MPI-ACC: An Integrated and Extensible Approach to Data Movement in Accelerator-Based Systems

Ashwin M. Aji,* James Dinan,† Darius Buntinas,† Pavan Balaji,† Wu-chun Feng,* Keith R. Bisset,‡ Rajeev Thakur†

*Department of Computer Science, Virginia Tech. {*aaji, feng*}*@cs.vt.edu*
†Math. and Comp. Sci. Div., Argonne National Lab. {*dinan, buntinas, balaji, thakur*}*@mcs.anl.gov*
‡Virginia Bioinformatics Institute, Virginia Tech. *kbisset@vbi.vt.edu*

***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 frameworks, 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's runtime system enables several key optimizations, including pipelining of data transfers and balancing of communication based on accelerator and node architecture. We demonstrate the extensible design of MPI-ACC by using the popular CUDA and OpenCL accelerator programming interfaces. We examine the impact of MPI-ACC on communication performance and evaluate application-level benefits on a large-scale epidemiology simulation.**

***Keywords*-MPI; GPU; CUDA; OpenCL; MPI-ACC**

I. Introduction

Graphics processing units (GPUs) have gained popularity as general-purpose computation accelerators and are now frequently integrated into high-performance computing systems. The widespread adoption of GPU commuting has been due in part to an amalgamation of performance, power, and energy efficiency. In fact, three of the top five fastest supercomputers in the world, according to the November 2011 Top500 list, use GPUs [1].

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) [2] and the Compute Unified Device Architecture (CUDA) [3] or Open Computing Language (OpenCL) [4] 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 frameworks, thus providing 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 GPU-Direct [5] 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-connected systems. MPI-ACC offers a significant improvement in productivity by providing a unified programming interface 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.

- We provide an extensible interface for integrating auxiliary memory systems (e.g., GPU memory) with MPI. The interface retains all of MPI's existing functionality, while providing transparent communication on buffers residing in GPU and host memories. The interface is carefully designed to be interoperable with different accelerator interfaces—including CUDA and OpenCL—without relying on interface-specific features such as unified virtual addressing (UVA), which is available only in CUDA.
- We develop an optimized runtime system that can perform efficient data movement across host and accelerator memories. The MPI-ACC runtime system performs pipelined data movement across I/O links and the system interconnect; leverages vendor- and platform-specific information such as PCIe and NUMA affinity; and caches accelerator metadata. Results indicate that MPI-ACC can provide up to 29% improvement in two-sided GPU-to-GPU communication latency.
- We demonstrate the utility of MPI-ACC using a large-scale epidemiology simulation parallelized for GPU clusters [6], which achieves a 20.25% reduction in the communication overhead. In addition to the expected speedup from the integrated and optimized data movement, MPI-ACC enables several memory access improvements for this application, resulting in an 11.1% improvement to the overall simulation execution time.

This paper is organized as follows. In Section II we provide an overview of the current state of practice for programming GPU-accelerated systems and describe MPI-ACC's impact on programmability. In Section III we present the MPI-ACC

optimized runtime system, and in Section IV we describe its application to a large-scale epidemiology simulation. In Section V we evaluate the communication and application-level performance of MPI-ACC. In Section VI we present related work, and in Section VII we summarize our conclusions.

II. Programming Accelerator-Based Systems

The most commonly used compute accelerators today are GPUs, which are connected to host processor and memory through the PCIe interconnect. These devices contain separate, high-throughput memory subsystems; and data must be explicitly moved between GPU and host memories by using special library DMA transfer operations. Some GPU libraries provide direct access to host memory, but such mechanisms still translate to implicit DMA transfers.

A. GPU Programming Models: CUDA and OpenCL

CUDA [3] and OpenCL [4] are the most commonly used parallel programming models for GPU computing. CUDA is a popular, proprietary GPU programming environment developed by NVIDIA; and OpenCL is an open standard for programming a variety of accelerator platforms, including GPUs, FPGAs, many-core processors, and conventional multicore CPUs. Both CUDA and OpenCL provide explicit library calls to perform DMA transfers from the host to device (H-D), device to host (D-H), device to device (D-D), and optionally host to host (H-H). In both CUDA and OpenCL, DMA transfers involving pinned host memory provide significantly higher performance than using pageable memory.

Despite their similarities, however, CUDA and OpenCL differ significantly in how accelerator memory is used. In OpenCL, device memory allocation requires a valid *context* object. All communication to this device memory allocation must also be performed by using the same context object. Thus, a device buffer in OpenCL has little meaning without information about the associated context. In contrast, context management is implicit in CUDA if the runtime library is used. CUDA data and corresponding `cudaStream_t` objects are explicitly associated with each other or implicitly use the default stream for data transfers. Furthermore, new versions of CUDA (v4.0 or later) also support unified virtual addressing (UVA), where the host memory and all the device memory regions (of compute capability 2.0 or higher) can all be addressed in a single address space. At runtime, the programmer can query whether a given pointer refers to host or device memory via the `cuPointerGetAttribute` function call. This feature is currently CUDA specific; and other accelerator models, such as OpenCL, do not support UVA.

B. MPI+GPU Hybrid Programming Models

Current MPI applications that utilize accelerators must perform data movement in two phases. MPI is used for internode communication of data residing in main memory, and CUDA or OpenCL is used within the node to transfer data between the CPU and GPU memories. In the simple example shown in Figure 1, the additional `host_buf` buffer is to facilitate

```
1  double *dev_buf, *host_buf;
2  cudaMalloc(&dev_buf, size);
3  cudaMallocHost(&host_buf, size);
4  if (my_rank == sender) { /* sender */
5    computation_on_GPU(dev_buf);
6    cudaMemcpy(host_buf, dev_buf, size, ...);
7    MPI_Send(host_buf, size, ...);
8  } else {                 /* receiver */
9    MPI_Recv(host_buf, size, ...);
10   cudaMemcpy(dev_buf, host_buf, size, ...);
11   computation_on_GPU(dev_buf);
12 }
```

Fig. 1. Hybrid MPI+CUDA program with manual data movement. For MPI+OpenCL, `clEnqueueReadBuffer` and `clEnqueueWriteBuffer` would be used in place of `cudaMemcpy`.

MPI communication of data stored in device memory. One can easily see that as the number of accelerators—and hence distinct memory regions per node—increases, manual data movement poses significant productivity challenges.

In addition to productivity challenges, 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. Manual blocking transfers between host and device serialize transfers, resulting in underutilization of the accelerator (PCIe) and network interconnects. Ideally, data movement between the GPU and CPU should be pipelined to fully utilize independent PCIe and network links; however, adding this level of complexity to already complex applications is infeasible. 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.

Integrated support for accelerator memory is therefore a critical feature for improving both the programmability and the performance of MPI programs that use accelerators. Several investigations have begun in this area [7], [8]; however, key challenges remain unsolved, namely, reducing overheads and integrating support for multiple accelerator models within existing MPI. The MVAPICH2-GPU project provides a standard-conforming integrated interface for MPI+CUDA hybrid programming [7], [9]; however, this system relies heavily on CUDA-specific features, such as UVA. Reliance on UVA introduces a new overhead to all MPI operations—regardless of whether buffers reside in GPU memory—because MPI must inspect the location of every buffer at runtime using CUDA's `cuPointerGetAttribute` function. Depending on the result of this query, different code paths will be executed to handle buffers residing in GPU or host memory. This query is expensive relative to extremely low-latency communication times and can add significant overhead to host-to-host communication operations. In Figure 2 we measure the impact of this query on the latency of intranode, *CPU-to-CPU*, communication using MVAPICH v1.8 on the experimental platform described in Section V.

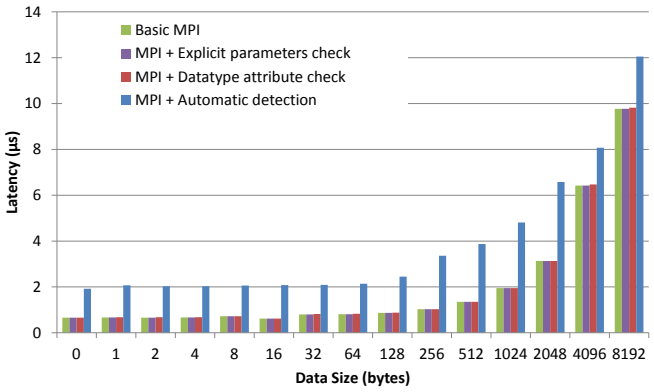


Fig. 2. Overhead of runtime checks incurred by intranode *CPU-CPU* communication operations. The slowdown due to automatic detection (via `cuPointerGetAttribute`) is 23% to 235% depending on the message size, while the slowdown for the datatype attribute check is at most only 3%. The explicit parameter check performs identically to the basic MPI version.

C. Extending MPI with Accelerator Awareness

This paper presents MPI-ACC, the first interface that integrates CUDA, OpenCL, and other models within an MPI-compliant interface while significantly reducing GPU-GPU and CPU-CPU communication overheads. Specifically, we extend MPICH2 [10], an open-source high-performance implementation of the MPI Standard.

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 UVA 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 buffers is mapped into the host's address space or explicitly copied to the host. Given these constraints, several extensions to the MPI interface are possible to enable MPI to correctly identify host memory and device memory pointers.

1) Automatic Detection: One approach to allow MPI to deal with accelerator buffers is to leverage the UVA feature of CUDA to automatically detect device buffers. This method requires no modifications to the MPI interface. As shown in Figure 2, however, the penalty for runtime checking can be significant and is incurred by all operations, including those that require no GPU data movement at all. In addition, this method currently works only with CUDA v4.0 or later and with NVIDIA devices with Compute Capability 2.0 or higher. This is not an extensible approach for other accelerator models such as OpenCL that do not map GPU buffers into the host virtual address space.

2) Explicit Parameters: A second approach is to define new MPI communication functions that include an additional parameter for each buffer, thus allowing the user to specify the buffer type. This method uses static compile-time binding to eliminate runtime penalties. A significant drawback to this design is that it requires a new binding to be created for almost every MPI function.

3) MPI Datatype Attributes: A third approach is to use datatype *attributes*. 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*. These attributes can be used to indicate buffer type and any other information to the MPI library. For convenience, accelerator variants of the built-in datatypes can be defined (e.g., `MPIACC_BUF_TYPE_CUDA` or `MPIACC_BUF_TYPE_OPENCL`). This approach introduces a lightweight runtime attribute check to each MPI operation, but the overhead is negligible, as shown in Figure 2. Moreover, this approach is the most extensible and maintains compatibility with the MPI Standard.

In practice, the second and third approaches perform identically (see Figure 2). It is a matter of user preference which type of is chosen.

III. MPI-ACC: Design and Optimizations

In this section, we introduce the design and optimizations of MPI-ACC. Once MPI-ACC has identified a device buffer, it leverages the PCIe and network link parallelism in order 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.

A. Pipeline Design

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.

For every new GPU data transfer, we request the buffer pool manager for some predetermined number of memory packets from the buffer pool for pipelining. We relinquish the packets back to the buffer pool once the data transfer is complete. This method provides fairness to the utilization of the buffer pool if there are multiple outstanding communication requests. But, how many packets should be requested, and what should the size of each packet be? Perfect pipeline efficiency is achieved

if the CPU-GPU channel and the internode CPU-CPU network channel are both operating at the same bandwidth. At this ideal data transfer rate, 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 use a corresponding number of CUDA streams and OpenCL command queues to perform asynchronous packet transfer between the CPU and GPU.

To calculate the ideal pipeline packet size, we first individually measure the network and PCIe bandwidths at different data sizes (Figure 3), then choose the packet size at the intersection point of the above channel rates, 64 KB for our experimental cluster (section V). 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.

The basic pipeline loop for a "send" operation is as follows ("receive" works the same way, but the direction of the operations is reversed). Before sending 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 find that the GPU synchronization/query calls on *already completed* CPU-GPU copies cause a significant overhead in our experimental cluster, which hurts the effective network bandwidth and forces us to choose a different pipeline packet size. For example, we measure the cost of stream query/synchronization operations as approximately 20 μs, even though the data transfer has been completed. Moreover, this overhead occurs every time a packet is sent over the network, as shown in Figure 3 by the "Effective Network Bandwidth" line. We observe that the impact of the synchronization overhead is huge for smaller packet sizes but becomes negligible for larger packet sizes (4 MB). Also, we find no overlap between the PCIe bandwidth and the effective network bandwidth rates, and the PCIe is always faster for all packet sizes. Thus, we pick the *smallest* packet size that can achieve the peak effective network bandwidth (in our case, this is 1 MB) as the pipeline transfer size for MPI-ACC. Smaller packet sizes ($<$1 MB) cause the effective network bandwidth to be latency bound and are thus not chosen as the pipeline parameters.

We emphasize that we support message transfers of all data sizes in MPI-ACC in order to enhance programmer productivity. In the implementation, however, 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.

B. Dynamic Choice of Pipeline Parameters

The effective data transfer bandwidth out of a node in current heterogeneous clusters may vary significantly. For example, the PCIe interfaces at the communication end points may be different ($\times 16$ vs. $\times 4$), or the GPUs can be of different generations with different DMA capabilities. A common scenario of heterogeneous performance involves multisocket, multicore CPU architectures that may have multiple memory and PCIe controllers (GPU or IB) per socket, where the access latencies vary significantly depending on the affinity of the executing CPU core and the target memory or PCIe controller. The sockets within a die are connected via quick links, such as QPI (Intel) and HT (AMD), and the additional intersocket data transfers hurt performance. For example, if the target memory module controller and the GPU PCIe controller are on different sockets, the GPU-CPU data transfer bandwidth can slow by as much as $2.5\times$ (Figure 4). Similarly, the InfiniBand network bandwidth can slow by 27% because of the varying network controller affinity.

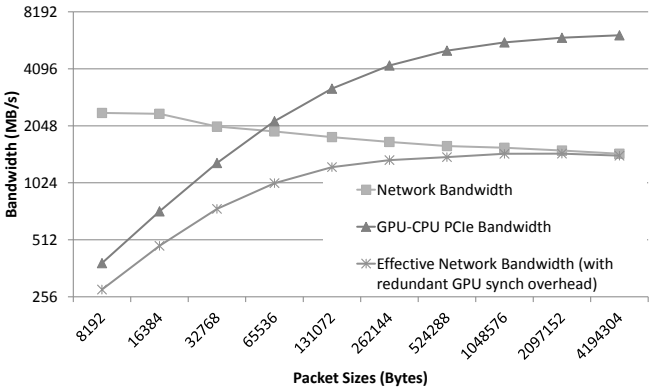


Fig. 3. Choosing the pipeline parameters: network – InfiniBand, transfer protocol – R3.

Because of a combination of these scenarios, the effective bandwidth can be different for the source and the destination. Since the communication performance at each communication stage can change dynamically and significantly, we choose the pipeline parameters (packet size) also dynamically at runtime for both the source and destination processes. We inspect a series of benchmark results; learn the dynamic system characteristics, such as the CPU socket binding; and then apply architecture-aware heuristics to choose the ideal transfer parameters for each communication request, all at runtime.

The sender sends a ready-to-send (RTS) message at the beginning of every MPI communication. The receiver sends a corresponding clear-to-send (CTS) message back, and then the sender begins to transfer the actual payload data. Thus, the sender encapsulates the local pipeline parameters within the RTS message and sends it across the network. The receiver inspects the sender's parameters and also the receiver's system characteristics (e.g., socket binding) and chooses the best pipeline parameters for the current communication transaction, based on our benchmark results and corresponding heuristics. The receiver then sends the CTS message back to the sender along with its chosen pipeline parameters. The sender uses the received pipeline parameters to perform data movement from

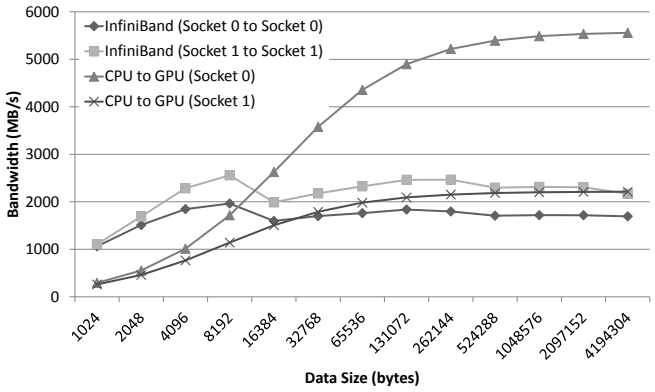


Fig. 4. NUMA and PCIe affinity issues affecting the *effective* bandwidth of CPU-GPU and InfiniBand network transfers. We dynamically choose the pipeline parameters depending on the runtime system characteristics at the source and the destination.

the GPU, as before. In this approach, the two participating processes both first pick an initial pipeline configuration, then coordinate via RTS/CTS messages to converge on a single packet size depending on the effective bandwidth of the participating processes.

C. 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 bby 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's external command queue or create several internal command queues to enable pipelined communication between the device and the host. 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. Also, 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 repeated initialization of these temporary OpenCL objects severely hurts performance because they are expensive operations. Moreover, we believe that in practice the OpenCL programmers are unlikely to use multiple contexts within the same program. Therefore, 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. 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 is considered sufficient in practice.

IV. Application of MPI-ACC to an Epidemiology Simulation Application

GPU-EpiSimdemics is a large-scale application that utilizes GPUs across multiple nodes to simulate the joint evolution of disease dynamics, human behavior, and social networks as an epidemic progresses [6]. We have accelerated the application further by using MPI-ACC instead of explicit internode GPU-GPU data movement. We discuss the relevant data structure and communication operations in GPU-EpiSimdemics and compare the performance of MPI-ACC with hand-tuned approaches to optimizing GPU-GPU communication. In addition to expected improvement in communication performance (presented in Section V-C), we observe significant improvement in the application's execution time because MPI-ACC forces some memory reads and writes to be moved away from the slower host memory to the faster device memory.

A. GPU-EpiSimdemics Algorithm

EpiSimdemics [11], [12] is an interaction-based, high-performance-computing-oriented simulation for studying large-scale epidemics. The computation structure of this implementation consists of three main components: persons, locations, and message brokers. Given a parallel system with N cores, or *processing elements* (PEs), people and locations are first partitioned in a round-robin fashion into N groups denoted by $P_1, P_2, \ldots, P_N$ and $L_1, L_2, \ldots, L_N$, respectively. Each PE then executes all the remaining phases of the EpiSimdemics algorithm on its local data set (P_i, L_i). Each PE also creates a copy of the message broker, denoted by $MB_1, MB_2, \ldots, MB_N$.

Next, a set of *visit messages* is computed for each person P_i for the current simulation day (or iteration). These messages are then sent to each location (which may be on a different PE) via the local message broker. Each location, upon receiving the visit messages, computes the probability of spread of infections for each individual at that location. Outcomes of these computations (infections) are then sent back to the "home" PEs of each person via the local message broker. The infection messages for each person on a PE are merged and processed, and the resulting health state of each infected person is updated. Thus, the simulation is composed of two major phases: *computeVisits* and *computeInteractions*. The messages that are computed as the outputs of one phase are communicated with the appropriate PEs as inputs to the next phase of the simulation. All the PEs in the system are synchronized after each phase in the simulation. These steps are executed for the required number of simulation days (or iterations), and the resulting social network dynamics are analyzed in detail.

GPU-EpiSimdemics is implemented by using MPI; each PE in the simulation corresponds to a separate MPI process. The *computeInteractions* phase of GPU-EpiSimdemics is offloaded

to be executed on the GPU while the remaining computations are performed on the CPU [6]. The *computeInteractions* phase of the simulation, by itself, is isolated in that it does not require any interprocess communication.

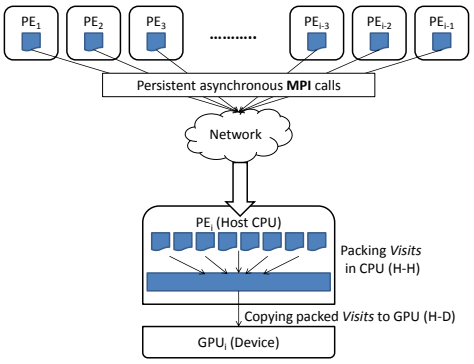
B. Optimizing Internode GPU-GPU Data Movement Using MPI-ACC

The visit messages that are computed as the outputs of the *computeVisits* phase become the inputs to the *computeInteractions* phase of GPU-EpiSimdemics. In the naïve data movement approach, the visit messages are first transferred to the remote CPU memory, and then the remote node copies them across the PCIe bus to the local GPU memory. If there are N PEs in the system, then each PE receives up to N visit message fragments. All-to-all or scatter/gather transfers of this data are not feasible because the number of visit messages is not known beforehand. Thus, fixed-size, *persistent* buffers are repeatedly reused, potentially resulting in gaps or fragments if fewer than N messages are received. Once all messages are received, they are packed into a single, contiguous visit buffer before the next phase of the simulation. The manual data movement+packing can be done in two ways: (1) pack the data completely on the CPU *before* copying to GPU memory (Figure 5a), or (2) pack the data simultaneously *during* the CPU-GPU transfer (Figure 5b).

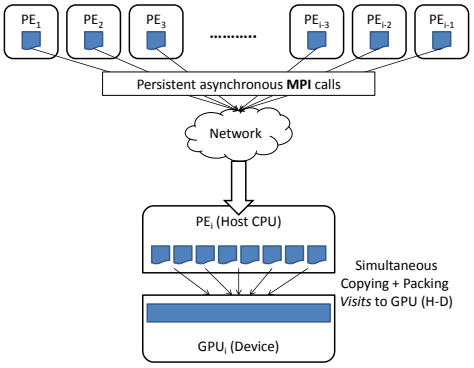
As part of the communication optimization process, we first replace the asynchronous MPI and `cudaMemcpy` calls in the receiver logic with the unified MPI-ACC calls; in other words, the visit messages are directly received into the remote GPU memory via MPI-ACC's integrated interface and internal pipeline mechanism (Figure 5c). The use of MPI-ACC causes the following changes to the simulation's execution logic. First, the receive buffers of the visit message fragments are created and initialized in the GPU memory, and not in the CPU memory. Second, the received visit message fragments are packed in the GPU itself. Third, the explicit transfer of the contiguous visit buffer from the host to the device is completely avoided. The PCIe bottleneck is significantly alleviated as a result ot the pipelining mechanism within MPI-ACC. The *computeInteractions* phase is then executed on the GPU following the interphase barrier, as before.

V. Evaluation

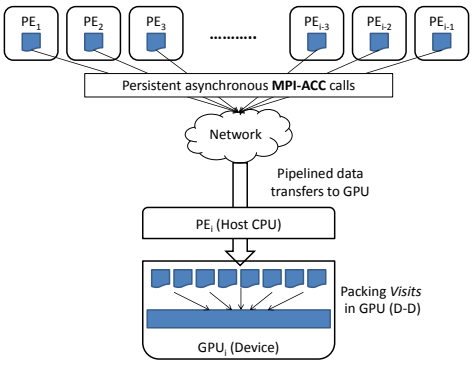
All our experiments are run on a four-node GPU cluster, where each node has a dual-socket oct-core AMD Opteron 6134 (Magny-Cours family) processor. Each node is also attached to two NVIDIA Tesla C2050 GPUs, which belong to the GF100 Fermi architecture family (compute capability 2.0). We use one CPU core and one GPU device per node for all our experiments. For our benchmark tests, we use two of the four nodes to perform the internode, inter-GPU latency and bandwidth experiments. Each CPU node has 32 GB of main memory, and each GPU has 3 GB of device memory. We use the CUDA v4.0 toolkit with the driver v285.05.23 as the GPU management software. MPICH is compiled with GCC v4.1.2 and on the Linux kernel v2.6.35.



(a) Manual MPI+CUDA optimizations – basic scheme. The *visit* messages are first packed on the host, then copied to the device.



(b) Manual MPI+CUDA optimizations – advanced scheme. The *visit* messages are simultaneously packed and copied to the device.



(c) MPI-ACC optimizations. The *visit* messages are received directly in the device and then packed.

Fig. 5. Applying MPI-ACC to GPU-EpiSimdemics.

A. Impact of Pipelined Data Transfer

In Figure 6 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 29% and is up to 14.6% better than the manual pipelined implementation, especially for larger data transfers. The man-

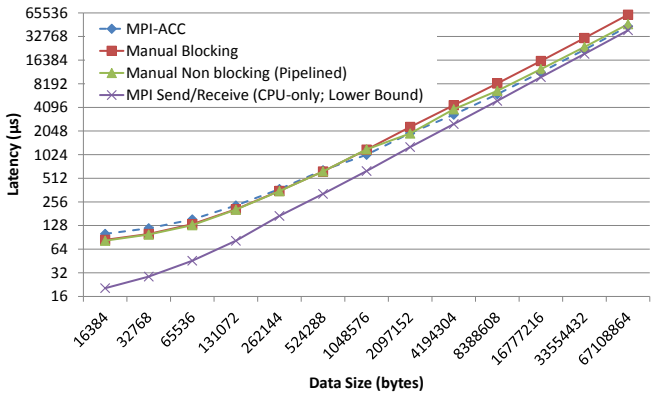


Fig. 6. Internode communication latency for GPU-to-GPU (CUDA) data transfers. Similar performance is observed for OpenCL data transfers. The chosen pipeline packet size for MPI-ACC is 1 MB.

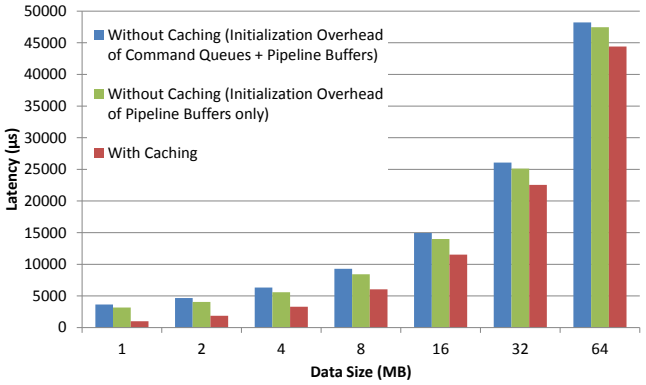
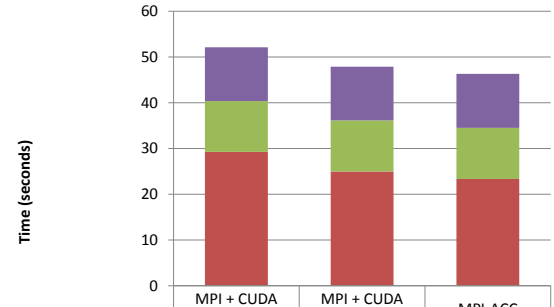


Fig. 7. OpenCL issues and optimizations: MPI-ACC performance with and without OpenCL object caching.

ual pipelined implementation performs poorly because of the repeated handshake messages (RTS and CTS) that are sent back and forth across the network before the data transfer. We also show that the performance of MPI-ACC is worse than the manual approaches for the messages that are smaller than the pipeline packet. The cause is the additional overhead of maintaining the data structures and bookkeeping logic required for pipelining. In practice, however, we use the MPI-ACC pipelining logic only to transfer data that is larger than the packet size, and we fall back to the default blocking approach for smaller data sizes.

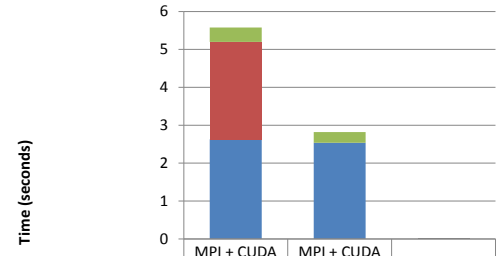
B. Impact of OpenCL Object Caching

Figure 7 shows that the OpenCL caching optimization improves performance from 8% for larger data sizes (64 MB) to 72.6% for smaller data sizes (1 MB). Even where the programmers provide their custom command queue, the pipeline buffers still have to be created for every MPI communication call, and so caching improves performance, as seen in the difference between the "initialization overhead of pipeline buffers only" and "with caching" data points.



	MPI + CUDA (Basic)	MPI + CUDA (Advanced)	MPI-ACC
Other Phases	11.75	11.71	11.80
Phase II (Interactions)	11.07	11.16	11.17
Phase I (Visits)	29.28	24.99	23.35

Fig. 8. MPI-ACC applied to GPU-EpiSimdemics.



	MPI + CUDA (Basic)	MPI + CUDA (Advanced)	MPI-ACC
D-D Copy (Packing)	0	0	0.003
GPU Receive Buffer Init	0	0	0.024
H-D Copy	0.382	0.282	0
H-H Copy (Packing)	2.570	0	0
CPU Receive Buffer Init	2.627	2.537	0

Fig. 9. Analyzing the *computeVisits* phase of GPU-EpiSimdemics

C. Application Evaluation: GPU-EpiSimdemics

GPU-EpiSimdemics was run on a problem consisting of a synthetic subpopulation from the state of Delaware (DE) with 247,876 persons, 204,995 locations, and an average of about 1.5 million visits being generated during every simulated day. All results are based on running GPU-EpiSimdemics for 55 simulated days. This data set was chosen because it fits in both the CPU and GPU memories on a single node and can be used as the basis for other performance calculations.

Figure 8 shows the execution profiles of GPU-EpiSimdemics before and after applying the MPI-ACC optimizations. We can see that, upon applying MPI-ACC to the simulation, the *computeVisits* phase has been accelerated by 20.25% and the overall simulation by 11.1% when compared with the basic data transfer scheme.

To understand the reason behind the performance benefits, we further profile the *computeVisits* phase and show the results in Figure 9. In particular, we isolate and measure the effect of the changes due to MPI-ACC—namely, changes in the buffer creation and initialization, buffer packing, and CPU-GPU data movement. We see that the time taken for buffer creation and packing on the GPU is two orders of magnitude faster than on the CPU. We attribute this result to the much faster GDDR5 RAM on the GPU device.

Without MPI-ACC, the experienced application programmer can certainly implement the pipelined data transfers above the MPI layer. However, this approach always leads to poor performance because of the repeated overhead of the MPI handshaking message exchanges. Also, the programmer is always forced to create and manage all the communication buffers on the slower CPU memory, a costly operation. Moreover, in our application, we see that the time taken for the blocking CPU-GPU buffer copy in the manual approaches (MPI+CUDA – basic and advanced schemes) is only about 1.3% of the *computeVisit*'s execution time, and therefore pipelining alone does not provide overall improvement. MPI-ACC forces the receive-buffer creation and packing routines to be moved to and executed on the faster GPU device memory, thus improving overall performance.

VI. Related Work

MVAPICH [9] is another implementation of MPI based on MPICH and is optimized for RDMA networks such as InfiniBand. MVAPICH2-GPU, which is the latest release of MVAPICH (v1.8), includes support for transferring CUDA memory regions across the network [7] (point-to-point, collective and one-sided communications). In order to use this, however, each participating system should have an NVIDIA GPU of compute capability 2.0 or higher and CUDA v4.0 or higher, because MVAPICH2-GPU leverages the UVA feature of CUDA [3]. On the other hand, MPI-ACC takes a more portable approach: we support data transfers among CUDA [3], OpenCL [4], 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, CPUs from IBM and Intel, 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 [13] is a library that helps improve programmer productivity when moving data between GPUs across the network. It provides a wrapper interface around the existing MPI and CUDA calls. Our contribution conforms to the MPI Standard, and our implementation removes the overhead of communication setup time, while maintaining productivity.

DCGN [14] is a novel programming environment 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. The actual control and data message transfers are handled by the underlying runtime layer, which hides the PCIe transfer details from the programmer. Our contribution is orthogonal to DCGN's in that we retain the original MPI communication and execution model while hiding the details of third-party CPU-GPU communication libraries from the end user.

VII. 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 implemented several optimizations in MPI-ACC, such as pipelining, dynamic adjustment of pipeline parameters based on PCIe affinity and NUMA effects, and efficient management of CUDA and OpenCL resource objects. We demonstrated 29% improvement in performance in GPU-to-GPU data communication compared with the manual blocking approach. We also applied MPI-ACC to GPU-EpiSimdemics, a large-scale epidemiology simulation, and achieved a performance improvement of 11.1%, where the communication was enhanced by 20.25%.

Acknowledgment

This work was supported in part by the U.S. Department of Energy contract DE-AC02-06CH11357, NSF grant I/UCRC IIP-0804155 via the NSF Center for High-Performance Reconfigurable Computing, NSF grant MRI-0960081, NSF grant CSR-0916719, NSF PetaApps Grant OCI-0904844, DTRA CNIMS Contract HDTRA1-11-D-0016-0001, and NIH MIDAS Grants 2U01GM070694-09 and 3U01FM070694-09S1.

References

[1] "The Top500 Supercomputer Sites," http://www.top500.org.

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

[3] NVIDIA, "CUDA," http://www.nvidia.com/object/cuda_home_new.html.

[4] Aaftab Munshi, "The OpenCL Specification," 2008, http://www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf.

[5] Mellanox and NVIDIA, "NVIDIA GPUDirect Technology – Accelerating GPU-based Systems," 2010, http://www.mellanox.com/pdf/whitepapers/TB_GPU_Direct.pdf.

[6] K. Bisset, A. M. Aji, M. Marathe, and W.-c. Feng, "High-Performance Biocomputing for Simulating the Spread of Contagion over Large Contact Networks," in *BMC Genomics*, April 2012, to appear.

[7] H. Wang, S. Potluri, M. Luo, A. Singh, S. Sur, and D. Panda, "MVAPICH2-GPU: Optimized GPU to GPU communication for InfiniBand clusters," *International Supercomputing Conference (ISC) '11*, 2011.

[8] F. Ji, A. M. Aji, J. Dinan, D. Buntinas, P. Balaji, W. Feng, and X. Ma, "Efficient Intranode Communication in GPU-Accelerated Systems," in *The Second International Workshop on Accelerators and Hybrid Exascale Systems (ASHES)*, May 2012, to appear.

[9] "MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE," http://mvapich.cse.ohio-state.edu/.

[10] Argonne National Laboratory, "MPICH2," http://www.mcs.anl.gov/mpi/mpich2.

[11] 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 *SC '08: ACM/IEEE Conference on Supercomputing*. Piscataway, NJ: IEEE Press, 2008, pp. 1–12.

[12] K. Bisset, X. Feng, M. Marathe, and S. Yardi, "Modeling Interaction between Individuals, Social Networks and Public Policy to Support Public Health Epidemiology," in *Proceedings of the 2009 Winter Simulation Conference*, Dec. 2009, pp. 2020–2031.

[13] O. Lawlor, "Message Passing for GPGPU Clusters: CudaMPI," in *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, Oct. 31 - Sept. 4 2009, pp. 1–8.

[14] J. Stuart and J. Owens, "Message Passing on Data-Parallel Architectures," in *IEEE International Symposium on Parallel Distributed Processing*, May 2009, pp. 1–12.