On the Efficacy of GPU-Integrated MPI for Scientific Applications

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

*Dept. of Comp. Sci., Virginia Tech, {aaji,lokendra,feng}@cs.vt.edu
†Dept. of Comp. Sci., North Carolina State Univ., fji@ncsu.edu, ma@csc.ncsu.edu
‡Dept. of Comp. Sci., Rice University, {mc29,ksm2,johnmc}@rice.edu
§Math. and Comp. Sci. Div., Argonne National Lab., {balaji,dinan,thakur}@mcs.anl.gov
¶Virginia Bioinformatics Inst., Virginia Tech, kbisset@vbi.vt.edu

ABSTRACT

Scientific computing applications are quickly adapting to leverage the massive parallelism of GPUs in large-scale clusters. However, the current hybrid programming models require application developers to explicitly manage the disjointed host and GPU memories, thus reducing both efficiency and productivity. Consequently, GPU-integrated MPI solutions, such as MPI-ACC and MVAPICH2-GPU, have been developed that provide unified programming interfaces and optimized implementations for end-to-end data communication among CPUs and GPUs. To date, however, there lacks an in-depth performance characterization of the new optimization spaces or the productivity impact of such GPU-integrated communication systems for scientific applications.

In this paper, we study the efficacy of GPU-integrated MPI on scientific applications from domains such as epidemiology simulation and seismology modeling, and we discuss the lessons learned. We use MPI-ACC as an example implementation and demonstrate how the programmer can seamlessly choose between either the CPU or the GPU as the logical communication end point, depending on the application's computational requirements. MPI-ACC also encourages programmers to explore novel application-specific optimizations, such as internode CPU-GPU communication with concurrent CPU-GPU computations, which can improve the overall cluster utilization. Furthermore, MPI-ACC internally implements scalable memory management techniques, thereby decoupling the low-level memory optimizations from the applications and making them scalable and portable across several architectures. Experimental results from a state-of-the-art cluster with hundreds of GPUs show that the MPI-ACC–driven new application-specific optimizations can improve the performance of an epidemiology simulation by up to 61.6% and the performance of a seismology modeling application by up to 44%, when compared with traditional hybrid MPI+GPU implementations. We conclude that GPU-integrated MPI significantly enhances programmer productivity and has the potential to improve the performance and portability of scientific applications, thus making a significant step toward GPUs being "first-class citizens" of hybrid CPU-GPU clusters.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
HPDC'13, June 17–21, 2013, New York, NY, USA.
Copyright 2013 ACM 978-1-4503-1910-2/13/06 ...$15.00.

Categories and Subject Descriptors: C.1.3 [*Processor Architectures*]: Other Architecture Styles – Heterogeneous (hybrid) systems; D.1.3 [*Programming Techniques*]: Concurrent Programming – Parallel programming

Keywords: MPI; GPGPU; MPI-ACC; Computational Epidemiology; Seismology

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 [13, 20, 25, 30]. The presence of GPUs in high-performance computing (HPC) clusters has also increased rapidly because of their unprecedented performance-per-power and performance-per-price ratios. In fact, 62 of today's top 500 fastest supercomputers (as of November 2012) employ general-purpose accelerators, 53 of which are GPUs [4].

While GPU-based clusters possess tremendous theoretical peak performance because of their inherent massive parallelism, the scientific codes that are hand-tuned and optimized for such architectures often achieve poor parallel efficiency [4, 9]. The reason is partially that data must be explicitly transferred between the disjoint memory spaces of the CPU and GPU, a process that in turn fragments the data communication model and restricts the degree of communication overlap with computations on the CPU *and* the GPU, thus effectively underutilizing the entire cluster. Also, significant programmer effort would be required to recover this performance through vendor- and system-specific optimizations, including GPU-Direct [3] and node and I/O topology awareness. Consequently, GPU-aware extensions to parallel programming models, such as the Message Passing Interface (MPI), have recently been developed, for example, MPI-ACC [6, 19] and MVAPICH2-GPU [28]. While such libraries provide a unified and highly efficient data communication mechanism for point-to-point, one-sided, and collective communications among CPUs and GPUs, an in-depth characterization of their impact on the execution profiles of scientific applications is yet to be performed. In this paper, we study the efficacy of GPU-integrated MPI libraries for GPU-accelerated

scientific applications, and we discuss the lessons learned. Our specific contributions are the following:

- We perform an in-depth analysis of hybrid MPI+GPU codes from two scientific application domains, namely, computational epidemiology [7, 9] and seismology modeling [23], and we identify the inherent inefficiencies in their fragmented data movement and execution profiles.
- We use MPI-ACC as an example GPU-integrated MPI solution and explore new design spaces for creating novel application specific optimizations.
- We evaluate our findings on *HokieSpeed*, a state-of-the-art hybrid CPU-GPU cluster housed at Virginia Tech. We use HPCToolkit [5], a performance analysis and visualization tool for parallel programs, to understand the performance and productivity tradeoffs of the different optimizations that were made possible by MPI-ACC.

We demonstrate how MPI-ACC can be used to easily explore and evaluate new optimization strategies. 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. Moreover, we demonstrate how MPI-ACC provides the flexibility to the programmer to seamlessly choose between CPU or GPU to execute the next task at hand, thus enhancing programmer productivity. For example, in the default MPI+GPU programming model, the CPU is traditionally used for prerequisite tasks, such as data marshaling or data partitioning, before the GPU computation. In contrast, MPI-ACC provides a logical integrated view of CPUs and GPUs so that the programmer can choose to move the raw data to the remote GPU itself, then execute the prerequisite tasks and the actual computation as consecutive GPU kernels. In addition, MPI-ACC internally implements scalable memory management techniques, thereby decoupling the low-level memory optimizations from the applications and making them scalable and portable across several architecture generations. In summary, we show that with MPI-ACC, the programmer can easily evaluate and quantify the tradeoffs of many communication-computation patterns and choose the ideal strategy for the given application and machine configuration. Our experimental results on HokieSpeed indicate that MPI-ACC–driven optimizations and the newly created communication-computation patterns can help improve the performance of the epidemiology simulation by 14.6% to up to 61.6% and the seismology modeling application by up to 44% over the traditional hybrid MPI+GPU models.

The rest of the 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. Sections 3 and 4 explain 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. Section 5 discusses related work, and Section 6 summarizes our conclusions.

2. APPLICATION DESIGN FOR HYBRID CPU-GPU SYSTEMS

2.1 Default MPI+GPU Design

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 [2] and OpenCL [15]. 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`.

The Message Passing Interface (MPI) 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 for further acceleration. 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

```
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.

Figure 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 GPU-integrated MPI models such as MPI-ACC, the code remains the same for *all* platforms (CUDA or OpenCL) and supported devices.

between the host CPUs (Figures 1a and 1b). This is the norm seen in most GPU-accelerated MPI applications today [9, 10, 12]. The basic approach (Figure 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 [3]).

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 [6] framework and MVAPICH-GPU [28] 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. For example, MPI-ACC enables automatic data pipelining for internode communication, NUMA affinity management, and direct GPU-to-GPU data movement (GPUDirect) for all applicable intranode CUDA communications [6, 19], thus providing a heavily optimized end-to-end communication platform.

Using GPU-integrated MPI, programmers only need 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 aforementioned complex data movement optimizations of the diverse accelerator technologies (Figure 1c). In this paper, we design, analyze, and evaluate GPU-accelerated scientific applications by using MPI-ACC as the chosen GPU-integrated MPI platform. However, our findings on the efficacy of MPI-ACC can directly be applied to other GPU-integrated MPI solutions without loss of generality.

3. CASE STUDY: EPIDEMIOLOGY SIMULATION

GPU-EpiSimdemics [7, 9] 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 (Figure 2a) and interact with each other 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.

3.1 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 (Figure 2a). 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 (Figure 2b), 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 (Figure 2c) 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 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.

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

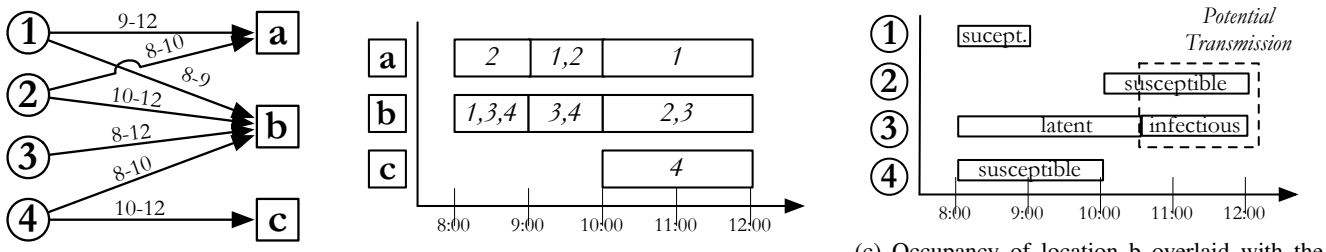
In GPU-EpiSimdemics, each PE in the simulation is implemented as a separate MPI process. Also, the *computeInteractions* phase of GPU-EpiSimdemics is offloaded and accelerated on the GPU while the rest of the computations are executed on the CPU [9].[1] In accordance with the GPU-EpiSimdemics algorithm, 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.

GPU-EpiSimdemics has two GPU computation modes: exclusive GPU computation, where all the visit messages are processed on the GPU, and cooperative CPU-GPU computation, where the visit messages are partitioned and concurrently processed on both the GPU and its host CPU. For each mode, we discuss the optimizations and tradeoffs. We also describe how MPI-ACC can be used to further optimize GPU-EpiSimdemics in both computation modes.

3.2.1 Exclusive GPU computation mode

In the exclusive GPU computation mode, all incoming visit messages are completely executed on the GPU during the *computeInteractions* phase. We present three optimizations with their tradeoffs.

[1]The current implementation of GPU-EpiSimdemics assumes one-to-one mapping of GPUs to MPI processes.

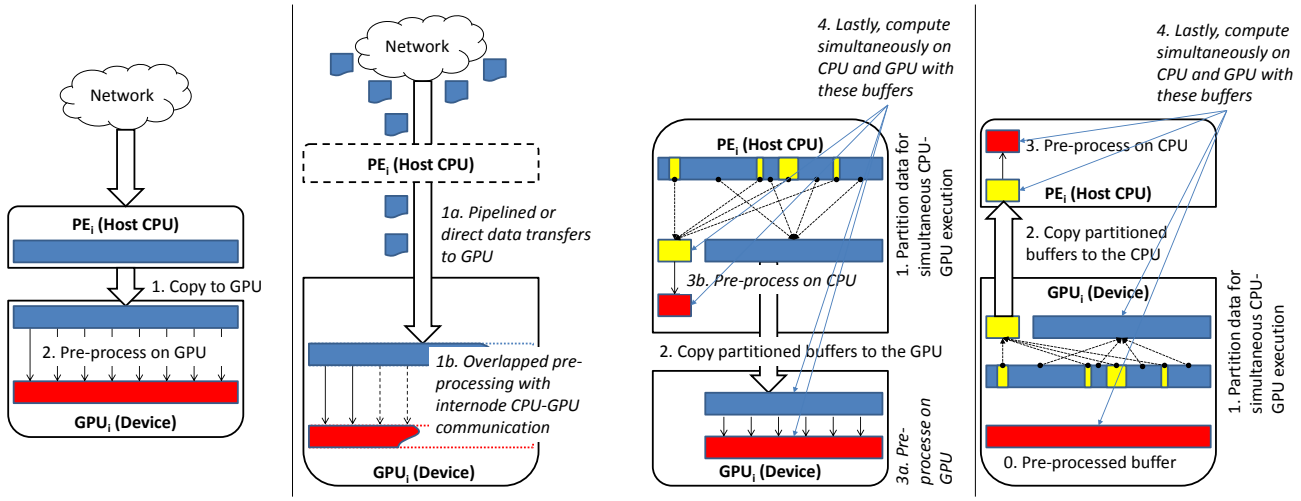


(a) Bipartite graph representation with people 1–4 and locations a–c.

(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.

Figure 2: Computational epidemiology simulation model (figure adapted from [7]).



(a) Exclusive GPU computation mode. Left: Manual MPI+CUDA optimizations, where the *visit* messages are received on the host, then copied to the device for preprocessing. Right: New MPI-ACC–enabled optimizations, where the *visit* messages are transparently pipelined into the device and preprocessing is overlapped.

(b) Cooperative CPU-GPU computation mode. Left: Manual MPI+CUDA optimizations, where data partitioning happens on the CPU. Right: New MPI-ACC–enabled optimizations, where the data distribution happens on the GPU. The preprocessing of the GPU data is still overlapped with communication.

Figure 3: Creating new optimizations for GPU-EpiSimdemics using MPI-ACC.

Basic MPI+GPU communication-computation pattern.

Internode CPU-GPU data communication: In the naïve data movement approach, each PE first receives all the visit messages in the CPU's main memory during the *computeVisits* phase, then transfers the aggregate data to the local GPU (device) memory across the PCIe bus at the beginning of the *computeInteractions* phase. The typical all-to-all or scatter/gather type of operation is not feasible because the number of pairwise visit message exchanges is not known beforehand in GPU-EpiSimdemics. Thus, each PE preallocates and registers fixed-sized *persistent* buffer fragments with the `MPI_Recv_init` call and posts the receive requests by subsequently calling `MPI_Start_all`. Whenever a buffer fragment is received, it is copied into a contiguous visit vector in the CPU's main memory. The *computeInteractions* phase of the simulation first copies the aggregated visit vector to the GPU memory. 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.

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 [9]. 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 another 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.

MPI-ACC–enabled optimizations.

In the basic version of GPU-EpiSimdemics, the GPU remains idle during the internode data communication phase of *computeVisits*, whereas the CPU remains idle during the preprocessing of the *computeInteractions* phase on the GPU. We use the performance analysis tool HPCTOOLKIT to quantify the resource idleness and identify potential code regions for MPI-ACC optimiza-

tion (Section 3.3: Figure 6a). With MPI-ACC, during the *computeVisits* phase, we transfer the visit message fragments from the source PE directly to the destination GPU's device memory. Internally, MPI-ACC may 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. The fixed-sized *persistent* buffer fragments are now preallocated *on the GPU* and registered with the `MPI_Recv_init` call, and the contiguous visit vector is not created in the GPU memory itself. Furthermore, as soon as a PE receives the visit buffer fragments on the GPU, we immediately launch small GPU kernels that preprocess on the received visit data, that is, unpack the partial visit messages to the 2D data matrix layout (Figure 3a). These preprocessing kernels execute asynchronously with respect to the CPU in a pipelined fashion and thus are completely overlapped by the visit data generation on the CPU and the internode CPU-GPU data transfers. In this way, the data layout transformation 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 of GPU-EpiSimdemics. 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. However, 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 most combinations of system configurations and input data sizes.

Advanced MPI+GPU optimizations without using MPI-ACC.

MPI-ACC enables efficient communication-computation overlap by pipelining the CPU-GPU data transfers with the preprocessing stages. However, the same optimizations can be implemented at the application level without using MPI-ACC, as follows. The fixed-sized *persistent* receive buffer fragments are preallocated on the CPU itself and registered with the `MPI_Recv_init` call, but the contiguous visit vector resides in GPU memory. 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 launch the small GPU preprocessing kernels. However, asynchronous CPU-GPU data transfers require the CPU receive buffer fragments to be nonpageable (pinned) memory. Without MPI-ACC, the pinned memory footprint increases with the number of processes, thus reducing the available pageable CPU memory and leading to poor CPU performance [26]. On the other hand, MPI-ACC internally creates and manages a *constant* pool of pinned memory for CPU-GPU transfers, which enables better scaling. Moreover, MPI-ACC exposes a natural interface to communicate with the target device, be it either the CPU or the GPU.

3.2.2 Cooperative CPU-GPU computation mode

The exclusive GPU computation mode achieved significant overlap of communication with computation during the preprocessing phase. When the infection calculation of the *computeInteractions* phase was executed on the GPU, however, the CPU remained idle. We again used HPCToolkit to analyze the resource idleness and identified opportunities for optimizations. Consequently, in the cooperative computation mode, all the incoming visit messages are partitioned and processed concurrently on the GPU and its host CPU during the *computeInteractions* phase, an approach that gives better parallel efficiency. Again, we present three optimizations with their tradeoffs.

Basic MPI+GPU with data partitioning on CPU.

In the MPI+GPU programming model, the incoming visit vector on the CPU is not transferred in its entirety to the GPU. Instead, the visit messages are first grouped by their target locations into buckets. Within each visit group, the amount of computation increases quadratically with the group size because it is an all-to-all person-person interaction computation within a location. Each visit group can be processed independently of the others but has to be processed by the same process or thread (CPU) or thread block (GPU). Therefore, data partitioning in GPU-EpiSimdemics is done at the granularity of *visit groups* and not individual visit messages.

At a high level, the threshold for data partitioning is chosen based on the computational capabilities of the target processors (e.g., GPUs get more populous visit groups for higher concurrency), so that the execution times on the CPU and the GPU remain approximately the same. The visit messages that are marked for GPU execution are then grouped and copied to the GPU device memory, while the CPU visit messages are grouped and remain on the host memory (Figure 3b).

Preprocessing and computation phases: In this computation mode, preprocessing, in other words, unpacking the visit structure layout to the person-timeline matrix layout, is concurrently executed on the CPU and GPU on their local visit messages (Figure 3b). Next, the CPU and GPU simultaneously execute *computeInteractions* and calculate the infections.

MPI-ACC–enabled optimizations with data partitioning on GPU.

In the MPI-ACC model, the computation of the *computeInteractions* phase is executed on the CPU and GPU concurrently. While this approach leads to better resource utilization, the data partitioning logic itself and the CPU-GPU data transfer of the partitioned data add nontrivial overheads that may offset the benefits of concurrent execution. However, our results in Section 3.3 indicate that executing the data partitioning logic *on the GPU* is about 53% faster than on the CPU because of the GPU's higher memory bandwidth. With MPI-ACC, the visit vector is directly received or pipelined into the GPU memory, and the data partitioning logic is executed on the GPU itself. Next, the CPU-specific partitioned visit groups are copied *to the CPU* (Figure 3b). As a general rule, if the GPU-driven data partitioning combined with the GPU-to-CPU data transfer performs better than the CPU-driven data partitioning combined with CPU-to-GPU data transfer, then GPU-driven data partitioning is a better option. Our experimental results (Section 3.3) indicate that for GPU-EpiSimdemics, the MPI-ACC enabled GPU-driven data partitioning performs better than the other data partitioning schemes.

The preprocessing phase *on the GPU* is still overlapped with the internode CPU-GPU communication by launching asynchronous GPU kernels, just like the exclusive GPU mode, thereby largely mitigating the preprocessing overhead. While this approach could lead to redundant computations for the CPU-specific visit groups on the GPU, the corresponding person-timeline matrix rows can be easily ignored in the subsequent execution phases. This approach will create some unnecessary memory footprint on the GPU; however, the benefits of overlapped preprocessing outweigh the issue of memory overuse. On the other hand, the preprocessing *on the CPU* is executed only after the data partitioning and GPU-to-CPU data transfer of CPU-specific visit groups. This step appears on the critical path and cannot be overlapped with any other step, but it

causes negligible overhead for GPU-EpiSimdemics because of the smaller data sets for the CPU execution.

Advanced MPI+GPU with data partitioning on GPU.

GPU-driven data partitioning can also be implemented without using MPI-ACC, where the visits vector is created on the GPU and the preprocessing stage is overlapped by the local CPU-GPU data communication, similar to the advanced MPI+GPU optimization of the exclusive GPU computation mode. The data partitioning on the GPU and the remaining computations follow from the MPI-ACC–enabled optimizations. As in the GPU exclusive computation mode, however, the pinned memory footprint increases with the number of processes, which leads to poor CPU performance and scaling. Moreover, from our experience, the back-and-forth CPU-GPU data movement in the GPU-driven data partitioning optimization seems convoluted without a GPU-integrated MPI interface. On the other hand, MPI-ACC provides a natural interface for GPU communication, which encourages application developers to explore new optimization techniques such as GPU-driven data partitioning and to evaluate them against the default and more traditional CPU-driven data partitioning schemes.

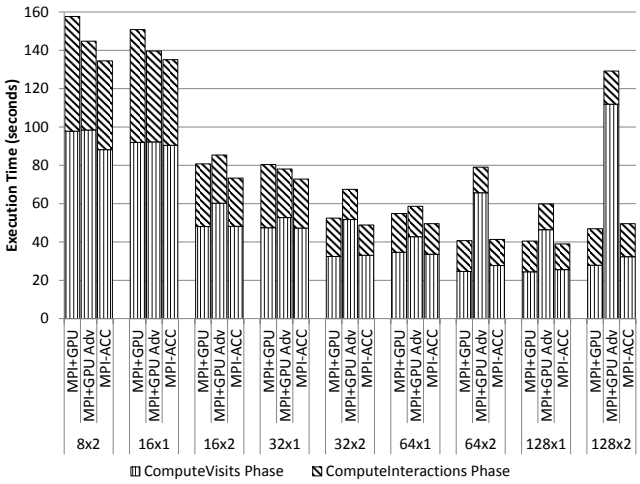
3.3 Evaluation and Discussion

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 v4.0 with driver version 270.41.34.

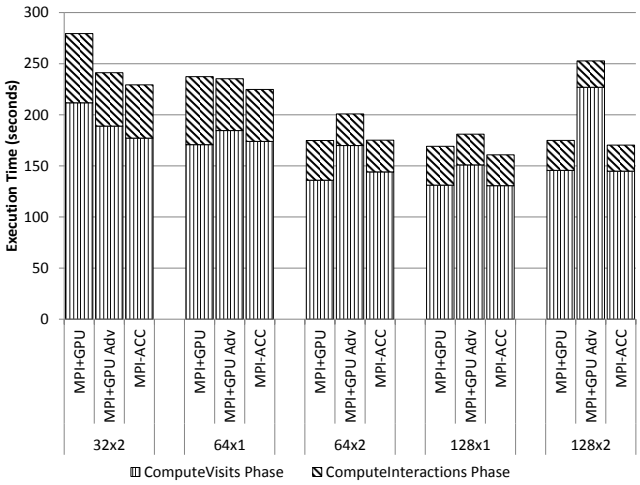
We compare the combined performance of all the phases of GPU-EpiSimdemics (*computeVisits* and *computeInteractions*), with and without the MPI-ACC–driven optimizations discussed earlier. We choose different-sized input data sets from synthetic populations from two U.S. states: Washington (WA) with a population of 5.7 million and California (CA) with a population of 33.1 million. We also vary the number of compute nodes from 8 to 128 and the number of GPU devices between 1 and 2. Each U.S. state begins its computation from the smallest node-GPU configuration that can fit the entire problem in the available GPU memory.

Our results in Figures 4a and 4b indicate that in the exclusive GPU computation mode, our MPI-ACC–driven optimizations perform better than the basic blocking MPI+GPU implementations by an average of 6.3% and by up to 14.6% for WA. Similarly, the new optimizations perform better by an average of 6.1% and by up to 17.9% for CA, where the gains come from the *computeInteractions* phase. The MPI-ACC–driven solution also outperforms the advanced pipelined MPI+GPU implementations by an average of 24.2% and up to 61.6% for WA and by an average of 13.1% and up to 32.5% for CA, but the gains come from the *computeVisits* phase. We also observe that the new optimizations are better than the basic MPI+GPU solution for smaller node configurations and are superior to the advanced MPI+GPU solution for larger nodes. For large number of nodes, however, the basic MPI+GPU solution can also outperform the MPI-ACC–enabled optimization by at most 5% for WA and only 0.1% for CA. Also, we find that the MPI-ACC–enabled solution is always better than or on par with the advanced MPI+GPU solution.

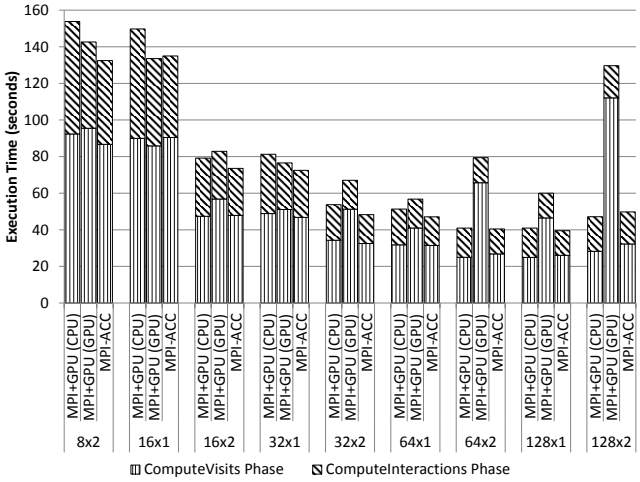
We see identical trends in the cooperative CPU-GPU computation mode (Figure 4c), where the MPI-ACC–driven GPU data-partitioning strategy usually performs better than both the MPI+GPU



(a) Exclusive GPU mode for Washington.



(b) Exclusive GPU mode for California.



(c) Cooperative CPU-GPU mode for Washington.

Figure 4: Execution profile of GPU-EpiSimdemics over various node configurations. The x-axis increases with the total number of MPI processes P, where P = Nodes * GPUs.

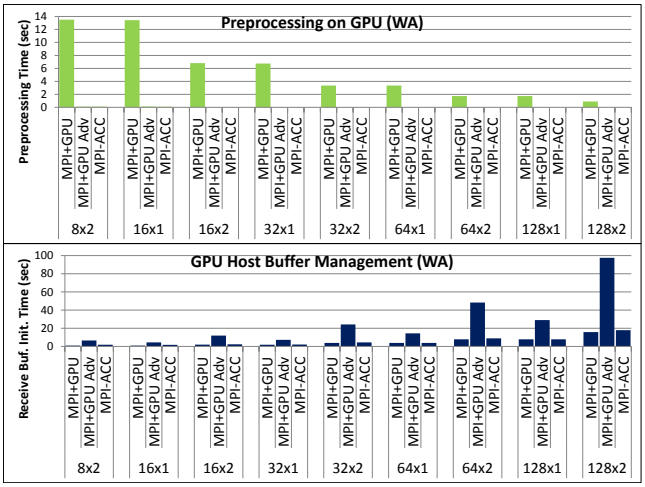


Figure 5: Analysis of MPI-ACC–enabled performance optimizations. The x-axis indicates the MPI process count P, where P = Nodes * GPUs.

implementations of CPU-driven and GPU-driven data-partitioning schemes. MPI-ACC is better than the CPU-driven MPI+GPU solution for smaller number of nodes, while it heavily outperforms the GPU-driven MPI+GPU solution for larger node configurations. The data-partitioning logic, by itself, performs about 53% faster on the GPU.

3.3.1 MPI-ACC–enabled optimization vs. basic MPI+GPU

The top portion of Figure 5 depicts an in-depth analysis of the benefits of overlapped preprocessing on the GPU in the exclusive GPU mode for the WA state. However, the following analysis holds good even for the CA state and the cooperative compute modes as well. We can see from the top portion of Figure 5 that for the MPI-ACC and advanced MPI+GPU implementations, the preprocessing step (data unpacking) of the *computeInteractions* phase is completely overlapped with the CPU to remote GPU communication. This is why the MPI-ACC solution outperforms the basic MPI+GPU solution for most node configurations.

Scalability analysis. For larger node configurations, however, the local operating data set in the *computeInteractions* phase becomes smaller, which means that the basic MPI+GPU solution takes less time to execute the preprocessing (data unpacking) stage. Thus, the gains over the basic MPI+GPU solution, achieved by overlapping the preprocessing step with GPU communication, also get diminished. Note that MPI-ACC or any other GPU-integrated MPI, by itself, does not impact the performance gains. In contrast, MPI-ACC enables the developer to create newer optimizations for better resource utilization, but the scalability of GPU-EpiSimdemics itself limits the scope for performance improvement. Thus, we see comparable performances of the basic MPI+GPU and the MPI-ACC–driven optimizations for larger number of nodes, but the threshold node configuration at which we see diminishing returns from the new optimization varies for different input data sets (states).

3.3.2 MPI-ACC–enabled optimization vs. advanced MPI+GPU

The bottom portion of Figure 5 shows that the receive buffer management time in the advanced MPI+GPU case increases rapidly for larger numbers of nodes. The reason is that the pinned memory footprint is an increasing function of the number of MPI processes, which largely reduces the available pageable CPU memory and leads to poor performance [26]. This is why the MPI-ACC–enabled solution outperforms the advanced MPI+GPU solution for most node configurations, especially for larger node configurations. We can also observe 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 (e.g. 64×2 vs. 128×1). This result is expected because both MPI processes on each node create pinned memory buffers, thus leading to even lesser pageable memory and poorer performance. On the other hand, MPI-ACC provides a more scalable solution by (1) managing a fixed-size pinned buffer pool for pipelining and (2) creating them at `MPI_Init` and destroying them at `MPI_Finalize`. Note that the pipelined data movement optimization, by itself, does not significantly improve performance in the application's context. Instead, MPI-ACC's efficient buffer pool management for pipelining provides huge benefits for the application.

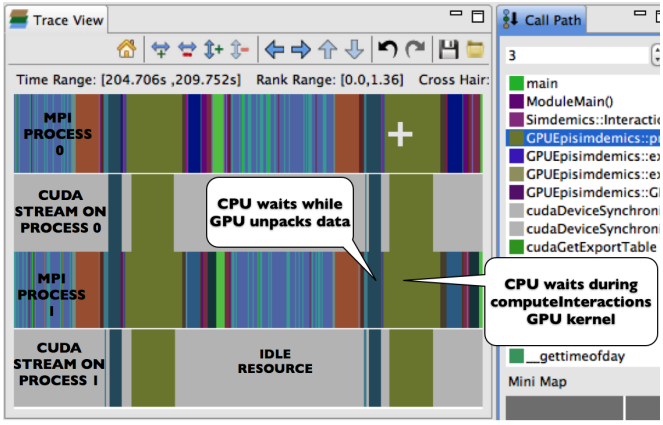
The basic MPI+GPU solution has the preprocessing overhead but does not suffer from any memory management issues. While the advanced MPI+GPU implementation gains from hiding the preprocessing overhead, it loses from nonscalable pinned memory management. Also, for the advanced MPI+GPU implementation, it turns out that the performance loss due to the inefficient pinned memory management is in general much more severe than the gains achieved by hiding the preprocessing overhead (figure 5). On the other hand, MPI-ACC gains from both hiding the preprocessing overhead and efficient pinned memory management. It is possible to create and manage an efficient pinned memory pool at the application level in the advanced MPI+GPU case, but doing so increases the complexity of the simulation and leads to poor programmer productivity. Ideally, the lower-level memory management logic should be decoupled from the high-level simulation implementation, as is made possible by MPI-ACC.

3.3.3 Analysis of resource utilization using HPCToolkit

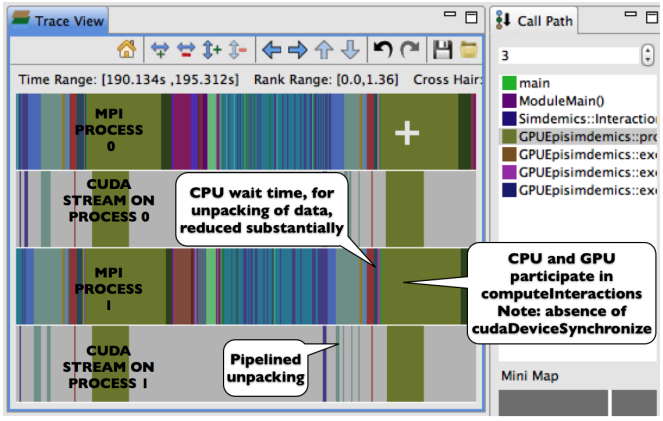
HPCTOOLKIT [5] is a sampling based performance analysis toolkit capable of quantifying scalability bottlenecks in parallel programs. In this paper, we use an extension of HPCTOOLKIT that works on hybrid (CPU-GPU) codes; the extension uses a combination of sampling and instrumentation of CUDA code to accurately identify regions of low CPU/GPU utilization. HPCTOOLKIT presents program execution information through two interfaces: `hpcviewer` and `hpctraceviewer`. `Hpcviewer` associates performance metrics with source code regions including lines, loops, procedures, and calling contexts. `Hpctraceviewer` renders hierarchical, timeline-based visualizations of parallel program executions.

In Figure 6, we present snapshots of the detailed execution profile of GPU-EpiSimdemics from the `hpctraceviewer` tool of HPCTOOLKIT. Figure 6a depicts the application without the MPI-ACC-driven optimizations. The `hpctraceviewer` tool presents the timeline information of all CPU processes and their corresponding CUDA streams. The *call path* pane on the right represents the call stack of the process/stream at the current crosshair position. Although we study a 32-process execution, we zoom in and show only the 0th and 1st processes and their associated CUDA streams, because the other processes exhibit identical behavior.

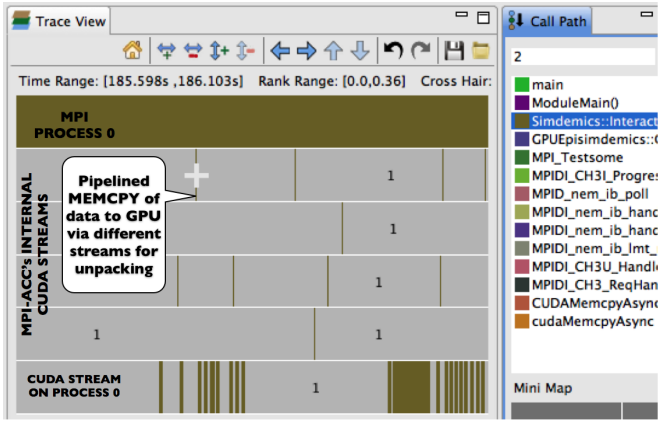
The figure depicts two iterations of the application, where a couple of *computeInteractions* phases, with the corresponding GPU activity, are surrounding a *computeVisits* phase, where there is no GPU activity. The GPU idle time during the *computeVisits* phase



(a) Manual MPI+CUDA optimizations. The *visit* messages are first received on the CPU and copied to the device; then the preprocessing (unpacking) takes place on the GPU.



(b) MPI-ACC optimizations. The *visit* messages are received directly in the device. Preprocessing (unpacking) on the GPU is pipelined and overlapped with data movement to the GPU. This leads to negligible CPU waiting while the GPU preprocesses/unpacks the data.



(c) MPI-ACC optimizations. This figure combines (b) with activity occurring on other streams. MPI-ACC employs multiple streams to push the data to the device asynchronously, while the application initiates the unpacking of data.

Figure 6: Analysis and evaluation of MPI-ACC–driven optimizations using HPCTOOLKIT. Application Case Study: GPU-EpiSimdemics.

can be reduced by offloading parts of the *computeVisits* computation to the GPU; but that is beyond the scope of this paper.

In the basic hybrid MPI+GPU programming model, the application launches kernels on the *default* CUDA stream for the *computeInteractions* phase, including the preprocessing (or data unpacking) and the main infection processing stages. In the figure, we can see a corresponding set of bars on the default CUDA stream in the *computeInteractions* phase, which denote the following: (1) a small, negligible sliver showing `cudaMemcpy` of the visit messages from the CPU to the GPU; (2) a medium-sized bar showing preprocessing (or data unpacking) on the GPU; and (3) a thick band showing the main infection computation kernel. The figure thus helps identify two distinct issues and opportunities for performance improvement in the *computeInteractions* phase of GPU-EpiSimdemics.

1. The thick band on the CUDA stream representing the main kernel of the *computeInteractions* phase has a corresponding thick `cudaDeviceSynchronize` band on the CPU side; that is, the CPU is idle while waiting for the GPU, thus indicating that some work from the GPU can be offloaded to the CPU.

2. The medium-sized bar on the CUDA stream representing the preprocessing (data unpacking) step has a corresponding `cudaDeviceSynchronize` bar on the CPU, which indicates that the CPU can start offloading the data to be unpacked to the GPU in stages, thus overlapping data transfers to the GPU with their unpacking on the GPU.

We resolve the first issue by using the cooperative CPU-GPU computation mode. The second issue is resolved in both the cooperative and the exclusive GPU modes, as discussed in Sections 3.2.1 and 3.2.2. We use MPI-ACC to pipeline the data unpacking before the *computeInteractions* phase by overlapping it with the *computeVisits* phase. We use a custom CUDA stream to execute the preprocessing kernel so that we can achieve an efficient overlap between the H-D data transfers within MPI-ACC and the preprocessing kernel of GPU-EpiSimdemics. Figure 6b, which represents HPCTOOLKIT's trace view on applying these optimizations, shows that the time wasted by the CPU in `cudaDeviceSynchronize` while the GPU unpacked the data has disappeared (compared with Figure 6a). This reduction in the CPU idle time characterizes the success of the MPI-ACC–driven optimizations.

Figure 6c shows a zoomed-in version of Figure 6b, where we can see the internal helper streams that are created within MPI-ACC along with the custom CUDA stream of one of the processes (only a subset of MPI-ACC's internal streams is shown here for brevity). While the GPU kernels of the *computeInteractions* phase are executed on the application's custom stream, the staggered bars in the MPI-ACC's internal streams represent the pipelined data transfers before the unpacking stage, thus showing efficient use of concurrency via multiple GPU streams.

Summary.

In summary, MPI-ACC helps achieve the following for both the GPU computation modes of GPU-EpiSimdemics:

- Provides a natural way of moving data to the desired computational resource (CPU or GPU), which encourages application developers to explore new optimization techniques such as GPU-driven data partitioning and to evaluate them against the more traditional optimization schemes

- Enables receiving and preprocessing the data on the GPU concurrently with the data generation on the CPU, thus enhancing the utilization of all the computation and communication resources of the system

- Efficiently manages pinned buffer pool to help pipeline the GPU data quickly via the host CPU

4. CASE STUDY: SEISMOLOGY MODELING

FDM-Seismology is our MPI+GPU hybrid implementation of an application that models the propagation of seismological waves using the finite-difference method by taking the Earth's velocity structures and seismic source models as input [23]. The application implements a parallel velocity-stress, staggered-grid finite-difference method [11, 14, 24] 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 [23] 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 [23]. This strategy helps simulate unbounded domains. In our application, PML (perfectly matched layers) absorbers [8] 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 [23].

4.1 Computation-Communication Patterns

The simulation operates on the input finite-difference (FD) 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 technique, also known as domain decomposition, allows the computation in the application to 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 (Figure 7a). These exchanges help each processor update its own wavefields after receiving wavefields generated at the neighbors.

These computations are run for a large number of iterations 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, then unpacked at the receiver to update its local data.

4.2 GPU Acceleration of FDM-Seismology

MPI+GPU with data marshaling on CPU (MPI+GPU): Our GPU-accelerated version of the application performs the velocity and stress computations as kernels on the GPU. 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 (Figure 7a). 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): A much faster GDDR5 memory module is available on the GPU. If the memory accesses are coalesced, the data-marshaling module performs much better than the CPU, which has the slower DDR3 memory. Also, the packing and unpacking operations of the data-marshaling stages can benefit from the highly multithreaded SIMT execution nature of GPU. Hence, it is a natural optimization to move the data-marshaling operations to the GPU (Figure 7b). Moreover, the CPU-GPU bulk data transfers that used to happen before and after each velocity-stress computation kernel are 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.

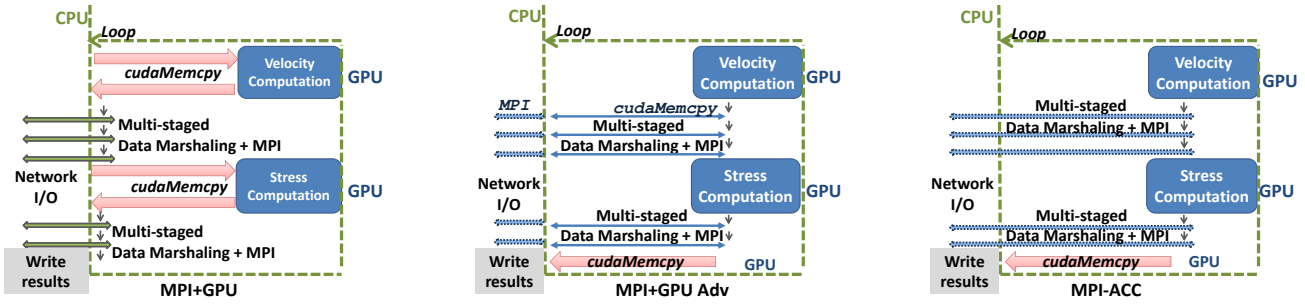
However, such an optimization has the following disadvantage in the absence of GPU-integrated MPI. All data-marshaling steps are separated by MPI communication, and each data-marshaling step depends on the preceding marshaling step *and* the received MPI data from the neighbors. In other words, after 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., GPU-Direct RDMA), we will have to largely rewrite the FDM-Seismology communication code to achieve the expected performance.

4.3 MPI-ACC–Enabled Optimizations

Since the MPI-ACC library enables MPI communication directly from the GPU buffer, the new version of the application retains the velocity and stress computation results on the GPU itself, performs the packing and unpacking operations using multiple threads on the GPU, and communicates the packed arrays directly from the GPU. Similar to the MPI+GPU Adv case, the bulk transfer of data happens only once at the end of each iteration, when results are written to a file.

The MPI-ACC–driven design of FDM-Seismology with data marshaling on the GPU greatly benefits from the reduction in the number of expensive synchronous bulk data transfer steps between the CPU and GPU. Also, since the data-marshaling step happens multiple times during a single iteration, the application needs to launch a series of marshaling kernels on the GPU. While consecutive kernel launches entail some kernel launch and synchronization overhead per kernel invocation, the benefits of faster data marshaling on the GPU and optimized MPI communication make the kernel synchronization overhead insignificant.

GPU-driven data marshaling provides the following benefits to MPI+GPU Adv and MPI-ACC–based designs of FDM-Seismology: (1) it removes the need for the expensive bulk `cudaMemcpy` data transfers that were used to copy the results from the GPU to the CPU after each set of velocity and stress computations; and (2)



(a) Basic MPI+GPU FDM-Seismology application with data marshaling on CPU.

(b) MPI+GPU FDM-Seismology application with data marshaling on GPU.

(c) MPI-ACC–driven FDM-Seismology application with data marshaling on GPU.

Figure 7: Communication-computation pattern in the FDM-Seismology application.

the application benefits from the multiple threads performing the data packing and unpacking operations in parallel, and on the faster GDDR5 device memory.

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

4.4 Evaluation and Discussion

In this section, we analyze the performance of the different phases of the FDM-Seismology application and evaluate the effect of MPI-ACC optimizations on the application. The platform for evaluation is the HokieSpeed cluster, whose details were specified in Section 3.3. We use strong scaling to understand the variation in the application's performance when the number of nodes increases and the internode data transfer size decreases. We vary the nodes from 2 to 128 with 1 GPU per node and use two different sized datasets, Dataset-1 and Dataset-2, as input. Our scalability experiments begin from the smallest number of nodes required to fit the given data in the GPU 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.

4.4.1 MPI-ACC–enabled optimizations vs. basic and advanced MPI+GPU

Figure 8 shows the performance of the FDM-Seismology application, with nodes varying from 16 to 128, when used with and without the MPI-ACC–enabled designs. In the figure, we report the average computation time across all the nodes, because the computation-communication costs vary largely, depending on the location of the node in the structured grid problem representation of FDM-Seismology. In the MPI+GPU case, we perform both data-marshaling operations and MPI communication on the CPU. In the MPI+GPU Adv case, we perform the data-marshaling operations

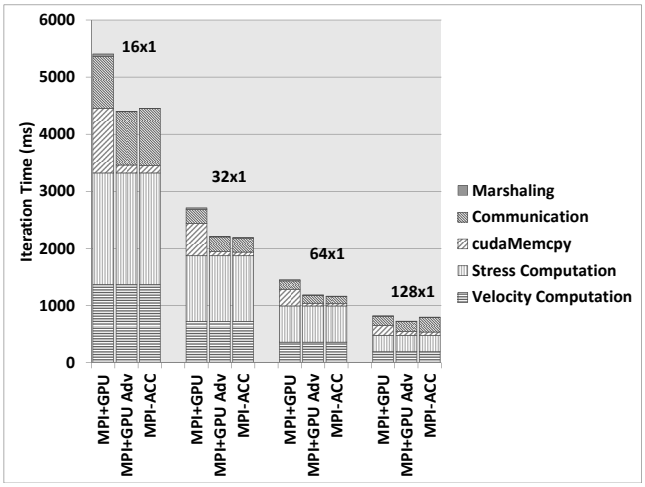


Figure 8: Analysis of the FDM-Seismology application when strongly scaled. The larger dataset, Dataset-2, is used for these results. Note: MPI Communication represents CPU-CPU data transfer time for the MPI+GPU and MPI+GPU Adv cases and GPU-GPU (pipelined) data transfer time for the MPI-ACC case.

on the GPU, while the MPI communication still takes place explicitly from the CPU.

One can see that although the velocity and stress computations take most of the application's computation time (>60%), the MPI-GPU Adv and MPI-ACC–driven design of the application see large performance improvements over the MPI+GPU case, primarily because of the reduction in expensive explicit bulk data transfer operations between the CPU and GPU (Figure 7c). In the MPI+GPU case, the application needs to move large wavefield data between the CPU and the GPU for the data-marshaling computation and MPI communication. On the other hand, when used with MPI-ACC, the application performs all the data-marshaling computation and MPI communication directly from the GPU, and it needs to transfer smaller-sized wavefield data from GPU to CPU only once at the end of the iteration for writing the result to the output file.

In the MPI+GPU Adv case, since the marshaling steps are performed on the GPU and much smaller-sized data arrays are moved between the CPU and GPU for MPI communication, we still benefit from avoiding the bulk data transfer steps. However, these small wavefield data movements have to be invoked many times and result in reduced programmer productivity. With MPI-ACC, the pro-

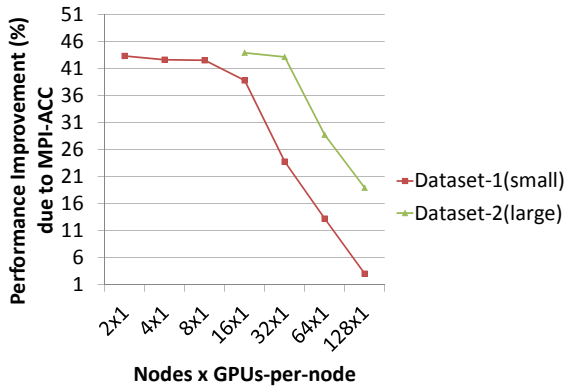


Figure 9: Scalability analysis of FDM-Seismology application with two datasets of different sizes. The baseline for speedup is the naïve MPI+GPU programming model with CPU data marshaling.

grammer enjoys productivity gain as well as the performance improvements as a result of pipelined data transfers via the CPU.

We also analyzed the FDM-Seismology application with HPCTOOKIT and observed CUDA stream activity similar to that in Figure 6c, where multiple internal CUDA streams asynchronously transfer the data over MPI while the application's stream is busy doing marshaling computations. This approach helps the application performance by increasing the resource utilization.

4.4.2 Scalability analysis

Figure 9 shows the performance improvement due to the MPI-ACC–enabled GPU data marshaling strategy over the basic hybrid MPI+GPU implementation with CPU data marshaling. We see that the performance benefits due to the GPU data marshaling design decrease with increasing number of nodes. The reason for this behavior is twofold:

- For a given dataset, the per-node data size decreases with increasing number of nodes. This reduces the costly CPU-to-GPU and GPU-to-CPU bulk data transfers (Figure 8) and thus minimizes the overall benefits of performing data marshaling on the GPU itself.
- As the number of nodes increase, 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 communication of the traditional hybrid MPI+GPU implementations will have less overhead than will the pipelined GPU-to-GPU communication of the MPI-ACC–enabled design.

While the pipelined data transfer optimization within MPI-ACC improves the communication performance to a certain degree, it has negligible impact on the performance gains of the application. 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.

5. RELATED WORK

GPUs have been used to accelerate many HPC applications across a range of fields in recent years. For large-scale applications that go beyond the capability of one node, manually mixing GPU data movement with MPI communication routines is still the status quo, and its optimization usually requires expertise [10, 16]. In this work, our experience with MPI-ACC [6], our GPU-integrated MPI implementation, shows that the manual hybrid programming model can be replaced with extended MPI support, with additional optimizations automatically made available to developers.

Several research groups have worked on designing and optimizing GPU-integrated data communication libraries. The cudaMPI library studies providing wrapper API functions by mixing CUDA and MPI data movement [21]. Similarly to MPI-ACC, Wang et al. propose to add CUDA [2] support to MVAPICH2 [22] and optimize the internode communication for InfiniBand networks [28]. All-to-all communication [27] and noncontiguous datatype communication [17, 29] have also been studied in the context of GPU-aware MPI. With a focus on intranode communication, our previous work [18, 19] extends transparent GPU buffers support for MPICH [1] and optimizes the cross-PCIe data movement by using shared memory data structures and interprocess communication (IPC) mechanisms. In contrast to those efforts, here we study the synergistic effect between GPU-accelerated MPI applications and a GPU-integrated MPI implementation.

6. CONCLUSION

In this paper, we studied the interactions of GPU-integrated MPI on the complex execution patterns of scientific applications from the domains of epidemiology (GPU-EpiSimdemics) and seismology (FDM-Seismology) on hybrid CPU-GPU clusters, and we presented the lessons learned. By using MPI-ACC as an example implementation, we created new optimization techniques, such as overlapped internode GPU communication with concurrent computations on the CPUs and GPUs and discussed their benefits and tradeoffs. We found that while MPI-ACC's internal pipeline optimization helped improve the end-to-end communication performance to a certain degree, its benefit was less in the context of the entire application. We also showed how MPI-ACC helped in naturally expressing the communication targets that were chosen based on the execution profiles of the tasks at hand. MPI-ACC decoupled the application logic from the low-level GPU communication optimizations, thereby significantly improving scalability and application portability across multiple GPU platforms and generations. Thus, GPU-integrated MPI helps move the GPUs toward being "first-class citizens" in the hybrid clusters. Our results on HokieSpeed, a state-of-the-art CPU-GPU cluster, showed that MPI-ACC can help improve the performance of GPU-EpiSimdemics and FDM-Seismology over the default MPI+GPU implementations by enhancing the CPU-GPU and network utilization. Using the HPCToolkit performance tools, we were able to measure, visualize, and quantify the benefits of MPI-ACC–driven optimizations in our application case studies.

Acknowledgments

This work was supported in part by the DOE grant DE-SC0001770, 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, the NSF PetaApps grant OCI-0904844, the NSF Blue Waters grant OCI-0832603, and by

the NSF grants CNS-0960081, CNS-0546301 and CNS-0958311, as well as 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.

7. REFERENCES

[1] MPICH2. http://www.mcs.anl.gov/research/projects/mpich2/.

[2] NVIDIA CUDA toolkit 4.1. http://developer.nvidia.com/cuda-toolkit-41.

[3] NVIDIA GPUDirect. http://developer.nvidia.com/gpudirect.

[4] TOP500. http://www.top500.org/lists/2012/06/highlights.

[5] ADHIANTO, L., BANERJEE, S., FAGAN, M., KRENTEL, M., MARIN, G., MELLOR-CRUMMEY, J., AND TALLENT, N. R. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency Computation: Practice and Experience 22* (April 2010), 685–701.

[6] AJI, A. M., DINAN, J., BUNTINAS, D., BALAJI, P., FENG, W.-C., BISSET, K. R., AND THAKUR, R. 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).

[7] BARRETT, C. L., BISSET, K. R., EUBANK, S. G., FENG, X., AND MARATHE, M. V. EpiSimdemics: an efficient algorithm for simulating the spread of infectious disease over large realistic social networks. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing* (2008), SC '08.

[8] BERENGER, J. A perfectly matched layer for the absorption of electromagnetic waves. *Journal of Computational Physics 114*, 2 (1994), 185–200.

[9] BISSET, K., AJI, A., MARATHE, M., AND CHUN FENG, W. High-performance biocomputing for simulating the spread of contagion over large contact networks. In *IEEE 1st International Conference on Computational Advances in Bio and Medical Sciences (ICCABS)* (Feb. 2011), pp. 26–32.

[10] BROWN, W. M., WANG, P., PLIMPTON, S. J., AND THARRINGTON, A. N. Implementing molecular dynamics on hybrid high performance computers – short range forces. *Computer Physics Communications 182*, 4 (2011), 898–911.

[11] COLLINO, F., AND TSOGKA, C. Application of the perfectly matched absorbing layer model to the linear elastodynamic problem in anisotropic heterogeneous media. *Geophysics 66*, 1 (2001), 294–307.

[12] ENDO, T., NUKADA, A., MATSUOKA, S., AND MARUYAMA, N. Linpack evaluation on a supercomputer with heterogeneous accelerators. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on* (April 2010), pp. 1–8.

[13] FENG, W.-C., CAO, Y., PATNAIK, D., AND RAMAKRISHNAN, N. Temporal Data Mining for Neuroscience. In *GPU Computing Gems*, W. mei W. Hwu, Ed. Morgan Kaufmann, February 2011. Emerald Edition.

[14] FESTA, G., AND NIELSEN, S. PML absorbing boundaries. *Bulletin of the Seismological Society of America 93*, 2 (2003), 891–903.

[15] GROUP, K. OpenCL 1.2. http://www.khronos.org/opencl/.

[16] HAMADA, T., NARUMI, T., YOKOTA, R., YASUOKA, K., NITADORI, K., AND TAIJI, M. 42 TFlops hierarchical N-body simulations on GPUs with applications in both astrophysics and turbulence. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (2009), ACM.

[17] JENKINS, J., DINAN, J., BALAJI, P., SAMATOVA, N. F., AND THAKUR, R. Enabling Fast, Noncontiguous GPU Data Movement in Hybrid MPI+GPU Environments. In *IEEE International Conference on Cluster Computing (Cluster)* (September 2012).

[18] JI, F., AJI, A., DINAN, J., BUNTINAS, D., BALAJI, P., FENG, W.-C., AND MA, X. Efficient Intranode Communication in GPU-Accelerated Systems. In *The 2nd Intl. Workshop on Accelerators and Hybrid Exascale Systems* (May 2012).

[19] JI, F., AJI, A. M., DINAN, J., BUNTINAS, D., BALAJI, P., THAKUR, R., FENG, W.-C., AND MA, X. DMA-Assisted, Intranode Communication in GPU Accelerated Systems. In *14th IEEE International Conference on High Performance Computing and Communications* (Liverpool, UK, June 2012).

[20] JOSEPH, R., RAVUNNIKUTTY, G., RANKA, S., D'AZEVEDO, E., AND KLASKY, S. Efficient GPU Implementation for Particle in Cell Algorithm. In *IEEE International Parallel Distributed Processing Symposium (IPDPS)* (May 2011), pp. 395–406.

[21] LAWLOR, O. Message passing for GPGPU clusters: CudaMPI. In *IEEE International Conference on Cluster Computing and Workshops, 2009. CLUSTER '09.* (Sept. 2009), pp. 1–8.

[22] LIU, J., JIANG, W., WYCKOFF, P., PANDA, D., ASHTON, D., BUNTINAS, D., GROPP, W., AND TOONEN, B. Design and implementation of MPICH2 over InfiniBand with RDMA support. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International* (April 2004), p. 16.

[23] MA, S., AND LIU, P. Modeling of the perfectly matched layer absorbing boundaries and intrinsic attenuation in explicit finite-element methods. *Bulletin of the Seismological Society of America 96*, 5 (2006), 1779–1794.

[24] MARCINKOVICH, C., AND OLSEN, K. On the implementation of perfectly matched layers in a three-dimensional fourth-order velocity-stress finite difference scheme. *J. Geophys. Res. 108*, B5 (2003), 2276.

[25] NERE, A., HASHMI, A., AND LIPASTI, M. Profiling Heterogeneous Multi-GPU Systems to Accelerate Cortically Inspired Learning Algorithms. In *IEEE International Parallel Distributed Processing Symposium (IPDPS),* (May 2011), pp. 906–920.

[26] NVIDIA. NVIDIA CUDA C Programming Guide version 4.0.

[27] SINGH, A. K., POTLURI, S., WANG, H., KANDALLA, K., SUR, S., AND PANDA, D. K. MPI alltoall personalized exchange on GPGPU clusters: Design alternatives and benefit. In *Workshop on Parallel Programming on Accelerator Clusters (PPAC '11), held in conjunction with Cluster '11* (Sept. 2011).

[28] WANG, H., POTLURI, S., LUO, M., SINGH, A., SUR, S., AND PANDA, D. MVAPICH2-GPU: Optimized GPU to GPU communication for InfiniBand clusters. *International Supercomputing Conference (ISC) '11* (2011).

[29] WANG, H., POTLURI, S., LUO, M., SINGH, A. K., OUYANG, X., SUR, S., AND PANDA, D. K. Optimized non-contiguous MPI datatype communication for GPU clusters: Design, implementation and evaluation with MVAPICH2. In *Proceedings of CLUSTER* (2011), IEEE.

[30] WEIGUO, L., SCHMIDT, B., VOSS, G., AND MULLER-WITTIG, W. Streaming Algorithms for Biological Sequence Alignment on GPUs. *IEEE Transactions on Parallel and Distributed Systems 18*, 9 (Sept. 2007), 1270–1281.

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