clMPI: An OpenCL Extension for Interoperation with the Message Passing Interface

Hiroyuki Takizawa*, Makoto Sugawara*, Shoichi Hirasawa*, Isaac Gelado*, Hiroaki Kobayashi*, and Wen-mei W. Hwu†

* Tohoku University, Sendai, Miyagi 980-8579, Japan
E-mail: {tacky@, makoto@sc., hirasawa@sc., koba@}isc.tohoku.ac.jp
† The University of Illinois at Urbana-Champaign
E-mail: w-hwu@illinois.edu
‡ Advanced Center for Computing and Communication, RIKEN

Abstract—This paper proposes an OpenCL extension, clMPI, that allows a programmer to think as if GPUs communicate without any help of CPUs. The clMPI extension offers some OpenCL commands of inter-node data transfers that are executed in the same manner as the other OpenCL commands. Thus, clMPI naturally extends the conventional OpenCL programming model so as to improve the MPI interoperability. Unlike conventional joint programming of MPI and OpenCL, CPUs do not need to be blocked to serialize dependent operations of MPI and OpenCL. Hence, an application can easily use the opportunities to overlap parallel activities of CPUs and GPUs. In addition, the implementation details of data transfers are hidden behind the extension, and application programmers can use the optimized data transfers without any tricky programming techniques. As a result, the extension can improve not only the performance but also the performance portability across different system configurations. The evaluation results show that the clMPI extension can use the optimized data transfer implementation and thereby increase the sustained performance by about 14% for the Himeno benchmark if the communication time cannot be overlapped with the computation time.

I. INTRODUCTION

Today, many high-performance computing (HPC) systems are equipped with graphics processing units (GPUs) as data-parallel accelerators in addition to conventional general-purpose processors (CPUs). For such a heterogeneous HPC system, application programmers need to manage the system heterogeneity while exploiting the parallelism involved in their applications; they have to develop their applications by combining multiple programming models such as MPI[1], OpenMP[2], and GPU programming models[3][4]. Those programming models have been designed independently, and are generally treated as orthogonal ones. Consequently, application programmers are supposed to appropriately combine those programming models for fully exploiting the potential of a heterogeneous HPC system.

One problem is that there is no standard, established way to combine those programming models even for common programming patterns. They are combined in various ways, and the combination is written directly in application codes. Particularly, GPU programming models such as CUDA[3] and OpenCL[4] need to collaborate closely with MPI for data transfers among GPUs in different nodes. The joint programming of MPI and CUDA/OpenCL is likely to be tricky and system-specific. Therefore, it would complicate application codes and degrade the readability, maintainability, and performance portability of the applications.

Another problem is that CPUs are supposed to play several roles at once in the joint programming of MPI and CUDA/OpenCL. To serialize MPI and CUDA/OpenCL operations due to their dependency, a CPU is usually blocked until the preceding operation is completed, and then is unblocked to start the subsequent operation. This kind of blocking often inhibits overlapping of computation and communication, and exposes the communication overheads. One idea would be to create a dedicated thread for synchronizing the operations. However, such multi-threading will further increase the programming complexity. Consequently, the application performance strongly depends on the programming skills and craftsmanship of the developers.

To address the problems, we need a “bridging” programming model that provides a standard way to combine MPI and GPU programming models. In this paper, we focus on OpenCL as the GPU programming model for high code portability, and propose an OpenCL extension for interoperation with MPI, though the idea could be trivially extrapolated to other GPU programming models such as CUDA. The proposed OpenCL extension named clMPI provides an illusion that GPUs are communicating with each other asynchronously with MPI processes running on CPUs, even though they internally use MPI and perform the data transfers with help of CPUs. To this end, the extension offers some OpenCL commands for inter-node communication and functions required for collaboration with MPI. The inter-node communication commands are executed in the same manner as the other OpenCL commands, and hence the OpenCL programming model is naturally extended so as
to improve the MPI interoperability.

The cIMPI extension provides a portable, standardized way to program the communication among GPUs in different nodes, and also among CPUs and remote GPUs; it can thereby hide the system-aware optimized implementations behind function calls. Hence, we can also expect that the cIMPI extension improves the performance portability across different system types, scales, and generations.

The rest of this paper is organized as follows. Section II briefly reviews the related work. Section III discusses the difficulties in joint programming of MPI and OpenCL. Then, Section IV proposes cIMPI, which is an OpenCL extension for the interoperation with MPI. Section V discusses the performance impact of cIMPI through some evaluation results. Finally, Section VI gives concluding remarks and our future work.

II. RELATED WORK

In the OpenCL programming model, a CPU works as a host that manages one or more compute devices such as GPUs. For the management, OpenCL provides various resources that are instantiated as OpenCL objects such as contexts, command queues, memory objects, and event objects. A unique handle is given to every object, and is used to access the resource. A context is a container of various resources, and is analogous to a CPU process. A command queue is used to interact with its corresponding compute device; a host enqueues a command to have its compute device execute a task. A memory object represents a memory chunk accessible from hosts and devices. An event object is bound with a command in the command queue to represent the status of the command, and is used to block the execution of other commands. Hence, it is used to describe the dependency among commands. Moreover, multiple events can be combined to an event list to express several previous commands.

For example, clEnqueueReadBuffer is a typical OpenCL function for enqueuing a command, which reads data from a memory object. The function signature is as follows:

```c
cl_int
clEnqueueReadBuffer( cl_command_queue cmd,
                     cl_mem buf,
                     cl_bool blocking,
                     size_t offset,
                     size_t size,
                     void* hbuf,
                     cl_uint numevts,
                     cl_event* wlist,
                     cl_event* evtret )
```

OpenCL command enqueuing functions take three arguments for event management: the number of events in the waiting list (numevts), the initial address of the waiting list (wlist), and the address to which the event object of the enqueued command is passed (evtret). The enqueued command is able to be executed when all the preceding commands associated with the event objects in the waiting list have been completed.

In joint programming of MPI and OpenCL, a programmer needs to consider not only host-device communication using OpenCL but also inter-node communication using MPI. So far, some researchers have presented several MPI extensions to GPUs to ease the joint programming of MPI and CUDA/OpenCL. We will refer to these approaches as GPU-aware MPI implementations. Lawlor has proposed cudaMPI [5] that provides an MPI-like interface for communication between remote GPUs. MPI-ACC[6] uses the MPI_Datatype argument to indicate that the memory buffer passed to an MPI function is located in the device memory. MVAPICH2-GPU[7] assumes Unified Virtual Addressing(UVA), which provides a single memory space for host and device memories, and checks if the memory buffer passed to an MPI function is in the device memory. Then, MVAPICH2-GPU internally uses different implementations by depending on whether the memory buffer in the device memory. Stuart et al. have discussed various design options of MPI extension to support accelerators [8]. Gelado et al. proposed GMAC that provides a single memory space shared by a CPU and a GPU and hence allows MPI functions to access device memory data [9]. Those extensions allow to use a GPU memory buffer as the end point of MPI communication; the extended MPI implementations enable to use MPI functions for inter-node communication from/to GPU memory buffers by internally using data transfer functions of CUDA/OpenCL.

By using GPU-aware MPI extensions, application developers do not need to explicitly describe the host-device data transfers such as clEnqueueWriteBuffer and clEnqueueReadBuffer. As with cIMPI, these extensions do not require tricky programming techniques to achieve efficient data transfers, because they hide the optimized implementations behind the MPI function calls. In addition, CUDA 5.0 on NVIDIA’s Kepler-generation devices will also allow MPI functions to take device buffers as parameters. This feature is called NVIDIA’s GPUDirect RDMA[10]. If a CUDA 5.0 compatible InfiniBand card is present, the MPI communication can happen without transferring any device data to host memory, though such devices are not available at this time.

In GPU-aware MPI extensions, all inter-node communications are still managed by the host thread visible to application developers. For example, if the data obtained by executing a kernel are needed by other nodes, the host thread needs to wait for the kernel execution completion in order to serialize the kernel execution and the MPI communication; the host thread is blocked until the kernel execution is completed. On the other hand, cIMPI can describe the dependency among MPI and OpenCL operations, and therefore the host thread does not need to wait for the completion of
requirements as well as inter-node communication. Needed by all nodes, which increases device memory size only a single memory space; they cannot have different this approach, multiple devices sharing one context can have while conforming the OpenCL specifications. However, in nodes, and thereby attain data sharing among remote devices creating a context shared by multiple devices across different same context. Some OpenCL implementations[12] support object shared by multiple devices, the new memory content synchronization points. Once a device updates a memory object visible to the devices are the same only at their model implicitly ensures that the contents of a memory in the same context. The OpenCL memory consistency in the preceding commands to serialize the operations.

Furthermore, MPI extension to OpenCL is not straightforward as Aji et al. discussed in [6]. To keep OpenCL data transfers transparent to MPI application programs, the MPI implementation must acquire valid command queues in some way. Aji et al. assume that an MPI process mostly uses only one command queue and its handle is thus cached by the MPI implementation to be used in subsequent communications, even though this assumption could be incorrect. Even if the cached command queue is available for subsequent communications, there may exist a more appropriate command queue for the communications. On the other hand, clMPI allows application programmers to specify the best command queue for communication. It should be emphasized that GPU-aware MPI extensions and clMPI are mutually-beneficial rather than conflicting. For example, although this work has implemented some custom MPI functions for pipelined data transfers, it is possible for clMPI to use MPI extensions for its implementation.

Stuart et al. have proposed DCGN[11]. As with clMPI, DCGN provides an illusion that GPUs communicate without any help of their hosts. Unlike clMPI, DCGN provides inter-node communication API functions that are called from GPU kernels. When the API is called by a kernel running on a GPU, the kernel sets regions of device memory that are monitored by a CPU thread. Then, the CPU thread reads necessary data from the device memory and thus handles the communication requests from the GPU. Accordingly, DCGN allows a kernel to initiate inter-node communication. However, the approach of monitoring the device memory needs a non-negligible runtime overhead. On the other hand, in clMPI, inter-node communication requests are represented as OpenCL commands, and hence the clMPI implementation can rely on the OpenCL event management mechanism to know the requests.

An OpenCL memory object is shared by multiple devices in the same context. The OpenCL memory consistency model implicitly ensures that the contents of a memory object visible to the devices are the same only at their synchronization points. Once a device updates a memory object shared by multiple devices, the new memory content is implicitly copied to the memory of every device in the same context. Some OpenCL implementations[12] support creating a context shared by multiple devices across different nodes, and thereby attain data sharing among remote devices while conforming the OpenCL specifications. However, in this approach, multiple devices sharing one context can have only a single memory space; they cannot have different memory contents, even though some of the contents are not needed by all nodes, which increases device memory size requirements as well as inter-node communication.

### III. Joint Programming of MPI and OpenCL

This section discusses some difficulties in joint programming of MPI and OpenCL. Figure 1 shows a simple code of the joint programming. In this code, a command to execute a kernel is first enqueued by invoking `clEnqueueNDRangeKernel`. Another command to read the kernel execution result is then enqueued by `clEnqueueReadBuffer`. Using the event object of the first command, `evt`, the execution of the second command is blocked until the first command is completed. The second command enqueued by `clEnqueueReadBuffer` can be either blocking or non-blocking. The function call is non-blocking if the third argument is `CL_FALSE`, otherwise it is blocking. If it is non-blocking, we have to use a synchronization function such as `clFinish` to make sure that the data have already been transferred from device memory to host memory in advance of calling `MPI_Sendrecv`. In this naive implementation, the data exchange with other nodes must be performed after the data transfer from device memory to host memory; those data transfers must be serialized. Similarly, `MPI_Sendrecv` and `clEnqueueWriteBuffer` must be serialized. Therefore, kernel execution and all data transfers are serialized, which results in a long communication time exposed to the total execution time. In addition, the host thread is blocked whenever MPI and OpenCL operations are serialized.

To make matters worse, there is no standard way for the joint programming of MPI and OpenCL. Even for simple point-to-point communication between two remote devices, we can consider at least the following three implementations. One is the naive implementation as shown in Figure 1. In the implementation, host memory buffers should be page-locked (pinned) for efficient data transfers\(^1\). Another implementation is to map device memory objects to host memory addresses by using `clEnqueueMapBuffer`, and then to invoke MPI functions to transfer data from/to the addresses. After the MPI communication, `clEnqueueUnmapMemObject` is invoked to unmap the device memory objects. The other implementation is to overlap host-device data transfers with inter-node data transfers. In this implementation, data of a device memory object are divided into data blocks of a fixed size, called a pipeline block size, and host-device data transfers of each block are overlapped with inter-node data transfers of other blocks in a pipelining fashion [7]. In this paper, the three aforementioned implementations are referred to as pinned, mapped, and pipelined data transfers. Among those implementations, the best one changes depending on several factors such as the message size, device types, device

\(^1\)Although the OpenCL standard does not provide any specific means to allocate pinned host memory buffers, most vendors rely on the usage of `clEnqueueMapBuffer` to provide programmers with pinned host memory buffers.
vendors, and device generations. Accordingly, an application developer might need to implement multiple versions to optimize data transfers for performance portability of an application program across various systems.

Another common approach to hide the communication overhead is to overlap the communication and computation [13]. To this end, the computation is usually divided into two stages. While executing the first stage computation, the first stage communication is performed to prepare for the second stage computation. If the computation and communication are inside a loop, the data for the first stage computation of the next iteration are transferred during executing the second stage computation of the current iteration.

In OpenCL programming, this overlapping optimization can be achieved using two in-order execution command queues. Figure 2 shows a simplified version of the Himeno benchmark code described in [13], which is originally written in CUDA and MPI. In the code, Jacobi_kernel_* functions invoke kernels using the command queue of the first argument to update the memory object of the second argument. The code assumes one-dimensional domain decomposition, in which each decomposed domain is further halved into upper and lower portions, A and B. Figure 3 illustrates the domain decomposition assumed by the code. The top plane of A and the bottom plane of B are halo regions that have to be updated every iteration by exchanging data with neighboring nodes. Hence, if the MPI rank of a process is an even number, during calculating A, the process updates the halo region included in B. Then, it calculates B during exchanging data for updating the halo of A. On the other hand, if the MPI rank of a process is an odd number, the process first calculates B during updating the halo of A. Then, it calculates A during exchanging data for updating the halo of B. As a result, the communication time is not exposed to the total execution time as shown in Figure 4(a) unless the computation time exceeds the communication time.

As the number of MPI processes increases, the computation time becomes shorter because the domain processed by each GPU becomes smaller. However, the second stage communication cannot start even if the first stage computation is completed earlier and hence the data are ready for the second stage communication as shown in Figure 4(b). This is because the host thread is often blocked and tied up in the first stage communication in order to serialize the MPI and OpenCL operations.

Since the code in Figure 2 is simple, there are some workaround techniques to solve this problem. However, in the case where more advanced optimization techniques such as the pipelining are applied to the data transfers, the host thread is stalled more frequently to timely synchronize MPI and OpenCL operations in multiple parallel activities of an application. In general, there are at least three parallel activities in an application: host computation,
device computation, and non-blocking MPI communication. If there are dependent operations of MPI and OpenCL, the host thread is usually blocked to serialize the operations, which inhibits overlapping the parallel activities. Multi-thread programming would be required to properly manage the parallel activities. In this way, an application code becomes more complicated and system-specific, resulting in low code readability, maintainability, and portability. This motivates us to design a bridging programming model that can explicitly describe the dependencies among MPI and OpenCL operations in order to initiate data transfers without any help of the host thread.

IV. AN OPENCL EXTENSION FOR INTEROPERATION WITH MPI

This paper proposes clMPI, an OpenCL extension to facilitate and standardize the joint programming of OpenCL and MPI. The key idea of this extension is to use OpenCL commands for inter-node data transfers as well as data transfers between hosts and local devices. In programming with the clMPI extension, for inter-node data transfers from/to device memory buffers, each MPI process simply sends inter-node communication commands to devices via command queues as well as the other OpenCL commands. Then, the communication commands are executed based on the OpenCL’s execution model.

The major advantages of clMPI are summarized as follows.

1) Performance portability: the implementation details of inter-node data transfers are hidden behind inter-node communication commands, and can be used via simple programming interface similar to the standard OpenCL interface.

2) Event management: a host thread is not responsible for serializing inter-node communication and host-device communication. Instead, an event object is used to block the subsequent command until the preceding command is completed.

3) Interoperability: clMPI can interoperate with MPI. The former is used for device memory data, and the latter is for host memory data.

A. Inter-node communication commands

The clMPI extension offers clEnqueueSendBuffer and clEnqueueRecvBuffer, which enqueue commands of transferring data from and to a device memory buffer, respectively. For example, the function signature of clEnqueueRecvBuffer is as follows.

```
c_int clEnqueueRecvBuffer(cl_command_queue cmd, cl_mem buf, cl_bool blocking, size_t offset, size_t size, int src, /* rank */ int tag, MPI_Comm comm, cl_uint numevts, const cl_event * wlist, cl_event * evtret )
```

When one MPI process invokes those functions for sending a command to a device, the device becomes a communicator device for one MPI communication and works as if it communicates instead of the host thread. The data sent to
the MPI rank are received by the communicator device, and the received data are stored in the memory space of the communicator device, i.e. `buf`. The MPI rank of the sender is given to the function, and the sender could be either the host thread or the communicator device associated with the MPI rank.

In the case where both of the sender and receiver submit inter-node communication commands to their devices, those devices communicate with each other. Figure 5 shows a simple example of communication between remote devices. In this code, the communicator device of rank 0 sends the data of a memory buffer object to the communicator device of rank 1 without explicitly calling any MPI functions. Accordingly, devices appear to communicate with remote devices without help of their host threads. The implementation details of inter-node communication by combining MPI and OpenCL are hidden behind the OpenCL command execution. Hence, the application can use optimized implementations of efficient data transfers without using tricky programming techniques. If one MPI process needs to use multiple communicator devices, a unique tag is given to each device managed by the MPI process of rank 1. A special argument of `MPI_Irecv` that the MPI process of rank 0 receives data from a remote device is supposed to be a communicator device and the data are in the device memory. If `MPI_CL_MEM` is given, the sender and receiver collaborate for efficient data transfers between host and device memories. A similar approach of using `MPI_Datatype` can be seen in [6], even though they extend only MPI but not OpenCL.

**B. Event management**

The clMPI extension allows a programmer to use event objects in order to express the dependency among inter-node communication commands and other OpenCL commands. If an inter-node communication command needs the result of its preceding command, the programmer can get the event object of the first stage communication, and use it to block the execution of the inter-node communication command. This ensures that the communication is performed after the preceding command is completed. In this way, inter-node communication commands are incorporated into the OpenCL execution model in a natural manner. Accordingly, MPI function calls are encapsulated in OpenCL commands whose dependencies with other OpenCL commands are accurately enforced by the command queues. Unlike the conventional joint programming of MPI and OpenCL, the host thread does not need to wait for the preceding command completion. After enqueuing the commands by non-blocking function calls, the host thread immediately becomes available for other computations and communications; an application programmer can consider as if a device is able to work independently from the host thread. In due time, the OpenCL runtime will release the clMPI command for timely execution of the MPI functions as shown in Figure 4(c), even though the two communications may or may not be performed concurrently.

Using the clMPI extension, the code in Figure 2 can be simply rewritten as the code in Figure 6. Since there are dependencies among the enqueued commands, they are expressed by using event objects bound with the commands. In Figure 2, the second stage computation, `jacobi_even_A` and `jacobi_odd_B` are blocked using event objects of the first communication, `e[1]`. The second stage communications are blocked using the event object of the first stage computation, `e[0]`. On the other hand, in Figure 6, all the function calls could be non-blocking, and the host thread is freed from controlling the computation and communication. Therefore, the host thread is just waiting at the end of the iteration by calling `clFinish`.

**C. MPI interoperability**

In clMPI, an MPI process uses clMPI commands for transferring data from/to a device memory buffer. If an MPI process need to transfer data from/to a host memory buffer, clMPI allows the MPI process to use standard MPI functions such as `MPI_Isend` and `MPI_Irecv` to communicate with remote devices as well as remote hosts. Figure 7 shows that the MPI process of rank 0 receives data from a remote device managed by the MPI process of rank 1. A special `MPI_Datatype` value, `MPI_CL_MEM`, is given to the third argument of `MPI_Irecv` in order to express that the sender is supposed to be a communicator device and the data are in the device memory. If `MPI_CL_MEM` is given, the sender and receiver collaborate for efficient data transfers between host and device memories. A similar approach of using `MPI_Datatype` can be seen in [6], even though they extend only MPI but not OpenCL.
cl_context ctx;
MPI_Request req;
cl_event evt[2];

if( rank == 0 ){
    /* receiving data from a remote device */
    MPI_Irecv(recvbuf, bufsz, MPI_CL_MEM, 1, 0,
               MPI_COMM_WORLD,&req);
    /* creating an event object of MPI_Irecv */
    evt[0] = clCreateEventFromMPIRequest(ctx,&req,NULL);
    /* executing a kernel during the data transfer */
    clEnqueueNDRangeKernel(...., &evt[1]);
    /* executing this after the completion of communication */
    clEnqueueWriteBuffer(cmd, buf, ..., 2, evt, NULL);
}
else if(rank==1){
    /* send data to a remote host */
    clEnqueueSendBuffer(cmd, buf, CL_TRUE, 0, bufsz, 0,...);
}

Figure 7. A code with the OpenCL extension for host-to-device communication

As shown in Figure 7, non-blocking MPI functions can be used for inter-node communication from/to a host memory buffer. Hence, the data need to be received before clEnqueueWriteBuffer is executed to write the data to the device memory of rank 0. In addition, a kernel is executed during the inter-node communication. To express the dependency among non-blocking MPI function calls and OpenCL commands, the clMPI extension offers a function to create an OpenCL event object that is corresponding to MPI_Request of a non-blocking MPI function call. Using the event object, another OpenCL command can be executed after the non-blocking MPI function is completed; MPI and OpenCL operations are serialized without any help of the host thread. In Figure 7, the event object is used to ensure that MPI_Irecv is completed before writing data to a device memory buffer.

The MPI interoperability is very important because many applications have already been developed in such a way that CPUs manage all inter-node communications via MPI function calls. Considering the importance, the clMPI extension is not designed as a stand-alone communication library but an OpenCL extension for interoperation with MPI. With the interoperability, legacy applications can be ported incrementally to heterogeneous computing systems by gradually replacing the MPI function calls with the clMPI extension. This does not mean that all inter-node communications should be replaced with the clMPI extension. We argue that both MPI and OpenCL need to be extended for their efficient interoperation.

Although the clMPI extension offers inter-node peer-to-peer communications among remote hosts and devices, it does not currently offer any collective communications. This is because the function calls of MPI collective communications are blocking and no OpenCL extension is required to describe the dependability among the collective communications and OpenCL commands. If optimized collective communications for device memory objects are required, we can hide the implementation details in MPI collective communication functions, rather than developing a set of special collective communication functions for device memory objects. As the MPI-3.0 standard will support non-blocking collective communications, some synchronization mechanisms between the non-blocking collective communications and OpenCL commands might be required in the future. In this case, it will be effective to further extend OpenCL to use its event management mechanism for the synchronization.

V. Evaluation and Discussions

In this section, the performance impact of the proposed extension is evaluated in terms of the sustained bandwidth of point-to-point communication and the sustained performances of application programs: the Himeno benchmark and the nanopowder growth simulation.

Two systems called Cichlid and RICC are used for the following evaluation. Cichlid is a small PC cluster system of four nodes, each of which contains one Intel Core i7 930 CPU running at 2.8 GHz and one NVIDIA Tesla C2070 GPU. The nodes are connected via the Gigabit Ethernet network. On the other hand, in the multi-purpose PC cluster of Riken Integrated Cluster of Clusters (RICC), 100 compute nodes are connected via InfiniBand DDR network. Each of the compute nodes has two Intel Xeon 5570 CPUs and one NVIDIA Tesla C1060 GPU. The system specifications are summarized in Table I.

A. Implementation

In this work, we have implemented the clMPI extension on top of NVIDIA’s OpenCL and Open MPI[14] as shown in Table I. As most of currently available OpenCL implementations are proprietary, the clMPI extension is designed so that it can be implemented on top of a proprietary OpenCL implementation. User event objects are internally used to create event objects of inter-node communication commands. Since there are several different behaviors between a user event object and a command’s event object, the runtime of the clMPI extension has been developed so that user event objects of inter-node communication commands can mimic

### Table I. System Specifications

<table>
<thead>
<tr>
<th>System</th>
<th>Cichlid</th>
<th>RICC</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Xeon E5620</td>
<td>Intel Xeon E5640</td>
</tr>
<tr>
<td>GPU</td>
<td>NVIDIA Tesla C2070</td>
<td>NVIDIA Tesla C1060</td>
</tr>
<tr>
<td>NIC</td>
<td>Gigabit Ethernet</td>
<td>InfiniBand DDR</td>
</tr>
<tr>
<td>OS</td>
<td>CentOS 6.5</td>
<td>RHEL 5.3</td>
</tr>
<tr>
<td>Compiler</td>
<td>GCC-4.8.4</td>
<td>Intel Compiler 11.1</td>
</tr>
<tr>
<td>Driver Ver.</td>
<td>290.10</td>
<td>295.41</td>
</tr>
<tr>
<td>OpenCL</td>
<td>OpenCL 1.1 (CUDA 4.1.1)</td>
<td>OpenCL 1.1 (CUDA 4.2.9)</td>
</tr>
<tr>
<td>MPI</td>
<td>Open MPI 1.6.0</td>
<td>Open MPI 1.6.1</td>
</tr>
</tbody>
</table>
event objects of standard OpenCL commands. Therefore, application programmers can use the event object of an inter-node communication command in the same way as that of a standard OpenCL command.

In our current implementation, the clMPI runtime internally spawns a thread for inter-node communication. Since existing OpenCL implementations are already spawning a CPU thread to support callbacks, the same thread can also be used to handle clMPI function calls. Thus, no additional thread would be needed if clMPI is implemented by OpenCL vendors.

As the clMPI implementation needs to call MPI functions from the host thread and the dedicated thread, the underlying MPI implementation is assumed to be thread-safe. More specifically, MPI_Init_thread should work with MPI_THREAD_MULTIPLE. To make Open MPI work correctly for InfiniBand in a multi-threaded environment, IP over InfiniBand (IPoIB) is used for performance evaluation on RICC.

In our current implementation, wrapper functions of some major MPI functions such as MPI_Send and MPI_Recv are developed so that those functions can perform pipelined data transfers of overlapping host-device communication with inter-node communication when MPI_CL_MEM is given as the MPI_Datatype parameter.

B. Point-to-point Communication Performance

One advantage of the clMPI extension over conventional joint programming of MPI and OpenCL is that the clMPI extension can hide the implementation details of system-aware optimization for efficient data transfers.

Figure 8 shows the difference in sustained bandwidth among pinned, mapped, and pipelined implementations described in Section III. In the figure, “pipelined(N)” indicates the results of pipelined data transfers with the pipeline buffer size of $N$ Mbytes. The evaluation results in Figure 8(a) show that the performance difference among the three implementations is small in the Cichlid system. This is because their sustained bandwidths are limited by the bandwidth of the GbE interconnect network. The time for host-device communication is much shorter than that of inter-node communication, and hence the pipelined implementation hardly improves the sustained bandwidth. On the other hand, in Figure 8(b), there is a big difference in sustained bandwidth among the three implementations. Moreover, the sustained bandwidth of the pipelined implementation changes with the pipeline buffer size. Pipelining with a relatively-small pipeline buffer is the most efficient when the message size is small because the pipeline buffer size needs to be smaller than the message size. On the other hand, a large pipeline buffer leads to a higher sustained bandwidth for large messages because the sustained bandwidth of sending each pipeline buffer usually increases with the pipeline buffer size. Accordingly, the optimal pipeline buffer size changes depending at least on the message size.

From the above results, it is obvious that system-aware optimizations are often required by multi-node GPU applications to achieve a high performance, and hence some abstractions of inter-node data transfers are necessary for high performance portability. For example, on RICC, the piped data transfer is always faster than the mapped one, while the mapped data transfer is faster for small messages on Cichlid due to the short latency of the implementation. The clMPI extension provides interfaces that abstract inter-node data transfers, and thereby allows an application programmer to use optimized data transfers without tricky programming techniques. An automatic selection mechanism of the data transfer implementations can be adopted behind the interfaces. The current implementation of the clMPI runtime can use either the pinned or mapped data transfer for small messages, and the pipelined data transfer can be performed for large messages. The pipelined data transfer can also be implemented using either the pinned or mapped data transfer. In the following evaluation, the mapper and pinned data transfers are used for Cichlid and RICC, respectively. Of course, other optimized data transfers can be incorporated into the runtime and available to application programs without changing their codes, which results in high performance portability across system types, scales, and probably generations.

C. Evaluation with the Himeno benchmark

The performance impact of using the clMPI extension is first evaluated by comparing the sustained performances of three implementations for the Himeno benchmark. One implementation is called the hand-optimized implementation presented in [13]. The hand-optimized implementation uses piped data transfers for exchanging halo data of about 750 KBytes. Another is called the serial implementation that is almost the same as the hand-optimized implementation but all the computations and communications are serialized. The performance of the serial implementation is supposed to be the lowest. The other is the implementation using the clMPI extension, called the clMPI implementation.

Figure 9 shows the sustained performances of the three implementations for the Himeno benchmark with M-size data. Since the hand-optimized implementation is well designed for overlapping the computations and communications, it can always achieve a higher performance than the serial implementation. The performance of the clMPI implementation is almost always comparable to that of the hand-optimized implementation because the communication times of both the hand-optimized and clMPI implementations are not exposed to their total execution times. Accordingly, the clMPI extension allows an application programmer to easily overlap the communication and computation by simply sending inter-node communication commands to devices while
considering the dependencies among OpenCL commands.

The results in Figure 9(a) are obtained using Cichlid whose network performance is low compared to the computation performance. The ratio of the computation time to the communication time in the serial implementation is also shown in the figure. Only in the case of Cichlid with four nodes, the ratio of the computation to the communication time cannot completely be overlapped with the computation time when pinned data transfers are used for communication. In this case, the performance of the hand-optimized implementation is clearly lower than the clMPI implementation. The main reason of the performance difference is that the mapped data transfer behind the clMPI implementation is faster than the pinned data transfers. These results clearly show the importance of system-dependent optimizations for highly-efficient data transfers. As the programming model of the clMPI extension encapsulates the data transfers, an application programmer does not need to know the implementation details, and can automatically use the optimized implementation from a simply-written code such as in Figure 6.

D. Evaluation with a Practical Application

The performance impact of the clMPI extension is further discussed by taking the nanopowder growth simulation [15] as an example of real applications. The simulation code has been developed for numerical analysis of the entire growth process of binary alloy nanopowders in thermal plasma synthesis. Although various phenomena are considered to simulate the nanopowder growth process, about 90% of the total execution time of the original code is spent for simulating the process of coagulation among nanoparticles.

In the following evaluation, the clMPI extension is applied to a parallel version of the simulation code, in which only the coagulation routine is parallelized using MPI, and its kernel loop is further accelerated using OpenCL. The other phenomena such as nucleation and condensation are computed by one host thread, and the coefficient data of about 42 Mbytes required by the coagulation routine are distributed from the host thread to each node at every simulation step. For the simulation code, two versions have been implemented to clarify the effect of using the optimized data transfers provided by the clMPI extension. One is the baseline implementation that just uses MPI_Isend and MPI_Recv for coefficient data distribution. The other is
the clMPI implementation, which uses MPI_Isend with
MPI_CL_MEM to send the coefficients in host memory buffers and clEnqueueRecvBuffer to receive them.

Figure 10 shows the results to compare the performances of the two implementations on RICC. Unlike the Himeno benchmark, the communication overheads are obviously exposed to the total execution time of this simulation program. Due to the decomposition method for MPI parallelization, the number of nodes must be a divisor of 40. Since the poor parallelism, the performance degrades when the number of nodes is 8.

As shown in the figure, the clMPI outperforms the baseline implementation because it can exploit an optimized implementation that overlaps the host-device communication with the inter-node communication in a pipelined fashion for sufficiently large messages. Accordingly, these results indicate that a higher performance can be achieved by appropriately interoperating MPI and OpenCL, and the clMPI enables us to express the interoperation in a simple and effective way.

In the above evaluation, by just replacing the combination of MPI_Recv and clEnqueueWriteBuffer with clEnqueueRecvBuffer, the pipeline data transfer is used for the communication and leads to a higher sustained bandwidth. Hence, the results also suggest that application programmers can incrementally improve their MPI programs so as to use the clMPI extension. This is very important because most of existing applications have been developed using MPI.

VI. CONCLUSIONS

This paper has proposed an OpenCL extension, clMPI, to improve the interoperability with MPI. In the clMPI extension, additional OpenCL commands are defined for encapsulating common programming patterns in inter-node communication involving compute devices. The additional commands are executed in the same way as the other OpenCL commands. Using OpenCL event objects, we can express the dependency among both conventional and additional commands. Therefore, inter-node communications indicated by the additional commands are incorporated into the OpenCL execution model in a natural manner.

As inter-node communications are abstracted as OpenCL commands, the implementation details of data transfers are hidden from application codes. Hence, clMPI will be able to exploit new features of the latest devices without any code change. As a result, clMPI would allow today’s applications to benefit from hardware improvements without making any code change, or even without recompiling the application. That is, clMPI can improve not only the performance but also the performance portabilities across system types, scales, and generations.

The performance evaluation results clearly show that clMPI can achieve efficient data transfers while hiding the complicated implementation details, resulting in higher performance and scalability. Moreover, using the clMPI extension, the host thread of an application is not blocked to serialize dependent operations of MPI and OpenCL. As a result, the clMPI extension allows an application programmer to easily use the opportunities to overlap communications with computations.

Although this work focuses on interoperation of MPI and OpenCL, we believe that the idea itself could be applicable to other programming models such as CUDA. Moreover, not only MPI peer-to-peer communications but also other time-consuming tasks such as file I/O would be encapsulated in other additional OpenCL commands. In the future, we will also need to consider synchronization between OpenCL commands and non-blocking collective communications. These extensions will be discussed in our future work.

ACKNOWLEDGMENTS

The authors would like to thank Prof. Mayasa Shigeta of Tohoku University for allowing us to use his simulation code in the performance evaluation. The authors would also like to thank the RIKEN Integrated Cluster of Clusters (RICC) at RIKEN for the user supports and the computer resources used for the performance evaluation.

This research was partially supported by JST CREST “An Evolutionary Approach to Construction of a Software Development Environment for Massively-Parallel Heterogeneous Systems” and Grant-in-Aid for Young Scientists(B) #23700028.

REFERENCES


