A proposal of GMPI: GPU self MPI for GPU clusters

Yuta Kuwahara[†1], Toshihiro Hanawa[†2], Taisuke Boku[†1†3]
[†1]Graduate School of Systems and Information Engineering, University of Tsukuba
[†2]Information Technology Center, The University of Tokyo
[†3]Center for Computational Sciences, University of Tsukuba
kuwahara@hpcs.cs.tsukuba.ac.jp

***Abstract*—Today, Compute Unified Device Architecture (CUDA) is a typical programming environment for NVIDIA's GPUs, which are the most commonly used GPUs in the world. Because of its high performance and excellent performance/power feature, GPUs by NVIDIA and CUDA are also widely used for various application executions on GPU-ready clusters. Since CUDA is originally designed for single node solution, it is not equipped with any feature for internode communication such as MPI. Therefore, the internode communication among GPUs must be initiated by CPU after exiting from CUDA kernel function. When the communication is finished, the GPU computation is resumed by the CPU with executing another CUDA kernel function. In this way, the control returns to CPU from the CUDA kernel every time when a communication is needed and its overhead degrades the communication performance. The performance degradation may be serious for a fine-grained parallel GPU computing where the communications frequently occur. Moreover, the programmer must describe a complicated code, which includes CPU part, GPU part and communication part from the original code. This work reduces the productivity of programming seriously. To address this problem, we propose a parallel communication system named "GMPI" where the MPI communication functions can be invoked directly from GPU kernels within the CUDA framework. By this system, the overhead for GPU kernel invocation and synchronization between CPU and GPU at internode communication among GPUs is reduced, and the programmability and productivity of any code porting from the original MPI on CPU is increased. In this paper, we describe the design, implementation and performance evaluation of the prototype of GMPI system on a general GPU cluster. Currently, the performance of Ping-Pong communication is in the same level as the conventional method for a certain size of messages. For Himeno Benchmark, on the other hand, the performance tuning is not sufficient and it achieves approximately 60% of performance of original MPI+CUDA code. However, we confirmed the programmability on these benchmarks is much higher than the traditional way.**

***Keywords*—GPU Computing, MPI Programming, Software Framework**

I. Introduction

In these days, accelerating devices such as GPU (Graphics Processing Unit), MIC (Many Integrated Cores) or FPGA have become important components for high performance and low power HPC systems. Especially, GPGPU (General Purpose GPU) computing, which relies on high floating point operation performance and wide memory bandwidth of GPU, is commonly introduced for a number of HPC platforms. In these systems, each computation node is equipped with single or multiple GPUs as well as CPUs. In recent Top500 List[1], such GPU clusters or GPU-ready MPPs occupy a large fraction in the system architecture. The latest version of Top500 List on November 2015 includes Titan[2] (2nd rank), TSUBAME2.5[3] (25th rank) or Tianhe-1A[4] (26th rank), and HA-PACS[5] in University of Tsukuba is also ranked as 41st position.

Currently, the GPUs by NVIDIA are the most popularly used on these HPC GPU clusters, and the application programs on them are mainly written in CUDA (Compute Unified Device Architecture)[6]. Most of these HPC applications are parallelized by MPI[7] also, then the users must describe their code by the combination of CUDA and MPI environment. In CUDA, a computation offloaded part to be performed by GPU must be described as "kernel" function in a special manner, and these kernel functions are called from the host CPU at certain time. Hereafter, we call such a GPU kernel function as "kernel function". Since it is impossible to invoke the internode communication within a kernel function, it should be stopped every time to perform it by the CPU. This restriction naturally forces a user to split his/her original MPI code into a number of kernel functions, then the invocation cost of these kernels causes a certain level of overhead. It is also required to synchronize the data generated in GPU before MPI communication, and it introduces another overhead additionally to the communication cost. In this way, current mostly used programming method of a coupling with CUDA and MPI implies an overhead for kernel function invocation and data synchronization while it also degrades the code productivity by splitting codes into multiple kernel functions. It will be serious for fine grained parallel execution where these additional costs are heavy as much as communication cost itself.

In order to port a traditional MPI code for CPU to CUDA+MPI environment, the user must rewrite the code with a number of kernel functions to be split by each MPI function call. Especially for a fine grain code, it causes a large effort since the number of kernel functions increases and the code becomes heavily complicated.

To address these problems, we propose "GMPI" which is a GPU-self MPI system where any MPI function can be called directly within kernel functions. In this new concept, a kernel function can freely call MPI functions as like as ordinary CPU+MPI programs, and the code porting to GPU+MPI becomes much easier than ordinary CUDA+MPI coding. In

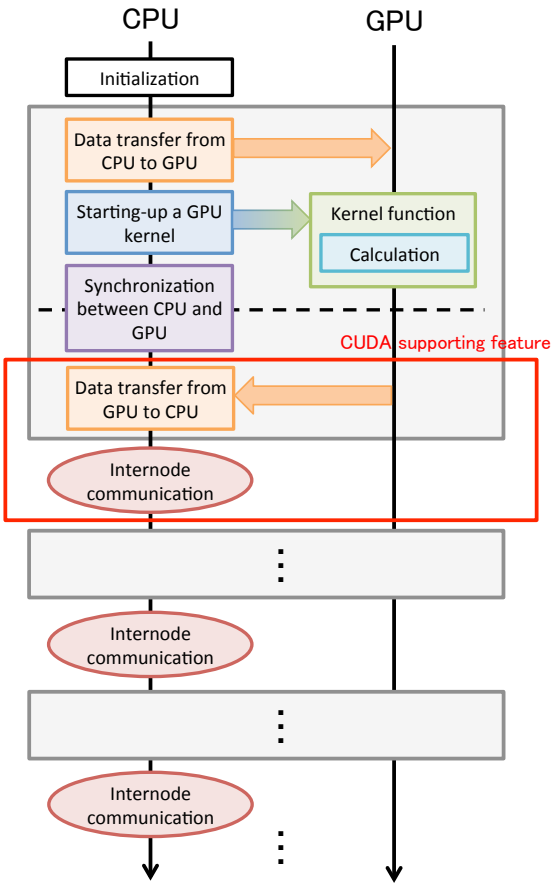

Fig. 1: Internode communication flow in GPU clusters

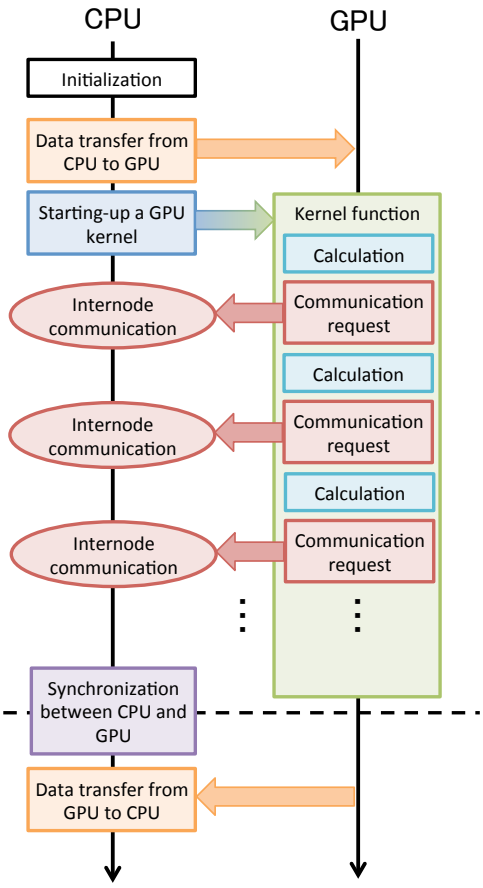

Fig. 2: Internode communication flow in GMPI system

this system we also expect to reduce the communication overhead which is caused by kernel function invocation and data synchronization in CUDA+MPI programming. Since we assume CUDA environment as background to support GMPI, it is applicable only for NVIDIA's GPUs.

II. Internode Communication on GPU Clusters

In this section, we summarize a general programming manner for internode communication on ordinary GPU clusters with CUDA, and describe the problem on it. In the rest of the paper, we call the host CPU as just "host" if it is not especially noted.

In general, GPU cannot take any action on inter-GPU communication regardless the partner is in or out of the node, thus the host must take care of the management and invocation of the communication. On the sender node, the host copies the data to be transferred from the device memory to the main memory of host, then invokes the communication function such as MPI_Send. On the receiver node, the host receives the message by an appropriate function such as MPI_Recv, then copies the data from host memory to the device memory.

In MPI environment, each MPI process executes the host program with CUDA kernel, and all the MPI function calls are performed by the host. Traditionally, MPI functions can handle only the host memory, so that it is required to transfer the target data between host memory and device memory. In CUDA, cudaMemcpy() or any appropriate function is used for it. However, the latency for this data transfer between these memories is high to cause an overhead for internode communication additionally.

In the recent MPI implementation to focus on GPU cluster computing such as MVAPICH2[8] or OpenMPI[9], it is possible to specify the device memory in GPU directly which is called CUDA supporting feature. For NVIDIA's GPUs, there is a feature called GPUDirect RDMA (GDR for short)[10] to allow PCIe devices such as InfiniBand HCA to access the device memory of GPU. Using this feature, the device memory address space that is declared from the host can be mapped on the PCIe address space. By accessing this mapped address space from any PCIe device, it can directly access the device memory. One of the earliest and typical examples is GPUDirect by Mellanox's InfiniBand HCA[11] that is ready for GDR feature. The HCA accesses the device memory without copying the target data to/from the host memory. Using the CUDA supporting feature to describe a program for MPI library with GDR, the total communication latency is reduced and high communication performance can

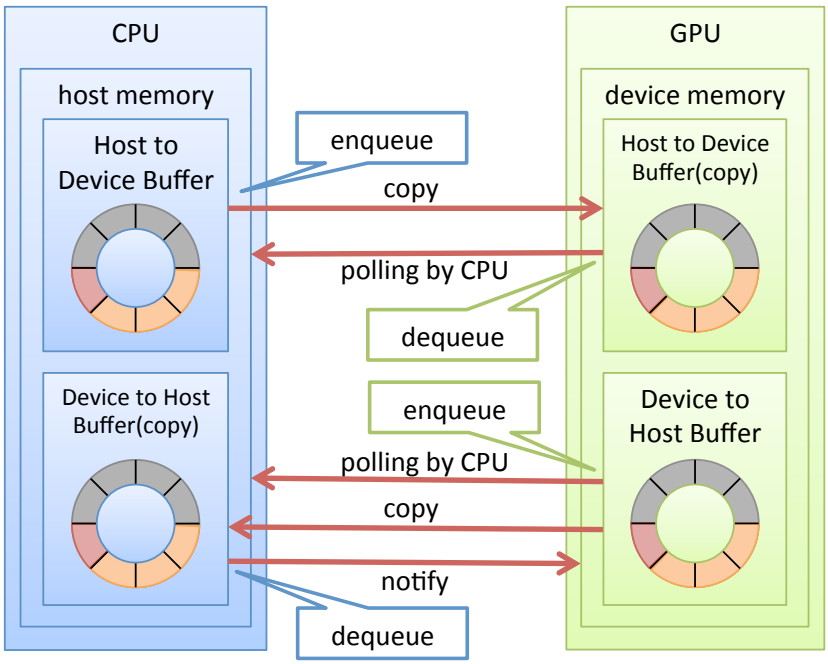

Fig. 3: Image of implementation of GMPI

be achieved.

In the environment of CUDA+MPI on GPU clusters, a program with internode communication is executed in a flow described in Fig. 1. Generally, the operations in the framed part must be described by the user. However, if the MPI supports the GDR feature, it is performed within MPI library. Although the GDR feature can omit a part of this process and reduce the latency, it is needed to quit from the executing kernel function before communication, and the host takes the actual communication. It means that the GPU kernel functions must finish its running before communication, then the next kernel function following the communication must be invoked. Moreover, it is required to synchronize the data within the host and device memories before or after the communication that causes another overhead.

III. GMPI

In this section, we propose a new programming feature for inter-GPU communication over nodes named "GMPI", and describe the design and implementation of it. GMPI is a feature where a running CUDA kernel can invoke MPI functions directly without quitting the function. During or after the MPI communication, the kernel continues to run and also can call the next MPI function, and so on. Currently, MPI functions that are supported by GMPI are shown in Tab. 1.

A. Concept

In Fig. 2, we show the program flow with communication on GMPI. However, this programming model is just from the user's viewpoint, and it is impossible to invoke any MPI function actually on GPU itself in the framework of CUDA. Instead of that, in the GMPI system, the host retrieves

Tab. 1: Supported MPI functions.

name	corresponding MPI function
GMPI_Isend	MPI_Isend
GMPI_Irecv	MPI_Irecv
GMPI_Wait	MPI_Wait
GMPI_Waitall	MPI_Waitall
GMPI_Allreduce	MPI_Allreduce

the communication request from the GPU by polling the memory mapped on the shared space between host and device memories. After it is requested to the host, it takes actual MPI function instead of GPU, then the result is returned to the GPU. Here, only the communication request and execution status are transferred between the host and GPU through the memory, and the actual data to be transferred is directly accessed by the communication devices (InfiniBand HCA) with GDR feature.

Remind that the GMPI function itself cannot reduce the communication overhead which is implied in a traditional way of CUDA+MPI programming because the actual MPI communication is performed in the same manner with traditional one. The important issue here is that the kernel function continues its run and there is no overhead for returning to GPU execution, that is, the invocation of next step of kernel function. If the cost to transfer the request of communication via memory polling is enough small compared with this kernel function invocation, we can reduce the total communication time. Moreover, it will save the user from describing a number of kernel functions to be split at any MPI function call. Thus, we may achieve both (1) reducing the total cost for internode communication by MPI and (2) reducing the programming cost or effort of users to port ordinary MPI programs into

CUDA-ready GPU systems.

B. Communication between Host and GPU

Fig. 3 shows the basic mechanism to invoke MPI communication from GPU kernel functions. Basically, all MPI communications are invoked on the host side according to the requests from GPU kernel function. The communication between host and GPU is performed through two sets of ring buffer on both sides, and each of host and GPU poll the status of ring buffers with each other. A set of ring buffer is used for requesting MPI activities from GPU to the host. The other set of ring buffer is used for returning the result of MPI communication from the host to GPU. The host can access the device memory on GPU anytime by memory copy functions in CUDA. For the memory copy functions, we describe later in this section. We assume a multicore feature on the host to run a supporting thread to watch the device memory by polling.

At first, we explain how to transfer the information from the host to GPU. After the host updates the content of "Host to Device Buffer" on the host side, it also updates "Host to Device Buffer (copy)" on GPU side. The host polls "Host to Device Buffer (copy)" periodically to confirm whether the information is read by GPU, by checking the head pointer position of the ring buffer. If the pointer is updated by GPU, the host updates the corresponding pointer on the host side of ring buffer to make the ring buffer consistent. Next, we explain how to transfer the information from GPU to the host. The host periodically polls "Device to Host Buffer" on GPU side to confirm the update from GPU side, by checking the tail pointer of the ring buffer. If any change is detected there, the host updates the content of "Device to Host Buffer (copy)". After it retrieves the data from this ring buffer, it also updates the head pointer of "Device to Host Buffer (copy)" to make it consistent. As described above, the host runs a status checking thread called "daemon thread" to check and update all the ring buffers to keep consistency. The actual memory copy to access the device memory on GPU takes relatively long time because it causes a transaction on PCIe bus for GPU memory access, and the polling does not affect seriously on the load of CPU core and memory buffer.

Since the device memory access from the host side is relatively expensive to take certain latency, it is not a good idea to transfer everything through these ring buffers. Therefore, we separate the information to specify the MPI communication parameters and actual data to be transferred, and use different way to handle them. We call the former set of data to contain everything except the actual data as "Attribute" in this paper. A set of Attribute includes the message type, message size, pointer for sending/receiving buffer and so on, which correspond to the arguments of MPI functions. For the handling of actual data to be transferred, we use GDR feature of MPI libraries, which are ready to use this feature to access the data on device memory directly, such as MVAPICH2-GDR. Our strategy is to transfer only Attribute to the host to invoke MPI functions, and to use GDR feature for actual data access not through the ring buffers.

To speed up the update of any ring buffer on the device memory, we manage it to be done by a set of 32 threads, which are in a Warp unit in CUDA. In a device function to update the ring buffer, each thread is assigned a unique integer ID which is created from blockIdx and threadIdx to identify whether that thread should work for this purpose or not and which element of Attribute is responsible to do it. To manage the coalesced data access by these threads effectively, we decided an Attribute to consist of up to 32 of elements of which size is 8 Bytes, so the size of Attribute is 256 Bytes (8 Bytes $\times$ 32 threads). Each of general data can contain pointer, integer data or any MPI-related data such as DataType. Every Attribute contains an integer to identify the MPI function to be called, and it is allocated as the first element of the Attribute. The host can find the number of arguments based on the identifier, so that there is no data on the size of Attribute. Following to it, all arguments of that MPI function are written as the elements of the Attribute to the ring buffer simultaneously by 32 threads according to the unique ID of each thread to be an index to the memory for the coalescing access. To control this feature effectively, the size of Attribute is fixed to 32 elements. We decided this number, 32, according to the typical number of arguments for commonly used MPI function and also considering the thread count in a Warp to make the best effect of memory coalescing to maximize the data set-up speed. As a result, the content of any ring buffer is blocked in the Attribute size, which is 256 Bytes unit.

C. MPI Function Invocation on Host Side

After the host detects the request from GPU through "Device to Host Buffer" by polling, it recognizes which MPI function is required to execute with certain arguments through the Attribute. Then, the daemon thread invokes actual MPI function. Please remind that all the request and status report is done through the ring buffers, and the host (daemon thread) and GPU run asynchronously to hide the communication latency.

As described in the next sub-section, we rely on the GDR feature of MPI and the "address pointer" to specify the data buffer for MPI communication on the device memory is finally passed to the real MPI function as the same meaning of pointer. Therefore, we do not have to care of the address conversion of buffer pointer and just handle it as a "general data" in the Attribute. In this way, it helps us to design the system without considering any problem on address management, and we can utilize the feature of GDR effectively.

After invoking the MPI function, the daemon thread in host waits for returning of the function, regardless it is synchronous or asynchronous function, and make an entry for sending back the return status to GPU. It is treated as in the same manner for Attribute sending from GPU but the reversed way to GPU at this time. Finally, the result and status report are returned to the kernel function, which invokes the MPI function call.

D. Communication Cost between Host and GPU

When the daemon thread on the host accesses a ring buffer on the device memory on GPU, a memory copy transaction

Tab. 2: Host-GPU data copy evaluation environment

Node configuration	
CPU	Intel Xeon CPU E5-2670 v2
# of cores	10cores/socket × 2 sockets
Clock freq.	2.50GHz
Peak performance	332.8 GFLOPS/node
Memory	128 GB, DDR3 1866 MHz × 4ch
Motherboard	Supermicro X9DRG-QF
GPU	NVIDIA K20
# of GPUs	2GPUs/node
Peak performance	1.76TFLOPS/GPU
Memory	5GB/GPU (with ECC)
InfiniBand HCA	Mellanox Connect-X3 FDR Dual-port (PCIe 3.0 x8)
Software	
OS	CentOS 6.5
GPU driver	NVIDIA-Linux-x86_64-340.29
CUDA	CUDA 6.5
MPI	MVAPICH2-GDR 2.1a

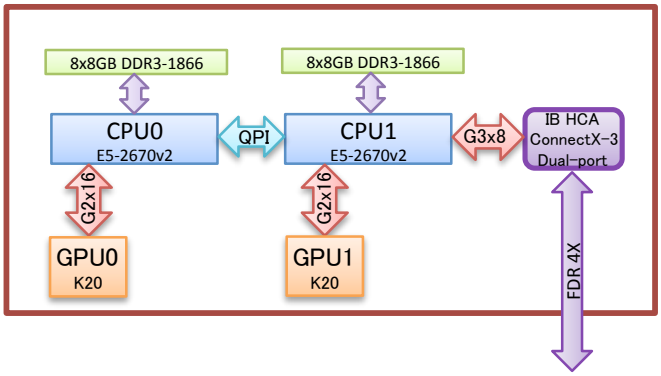

Fig. 4: Block diagram of evaluated node

is invoked on. In general, cudaMemcpy function is used for it as a general API for memory copy in CUDA, however this function causes relatively long latency. Before current implementation, we tried another feature of CUDA, which seemed to be more sophisticated to utilize "mapped page-locked memory" where the actual data exists on the host side and GPU can access it. However, we found that it takes a couple of μ seconds of cost and the overhead is too large. To save this cost, we decided to use "gdrcopy()"[12] function to make a copy of the specified portion of device memory into the host memory. The latency of gdrcopy() is shorter than either cudaMemcpy or mapped page-locked memory.

However, the latency of gdrcopy() increases almost proportional to the data size to be copied. In the GMPI system, the first element of Attribute, which identifies the MPI function, is copied at first to the host, then only the required elements are copied later. It seems to be redundant in ordinary memory system, but we confirmed it is better way than copying a fixed size data (256 Bytes of Attribute elements) at once. Please remind that an Attribute entry always managed as 256 Bytes unit for coalescing access on device memory by GPU threads, however the actual data transfer to the host memory is performed only for the actually required data.

IV. Preliminary Performance Evaluation

In this section, we examine the cost for kernel function invocation and data synchronization to transfer Attribute between the host and GPU, prior to the evaluation of GMPI system.

A. Environment

Tab. 2 and Fig. 4 show the node configuration and block diagram of the target environment, respectively. Numactl feature is used to invoke the program on the host CPU as well as specifying GPU to be used by CUDA_VISIBLE_DEVICES environment variable to specify the directly attached GPU to the CPU socket for minimizing the data transfer overhead. For MVAPICH2 library, the environment variable MV2_USE_CUDA is always on to use GDR and CUDA supporting feature.

B. Cost for Kernel Function Invocation and Synchronization

In the traditional CUDA+MPI method, each MPI function call requires to return the control from kernel functions to the host and the following kernel function is invoked again after MPI function finished. It means that each MPI function call causes an overhead for kernel function invocation and data synchronization between the host and GPU. In GMPI, an overhead for data copy between the host and device memory to pass Attributes and result instead of invoking kernel functions. Therefore, it is desired that the former overhead is smaller than the latter one in the meaning of performance. Beside of the performance issue, we think that the GMPI system reduces the programming cost of users, however it is difficult to evaluate generally. We discuss later on this issue.

To examine how the overhead differs between two methods, we evaluated the time for kernel function invocation and data synchronization in several ways. We evaluated three patterns; (1) kernel function invocation only, (2) kernel function invocation and data synchronization by cudaDeviceSynchronize(), and (3) kernel function invocation and data synchronization by cudaStreamSynchronize(). The result is shown in Tab. 3. From this result, we found that the kernel function invocation takes approximately 3.2 μ seconds, and the whole operation including data synchronization takes 11 μ seconds. In general, it is always required to perform synchronization before calling MPI functions under GDR feature. Therefore, we need to consider the total of them.

C. Evaluation on Attribute Transfer Cost

In GMPI, each thread on a CUDA Warp of 32 threads makes a copy of single element of Attribute from the global memory to the ring buffer entry. Since the first element is used to identify the MPI function itself, the maximum number of arguments is 31, which can be transferred by a single entry of Attribute. We examined the data copy latency for gdrcopy() varying the element size from 1 to 31. In this evaluation, we took care of the synchronization due to BIOS behavior of CUDA kernel function. The data stored into GPU global memory by CUDA thread or the host is not written back immediately while the kernel function is running. This

Tab. 3: Overhead for kernel function invocation and synchronization

Method	Processing time [μs]
Kernel invocation only	3.23
Kernel invocation and cudaDeviceSynchronize()	12.5
Kernel invocation and cudaStreamSynchronize()	10.9

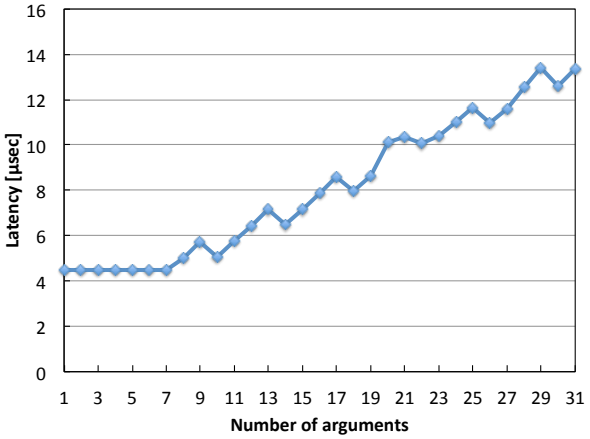

Fig. 5: Cost evaluation for Attribute transfer

design does not cause problem in typical usage, in which performs synchronization after invocation of kernel function on the host. Therefore, we inserted __threadfense() function on GPU and _mm_sfense() on the host to guarantee the data synchronization. To measure the execution time for the data copy and the synchronization on GPU side, we use clock64() function which reads the hardware cycle counter on GPU to measure the time precisely. This feature is suitable for such a purpose because the time measuring from host CPU side always implies much of hidden overhead. By this function, we purely measured the time consumed within GPU.

The result is shown in Fig. 5. The horizontal axis represents the number of arguments in Attribute and the vertical axis shows the time consumption. From the result, it is shown that the cost for Attribute transfer takes 4 to 13 μ seconds according to the number of arguments where larger number of arguments takes longer time. There are some perturbations on the measurement and we could not erase it. However, we can observe that approximately 22 or 23 of arguments transfer costs less than 11 μ seconds which is the overhead by the traditional way. Especially for the case with 7 or less number of arguments, the cost is almost constant with only 4.5 μ seconds. This result encourages us to give a possibility of advantage by GMPI against to the traditional method with frequent kernel function invocations.

V. Parallel Execution Model under GMPI

In the above sections, we concentrated only to how to implement the MPI communication invocation by a series of protocol between GPU and the host. In this section, we briefly consider the parallel execution model of GPU computing with GMPI.

Not like an ordinary thread-level parallelization such as in OpenMP on CPU, the parallel execution with SIMT (Single Instruction and Multiple Thread) execution model in CUDA relies on the concept where a large number of threads exist in parallel at the time of kernel function invocation, and every operation in it is executed in parallel under a single stream of execution. Sometime, the execution stream is split into two ways (maybe after *if* statement) but it does not mean the simultaneous execution of two streams. Actually, the other side of stream is stopped during the *if taken* side of stream execution, which is called "Warp Split" to reduce the parallel efficiency. Anyway, we assume the execution stream is always one at a time.

Well, in such a code, what is the semantics of GMPI call? We like to explain the situation by referring the case of OpenMP code. In the most of OpenMP code, every programmer describes the MPI function call in a "single context" where only a single thread executes the MPI communication, then the transferred data are referred by multiple threads. Recent MPI is also ready for multi-thread safeness where these parallel threads can call MPI functions individually, but it generally reduces the visibility of code as well as the communication performance degradation caused by fine grained communication. Most of users describe the code as in the concept of *incremental* parallelization on OpenMP where they parallelize the code part by part to improve the computation speed. In this style, they basically do not touch the MPI communication to keep it as in a single thread execution protecting the MPI part in critical by *single* or *master* clauses within *parallel* portion in OpenMP. Otherwise, they parallelize the sequential code to parallelize each *for* loop in step by step manner by *parallel for* clause. In this case, the MPI function calls are also kept in a sequential context.

There is a new trend of OpenMP programming to introduce the concept of \ em task aggressively to exploit a large degree of asynchronicity supported by thread-safe MPI execution, however such a programming style is different from the simple data parallel concept where GPU programming relies on. Therefore, we consider the traditional OpenMP+MPI programming with single stream of serialized MPI function call as the typical application framework for GMPI.

According to this idea, we rule the execution model of GMPI as follows. A user can describe any GMPI function call in his/her kernel function, but that function is executed "only once at that time". It means that each parallel thread does not execute the MPI function in parallel, but only single

```
__global__ void send_kernel(gmpi_buf_t *buf, int *src, int *dst, MPI_Request *request, MPI_Status *status)
    {
  for(int i = 0; i < N_ITER; i++) {
    GMPI_Isend(buf, src, DATA_SIZE, MPI_INT, 1, i, MPI_COMM_WORLD, request);
    GMPI_Irecv(buf, dst, DATA_SIZE, MPI_INT, 1, i, MPI_COMM_WORLD, request);
    GMPI_Wait(buf, request, status);
    GMPI_Synchronize(buf);
  }
}

__global__ void recv_kernel(gmpi_buf_t *buf, int *src, int *dst, MPI_Request *request, MPI_Status *status)
    {
  for(int i = 0; i < N_ITER; i++) {
    GMPI_Irecv(buf, dst, DATA_SIZE, MPI_INT, 0, i, MPI_COMM_WORLD, request);
    GMPI_Wait(buf, request, status);
    GMPI_Isend(buf, src, DATA_SIZE, MPI_INT, 0, i, MPI_COMM_WORLD, request);
    GMPI_Synchronize(buf);
  }
}
```

Fig. 6: Example of Ping-Pong communication

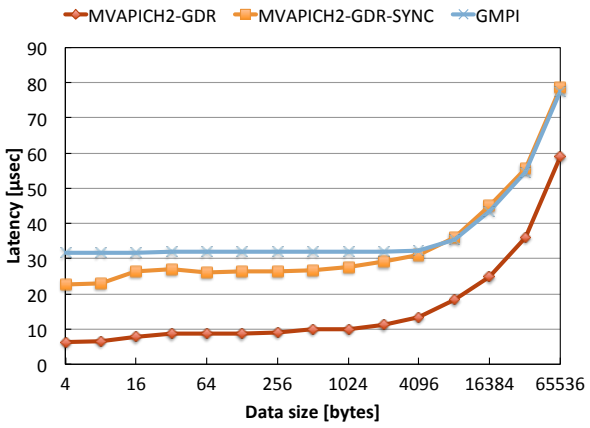

Fig. 7: Performance of Ping-Pong communication

context of MPI function call is executed. We think it does not cause any confusion on user's programming scope as same way in the OpenMP parallelization with MPI for single stream of communication. This is the general rule to understand for GMPI parallel execution model. Under this assumption, we partially utilize a set of threads in a Warp to help the data copy of ring buffer as described previously.

VI. Performance Evaluation

In this section, we evaluate the performance of GMPI on two benchmarks; Ping-Pong communication and Himeno Benchmark[13]. All the performance is evaluated on the same environment shown in Tab. 2.

A. Performance Evaluation on Ping-Pong Communication

At first, we explain the GMPI version of code for Ping-Pong communication. The code in Fig. 6 shows the actual GMPI program for it. In this evaluation, two GPUs on different node execute ether send_kernel() or recv_kernel() for Ping-Pong communication. One initiates the sending data while the other initiates the receiving it, then the sender and receiver are changed and they repeat it.

GMPI functions are defined as device functions which can be called from the kernel functions and declared by __device__ prefix. In this code, GMPI_Isend(), GMPI_Irecv() and GMPI_Wait() are the GMPI functions. In each device function, the identifier for MPI function to be called actually on the host is set as the first element of Attribute, then all actual arguments follow. The sequence of arguments is completely the same with that in the original corresponding MPI function. *src* and *dst* are the pointers to the buffers allocated in the device memory by cudaMalloc() or just statically declared. A user can describe the buffer pointer just as like as in GDR-ready MPI because they are actually passed to the MPI function with CUDA GDR feature on the host side.

GMPI_Synchronize() is an extra function originally designed for GMPI to complete all the requested MPI functions on the host. Here, the argument "buf" is a pointer to the ring buffer management entity with a special structure of "gmpi_buf_t". GMPI_Synchronize() checks the counter variables both on two ring buffers ("Device to Host Buffer" and "Host to Device Buffer (copy)") and wait for the matching of them to confirm the completion of all requests. In the future, this function should be automatically included within any GMPI functions such as GMPI_Waitall() where the synchronization with the host is required.

As shown in this example, a user can describe the MPI communication almost in the same manner with its original MPI program on CPU. We think this programming method greatly reduces the user's effort to port the code to CUDA+MPI as well as reducing the communication overhead by frequent invocation of kernel functions.

The result of Ping-Pong benchmark is shown in Fig. 7. For the reference, we also show two cases by MPI; "MVAPICH2-GDR" for the case with MVAPICH2 without kernel function invocation cost and "MVAPICH2-GDR-SYNC" for the case with all overheads for kernel function invocation and data synchronization. The former one is shown just as the basic

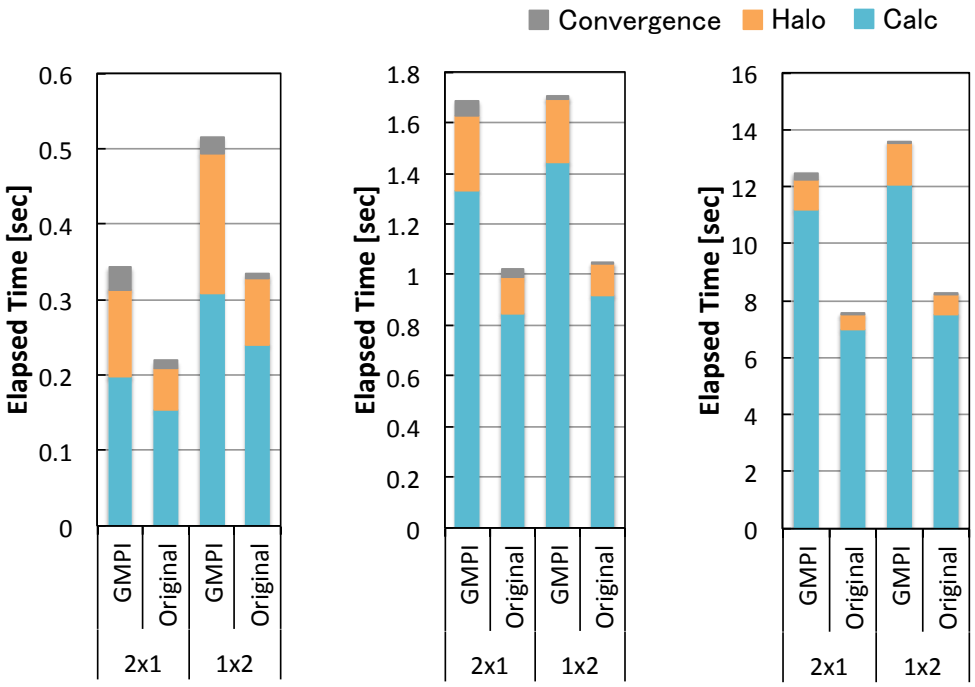

Fig. 8: Performance result of Himeno Benchmark (from the left: SMALL, MIDDLE and LARGE)

Tab. 4: Himeno Benchmark problem size(i × j × k)

SMALL	MIDDLE	LARGE
64 × 64 × 128	128 × 128 × 256	256 × 256 × 512

Tab. 5: Himeno Benchmark lines of code

Method	Lines of code
CUDA+MPI version	1280
GMPI implementation	1246

performance of MVAPICH2-GDR, and the latter one corresponds to the actual situation of real applications. The Ping-Pong communication is executed 1,000 times varying the data size from 4 Bytes to 64 KBytes. The vertical axis represents the average latency for single iteration.

From the result, it is shown that the overhead of transferring Attribute and result status between the host and GPU can be negligible when the message size increases, and the difference between GMPI and traditional way (MVAPICH2-GDR-SYNC) is reduced. For message size with 4K Bytes or larger, GMPI achieves the same performance with the traditional way. Based on our preliminary performance evaluation on the cost to transfer Attribute, it is expected that the performance of GMPI is the same with MVAPICH2-GDR-SYNC, however the result shows there exists several μ seconds of overhead on GMPI. We think this extra overhead is caused by the synchronization on the local device memory. From the result, it seems to be approximately 10 μ seconds of overhead. It will be a future work for GMPI to reduce this overhead. However, we can confirm that GMPI works almost as we expected, and the program description is much easier than the traditional way.

B. Performance Evaluation on Himeno Benchmark

Next, we evaluate the performance of 2-D Himeno Benchmark for the confirmation of the correctness of GMPI system including its programming and behavior as well as the actual performance. 2-D Himeno Benchmark is a simple stencil computing benchmark by domain decomposition on 2-D problem space and it includes the nearest neighboring communication on the halo of the decomposed domain after the internal computation by Jacobi Method and a collective communication to get the convergence status by MPI_Allreduce() for a scalar variable. The original Himeno Benchmark with MPI for CPU execution is rewritten for performance optimization on Kepler K20 GPU and ready for domain decomposition in either i or j dimension[14][15]. Hereafter, we call it as "CUDA+MPI version". Since we have only two nodes for the tested on this evaluation, the domain is decomposed in 1×2 or 2×1 for i and j dimensions. The iteration count for total computation including Jacobi Method, halo exchanging and collective communication is fixed to 1,000 and we examine three problem sizes shown in Tab. 4. For the communication on the benchmark, we use GMPI_Waitall() and GMPI_Allreduce() additionally to GMPI_Isend() and GMPI_Irecv(). We implement this benchmark as a kernel function for computation on inner data, nearest neighboring communication for halo exchange,

and the convergence check where these three functions are described as device functions for visibility. All MPI functions are replaced to GMPI functions of course.

The lines of codes of CUDA+MPI version and GMPI implementation are shown in Tab. 5. The lines of code of GMPI implementation has become shorter than CUDA+MPI version. The result is shown in Fig. 8. It is shown that the total performance of entire benchmark achieves approximately 60% of CUDA+MPI version on any problem size. This result does not fit to our previous performance evaluation on Ping-Pong benchmark because the message size is enough large where GMPI is expected to achieve the same performance with the traditional way in the communication. To analyze the reason of this performance degradation, we examined the execution time of each device functions on computation, halo exchange and convergence check by Clock64() function for precise measurement. The breakdown of the execution time is shown as colored bars in the figure. The communication time for Convergence is larger than CUDA+MPI version, however it is reasonable because the data size is 8 Bytes scalar (double precision floating point) and it is too small for GMPI. The communication time for halo exchange on GMPI is also larger than CUDA+MPI version, however the difference gets smaller when the problem size increases because of the message size increased. On the other hand, we found the computation time is increased in GMPI implementation as shown as blue-colored bars in the figure. It is weird because there is no difference in the inner data computation for Jacobi Method both in GMPI and CUDA+MPI version, and this increase of computation time essentially causes the performance difference.

We are analyzing the reason of this strange result on the computation time. Currently, the reason is unclear and we are guessing there are some side effects on BIOS scheduling, data copy handling, etc. It is our current future work to complete, then we examine the communication performance in detail toward evaluating more GMPI applications.

As described above, there are several problems on the performance of GMPI not just on the communication but also on other GPU behaviors. However, in both benchmarks, we confirmed that the programmability or portability of original MPI program to CUDA environment becomes much easier by GMPI compared with the traditional CUDA+MPI style. If the performance degradation is reduced in a certain level, GMPI will be quite beneficial for the application users who want to port their MPI code to CUDA environment.

VII. Related Works

In the University of Electro-Communications, a system named FLAT is proposed as a programming framework to allow MPI function calls in a kernel function in CUDA[16]. FLAT is implemented as a preprocessor to the compiler and it converts the target code in source-to-source manner to expand the MPI function call to be rewritten to the form of traditional way, that is, a collection of split kernel functions and MPI invocation on the host, in CUDA source code. While this system is basically a translator to the MPI program and there is no fundamental change on the execution model, our GMPI solution is providing the real function call within the kernel function to request the invocation of MPI function to the host. Thus, FLAT does not solve the problem on kernel function invocation and synchronization cost, however GMPI has a room for performance tuning on data copy and polling.

CUQU (CUDA queue)[17] and CUOS (CUDA Offload System services)[18] are the system based on a similar idea with GMPI, developed in Sapienza University of Rome. CUQU is developed under CUDA environment and the communication between the host and GPU is performed over page-locked memory (pinned memory) we also examined before. CUOS is a prototype of framework to call the service on the host from GPU kernel function, and an example to invoke the MPI synchronous communication is implemented on CUQU. However, its implementation depends on CUDA 4.0 and the project is finished on May 2011, so there is no follow up research on it. We could not retrieve the working library and there is no way to confirm nor compare them with GMPI. In GMPI, we are improving the system according to the development of recent CUDA features to achieve higher performance.

Another related work is in the University of Texas at Austin, named GPUnet[19]. GPUnet is based on RDMA over InfiniBand technology to provide the feature to handle socket on GPU. It utilizes the ring buffer structure shared between the host and GPU, and GPU requests a socket level communication to the host. The host accepts the request based on the polling of shared memory, and handles the request. Although there is a difference between GPUnet and GMPI on the way to handle the information between the host and GPU, this is the most similar concept with GMPI. However, GPUnet targets the socket level communication which seems to be too fine grained and detailed protocol for HPC applications while GMPI is based on MPI level communication to achieve higher abstraction and efficient execution. GPUnet focuses on lower level of communication to provide various services such as GPU-ready embedded communication system, and not effective for HPC applications.

VIII. Conclusions and Future Works

In this paper, we proposed a new programming framework named GMPI which enables direct communication invocation on MPI level over CUDA GPGPU environment for HPC applications, and implemented it with several basic performance evaluations. In traditional way, upon the restriction of MPI implementation, kernel function needs to be exited and resumed every time on inter-node communication. This results in the increase of implementation and communication costs. Our proposal method, GMPI, removes this restriction and those costs.

In the preliminary performance evaluation on the communication between the host and GPU, we examined the number of arguments to pass to MPI through the communication feature within a node. We found that the request overhead is same level compared with the overhead for kernel function

invocation and data synchronization in the traditional way. On the actual benchmarks, however, there some hidden overhead exists and we need to optimize the system to reduce them.

Ping-Pong benchmark on GMPI achieves from approximately 70% to comparable performance with the traditional way keeping the easiness of programming for users. On Himeno Benchmark, the performance stayed around 60% of the CUDA+MPI version, however we found the problem exists not on the communication part but on the computation part which is affected by some side-effect of GMPI system. The analysis is under going to achieve the comparable performance with the traditional CUDA+MPI style.

Our future works include the followings. There are several issues on performance tuning and reducing the overhead on GMPI system. It is still under development and we continue to detect any overhead or redundancy on the processing in the system. Most importantly, we need to examine what happens on Himeno Benchmark where the computation time itself increased within a GPU. After the performance tuning, we apply GMPI for basic benchmarks such as NAS Parallel Benchmarks[20] and other typical applications to evaluate its practical performance and programmability for further productive parallel GPU computation. Finally, we will apply GMPI for our original GPU communication system named TCA (Tightly Coupled Accelerators)[21] for further performance improvement.

Acknowledgment

The present study was supported in part by the JST/CREST program entitled "Research and Development on Unified Environment of Accelerated Computing and Interconnection for Post-Petascale Era" in the research area of "Development of System Software Technologies for post-Peta Scale High Performance Computing." The use of HA-PACS/TCA in the present study is offered under the "Interdisciplinary Computational Science Program" in Center for Computational Sciences, University of Tsukuba. The authors would like to thank all these supports for this work.

REFERENCES

[1] TOP500 Supercomputer Sites (online), http://top500.org/
[2] The Oak Ridge Leadership Computing Facility introduces Titan (online), https://www.olcf.ornl.gov/titan/
[3] GSIC. TSUBAME Computing Services (online), http://tsubame.gsic.titech.ac.jp/en
[4] The calculations were performed on the TianHe-1 (A) supercomputer located at National Supercomputer Center in Tianjin (online), http://nscc-tj.gov.cn/en/resources/resources_1.asp#TH-1A
[5] Center for Computational Science, University of Tsukuba : HA-PACS Project (online), http://www.ccs.tsukuba.ac.jp/eng/research-activities/projects/ha-pacs/
[6] CUDA C Programming Guide (online), http://docs.nvidia.com/cuda/cuda-runtime-api/index.html
[7] Message Passing Interface (MPI) Forum Home Page (online), http://www.mpi-forum.org/
[8] MVAPICH2: High performance MPI over InfiniBand/10GigE/iWARP and RoCE (online), http://mvapich.cse.ohio-state.edu/
[9] Open MPI: Open Source High Performance Computing (online), http://www.open-mpi.org/
[10] NVIDIA Corp. : NVIDIA GPUDirect. (online), http://developer.nvidia.com/gpudirect
[11] Mellanox GPUDirect RDMA User Manual Rev 1.0 (online), http://www.mellanox.com/related-docs/prod_software/Mellanox_GPUDirect_User_Manual_v1.0.pdf
[12] NVIDIA gdrcopy. (online), https://github.com/NVIDIA/
[13] Himeno benchmark, RIKEN, Japan. (online), http://accc.riken.jp/en/supercom/himenobmt/
[14] E. Phillips, M. Fatica, "Implementing the Himeno benchmark with CUDA on GPU clusters Parallel & Distributed Processing." (IPDPS), 2010 IEEE International Symposium, pp.1 – 10, Apr. 2010.
[15] T. Hanawa, Y. Kodama, T. Boku, M. Sato, "Tightly Coupled Accelerators Architecture for Low-latency Inter-Node Communication Between Accelerators." in SC14 poster, Nov. 2014.
[16] K. Shima, M. Yoshimi, T. Miyoshi, M. Kondo, H. Irie, H. Honda, T. Yoshinaga, "FLAT: An MPI Friendly GPGPU Programming Framework for GPU Clusters." in IPSJ Transactions on Advanced Computing System (ACS) , Vol. 6, No.4, pp. 105 – 116 (2013) (in Japanese).
[17] cuqu A CPU ↔ GPU messaging queue (online), https://code.google.com/p/cuqu/
[18] cuos Offloaded System services for CUDA (online), https://code.google.com/p/cuos/
[19] S. Kim, S. Huh, X. Zhang, Y. Hu, A. Wated, E. Witchel, M. Silberstein, "GPUnet: Networking Abstractions for GPU Programs." in 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), pages 201 – - 216, Broomfield, CO, October 2014. USENIX Association.
[20] NAS Parallel Benchmark (online), http://mikilab.doshisha.ac.jp/dia/smpp/01_bench/naspara.html
[21] T. Hanawa, Y. Kodama, T. Boku, and M. Sato, "Tightly Coupled Accelerators Architecture for Minimizing Communication Latency among Accelerators." in The Third International Workshop on Accelerators and Hybrid Exascale Systems (AsHES) in conjunction with IEEE 27th International Parallel and Distributed Processing Symposium (IPDPS), May 2013, pp. 1030 – - 1039