Processing MPI Derived Datatypes on Noncontiguous GPU-Resident Data

John Jenkins, James Dinan, Pavan Balaji, *Senior Member, IEEE,* Tom Peterka, Nagiza F. Samatova, *Member, IEEE,* and Rajeev Thakur

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.

Index Terms—MPI, Graphics Processing Unit, CUDA, Datatype

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.

Numerous technical challenges arise from adding a fundamentally different computing architecture to existing systems. Aside from the cost of developing, porting, and optimizing codes to run on the GPU, a greater concern is 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, communication operations involving data resident in GPU memory requires 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, 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 of data for use in MPI operations. 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], no generalized packing methodology exists 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 to accelerate

- *J. Jenkins and N. F. Samatova are with the Department of Computer Science, North Carolina State University, Raleigh, NC 27695*
  E-mail: jpjenki2@ncsu.edu, samatova@csc.ncsu.edu
- *J. Dinan, P. Balaji, T. Peterka, and R. Thakur are with the Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439*
  E-mail: dinan@mcs.anl.gov, balaji@mcs.anl.gov, thakur@mcs.anl.gov

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 (though exposing GPUs to MPI and other distributed programming models is an active area of research [9], [10], [11], [12], [13], [14]—see Section 1 of Supplementary Material). 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*: As a first step toward 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*: 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, in order to better match GPU hardware characteristics.

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 present experimentation illustrating such 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 Sections 4 and 5. In Section 6 we provide concluding remarks and discussion.

This paper is an extended version of a previously published paper [15]. In this paper, we provide a substantially expanded set of experimental results, including application benchmarks, comparison with MVAPICH GPU-enabled communication, and a more complete set of derived datatypes in microbenchmarks.

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

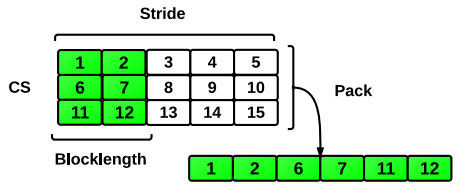


Fig. 1. An array slice, an MPI vector datatype `CS` encoding it, and the slice's packed form. The corresponding datatype initializer (for C element type `double`) is `MPI_Type_vector(3, 2, 5, MPI_DOUBLE, &CS)`.

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. Other datatypes include 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.

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

In order to avoid initiating I/O or network operations for each individual piece of data, MPI implementations *pack* the data into contiguous buffers. For the computational aspect of this process to be efficient, 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 $\langle$*type*, *displacement*$\rangle$ 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 contiguous data encoded by the type.

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 is 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. Fourth, 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 MVAPICH's buffer-pool-based framework in a simple manner, however.

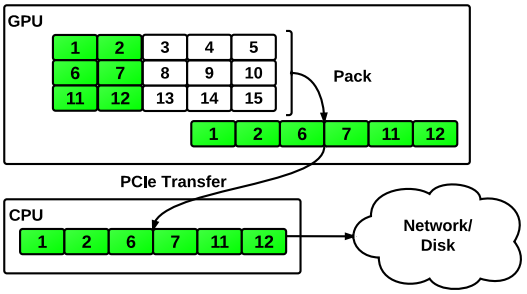


Fig. 2. Communication pattern necessitating GPU packing (unpacking if arrows are reversed).

3 In-GPU Datatype Processing

The communication data flow driving our datatype processing is shown in Figure 2, using as an example the `CS` datatype from Figure 1. Given a datatype definition, the data is packed within GPU memory by using a kernel, then is 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`. 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 MPI implementations such as dataloops typically use, GPU best practices 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 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). 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 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.

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

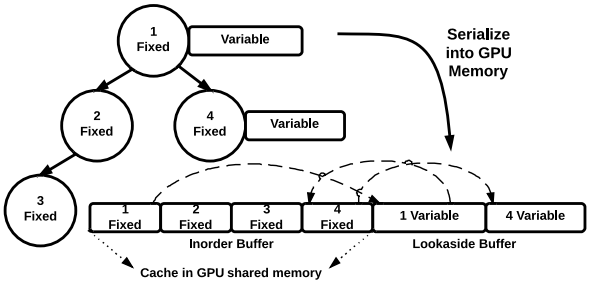


Fig. 3. Example type tree, serialized into GPU memory. Branches in trees appear only for `struct` types.

See Table 1 for a listing of datatypes with their fixed- and variable-length parameters.

Figure 3 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 datum requires no additional linkage information for nearly every type. The only exception is a `struct` type with multiple derived datatype children, requiring additional pointers in the 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

CPU-side datatype processing implementations, such as dataloops, 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. Hence, a straightforward "port" of existing methods is undesirable. 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 dataloop's datatype 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.

Recall that datatypes, and hence any datatype encoding, are formally represented as a list of $\langle$*type*, *displacement*$\rangle$ pairs. To facilitate our parallel traversal, we assign a canonical integer ID to each pair in the sequence. Then, given an ID and the datatype encoding, we can compute in which part of the encoded datatype the primitive appears. For example, consider the `vector` type `CS` in Figure 1, with underlying type `MPI_DOUBLE` (making `CS` a *leaf* type). There are three blocks, each containing two primitives. Given the canonical ID 3 (position 7 in the figure), we can conclude that the primitive resides in block 1 of the type by the computation "ID / blocklength" and is element 1 in the block by the computation "ID % blocklength." In other words, we compute the block of the datatype in which the primitive appears, then compute the location within that block. This calculation can similarly be performed for other datatypes.

When defining derived datatypes, 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`). When mapping canonical IDs to locations within a derived datatype, computations must be performed with respect to the number of primitives within the child datatype. So, if `CS` has `CSvec` as a child type, the global ID mapping to block and block offset in `CS` would occur by means of the computations "(ID / # primitives) / blocklength" and "(ID / # primitives) % blocklength," where the number of primitives is with respect to `CSvec`. Further, we can recursively perform the same operations on `CSvec` by binding the *global* ID to the *local* ID within `CSvec`, using "(ID % # primitives)."

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, aside 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. The composed data structure `type` is based on Table 1. 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. By "recursively" assigning the element ID to a pack based on the type being visited on Line 9 (see Section 3.2.1), the algorithm need merely track and update the memory read and write offsets as each level of the tree is visited. 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.

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.#primitives
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

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.

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.#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 the composite types `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.

For the datatypes with variable-length parameters, such as `indexed`, the process is more nuanced. To avoid a per-thread linear scan of the blocklengths, preprocessing is performed to allow a logarithmic-time binary search. First, a prefix-sum is performed on the indexed type's list of blocklengths. Then, given a count of n and a list of prefix-summed blocklengths $b_0, b_1, \ldots, b_n$, the terminating condition for thread (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 $h = n - 1$.

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 Increment type pointer by `sizeof` (vector type)
8 // type ← CSvec
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]

Having observed that all writes are performed into a contiguous buffer and are thus highly coalesced by adjacent GPU threads, we enable *zero-copy* memory transactions in order to dramatically improve the packing operation. 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, 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.

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 also look at full-context GPU-to-GPU communication through a non-contiguous ping-pong test, comparing with MVAPICH version 1.8. Moreover, we examine the effects of GPU resource contention on packing and memory copy operations by modifying the issuing order of packing and other operations. For all tests, we used 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. An extended collection of experiments can be found in Section 2 of the Supplementary Material.

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 analyze the performance implications of block width.

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 use a `struct` type to test the effect of thread divergence on writing. We use a simple C-style struct containing an 8-byte `double`, two 4-byte `ints`, and a `char`, 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 4 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.

A number of interesting trends can be observed for the different datatypes. First, since a relatively large gap exists 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 [16], 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. Besides latency, 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. While the packing kernel is up to 20 times faster for an 8-byte-blocklength `vector`, the memory copy outperforms the packing kernel in all cases for a 128-byte-blocklength `vector`, especially for small and medium-sized inputs, because of the additional kernel

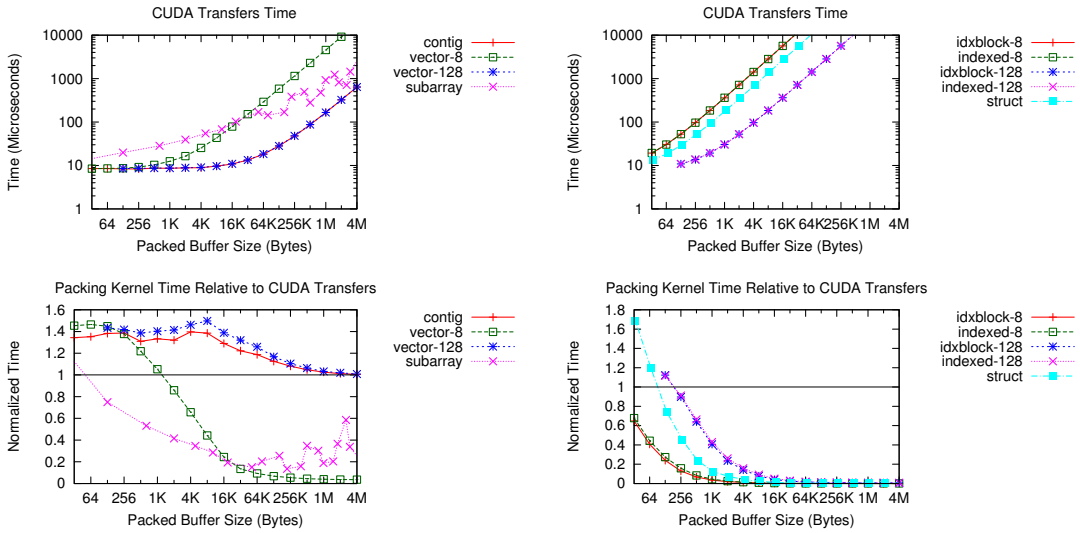


Fig. 4. Time-to-CPU packing time using the CUDA API, and corresponding relative performance of packing kernel.

latency. The relative performance for large blocks converges as the PCIe bandwidth is reached.

The four-dimensional `subarray` type, despite being reasonably mapped to the CUDA API, sees major performance improvements when using a kernelized packing operation. Since the three-dimensional memory copies must be made iteratively in order 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 performance of each type except the `struct` type show an approximately 20–30% overhead, reaching near parity for buffers larger than a megabyte (in packed form). This overhead, however, amounts to between about two and five microseconds for most inputs. The differences in 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. The differences in the `struct` implementations (a 20% to 80% overhead compared with that os the hand-coded version) are a result of hard-coding the relative location of each `struct` primitive, benefiting from compiler optimization and greatly simplified traversal logic, and is an exceptional, nongeneral case. For more detailed results, refer to Section 2.1 of the Supplementary Material.

Since the `vector` type is one of the more widely used MPI datatypes and performance is highly dependent on the parameterization, we further explore the `vector` type's performance characteristics in Figure 5. We fix the number of blocks and compare 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 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 these are not shown in the paper.

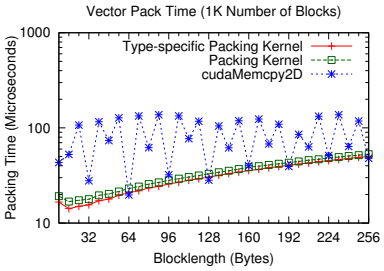


Fig. 5. `vector` pack performance vs. `cudaMemcpy2D`.

4.3 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`), 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. For this test, version 1 of GPUDirect is used, allowing both CUDA and InfiniBand drivers to pin the same memory and avoid extraneous memory copies. Figure 6 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

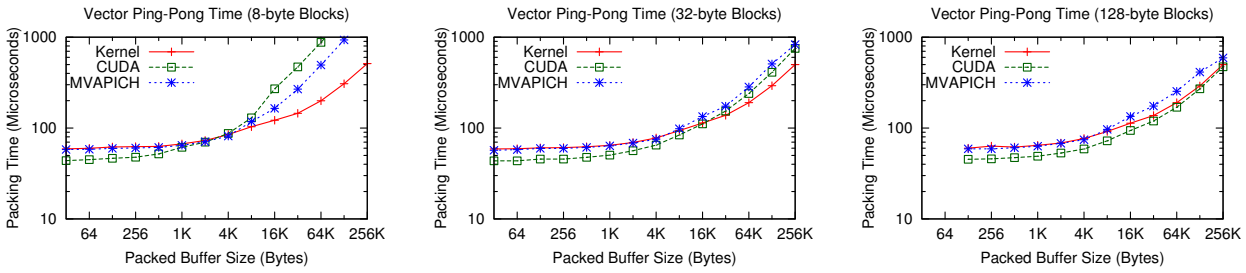


Fig. 6. GPU-to-GPU ping-pong test, on the `vector` type with 8-, 32-, and 128-byte blocks, compared with `cudaMemcpy2D`. The `vector` stride is aligned to maximize CUDA performance.

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- and medium-sized buffers and begins to outperform MVAPICH's vector 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.

4.4 Resource Contention Effects on Packing

The packing methodology 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 only a single kernel to be run on each multiprocessor—can easily lead to starvation of a packing kernel, in turn leading 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.

To induce these contention scenarios, we use a few simple operations to stress the resource in question. We call these the application (user) 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.

The parameter space for this experiment is enormous, so we chose 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.

Our experimental results are shown in Table 2. We time each operation in isolation as a baseline. To measure contention effects induced by the first-come, first-serve GPU scheduling policy, we initiate one of the operations (either application operation or pack/copy) followed by the other operation, measuring the completion time of the latter. For example, the row "User→Pack" initiates the application operation followed by the packing operation. We also measure the completion time of both operations as a whole, to assess the degree of overlap between the operations.

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 User→Pack case, however, 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 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

TABLE 2
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 latter 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

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 shown in Table 2. 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

In this section, we evaluate GPU datatype processing on both a stencil computation and an analysis code.

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 [17]. 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 3 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 [18] 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 [19], which converts a point cloud to a polyhedral mesh. When executed in a spatially decomposed data-parallel manner, each MPI process computes the data structure shown in Figure 7.

TABLE 3
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

```
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;
};
```

Fig. 7. HACC analysis data structure to pack.

When writing and reading results from parallel storage using MPI-IO, the data in Figure 7 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 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.

TABLE 4
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

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 4 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 pack the GPU-resident data 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 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 have shown 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 efficient packing is available, applications could focus more on optimizing the local patterns.

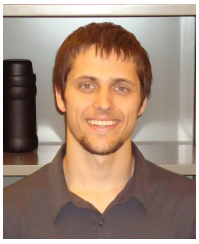
Overall, we view our method as complementing the goal of robust integration of GPU technology into high-performance data movement frameworks such as MPI, as well as a baseline for future MPI library implementations. A complete solution to GPU data movement within MPI not only would minimize internal memory copies and fully utilize current/future architecture-specific optimizations but also would be able to flexibly determine the best methodology for transferring the data, especially noncontiguous data. Refer to Section 3 of the Supplementary Material for an extended discussion of these concerns.

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] 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.
[12] O. Lawlor, "Message passing for GPGPU clusters: CudaMPI," in *IEEE Cluster PPAC Workshop*, 2009, pp. 1–8.
[13] 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.
[14] 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.
[15] J. Jenkins, J. Dinan, P. Balaji, N. F. Samatova, and R. Thakur, "Enabling fast, noncontiguous gpu data movement in hybrid mpi+gpu environments," in *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, 2012, pp. 468–476.
[16] N. Brookwood, "AMD fusion family of APUs: Enabling a superior, immersive PC experience," *Insight*, vol. 64, pp. 1–8, 2010.
[17] 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
[18] 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.
[19] 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.



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 a software architect at Intel Corporation, engaged high performance computing research. He was previously the James Wallace Givens postdoctoral fellow at Argonne National Laboratory. He received his Ph.D. and M.S. degrees in computer science from the Ohio State University, in Columbus, Ohio. He received his B.S. degree in computer systems 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 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. 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. He 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 the Computer Science Department of North Carolina State University and a senior research scientist in the Mathematics and Computer Science Division of Oak Ridge National Laboratory. She received a 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, Moscow, in 1993. She also obtained an M.S. degree in computer science in 1998 from the University of Tennessee, Knoxville. She 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 of the Mathematics and Computer Science Division at Argonne National Laboratory, where he is also a senior 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.

. The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory ("Argonne"). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.