2010 16th International Conference on Parallel and Distributed Systems

GMH: A Message Passing Toolkit for GPU Clusters

Jie Chen and William Watson III
The Scientific Computing Group
Jefferson Lab
Newport News, Virginia 23606, USA
Email: {chen,watson}@jlab.org

Weizhen Mao
Department of Computer Science
College of William and Mary
Williamsburg, Virginia 23187, USA
Email: wm@cs.wm.edu

***Abstract*—Driven by the market demand for high-definition 3D graphics, commodity graphics processing units (GPUs) have evolved into highly parallel, multi-threaded, many-core processors, which are ideal for data parallel computing. Many applications have been ported to run on a single GPU with tremendous speedups using general C-style programming languages such as CUDA. However, large applications require multiple GPUs and demand explicit message passing. This paper presents a message passing toolkit, called GMH (GPU Message Handler), on NVIDIA GPUs. This toolkit utilizes a data-parallel thread group as a way to map multiple GPUs on a single host to an MPI rank, and introduces a notion of virtual GPUs as a way to bind a thread to a GPU automatically. This toolkit provides high performance MPI style point-to-point and collective communication, but more importantly, facilitates event-driven APIs to allow an application to be managed and executed by the toolkit at runtime.**

***Keywords*-GPU; Cluster; Message Passing; MPI; CUDA**

I. Introduction

In the past few decades, the performance of CPUs has steadily increased according to Moore's law. However, the performance of scientific applications has not enjoyed the same increase despite higher clock rates, larger memory caches, and instruction-level parallelism of these CPUs. This brings the recent multi-core processors [16], which offer better performance for data parallel applications by executing multiple threads simultaneously. However, the relative small number of processing cores and limited memory bandwidth on this type of CPU prohibit further increases in the performance for data parallel applications. On the other hand, a modern graphics processing unit (GPU) contains a scalable array of multi-threaded Streaming Multiprocessors (SMs) and offers extremely high memory bandwidth. Each SM consists of many Scalar Processor (SP) cores with on-chip shared memory and can execute hundreds of threads in parallel efficiently in the Single Instruction Multiple Thread (SIMT) fashion. Consequently, General-Purpose computation on GPUs (GPGPU) [13] has taken off. Excellent GPU performance speedups have become common place in fields from molecular dynamics [2] to lattice quantum chromodynamics (LQCD) [3]. For many applications, GPUs have been excellent platforms providing much better performance-to-cost ratio relative to their latest mutil-core CPU counterparts [4].

Recently, AMD and NVIDIA have begun producing GPUs not only tailored for the gaming community but also suitable for high performance computing applications. Meanwhile, GPU software development tools have evolved rapidly as well. Several general purpose high level GPU programming toolkits such as CUDA [9], Brook+ [1], and most recently OpenCL [10] have replaced traditional computer graphics development libraries, such as OpenGL [11], whose Application Program Interfaces(APIs) are not suitable for GPGPU. These new toolkits provide more natural GPGPU programming environments, and expose more hardware capabilities. This paper focuses on CUDA, which is stable and widely used by the scientific computing community.

CUDA is a general purpose programming system for NVIDIA GPUs and was first released in the end of 2007. It extends the C programming language to support executing a GPU function (*kernel*) on a single GPU in the SIMT fashion. A GPU kernel is very much like a regular CPU function with minor notational change, and is executed by all SMs using tens of thousands of threads at the same time to achieve high performance. CUDA allows asynchronous kernel execution and offers the capability of concurrent operations of kernels and host-GPU memory DMA transactions on different "streams". Furthermore, CUDA also provides event recording APIs that enable applications to check the progress of individual operations within an asynchronously executing stream. Note that CPUs (hosts) are involved in starting GPU kernels, managing GPU memory, and handling communication among devices on the PCI buses.

Many scientific applications have achieved remarkable speedups running on a single GPU using CUDA in comparison to running on a multi-core CPU. For example, a lattice QCD linear equation solver [3] using mixed precision on NVIDIA GTX 280 has achieved more than a 10 times performance increase relative to executing the same algorithm on a host with dual quad-core Intel Nehalem processors. However, large scientific applications may demand multiple GPUs either on the same host or within a single GPU cluster [5] because of the large memory requirement of this type of application.

1521-9097/10 $26.00 © 2010 IEEE
DOI 10.1109/ICPADS.2010.35

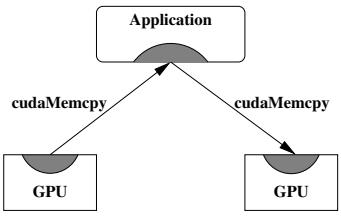

Figure 1. Exchange a single message between two GPUs

Using multiple GPUs for multiple independent processes or threads is trivial. However, exchanging messages among multiple GPUs for large applications is cumbersome. Fig. 1 describes what is involved in an exchange of a single message between two GPUs on a single host in the CUDA programming environment using a single host buffer in a multi-threaded application.

Since a GPU kernel can not manage GPU memory directly, a datum in a GPU memory location has to be first transferred out from the GPU to the host by the *cudaMemcpy* host function followed by another *cudaMemcpy* function call to send the datum from the host to the other GPU. Multi-threaded applications have to synchronize memory access to the host memory buffers that are used for the transfers. In contrast, the same type of transfers will be more complicated for applications using multiple processes on the same host or on different hosts because of the required message passing from one process to another. Furthermore, to port single GPU applications to run on multiple GPUs or to develop applications for GPU clusters, developers usually face several obstacles: splitting kernels to handle calculations for interior volume and surface area respectively, overlapping computation and communication, and so on.

To reduce the complexity of programming for multiple GPUs, this paper presents a message passing toolkit for GPU clusters, called the GPU message handler (GMH), which not only provides high performance point-to-point and collective GPU communication but also facilitates a new framework to allow applications to be managed and executed by the toolkit at runtime.

This paper is organized as follows. Section 2 reviews previous related work. Section 3 describes the design and implementation of the GMH toolkit. Section 4 overviews the software and hardware environment where performance evaluations are carried out. Section 5 presents the performance data for the toolkit. Section 6 concludes.

II. Related Work

Similar to parallel applications using multiple CPUs on clusters, there are two major paradigms for programming and implementing GPU parallel applications using GPU clusters: shared memory paradigm that share data between processes through shared memory and message passing paradigm that exchange messages between processes or threads running concurrently.

On CPU clusters, shared memory parallel programs are widely considered easier to develop than message passing programs. There has been research that extends shared memory paradigm to GPU clusters. For example, a software based distributed shared memory can be implemented by modifying each memory access to use a GPU page table, but the performance of this type of implementation shows a drastic performance slowdown [8]. Therefore, distributed memory with explicit message passing is the only viable option known to scale well for GPU clusters.

The Message Passing Interface (MPI) [15] is the de facto standard for developing parallel programs for CPU clusters. Recently, there have been several research efforts to provide MPI style message passing for parallel GPU applications. For example, cudaMPI [6] replicates most MPI functions. It offers message passing from CPU to GPU or from GPU to GPU, and delivers reasonably good performance. However, it only supports the configuration of one GPU on one computing host, it does not provide any new framework that simplifies development efforts of parallel GPU applications, and it does not isolate its message passing from MPI implementations, which may result in conflicts in sending and receiving messages when applications mix cudaMPI and MPI together. Nonetheless, cudaMPI takes a CPU centric approach in the sense that all message exchanges are dictated by CPUs. On the other hand, DCGN [17] adopts a GPU centric approach in the way that GPUs initiate message exchanges while CPUs serve as mere I/O processors to poll for GPU communication. Unfortunately, it requires hundreds of microseconds extra overhead for each GPU message due to CPU polling for messages. In addition, DCGN implementation does not support any application using just one host with multiple GPUs and does not allow mixing DCGN and MPI together. If 100% of any application could run on GPUs, the GPU centric message passing would be clearly preferable over the CPU centric message passing. Until that happens, the CPU centric message passing remains an ideal choice for most GPU parallel applications.

The GMH toolkit developed at Jefferson Lab, a national laboratory under the Office of Science in the Department of Energy, follows the CPU centric message passing approach. It utilizes MPI to deliver messages among hosts where GPUs are situated. It uses a thread group on each individual computing host as a way to map GPUs to an MPI rank. Each thread in the thread group is bonded to a particular GPU automatically. The message exchanges among GPUs on a single host are carried out without host memory copies. Furthermore, GMH is implemented as an MPI library, therefore GMH applications can mix GMH and MPI together. More importantly, GMH not only provides MPI style communication primitives but also offers an event-driven flow control framework that allows applications to be

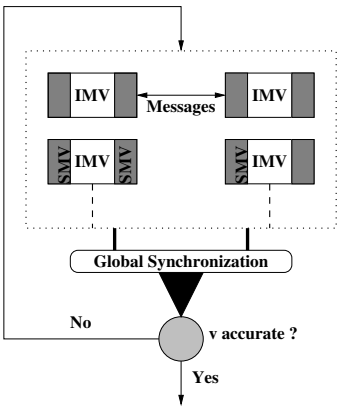

Figure 2. Event flow model of a parallel iterative solver

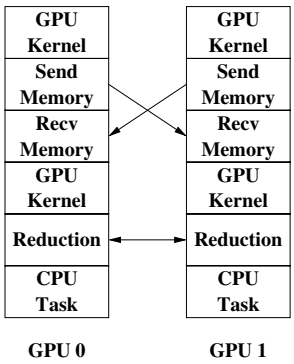

Figure 3. GMH run time architecture

executed by GMH at run time.

III. The GMH Toolkit

Solving linear systems is a common occurrence in many scientific applications. For example, iterative linear system solvers, such as the BiCGSTAB [14] method, have been utilized frequently in theoretical physics LQCD calculations [3]. A successful convergence of a parallel BiCGSTAB execution requires many iterations, during which many steps of parallel matrix-vector multiplications are carried out followed by a global synchronization. Thus a parallel iterative solver can be encapsulated into an *event flow* model as illustrated in Fig. 2, where IMV stands for interior volume matrix vector multiplications, and SMV stands for surface area matrix vector multiplications. In the figure, message exchanges among GPUs are initiated at the same time when IMV kernels start execution. The SMV kernels start running once surface messages are exchanged. The above steps repeat many times until a global synchronization is performed. Then, the accuracy of the solution vector is checked to determine whether the iteration process continues.

GMH provides a set of C programming APIs to enable an *event flow* message passing programming development in addition to the conventional MPI style message passing primitives. Applications can either use the MPI style APIs in the GMH library to control iterative processes or register commands and events with GMH, whose command queues and event handlers manage these iterative processes for the applications. The later approach can significantly reduce the effort of porting GPU applications running on a single GPU to the applications running on multiple GPUs because developers could concentrate on algorithms and event flow instead of programming details.

A. GMH Architecture

The GMH toolkit is a multi-threaded event driven message passing C programming library. It utilizes a dedicated CPU core to handle each GPU to ensure good performance. It uses MPI as its underlying inter-host communication mechanism and eliminates host memory copies for intra-host GPU communication. It depends on MPI implementations with fully threaded support to avoid unnecessary polling of underlying communication buffers, which adds overhead to each message. Furthermore, GMH automatically assigns and binds a CPU thread to a GPU due to a one-to-one mapping of CPU threads to GPUs, which is a mandate from CUDA.

A GMH thread controlling a GPU owns a command queue, into which requests/commands such as GPU kernels or communication calls are funneled. The GMH toolkit executes developer specified requests in the command queues either synchronously or asynchronously at runtime. The command queue can hold multiple streams so that concurrent memory transfers and kernel executions can take place. A command is able to generate an event so that any command can have an event dependency specified such that the command can not be executed until some event has finished. Similarly, GMH can explicitly wait on one or a set of events before executing the next available command. Fig. 3 illustrates one example of GMH command queues on two threads managing two GPUs at run time.

B. GMH Implementation

The implementation of GMH faces several challenges. First of all, each GPU has to be presented as a communication end point to hide the difference between inter-host and intra-host GPU communication. Secondly, the GPUs on a host must be managed automatically to avoid static mapping between GMH threads and GPU devices. These two challenges along with the goals of making GMH a *high performance*, *flexible* and *easy to use* toolkit shape the final implementation.

1) GMH Environment: The GMH environment can be initialized by calling *gmh_init* with the number of requested GPUs on a single host. The initialization routine spawns the number of threads equal to the number of requested GPUs on the host. Each thread is bound to a GPU and executes a user supplied thread function once the GMH environment is set up. To better manage multiple threads and GPUs on a single

host, GMH employs a concept of *virtual GPU*: a virtual GPU device number that always starts from 0 is assigned to a thread once the thread is bound to a GPU. The adoption of the concept of virtual GPUs enables GMH to launch either one process or multiple processes on a single host without causing GPU resource conflicts among GMH threads. Since there is a one-to-one mapping between a GMH thread and a GPU, this paper treats these two terms the same from here on. In addition, a communication end point for a GMH thread is a unique integer that combines the virtual GPU id of the thread with the MPI rank of the process that owns the thread attached to the GPU. The combined integers are called GMH ranks.

2) GMH Memory: A GPU memory location is encapsulated into a GMH memory structure called *gmh_mem_t*, which holds information about the GPU memory location, its size and permission. The GMH memory is created with a default option that a page locked (pinned) host memory of the same size is also allocated.

3) GMH Commands and Events: Each GMH thread attached to a GPU contains a command queue into which an application inserts any supported commands through the GMH APIs. A command queue *qmh_tasklist_t* is FIFO in nature, but it may contain multiple streams. A command has to be associated with a stream upon creation. Commands on different streams can be executed concurrently to offer the capability of overlapping communication and computation. Even though the GMH commands are the most important building blocks of the GMH toolkit, developers never interact with commands directly but rather through APIs.

When applications issue a *gmh_init* function call, the created command queues are delivered back to the applications as the first argument to the user provided thread functions. The following is the definition for the user provided thread function:

void (*gmh_thread_func_t) (gmh_task_list_t , void *);*

Each GMH command can be either synchronous or asynchronous. An asynchronous command generates a GMH event which can be used to track the progress of the command. Each GMH event is comprised of one CUDA event, one MPI request, or both. Applications can wait on a single event or wait on a set of events. Thus, events can be used to build up a dependency structure to enable a command to wait for a set of events to finish before the command can start to execute. The following code segment illustrates the concepts of streams, commands and events.

gmh_tasklist_t l; *gmh_stream_t s*;
gmh_event_t ev, ∗evs_wait;
gmh_kernel_t func;
int nevs; *void *arg*;

gmh_create_command_stream (l, &s, 0);
gmh_add_gpu_kernel (l, s, func, arg, nevs, evs_wait, &ev);

In the above code segment, a command stream *s* is created for the command queue *l*. A new command, which is a GPU kernel specified by a user function *func* along with a user argument *arg*, starts only after *nevs* number of events stored in *evs_wait* have finished and then generates a new event denoted by *ev*.

4) GMH Point-to-point Communication: GMH provides two flavors of communication APIs for applications. One set of APIs is very much MPI alike with communication end points defined by GMH ranks which are used to either identify MPI processes across a cluster or to distinguish GMH threads within a single host. GMH mandates the underlying MPI implementation to be fully threaded since each necessary MPI call is invoked by individual GMH thread attached to a particular GPU. The following code segment describes the conventional asynchronous communication APIs.

*void *sbuf, *rbuf*; *gmh_datatype_t type*;
gmh_tasklist_t l; *gmh_stream_t s*;
int dest, src, tag, count;
gmh_event_t sev, rev;

gmh_Isend(l, s, sbuf, count,type, dest,tag, &sev);
gmh_Irecv(l, s, rbuf, count, type, src,tag, &rev);

The other set of APIs takes the *event flow* approach. Applications register *send* or *receive* commands to GMH which executes these commands at run time. The following code segment describes the idea, where *block* specifies whether a routine is a blocked call.

gmh_tasklist_t l; *gmh_datatype_t type*;
gmh_mem_t sbuf, rbuf;
int dest, src, block, nevs;
gmh_event_t sev, rev, ev, ∗evs_wait;

gmh_add_send_buffer(l, s, sbuf, block, dest, tag, nevs, evs_wait, &sev);
gmh_add_recv_buffer(l, s, rbuf, block, src, tag, nevs, evs_wait, &rev);

5) GMH Collective Communication: Collective communication involves global data movement and global control among all GPUs in a cluster. Unlike point-to-point communication routines, collective communication is synchronous, which means that all events generated before a collective call have to be finished before the call starts. Once again, GMH provides one set of APIs that are very similar to MPIs and offers another set of APIs allowing applications to register collective communication commands to GMH. More importantly, GMH leverages the power of the underlying MPI collective functions to achieve high performance for collective communication. The following code snippet

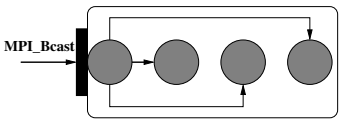

Figure 4. Mapping multiple GPUs to an MPI rank

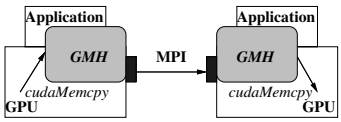

Figure 5. GMH applications send GPU data across network

illustrates functionalities of collective communication.

gmh_tasklist_t l; *gmh_mem_t sbuf, rbuf*;
int root, type, op;

gmh_bcast(l, sbuf, root);
gmh_reduce(l, sbuf, rbuf, type, op, root);

In comparison to point-to-point communication, GMH maps multiple GPUs to a single MPI rank during collective communication. For example, a GMH thread with a virtual GPU id of 0 on a host forwards the messages delivered by *MPI_Bcast* to other GPUs on the same host upon the execution of *gmh_bcast*. Fig. 4 illustrates the concept of mapping multiple GPUs to a single MPI rank.

6) GMH Event Run Loop: A GMH thread executes all registered commands in an event loop by issuing *gmh_start(list)*. The event loop executes each command inside the command queue one after another in the FIFO order. Commands can run concurrently if they are on different streams. Synchronizations can be achieved through three different ways: explicit invocations of *gmh_wait_for_events*; executing commands with defined event dependency structures; any collective communication call. The end of the loop is determined by a control CPU task, which is a C function returning either *GMH_EXIT* or *GMH_RERUN*. The control task is registered through the following function and is a synchronous command.

gmh_tasklist_t l;
gmh_control_task_t task;
void arg*;

gmh_add_control_task (l, task, void);

IV. Hardware and Software Environment

Our test environment contains 16 hosts connected by quad data rate(QDR) infiniband networks, which provide up to 40 Gbits/sec network bandwidth. Each host is equipped with two Intel Nehalem E5530 quad-core CPUs running at 2.4 GHz. Each host has 24 GB ECC DDR3 memory clocked at 1331 MHz. In addition, each host has two NVIDIA GTX285 GPUs, each with 2 GB GDDR3 memory and 240 processing cores running at 1.51 GHz. Each GPU provides internal memory bandwidth of 121 GB/sec and supports PCI Express 2.0x16 host interface that provides bi-directional CPU-GPU memory bandwidth up to 6.4 GB/sec.

Each single computing host is running CentOS 5.3 with Linux Kernel 2.6.18. The MPI implementation is mvapich2-1.2 from Ohio University [7] with OFED [12] version 1.4.2. All performance test programs are compiled with gcc version 4.1.2 using optimization flag "-O3". The GPU driver version is 190.29 and CUDA toolkit is version 2.3.

V. Performance Results of GMH

To fully comprehend GMH, detailed performance tests are conducted. These tests exercise point-to-point throughput and latency (half round-trip time) involving two GPUs on a single host or two GPUs on two hosts, In addition, these tests analyze how well the collective communication performs involving all GPUs in the test cluster.

A. Memory Transfers between GPUs and CPUs

The GMH toolkit, like any other GPU message passing library, utilizes the synchronous function of *cudaMemcpy* or the asynchronous function of *cudaMemcpyAsync* to transfer data from GPUs to CPUs before sending the data to other GPUs and to move data from CPUs to GPUs after receiving the data from other GPUs. Since GMH focuses on the capability of overlapping communication and computation, it uses *cudaMemcpyAsync* for all point-to-point communication primitives and uses *cudaMemcpy* for all collective communication routines. Fig. 5 illustrates how *cudaMemcpy* is used in a single message exchange between two GPUs on two hosts.

To understand the performance characteristics of point-to-point communication, the memory transfer bandwidth and latency values are collected. Especially, the bandwidth values from CPUs to GPUs are compared with the bandwidth values from GPUs to CPUs. Unlike conventional network devices which have the same bandwidth values regardless of directions of data transfers, a GTX 285 GPU on the test platform can have different transfer bandwidths depending on the direction of data transfers. Fig. 6 shows the results of latency and bandwidth for memory transfers between GPUs and CPUs along with the performance results from mvapich2 for comparison purposes.

In Fig. 6, a *cudaMemcpy* takes about 12 μs to transfer a small message from a GPU to a CPU and vice versa. In contrast, a *cudaMemcpyAsync* along with a proper synchronization mechanism such as *cudaEventSynchronize* takes about 32 μs to transfer a small message. More importantly, the transfer bandwidth from a GTX 285 to a CPU saturates at about 1800 MB/sec in comparison to a rather large transfer bandwidth of 4700 MB/sec from a CPU to a GTX 285. For comparison purposes, the network bandwidth from

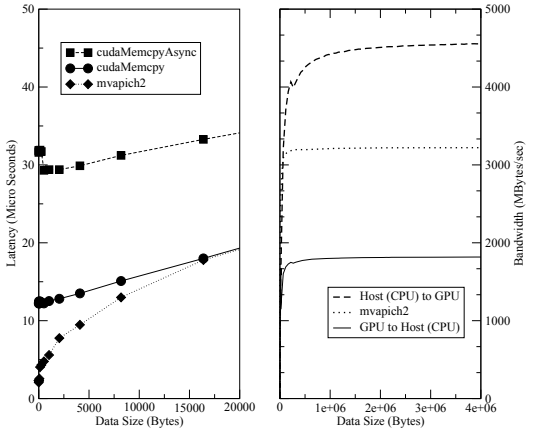

Figure 6. The performance values of memory transfers between CPUs and GPUs

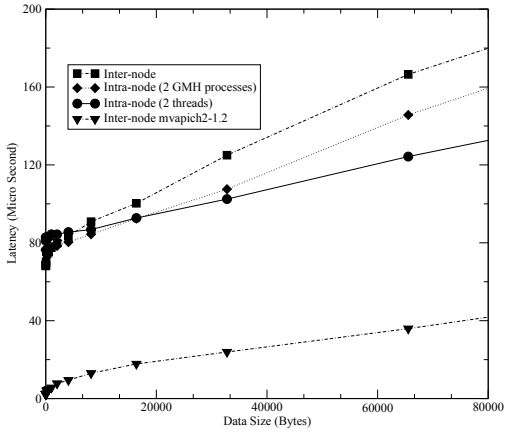

Figure 7. GMH application-to-application latency

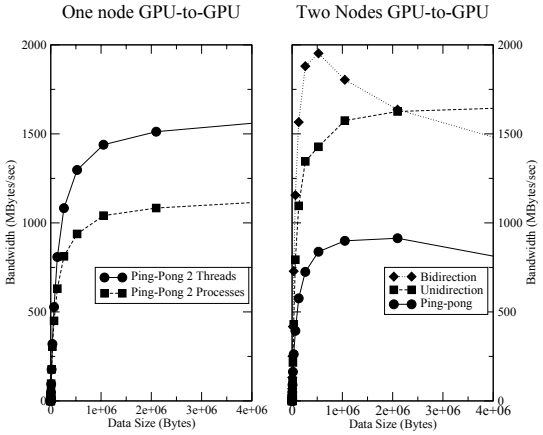

Figure 8. GMH point-to-point bandwidth

mvapich2 is approaching 3200 MB/sec, and the latency values are very low at $3\mu s$ for small messages. The low memory transfer bandwidth from a GTX 285 GPU to a CPU thus dictates the overall communication bandwidth of GMH because of relatively large network bandwidth from mvapich2 under QDR infiniband networks.

B. Point-to-Point Communication

Point-to-point communication with GMH was measured using micro-benchmarks of sends, receives, and ping-pong tests. Many iterations of each type of test with varying sizes are performed to accumulate the performance results. Special attention is focused on the difference between intra-host GPU communication and inter-host GPU communication.

1) Small Packet Latency: The latency benchmark measures how long it takes a datum to travel from one GPU to another GPU. The performance data are obtained by taking half the average round-trip time for various data sizes. Fig. 7 presents the results of latency for GMH when data are transferred between two GPUs on a single host either using two GMH threads or using two GMH processes, and between two GPUs on two hosts. Fig. 7 also displays the latency results of mvapich2 between two hosts for comparison purposes.

In Fig. 7, the latency values for small messages between two GPUs on two hosts is 68 μs. This is very close to $(2 \times 32 + 3)$ μs, where $32\mu s$ is the average latency value for a memory transfer between a CPU and a GPU and $3\mu s$ is the latency value of a small message for mvapich2 between two hosts. Clearly, there is no incurred overhead in latency for small messages, which affirms that the implementation of GMH is highly efficient. However, the latency values ($\approx 80\mu s$) for small messages between two GPUs on a single host using two threads are actually larger than the latency values between two processes. This is caused by the fine grain synchronization overhead between two threads when transferring data between them. Nonetheless, the extra cost of copying data between processes using MPI eventually overtakes the cost of thread synchronization when data transfer size increases. This is shown in Fig. 7, where the latency values for the intra-host threaded data transfer become smaller than the latency values for the other two cases. This underscores the value of using threads as intra-host GPU data transfer mechanisms for the GMH implementation.

2) Bandwidth: The experiments that measure three types of bandwidth are carried out. These three different types of bandwidth capture different communication patterns occurring in typical user applications. In the *bidirectional ping-pong bandwidth*, data flow back and forth, in a ping-pong fashion. In the *unidirectional bandwidth*, data flow in one direction only, which reveals the bandwidth capability of underlying devices. Finally, the *bidirectional simultaneous bandwidth* simulates data transfers in both directions simultaneously. Fig. 8 shows the results of these types of bandwidth for GMH.

The left part of Fig. 8 shows the values of the ping-

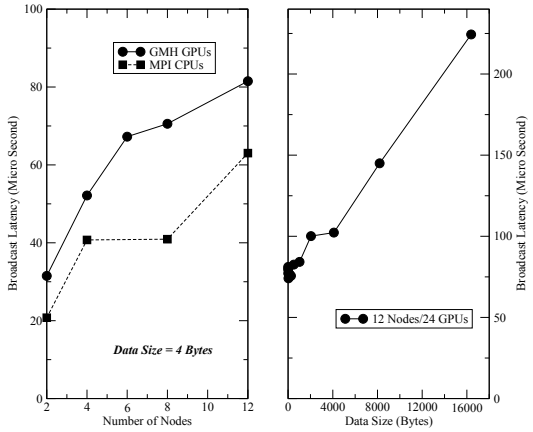

Figure 9. GMH broadcast latency values

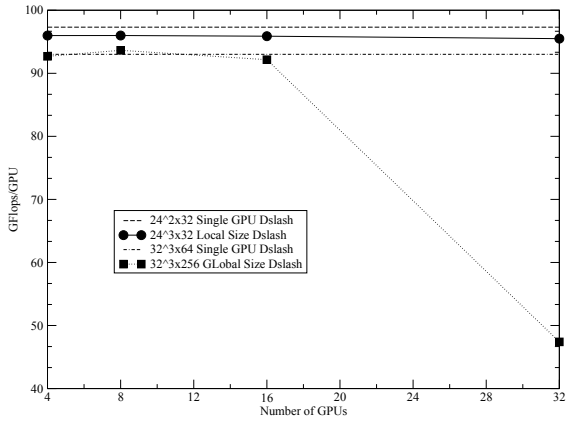

Figure 10. The performance values of *dslash* using GMH

pong data transfer bandwidth between two GPUs on a single host. Clearly, the GMH bandwidth values using two threads are much better than the GMH bandwidth values using two GMH processes, due to the extra host memory copies when processes are used to transfer data. The right part of Fig. 8 shows the bandwidth values of three different types of data transfer between two GPUs on two hosts. It is as expected that the values of the bidirectional simultaneous bandwidth are roughly twice those of ping-pong bandwidth. Recall from Section 5.1 that the memory transfer speed from GPUs to CPUs is 1800 MB/s. The results of the unidirectional bandwidth are clearly approaching the above number, demonstrating the high performance nature of the GMH toolkit.

C. Collective Communication

Most collective communication primitives in the MPI specification are implemented in the GMH toolkit. There are two steps involved in executing a GMH collective communication function: 1) GMH utilizes the underlying MPI implementation to attain the inter-node collective communication on host memories with mapping of multiple GPUs on a host to a single MPI rank; 2) GMH employs each thread attached to a GPU to copy data from or to the MPI buffer populated or used by the corresponding MPI function to or from every GPU on the host. Thus, every GMH collective communication function inevitably incurs an overhead of memory copies from CPUs to GPUs and vice versa. Fig. 9 presents the latency values for GMH broadcast.

Recall that *cudaMemcpy* is used in all GMH collective communication routines, which are all synchronous calls, and each *cudaMemcpy* takes about $12\mu s$. The left part of Fig. 9 compares the broadcast latency values for message size of 4 bytes between GMH and mvapich2. Clearly, it takes about 20 μs extra time for GMH to do a broadcast than what mvapich2 takes. The extra time is inline with the duration for one CPU-GPU and one GPU-CPU memory transfers even though there are many CPU-GPU and GPU-CPU memory copies involved on every host participating the broadcast. This is because most of the memory copies are hidden within the MPI binomial broadcast communication [19] except the first GPU-CPU copy on the broadcast root node and the CPU-GPU copy on the last node. Once again, the fact that there is no additional unaccountable overhead reaffirms that the GMH implementation is close to optimal. The right part of Fig. 9 illustrates the linear increase in broadcast latency value as the data size increases. This comes at no surprise because of the linear increase in memory copy latency and linear increase in MPI latency against data size.

D. LQCD Benchmark

The LQCD benchmark *dslash* [18] is one of the most computing intensive parts within typical LQCD iterative solvers. At each iterative step, it sums up two matrix-vector products along each of x, y, z, t direction for each and every 4-dimensional discretized site. On each site and for every single step, 3×3 complex matrices and complex vectors from all the neighboring sites have to be retrieved before the calculations. The single GPU version of this benchmark achieves about 98 Gflops on a single GTX285 GPU.

To parallelize the benchmark, the original *dslash* GPU kernel has to be split into two kernels: one carries out the calculations for the interior volume the same way as the single GPU version; the other kernel has to wait for the matrices and vectors sent from neighbors before carrying out the calculations. The GMH toolkit simplifies the effort of parallelizing the benchmark by reducing the effort of handling message passing, overlapping computation and communication, and so on. Fig. 10 presents normalized *dslash* performance results for the fixed global lattice size (*strong scaling*) of $32^3 \times 256$ and the fixed local lattice size (*weak scaling*) of $24^3 \times 32$ for multiple GPUs.

For the fixed local lattice size of $24^3 \times 32$, the performance of *dslash* for multiple GPUs only drops by about

3% compared to the performance of the single GPU version of *dslash* and stays nearly constant. On the other hand, the performance values for the fixed global lattice size of $32^3 \times 256$ are almost the same as the performance value for single GPU up to 16 GPUs with significant dropping in performance when there are 32 GPUs. The reason for the above performance drop is due to the effect of large surface-to-volume ratio when the fixed global volume is used with the large number of GPUs . The tiny performance degradation and excellent scalability of *dslash* for multiple GPUs demonstrate the high performance implementation of the GMH toolkit.

VI. Conclusions

This paper presents GMH, a message passing toolkit for multiple GPUs. This toolkit provides not only high performance MPI style communication primitives but also an event flow programming framework to ease the development effort for parallel iterative numerical solvers. At present, it is the only known GPU message passing toolkit that enables applications to transfer data among GPUs on just one host as well as within a cluster. In addition, it is the only known toolkit that allows applications mixing with any thread safe MPI implementation. It utilizes a thread group on a single host as a way to map GPU resources to an MPI rank, eliminates host memory copies when transferring data among GPUs on a single host, and introduces the notion of "virtual GPU" as a way to bind a thread to a GPU automatically when there are multiple GPUs on a single host. In addition, GMH delivers high point-to-point data transfer bandwidth only limited by the underlying GPU to CPU memory transfer bandwidth, and offers low point-to-point transfer and collective communication latency only restrained by the internal GPU-CPU memory transfer latency. More importantly, GMH offers flexible programming interfaces such that developers can either stick with the conventional MPI programming style or focus on the event flow of applications, which can be managed and executed by GMH at run time. Finally, the source code for GMH and its test programs used in this paper can be found at ftp://ftp.jlab.org/pub/hpc/gmh.tar.gz.

Acknowledgment

This work is supported by Jefferson Science Associates, LLC under U.S. DOE Contract DE-AC05-06OR23177.

References

[1] Advanced Micro Devices Inc., Brook+ SC07 BOF Session, In *Supercomputing Conference*, 2007.

[2] J. A. Anderson, C. D. Lorenz, and A. Travesset, General purpose molecular dynamics simulations fully implemented on graphics processing units, In *Journal of Chemical Physics*, vol. 227, no. 10, 5342-5359, 2008.

[3] M. A. Clark, R. Babich, K. Barros, R. C. Brower, and C. Rebbi, Solving Lattice QCD systems of equations using mixed precision solvers on GPUs, *arXiv:0911.3191v2* [hep-lat].

[4] Z. Fan, F. Qiu, A. Kaufman, S. Yoakum-Stover, GPU Cluster for High Performance Computing. In *Proceedings of ACM/IEEE Supercomputing Conference*, 41-47, Nov. 2004.

[5] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, S. H. M. Buijssen, M. Grajewski, and S. Turek, Exploring weak scalability for FEM calculations on a GPU-enhanced cluster, In *Parallel Computing*, vol. 33, no. 10-11, 685-699, 2007.

[6] O. S. Lawlor, Message Passing for GPGPU Clusters: cudaMPI, In *Proceedings of IEEE Cluster 2009*, 2009.

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

[8] A. Moerschell and J. D. Owens, Distributed texture memory in a multi-gpu environment, In *Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, 31-38, 2006.

[9] J. Nickolls, I. Buck, and M. Garland, Scalable Parallel Computing with CUDA, in *ACM Queue*, vol. 6, issue 2, 40-53, 2008.

[10] OpenCL: The open standard for parallel programming of heterogeneous systems, http://www.khronos.org/opencl/ .

[11] OpenGL organization, http://www.opengl.org/ .

[12] OPENFABRICS ALLIANCE, http://www.openfabrics.org/ .

[13] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, GPU Computing, In *Proceedings of the IEEE*, vol. 96, 879-899,2008.

[14] C. Ouarraui and D Kaeli, Developing Object-oriented Parallel Iterative Methods, In *International Journal High Performance Computing and Networking*, vol. 1, Nos. 1/2/3 , 2004.

[15] P. S. Pacheco, *Parallel Programming with MPI*, Morgan Kaufmann Publishers Inc., San Francisco, CA, 1996.

[16] L. Spracklen, S. G. Abraham, Chip Multithreading: Opportunities and Challenges, In *Proceedings of the International Symposium on High-Performance Computer Architecture*, 248-252, 2005.

[17] J. Stuart and J. D. Owens, Message Passing on Data-Parallel Architecture, In *IEEE International Symposium on Parallel & Distributed Processing*, 1-12, 2009.

[18] P. Vranas, The BlueGene/L Supercomputer and Quantum ChromoDynamics, In *Proceedings of ACM/IEEE Supercomputing Conference*, 2006.

[19] D. M. Wadsworth, Z. Chen, Performance of MPI broadcast algorithms, In *IEEE International Symposium on Parallel & Distributed Processing*, 1-7,2008.