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

John Jenkins,* James Dinan,† Pavan Balaji,† Nagiza F. Samatova,* Rajeev Thakur†

*Deptartment 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*—Lack of efficient and transparent interaction with GPU data in hybrid MPI+GPU environments challenges GPU acceleration of large-scale scientific computations. A particular challenge is the transfer of *noncontiguous* data to and from GPU memory. MPI implementations currently do not provide an efficient means of utilizing datatypes for noncontiguous communication of data in GPU memory.**

To address this gap, we present an MPI datatype-processing system capable of efficiently processing arbitrary datatypes directly on the GPU. We present a means for converting conventional datatype representations into a GPU-amenable format. 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. In addition, we identify and evaluate communication patterns that may cause resource contention with packing operations, providing a baseline for adaptively selecting data-processing strategies.

I. Introduction

Graphics processing units (GPUs) have generated significant interest in the HPC community, as evidenced in recent Top500 lists of supercomputers [1]. The currently prevailing GPU accelerator model consists of discrete GPU hardware with memory separate from the CPU's RAM. Hence, communications involving data resident in GPU memory require moving data between GPU and CPU memories, 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 falls on application developers.

A particularly challenging problem in enabling MPI to interact directly with data in GPU memory is the communication of *noncontiguous* data, defined through the MPI *datatypes* specification. For example, stencil computations on the GPU require noncontiguous array boundary updates (cell exchange) between processes [5], [6], [7].

To fully utilize the PCIe bus and network interconnect for noncontiguous communication, one must *pack* the data into a contiguous buffer prior to transfer, since transferring per point or by extent to the CPU suffers from unacceptably high latencies or wasted bandwidth, respectively. Effective implementations exist for packing noncontiguous data residing in CPU memory [8], but there exists no generalized packing algorithm for data residing in GPU memory that takes advantage of GPU parallelism and memory characteristics.

To address this gap in functionality, we present the design of an in-GPU noncontiguous MPI datatype processing system. We focus on Nvidia's CUDA interface, although the techniques presented are broadly applicable across accelerator hardware and programming models. We identify and address three key challenges in enabling efficient processing of noncontiguous MPI datatypes in GPU memory:

1) *Datatype Representation in GPU Memory*: To optimize for GPU memory access patterns, we serialize the tree-based MPI datatype representation in GPU memory, separating it into a cacheable, constant-length parameter space and a variable-length parameter space.
2) *Parallel GPU Packing Kernel*: To further exploit GPU hardware characteristics, we present a *fine-grained*, *dependency-free* parallel packing algorithm based on canonical datum identification.
3) *Packing in the Presence of Resource Contention*: We identify pitfalls in the GPU scheduling policy, identifying algorithm patterns for which packing operations interfere with application performance, in order to provide a baseline for adaptively selecting a packing strategy.

We demonstrate comparable or better processing of noncontiguous data when compared with CUDA's built-in transfer routines, with lower overhead than that from hand-coded packing kernels. We also show up to 700% end-to-end performance improvement for communicating large, noncontiguous vector data. Our system additionally supports arbitrary datatypes for which, to our knowledge, no equivalent exists.

This paper is organized as follows. Sections II provides background about defining and processing noncontiguous data using the MPI datatypes specification and the GPU architecture and programming model. Section III-A discusses the opti-

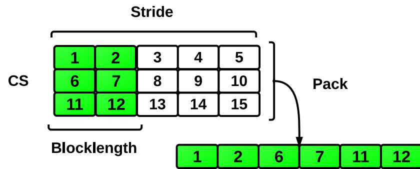

Figure 1. Array slice with a width of two elements, an MPI `vector` datatype `CS` encoding it, and the slice's subsequent packed form.

mization of the datatype representation, while Section III-B discusses the packing algorithm, given the GPU datatype representation. A detailed evaluation of GPU datatype processing is given in Section IV. In Section V we review related work, and in Section VI we provide concluding remarks.

II. Background

Datatypes in the Message Passing Interface (MPI) Standard [2] allow users to portably communicate noncontiguous data with minimal effort. For example, Figure 1 shows the application of a simple `vector` type, called `CS`, to define a column vector. `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 datatypes specification supports *composition*, layering datatypes to create complex selections of data. For example, each "element" of `CS` could itself be a noncontiguous set of data defined by a datatype, such as array subvolumes. Packing in this context would pack, for each "element" of `CS`, the data specified by its underlying 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.

MPI datatypes define data layouts of varying complexity. 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, 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.

To avoid the poor resource utilization resulting from initiating an I/O or network operation for each piece of data, MPI implementations *pack* the data into contiguous buffers prior to performing the operation. Low-overhead packing necessitates creating a simple *datatype representation* 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. Datatypes can be encoded by using a natural tree structure, where each node in the tree represents a datatype. This structure is captured in the MPICH implementation of *dataloops* [8], which records type-specific parameters and propagates information about datatypes necessary for a simple traversal. The propagated information includes the *extent* and *size* of the child datatype, where the extent is the distance between successive child datatypes and the size is the amount of data encoded by the type, if stored contiguously.

A. GPU Architecture and Programming Model

For this paper, we focus on Nvidia's Compute Unified Device Architecture (CUDA) [4], though our method 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 available cores for computation at a given time. The threading model is *single instruction, multiple thread*, or SIMT, which executes a group of threads (a *warp*, typically 32) in lockstep. Unlike SIMD (single instruction, multiple data), SIMT 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 less powerful, but they make up for these differences in sheer parallelism potential and negligible context switch overhead.

GPU main memory is optimized for parallel access in large chunks (typically 128 B) that are *coalesced* by threads in a warp; if adjacent threads access adjacent memory, the operations are combined into a single memory transaction. The main memory is a high-latency, high-bandwidth resource with a small L2 cache. Multiprocessors also contain a small and fast user-controlled scratch cache, called *shared memory*.

One should address a number of optimization goals when devising GPU algorithms. First, PCIe bus activity should be minimized, because of high latency and transfer rates that pale in comparison to GPU hardware. Second, memory access patterns on the GPU should be regular and exhibit locality with respect to threads. Third, shared memory should be used as much as possible in order to avoid multiple main memory accesses. Additionally, GPU algorithms must exhibit fine-grained parallelism so that the hardware can utilize context switching to hide main memory access latency and instruction pipeline stalls.

III. In-GPU Datatype Processing

The communication data flow driving our methodology is shown in Figure 2, using as an example the `CS` datatype from Figure 1. Given a datatype, the data in GPU memory is packed by using a kernel, then transferred to CPU memory. To fully optimize GPU resources, we reorganize the datatype representation in GPU memory and design a fine-grained parallel packing algorithm. For illustration, we assume that `CS` is composed of a second `vector` type `CSvec`. In other words, `CSvec` is a child datatype of `CS`.

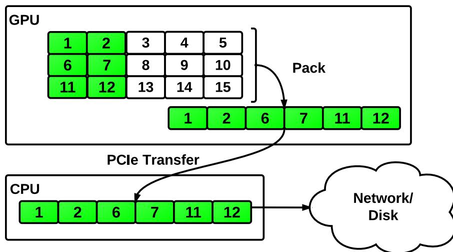

Figure 2. Communication pattern necessitating GPU packing. Reversing the arrow directions produces the pattern necessitating GPU unpacking.

Table I
PARAMETERS FOR MPI DATATYPES IN OUR DESIGN. "COMMON" CONTAINS PARAMETERS SHARED BY ALL DATATYPES.

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

A. MPI Datatype Encoding in GPU Memory

GPU best practices suggest storing the type representation contiguously instead of as a dynamic tree, loading into shared memory once upon kernel invocation. However, many datatypes have a variable-length encoding that may not fit into shared memory, such as the `indexed` type, and hundreds of threads on an SM may access this information.

Thus, we enforce a cache policy from which all GPU threads can benefit. The datatype representation is separated into fixed- and variable-length parameter spaces, by using a serialization order corresponding to a preorder traversal of the type tree. With variable-length datatype fields left aside, we observe that the small-sized fixed-length parameters of the type tree can be stored in shared memory. See Table I for a listing of datatypes with their fixed- and variable-length parameters.

Figure 3 shows an example type tree. The fixed-length parameters are stored contiguously, while the variable-length parameters are stored in a separate buffer, called the *lookaside buffer*. For each datatype with variable-length parameters, a pointer to the lookaside buffer is included into the type's fixed-length parameters, called the *lookaside offset*.

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. The

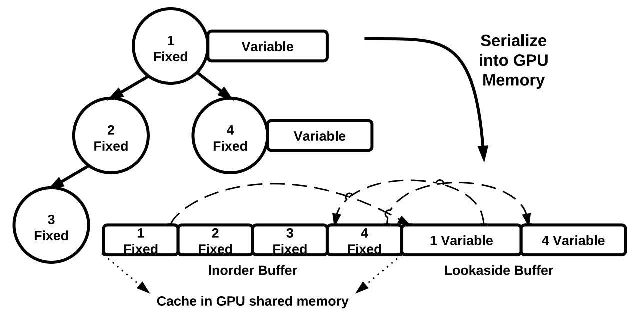

Figure 3. Example type tree in CPU memory, separated and serialized preorder into GPU memory by its fixed-and-variable-length parameters. Branches in trees only appear for `struct` types.

encoding for `CS` is just the fixed parameters in rows Common and `vector` in Table I, followed by the same parameters encoding `CSvec`. A single `indexed` type is equally simple, with similarly small fixed-length parameters followed by a potentially large list of blocklengths and displacements.

B. Parallel GPU Packing Kernel

Current CPU-based datatype processing algorithms utilize a depth-first buffer-filling strategy that maintains traversal state in a stack structure. At best, this approach exposes coarse-grained parallelism and would require storage that would not fit in shared memory for hundreds of threads. Hence, we instead expose a fine-grained parallel packing strategy for a traversal algorithm that uses the GPU memory hierarchy efficiently.

We enable a *dependency-free* parallel traversal by enriching the datatypes encoding with minimal additional knowledge about child datatypes. The *number of primitives* of child datatypes can be propagated through the type tree, so that the parent type (e.g., `CS`) records the number of primitives in each instance of the child datatype (e.g., `CSvec`). Then, each thread can be assigned a primitive and traverse the tree based on where within the type the primitive falls at each level, requiring only register storage for the traversal state and avoiding any interthread coordination.

To be more specific, the traversal algorithm assigns each primitive datum to a thread and traverses the type tree in a top-down fashion. For each type encountered, read and write offsets for the primitive are updated by using the type and the child datatype's extent, size, and number of primitives. Once the "leaf" datatype is processed, the offsets point to the assigned primitive and where to place it. Algorithm 1 shows the general process. In Line 14, we can change packing to unpacking by switching the direction of the read/write. In Line 12, pointer-jumping is necessary only for `struct` types with multiple derived children; see Section III-A.

The functions `inc_read` and `inc_write` are simple to compute for the `vector` and `subarray` types, as they have a very regular structure. The `vector` type has an $O(1)$ complexity, while 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 are shown to-

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

```
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 type not leaf 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.primitives
10     // process child type; for non-struct,
11     //   translates to type +=sizeof(type)
12     type ← type.child
13 // finished processing datatypes, perform r/w
14 pack_buffer [out] ← user_buffer [in]
```

gether in Algorithm 2. For the type `CS` (and `CSvec`), Trace 3 shows the execution trace of a single thread traversing to its corresponding primitive. Note that the execution trace for this type is the same across all threads launched.

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

For datatypes with variable-length parameters, such as `indexed`, the process is more nuanced. In order to avoid performing a per thread linear scan of the blocklengths, a prefix-sum is performed on the indexed type's list of blocklengths as a preprocessing step. Then, given n blocks and a list of prefix-summed blocklengths $b_0, b_1, \ldots, b_n$, a binary search is performed with terminating condition

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

where $0 \leq h < n$, p is the number of primitives per child datatype and i is the thread (primitive) ID.

We observe that all writes in our algorithm are performed into a contiguous buffer and are thus highly coalesced by

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.primitives
6  Is type a leaf type? (no)
7  // type ← CSvec
8  Increment type pointer by sizeof (vector type)
9  Increment in, out using Alg. 2, with ID, type
10 ID ← ID % type.primitives
11 Is type a leaf type? (yes)
12 pack_buffer [out] ← user_buffer [in]
```

adjacent GPU threads. Given this insight, we enable *zero-copy* memory transactions on the GPU. Instead of packing the data into GPU main 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 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.

C. Packing in the Presence of Resource Contention

A number of communication patterns can introduce resource contention, centered on the greedy, nonpremptive nature of GPU resource scheduling. Contention can occur on both the computational and PCIe levels, when users perform GPU activity during asynchronous communication, or when multiple users or MPI processes on a node access a single GPU. Synchronous communication patterns, 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 there are operations competing for GPU resources, a number of methods can be used to get the data onto the CPU. The simplest solutions of transferring by extent and point-by-point are highly inefficient. Transferring the entire extent of a datatype wastes bandwidth and still requires packing on the CPU. 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. Another option

is to devote a *persistent kernel* for use by MPI operations and use signaling and polling to initiate packing, similar to Stuart et al.'s implementation of message passing on many-core processors [9]. However, since we show latency costs to be extremely important when performing the packing operation and since Stuart et al.'s method produced an increase in these costs, we do not consider this approach (see Sections IV-B and IV-C).

Unfortunately, no way currently exists in 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, not addressed by this paper.

IV. Experimental Evaluation

We evaluate our datatype processing method with packing microbenchmarks on numerous MPI datatypes, comparing with appropriate CUDA alternatives as well as optimized type-specific packing kernels. We additionally evaluate each component of the packing algorithm and show a full context evaluation of GPU-to-GPU communication through a ping-pong test on noncontiguous data. Furthermore, 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 use an Nvidia C2050 GPU with version 4.1 of CUDA, connected to an AMD Opteron 6128. We pin CPU memory used in transfers to enable DMA and enable zero-copy for all datatypes but the `struct` type during packing. For the communication benchmarks, we use two such nodes, connected by QDR Infiniband.

A. Test Datatypes and CUDA Transfer Operations

To benchmark two-and-three dimensional subarrays such as column vectors, we use a `vector` type, compared with the CUDA alternative of `cudaMemcpy2D`. We fix the stride to 512 bytes, which enables maximum performance of the CUDA operation; unaligned arrays greatly hamper CUDA's performance in this regard. We also vary the blocklength to study the performance implications of block width.

For subarrays outside the scope of `vector` representation, we use a four-dimensional `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, we use the same data format as in our test `vector` type, meaning a constant blocklength and a regular displacement pattern. While simpler datatypes would be used in practice, this configuration is a reasonable indicator of `indexed` performance; varying blocklengths cause less divergence than does a uniform blocklength, and a regular displacement allows us to control coalescence in

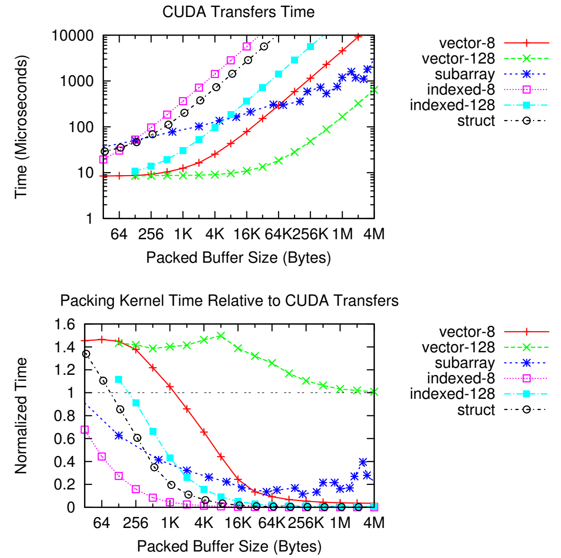

Figure 4. Baseline packing time for several MPI datatypes using the CUDA API, and relative performance of the packing kernel compared with that of CUDA. Trailing numbers in datatype names represent blocklengths, in bytes.

a fine-grained manner. For comparison, we transfer the data block by block using `cudaMemcpy`.

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`. 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.

B. Noncontiguous Packing Performance

For each datatype in Section IV-A, we evaluate the performance of packing from GPU memory into CPU memory, with respect to the size of the packed buffer. We compare our packing algorithm with both CUDA transfer routines and type-specific packing kernels in Figures 4 and 5, respectively.

A number of interesting trends can be observed for the different datatypes. First, transfers on the lower kilobyte level are latency-bound for both CUDA transfer routines and the packing kernel. Given the current architecture of discrete GPUs, little can be done to improve these results, though architectures such as AMD's Fusion [10], which integrate both CPU and GPU cores onto the same die and share address spaces, show promise in bridging this gap in the future. Furthermore, there is a small difference in the latency of issuing kernels and memory operations. Kernel-based packing is thus adversely affected for smaller input sizes, performing worse than the alternative CUDA-based methods (though only by a few microseconds).

Second, the packing kernel is clearly preferable for types that do not have a CUDA equivalent (e.g. `cudaMemcpy2D`),

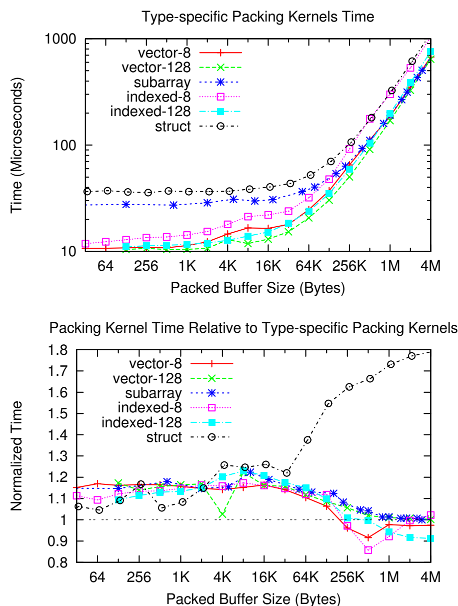

Figure 5. Type-specific packing kernel times and relative generalized pack performance. Trailing numbers in datatype names represent blocklengths, in bytes.

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, while the packing kernel performs approximately 20 times faster. For a larger number of contiguous columns (16 `doubles` per stride in Figure 4), 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 PCIe bandwidth limit, so the relative performance difference begins to converge.

The four-dimensional `subarray` type, despite being reasonably mapped to CUDA transfer routines, 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.

Compared with type-specific implementations, the generic packing algorithm performs well, with little discernible 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

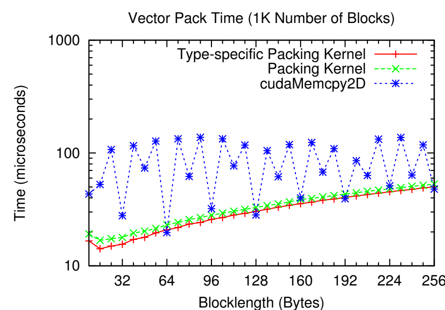

Figure 6. `vector` packing performance, using 1024 blocks with various blocklengths, vs. `cudaMemcpy2D`.

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 different parameterizations lead to significantly different performance, so the performance gap in the different `vectors` in Figure 4 needs to be further explored. Figure 6 fixes the number of blocks in the `vector` and compares the completion times of the packing kernel and the two-dimensional memory copy, varying by the blocklength. As seen in the figure, the performance of CUDA is highly dependent 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, though 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 `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. These operations represent the communication step of a variety of stencil codes, meaning that performance differences from the transfers would be similarly seen in an application context. Table II 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. We cannot currently explain CUDA's X-Z plane transfer performance regression in the $512 \times 512 \times 512$ case.

C. Noncontiguous Packing Performance by Component

The performance metrics in Section IV-B 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 7 shows the performance under three contexts: the full context as presented in Section IV-B, the completion time of

Table II
TRANSFER OF 3-D ARRAY FACES OF DOUBLE-PRECISION VALUES TO THE 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

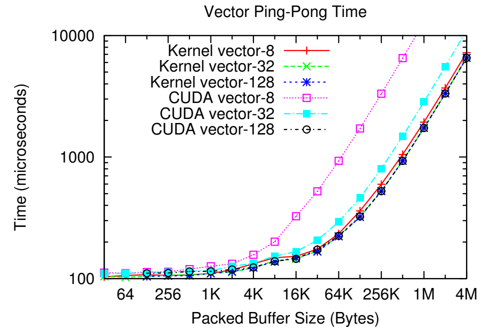

Figure 8. GPU-to-GPU ping-pong test, on the `vector` type with 8, 32, and 128 byte blocks, using our packing kernel (Kernel) and `cudaMemcpy2D` (CUDA).

packing into GPU memory (avoiding PCIe transfers), and the datatype traversal time. Note that the packing operations for small-sized messages are latency bound, meaning the issuing of the packing kernel is the dominant cost.

For medium and large-sized messages, the efficiency of the traversal operation is dependent on the complexity of the type used. For instance, the `vector` type completes quickly because of the simplicity of the traversal logic, while the `subarray` type suffers in performance because of the additional logic and integer computation necessary to represent and pack a subarray of arbitrary dimension.

For types with variable-length parameters, such as `indexed`, the problem becomes memory-bound with respect to the input type and thus sees lesser performance on the traversal. These types must pay the penalty of accessing GPU main memory for every point retrieved, adding significant overhead. The worst case for `indexed` occurs when there is a large set of approximately uniform blocklengths, both increasing the size of the variable-length parameter space and maximizing branch divergence and nonlocality in the search. Similar trends are seen in the `struct` type, though to a higher degree because of an even higher reliance on the variable length parameters to perform the point retrieval; each block can be a separate datatype (see Table I).

The impact of the read/write stage of packing on performance is determined by the data layout and whether the type has variable-length parameters. 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 IV-A, zero-copy is not used in the `struct` type, causing a higher relative performance degradation than seen in the other types due to the serialization of the packing and PCIe operations.

D. 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 communication. Because of the poor 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; communication with data packed at the rate of 4 MB/s will perform poorly. Furthermore, we do not consider the use of GPUDirect, a kernel patch provided by Nvidia that allows both Infiniband and CUDA to pin the same block of memory. This will be the focus of future work, though the integration of it will equally benefit the packing algorithm and CUDA alternatives. Figure 8 shows the completion time of a GPU-to-GPU ping-pong benchmark. The sender packs the data from GPU memory into contiguous CPU memory, immediately followed by a send operation, while the receiver receives and unpacks the message into GPU memory. This process is then repeated back to the original sender.

The efficiency of the communication is dependent, as expected, on the data layout. A small blocklength, which favors the packing operation, causes 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, as network latency was measured to be 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.

E. Resource Contention Effects on Packing

To induce the contention scenarios discussed in Section III-C, 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 this is to tie it closely to a packing operation

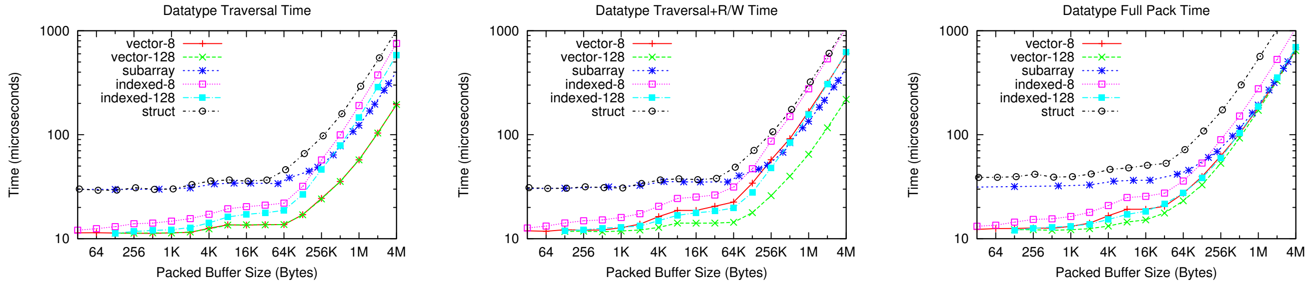

Figure 7. Packing time, by component. "Traversal" refers to calculating input/output offsets. "R/W" refers to the read/write operation at the end of the traversal operation. "Full" refers to packing and sending the result across the PCIe bus into CPU memory. Trailing numbers in datatype names represent blocklengths, in bytes.

(using the `vector` type), with packing time similar to the application operation time.

To form a baseline for comparison, we time each operation in isolation. To measure contention effects on the pack/copy operation, we initiate the application operation, then initiate and time the noncontiguous pack/copy. To measure contention effects on the application operation, we initiate the pack/copy operation, 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.

For the following experiments, we used a vector of double-precision primitives 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 to best show the effects of resource contention due to each operation having a similar run time.

Table III shows these exemplars. 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 there are available SMs. The two-dimensional memory copy, avoiding the SMs entirely, sees no degradation in performance.

For the PCIe experiment from GPU to CPU, both the application operation and the pack/memory copies suffer, since both must use the same lane of the bridge. However, an interesting finding can be seen in the app-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), 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).

For the PCIe experiment from CPU to GPU, 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, although 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.

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

V. Related Work

A number of efforts have been made to integrate GPU functionality into HPC environments, with modifications at both the programming model and library levels to account for discrete GPU main memory. At the programming model level, Gelado et al. created the Asymmetric Distributed Shared Memory (ADSM) model [11] to unify GPU address spaces across a cluster, providing coherence on contiguous chunks of memory. The consistency model would have to become much more complex to enable consistency for noncontiguous data or partial data within a chunk, so our work does not apply here without modifying the underlying memory model.

Zippy [12] combines the message-passing and shared-memory models and provides a single address space for all GPUs in the cluster, using MPI as its backend. Zippy works on array-based data with dimensionality extending beyond the two-and-three dimensional arrays representable by CUDA. 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.

At the library level, Distributed Computing for GPU Networks (DCGN) [9] provides GPU-sourced MPI communication through signaling/polling mechanisms on the host CPU and can leverage our work for noncontiguous communication. Unfortunately, given current architectural constraints, the signaling and polling operations are cycle-consuming and lead to high latencies in GPU-sourced communication routines.

Lawlor describes the system `cudaMPI`, which works on top of MPI [13], focusing on the application of the la-

Table III
APPLICATION 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 TIME BETWEEN INITIALIZATION OF THE PACKING/CUDA OPERATION AND ITS COMPLETION. SECTION IV-E DISCUSSES THE PARAMETERS.

	SM			PCIe (CPU→GPU)			PCIe (GPU→CPU)		
Workload Order	**App**	**Type Proc.**	**Total**	**App**	**Type Proc.**	**Total**	**App**	**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
App→Pack	-	3.52	3.55	-	3.65	4.08	-	3.18	5.09
App→CUDA	-	3.00	3.03	-	3.66	4.06	-	5.53	5.54
Pack→App	3.53	-	3.56	4.08	-	4.11	5.08	-	5.11
CUDA→App	1.03	-	3.00	4.05	-	4.07	5.53	-	5.53

tency/bandwidth performance model on GPUs for performance projection under different configurations. He also discusses noncontiguous memory transfer as an application-specific column-vector transfer. Our work directly applies to this framework. However, Lawlor does not take into consideration MPI datatypes.

The MVAPICH2 team has made their MPI implementation partially aware of the CUDA memory space [14]. They have provided the ability to communicate contiguous GPU buffers and, more recently, buffers in GPU memory that can be represented as a single `vector` type [15]. However, their methodology is based solely on CUDA's two-dimensional memory copies and thus cannot be extended to other datatypes. Our algorithm can be integrated into their buffer-pool-based framework in a simple manner.

VI. CONCLUDING REMARKS

We have presented one important aspect of integrating GPUs into HPC environments: the processing of arbitrary, noncontiguous 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 nonexclusive data layouts: highly noncontiguous data, and irregularly located data. These cases are especially important for future applications because of the extensive research being carried out 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 would be helpful from an optimization standpoint: algorithms could have local access patterns that differ from global communication patterns; and if efficient packing were available, applications could focus more on optimizing the local patterns.

ACKNOWLEDGMENTS

This work was supported in part by the U.S. Department of Energy Office of Science and contract DE-AC02-06CH11357, the National Science Foundation Expeditions in Computing and Grant No. 0958311, as well as NVIDIA donations.

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] 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.

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

[11] 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*, 2010, pp. 347–358.

[12] 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.

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

[14] 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*, ser. ISC '11, 2011.

[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*, ser. Cluster '11, 2011.