Supplemental Material: MPI Derived Datatypes Processing on Noncontiguous GPU-resident Data

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

1 Related Work

A number of efforts have been undertaken to integrate GPU functionality into an HPC environment, with modifications at the application, programming model, and library levels to account for a discrete GPU main memory space. Work related to MVAPICH [1], [2] is discussed in Section 2.3 of the Main Material.

At the application level, algorithms that use both MPI and GPUs, such as Jacobsen et al.'s flow computation algorithm [3], are modified to allow efficient GPU computation, for example, changing the problem space partitioning to benefit GPU access patterns. MPI datatypes differ from these specialized data structures in that the datatypes efficiently encode a subset of the data structures used, for use in communication and I/O routines.

At the programming model level, the asymmetric distributed shared memory model provides a single GPU address space across a cluster, while leaving GPUs aware of only their local memory space [4]. The consistency model is designed for and allows operating and processing on the shared address space in contiguous chunks with memory coherence; it would have to become more complex in order to enable the transfer and consistency of noncontiguous data or partial data within a contiguous buffer.

Zippy [5] combines the message-passing and shared-memory models (based on Global Arrays) and provides a single address space for all GPUs in the cluster, using MPI as its backend. Zippy works specifically on multidimensional array-based data, so our work is applicable both to representing an area that needs to be transferred (such as noncontiguous array boundaries) and to subsequently packaging that data efficiently.

At the library level, Distributed Computing for GPU Networks (DCGN) [6] extends MPI and utilizes signaling/polling mechanisms to allow for GPU-sourced communication. It also uses existing MPI libraries as a backend, meaning our work can directly benefit theirs. Unfortunately, given the current architectural constraints, the signaling and polling operations are cycle-consuming and lead to high latencies in GPU-sourced communication routines.

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

- *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

2 Additional Experiments

2.1 Packing Comparison with Type-Specific Kernels

Figure 1 shows the performance of generalized packing relative to hand-optimized packing kernels. A summary of these results can be found in Section 4.2 of the Main Material.

2.2 3-D Plane Transfer

For three-dimensional arrays, a single `vector` type can be used to send each face of the array: the fully contiguous X-Y face, the contiguous-per-row X-Z face, and the noncontiguous Y-Z face. Together, these operations represent the communication step of a variety of matrix algorithms, such as stencil computation. Table 1 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

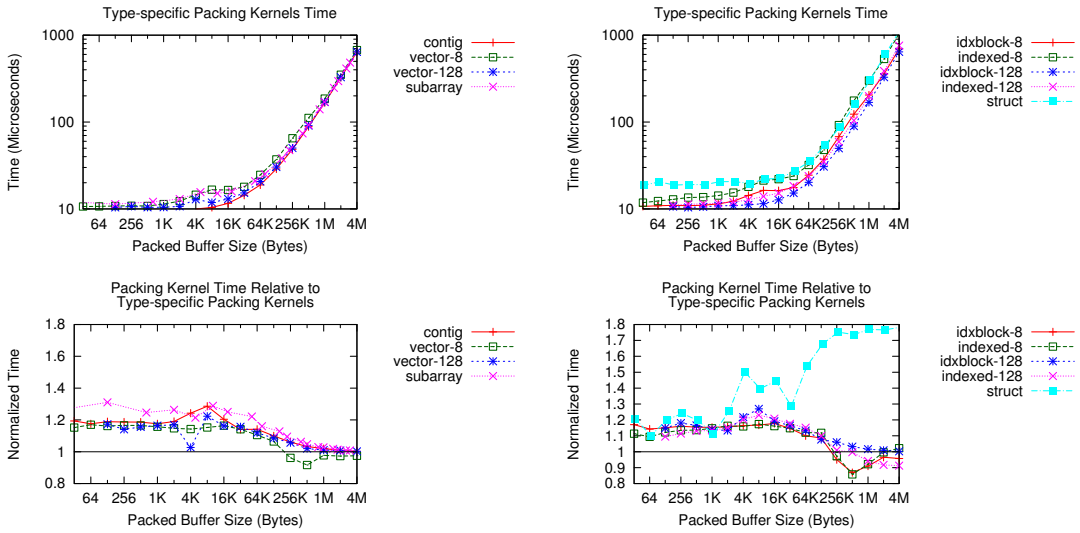

Fig. 1. Hand-coded packing kernel times and relative generalized pack performance.

getting noncontiguous data. Note that the CUDA memory copy seems to degrade in performance for the X-Z plane transfer in the $512 \times 512 \times 512$ case. We cannot currently explain this behavior.

TABLE 1
Two-dimensional plane transfer to CPU versus cudaMemcpy2D.

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

2.3 Noncontiguous Packing Performance by Component

The performance metrics in Section 4.2 of the Main Material leaves out some key information about our packing methodology. For instance, what are the costs of PCIe transfers? What is the effect of memory layout on the overall performance? To answer these questions, Figure 2 shows packing performance under three contexts: the full context as presented in Section 4.2 of the Main Material, the completion time of packing into GPU memory (avoiding PCIe transfers), and the datatype traversal time. Note that the packing operations for small messages are latency bound, meaning the issuing of the packing kernel is the dominant cost.

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

For types with variable-length parameters, such as `indexed`, the problem becomes memory-bound with respect to the input type and sees less performance on the traversal. The `indexed` type, performing a binary search, must access GPU main memory for every point retrieved, although coalescence between adjacent threads in the search helps reduce the cost. Note that the worst case for `indexed` occurs with a large set of approximately uniform blocklengths, maximizing the size of the variable-length parameter space as well as branch divergence in the search. Similar trends are seen in the `struct` type, although to a higher degree because each block is a separate datatype (see Table 1 in the Main Material). The cost of performing the binary search for these types can be seen by comparing the `indexed` and `indexedblock` types. Specifically, the binary search implementation of `indexed` type traversal causes significant overhead, although for packed buffer sizes less than 64 KB the absolute overhead is no more than 9 microseconds.

The impact of the read/write stage of packing on performance is determined by the encoded data layout. The best example is shown in the `indexed` and `vector` types. With a small blocklength and thus high noncontiguity, reading the values is the bottleneck of the datatype processing. With a large blocklength and thus a higher degree of contiguity, the reading is an efficient process because of the much higher degree of coalescence. If the type has variable-

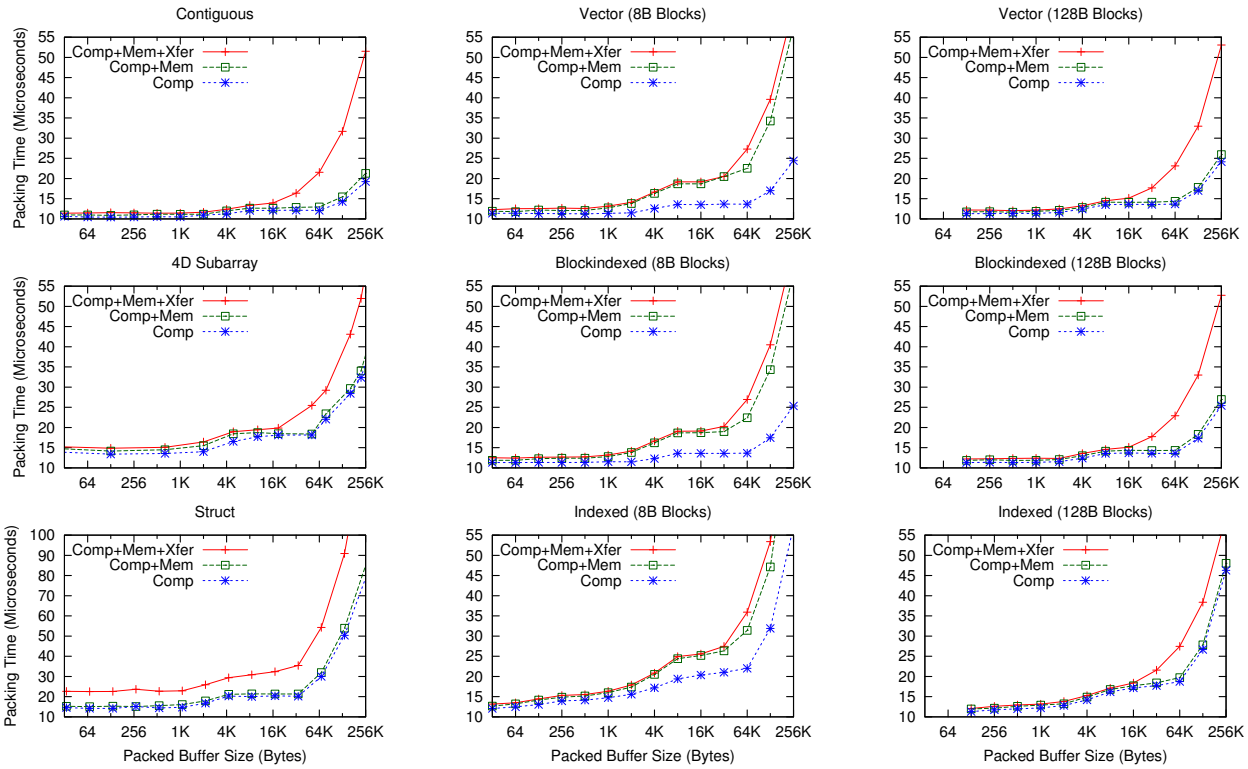

Fig. 2. Packing time, by component. “Comp”: traversing the type, computing input/output offsets. “Mem”: performing the read/write operation at the end of the traversal operation. “Xfer”: sending the packed data across the PCIe bus.

length parameters, then the traversal is the primary cost, but significant overhead can still be seen when packing highly noncontiguous data, such as with the `indexed` type with 8-byte blocks.

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

2.4 Packing Performance: Efficacy of Zero-copy

As the usage of zero-copy is prevalent throughout our experimentation as an optimization of GPU-to-CPU data movement, we performed additional benchmarks to determine the performance benefits of it. Figure 3 shows two metrics, the completion time of our packing method without zero-copy enabled and the relative completion time of packing with zero-copy enabled. As can be seen, the completion time with zero-copy is roughly 60–75% of the time without, corresponding to a speedup of 1.33 to 1.66. As mentioned in the Main Material, the performance benefit is the result of implicitly pipelining the packing and PCIe operations through zero-copy.

3 Extended Discussion

3.1 Choosing An Appropriate Data Movement Methodology

As observed in the Main Material, a large number of factors must be considered when determining whether, for particular memory layouts and levels of system activity, to perform kernelized packing, type-specific hand-optimized versions, or CUDA DMA. This determination would ideally take into account the degree of noncontiguity of the data, the availability of higher-performing type-specific kernels or CUDA alternatives, and awareness of competing operations for limited GPU resources. For example, CUDA DMA can be used in place of the generic packing algorithm for a single `vector` type (e.g., `CSvec`) by merely analyzing the strides/blocklengths for CUDA-optimized parameters.

For GPU-aware datatype processing implementations, we expect that the simplest and most efficient way to track this information is as the datatype is being built, with small checks at communication time when the buffer address is known. With small additions to the datatype data structures, one can easily track most of the information necessary to make an informed decision. For example, flags can be used in the `vector` type representation to flag the type as suitable for DMA based on the blocklength and stride, the performance of which we have shown to be sensitive to changes in the parameters. At communication time, unaligned buffers can override the flag. Determining whether `subarrays` and `vectors` can be packed using a

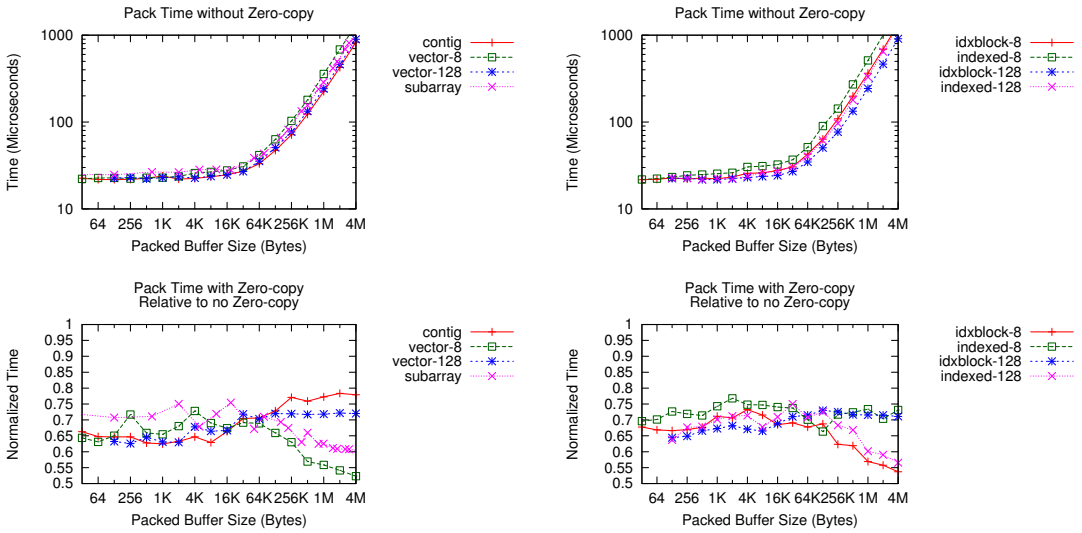

Fig. 3. Time-to-CPU packing time of the packing kernel, with and without zero-copy.

CUDA two- or three-dimensional memory copy is a simple matter, and sizes can be checked against known or heuristic values to tradeoff kernel latency and DMA efficiencies. Moreover, composed types are likely to be more efficiently packed by using a kernel (to avoid latency from multiple kernel and/or DMA issuance), but a hybrid implementation where processing a "parent" datatype can involve issuing multiple packing kernels for appropriate child datatypes is also an intriguing possibility; further research is necessary to evaluate the efficacy of such an approach.

The changes described in the previous paragraph touch on all aspects of datatype processing. A much simpler first step to make these choices is to allow the user full discretion in what processing strategy is used. The MPI standard allows hints in the form of *attributes* to be passed to datatypes, which were used in the recent MPI-ACC work [8] to eliminate the overhead of using CUDA UVA. This attribute interface (`MPI_Type_create_keyval` and `MPI_Type_set_attr`), as in the MPI-ACC work, can be easily used by the MPI runtime to specify what packing strategy to pursue (DMA vs. kernel). Thus, if an application is using double-buffering to perform computation during communication routines, the DMA method can be specified for use in order to avoid kernel starvation.

Handling GPU Resource Contention

As seen in Section 4.4 of the Main Material, the optimal choice of how to pack GPU data depends heavily on what other operations are going on in the system. To summarize the results, starvation can occur for both compute kernels (if the issued thread blocks are large enough to occupy GPU multiprocessors) and PCIe transfers using the same lane.

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

When the datatype is nontrivial and resource contention is preventing a packing kernel from being run, a number of methods can be used to get the data onto the CPU. The two simplest ones are transferring by extent and transferring point by point, both of which are highly inefficient. Transferring the extent of a datatype wastes bandwidth and still requires packing on the CPU end. Transferring point by point suffers from the high latency of initiating copies from the CPU. Both have the potential for interfering with user kernels that rely on host-device transfers. Performing some combination of the two, similar to data sieving in the ROMIO MPI-IO implementation [9], would need more complex processing and memory management on the CPU side and would still have the problems of both methods, albeit reduced in severity. Another option is to devote a *persistent kernel* for use by MPI operations and utilize signaling and polling to initiate packing, similar to Stuart and Owens's implementation of message passing on many-core processors [6]. However, since we show latency costs to be extremely important when performing the packing operation and since their method produced an increase in these costs, we do not consider this approach (see Section 4.2 of the Main Material and Section 2.3).

Unfortunately, no way currently exists within the CUDA or OpenCL interfaces to query the level of resource utilization on the GPU aside from high-level utilization (provided through the NVIDIA driver), complicating the selection of a globally efficient strategy without application-specific knowledge, which can similarly be provided by users through datatype attributes. 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.

3.2 Further Optimizations

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

In order to improve the overall communication process when GPUs are involved, three versions of GPUDirect currently are available. The first version for use with InfiniBand clusters allows the CPU to pin the same physical memory to be used by both InfiniBand drivers and CUDA, hence avoiding an internal memory copy on the CPU in communication. This optimization, used in our presented results, is orthogonal to our packing methodology. The second version allows direct GPU-to-GPU communication for GPUs that are on the same system/PCIe bus, never reaching CPU main memory. The third version allows remote direct memory access (RDMA) to and from GPUs in the same network (the network card must on the same system/PCIe bus as the corresponding GPU). The second and third optimizations are also orthogonal to our packing methodology, but with an important implication for our method: the packing operation would now pack into GPU memory rather than CPU memory.

References

[1] 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.

[2] 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.

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

[4] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W. mei W. Hwu, "An asymmetric distributed shared memory model for heterogeneous parallel systems," in *ASPLOS '10 Proc. of the fifteenth edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, 2010, pp. 347–358.

[5] 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.

[6] 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.

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

[8] A. M. Aji, J. Dinan, D. Buntinas, P. Balaji, W.-c. Feng, K. R. Bisset, and R. Thakur, "MPI-ACC: An integrated and extensible approach to data movement in accelerator-based systems," in *14th IEEE International Conference on High Performance Computing and Communications*, Liverpool, UK, June 2012.

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