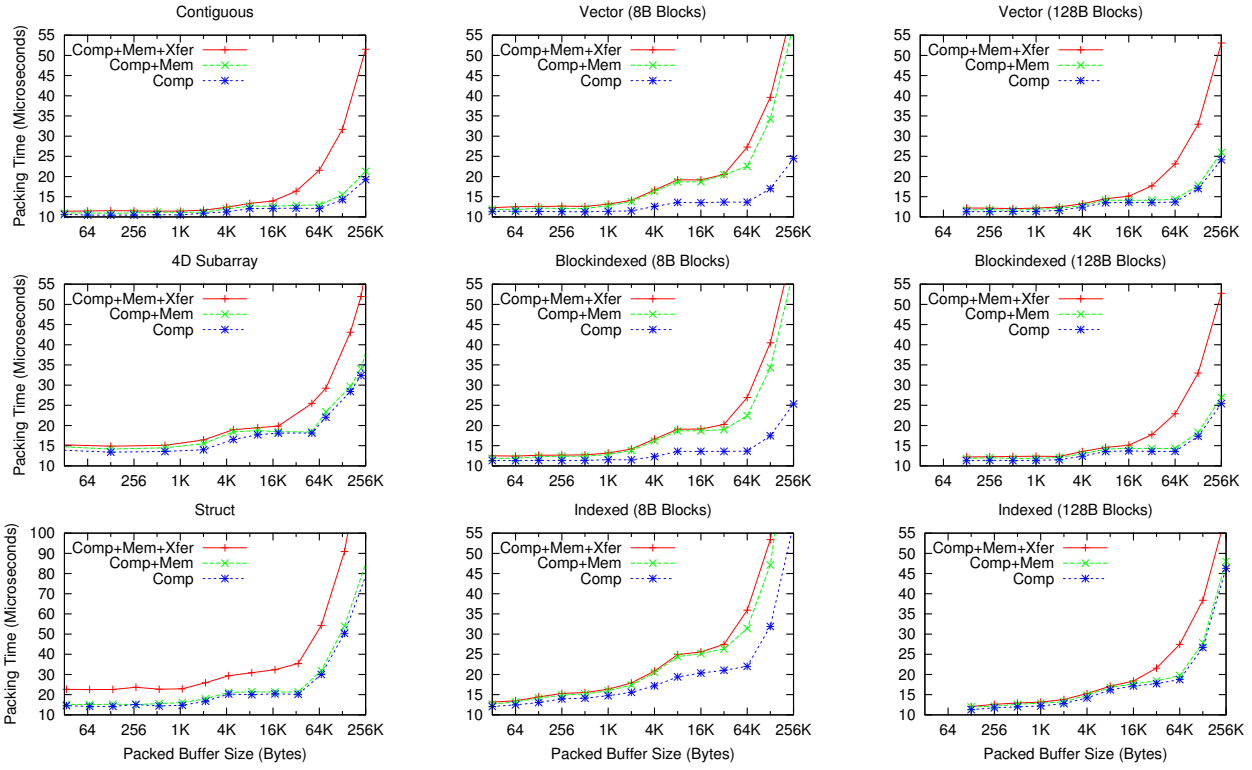


Fig. 8. Packing time, by component. “Comp”: traversing the type, computing input/output offsets. “Mem”: performing the read/write operation at the end of the traversal operation. “Xfer”: sending the packed data across the PCIe bus.

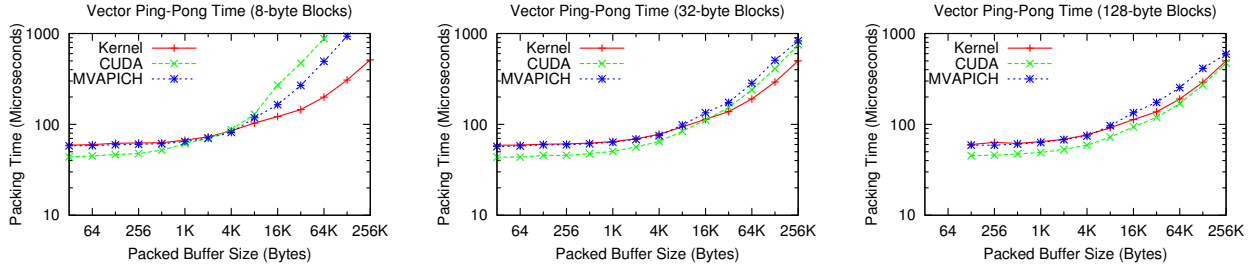


Fig. 9. GPU-to-GPU ping-pong test, on the `vector` type with 8-, 32-, and 128-byte blocks, compared with `cudaMemcpy2D`. The `vector` stride is aligned to maximize CUDA performance.

communication algorithm for large-sized buffers. MVAPICH uses a specialized communication routine for vectors, performing a two-dimensional memory copy into GPU memory and then transferring the now-contiguous data to the CPU. While this avoids poor PCIe utilization from narrow vector blocklengths as seen from two-dimensional copying directly to the CPU, the approach is more memory intensive, using two sets of memory operations. Furthermore, no overlapping of PCIe and packing activity is performed. Through our use of zero-copy, both of these problems are overcome.

4.5 Resource Contention Effects on Packing

To induce the contention scenarios discussed in Section 3.3, we use a few simple operations to stress the resource in question. We call these the application (app) operations. For both directions of PCIe activity, we merely issue a memory copy. For SM contention, we utilize a vector add operation. The reason we do so is to tie it closely to a packing operation (using the `vector` type), with packing time similar to the application operation time.

As a baseline, we time each operation in isolation. To measure contention effects on the pack/copy operation, we initiate the application operation and then initiate and time the noncontiguous pack/copy. To measure contention effects on the application operation, we initiate the pack/copy operation and then initiate and time the application operation. Regardless of the operation, we measure the amount of time it takes to finish both, in order to see the degree of overlap occurring in the operations. This methodology is based on the first-in, first-out nature of the CUDA scheduler.

The parameter space for an experiment of this variety is enormous, so we have chosen a representative exemplar

that best highlights the contention trends. 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 buffer), we chose these values so that each operation has a similar run time, in order to simplify analysis. Since the trends are based on GPU schedule operation, we expect similar results for other datatypes and operations, although on differing scales.

Table 3 shows this exemplar. For the SM experiment, the order of initiation is critical. When using the packing kernel, either operation, when initiated after the other, gets starved out, starting only when SMs are available. 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.

For the GPU-to-CPU PCIe experiment, both the application operation and the pack/memory copies suffer, since both must use the same lane of the bridge. In the app-then-pack case, however, the scheduling mechanism seems to treat the SM-issued bus transactions (through zero-copy) more favorably. Using CUDA memory copies instead of the pack does not overlap at all with the application memory copy and vice versa, since the transfers are completely serialized on the CPU end (regardless of using different CUDA streams).

For the CPU-to-GPU PCIe experiment, while we would expect an insignificant degree of contention because of the operations using different PCIe lanes (PCIe is full duplex), we actually see some degradation in the time taken, although the totals for issuing both concurrently are much less than that for the completely serial case. We 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.

More complex contention scenarios, such as mixed PCIe/SM loads and multiple users, are not shown because of the countless possibilities they entail, although we can make a few observations. For algorithm patterns that interleave PCIe transfers and kernels, the scheduler has more flexibility to insert other operations between them. Therefore, the starvation would not be as strict as that occurring in some of the cases in Table 3. Perhaps, in future GPU architectures, advanced schedulers will be able to enable resource sharing on a finer-grained level, increasing the fairness with respect to performance of multiple application contexts hitting on the same hardware.

5 Evaluation with Applications

5.1 Stencil Computation

To evaluate our packing methodology on a publicly available, commonly used application kernel, we modified the parallel, two-dimensional, nine-point stencil code from the Scalable Heterogeneous Computing (SHOC) benchmarking suite [14]. Specifically, the original halo exchange consists of up to two contiguous exchanges (with the "north" and "south" neighbors) and up to two strided exchanges (with the "east" and "west" neighbors). The GPU stencil benchmark copies all halo regions into CPU memory, performs the halo exchange, and transfers all results back to the GPU. We replace the noncontiguous GPU copying code, which relies on CUDA DMA, with our packing methodology.

TABLE 4
SHOC stencil double-precision (DP) and single-precision (SP) mean GFLOPS per node, using both CUDA DMA and kernelized packing to perform the halo exchange.

Per-Node Size	DP GFLOPS		SP GFLOPS	
	w/CUDA	w/Pack	w/CUDA	w/Pack
128x128	3.76	3.84	3.80	3.87
256x256	13.79	13.81	15.12	15.14
512x512	40.05	40.82	46.87	47.80
1024x1024	88.34	87.35	125.82	124.88
2048x2048	130.63	130.97	213.73	214.72

Table 4 shows mean stencil GFLOPS for four nodes for varying per-node problem sizes and for single- and double-precision floating-point data. As is shown, the time using a packing kernel is nearly equivalent to that using CUDA DMA. We attribute the likeness in performance to the ratio of computation to communication in the overall stencil cost as well as to the fact that half of the transfers performed are over contiguous data.

5.2 Analysis Code

The next application benchmark is taken from the analysis of cosmological simulations. The HACC [15] cosmology code is a framework for N-body particle simulations of dark matter tracer particles. Some analysis tasks such as identifying cosmological voids are enabled by the conversion of raw particle data to a Voronoi tessellation [16], which converts a point cloud to a polyhedral mesh. When executed in a spatially decomposed data-parallel manner, each MPI process computes the following data structure:

```
struct vblock_t {
  int num_verts, num_cells;
  int num_cell_verts, num_complete_cells;
  int num cell_faces, num_face_verts;
  int num_orig_particles;
  float mins[3], maxs[3];
  float *vertices, *sites;
  float *areas, *vols;
  int *cells, *face_verts;
  int *num_cell_faces, *num_face_verts;
};
```

When writing and reading results from parallel storage using MPI-IO, the above data are accessed by using a single custom MPI datatype by each MPI process. This is a packing challenge because it contains a combination of integer and floating-point scalars and vectors, together with pointers that need to be followed in order to access the actual data members. Each process contains a different

TABLE 3
User workloads in contention with the pack kernel and CUDA API calls, using the `vector` type, in milliseconds. *Type Proc.*: time between initialization of the packing/CUDA operation and it's completion.

	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

TABLE 5
HACC analysis structure packing times in milliseconds by rank. *CPU Ref.*: reference CPU packing time. *CUDA DMA*: GPU-to-CPU packing time using memory copies for each GPU buffer. *Kernel*: GPU-to-CPU kernelized packing time.

Rank	CPU Ref.	CUDA DMA	Kernel
0	0.96	0.43	0.35
1	0.40	0.25	0.16
2	1.68	0.65	0.57
3	1.53	0.61	0.53
4	0.92	0.42	0.34
5	0.26	0.19	0.11
6	0.83	0.39	0.31
7	0.55	0.29	0.20

number of particles, hence different lengths of buffers that need to be fetched. Traversing the datatype results in a set of contiguous pieces combined in a noncontiguous fashion.

To assess the performance of packing this datatype, we first logged the memory accesses of the CPU packing done by MPI for a test run of 32,768 dark matter tracer particles converted to a Voronoi mesh using eight MPI processes. Each process produced a trace that logged the base type, quantity, and starting address associated with fetching each structure member.

We then regenerated the identical memory access pattern in our benchmark and compared the performance of three versions of datatype packing. Table 5 shows those results. "CPU Ref." is the time to pack the original MPI data type using the CPU only. The "CUDA DMA" column is the time to manually pack a single buffer using a sequence of GPU-to-CPU copies, one for each `struct` field solely using `cudaMemcpy`. The "Kernel" column is our GPU packing kernel version. Our results show a 13–43% reduction in time-to-CPU by using packing, with a median reduction of 20.8%. We attribute these results to the reduced latency costs in issuing a single kernel versus multiple copies.

6 Related Work

A number of efforts have been undertaken 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 [17], are modified to allow efficient GPU computation, such as changing the problem space partitioning to benefit GPU access patterns. 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, the Asymmetric Distributed Shared Memory model (ADSM) provides a single GPU address space across a cluster, while leaving GPUs aware of only their local memory space [18]. The consistency model is designed for and allows operating and processing on the shared address space in contiguous chunks with memory coherence; it would have to become more complex in order to enable the transfer and consistency of noncontiguous data or partial data within a contiguous buffer.

Zippy [19] 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 multidimensional array-based data, so our work is applicable both to 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) [12] extends MPI and utilizes 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.

Similarly, `cudaMPI` works on top of MPI, focusing on performance implications of different memory configurations, such as pinned vs. not pinned. Specifically, Lawlor [20]. focuses on the application of the latency/bandwidth performance model, which comes into play when doing anything GPU-related that tends toward high-latency, high-bandwidth operations. Additionally, Lawlor briefly discusses noncontiguous memory transfer onto the CPU, but only as an application-specific column-vector transfer, and does not take into consideration MPI datatypes in general. Similar to our method, however, he issues a kernel to pack this data; our work thus directly applies to his framework.

7 Concluding Remarks

Since GPUs are expected to continue evolving in order to be capable of more general-purpose computations, they need to be integrated into widely used libraries in the HPC community, such as MPI. We have presented one important aspect toward this end: the processing of arbitrary, noncontiguous GPU-resident data. We show that kernelizing the packing operation leads to huge performance improvements in datatypes that describe two nonexclusive data layouts: highly noncontiguous data and irregularly located data. These cases are particularly important as GPUs continue to branch out in terms of the complexity of operations performed on them; algorithms could have local access patterns that differ from global communication patterns, and if there is efficient packing available, applications could focus more on optimizing the local patterns.

Overall, we view our method as complementing the goal of robust integration of GPU technology into high-performance data movement frameworks such as MPI, as well as a baseline for future MPI library implementations. A complete solution to GPU data movement within MPI not only would minimize internal memory copies and fully utilize current/future versions of GPUDirect but also would be able to flexibly determine the best methodology for transferring the data, especially noncontiguous data.

This determination would ideally take into account the degree of noncontiguity of the data, the availability of higher-performing type-specific kernels or CUDA alternatives, and awareness of competing operations for limited GPU resources. For example, CUDA DMA can be used in place of the generic packing algorithm for a single `vector` type (e.g., `CSvec`) by merely analyzing the strides/blocklengths for CUDA-optimized parameters.

Furthermore, while we did not explore pipelining the communication process (our benchmarks were bottlenecked by PCIe latency for small messages and network bandwidth for large messages), our packing methodology can provide such capability in future work. Given a pipeline unit of an arbitrary size, we can modify the point-to-thread mapping by simply offseting the elements to read based on the amount of pipelined data read. Given the existing datatype encoding, computing the number of elements to fit in a pipeline unit can be easily done on the host, in a style similar to Algorithm 1. This functionality is important for systems with increasingly high network capabilities, and our design is capable of performing pipelining with little change to the underlying methods.

Furthermore, through experiments on resource contention, we have shown the need, for more complex resource scheduling and management on the GPU. Currently, a user can do little to prevent resource contention, other than fine-tuning and organizing the code to explicitly minimize contention. Fortunately, the MPI Standard allows hints in the form of attributes to be passed to datatypes, which were used in the recent MPI-ACC work [21] to eliminate the overhead of using CUDA UVA. While there may be no way to avoid resource contention, at least the user can have some say in handling it. In order 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.

Acknowledgments

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.

References

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

[2] MPI Forum, "MPI-2: Extensions to the message-passing interface," Univ. of Tennessee, Knoxville, Tech. Rep., 1996.

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

[4] NVIDIA, "NVIDIA CUDA compute unified device architecture," http://developer.nvidia.com/category/zone/cuda-zone.

[5] 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.

[6] 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*, 2010, pp. 1–13.

[7] A. Schafer and D. Fey, "High performance stencil code algorithms for GPGPUs," in *International Conference on Computational Science (ICCS)*, 2011.

[8] R. Ross, N. Miller, and W. Gropp, "Implementing fast and reusable datatype processing," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, J. Dongarra, D. Laforenza, and S. Orlando, Eds., vol. 2840. Springer Berlin / Heidelberg, 2003, pp. 404–413.

[9] 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.

[10] 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.

[11] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective I/O in ROMIO," in *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation (FRONTIERS '99*. Washington, DC: IEEE Computer Society, 1999, pp. 182–189. [Online]. Available: http://dl.acm.org/citation.cfm?id=795668.796733

[12] J. A. Stuart and J. D. Owens, "Message passing on data-parallel architectures," in *Proc. of the 23rd IEEE International Parallel and Distributed Processing Symposium*, May 2009.

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

[14] 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 *Proc. of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units GPGPU '10*. New York: ACM, 2010, pp. 63–74. [Online]. Available: http://doi.acm.org/10.1145/1735688.1735702

[15] S. Habib, V. Morozov, H. Finkel, A. Pope, K. Heitmann, K. Kumaran, T. Peterka, J. Insley, D. Daniel, P. Fasel, N. Frontiere, and Z. Lukic, "The Universe at Extreme Scale: Multi-Petaflop Sky Simulation on the BG/Q," *ArXiv e-prints*, Nov. 2012.

[16] T. Peterka, J. Kwan, A. Pope, H. Finkel, K. Heitmann, S. Habib, J. Wang, and G. Zagaris, "Meshing the universe: Integrating analysis in cosmological simulations," in *Proc. of the SC12 Ultrascale Visualization Workshop*, Salt Lake City, UT, 2012.

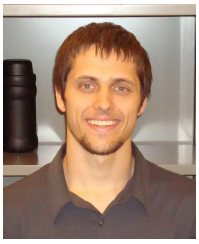
[17] D. A. Jacobsen, J. C. Thibault, and I. Senocak, "An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters," in *Proc. of the 48th AIAA Aerospace Sciences Meeting*, 2010.

[18] 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 Proc. of the fifteenth edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, 2010, pp. 347–358.

[19] Z. Fan, F. Qiu, and A. E. Kaufman, "Zippy: A framework for computation and visualization on a GPU cluster," *Computer Graphics Forum*, vol. 27, no. 2, pp. 341–350, 2008.

[20] O. Lawlor, "Message passing for GPGPU clusters: CudaMPI," in *IEEE Cluster PPAC Workshop*, 2009, pp. 1–8.

[21] 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 *14th IEEE International Conference on High Performance Computing and Communications*, Liverpool, UK, June 2012.



John Jenkins is a Ph.D. candidate at North Carolina State University. His research interests include parallel runtime data management, accelerator architecture and algorithms, and scientific data analysis. John received his B.S. in Computer Science from Lafayette College in 2010.

James Dinan is the James Wallace Givens postdoctoral fellow at Argonne National Laboratory. He received his Ph.D. and M.S. degrees in computer science from the The Ohio State University, in Columbus, Ohio. He received his B.S. degree in computer system engineering from the University of Massachusetts, Amherst. His research interests include parallel programming models, high-performance runtime systems, distributed algorithms, scientific computing applications, and computer architecture.

Pavan Balaji holds appointments as a computer scientist and group lead at the Argonne National Laboratory, as a research fellow of the Computation Institute at the University of Chicago, and as an institute fellow of the Northwestern-Argonne Institute of Science and Engineering at Northwestern University. His research interests include parallel programming models and runtime systems for communication and I/O, modern system architecture (multicore, accelerators, complex memory subsystems, high-speed networks), cloud computing systems, and job scheduling and resource management. He has nearly 100 publications in these areas and has delivered nearly 120 talks and tutorials at various conferences and research institutes. He is a recipient of the U.S. Department of Energy's Early Career Award. He has also received several other awards including the Director's Technical Achievement award at Los Alamos National Laboratory, an Outstanding Researcher award at the Ohio State University, and five best-paper awards. He serves as the worldwide chairperson for the IEEE Technical Committee on Scalable Computing (TCSC). He has also served as a chair or editor for nearly 50 journals, conferences, and workshops and as a technical program committee member in numerous conferences and workshops. He is a senior member of the IEEE and a professional member of the ACM.

Tom Peterka is an assistant computer scientist at Argonne National Laboratory, a fellow at the Computation Institute of the University of Chicago, and an adjunct assistant professor at the University of Illinois at Chicago. His interests are in large-scale parallelism for scientific visualization and analysis of scientific datasets. He has contributed to three best-paper awards and numerous publications in ACM and IEEE conference and journals. Tom earned his Ph.D. in computer science from the University of Illinois at Chicago, where he was a James Scholarship and University Fellowship winner.

Nagiza F. Samatova is an Associate Professor in Computer Science Department of North Carolina State University and a Senior Research Scientist in Computer Science and Mathematics Division of Oak Ridge National Laboratory. She received the B.S. degree in applied mathematics from Tashkent State University, Uzbekistan, in 1991 and her Ph.D. degree in mathematics from the Computing Center of Russian Academy of Sciences (CCAS), Moscow, in 1993. She also obtained an M.S. degree in Computer Science in 1998 from the University of Tennessee, Knoxville, USA. Dr. Samatova specializes in High Performance Data Analytics, Data Management, Scientific and High Performance Computing, Graph Theory and Algorithms, Bioinformatics, Systems Biology, and Machine Learning. She is the author of over 150 publications in peer-reviewed journals and conference proceedings.

Rajeev Thakur is the deputy director on the Mathematics and Computer Science Division at Argonne National Laboratory, where he is also a senior computer scientist. He is also a senior fellow in the Computation Institute at the University of Chicago and an adjunct professor in the Department of Electrical Engineering and Computer Science at Northwestern University. He received his Ph.D. in computer engineering from Syracuse University. His research interests are in the area of high-performance computing in general and particularly in parallel programming models, runtime systems, communication libraries, and scalable parallel I/O. He is a member of the MPI Forum that defines the Message Passing Interface (MPI) standard. He is also co-author of the MPICH implementation of MPI and the ROMIO implementation of MPI-IO, which have thousands of users all over the world and form the basis of commercial MPI implementations from IBM, Cray, Intel, Microsoft, and other vendors. MPICH received an R&D 100 Award in 2005. Rajeev is a co-author of the book "Using MPI-2: Advanced Features of the Message Passing Interface" published by MIT Press, which has also been translated into Japanese. He was an associate editor of IEEE Transactions on Parallel and Distributed Systems (2003-2007) and was Technical Program Chair of the SC12 conference.