Synchronization and Ordering Semantics in Hybrid MPI+GPU Programming

Ashwin M. Aji,* Pavan Balaji,† James Dinan,† Wu-chun Feng,* Rajeev Thakur†

*Dept. of Computer Science, Virginia Tech {aaji, feng}@cs.vt.edu
†Math. and Comp. Sci. Div., Argonne National Lab. {balaji, dinan, thakur}@mcs.anl.gov

***Abstract*—Despite the vast interest in accelerator-based systems, programming large multinode GPUs is still a complex task, particularly with respect to optimal data movement across the host-GPU PCIe connection and then across the network. In order to address such issues, GPU-integrated MPI solutions have been developed that integrate GPU data movement into existing MPI implementations. Currently available GPU-integrated MPI frameworks differ in aspects related to the buffer synchronization and ordering semantics they provide to users. The noteworthy models are (1) unified virtual addressing (UVA)–based approach and (2) MPI attributes–based approach. In this paper, we compare these approaches, for both programmability and performance, and demonstrate that the UVA-based design is useful for isolated communication with no data dependencies or ordering requirements, while the attributes-based design might be more appropriate when multiple interdependent MPI and GPU operations are interleaved.**

***Index Terms*—MPI; GPGPU; Unified Virtual Addressing; CUDA; OpenCL; MPI-ACC**

I. Introduction

Graphics processing units (GPUs) have gained widespread use as general-purpose computational accelerators and have been studied extensively across a broad range of scientific applications [1]–[3]. 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 the fastest supercomputers in November 2012 employed general-purpose accelerators, 53 of which were GPUs [4].

Despite the growing prominence of accelerators in HPC, programming large systems equipped with GPUs is still a complex task. In particular, when data has to be moved from a GPU on one node to a GPU on another, such movement of data must be coordinated across the host-to-GPU PCIe connection on the source node, over the network, and finally over the GPU-to-host PCIe connection on the destination node. This already complex task becomes even more overwhelming when pipelining of data, topology awareness with respect to the GPU proximity to the network, and vendor-specific features such as GPUDirect [5] need to be taken into account for optimal performance.

In order to address such issues, a number of GPU-integrated Message Passing Interface (MPI) [6] solutions have been recently developed [7]–[11]. These solutions integrate data movement from dominant GPU programming models such as Compute Unified Device Architecture (CUDA) [12] or Open Computing Language (OpenCL) [13] into the MPI implementation. Thus, the MPI implementation can move data from host memory to host memory (like traditional MPI), from GPU memory to GPU memory, or any other combination such as from GPU memory on one process to host memory to another. Such solutions also bring the rich set of MPI functionality, such as moving noncontiguous data through derived datatypes [11], to GPU data movement.

While most GPU-integrated MPI frameworks are broadly similar to each other, they differ in subtle aspects related to the kind of buffer semantics they allow of users and consequently the buffer synchronization and ordering semantics they provide to users. In this paper, we investigate the design differences between two such semantics that are found in current GPU-integrated MPI frameworks: (1) unified virtual addressing (UVA [14])–based approach and (2) MPI attributes–based approach.

We find that the UVA-based approach does not allow users to provide enough semantic information about GPU stream ordering and completion semantics. This makes them fundamentally restricted to a model where data movement is completely unordered from other data accesses or computation being carried out on the GPU, thus placing the onus of data synchronization on the user. The MPI attributes–based approach, on the other hand, allows users to provide richer information to the MPI implementation that it can take advantage of for performance and correctness without forcing the user to manage these synchronization semantics. Our experiments with benchmarks using multiple GPU streams and MPI communication operations show that while the UVA-based design is useful for isolated point-to-point communication with no data dependencies or ordering requirements, the attribute-based design might be more appropriate when multiple interdependent MPI and GPU operations are interleaved. Further, in some cases, applications using the attribute-based design can outperform the UVA design by up to 34.2%.

The rest of the paper is organized as follows. We provide background on relevant technologies in Section II. In Section III, we present details on how GPU-integrated MPI frameworks work and the models they use. Synchronization semantics that are used in GPU-integrated MPI frameworks and their impact are discussed in Section IV. Experimental re-

sults and their analysis are presented in Section V. Section VI discusses the limitations of current hardware in this area and provides a peek at new features in upcoming hardware. Other relevant literature related to this paper is presented in Section VII. Our conclusions are presented in Section VIII.

II. Background

In this section we briefly discuss two types of programming models: GPU programming models and MPI+GPU hybrid programming models.

A. GPU Programming Models: CUDA and OpenCL

Most of today's GPUs are connected to the host processor and memory through the PCIe interconnect. The high-end GPUs typically contain separate, high-throughput memory subsystems (e.g., GDDR5); 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.

CUDA [12] and OpenCL [13] are two of the commonly used *explicit* GPU programming models, where GPU-specific code is written to be executed exclusively on the GPU device. 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 multi-core 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 does using pageable memory.

1) GPU Streams and Synchronization Semantics: GPUs have hardware queues for enqueueing GPU operations; for example, NVIDIA GPUs (compute capability 2.0 and above) have one hardware queue each for enqueueing kernels, D-H data transfers, and H-D data transfers. In this way, one can potentially overlap kernel execution with H-D and D-H transfers simultaneously. In addition, CUDA and OpenCL both provide GPU workflow abstractions, called *streams* (cudaStream_t) and *command queues* (cl_command_queue).[1] A GPU stream denotes a sequence of operations that execute in issue order on the GPU [15]. Operations from different streams can execute concurrently and may be interleaved, while operations within the same stream are processed serially. Synchronization between streams is explicit, whereas the synchronization within a stream is implicit. Also, all the stream operations are asynchronous with respect to the host CPU. We note that if a data element is shared among multiple streams, say one stream for kernel execution and another for D-H transfers, the streams must be explicitly synchronized for correctness; otherwise the behavior is undefined.

[1]CUDA streams and OpenCL command queues are referred to as *GPU streams* henceforth in this paper.

2) GPU Data Representation and Address Spaces: Despite their apparent similarities, however, CUDA and OpenCL differ significantly in how accelerator memory is used and how data buffers are created and modified. In OpenCL, device memory allocation requires a valid *context* object. All processing and 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.

In OpenCL, data is encapsulated by a cl_mem object, whereas data is represented by a void * in CUDA. CUDA (v4.0 or later) also supports 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 by a single address space. At runtime, the programmer can use the cuPointerGetAttribute function call to query whether a given pointer refers to host or device memory. The UVA 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. Consider a simple example where the sender computes on the GPU and sends the results to the receiver GPU, which then does some more computations on the GPU. One can implement this logic in several ways using the hybrid MPI+GPU programming model as shown in Figure 1. In this simple set of examples, the additional host_buf buffer is used *only* to facilitate 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 and performance challenges.

Figure 1a describes the manual blocking transfer logic between host and device, which serializes GPU execution and data transfers, resulting in underutilization of the PCIe and network interconnects. Figure 1b shows how the data movement between the GPU and CPU can be pipelined to fully utilize the independent PCIe and network links. However, adding this level of code complexity to already complex applications is impractical and can be error prone. 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 [5]).

Synchronization Semantics in MPI+GPU Models: In the hybrid programming model with interleaved GPU and MPI operations, the programmer must adhere to the programming semantics of both models. When GPU data transfer operations

```
double *dev_buf, *host_buf;
cudaMalloc(&dev_buf, size);
cudaMallocHost(&host_buf, size);
if (my_rank == sender) { /* sender */
  computation_on_GPU(dev_buf);
  /* implicit GPU sync for the default CUDA stream */
  cudaMemcpy(host_buf, dev_buf, size, ...);
  /* dev_buf is reused; async GPU kernel launch */
  more_computation_on_GPU(dev_buf);
  MPI_Send(host_buf, size, ...);
}
```

(a) Hybrid MPI+CUDA program with manual synchronous data movement (sender's logic only). This approach loses data transfer performance but gains a bit when the second GPU kernel is overlapped with MPI.

```
double *dev_buf, *host_buf;
cudaStream_t kernel_stream, streams[chunks];
cudaMalloc(&dev_buf, size);
cudaMallocHost(&host_buf, size);
if (my_rank == sender) { /* sender */
  computation_on_GPU(dev_buf, kernel_stream);
  /* explicit GPU sync between GPU streams */
  cudaStreamSynchronize(kernel_stream);
  for(j=0;j<chunks;j++) {
    cudaMemcpyAsync(host_buf+offset, dev_buf+offset,
                    D2H, streams[j], ...);
  }
  for(j=0;j<chunks;j++) {
    /* explicit GPU sync before MPI */
    cudaStreamSynchronize(streams[j]);
    MPI_Isend(host_buf+offset, ...);
  }
  /* explicit MPI sync before GPU kernel */
  MPI_Waitall();
  more_computation_on_GPU(dev_buf);
}
```

(b) Hybrid MPI+CUDA program with manual asynchronous data movement (sender's logic only). This approach loses productivity but gains data transfer performance. But, the second GPU kernel is not overlapped with MPI.

Fig. 1: Tradeoffs with hybrid MPI+CUDA program design. For MPI+OpenCL, clEnqueueReadBuffer and clEnqueueWriteBuffer would be used in place of cudaMemcpy. Similarly, clGetEventInfo or clFinish would be used in place of cudaStreamSynchronize.

on GPU streams (D-H) are followed by MPI operations on the copied host data, the programmer *explicitly* waits or checks the status of the GPU streams before calling MPI. The reason is that MPI operates on ready host buffers and, in this case, the host buffer will be undefined until the GPU stream has completed its D-H operation. Similarly, if nonblocking MPI operations are followed by GPU data transfer operations on the same data, the programmer *explicitly* waits for the completion of MPI before performing the GPU data transfer, as shown in Figure 1.

GPU operations are performed only on GPU data, while MPI operations are performed only on CPU data. When data changes devices (i.e., GPU data is copied to the CPU), the programming model also changes from CUDA/OpenCL to MPI. However, explicit synchronization between GPU operations and MPI is required only for dependent data operations.

```
double *dev_buf, *host_buf;
if (my_rank == sender) { /* send from GPU (CUDA) */
  MPI_Send(dev_buf, ...);
} else { /* receive into host */
  MPI_Recv(host_buf, ...);
}
```

(a) UVA-based design: example MPI code where a device buffer is sent and received as a host buffer.

```
double *cuda_dev_buf; cl_mem ocl_dev_buf;
/* initialize a custom type */
MPI_Type_dup(MPI_CHAR, &type);
if (my_rank == sender) { /* send from GPU (CUDA) */
  MPI_Type_set_attr(type, BUF_TYPE, BUF_TYPE_CUDA);
  MPI_Send(cuda_dev_buf, type, ...);
} else { /* receive into GPU (OpenCL) */
  MPI_Type_set_attr(type, BUF_TYPE, BUF_TYPE_OPENCL);
  MPI_Recv(ocl_dev_buf, type, ...);
}
MPI_Type_free(&type);
```

(b) MPI Attribute-based design: example MPI code where a device CUDA buffer is sent and received as an OpenCL device buffer.

Fig. 2: Design of GPU-integrated MPI frameworks.

III. Design of GPU-Integrated MPI Frameworks

While, conceptually, most GPU-integrated MPI frameworks are broadly similar to each other, they differ with respect to the user programming semantics they provide. Specifically, how GPU execution and communication integrates with MPI communication can be different for different frameworks. In this section, we discuss two such programming semantics that are found in current GPU-integrated MPI frameworks: (1) UVA-based design and (2) MPI attributes–based design.

a) UVA-based design: UVA is a CUDA-specific concept that allows information about the buffer type to be encoded inside a void * size argument. This allows both the host memory and the GPU memory to be represented within a common 64-bit virtual address space. In such a model, the user would pass a void * communication buffer argument to MPI, as it would do in a traditional MPI library (Figure 2a). The MPI implementation would, internally, query for the buffer type attribute using CUDA's cuPointerGetAttribute function. With this, the MPI implementation can identify whether the buffer resides on host memory or on GPU memory and thus decide whether to perform a pipelined data transfer for GPU data over the PCIe and network interconnects or to fall back to the traditional CPU data transfer logic. The cuPointerGetAttribute function can also be used to query the actual GPU device number on which the buffer resides. The MVAPICH2-GPU implementation [16], for instance, uses the UVA model. However, the cuPointerGetAttribute function is expensive relative to extremely low-latency communication times and can add significant overhead to host-to-host communication operations. Figure 3 shows the impact of this query on the latency of intranode, *CPU-to-CPU*, communication using MVAPICH v1.8 on our experimental platform described in Section V.

Apart from the obvious downside that UVA is CUDA-

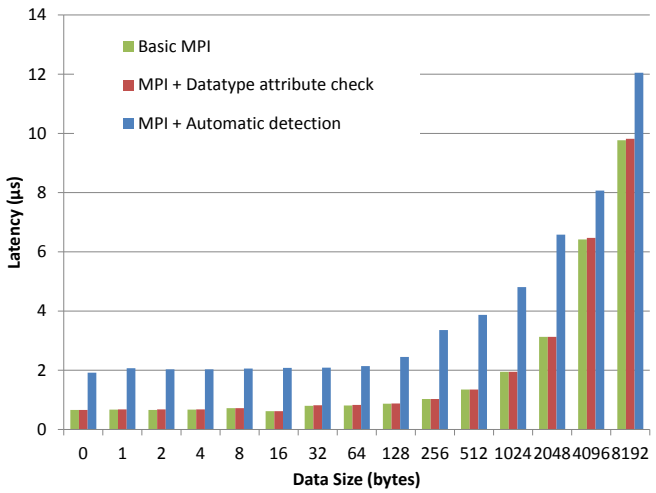


Fig. 3: Overhead of runtime checks incurred by intranode *CPU-CPU* communication operations. The slowdown from automatic detection (via cuPointerGetAttribute) is 23% to 235%, while the slowdown from the attribute check is at most only 3%.

specific and is not relevant to other programming models such as OpenCL, this approach is fundamentally limited by the amount of information that can be passed by the user to MPI. Specifically, only information that can be encoded within a 64-bit space can be passed to the MPI implementation. For example, the user cannot inform the MPI implementation which stream (in CUDA) or command queue (in OpenCL) can be used for the data transfer or whether MPI needs to order the data transfer relative to an event on a stream or command queue.

b) MPI communicator or datatype attributes–based design: Another programming semantic that can be exposed by the MPI implementation is to use MPI communicator or datatype *attributes*. MPI communicators specify a collection of MPI processes and a communication context. 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 communicators or datatypes through attributes. These attributes can be used to indicate buffer type and any other information to the MPI library (Figure 2b). Communicator attributes allow processes to mark a communicator as "special" in that it would always move data to/from GPUs. However, they are restricted in that they cannot move data from the GPU on one process to host memory of another process. Datatype attributes allow processes to perform a specific communication to/from GPU memory or host memory and thus are fully generic and can describe any CPU/GPU communication pattern desired by the user. This approach introduces a lightweight runtime attribute check to each MPI operation, but the overhead is negligible, as shown in Figure 3. Moreover, this approach is more extensible and maintains compatibility with the MPI standard. The MPI-ACC framework [8] uses this communication model.

An important capability of this model is that there is no restriction on the amount of information that the user can pass to the MPI implementation. The MPI implementation can define attributes for the buffer type, buffer locality (e.g., which GPU), which stream to use, ordering semantics, or basically anything else. The MPI implementation can allow some of these attributes to be optional for the user, for programming convenience, without restricting the user from providing it as needed.

IV. Synchronization Semantics in GPU-Integrated MPI Frameworks

In this section, we describe the synchronization and ordering semantics when GPU execution and data movement operations are interleaved with GPU-integrated MPI libraries.

A. Synchronization Semantics of the UVA-Based Design

In the naïve MPI+GPU programming models, the synchronization semantics occurred only when the data changed devices; that is, if GPU data was copied to the CPU, the programming semantics would also change from implicit synchronization in CUDA/OpenCL to explicit synchronization in MPI or vice versa. However, since GPU-integrated MPI can operate directly on GPU data, the synchronization semantics must be carefully defined. If GPU operations on a CUDA stream, such as CUDA kernels or H-D transfers, are followed by direct MPI operations on the same data (i.e., if there is data dependence), there should be some form of synchronization (explicit or implicit) between the GPU operation and the MPI call even though the data has not changed devices. If the MPI call does not have any data dependence with the preceding GPU operation, then synchronization is not needed.

In the GPU programming model, however, streams are used to implicitly indicate data dependence and maintain execution order semantics. We could imagine MPI's data-dependent operations as GPU operations that belong to the same stream as the preceding GPU operation, but it is impossible to pass additional information, such as the dependent GPU stream, to the UVA-based MPI model by just a void * argument. Since the UVA-based model has no mechanism to implicitly express data dependence, GPU operations and their dependent MPI function calls are required to *explicitly* synchronize for correctness. On the other hand, if a GPU operation follows a nonblocking MPI operation, MPI semantics requires that MPI be *explicitly* waited upon before reusing the data.

Consider a synthetic MPI application example with N processes, where MPI process 0 computes a large array on its local GPU and sends chunks of the array directly from the GPU to the remaining $N-1$ processes, similar to a scatter operation. For the sake of argument, let us also assume that the GPU array computation is data parallel, that is, the array elements can be processed independently and can be pipelined. This example application can be implemented by using the UVA-based design in a couple of ways as shown in Figure 4, where the GPU kernel execution is pipelined with data movement, but MPI can be synchronous or asynchronous. Clearly, while the example in Figure 4a is simpler, it is also

```
cudaStream_t myStream[N];
for(rank = 1; rank < N; rank++) {
 fooKernel<<<b, t, myStream[rank]>>>(dev_buf+offset);
}
for(rank = 1; rank < N; rank++) {
 /* explicit GPU stream sync before MPI */
 cudaStreamSynchronize(myStream[rank]);
 MPI_Send(dev_buf+offset, rank, ...);
}
```

(a) Simple UVA-based design: explicit GPU synchronization with synchronous MPI.

```
cudaStream_t myStream[N];
int processed[N] = {1, 0};
for(rank = 1; rank < N; rank++) {
 fooKernel<<<b, t, myStream[rank]>>>(dev_buf+offset);
}
numProcessed = 0; rank = 1;
while(numProcessed < N - 1) {
 /* explicit GPU stream query before MPI */
 if (cudaStreamQuery(myStream[rank])==cudaSuccess) {
    MPI_Isend(dev_buf+offset, rank, ...);
    numProcessed++;
    processed[rank] = 1;
 }
 MPI_Testany(...); /* check progress */
 flag = 1;
 if(numProcessed < N - 1) /* find next rank */
   while(flag) {
    rank=(rank+1)%N; flag=processed[rank];
   }
}
MPI_Waitall();
```

(b) Advanced UVA-based design: explicit GPU synchronization with asynchronous MPI.

Fig. 4: Data-dependent MPI+GPU program with explicit GPU synchronization and designed with UVA-based MPI.

less efficient because of the blocking GPU and MPI calls. Moreover, while the GPU kernels in the different streams can finish in any order, the program waits for them in issue order. This unnecessary wait is removed in Figure 4b, but the code becomes more complex.

Disadvantages of the UVA-Based Design: While the UVA-based design provides an ideal API by perfectly conforming to the MPI standard for end-to-end CPU-GPU communication, its code semantics forces explicit synchronization between data-dependent (ordered) and interleaved GPU and MPI operations. The MPI implementation can potentially avoid the explicit synchronization semantics by conservatively invoking cudaDeviceSynchronize before performing data movement, but this approach will obviously hurt performance and is impractical. Moreover, the UVA-based design is not extensible for other accelerator models such as OpenCL that do not support UVA, because it is impossible to pass the cl_context and cl_command_queue arguments to MPI through just the void * argument.

B. Synchronization Semantics of the MPI Attribute-Based Design

Since MPI can directly operate on GPU data, the synchronization semantics of the MPI attribute-based model must also be carefully defined, that is, explicit vs. implicit. Of course,

```
cudaStream_t myStream[N];
for(rank = 1; rank < N; rank++) {
 fooKernel<<<b, t, myStream[rank]>>>(dev_buf+offset);
 /* implicit GPU stream sync before MPI */
 MPI_Type_dup(MPI_CHAR, &new_type);
 MPI_Type_set_attr(new_type, BUF_TYPE, BUF_TYPE_CUDA);
 MPI_Type_set_attr(new_type, STREAM_TYPE, myStream[rank]);
 MPI_Isend(dev_buf+offset, new_type, rank, ...);
 MPI_Type_free(&new_type);
}
/* explicit MPI sync */
MPI_Waitall();
```

Fig. 5: Data-dependent MPI+GPU program designed with the MPI attribute-based design of MPI-ACC. The example showcases *implicit* GPU synchronization with asynchronous MPI.

one can treat the attribute-based model just like the UVA-based model and introduce explicit synchronization semantics between data-dependent GPU and MPI calls. But, we can do better with this model because there is no restriction in the amount of information that the user can pass to the MPI implementation via the MPI attribute metadata. Since GPU streams implicitly indicate data dependence on GPUs, the programmer can now pass the stream parameter itself as one of the MPI attributes. The MPI implementation can use this stream information to perform additional optimizations for best performance. For example, if a stream parameter is associated with an asynchronous MPI_Isend call, the stream could be added to a stream pool that is periodically queried for completion instead of blocking on cudaStreamSynchronize immediately. The MPI implementer is free to apply different heuristics and additional optimizations on the stream parameter, as needed. Since there exists a mechanism in the attribute-based model to implicitly express data dependence, GPU operations and their dependent MPI function calls can be either implicitly or explicitly synchronized for correctness. On the other hand, if a GPU operation follows a nonblocking MPI operation, MPI semantics requires that MPI be explicitly waited upon before reusing the data.

The synthetic MPI application example from Figure 4 can now be easily implemented in MPI-ACC by using the attribute-based design and implicit synchronization as shown in Figure 5. Note that the stream parameter that is passed to MPI_Isend is different for each loop iteration (or rank) and we have to set that attribute in every loop. The GPU buffer attribute type is constant and is set once before the loop execution. This example is fully nonblocking with asynchronous and interleaved GPU and MPI operations. With this design, we can do away with the complex logic of looping over the GPU streams and explicitly coordinating between the GPU and MPI operations. Instead, by passing more information to MPI, the synchronization semantics can be implicitly controlled within the MPI implementation.

Design of Stream Synchronization in MPI-ACC: Figure 6 illustrates our approach to synchronizing GPU and MPI operations within MPI-ACC. We consider only GPU-specific MPI operations for this discussion; the CPU data

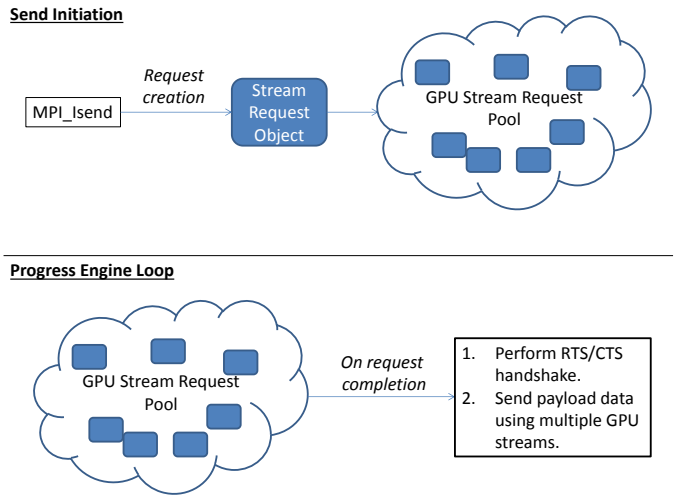


Fig. 6: MPI-ACC's design for MPI+GPU synchronization. Example: MPI_Isend.

is handled separately as before. When the programmer initiates an asynchronous MPI call on a GPU buffer with a dependent stream parameter, we have two options: (1) we can use the application's stream itself for the data transfers, which means that we avoid the complex management of streams and their synchronization within MPI, or (2) we can wait for the completion of the application stream and use MPI-ACC's multiple internal streams for more efficient data transfer. We use the second approach because, in practice, multiple streams are more efficient for pipelining, although their management can become complex. Once the user initiates an MPI command, we simply create a corresponding request object in MPI-ACC, add it to the outstanding GPU request pool, and return. We do not query or wait for the stream object immediately. MPICH's progress engine periodically checks for unfinished MPI transactions and tries to complete them when possible. We leverage and extend the progress engine to periodically query for all the unfinished stream requests. If we find that a stream request has completed, it means that the data dependence semantics have been observed and that we are free to communicate the corresponding GPU data by using our internal streams. For every completed stream request, we follow MPI's communication protocol to send the rest of the data to the receiver, namely, send the Ready-To-Send (RTS) packet to receiver, wait for the Clear-To-Send (CTS) packet, and then transfer the actual GPU payload data to the receiver. On the other hand, if the programmer initiates an MPI call without a dependent stream parameter, it means that the data is immediately ready to be transferred. In this case, we do not add this request to the stream pool but directly send an RTS packet to the receiver to initiate data communication.

Summary: By using three implementations of a synthetic example program (Figures 4 and 5), we demonstrated that with the UVA-based design one can use only the explicit synchronization method. On the other hand, with the MPI attribute-based design, we can use either explicit or implicit synchronization, and the programmer can choose the preferred programming style. The attribute-based design can be considered somewhat like a superset of the UVA-based design. To use implicit GPU synchronization with attribute-based design, the programmer sets the stream parameter as an MPI attribute, while explicit synchronization can be used by simply not setting it. The explicit synchronization of the advanced UVA approach (Figure 4b) is more complex to code when compared with the simple UVA- and attribute-based approaches but is more likely to achieve the best performance. On the other hand, the attribute-based implicit GPU synchronization example is the most straightforward to code, but its performance depends on MPI's internal implementation, for example, the stream request pool management.

V. Evaluation and Analysis

In this section, we evaluate the performance of MPI-ACC's stream management implementation, which is key to enabling implicit synchronization semantics in MPI+GPU programs. To this end, we use code variants of the simple and advanced UVA-based approaches (Figure 4) and the MPI attribute–based approach (Figure 5), and we compare the performances of both implicit and explicit synchronization semantics by using MPI-ACC as the MPI implementation. All our experiments are run on two nodes of 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). 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. MPI-ACC, which is based on MPICH, is compiled with GCC v4.1.2 and on the Linux kernel v2.6.35.

A. Benchmark with Concurrent Multi-GPU Kernels

In the aforementioned MPI+GPU examples, separate streams are used to launch multiple kernels; hence, they could execute concurrently given sufficient GPU resources. If the program is run on a single GPU with a single hardware execution queue, the kernels are likely to execute and complete serially in issue order. In such cases, the UVA-based approach and attribute-based approach will perform identically to each other, because the advanced UVA-based implementation and MPI-ACC's stream management are specifically designed to take advantage of out-of-order stream completion to reduce the inter-stream wait time. On the other hand, if the kernels can run concurrently on multiple GPUs within a single node or within a single GPU itself (NVIDIA GPUs with compute capability 2.0 or higher) so that streams complete out of order, MPI-ACC (and the advanced UVA-based method) will service the streams *in-completion* order instead. However, current NVIDIA GPUs (pre-Kepler) have a single hardware queue for kernel execution that restricts the kernels to execute and finish in issue order.

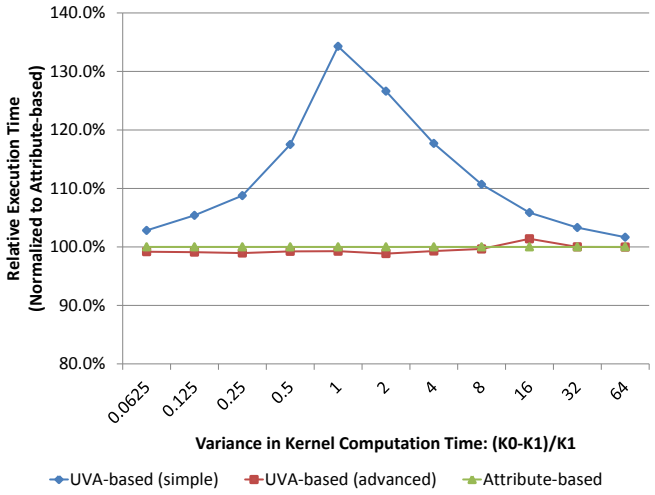
To showcase the efficacy of MPI-ACC's stream management for out-of-order stream completion, we run the above examples on multi-GPU nodes, where each GPU has its own hardware execution queue. In this experiment, we use three MPI processes: MPI rank 0 runs the GPU kernels on both GPUs and is also the sender process, while ranks 1 and 2 are just the receiver processes. Rank 0 is assigned exclusively to a single node, and the other two ranks are assigned to the other node in the cluster. Rank 0 issues kernels K0 and K1 concurrently on GPU 0 and GPU 1, respectively, by first launching K0 on GPU 0 followed by K1 on GPU 1. Upon kernel completion, rank 0 sends data from GPU 0 to rank 1 (MPI0) and from GPU 1 to rank 2 (MPI1) by using MPI-ACC as the GPU-integrated MPI, but the kernels need not complete in issue order. The ordering of the MPI calls is also irrelevant in this example.

1) Effect of kernel computation size: The simple UVA-based approach will always wait in issue order; that is, its execution order will always be K0–MPI0–K1–MPI1. If both K0 and K1 have the same execution time or K1 always takes longer to finish than K0, then the performance of the simple UVA-based example will be identical to MPI-ACC. On the other hand, if K1 finishes earlier, then the simple UVA-based example will unnecessarily delay MPI1 until K0 and MPI0 have finished. The stream management of MPI-ACC can detect K1's completion and issue MPI1 first. If the combined execution time for K1 and MPI1 is equal to the execution time of K0 alone, then the effect of K1 and MPI1 can completely be hidden by using MPI-ACC. Furthermore, if K0 is very much larger than K1 and MPI1 combined, we will see diminishing returns from overlapping the negligible K1 and MPI1 operations.

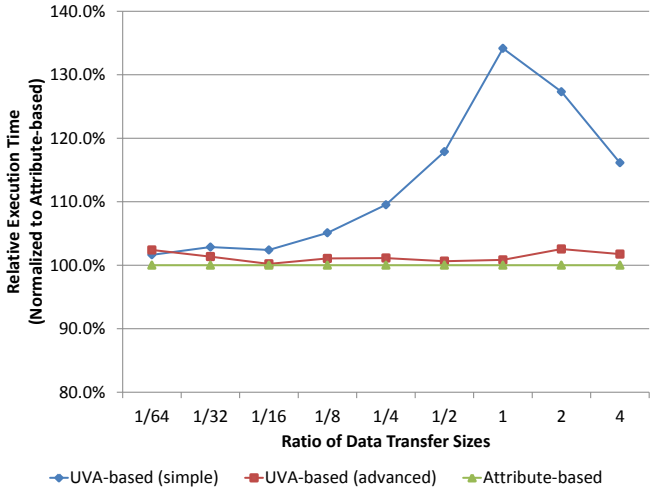
In our experiment, we artificially vary the compute time of K0 and keep K1 constant by simple loop manipulation. Also, we ensure that K1 finishes first and that data from GPU 1 will always be ready to be transferred before GPU 0. We begin with K0 being equal to K1, then gradually increment the loop iterations of K0. From Figure 7a, we see that MPI-ACC (and the advanced UVA-based example) can outperform the simple UVA-based approach by up to 34.2%, which we find to be the point of perfect overlap between K0 and $K1 + MPI1$.

2) Effect of data transfer size: The relative data transfer time of MPI1 with respect to MPI0 will also determine the benefits of MPI-ACC's in-completion order semantics versus the in-issue servicing of the simple UVA-based example. We choose the optimal loop iterations for K0 and K1 based on the previous experiment for maximum overlap, and we vary the data transfer size of MPI1 relative to MPI0. From Figure 7b, we see that MPI-ACC (and the advanced UVA-based example) can outperform the simple UVA-based approach by up to 34.1% at the optimal point.

Also, the performance difference between MPI-ACC and the advanced UVA-based example is negligible in both experiments. This means that the attribute-based design of MPI-ACC improves productivity with implicit semantics while achieving the same performance as the manual hand-optimized UVA-based example with explicit synchronization.



(a) Analyzing the effect of kernel computation size. Kernel0 is run on GPU 0, and its compute iterations and execution time are varied, while Kernel1 is run on GPU 1 and has a constant execution time.



(b) Analyzing the effect of data transfer size. Kernel0 and Kernel1 are fixed at the optimal ratio and the data transfer size from GPU 1 is varied.

Fig. 7: Performance comparison between the explicit and implicit GPU synchronization semantics. Both the UVA-based models use explicit synchronization semantics while the attribute-based model uses implicit synchronization.

B. Benchmark with OpenCL Command Queues

In OpenCL, synchronization semantics are defined implicitly via command queues or explicitly via user events. Command queues can be initialized as in-order, which denotes implicit ordering, or as out-of-order, which denotes that the GPU operations in the command queue can be treated as completely unordered. MPI-ACC can leverage the rich stream and event information that is passed by the application developer to make transparent optimization decisions, as needed. Consider a simple example, where an MPI process initiates an asynchronous H-D transfer to the GPU and then immediately calls MPI_Send on some independent buffer, but both being applied to the same command queue. MPI-ACC queries the command

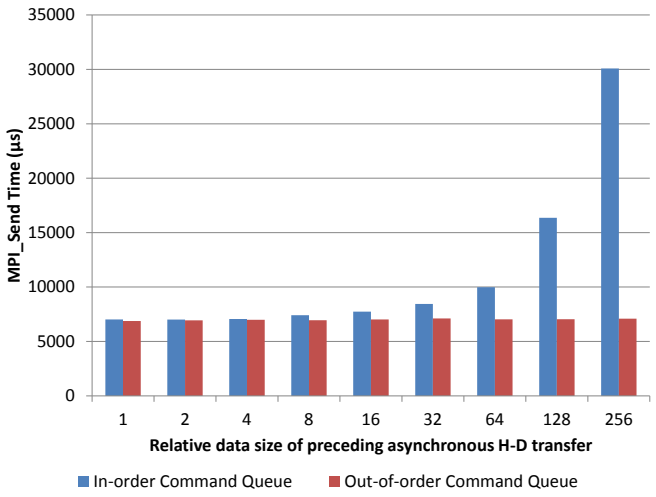


Fig. 8: MPI-ACC's attribute-based design leveraging OpenCL's out-of-order command queue semantics. The x-axis indicates the relative data sizes for an asynchronous H-D transfer. MPI-ACC waits for in-order queues but simultaneously performs H-D and MPI_Send for out-of-order queues.

queue information and synchronizes on the command queue only if it is not an out-of-order queue. The communication time of the MPI_Send operation can largely vary if MPI does not respect the queue semantics, as shown in Figure 8.

VI. Discussion

Understanding and effectively utilizing synchronization and ordering semantics of operations are fundamental to writing scalable applications, as we head to increasingly complex and parallel hardware. The ability to process computation and data as they arrive, by minimizing stalls imposed by dependencies and other runtime restrictions, is critical to achieving high performance.

While such goals are already observed in current GPU programming models such as CUDA and OpenCL, runtime driver implementations of some current hardware make it hard to easily realize them in practice. CUDA provides streams and OpenCL provides command queues that are essentially methodologies for specifying ordering or dependency requirements between different operations. However, several restrictions in the runtime environment make it hard to effectively utilize hardware even if no ordering or dependency requirements are provided.

a) CUDA Stream-0 Semantics: In general, operations issued on a single stream follow a strict dependency order, while operations issued on different streams can be processed and completed out of order. However, CUDA stream 0 is unique in that it is completely ordered with all operations issued on any stream of the device. That is, issuing operations on stream 0 would be functionally equivalent to synchronizing the entire device before and after each operation.

Figure 9(a) illustrates the order in which commands were issued to the different streams. Specifically, the following order was issued for issuing the operations: (a) C1 and C2 were issued on stream 1; (b) C3 and C4 were issued on stream 0; (c) C5 and C6 were issued on stream 1; (d) C7 and C8 were issued on stream 3.

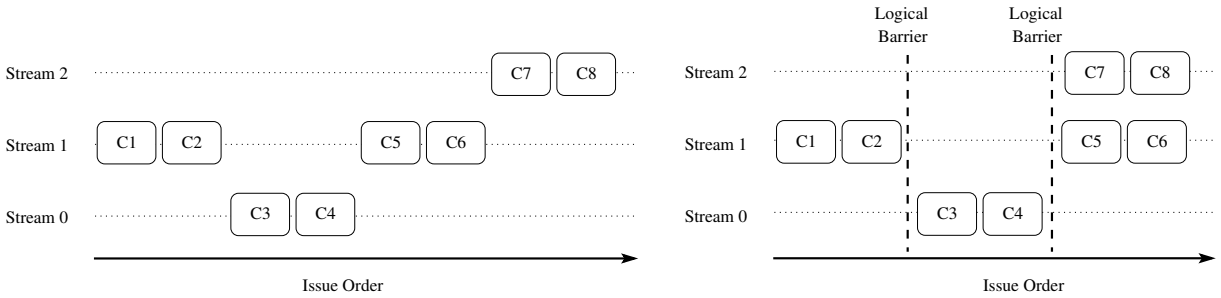
Ideally, the commands on different streams would be completely out of order and would execute whenever a particular resource on an accelerator was available to execute the given command. For instance, whenever a GPU was free to move data to or from the host memory, if a data movement operation was available to be executed on any of the streams, it would be executed. However, the strict ordering semantics imposed by stream 0 make this impossible; instead, a more conservative execution order is used in practice, as illustrated in Figure 9(b).

b) Hardware Queue Semantics and Ordering: While the user application is allowed to create a large number of streams or command queues, the number of hardware queues exposed by the GPU device, and hence the parallelism exposed by the GPU device, is limited. Current NVIDIA GPUs expose three hardware queues: one for computational kernels, one for device-to-host data transfers, and one for host-to-device data transfers. Commands issued to any of the user streams or command queues eventually end up in one of these hardware queues. The limited number of hardware queues is not too much of a concern by itself; however, current hardware enforces strict ordering semantics on these hardware queues that make this a significant problem.

Figure 10(a) illustrates the issue order of commands on the different user streams, and Figure 10(b) illustrates how they are queued up on the hardware queues. To avoid the issue of the above-mentioned CUDA stream-0 semantics, let us assume that all commands are issued on nonzero streams. Commands starting with "K" refer to computational kernels, those starting with "DH" refer to device-to-host data transfers, and those starting with "HD" refer to host-to-device data transfers.

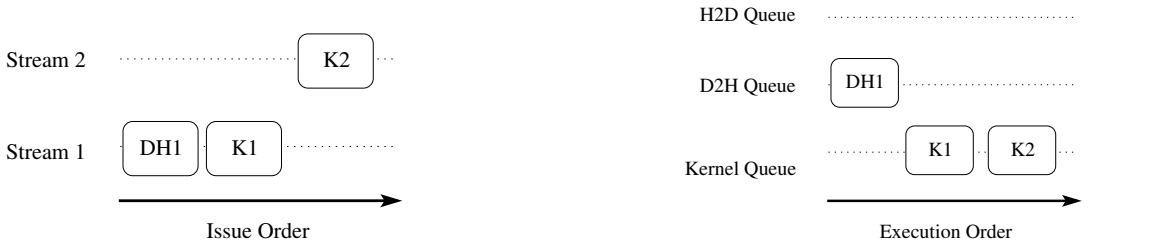
Note that $K1$ has a dependency on $DH1$, which requires it to stall for $DH1$ to finish. However, even though $K2$ is on a completely different stream and has a dependency on neither $K1$ nor $DH1$, it is forced to stall because the hardware queue for computational kernels is blocked by $K1$. This is a completely implementation-specific and unnecessary restriction that can be avoided either by allowing the hardware queues to better track dependencies between operations or by providing additional hardware queues that have no ordering restrictions between them.

c) Kernel Completion Signaling: One approach for the user or the MPI implementation to know when the dependency on a stream is satisfied is to request a completion event for previous operations issued on the stream. While a completion event guarantees that the operation is complete, every completed operation does not necessarily generate a unique event. Specifically, for performance reasons, some GPUs coalesce completion events for sequentially issued kernels. Thus, if two kernels, $K1$ and $K2$, are issued in sequence (whether on the same stream or different streams) and if an event is requested after each kernel, these two events will be coalesced into a single signaling event that is delivered after both kernels have completed. This is not a significant concern when the two



(a) Issue order of commands. (b) Execution order of commands.

Fig. 9: CUDA stream-0 semantics.



(a) Issue order of commands. (b) Execution order of commands.

Fig. 10: Hardware queue semantics.

kernels are on the same stream. If the two kernels were on different stream, however, and if the user issued a data transfer operation on the same stream as $K1$, it would have to wait until $K2$ has finished as well since it would not be signaled of $K1$'s completion independently. This issue is illustrated in Figure 11.

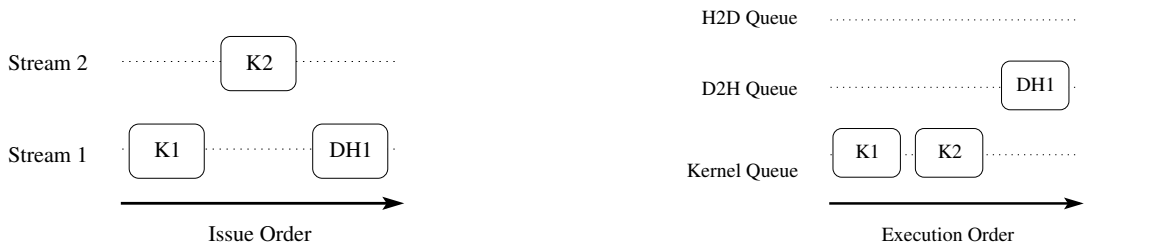
d) Implications to GPU-integrated MPI implementations: All of the above hardware and software nuances are especially important when designing GPU-integrate MPI implementations. For example, if the user issues several kernels in sequence on different streams, any data transfer operation issued by MPI on *any* of the streams has to wait for *all* kernels to finish because of the kernel completion signaling semantics. This is of particular concern when the data transfer operation is dependent on the completion of any one of issued kernels because, in such a case, the data transfer engine would be idle for the execution of all the kernels. If the user issues any GPU operation to the (default) stream 0 in CUDA, any MPI operation on that GPU will be stalled until the stream 0 has completed all its operations. NVIDIA's next generation Kepler architecture comes with multiple hardware queues where kernels and data transfer operations can truly finish out of issue order, which could largely alleviate the hardware queue ordering problem. Since our scalable stream management design within MPI-ACC can substantially gain from out of order completion of GPU operations, we expect our attribute-based design and implicit synchronization semantics to achieve programmer productivity and performance for future architecture designs as well.

VII. Related Work

MVAPICH [7] 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 [16] (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 [12]. On the other hand, MPI-ACC takes a more portable approach: we support data transfers among CUDA [12], OpenCL [13], 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 [17] 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.

GPUs have been used to accelerate many HPC applications across a range of fields in recent years [1], [2], [18], [19]. For large-scale applications that go beyond the capability of one node, manually mixing GPU data movement with



(a) Issue order of commands.

(b) Execution order of commands.

Fig. 11: Kernel completion signaling.

MPI communication routines is still the status quo, and its optimization usually requires expertise [20], [21]. In this work, our experience with MPI-ACC [8], 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.

VIII. Conclusions

In this paper, we investigated the synchronization and ordering semantics that are used by different GPU-integrated MPI frameworks and are based on the UVA-based programming approach and the MPI attributes–based approach. We demonstrated that the UVA-based approach does not allow users to provide enough semantic information about GPU stream ordering and completion semantics. The MPI attributes–based approach, on the other hand, allows users to provide richer information to the MPI implementation that it can take advantage of for performance and correctness without forcing the user to manage these synchronization semantics. Through benchmark experiments using multiple GPU streams and MPI communication operations we showed that while the UVA-based design is useful for isolated point-to-point communication with no data dependencies or ordering requirements, the attribute-based design might be more appropriate when multiple interdependent MPI and GPU operations are interleaved.

Acknowledgment

This work was partially supported by the U.S. Department of Energy under grant DE-SC0001770 and contracts DE-AC02-06CH11357, DE-AC05-00OR22725, and DE-ACO6-76RL01830, and an NVIDIA Graduate Fellowship.

References

[1] L. Weiguo, B. Schmidt, G. Voss, and W. Muller-Wittig, "Streaming Algorithms for Biological Sequence Alignment on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 9, pp. 1270–1281, Sept. 2007.

[2] A. Nere, A. Hashmi, and M. Lipasti, "Profiling Heterogeneous Multi-GPU Systems to Accelerate Cortically Inspired Learning Algorithms," in *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, May 2011, pp. 906–920.

[3] W.-C. Feng, Y. Cao, D. Patnaik, and N. Ramakrishnan, "Temporal Data Mining for Neuroscience," in *GPU Computing Gems*, Feb. 2011, emerald Edition.

[4] "TOP500," http://www.top500.org/lists/2012/11/highlights.

[5] "NVIDIA GPUDirect," http://developer.nvidia.com/gpudirect.

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

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

[8] A. M. Aji, J. S. Dinan, D. T. Buntinas, P. Balaji, W. chun Feng, K. R. Bisset, and R. S. Thakur, "MPI-ACC: An Integrated and Extensible Approach to Data Movement in Accelerator-Based Systems," in *IEEE International Conference on High Performance Computing and Communications (HPCC)*, Liverpool, UK, June 2012.

[9] F. Ji, A. Aji, J. Dinan, D. Buntinas, P. Balaji, W.-C. 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.

[10] F. Ji, A. M. Aji, J. S. Dinan, D. T. Buntinas, P. Balaji, R. S. Thakur, W. chun Feng, and X. Ma, "DMA-Assisted, Intranode Communication in GPU Accelerated Systems," in *IEEE International Conference on High Performance Computing and Communications (HPCC)*, Liverpool, UK, June 2012.

[11] J. Jenkins, J. S. Dinan, P. Balaji, N. F. Samatova, and R. S. Thakur, "Enabling Fast, Noncontiguous GPU Data Movement in Hybrid MPI+GPU Environments," in *IEEE International Conference on Cluster Computing (Cluster)*, Beijing, China, Sep. 2012.

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

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

[14] Nvidia, "NVIDIA CUDA C Programming Guide version 4.0," May 2011.

[15] Steve Rennich, NVIDIA, "CUDA C/C++ Streams and Concurrency," http://developer.download.nvidia.com/CUDA/training/ StreamsAndConcurrencyWebinar.pdf.

[16] 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.

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

[18] J. C. Phillips, J. E. Stone, and K. Schulten, "Adapting a Message-Driven Parallel Application to GPU-Accelerated Clusters," in *International Conference on High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008*, Nov. 2008, pp. 1–9.

[19] L. Ligowski and W. Rudnicki, "An Efficient Implementation of Smith Waterman Algorithm on GPU Using CUDA, for Massively Parallel Scanning of Sequence Databases," in *IEEE International Symposium on Parallel Distributed Processing, 2009. IPDPS 2009*, May 2009, pp. 1–8.

[20] T. Hamada, T. Narumi, R. Yokota, K. Yasuoka, K. Nitadori, and M. Taiji, "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*, ser. SC '09. New York: ACM, 2009, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/1654059.1654123

[21] W. M. Brown, P. Wang, S. J. Plimpton, and A. N. Tharrington, "Implementing Molecular Dynamics on Hybrid High Performance Computers – Short Range Forces," *Computer Physics Communications*, vol. 182, no. 4, pp. 898–911, 2011.