MPI-ACC: Accelerator-Aware MPI for Scientific Applications

Ashwin M. Aji, Lokendra S. Panwar, Feng Ji, Karthik Murthy, Milind Chabbi, Pavan Balaji, Keith R. Bisset, James Dinan, Wu-chun Feng, John Mellor-Crummey, Xiaosong Ma, and Rajeev Thakur

Abstract—Data movement in high-performance computing systems accelerated by graphics processing units (GPUs) remains a challenging problem. Data communication in popular parallel programming models, such as the Message Passing Interface (MPI), is currently limited to the data stored in the CPU memory space. Auxiliary memory systems, such as GPU memory, are not integrated into such data movement standards, thus providing applications with no direct mechanism to perform end-to-end data movement. We introduce MPI-ACC, an integrated and extensible framework that allows end-to-end data movement in accelerator-based systems. MPI-ACC provides productivity and performance benefits by integrating support for auxiliary memory spaces into MPI. MPI-ACC supports data transfer among CUDA, OpenCL and CPU memory spaces and is extensible to other offload models as well. MPI-ACC's runtime system enables several key optimizations, including pipelining of data transfers, scalable memory management techniques, and balancing of communication based on accelerator and node architecture. MPI-ACC is designed to work concurrently with other GPU workloads with minimum contention. We describe how MPI-ACC can be used to design new communication-computation patterns in scientific applications from domains such as epidemiology simulation and seismology modeling, and we discuss the lessons learned. We present experimental results on a state-of-the-art cluster with hundreds of GPUs; and we compare the performance and productivity of MPI-ACC with MVAPICH, a popular CUDA-aware MPI solution. MPI-ACC encourages programmers to explore novel application-specific optimizations for improved overall cluster utilization.

Index Terms—Heterogeneous (hybrid) systems, parallel systems, distributed architectures, concurrent programming

1 Introduction

Graphics processing units (GPUs) have gained widespread use as general-purpose computational accelerators and have been studied extensively across a broad range of scientific applications [1], [2], [3]. The presence of general-purpose accelerators in high-performance computing (HPC) clusters has also steadily increased, and 15 percent of today's top 500 fastest supercomputers (as of November 2014) employ general-purpose accelerators [4].

Nevertheless, despite the growing prominence of accelerators in HPC, data movement on systems with GPU accelerators remains a significant problem. Hybrid programming with the Message Passing Interface (MPI) [5] and the Compute Unified Device Architecture (CUDA) [6] or Open Computing Language (OpenCL) [7] is the dominant means of utilizing GPU clusters; however, data movement between processes is currently limited to data residing in the host memory. The ability to interact with auxiliary memory systems, such as GPU memory, has not been integrated into such data movement standards, thus leaving applications with no direct mechanism to perform end-to-end data movement. Currently, transmission of data from accelerator memory must be done by explicitly copying data to host memory before performing any communication operations. This process impacts productivity and can lead to a severe loss in performance. Significant programmer effort would be required to recover this performance through vendor- and system-specific optimizations, including GPUDirect [8] and node and I/O topology awareness.

We introduce MPI-ACC, an integrated and extensible framework that provides end-to-end data movement in accelerator-based clusters. MPI-ACC significantly improves productivity by providing a unified programming interface, compatible with both CUDA and OpenCL, that can allow end-to-end data movement irrespective of whether data resides in host or accelerator memory. In addition, MPI-ACC allows applications to easily and portably leverage vendor- and platform-specific capabilities in order to optimize data movement performance. Our specific contributions in this paper are as follows.

- An extensible interface for integrating auxiliary memory systems (e.g., GPU memory) with MPI
- An efficient runtime system, which is heavily optimized for a variety of vendors and platforms (CUDA and OpenCL) and carefully designed to minimize contention with existing workloads

- *A.M. Aji, L.S. Panwar, and W.-c. Feng are with the Department of Computer Science, Virginia Tech, Blacksburg, VA 24061. E-mail: {aaji, lokendra, feng}@cs.vt.edu.*
- *F. Ji and X. Ma are with the Department of Computer Science, North Carolina State University, Raleigh, NC. E-mail: fji@ncsu.edu, ma@csc.ncsu.edu.*
- *K. Murthy, M. Chabbi, and J. Mellor-Crummey are with the Department of Computer Science, Rice University, Houston, TX. E-mail: {ksm2, mc29, johnmc}@rice.edu.*
- *P. Balaji, J. Dinan, and R. Thakur are with the Mathematical and Computer Science, Argonne National Laboratory. E-mail: {balaji, dinan, thakur}@mcs.anl.gov.*
- *K.R. Bisset is with the Virginia Bioinformatics Inst., Virginia Tech, Blacksburg, VA. E-mail: kbisset@vbi.vt.edu.*

Manuscript received 23 Apr. 2014; revised 9 Mar. 2015; accepted 19 Apr. 2015. Date of publication 16 June 2015; date of current version 13 Apr. 2016. Recommended for acceptance by D. Padua.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2015.2446479

- An in-depth study of high-performance simulation codes from two scientific application domains (computational epidemiology [9], [10] and seismology modeling [11])

We evaluate our findings on *HokieSpeed*, a state-of-the-art hybrid CPU-GPU cluster housed at Virginia Tech. Microbenchmark results indicate that MPI-ACC can provide up to 48 percent improvement in two-sided GPU-to-GPU communication latency. We show that MPI-ACC's design does not oversubscribe the GPU, thereby minimizing contention with other concurrent GPU workloads. We demonstrate how MPI-ACC can be used in epidemiology and seismology modeling applications to easily explore and evaluate new optimizations at the application level. In particular, we overlap MPI-ACC CPU-GPU communication calls with computation on the CPU *as well as* the GPU, thus resulting in better overall cluster utilization. Results indicate that the MPI-ACC–driven communication-computation patterns can help improve the performance of the epidemiology simulation by up to 13.3 percent and the seismology modeling application by up to 44 percent over the traditional hybrid MPI+GPU models. Moreover, MPI-ACC decouples the low-level memory optimizations from the applications, thereby making them scalable and portable across several architecture generations. MPI-ACC enables the programmer to seamlessly choose between CPU, GPU, or any accelerator device as the communication target, thus enhancing programmer productivity.

This paper is organized as follows. Section 2 introduces the current MPI and GPU programming models and describes the current hybrid application programming approaches for CPU-GPU clusters. We discuss related work in Section 3. In Section 4, we present MPI-ACC's design and its optimized runtime system. Section 5 explains the execution profiles of the epidemiology and seismology modeling applications, their inefficient default MPI+GPU designs, and the way GPU-integrated MPI can be used to optimize their performances while improving productivity. In Section 6, we evaluate the communication and application-level performance of MPI-ACC. Section 7 evaluates the contention impact of MPI-ACC on concurrent GPU workloads. Section 8 summarizes our conclusions.

2 Motivation

In this section, we describe the issues in the traditional CPU-GPU application design and illustrate how GPU-integrated MPI can help alleviate them.

2.1 Designing MPI+GPU Applications

Graphics processing units have become more amenable to general-purpose computations over the past few years, largely as a result of the more programmable GPU hardware and increasingly mature GPU programming models, such as CUDA [6] and OpenCL [7]. Today's discrete GPUs reside on PCIe and are equipped with very high-throughput GDDR5 *device* memory on the GPU cards. To fully utilize the benefits of the ultra-fast memory subsystem, however, current GPU programmers must explicitly transfer data between the main memory and the device memory across PCIe, by issuing direct memory access (DMA) calls such as `cudaMemcpy` or `clEnqueueWriteBuffer`.

```
computation_on_GPU(gpu_buf);
cudaMemcpy(host_buf, gpu_buf, size, D2H ...);
MPI_Send(host_buf, size, ...);
```

(a) Basic hybrid MPI+GPU with synchronous execution – high productivity and low performance.

```
int processed[chunks] = {0};
for(j=0;j<chunks;j++) {
  computation_on_GPU(gpu_buf+offset, streams[j]);
  cudaMemcpyAsync(host_buf+offset, gpu_buf+offset,
                  D2H, streams[j], ...);
}
numProcessed = 0; j = 0; flag = 1;
while (numProcessed < chunks) {
  if(cudaStreamQuery(streams[j] == cudaSuccess) {
    MPI_Isend(host_buf+offset,...);/* start MPI */
    numProcessed++;
    processed[j] = 1;
  }
  MPI_Testany(...); /* check progress */
  if(numProcessed < chunks) /* find next chunk */
    while(flag) {
      j=(j+1)%chunks; flag=processed[j];
    }
}
MPI_Waitall();
```

(b) Advanced hybrid MPI+GPU with pipelined execution – low productivity and high performance.

```
for(j=0;j<chunks;j++) {
  computation_on_GPU(gpu_buf+offset, streams[j]);
  MPI_Isend(gpu_buf+offset, ...);
}
MPI_Waitall();
```

(c) GPU-integrated MPI with pipelined execution – high productivity and high performance.

Fig. 1. Designing hybrid CPU-GPU applications. For the manual MPI+GPU model with OpenCL, clEnqueueReadBuffer and clEnqueueWriteBuffer would be used in place of cudaMemcpy. For MPI-ACC, the code remains the same for *all* platforms (CUDA or OpenCL) and supported devices.

The Message Passing Interface is one of the most widely adopted parallel programming models for developing scalable, distributed-memory applications. MPI-based applications are typically designed by identifying parallel tasks and assigning them to multiple processes. In the default hybrid MPI+GPU programming model, the compute-intensive portions of each process are offloaded to the local GPU. Data is transferred between processes by explicit messages in MPI. However, the current MPI standard assumes a CPU-centric single-memory model for communication. The default MPI+GPU programming model employs a *hybrid* two-staged data movement model, where data copies are performed between main memory and the local GPU's device memory that are preceded and/or followed by MPI communication between the host CPUs (Figs. 1a and 1b). This is the norm seen in most GPU-accelerated MPI applications today [10], [12], [13]. The basic approach (Fig. 1a) has less complex code, but the blocking and staged data movement severely reduce performance because of the inefficient utilization of the communication channels. On the other hand, overlapped communication via pipelining efficiently utilizes all the communication channels but requires significant programmer effort, in other words, low productivity. Moreover, this approach leads to tight coupling between the high-level application logic and low-level data movement optimizations; hence, the application developer has to maintain several code variants for different GPU architectures and vendors. In addition, construction of such a sophisticated data movement scheme above the MPI runtime system incurs repeated protocol overheads and eliminates opportunities for low-level optimizations. Moreover, users who need high performance are faced with the complexity of leveraging a multitude of platform-specific optimizations that continue to evolve with the underlying technology (e.g, GPUDirect [8]).

2.2 Application Design Using GPU-Integrated MPI Frameworks

To bridge the gap between the disjointed MPI and GPU programming models, researchers have recently developed GPU-integrated MPI solutions such as our MPI-ACC [14] framework and MVAPICH-GPU [15] by Wang et al. These frameworks provide a unified MPI data transmission interface for both host and GPU memories; in other words, the programmer can use either the CPU buffer or the GPU buffer directly as the communication parameter in MPI routines. The goal of such GPU-integrated MPI platforms is to decouple the complex, low-level, GPU-specific data movement optimizations from the application logic, thus providing the following benefits: (1) portability—the application can be more portable across multiple accelerator platforms; and (2) forward compatibility—with the *same* code, the application can automatically achieve performance improvements from new GPU technologies (e.g., GPUDirect RDMA) if applicable and supported by the MPI implementation. In addition to enhanced programmability, transparent architecture-specific and vendor-specific performance optimizations can be provided within the MPI layer.

Using GPU-integrated MPI, programmers need only to write GPU kernels and regular host CPU codes for computation and invoke the standard MPI functions for CPU-GPU data communication, without worrying about the complex data movement optimizations of the diverse accelerator technologies (Fig. 1c).

3 Related Work

MVAPICH [16] is an implementation of MPI based on MPICH [17] and is optimized for RDMA networks such as InfiniBand. From v1.8 onward, MVAPICH has included support for transferring CUDA memory regions across the network (point-to-point, collective, and one-sided communications), but its design relies heavily on CUDA's Unified Virtual Addressing (UVA) feature. On the other hand, MPI-ACC takes a more portable approach: we support data transfers among CUDA, OpenCL, and CPU memory regions; and our design is independent of library version or device family. By including OpenCL support in MPI-ACC, we automatically enable data movement between a variety of devices, including GPUs from NVIDIA and AMD, IBM and Intel CPUs, Intel MICs, AMD Fusion, and IBM's Cell Broadband Engine. Also, we make no assumptions about the availability of key hardware features (e.g., UVA) in our interface design, thus making MPI-ACC a truly generic framework for heterogeneous CPU-GPU systems.

CudaMPI [18] and GMH [19] are new libraries that are designed to improve programmer productivity when managing data and compute among GPUs across the network. Both these approaches are host-centric programming models and provide new programming abstractions on top of existing MPI and CUDA calls. CaravelaMPI [20] is another MPI-style library solution for data management across GPUs, but the solution is limited to the custom Caravela API and not applicable to general MPI programs. In contrast, MPI-ACC completely conforms to the MPI standard itself, and our implementation removes the overhead of communication setup time, while maintaining productivity.

DCGN [21] is a device-centric programming model that moves away from the GPU-as-a-worker programming model. DCGN assigns ranks to GPU threads in the system and allows them to communicate among each other by using MPI-like library calls, and a CPU-based polling runtime handles GPU control and data transfers. MPI-ACC is orthogonal to DCGN in that we retain the host-centric MPI communication and execution model while hiding the details of third-party CPU-GPU communication libraries from the end user.

Partitioned Global Address Space (PGAS) models, such as Chapel, Global Arrays, and OpenSHMEM, provide a globally shared memory abstraction to distributed-memory systems. Researchers have explored extending PGAS models to include GPUs as part of the shared-memory abstraction [22], [23]. PGAS models can use MPI itself as the underlying runtime layer [24], [25] and can be considered as complementary efforts to MPI.

4 MPI-ACC: Design and Optimizations

In this section, we describe the design, implementation, and optimizations of MPI-ACC, the first interface that integrates CUDA, OpenCL, and other models within an MPI-compliant interface.

4.1 API Design

We discuss MPI-ACC's API design considerations and compare the tradeoffs of our solution with the the API design of MVAPICH. In an MPI communication call, the user passes a void pointer that indicates the location of the data on which the user wishes to operate. To the MPI library, a pointer to host memory is indistinguishable from a pointer to GPU memory. The MPI implementation needs a mechanism to determine whether the given pointer can be dereferenced directly or whether data must be explicitly copied from the device by invoking GPU library functions. Moreover, memory is referenced differently in different GPU programming models. For example, CUDA memory buffers are void pointers, but they cannot be dereferenced by the host unless Unified Virtual Addressing is enabled. On the other hand, OpenCL memory buffers are represented as opaque cl_mem handles that internally translate to the physical device memory location but cannot be dereferenced by the host unless the buffer is mapped into the host's address space or explicitly copied to the host.

MVAPICH's Automatic Detection Approach. MVAPICH allows MPI to deal with accelerator buffers by leveraging the UVA feature of CUDA to automatically detect device buffers. This method requires no modifications to the MPI interface. Also, we have shown previously that while their approach works well for standalone point-to-point

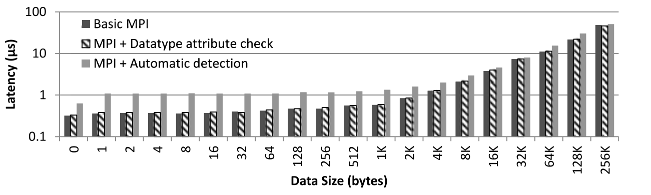


Fig. 2. Overhead of runtime checks incurred by intranode *CPU-CPU* communication operations. The slowdown due to automatic detection (via cuPointerGetAttribute) can be up to 205 percent (average: 127.8 percent), while the slowdown for the datatype attribute check is at most 6.3 percent (average: 2.6 percent).

communication, programmers have to *explicitly* synchronize between interleaved and dependent MPI and CUDA operations, thereby requiring significant programmer effort to achieve ideal performance [26]. Moreover, as shown in Fig. 2, the penalty for runtime checking can be significant and is incurred by all operations, including those that require no GPU data movement at all. Furthermore, the automatic detection approach is not extensible for other accelerator models such as OpenCL that do not map GPU buffers into the host virtual address space.

MPI-ACC's Datatype Attributes Approach. MPI datatypes are used to specify the type and layout of buffers passed to the MPI library. The MPI standard defines an interface for attaching metadata to MPI datatypes through datatype *attributes*. In MPI-ACC, we use these MPI datatype *attributes* to indicate buffer type (e.g., CPU, CUDA, or OpenCL), buffer locality (e.g., which GPU), the stream or the event to synchronize upon, or just any other information to the MPI library. With the datatype attributes, there is no restriction on the amount of information that the user can pass to the MPI implementation. With our design, one can simply *implicitly* denote ordering of MPI and GPU operations by associating GPU events or streams with MPI calls, and the MPI-ACC implementation applies different heuristics to synchronize and make efficient communication progress. We have shown in our prior work [26] that this approach improves productivity and performance, while being compatible with the MPI standard. Moreover, our approach introduces a lightweight runtime attribute check to each MPI operation, but the overhead is much less than with automatic detection, as shown in Fig. 2. Since MPI-ACC supports both CUDA and OpenCL and since OpenCL is compatible with several platforms and vendors, we consider MPI-ACC to be a more portable solution than MVAPICH. Table 1 summarizes the above differences.

TABLE 1
Comparison of MPI-ACC with MVAPICH

	MPI-ACC	MVAPICH
GPU Buffer Identification	MPI Datatype attributes	`void*` (UVA–based)
MPI+GPU Synch. Method	Implicit	Explicit
Performance/Progress for MPI+GPU Ops.	Automatic (through the MPI implementation)	Manual (through the programmer)
Software Platforms	CUDA and OpenCL (any version)	CUDA v4.0 and newer only
Hardware Platforms	AMD (CPU, GPU and APU), NVIDIA GPU, Intel (CPU and Xeon Phi), FPGA, etc	NVIDIA GPU only

4.2 Optimizations

Once MPI-ACC has identified a device buffer, it leverages PCIe and network link parallelism to optimize the data transfer via pipelining. Pipelined data transfer parameters are dynamically selected based on NUMA and PCIe affinity to further improve communication performance.

4.2.1 Data Pipelining

We hide the PCIe latency between the CPU and GPU by dividing the data into smaller chunks and performing pipelined data transfers between the GPU, the CPU, and the network. To orchestrate the pipelined data movement, we create a temporary pool of host-side buffers that are registered with the GPU driver (CUDA or OpenCL) for faster DMA transfers. The buffer pool is created at MPI_Init time and destroyed during MPI_Finalize. The system administrator can choose to enable CUDA and/or OpenCL when configuring the MPICH installation. Depending on the choice of the GPU library, the buffer pool is created by calling either cudaMallocHost for CUDA or clCreateBuffer (with the CL_MEM_ALLOC_HOST_PTR flag) for OpenCL.

To calculate the ideal pipeline packet size, we first individually measure the network and PCIe bandwidths at different data sizes (Fig. 3), then choose the packet size at the intersection point of the above channel rates, 64 KB for our experimental cluster (Section 6). If the performance at the intersection point is still latency bound for both data channels (network and PCIe), then we pick the pipeline packet size to be the size of the smallest packet at which the slower data channel reaches peak bandwidth. The end-to-end data transfer will then also work at the net peak bandwidth of the slower data channel. Also, only two packets are needed to do pipelining by double buffering: one channel receives the GPU packet to the host, while the other sends the previous GPU packet over the network. We therefore use two CUDA streams and two OpenCL command queues per device per MPI request to facilitate pipelining.

The basic pipeline loop for a "send" operation is as follows ("receive" works the same way, but the direction of the operations is reversed). Every time we prepare to send a packet over the network, we check for the completion of the previous GPU-to-CPU transfer by calling cudaStreamSynchronize or a loop of cudaStreamQuery for CUDA (or the corresponding OpenCL calls). However, we found that the GPU synchronization/query calls on *already completed* CPU-GPU copies caused a significant overhead in our experimental cluster, which hurt the effective network bandwidth and forced us to

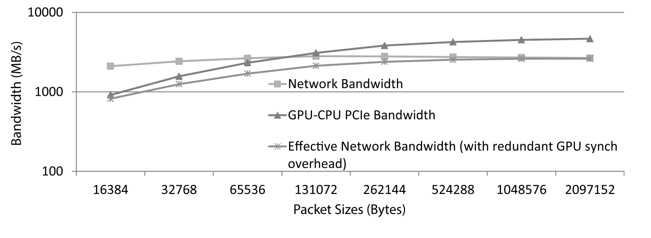


Fig. 3. Choosing the pipeline parameters: network—InfiniBand, transfer protocol – R3.

choose a different pipeline packet size. For example, we measured the cost of stream query/synchronization operations as approximately 20 μs, even though the data transfer had been completed. Moreover, this overhead occurs every time a packet is sent over the network, as shown in Fig. 3 by the "Effective Network Bandwidth" line. We observed that the impact of the synchronization overhead is huge for smaller packet sizes but becomes negligible for larger packet sizes (2 MB). Also, we found no overlap between the PCIe bandwidth and the effective network bandwidth rates, and the PCIe was always faster for all packet sizes. Thus, we picked the *smallest* packet size that could achieve the peak effective network bandwidth (in our case, 256 KB) as the pipeline transfer size for MPI-ACC. Smaller packet sizes ($<$256 KB) caused the effective network bandwidth to be latency bound and were thus not chosen as the pipeline parameters. In MPI-ACC, we use the pipelining approach to transfer large messages—namely, messages that are at least as large as the chosen packet size—and fall back to the nonpipelined approach when transferring smaller messages.

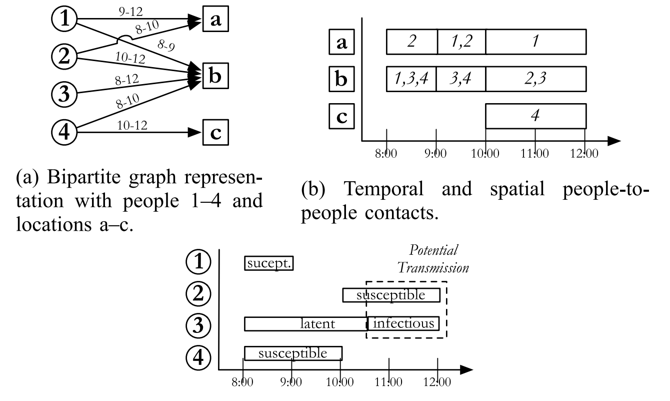
4.2.2 OpenCL Issues and Optimizations

In OpenCL, device data is encapsulated as a cl_mem object that is created by using a valid cl_context object. To transfer the data to/from the host, the programmer needs valid cl_device_id and cl_command_queue objects, which are all created by using the same context as the device data. At a minimum, the MPI interface for OpenCL communication requires the target OpenCL memory object, context, and device ID objects as parameters. The command queue parameter is optional and can be created by using the above parameters. Within the MPICH implementation, we either use the user-provided command queue or create several internal command queues for device-host data transfers. Within MPICH, we also create a temporary OpenCL buffer pool of pinned host-side memory for pipelining. However, OpenCL requires that the internal command queues and the pipeline buffers also be created by using the same context as the device data. Moreover, in theory, the OpenCL context could change for every MPI communication call, and so the internal OpenCL objects cannot be created at MPI_Init time. Instead, they must be created at the beginning of every MPI call and destroyed at the end of it.

The initialization of these temporary OpenCL objects is expensive, and their repeated usage severely hurts performance. We cache the command queue and pipeline buffer objects after the first communication call and reuse them if the same OpenCL context and device ID are used for the subsequent calls, which is a plausible scenario. If any future call involves a different context or device ID, we clear and replace our cache with the most recently used OpenCL objects. In this way, we can amortize the high OpenCL initialization cost across multiple calls and significantly improve performance. We use a caching window of one, which we consider to be sufficient in practice.

5 Application Case Studies

In this section, we perform an in-depth analysis of the default MPI+GPU application design in scientific applications from computational epidemiology and seismology modeling. We identify the inherent data movement inefficiencies and show how MPI-ACC can be used to explore new design spaces and create novel application-specific optimizations.



(a) Bipartite graph representation with people 1–4 and locations a–c.

(b) Temporal and spatial people-to-people contacts.

(c) Occupancy of location b overlaid with the health states of its occupants. The dotted area shows the time of possible disease transmission.

Fig. 4. Computational epidemiology simulation model (figure adapted from [9]).

5.1 EpiSimdemics

GPU-EpiSimdemics [9], [10] is a high-performance, agent-based simulation program for studying the spread of epidemics through large-scale social contact networks and the coevolution of disease, human behavior, and the social contact network. The participating entities in GPU-EpiSimdemics are *persons* and *locations*, which are represented as a bipartite graph (Fig. 4a) and interact with one another iteratively over a predetermined number of iterations (or simulation days). The output of the simulation is the relevant disease statistics of the contagion diffusion, such as the total number of infected persons or an infection graph showing who infected whom and the time and location of the infection.

5.1.1 Phases

Each iteration of GPU-EpiSimdemics consists of two phases: *computeVisits* and *computeInteractions*. During the *computeVisits* phase, all the person objects of every processing element (or PE) first determine the schedules for the current day, namely, the locations to be visited and the duration of each visit. These *visit* messages are sent to the destination location's host PE (Fig. 4a). Computation of the schedules is overlapped with communication of the corresponding visit messages.

In the *computeInteractions* phase, each PE first groups the received visit messages by their target locations. Next, each PE computes the probability of infection transmission between every pair of spatially and temporally colocated people in its local location objects (Fig. 4b), which determines the overall disease spread information of that location. The infection transmission function depends on the current health states (e.g., susceptible, infectious, latent) of the people involved in the interaction (Fig. 4c) and the transmissibility factor of the disease. These *infection* messages are sent back to the "home" PEs of the infected persons. Each PE, upon receiving its infection messages, updates the health states of the infected individuals, which will influence their schedules for the following simulation day. Thus, the messages that are computed as the output of one phase

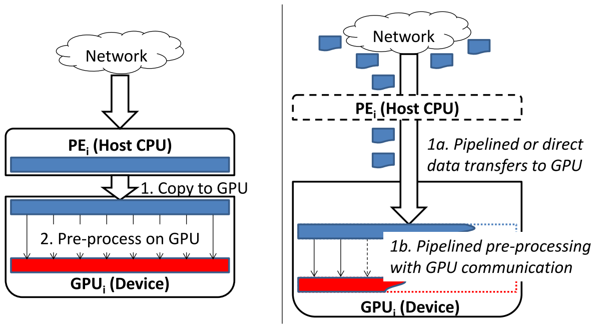


Fig. 5. Exclusive GPU computation mode of GPU-EpiSimdemics. Left: manual MPI+CUDA design. Right: MPI-ACC–enabled design, where the *visit* messages are transferred and preprocessed on the device in a pipelined manner.

are transferred to the appropriate PEs as inputs of the next phase of the simulation. The system is synchronized by barriers after each simulation phase.

5.1.2 Computation-Communication Patterns and MPI-ACC-Driven Optimizations

In GPU-EpiSimdemics, each PE in the simulation is implemented as a separate MPI process. Also, the *computeInteractions* phase of GPU-EpiSimdemics is offloaded and accelerated on the GPU while the rest of the computations are executed on the CPU [10]. The current implementation of GPU-EpiSimdemics assumes one-to-one mapping of GPUs to MPI processes. In accordance with the GPU-EpiSimdemics algorithm, the output data elements from the *computeVisits* phase (i.e., visit messages) are first received over the network, then merged, grouped, and preprocessed before the GPU can begin the *computeInteractions* phase of GPU-EpiSimdemics. Moreover, there are two GPU computation modes depending on how the visit messages are processed on the GPUs. In this paper, we discuss the exclusive GPU computation mode, but discussion of the cooperative CPU-GPU computation mode can be found in our prior work [27].

Preprocessing phase on the GPU. As a preprocessing step in the *computeInteractions* phase, we modify the data layout of the visit messages to be more amenable to the massive parallel architecture of the GPU [10]. Specifically, we unpack the visit message structures to a 2D time-bin matrix, where each row of the matrix represents a person-location pair and the cells in the row represents fixed time slots of the day: that is, each visit message corresponds to a single row in the person-timeline matrix. Depending on the start time and duration of a person's visit to a location, the corresponding row cells are marked as *visited*. The preprocessing logic of data unpacking is implemented as a separate GPU kernel at the beginning of the *computeInteractions* phase. The matrix data representation enables a much better SIMDization of the *computeInteractions* code execution, which significantly improves the GPU performance. However, we achieve the benefits at the cost of a larger memory footprint for the person-timeline matrix, as well as a computational overhead for the data unpacking.

Basic MPI+GPU communication-computation pipeline. In the naïve data movement approach, in each PE, we receive the visit messages in the main memory during the *computeVisits* phase, transfer the aggregate data to the local GPU (device) memory across the PCIe bus, and then begin the preprocessing step of the *computeInteractions* phase (Fig. 5). While all PEs communicate the visit messages with every other PE, the number of pairwise visit message exchanges is not known beforehand; in other words, it is not a typical collective all-to-all or a scatter/gather operation. On the other hand, each PE preallocates CPU buffer fragments and registers them as *persistent* receive requests with the MPI library by using the `MPI_Recv_init` call to enable persistent point-to-point communication among the PEs. Moreover, each PE has to create persistent CPU receive buffers corresponding to every other participating PE in the simulation; that is, receive buffers increase with number of PEs and cannot be reused from a constant buffer pool.

Whenever a buffer fragment is received into the corresponding receive buffer, it is copied into a contiguous visit vector in the CPU's main memory. The *computeInteractions* phase of the simulation then copies the aggregated visit vector to the GPU memory for preprocessing. While the CPU-CPU communication of visit messages is somewhat overlapped with their computation on the source CPUs, the GPU and the PCIe bus will remain idle until the visit messages are completely received, merged, and ready to be transferred to the GPU.

Advanced MPI+GPU communication-computation pipeline. In this optimization, we still preallocate persistent receive buffers *on the CPU* in each PE and register them with the MPI library as the communication endpoints by calling `MPI_Recv_init`. But, we create the contiguous visit vector in GPU memory, so that whenever a PE receives a visit buffer fragment on the CPU, we immediately enqueue an asynchronous CPU-GPU data transfer to the contiguous visit vector and also enqueue the associated GPU preprocessing kernels, thereby manually creating the communication-computation pipeline (Fig. 5).

In order to enable asynchronous CPU-GPU data transfers, however, the persistent receive buffers must be nonpageable (pinned) memory. Also, since the number of receive buffers increases with the number of PEs, the pinned memory footprint also increases with the number of PEs. This design reduces the available pageable CPU memory, a situation that could lead to poor CPU performance [6].

The pinned memory management logic can be implemented at the application level in a couple of ways. In the first approach, the pinned memory pool is created before the *computeVisits* phase begins and is destroyed once the phase finishes, but the memory management routines are invoked every simulation iteration. While this approach is relatively simple to implement, repeated memory management leads to significant performance overhead. In the second approach, the pinned memory pool is created once, before the main simulation loop, and is destroyed after the loop ends, thus avoiding the performance overhead of repeated memory management. However, this design reduces the available pageable CPU memory, not only for the *computeVisits* phase, but also for the other phases of the simulation, including *computeInteractions*. We discuss the performance tradeoffs of these two memory management techniques in Section 6.2.

MPI-ACC–enabled communication-computation pipeline. Since MPI-ACC handles both CPU and GPU buffers, in each PE we preallocate persistent buffer fragments *on the*

GPU and register them with the MPI library by using `MPI_Recv_init`. Without MPI-ACC's GPU support, one cannot create persistent buffers on the GPU as MPI communication endpoints. In this approach, the receive buffers *on the GPU* increase with number of PEs, and the approach does not require a growing pinned memory pool on the host. Furthermore, MPI-ACC internally creates a *constant* pool of pinned memory during `MPI_Init` and automatically reuses it for all pipelined data communication, thereby providing better programmability and application scalability. Internally, MPI-ACC may either pipeline the internode CPU-GPU data transfers via the host CPU's memory or use direct GPU transfer techniques (e.g., GPUDirect RDMA), if possible; but these details are hidden from the programmer.

Along with the preallocated persistent GPU buffer fragments, the contiguous visit vector is created in the GPU memory itself. As soon as a PE receives the visit buffer fragments on the GPU, we enqueue kernels to copy data within the device to the contiguous visit vector and also enqueue the associated GPU preprocessing kernels, thereby creating the end-to-end communication-computation pipeline. Thus, we completely overlap the visit data generation on the CPU with internode CPU-GPU data transfers and GPU preprocessing. In this way, the preprocessing overhead is completely hidden and removed from the *computeInteractions* phase. Moreover, the CPU, GPU, and the interconnection networks are all kept busy, performing either data transfers or the preprocessing execution.

MPI-ACC's internal pipelined CPU-GPU data transfer largely hides the PCIe transfer latency during the *computeVisits* phase. However, it still adds a non-negligible cost to the overall communication time when compared with the CPU-CPU data transfers of the default MPI+GPU implementation. Nevertheless, our experimental results show that the gains achieved in the *computeInteractions* phase due to the preprocessing overlap outweigh the communication overheads of the *computeVisits* phase for all system configurations and input data sizes.

In summary, MPI-ACC helps the programmer focus on the high-level application logic by enabling automatic and efficient low-level memory management techniques. Moreover, MPI-ACC exposes a natural interface to communicate with the target device (CPU or GPU), without treating CPUs as explicit communication relays.

5.2 FDM-Seismology

FDM-Seismology is our MPI+GPU hybrid implementation of an application that models the propagation of seismological waves using the finite-difference (FD) method by taking the Earth's velocity structures and seismic source models as input [11]. The application implements a parallel velocity-stress, staggered-grid finite-difference method for propagation of waves in a layered medium. In this method, the domain is divided into a three-dimensional grid, and a one-point-integration scheme is used for each grid cell. Since the computational domain is truncated in order to keep the computation tractable, absorbing boundary conditions (ABCs) are placed around the region of interest so as to keep the reflections minimal when boundaries are impinged by the outgoing waves. This strategy helps simulate unbounded domains. In our application, PML (perfectly matched layers) absorbers [28] are being used as ABCs for their superior efficiency and minimal reflection coefficient. The use of a one-point integration scheme leads to an easy and efficient implementation of the PML absorbing boundaries and allows the use of irregular elements in the PML region [11].

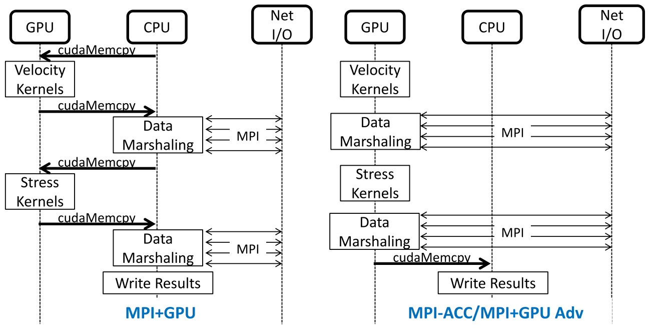


Fig. 6. Communication-computation pattern in the FDM-Seismology application. Left: basic MPI+GPU execution mode with data marshaling on CPU. Right: execution modes with data marshaling on GPU. MPI-ACC automatically communicates the GPU data; the MPI+GPU Adv case explicitly stages the communication via the CPU.

5.2.1 Computation-Communication Patterns

The simulation operates on the input finite-difference model and generates a three-dimensional grid as a first step. Our MPI-based parallel version of the application divides the input FD model into submodels along different axes such that each submodel can be computed on different CPUs (or nodes). This domain decomposition technique helps the application scale to a large number of nodes. Each processor computes the velocity and stress wavefields in its own subdomain and then exchanges the wavefields with the nodes operating on neighbor subdomains, after each set of velocity or stress computation (Fig. 6). Each processor updates its own wavefields after receiving all its neighbors' wavefields.

These computations are run multiple times for better accuracy and convergence of results. In every iteration, each node computes the velocity components followed by the stress components of the seismic wave propagation. The wavefield exchanges with neighbors take place after each set of velocity and stress computations. This MPI communication takes place in multiple stages wherein each communication is followed by an update of local wavefields and a small postcommunication computation on local wavefields. At the end of each iteration, the updated local wavefields are written to a file.

The velocity and stress wavefields are stored as large multidimensional arrays on each node. In order to optimize the MPI computation between neighbors of the FD domain grid, only a few elements of the wavefields, those needed by the neighboring node for its own local update, are communicated to the neighbor, rather than whole arrays. Hence, each MPI communication is surrounded by data-marshaling steps, where the required elements are packed into a smaller array at the source, communicated, and then unpacked at the receiver in order to update its local data.

5.2.2 GPU Acceleration of FDM-Seismology

We describe a couple of GPU execution modes of FDM-Seismology.

MPI+GPU with data marshaling on CPU (MPI+GPU). Our GPU-accelerated version of FDM-Seismology performs the velocity and stress computations as GPU kernels. In order to transfer the wavefields to other nodes, it first copies the bulk data from the GPU buffers to CPU memory over the PCIe bus and then transfers the individual wavefields over MPI to the neighboring nodes (Fig. 6). All the data-marshaling operations and small postcommunication computations are performed on the CPU itself. The newly updated local wavefields that are received over MPI are then bulk transferred back to the GPU before the start of the next stress or velocity computation on the GPU.

MPI+GPU with data marshaling on GPU (MPI+GPU Adv). In this execution mode, the data-marshaling operations are moved to the GPU to leverage the faster GDDR5 memory module and the massively parallel GPU architecture. Consequently, the CPU-GPU bulk data transfers before and after each velocity-stress computation kernel are completely avoided. The need to explicitly bulk transfer data from the GPU to the CPU arises only at the end of the iteration, when the results are transferred to the CPU to be written to a file (Fig. 6).

5.2.3 *MPI-ACC-Enabled Optimizations*

GPU-based data marshaling suffers from the following disadvantage in the absence of GPU-integrated MPI. All data-marshaling steps are separated by MPI communication, and each data-marshaling step depends on the previously marshaled data *and* the received MPI data from the neighbors. In other words, after each data-marshaling step, data has to be explicitly moved from the GPU to the CPU only for MPI communication. Similarly, the received MPI data has to be explicitly moved back to the GPU before the next marshaling step. In this scenario, the application uses the CPU only as a communication relay. If the GPU communication technology changes (e.g., GPUDirect RDMA), we will have to largely rewrite the FDM-Seismology communication code to achieve the expected performance.

With MPI-ACC as the communication library, we still perform data marshaling on the GPU, but we communicate the marshaled data directly to and from the GPU without explicitly using the CPU for data staging. Also, the bulk transfer of data still happens only once at the end of each iteration, in order to write the results to a file. But, the data-marshaling step happens multiple times during a single iteration; and consequently the application launches a series of GPU kernels. While consecutive kernels entail launch and synchronization overhead per kernel invocation, the benefits of faster data marshaling on the GPU and optimized MPI communication outweigh the kernel overheads.

Other than the benefits resulting from GPU-driven data marshaling, a GPU-integrated MPI library benefits the FDM-Seismology application in the following ways: (1) it significantly enhances the productivity of the programmer, who is no longer constrained by the fixed CPU-only MPI communication and can easily choose the appropriate device as the communication target end-point; (2) the pipelined data transfers within MPI-ACC further improve the communication performance over the network; and (3) regardless of the GPU communication technology that

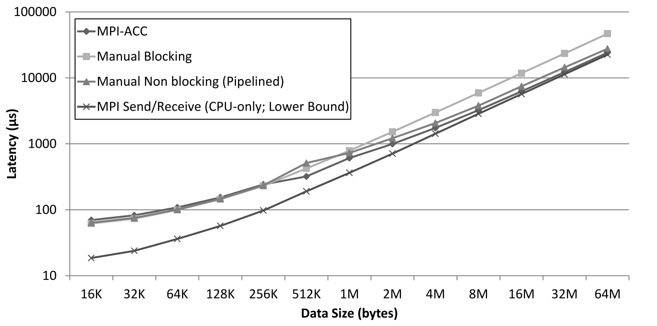


Fig. 7. Internode communication latency for GPU-to-GPU (CUDA) data transfers, InfiniBand transfer protocol: R3. Similar performance is observed for OpenCL data transfers. The chosen pipeline packet size for MPI-ACC is 256 KB.

may become available in the future, our MPI-ACC–driven FDM-Seismology code will not change and will automatically benefit from the performance upgrades that are made available by the subsequent GPU-integrated MPI implementations (e.g., support for GPU-Direct RDMA).

6 Evaluation

In this section, we describe our experimental setup followed by the performance evaluation of MPI-ACC via latency microbenchmarks. Next, we demonstrate the efficacy of the MPI-ACC–enabled optimizations in GPU-EpiSimdemics and FDM-Seismology. Using both microbenchmarks and GPU-EpiSimdemics, we discuss the impact of shared resource (hardware and software) contention on MPI-ACC's communication performance.

We conducted our experiments on *HokieSpeed*, a state-of-the-art, 212-teraflop hybrid CPU-GPU supercomputer housed at Virginia Tech. Each HokieSpeed node contains two hex-core Intel Xeon E5645 CPUs running at 2.40 GHz and two NVIDIA Tesla M2050 GPUs. The host memory capacity is 24 GB, and each GPU has a 3 GB device memory. The internode interconnect is QDR InfiniBand. We used up to 128 HokieSpeed nodes and both GPUs per node for our experiments. We used the GCC v4.4.7 compiler and CUDA v5.0 with driver version 310.19.

6.1 Microbenchmark Analysis

Impact of Pipelined Data Transfer: In Fig. 7, we compare the performance of MPI-ACC with the manual blocking and manual pipelined implementations. Our internode GPU-to-GPU latency tests show that MPI-ACC is better than the manual blocking approach by up to 48.3 percent and is up to 18.2 percent better than the manual pipelined implementation, especially for larger data transfers. The manual pipelined implementation repeatedly invokes MPI calls, causing multiple handshake messages to be sent back and forth across the network and thus hurting performance. On the other hand, we perform the handshake only once in MPI-ACC to establish the send-receiver identities, followed by low-overhead pipelining. We perform pipelining in MPI-ACC only for messages that are larger than the pipeline packet size, and we fall back to the default blocking approach for smaller data sizes. Hence, we see that the performance of MPI-ACC is comparable to the manual blocking approach for smaller message sizes.

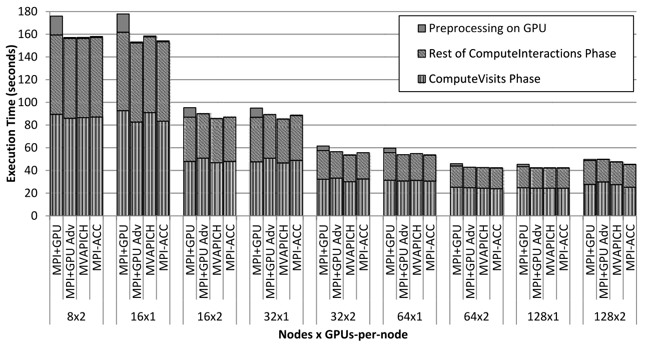


Fig. 8. Execution profile of GPU-EpiSimdemics over various node configurations.

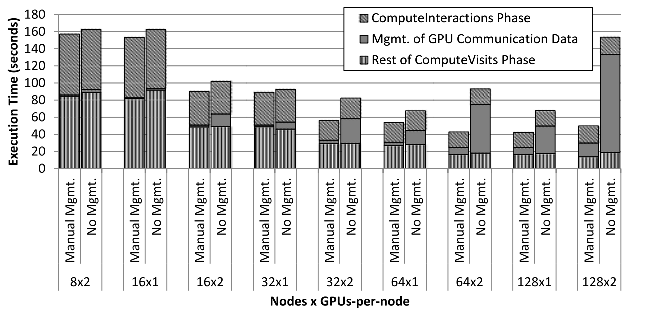


Fig. 9. Analysis of the data management complexity versus performance tradeoffs. Manual data management achieves better performance at the cost of high code complexity. The case with no explicit data management has simpler code but performs poorly.

Impact of OpenCL Object Caching. Our OpenCL caching optimization improves the internode GPU-to-GPU communication latency from 3 percent for larger data sizes (64 MB) to 88.7 percent for smaller data sizes ($<$ 256 KB). Even where the programmers provide their custom command queue, the pipeline buffers still have to be created for every MPI communication call; hence, caching improves performance.

6.2 Case Study Analysis: EpiSimdemics

We compare the combined performance of all the phases of GPU-EpiSimdemics (*computeVisits* and *computeInteractions*), with and without the MPI-ACC–driven optimizations discussed in Section 5.1.2. We choose different-sized input datasets of synthetic populations from two U.S. states: Washington (WA), with a population of 5.7 million, and California (CA), with a population of 33.1 million. In this paper, we present results and detailed analysis of WA; analysis of CA is described in our prior work [27]. We also vary the number of compute nodes from 8 to 128 and the number of GPU devices between 1 and 2. We begin from the smallest node-GPU configuration that can fit the entire problem in the available GPU memory. We also compare the application performance when MVAPICH (v2.0a) is used as the GPU-aware MPI implementation of choice.

Our results in Fig. 8 indicate that our MPI-ACC–driven optimizations perform better than the basic blocking MPI +GPU implementations by an average of 9.2 percent and by up to 13.3 percent for WA. The performance of the MPI-ACC–driven solution is similar to the performance of the MVAPICH-based and the manual MPI+GPU (advanced) implementations. Since the CPU-GPU transfer is not a bottleneck in GPU-EpiSimdemics, the specific data pipelining logic of either MPI-ACC or MVAPICH does not directly affect the performance gains. On the other hand, the preprocessing step (data unpacking) of the *computeInteractions* phase is completely overlapped with the asynchronous CPU-to-remote-GPU communication, for all node configurations. Note that the advanced MPI+GPU implementation uses the manual pinned memory management techniques that we implemented at the application level, which achieves better performance but with a much more complex code.

For larger node configurations, the local operating dataset in the *computeInteractions* phase becomes smaller, and hence that the basic MPI+GPU solution takes less time to execute the preprocessing stage; in other words, the absolute gains achieved by hiding the preprocessing step get diminished for GPU-EpiSimdemics. However, we have shown that MPI-ACC can enable the developer to create newer optimizations for better latency hiding and resource utilization.

Data management complexity versus performance tradeoffs. While the advanced MPI+GPU implementation achieved comparable performance to the MPI-ACC–based solution, it put the burden of explicit data management on the application programmer. We discussed in Section 5.1.2 that, on the other hand, the user can write simpler code and avoid explicit data management but has to repeatedly create and destroy the receive buffers for every simulation iteration, thereby losing performance. Fig. 9 shows the performance tradeoffs of the two approaches. We observe that explicit data management is better for all node configurations and can achieve up to $4.5\times$ performance improvement. Without data management, the pinned memory footprint of the receive buffers increases with the number of MPI processes, thereby entailing bigger performance losses for larger nodes. To quantify the degree of performance loss, we measured the individual memory allocation costs using simple microbenchmarks and found that CUDA's pinned memory allocator (`cudaMallocHost`) was about 26 percent slower than the vanilla CPU memory allocator (`malloc`) for single CUDA contexts. We also observed that the pinned memory allocation cost increased linearly with the number of GPUs or CUDA contexts, whereas memory management in multiple processes and CUDA contexts should ideally be handled independently in parallel. Consequently, in Fig. 9, we see that for the same number of MPI processes, the node configuration with two MPI processes (or GPUs) per node performs worse than the node with a single MPI process; for example, the 64×2 configuration is slower than the 128×1 one. Thus, efficient pinned memory management is essential for superior performance, and MPI-ACC provides that automatically to the programmers.

Discussion. The basic MPI+GPU solution has preprocessing overhead but does not have significant memory management issues. While the advanced MPI+GPU implementation gains from hiding the preprocessing overhead, it loses from either nonscalable pinned memory management or poor programmer productivity. On the other hand, MPI-ACC provides a more scalable solution by (1) automatically managing a fixed-size pinned buffer pool for pipelining and (2) creating buffer pools just once at `MPI_Init` and destroying them at `MPI_Finalize`. MPI-ACC thus gains from both hiding the preprocessing overhead and efficient pinned memory management. MPI-ACC decouples the lower-level

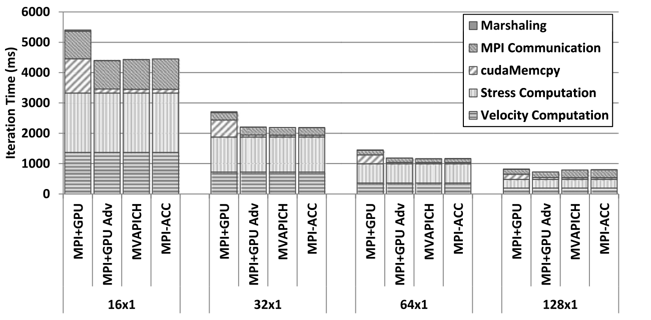


Fig. 10. Analyzing the FDM-Seismology application with the larger input data (Dataset-2). Note that MPI Communication refers to CPU-CPU data transfers for the MPI+GPU and MPI+GPU Adv cases and GPU-GPU (pipelined) data transfers for MPI-ACC and MVAPICH. The performance difference between MPI-ACC(CUDA) and MPI-ACC(OpenCL) is negligible and only the MPI-ACC(CUDA) result is shown in this figure.

memory management logic from the high-level simulation implementation, thereby enabling both performance and productivity.

6.3 Case Study Analysis: FDM-Seismology

In this section, we analyze the performance of the different phases of the FDM-Seismology application and evaluate the effect of MPI-ACC and MVAPICH on the application. FDM-Seismology is implemented with both CUDA and OpenCL, and while MPI-ACC is evaluated using both CUDA and OpenCL, MVAPICH can be evaluated only with the CUDA-based code. We vary the nodes from 2 to 128 with 1 GPU per node and use small and large datasets as input. Our scalability experiments begin from the smallest number of nodes required to fit the given data in the device memory. For the larger input data (i.e., Dataset-2), the size of the MPI transfers increases by $2\times$, while the size of data to be marshaled increases by $4\times$ when compared with the smaller Dataset-1.

Fig. 10 shows the performance of the FDM-Seismology application, with and without the GPU-based data marshaling. We report the average wall-clock time across all the processes because the computation-communication costs vary depending on the virtual location of the process in the application's structured grid representation. The application's running time is composed mainly of velocity and stress computations (>60 percent) and does not change for the three application designs.

In the basic MPI+GPU case, we perform both data-marshaling operations and MPI communication from the CPU. Thus, the application has to move large wavefield data between the CPU and the GPU for data marshaling and MPI communication after every stress and velocity computation phase over every iteration. In the MPI+GPU Adv, MVAPICH, and MPI-ACC–driven scenarios, we perform data marshaling on the GPU itself; hence, smaller-sized wavefield data is transferred from the GPU to the CPU only once per iteration for output generation. By performing data marshaling on the GPU, we avoid the large bulk CPU-GPU data transfers and improve the overall performance over the basic MPI+GPU design with data marshaling on the CPU. Data marshaling on the GPU provides performance gains while MPI-ACC improves programmer productivity by directly communicating the GPU buffers (CUDA/OpenCL) in the application.

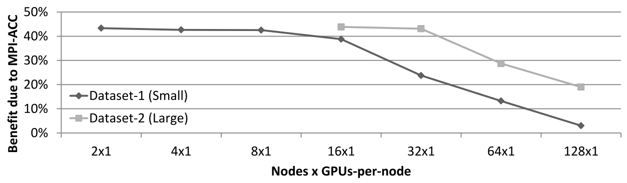


Fig. 11. Scalability analysis of FDM-Seismology application with two datasets of different sizes. The baseline for speedup is the naïve MPI+GPU programming model with CPU data marshaling.

Scalability analysis. Fig. 11 shows the performance improvement due to the MPI-ACC–enabled GPU data marshaling strategy over the basic MPI+GPU implementation with CPU data marshaling. We see that the performance benefits due to the GPU data marshaling decrease with increasing number of nodes, because of the following reasons. For a given dataset, the per-node data size decreases with increasing number of nodes. Thus, the costly CPU-GPU bulk data transfers are reduced (Fig. 10), and the overall benefits of GPU-based data marshaling itself are minimized. Also, for a larger number of nodes, the application's MPI communication cost becomes significant when compared with the computation and data marshaling costs. In such a scenario, the CPU-to-CPU MPI communication of the MPI+GPU and MPI+GPU Adv implementations will have less overhead than does the pipelined GPU-to-GPU MPI communication of the MPI-ACC–enabled design. If newer technologies such as GPUDirect-RDMA are integrated into MPI, we can expect the GPU-to-GPU communication overhead to be reduced, but the overall benefits of GPU data marshaling itself will still be limited because of the reduced per-process working set.

7 Analysis of Contention

In this section, we provide some insights into the scalable design of MPI-ACC. Specifically, we show that MPI-ACC is designed to work concurrently with other existing GPU workloads with minimum contention; that is, one should be able to perform MPI-ACC GPU-GPU communication and other user-specified GPU tasks (kernel or data transfers) simultaneously with minimum performance degradation for both tasks. We analyze the contention effects of MPI-ACC, MVAPICH and manual MPI+GPU on concurrent GPU and PCIe workloads.

Sources of contention. NVIDIA Fermi GPUs have one hardware queue each for enqueueing GPU kernels, D2H data transfers, and H2D data transfers. Operations on different hardware queues can potentially overlap. For example, a GPU kernel can overlap with H2D and D2H transfers simultaneously. However, operations enqueued to the same hardware queue will be processed serially. If a GPU task oversubscribes a hardware queue by aggressively enqueueing multiple operations of the same type, then it can severely slow other GPU tasks contending to use the same hardware queue.

GPU streams are software workflow abstractions for a sequence of operations that execute in issue-order on the GPU. Stream operations are directed to the appropriate hardware queue depending on the operation type. Operations from different streams can execute concurrently and

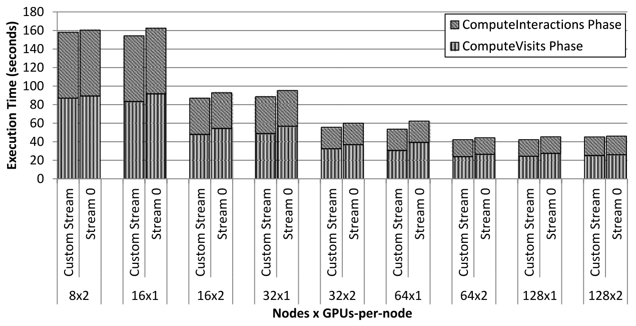


Fig. 12. Characterizing the contention impact of CUDA's stream-0 in GPU-EpiSimdemics.

may be interleaved, whereas operations within the same stream are processed serially, leading to software contention.

In summary, contention among GPU operations can be of two types: hardware contention, where one or more hardware queues of the GPU are oversubscribed by the same type of operation, or software contention, where different types of operations may be issued but to the same GPU stream. In MPI-ACC, we have carefully minimized both types of contention by staggered enqueueing of H2D and D2H operations to different GPU streams, thereby enabling maximum concurrency.

Microbenchmark design. We extended the SHOC benchmark suite's `contention-mt` application for the microbenchmark study. The benchmark creates two MPI processes, each on a separate node and controlling the two local GPUs. Each MPI process is also dual-threaded and concurrently runs one task per thread, where task-0 by thread-0 does point-to-point GPU-to-GPU *MPI* communication with the other process and task-1 by thread-1 executes local non-MPI GPU tasks, such as compute kernels or H2D and D2H data transfer loops. CUDA allows the same GPU context to be shared among all the threads (tasks) in the process. We share the local GPU between both tasks. To measure the contention impact, we first execute tasks 0 and 1 independently without contention and then execute them concurrently to induce contention. Each task is run for 100 loop iterations for both the independent and concurrent runs. We measure and report the performance difference between the tasks' independent and concurrent runs as the incurred contention.

7.1 Discussion of Software Contention

CUDA's stream-0 (default stream) is unique in that it is completely ordered with all operations issued on any stream of the device. That is, issuing operations on stream-0 would be functionally equivalent to synchronizing the entire device before and after each operation. Although MPI-ACC privately creates and uses custom streams to minimize software contention with other streams, a concurrent user operation to stream-0 can inadvertently stall any MPI-ACC operation on that GPU until stream-0 has completed. On the other hand, OpenCL does not have special queues and does not incur software contention.

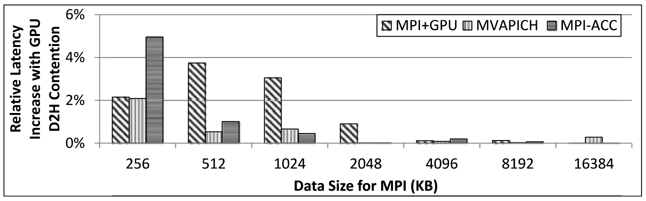
Contention due to stream-0 can be seen even in GPU-EpiSimdemics, and we analyze its effect as follows. In GPU-EpiSimdemics, the internode CPU-GPU communication of the visit messages is overlapped with a preprocessing kernel that performs data layout transformation (Section 5.1). While we use non-0 streams within MPI-ACC for the internode communication of visit messages, the preprocessing kernel may be launched with the user's chosen CUDA stream. Fig. 12 shows that the performance of GPU-EpiSimdemics is about 6.6 percent slower when the preprocessing kernels use stream-0 instead of a non-0 stream, and the slowdown can be up to 16.3 percent for some node configurations. While MPI-ACC's streams are designed to scale, a poor application design using stream-0 can cause an apparent slowdown in MPI-ACC's data transfer performance.

7.2 Minimizing the Hardware Contention

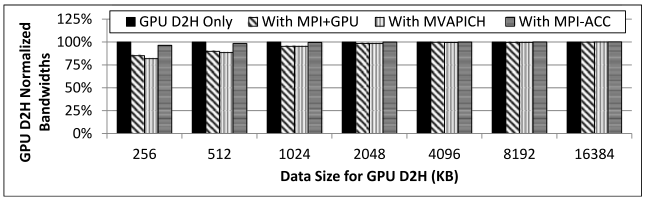
MPI-ACC uses the D2H and H2D hardware queues of the GPU for send and receive, respectively. In theory, MPI-ACC communication can overlap with kernel invocations or other data transfer operations in the opposite direction, that is, using the other data transfer queue. However, MPI-ACC can cause contention with another data transfer operation in the same direction. For example, `MPI_Send` can contend with a concurrent D2H data transfer. MPI-ACC operations can also potentially contend with the on-device memory controller. For example, `MPI_Send` or `MPI_Recv` can slow a global-memory-intensive kernel that is accessing the same memory module. In this section, we quantify and evaluate the global memory and PCIe contention effects.

Global memory contention analysis. We study the impact of global memory contention by executing MPI-ACC operations in task-0 and global memory read/write benchmarks in task-1 with custom CUDA streams. Our experiments indicate that the performance drop due to contention in the MPI-ACC communication is about 4 percent, whereas the global memory kernels slow by about 8 percent. The average MPI-ACC call runs longer than an average global memory access, so MPI-ACC has less relative performance reduction. On the other hand, the performance impact of MPI-ACC on on-chip (local) memory accesses and simple computational kernels using custom CUDA streams is less, where the performance degradation in the MPI-ACC communication is about 3 percent and the GPU workloads do not have any noticeable slowdown. Because of space constraints, we omit explicit performance graphs.

PCIe contention analysis with data transfers in the opposite direction. We study the impact of PCIe contention by having task-0 perform `MPI_Send` or `MPI_Recv` communication operations with GPU-0, while task-1 executes H2D or D2H calls. This approach gives four different task combinations, of which two combinations perform bidirectional data transfers and two combinations transfer data in the same direction. In this paper, we report the results by running `MPI_Send` (task-0) concurrently with H2D and D2H transfers on the same GPU (task-1). The contention analysis of `MPI_Recv` with H2D and D2H transfers is identical, and we exclude it because of space constraints. If task-0 and task-1 perform bidirectional data transfers and use custom CUDA streams, then we find that the average slowdown of task-0 is 6 percent and the H2D task (task-1) has a negligible slowdown. Ideally, if the bidirectional bandwidth were to be twice the unidirectional bandwidth, then both the concurrent tasks would have no slowdown. In our experimental platform, however, the bidirectional bandwidth is only about 19.3 percent more than the unidirectional bandwidth

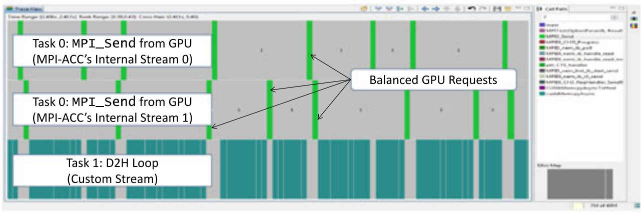


(a) Impact on the `MPI_Send` communication latency. Task-1 (D2H) transfers 16MB for all cases.

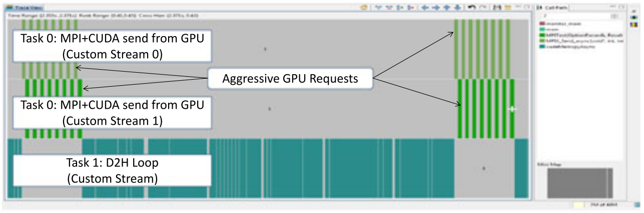


(b) Impact on the local D2H GPU operations. Task-0 (MPI) transfers 16MB for all cases. MPI-ACC causes least contention for the D2H task.

Fig. 13. Impact of contention due to concurrent `MPI_Send` and local D2H GPU operations. MPI-ACC is evaluated against MVAPICH and the manual MPI+GPU implementations.


Task 0: MPI_Send from GPU (MPI-ACC's Internal Stream 0)
Balanced GPU Requests
Task 0: MPI_Send from GPU (MPI-ACC's Internal Stream 1)
Task 1: D2H Loop (Custom Stream)

(a) Impact of MPI-ACC's `MPI_Send` with concurrent D2H operations.


Task 0: MPI+CUDA send from GPU (Custom Stream 0)
Aggressive GPU Requests
Task 0: MPI+CUDA send from GPU (Custom Stream 1)
Task 1: D2H Loop (Custom Stream)

(b) Impact of manual MPI+GPU send task with concurrent D2H operations.

Fig. 14. Using HPCToolkit to understand the contention impacts of MPI-ACC and local GPU data transfer operations.

according to the `simpleMultiCopy` CUDA SDK benchmark. Thus, task-0's slowdown is due to slower bidirectional bandwidth and not due to any possible MPI-ACC–related contention effects.

PCIe contention analysis with data transfers in the same direction. For this study, we analyze contention effects when `MPI_Send` (task-0) is invoked concurrently with standalone D2H transfers on the same GPU (task-1). We analyze the contention impacts of three `MPI_Send` implementations: MPI-ACC, MVAPICH, and manual pipelining using asynchronous MPI and CUDA. Since the Fermi GPUs have a single data transfer hardware queue in each direction, a task that oversubscribes the GPU can significantly slow any other task that uses the same queue. In fact, we show that MPI-ACC induces less contention than MVAPICH and the manual asynchronous MPI+GPU approaches of GPU data communication. We show that MPI-ACC enqueues commands to the GPU hardware queue in a balanced manner, thereby minimizing the apparent performance slowdown in the D2H task (task-1) while incurring a modest slowdown to the MPI communication (task-0) itself.

Fig. 13a shows the relative increase in the MPI latency due to contention from the D2H task. For this experiment, the D2H task consistently transfers 16 MB between the device and the host, whereas the data size for the MPI task is varied. MPI-ACC shows a maximum slowdown of about 5 percent for relatively small MPI data transfers, and the slowdown for larger MPI data transfers is negligible, on average. The other implementations demonstrate less to negligible slowdown for all data sizes.

Fig. 13b shows the normalized bandwidth of the D2H task when run concurrently with MPI. For this experiment, the MPI task consistently performs 16 MB GPU-GPU data transfers across the network, whereas the data size for the local D2H task is varied. We see that MPI-ACC causes a maximum performance slowdown of 3.8 percent to the D2H task for relatively small data, and the performance slowdown for larger D2H data sizes is negligible. However, MVAPICH and the manual asynchronous MPI+GPU implementation causes a slowdown of about 18 percent for smaller D2H data transfers. This result indicates that MPI-ACC enqueues GPU tasks to the hardware queues in a more balanced manner, whereas MVAPICH may oversubscribe the hardware queues thereby causing significant performance variations to the other GPU tasks.

HPCToolkit analysis. HPCToolkit [29], [30] is a sampling-based performance analysis toolkit capable of quantifying scalability bottlenecks in parallel programs. We use HPCToolkit's `Hpctraceviewer` interface to understand why MPI-ACC causes less contention than the manual MPI +GPU implementations do. `Hpctraceviewer` renders hierarchical, timeline-based visualizations of parallel hybrid CPU-GPU programs. Fig. 14 presents screenshots of the detailed execution profile of our contention benchmark. The `hpctraceviewer` tool presents the timeline information of all CPU processes, threads, and their corresponding CUDA streams. However, we zoom in and show only the timelines of the relevant CUDA streams associated with both tasks of the 0th process. The other process exhibits identical behavior and is excluded from the figure.

Fig. 14a shows the effect of MPI-ACC's send operation interacting with the D2H data transfers of task-1. Since both tasks issue D2H commands and there is only one D2H queue on Fermi, we can see that only one of the CUDA streams is active at any given point in time. Moreover, the MPI-ACC's pipelining logic has been designed to issue GPU commands only when the next pipeline stage is ready. This design does not oversubscribe the GPU and leads to balanced execution, which can be seen by the interleaved bars in the MPI-related timeline. Fig. 14b depicts the contention effect of the manual pipelined MPI+GPU implementation. In this example implementation, we enqueue all the pipeline stages upfront, which is an acceptable design for standalone point-to-point communication. This design oversubscribes the GPU, however, and can be seen as clusters of bars in the MPI-related timeline. Of course, if one designs the manual MPI+GPU implementation similar to our MPI-ACC design, then the associated timeline figure will look like Fig. 14a. Since the manual MPI+GPU implementation is more aggressive in enqueuing GPU operations, the D2H operations of task-1 tend to wait more. That is why, on average, MPI-ACC causes the least performance perturbation to the D2H task (Fig. 13b).

7.3 Summary

In this section, we provided insights into the scalable design of MPI-ACC and compared its performance with MVAPICH and manual MPI+GPU implementations. Specifically, we showed that MPI-ACC delivers maximum concurrency by carefully ordering multiple GPU streams and efficiently balancing the H2D and D2H hardware queues for data pipelining, without oversubscribing the GPU resource.

8 Conclusion

In this paper, we introduced MPI-ACC, an integrated and extensible framework that allows end-to-end data movement in accelerator-connected systems. We discussed MPI-ACC's API design choices and a comprehensive set of optimizations including data pipelining and buffer management. We studied the efficacy of MPI-ACC for scientific applications from the domains of epidemiology (GPU-EpiSimdemics) and seismology (FDM -Seismology), and we presented the lessons learned and tradeoffs. We found that MPI-ACC's internal pipeline optimization not only helps improve the end-to-end communication performance but also enables novel optimization opportunities at the application level, which significantly enhance the CPU-GPU and network utilization. With MPI-ACC, one can naturally express the communication target without explicitly treating the CPUs as communication relays. MPI-ACC decouples the application logic from the low-level GPU communication optimizations, thereby significantly improving scalability and application portability across multiple GPU platforms and generations.

Acknowledgments

This work was supported in part by the US Department of Energy (DOE) grant DE-SC0001770; by contracts DE-AC02-06CH11357, DE-AC05-00OR22725, and DE-ACO6-76RL01830; by the DOE Office of Science under cooperative agreement number DE-FC02-07ER25800; by the DOE GTO via grant EE0002758 from Fugro Consultants; by the DTRA CNIMS contract HDTRA1-11-D-0016-0001; by the NSF PetaApps grant OCI-0904844; by the NSF Blue Waters grant OCI-0832603; by the NSF grants CNS-0960081, CNS-054630, and CNS-0958311; and by an NVIDIA Graduate Fellowship, NVIDIA Professor Partnership, and CUDA Research Center at Virginia Tech. This research used the HokieSpeed heterogeneous computing resource at Virginia Tech, which is supported by the National Science Foundation under contract CNS-0960081. The authors thank Darius Buntinas and Hao Wang for their valuable technical discussions and feedback.

References

[1] L. Weiguo, B. Schmidt, G. Voss, and W. Muller-Wittig, "Streaming algorithms for biological sequence alignment on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 9, pp. 1270–1281, Sep. 2007.

[2] A. Nere, A. Hashmi, and M. Lipasti, "Profiling heterogeneous multi-GPU systems to accelerate cortically inspired learning algorithms," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2011, pp. 906–920.

[3] W.-C. Feng, Y. Cao, D. Patnaik, and N. Ramakrishnan, "Temporal data mining for neuroscience," in *GPU Computing Gems*, Emerald ed. Amsterdam, The Netherlands: Elsevier, Feb. 2011.

[4] TOP500 Supercomputer Sites [Online]. Available: http://www.top500.org, 2015.

[5] *MPI: A Message-Passing Interface Standard Version 2.2*. Message Passing Interface Forum, 2009.

[6] NVIDIA CUDA C Programming Guide. (2014). [Online]. Available: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[7] L. Howes and A. Munshi. (2014). *The OpenCL Specification*. Khronos OpenCL Working Group [Online]. Available: https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf.

[8] NVIDIA GPUDirect [Online]. Available: https://developer.nvidia.com/gpudirect, 2015.

[9] C. L. Barrett, K. R. Bisset, S. G. Eubank, X. Feng, and M. V. Marathe, "EpiSimdemics: An efficient algorithm for simulating the spread of infectious disease over large realistic social networks," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2008, pp. 1–12.

[10] K. R. Bisset, A. M. Aji, M. V. Marathe, and W.-c. Feng, "High-performance biocomputing for simulating the spread of contagion over large contact networks," *BMC Genomics*, vol. 13, no. S2, p. S3, Apr. 2012.

[11] S. Ma and P. Liu, "Modeling of the perfectly matched layer absorbing boundaries and intrinsic attenuation in explicit finite element methods," *Bull. Seismological Soc. America*, vol. 96, no. 5, pp. 1779–1794, 2006.

[12] W. M. Brown, P. Wang, S. J. Plimpton, and A. N. Tharrington, "Implementing molecular dynamics on hybrid high performance computers - short range forces," *Comput. Phys. Commun.*, vol. 182, no. 4, pp. 898–911, 2011.

[13] T. Endo, A. Nukada, S. Matsuoka, and N. Maruyama, "Linpack evaluation on a supercomputer with heterogeneous accelerators," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2010, pp. 1–8.

[14] 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 *Proc. 14th IEEE Int. Conf. High Perform. Comput. Commun.*, 2012, pp. 647–654.

[15] H. Wang, S. Potluri, D. Bureddy, C. Rosales, and D. K. Panda, "GPU-aware MPI on RDMA-enabled clusters: Design, implementation and evaluation," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 10, pp. 2595–2605, Oct. 2014.

[16] MVAPICH: MPI over infiniband, 10GigE/iWARP and RoCE [Online]. Available: http://mvapich.cse.ohio-state.edu/, 2015.

[17] MPICH: High-performance portable MPI [Online]. Available: http://www.mpich.org, 2015.

[18] O. S. Lawlor, "Message passing for GPGPU clusters: CudaMPI," in *Proc. IEEE Int. Conf. Cluster Comput. Workshops*, 2009, pp. 1–8.

[19] J. Chen, W. Watson, and W. Mao, "GMH: A message passing toolkit for GPU clusters," in *Proc. IEEE 16th Int. Conf. Parallel Distrib. Syst.*, 2010, pp. 35–42.

[20] S. Yamagiwa and L. Sousa, "CaravelaMPI: Message passing interface for parallel GPU-Based applications," in *Proc. 8th Int. Symp. Parallel Distrib. Comput.*, 2009, pp. 161–168.

[21] J. Stuart and J. Owens, "Message passing on data-parallel architectures," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2009, pp. 1–12.

[22] S. Potluri, D. Bureddy, H. Wang, H. Subramoni, and D. Panda, "Extending OpenSHMEM for GPU computing," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2013, pp. 1001–1012.

[23] A. Sidelnik, B. L. Chamberlain, M. J. Garzaran, and D. Padua, "Using the high productivity language chapel to target GPGPU architectures," UIUC Dept. Comput. Sci., Champaign, IL, USA, 2011.

[24] J. Hammond, S. Ghosh, and B. Chapman, "Implementing OpenSHMEM using MPI-3 one-sided communication," in *Proc. 1st Workshop OpenSHMEM Related Technologies. Experiences, Implementations, Tools*, 2014, pp. 44–58.

[25] J. Dinan, P. Balaji, J. R. Hammond, S. Krishnamoorthy, and V. Tipparaju, "Supporting the global arrays PGAS model using MPI one-sided communication," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2012, pp. 739–750.

[26] A. M. Aji, P. Balaji, J. Dinan, W.-c. Feng, and R. Thakur, "Synchronization and ordering semantics in hybrid MPI GPU programming," in *Proc. 3rd Int. Workshop Accelerators Hybrid Exascale Syst.*, 2013, pp. 1020–1029.

[27] A. M. Aji, L. S. Panwar, F. Ji, M. Chabbi, K. Murthy, P. Balaji, K. R. Bisset, J. Dinan, W.-C. Feng, J. Mellor-Crummy, X. Ma, and R. Thakur, "On the efficacy of GPU-integrated MPI for scientific applications," in *Proc. ACM Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2013, pp. 191–202.
[28] J. Berenger, "A perfectly matched layer for the absorption of electromagnetic waves," *J. Comput. Phy.*, vol. 114, no. 2, pp. 185–200, 1994.
[29] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "HPCToolkit: Tools for performance analysis of optimized parallel programs," *Concurrency Comput.: Practice Experience*, vol. 22, pp. 685–701, Apr. 2010.
[30] M. Chabbi, K. Murthy, M. Fagan, and J. Mellor-Crummey, "Effective sampling-driven performance tools for GPU-accelerated supercomputers," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2013, pp. 1–12.

Ashwin M. Aji received the MS degree from the Department of Computer Science, Virginia Tech, where he is currently working toward the PhD degree.

Lokendra S. Panwar is working toward the MS degree in the Department of Computer Science, Virginia Tech.

Feng Ji is currently working toward the PhD degree in the Department of Computer Science, North Carolina State University.

Karthik Murthy received the MA degree in computer science from the University of Texas, Austin. He is currently working toward the PhD degree in the Department of Computer Science, Rice University.

Milind Chabbi received the MS degree in computer science from the University of Arizona, Tucson. He is currently working toward the PhD degree in the Department of Computer Science, Rice University.

Pavan Balaji is a computer scientist at Argonne National Laboratory, a fellow of the Northwestern-Argonne Institute of Science and Engineering at Northwestern University, and a research fellow in the Computation Institute at the University of Chicago.

Keith R. Bisset is a simulation and system software development scientist at the Virginia Bioinformatics Institute, Virginia Tech.

James Dinan received the PhD degree from the Ohio State University and completed a postdoctoral fellowship at Argonne National Laboratory.

Wu-chun Feng is a professor and Elizabeth & James E. Turner fellow in the Department of Computer Science, Department of Electrical and Computer Engineering, and Virginia Bioinformatics Institute at Virginia Tech.

John Mellor-Crummey is a professor in the Department of Computer Science and the Department of Electrical and Computer Engineering, Rice University.

Xiaosong Ma is a senior scientist at Qatar Computing Research Institute and an associate professor in the Department of Computer Science, North Carolina State University.

Rajeev Thakur is the deputy director in the Mathematical and Computer Science Division, Argonne National Laboratory, 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.