Hindawi Publishing Corporation
Scientific Programming
Volume 2015, Article ID 576498, 16 pages
http://dx.doi.org/10.1155/2015/576498

Research Article

Optimized Data Transfers Based on the OpenCL Event Management Mechanism

Hiroyuki Takizawa,[1] Shoichi Hirasawa,[1] Makoto Sugawara,[2] Isaac Gelado,[3] Hiroaki Kobayashi,[2] and Wen-mei W. Hwu[4]

[1]*Tohoku University/JST CREST, Sendai, Miyagi 980-8579, Japan*
[2]*Tohoku University, Sendai, Miyagi 980-8578, Japan*
[3]*NVIDIA Research, Santa Clara, CA 95050, USA*
[4]*The University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA*

Correspondence should be addressed to Hiroyuki Takizawa; takizawa@cc.tohoku.ac.jp

Received 15 May 2014; Accepted 29 September 2014

Academic Editor: Sunita Chandrasekaran

In standard OpenCL programming, hosts are supposed to control their compute devices. Since compute devices are dedicated to kernel computation, only hosts can execute several kinds of data transfers such as internode communication and file access. These data transfers require one host to simultaneously play two or more roles due to the need for collaboration between the host and devices. The codes for such data transfers are likely to be system-specific, resulting in low portability. This paper proposes an OpenCL extension that incorporates such data transfers into the OpenCL event management mechanism. Unlike the current OpenCL standard, the main thread running on the host is not blocked to serialize dependent operations. Hence, an application can easily use the opportunities to overlap parallel activities of hosts and compute devices. 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. The evaluation results show that the proposed extension can use the optimized data transfer implementation and thereby increase the sustained data transfer performance by about 18% for a real application accessing a big data file.

1. Introduction

Today, many high-performance computing (HPC) systems are equipped with graphics processing units (GPUs) serving 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. For the rest of the paper, we will follow the OpenCL terminology and refer to the CPUs as *hosts* and data-parallel accelerators as *compute devices*.

One difficulty in programming such a heterogeneous system is that a programmer has to take the responsibility for appointing the right processors to the right tasks. In the current OpenCL standard, only the host can perform some of tasks because the compute device is dedicated to kernel computation. For example, only the host can access files and communicate with other nodes. To write the computation results of a kernel into a file, the results have to be first transferred from the device memory to the host memory after the kernel execution, and then the host writes the results to the file.

From the viewpoint of programmers, accelerator programming models such as CUDA [1] and OpenCL [2] are used for data transfers between the device memory and the host memory, MPI [3] is used for internode data communication, and file functions of each programming language, such as `fprintf` and `fscanf` in the C programming, are used for the file I/O. Hence, these three categories of data transfers are described with different programming models. Some data transfers done by different programming models could be dependent; a certain data transfer can be done

only after its preceding data transfer. *In order to enforce such dependence*, one popular way is to block the host thread until the preceding data transfer has finished. This kind of blocking often inhibits overlapping parallel activities of the host and the device and exposes the data transfer latencies to the total execution time. One may create a dedicated host thread for synchronizing the dependent data transfers. However, such multithreading will further increase the programming complexity. Consequently, the application performance strongly depends on the programming skills and craftsmanship of the developers.

Another difficulty is that there is no standard way to coding those data transfers even for common data transfer patterns. Since application programmers are supposed to appropriately combine those data transfers for fully exploiting the potential of a heterogeneous HPC system, the code is often specialized for a particular system. For example, one compute device may be capable of directly accessing a file, and another may not. In this case, the file access code for the former device would be totally different from that for the latter one. Therefore, the code for data transfers is likely to be system-specific and some abstractions are required to achieve functional portability as well as performance portability. Although OpenCL has been designed for programming various compute devices, it provides interfaces only for data transfers between the host memory and the device memory, but not for the other kinds of data transfers.

To overcome the above difficulties, we need a "bridging" programming model that provides a standard way for coding data transfers among various memory spaces and storages of a heterogeneous parallel system in a unified fashion. In this paper, we focus on OpenCL as the accelerator programming model for high code portability and propose an OpenCL extension for abstraction of data transfers, though the idea could be trivially extrapolated to other GPU programming models such as CUDA. The proposed OpenCL extension named *clDataTransfer* provides an illusion that the compute devices are transferring data directly to files or other nodes. This paper focuses especially on internode communication and file access as typical data transfers that need collaboration of hosts and devices. The extension offers some OpenCL commands and functions for the data transfers. The internode communication and file access commands are executed in the same manner as the other OpenCL commands, and hence the OpenCL programming model is naturally extended so as to seamlessly access file data and also to improve the MPI interoperability.

The clDataTransfer extension provides a portable, standardized way to programming of internode communications and file accesses from/to the device memory. Although MPI and file functions are used internally to perform those data transfers with help of the hosts, those internal behaviors are invisible to application programmers; it can thereby hide the system-aware optimized implementations behind function calls. Hence, we can also expect that the clDataTransfer extension improves the performance portability of OpenCL applications across different system types, scales, and generations.

The rest of this paper is organized as follows. Section 2 briefly reviews the related work. Section 3 discusses the difficulties in joint programming of OpenCL, MPI, and the standard I/O package of the C library, so-called *Stdio*. Then, Section 4 proposes clDataTransfer, which is an OpenCL extension for the collaboration with MPI and Stdio. Section 5 discusses the performance impact of clDataTransfer through some evaluation results. Finally, Section 6 gives concluding remarks and our future work.

2. Related Work

In the OpenCL programming model, a CPU works as a *host* that manages one or more *compute devices* such as GPUs. To manage the interaction between the host and devices, 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 transfers data from the device memory to the host memory. The function signature is as in Algorithm 1.

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 internode 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 [4] that provides an MPI-like interface for communication between remote GPUs. MPI-ACC [5] uses the `MPI_Datatype` argument to indicate that the memory buffer passed to an MPI function is located in the device memory. MVAPICH2-GPU [6] 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 depending on whether the memory buffer is in the device memory or the host memory. Stuart et al. have discussed

```
cl_int
clEnqueueReadBuffer( cl_command_queue cmd, /* command queue */
                     cl_mem buf,           /* memory buffer */
                     cl_bool blocking,     / blocking */
                     size_t offset,        /* offset */
                     size_t size,          /* buffer size */
                     void* hbuf,           /* buffer pointer */
                     cl_uint numevts,      /* the number of events in the list */
                     cl_event* wlist,      /* event list */
                     cl_evett* evtret )    /* event object of event object */
```

Algorithm 1

various design options of MPI extension to support accelerators [7]. 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 [8]. Those extensions allow an application to use a GPU memory buffer as the end point of MPI communication; the extended MPI implementations enable using MPI functions for internode 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 clDataTransfer, 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 GPU-aware MPI extensions, all internode 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.

Furthermore, MPI extension to OpenCL is not straightforward, as Aji et al. discussed in [5]. 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. clDataTransfer allows application programmers to specify the best command queue for communication. It should be emphasized that GPU-aware MPI extensions and clDataTransfer are mutually beneficial rather than conflicting. For example, although this work has implemented pipelined data transfers using standard MPI functions, it is possible for clDataTransfer to use MPI extensions for its implementation.

Stuart and Owens have proposed DCGN [9]. As with clDataTransfer, DCGN provides an illusion that GPUs communicate without any help of their hosts. Unlike clDataTransfer, DCGN provides internode 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 internode communication. However, the requirement for host to monitor the device memory incurs a nonnegligible runtime overhead. On the other hand, in clDataTransfer, internode communication requests are represented as OpenCL commands. Hence, the host initiates the commands and the clDataTransfer implementation can rely on the OpenCL event management mechanism to synchronize with the commands.

An OpenCL memory object in the same context is shared by multiple devices. 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 [10] 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 if some of the contents are not needed by all nodes. As a result, the contents could unnecessarily be duplicated to the device memory of every node, increasing the aggregated memory usage and also internode communications for the duplication.

GPU computing is employed not only for conventional HPC applications but also for data-intensive applications, for example, [11, 12], in which the data sizes are large and hence are stored in files. As only hosts can access the data stored in files, GPU computing requires additional data transfers between hosts and GPUs. Nonetheless, GPUs are effective to accelerate the kernel execution and reduce the total execution time in practical data-intensive applications. Overlapping the kernel execution with various data transfers such as file accesses and host-device data transfers is a key technique to reduce the data transfer latencies and obviously has common code patterns. However, as far as we know, there is no standard way to develop this pattern in a manner that is reusable in other applications. As recent and future

```
(1) cl_command_queue cmd;
(2) cl_kernel kern;
(3) cl_event evt;
(4)
(5) for(int i(0);i<N;++i){
(6)    // (1) computation on a device
(7)    clEnqueueNDRangeKernel(cmd,kern,...,0,NULL,&evt);
(8)
(9)    // (2) read the result from device to host
(10)   clEnqueueReadBuffer(cmd,...,1,&evt,NULL);
(11)   clFinish(cmd); // the host thread is blocked
(12)
(13)   // (3) exchange data with other nodes
(14)   MPI_Sendrecv(...); // blocking function call
(15)
(16)   // (4) write the received data to device memory
(17)   clEnqueueWriteBuffer(cmd,...);
(18) }
```

Listing 1: A simple pseudocode combining OpenCL and MPI.

HPC systems have hierarchical storage subsystems, high-speed local storages using nonvolatile memories will be available. In those cases, the overlapping would become more significant because host-device data transfer overheads increase relatively to the file access overhead.

3. Difficulties in Joint Programming

This section discusses some difficulties in joint programming of OpenCL and other libraries, such as MPI, which are called by host threads. Listing 1 shows a simple code of the joint programming of MPI and OpenCL. 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 nonblocking. The function call is nonblocking if the third argument is `CL_FALSE`; otherwise it is blocking. If it is nonblocking, 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. Although Listing 1 shows an example of joint programming of MPI and OpenCL, the same difficulties arise when combining OpenCL and Stdio (or any other file access programming interfaces).

To make matters worse, there is no standard way for the joint programming. 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 Listing 1. In the implementation, host memory buffers should be page-locked (pinned) for efficient data transfers (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). This can be also a point to make different vendors require different implementations to exploit pinned memory. 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 internode 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 internode data transfers of other blocks in a pipelining fashion [6]. 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 vendors, and device generations. Also in the cases of overlapping host-device data transfers with file accesses there are many implementation options and parameters due to the variety of file access speeds in a hierarchical storage subsystem. Accordingly, an application developer might need to implement multiple versions to optimize data transfers

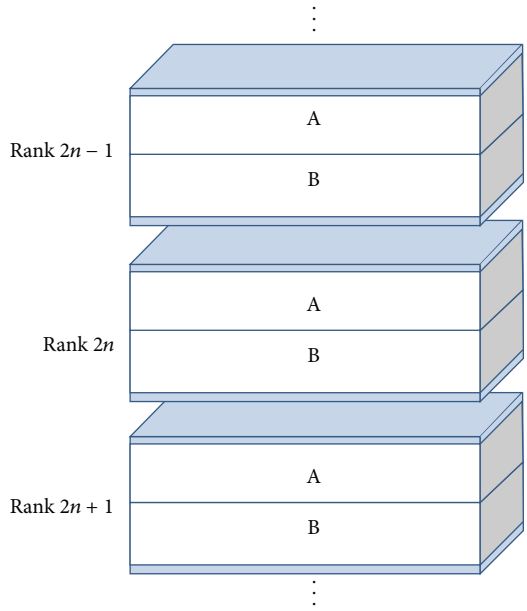


Figure 1: One-dimensional domain decomposition.

for performance portability of an application program across various systems.

Another common approach to hide the communication overhead is to overlap the data transfers and computation through double buffering [11, 13]. To this end, the computation is usually divided into two stages. While executing the first stage computation, the first stage data transfer is performed to prepare for the second stage computation. If the computation and data transfer are inside a loop, the second stage data transfer for the first stage computation of the next iteration is performed during the second stage computation of the current iteration.

In OpenCL programming, this overlapping optimization can be achieved using two in-order execution command queues. Listing 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 in Lines (9), (18), (28), and (35) invoke kernels using the command queue `cmd1` to update the memory object specified by 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 1 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 2(a) unless the communication time exceeds the computation 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 2(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 Listing 2 is simple, there are some workaround techniques to solve this problem. However, in the case where more advanced optimization techniques such as 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 nonblocking MPI communication. If there are dependent operations of MPI and OpenCL, the host thread is usually blocked to serialize the operations, which inhibits overlapping of the parallel activities. Also, host thread blocking is often used even in a serial application if the host thread needs to load data from a file, send them to the device memory, and retrieve the computation results from the device memory. Multithread programming or complex asynchronous I/O APIs would be required to properly manage those 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, OpenCL, and file access operations in order to initiate data transfers without any help of the host thread.

4. An OpenCL Extension for Collaboration with MPI and Stdio

This paper proposes *clDataTransfer*, an OpenCL extension to facilitate and standardize the joint programming of MPI, Stdio, and OpenCL. The key idea of this extension is to use OpenCL commands for internode data transfers, file accesses, and data transfers between hosts and local devices.

The major advantages of clDataTransfer are summarized as follows.

(1) Performance portability: the implementation details of internode data transfers and file accesses are hidden behind extended commands and can be used via a simple programming interface similar to the standard OpenCL interface.

(2) Event management: a host thread is not responsible for serializing internode communications, file operations, and host-device communications. Instead, an event object is used to block the subsequent commands until the preceding command is completed.

```
(1) cl_command_queue cmd1, cmd2;
(2) cl_mem p_new, p_old, p_tmp;
(3)
(4) for(int i(0);i<N;++i){
(5)   //swap pointers
(6)   p_tmp = p_new; p_new = p_old; p_old = p_tmp;
(7)   if( rank%2 == 0)  {
(8)     // the upper portion is calculated
(9)     jacobi_kernel_even_A(cmd1,p_new,...);
(10)    // the bottom plane is updated
(11)    MPI_Irecv(...);
(12)    clEnqueueReadBuffer(cmd2,p_old,CL_FALSE,...);
(13)    clFinish(cmd2);  // blocking
(14)    MPI_Send(...);     // blocking
(15)    MPI_Wait(...);     // blocking
(16)    clEnqueueWriteBuffer(cmd2,p_old,CL_FALSE,...);
(17)    // the lower portion is calculated
(18)    jacobi_kernel_even_B(cmd2,p_new,...);
(19)    // the top plane is updated
(20)    MPI_Irecv(...);
(21)    clEnqueueReadBuffer(cmd1,p_new,CL_FALSE,...);
(22)    clFinish(cmd1);  // blocking
(23)    MPI_Send(...);     // blocking
(24)    MPI_Wait(...);     // blocking
(25)    clEnqueueWriteBuffer(cmd1,p_new,CL_FALSE,...);
(26)  }
(27)  else {
(28)    jacobi_kernel_odd_B(cmd1,p_new,...);
(29)    MPI_Irecv(...);
(30)    clEnqueueReadBuffer(cmd2,p_old,CL_FALSE,...);
(31)    clFinish(cmd2);  // blocking
(32)    MPI_Send(...);     // blocking
(33)    MPI_Wait(...);     // blocking
(34)    clEnqueueWriteBuffer(cmd2,p_old,CL_FALSE,...);
(35)    jacobi_kernel_odd_A(cmd2,p_new,...);
(36)    MPI_Irecv(...);
(37)    clEnqueueReadBuffer(cmd1,p_new,CL_FALSE,...);
(38)    clFinish(cmd1);  // blocking
(39)    MPI_Send(...);     // blocking MPI_Wait (...);   // blocking
(40)    clEnqueueWriteBuffer(cmd1,p_new,CL_FALSE,...);
(41)  } clFinish(cmd1);clFinish(cmd2);   /* error calculation */
(42)}
```

Listing 2: A Himeno benchmark code with overlapping communication and computation.

(3) Collaboration for latency hiding: clDataTransfer can collaborate with MPI and Stdio in order to hide data transfer latencies in a pipelining fashion.

By encapsulating file accesses into OpenCL commands, the clDataTransfer extension offers two file access commands: `clEnqueueReadBufferToStdioFile` and `clEnqueueWriteBufferFromStdioFile`. `clEnqueueReadBufferToStdioFile` reads data from a device memory buffer and writes the data to a file, and `clEnqueueWriteBufferFromStdioFile` reads data from a file and writes the data to a device memory buffer. The function signatures are as in Algorithm 2.

Similarly, the clDataTransfer extension offers `clEnqueueSendBuffer` and `clEnqueueRecvBuffer`, which enqueue commands of transferring data from and to a device memory buffer, respectively. These clDataTransfer functions are direct counterparts of `MPI_Send` and `MPI_Recv` [3] and hence take the same arguments of rank, tag, and communicator as those two MPI functions. For example, the function signature of `clEnqueueRecvBuffer` is as in Algorithm 3.

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,

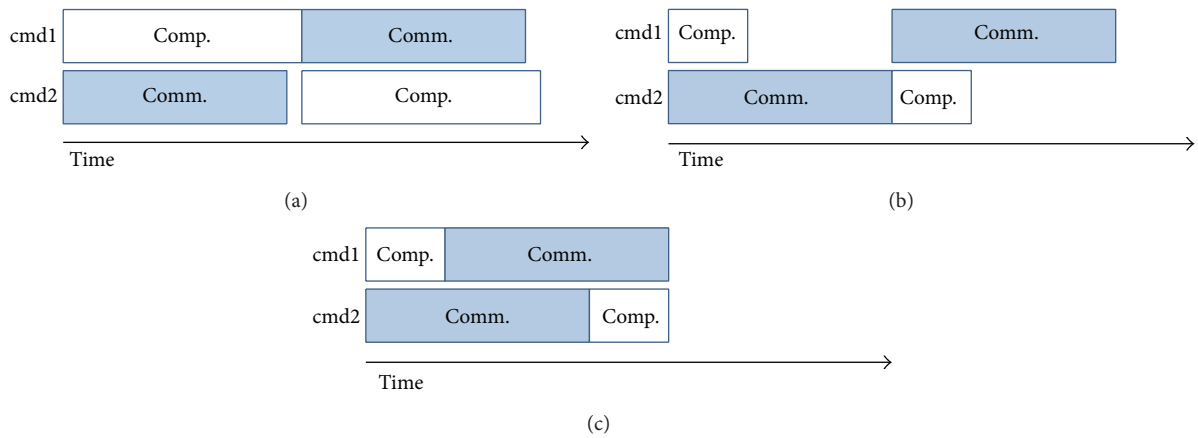


Figure 2: Overlapping communications and computations. (a) The communication time is overlapped with the computation time. (b) The computation time is too short to hide the communication time. Since joint programming of OpenCL and MPI cannot express the dependency between the first communication and the second computation, the host thread is blocked to execute them in a correct order. (c) The second communication can potentially start earlier because the host thread is not blocked.

```
cl_int clEnqueueReadBufferToStdioFile(
                              cl_command_queue cmd, /* command queue */
                              cl_mem mem,   /* memory buffer to be read */
                              cl_bool blk,  /* blocking function call      */
                              size_t off,   /* offset */
                              size_t bsz,   /* buffer size */
                              FILE* fp,     /* file pointer */
                              cl_uint nev,  /* the number of events in the list */
                              const cl_event* evl, /* event list */
                              cl_event* evt) /* event object of the function call */
cl_int clEnqueueWriteBufferFromStdioFile(
                              cl_command_queue cmd, /* command queue */
                              cl_mem mem,   /* memory buffer to be written */
                              cl_bool blk,  /* blocking function call      */
                              size_t off,   /* offset */
                              size_t bsz,   /* buffer size */
                              FILE* fp,     /* file pointer */
                              cl_uint nev,  /* the number of events in the list */
                              const cl_event* evl, /* event list */
                              cl_event* evt) /* event object of the function call */
```

Algorithm 2

```
cl_int
clEnqueueRecvBuffer(cl_command_queue cmd, /* command queue */
                    cl_mem buf, /* memory buffer to receive data */
                    cl_bool blocking, /* blocking function call */
                    size_t offset,  /* offset */
                    size_t size,    /* buffer size */
                    int src,        /* sender's rank */
                    int tag,        /* tag */
                    MPI_Comm comm,  /* communicator */
                    cl_uint numevts, /* the number of events in the list */
                    const cl_event* wlist, /* event list */
                    cl_event* evtret ) /* event object of the function call */
```

Algorithm 3

```
(1) if( rank == 0 ){
(2)   clEnqueueSendBuffer(cmd, buf, CL_TRUE, off, sz, 1,...);
(3) }
(4) else if(rank == 1){
(5)   clEnqueueRecvBuffer(cmd, buf, CL_TRUE, off, sz, 0,...);
(6) }
```

Listing 3: A code with the OpenCL extension for device-to-device communication.

and the received data are stored in the memory space of the communicator device, that is, `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 the sender and the receiver submit internode communication commands to their devices, those devices communicate with each other. Listing 3 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 internode 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 MPI communication to specify which communicator device handles it.

4.1. Event Management. The clDataTransfer extension allows a programmer to use event objects in order to express the dependency among internode communication commands, storage file access commands, and other OpenCL commands. If a data transfer command provided by clDataTransfer needs the result of its preceding command, the programmer can get the event object of the preceding command and use it to block the execution of the data transfer command. This ensures that the data transfer is performed after the preceding command is completed. In this way, data transfer commands of clDataTransfer are incorporated into the OpenCL execution model in a natural manner. Accordingly, function calls of MPI and Stdio 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, Stdio, and OpenCL, the host thread does not need to wait for the preceding command completion. After enqueuing the commands by nonblocking function calls, the host thread immediately becomes available for other computations and data transfers; 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 clDataTransfer command for timely execution of the MPI functions as shown in Figure 2(c), even though the two communications may or may not be performed concurrently.

Using the clDataTransfer extension, the code in Listing 2 can be simply rewritten as the code in Listing 4. This is an example that demonstrates simplification of common patterns in joint programming of OpenCL and other programming models. In this particular case, the clDataTransfer extension can halve the number of code lines for describing the same computation as the joint programming of OpenCL and MPI. Since there are dependencies among the enqueued commands, they are expressed by using event objects bound with the commands. In Listing 2, the second stage computations, `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 Listing 4, the dependencies among the function calls are managed by the OpenCL event management mechanism, and the host thread is thus freed from controlling the computation and communication. In the code, `clEnqueueSendrecvBuffer` enqueues an OpenCL command for exchanging data between two MPI processes by internally invoking `MPI_Sendrecv` under control of the OpenCL event management. Therefore, the host thread is just waiting at the end of the iteration by calling `clFinish`.

4.2. Interoperability with Existing MPI Functions. In clDataTransfer, an MPI process uses clDataTransfer commands for transferring data from/to a device memory buffer. If an MPI process needs to transfer data from/to a host memory buffer, clDataTransfer 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. Listing 5 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 [5], even though they extend only MPI but not OpenCL.

As shown in Listing 5, nonblocking MPI functions can be used for internode communication from/to a host memory buffer. Hence, the data need to be received

```
(1) cl_command_queue cmd1, cmd2;
(2) cl_mem p_new, p_old, p_tmp;
(3) cl_event e[2];
(4)
(5) for(int i(0);i<N;++i){
(6)   p_tmp = p_new; p_new = p_old; p_old = p_tmp;
(7)   if( rank%2 == 0) {
(8)     jacobi_kernel_even_A(cmd1,p_new...0,NULL,&e[0]);
(9)     clEnqueueSendrecvBuffer(cmd2,p_old,...0,NULL,&e[1]);
(10)    jacobi_kernel_even_B(cmd2,p_new...1,&e[1],NULL);
(11)    clEnqueueSendrecvBuffer(cmd1,p_new,...1,&e[0],NULL);
(12)  }
(13)  else {
(14)    jacobi_kernel_odd_B(cmd2,p_new...0,NULL,&e[0]);
(15)    clEnqueueSendrecvBuffer(cmd1,p_old,...0,NULL,&e[1]);
(16)    jacobi_kernel_odd_A(cmd1,p_new...1,&e[1],NULL);
(17)    clEnqueueSendrecvBuffer(cmd2,p_new,...1,&e[0],NULL);
(18)  }
(19)  clFinish(cmd1);clFinish(cmd2);
(20)  /* error calculation */
(21)}
```

Listing 4: A Himeno benchmark code with the proposed OpenCL extension.

```
(1) cl_context ctx;
(2) MPI_Request req;
(3) cl_event evt[2];
(4)
(5) if( rank == 0 ){
(6)   /* receiving data from a remote device */
(7)   MPI_Irecv(recvbuf, bufsz, MPI_CL_MEM, 1, 0, MPI_COMM_WORLD,&req);
(8)   /* creating an event object of MPI_Irecv */
(9)   evt[0] = clCreateEventFromMPIRequest(ctx,&req,NULL);
(10)  /* executing a kernel during the data transfer */
(11)  clEnqueueNDRangeKernel(..., &evt[1]);
(12)
(13)  /* executing this after the computation and communication */
(14)  clEnqueueWriteBuffer(cmd, buf, ..., 2, evt, NULL);
(15)}
(16)else if(rank == 1){
(17)  /* send data to a remote host */
(18)  clEnqueueSendBuffer(cmd, buf, CL_TRUE, 0, bufsz, 0,...);
(19)}
```

Listing 5: A code with the OpenCL extension for host-to-device communication.

before `clEnqueueWriteBuffer` in lines (14) is executed to write the data to the device memory of rank 0. In addition, a kernel in line (11) is executed during the internode communication. To express the dependency among nonblocking MPI function calls and OpenCL commands, the clDataTransfer extension offers a function to create an OpenCL event object that corresponds to `MPI_Request` of a nonblocking MPI function call. Using the event object, another OpenCL command can be executed after the nonblocking MPI function is completed; the dependence between an MPI operation and an OpenCL operation is properly enforced without host intervention. In Listing 5, 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 internode communications via MPI function calls. Considering the importance, the clDataTransfer extension is not designed as a standalone communication library but an OpenCL extension for interoperation with

Table 1: System specifications.

System	Masamune	Cichlid	RICC
CPU	Intel Xeon E5-2670	Intel Core i7 930	Intel Xeon 5570
GPU	GeForce GTX TITAN	Tesla C2070	Tesla C1060
NIC	GbE 1000BASE-T	GbE 1000BASE-T	InfiniBand DDR
OS	CentOS 6.4	CentOS 6.0	RHEL 5.3
Compiler	GCC-4.4.7	GCC-4.4.4	Intel Compiler 11.1
GPU Driver	319.37	290.10	295.41
OpenCL	OpenCL1.1 (CUDA5.5)	OpenCL1.1 (CUDA4.1.1)	OpenCL1.1 (CUDA 4.2.9)
MPI	Open MPI 1.5.4	Open MPI 1.6.0	Open MPI 1.6.1
Storage	SSD (Intel 910 400 GB)	NFS	NFS

MPI. With the interoperability, legacy applications can be ported incrementally to heterogeneous computing systems by gradually replacing the MPI function calls with the clDataTransfer extension. This does not mean that all internode communications should be replaced with the clDataTransfer extension. We argue that both MPI and OpenCL need to be extended for their efficient interoperation.

Although the clDataTransfer extension offers internode 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 nonblocking collective communications, some synchronization mechanisms between the nonblocking 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.

5. Evaluation and Discussions

In this section, the performance impact of the proposed extension is discussed by showing the effects of hiding the host-device data transfer latency and the performance improvement. In this work, a GPU program of the Smith Waterman algorithm [11] is first used to evaluate the performance gain by overlapping host-device data transfers with file accesses. Then, the Himeno benchmark [13] and the nanopowder growth simulation [14] are adopted for the evaluation of MPI interoperability, which is improved by the proposed extension.

Three systems called Masamune, Cichlid, and RICC are used for the following evaluation. Masamune is a single node PC with Intel Xeon E5-2670 CPU running at 2.60 GHz and one NVIDIA GeForce GTX TITAN GPU. 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 multipurpose PC cluster of RIKEN Integrated Cluster of Clusters (RICC), 100 compute nodes are connected via an 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 1.

5.1. Implementation. In this work, we have implemented the clDataTransfer extension on top of NVIDIA's OpenCL and Open MPI [15] as shown in Table 1. As most of currently available OpenCL implementations are proprietary, the clDataTransfer extension is designed so that it can be implemented on top of a proprietary OpenCL implementation. In the implementation, we have to consider at least three points. One point is how to implement clDataTransfer commands that mimic standard OpenCL commands. Another is how to implement nonblocking function calls. The other is how to implement pipelined data transfers.

To implement clDataTransfer commands whose execution is managed by the OpenCL event management system, user event objects are internally used to create event objects of those additional commands provided by the clDataTransfer extension. Since there are several different behaviors between standard event objects and user event objects, the runtime of the clDataTransfer extension has been developed so that user event objects of additional commands can mimic event objects of standard OpenCL commands. A simplified pseudocode of a clDataTransfer function is shown in Listing 6. When the function is executed, from the viewpoint of application programmers, the clDataTransfer runtime appears to work as follows. A user event object whose execution status is `CL_SUBMITTED` is first created when a clDataTransfer command is enqueued. Then, the clDataTransfer runtime automatically changes the execution status to `CL_COMPLETE` when the command is completed. This allows other commands to wait for the completion of a clDataTransfer command by using its user event object. Therefore, application programmers can use the event object of a clDataTransfer command in the same way as that of a standard OpenCL command.

The clDataTransfer function in Listing 6 can be invoked in either blocking or nonblocking mode. To invoke a clDataTransfer function without blocking the host thread, the clDataTransfer runtime internally spawns another thread dedicated to data transfers. Since most existing OpenCL

```
(1) cl_int clDataTransferFunc( ...,
(2)                            cl_uint numevts,       /* the number of events in the list */
(3)                            cl_event* wlist,       /* event list */
(4)                            cl_evett* evtret )     /* event object of event object */
(5) {
(6)   /* create a new user event object whose status is CL_SUBMITTED */
(7)   *evtret = clCreateUserEvent(...);
(8)
(9)   if( non_blocking = CL_TRUE)
(10)    pthread_create(...,cldtThreadFunc,...);
(11)  else
(12)    cldtThreadFunc(...);
(13)
(14)  return CL_SUCCESS;
(15)}
(16)
(17)/* numevt, wlist, and evtret are passed from the caller */
(18)void* cldtThreadFunc(void* p)
(19){
(20)  clWaitForEvent(numevt, wlist);
(21)
(22)  /* pipelined data transfer */
(23)
(24)  clSetUserEventStatus(*evtret, CL_COMPLETE);
(25)  return NULL;
(26)}
```

Listing 6: A simple pseudocode of a clDataTransfer function.

implementations are already spawning a CPU thread to support callbacks, the same thread can technically be used to handle the clDataTransfer function calls. Thus, no additional thread would be needed if clDataTransfer is implemented by OpenCL vendors.

As the clDataTransfer implementation needs to call MPI and file access functions from the host thread and the dedicated thread, their underlying implementations are assumed to be thread-safe. File access functions are generally thread-safe. On the other hand, in MPI, `MPI_Init_thread` should work with `MPI_THREAD_MULTIPLE`. To make Open MPI work correctly for InfiniBand in a multithreaded environment, IP over InfiniBand (IPoIB) is used for performance evaluation on RICC.

In our current implementation, pipelined data transfers are implemented by ourselves by reference to some papers on GPU-aware MPI implementations [5, 6] and encapsulated in clDataTransfer commands as shown in Listing 6. So far, wrapper functions of file I/O functions and some major MPI functions such as `MPI_Send` and `MPI_Recv` have been developed so that those functions can perform pipelined data transfers of overlapping host-device communication with internode communication when `MPI_CL_MEM` is given as the `MPI_Datatype` parameter.

5.2. Evaluation of File Access Performance

5.2.1. Evaluation of Sustained Data Transfer Bandwidths. The sustained bandwidths of data transfers from files to device memory buffers are evaluated to show that `clEnqueueWriteBufferFromStdioFile` can reduce the data transfer time compared to conventional serialized data transfers. To evaluate the sustained bandwidths with different storage's bandwidths, the solid state drive (SSD) and the hard disk drive (HDD) of Masamune are used as the local storages, and a shared file system of NFS is used as the global storage and accessed from Cichlid.

First, we evaluate how much the clDataTransfer extension can improve the sustained bandwidth. In the case of using `clEnqueueWriteBufferFromStdioFile`, data are read from a file and then sent to a device memory buffer. The bandwidth of a storage is lower than that of the data transfer between the host and the device via the PCI-express bus. Hence, the sustained bandwidth of the data transfer is limited by the storage bandwidth. Since `clEnqueueWriteBufferFromSdtioFile` enables the host-device data transfer to be overlapped with the file read, it can reduce the data transfer time and hence achieve a higher sustained bandwidth than the sequential execution of those two data transfers.

Figure 3 shows the sustained bandwidths obtained with changing the data size and the pipeline buffer size. The vertical axis shows the sustained bandwidth, and the horizontal axis is the data size. In the figure, *Serial* means the data transfer time in the case of not hiding the host-device data transfer latency and N-pipe means the data transfer time of the pipelined implementation with an N-byte pipeline buffer. By hiding the latency more, the data transfer time approaches to the file read time, which is *FileRead* in the figure. These

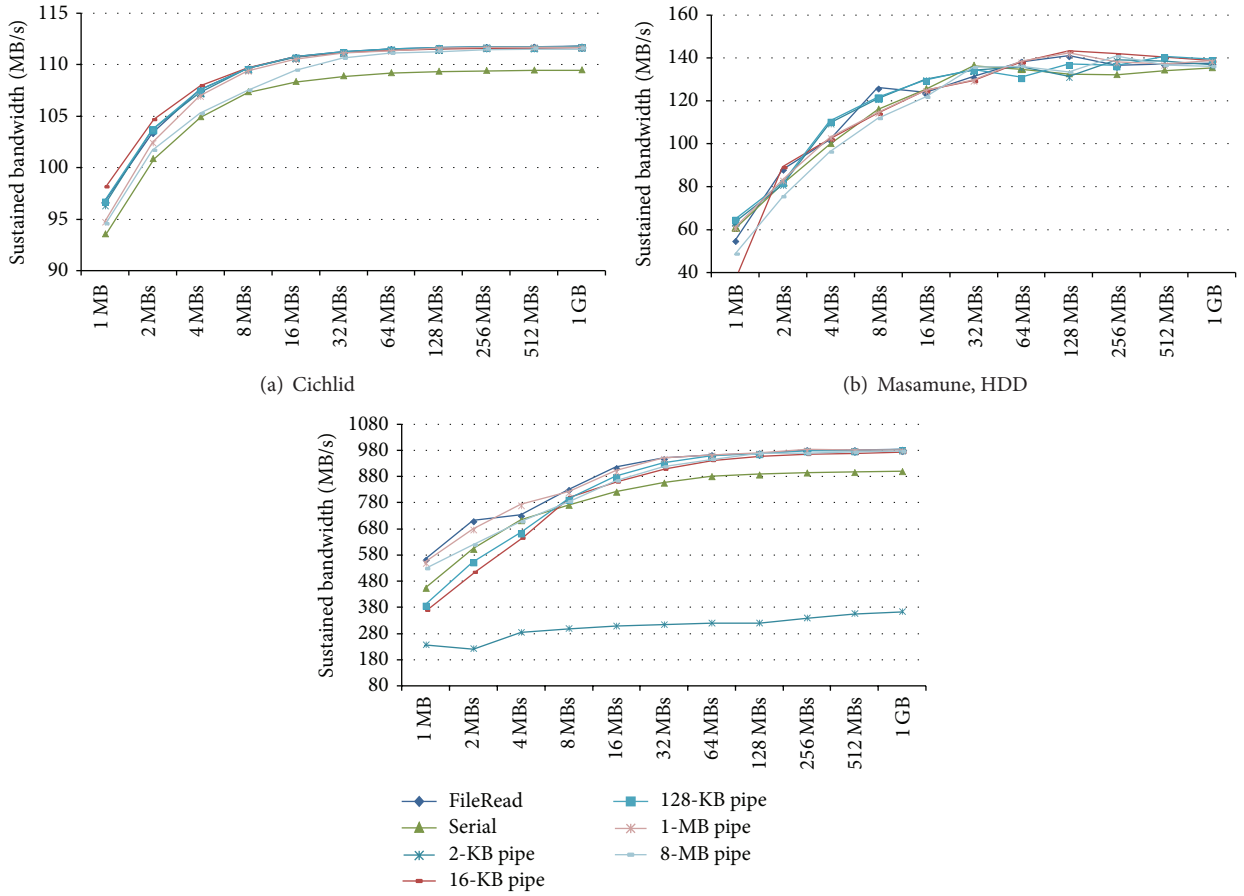


Figure 3: Sustained bandwidth (MB/s) of `clEnqueueReadBufferToStdioFile`.

results indicate that the clDataTransfer extension can hide the host-device data transfer latency and hence the sustained performance of the data transfer from a file to a device memory buffer is almost comparable to the sustained bandwidth of just reading a file, that is, FileRead. A programmer can use the optimized data transfer implementation by just enqueuing a clDataTransfer command.

In the case of reading from the HDD of Masamune, the file read time varies widely as shown in Figure 3. This is likely due to the bandwidth of the disk and the behaviors of the read-ahead thread in the OS kernel. As a result, the performance gain is unseen. The FileRead performance is sometimes even lower than that of `clEnqueueWriteBufferFromStdioFile` because of the intrinsic measurement accuracy.

5.2.2. Evaluation with the Smith Waterman Algorithm. In this work, a CUDA program of the Smith Waterman algorithm [11] is ported to OpenCL. Then, the performance of the OpenCL version is evaluated to show that clDataTransfer can hide the host-device data transfer latency of a real application by overlapping it with the file access latency. In the Smith Waterman program, the data transfer time can be overlapped with the computation time. However, the data transfer time is still partially exposed to the total execution time if the computation time is shorter than the data transfer time. The exposed data transfer time depends on the problem size. Therefore, in this evaluation, the overlap of computation and data transfer is disabled, and the fully exposed data transfer time is evaluated to clearly show the effect of overlapping the host-device data transfer latency with the file access latency.

The OpenCL program repeatedly reads the data in files to host memory buffers and sends them to device memory buffers. Suppose that `d_db` and `h_db` are handles of a device memory buffer and a host memory buffer, respectively. Their buffer size is `readsz`, and the file pointer is `fp`. Then, the original code has the following code pattern:

```
fread(h_db, readsz, 1, fp);
clEnqueueWriteBuffer(cmd,
d_db, CL_TRUE, 0, readsz, h_db, 0, NULL, NULL);
```

The above pattern is replaced with an additional OpenCL command enqueued by

```
clEnqueueWriteBufferFromStdioFile
(cmd, d_db, CL_TRUE, 0, readsz, fp, 0,
NULL, NULL);
```

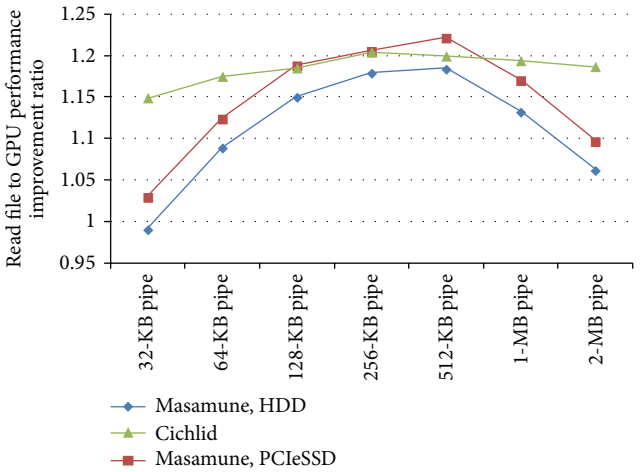


FIGURE 4: The improvement ratio of data transfer performance for the Smith Waterman algorithm.

The results of evaluating the data transfer time with changing the pipeline buffer size are shown in Figure 4. Here, the data transfer time is the total time of data transfers from a database file to a device memory buffer. These results indicate that the clDataTransfer extension can reduce the data transfer time if the pipeline buffer size is appropriately configured. The performance improvement of the clDataTransfer extension decreases if the pipeline buffer size is too small due to the runtime overhead of the pipeline implementation. It also decreases if the pipeline buffer size is too large compared to the data size, because pipelining with a too large buffer does not benefit from overlapping of data transfers. Accordingly, the optimal pipeline buffer size depends not only on the storage performance but also on the data size to be transferred from a file to a device memory buffer. The pipeline buffer size has to be dynamically adjusted because the data size is usually determined at runtime. Figure 4 discusses the effect of changing the pipeline buffer size on performance. Since the clDataTransfer extension hides the implementation details of data transfers, it is technically possible to employ empirical parameter tuning or autotuning for automatically finding the optimal pipeline buffer size, as in MVAPICH2-GPU's CUDA support.

In the Smith Waterman program, the data size to be read from a file ranges from 511 bytes to 4 Mbytes and hence is relatively small. The sustained bandwidths of both the file read and the host-device data transfer become lower for the transfer of a small data chunk. If the program is used for large input data, we believe that the performance improvement by clDataTransfer would become more remarkable as indicated in Figure 3.

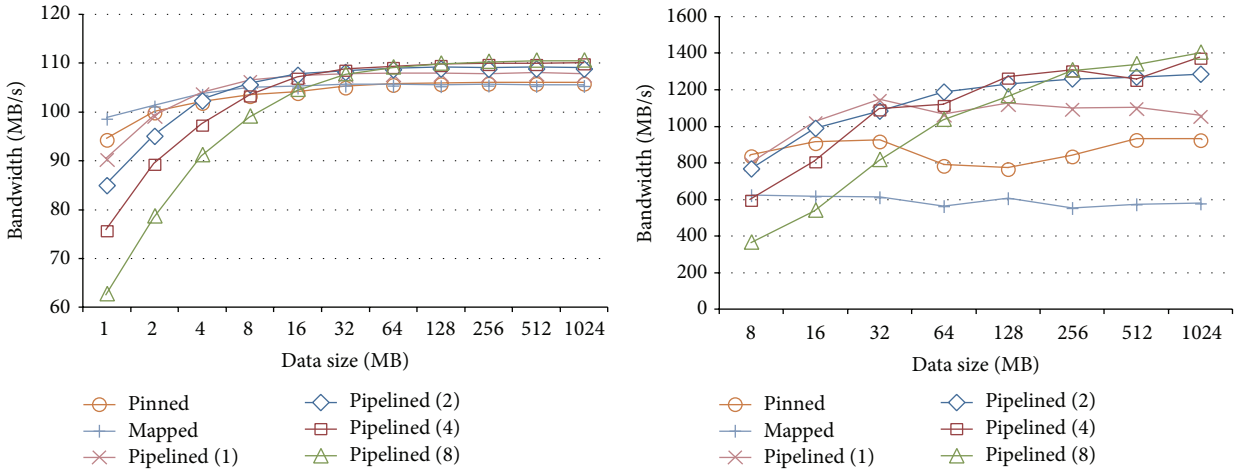
5.3. Evaluation of Internode Communication Performance

5.3.1. Point-to-Point Communication Performance. One advantage of the clDataTransfer extension over conventional joint programming of MPI and OpenCL is that the clDataTransfer extension can hide the implementation details of system-aware optimization for efficient data transfers.

Figure 5 shows the difference in sustained bandwidth among pinned, mapped, and pipelined implementations described in Section 3. 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 5(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 internode communication, and hence the pipelined implementation hardly improves the sustained bandwidth. On the other hand, in Figure 5(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 multinode GPU applications to achieve a high performance, and hence some abstractions of internode data transfers are necessary for high performance-portability. For example, on RICC, the pinned 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 clDataTransfer extension provides interfaces that abstract internode 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 clDataTransfer runtime can use either the pinned or the 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 the mapped data transfer. In the following evaluation, the mapped 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.

5.3.2. Evaluation with the Himeno Benchmark. The performance impact of using the clDataTransfer 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 pinned 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



(a) Cichlid (b) RICC

Figure 5: Sustained bandwidth of peer-to-peer communication.

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 clDataTransfer extension, called the clDataTransfer implementation.

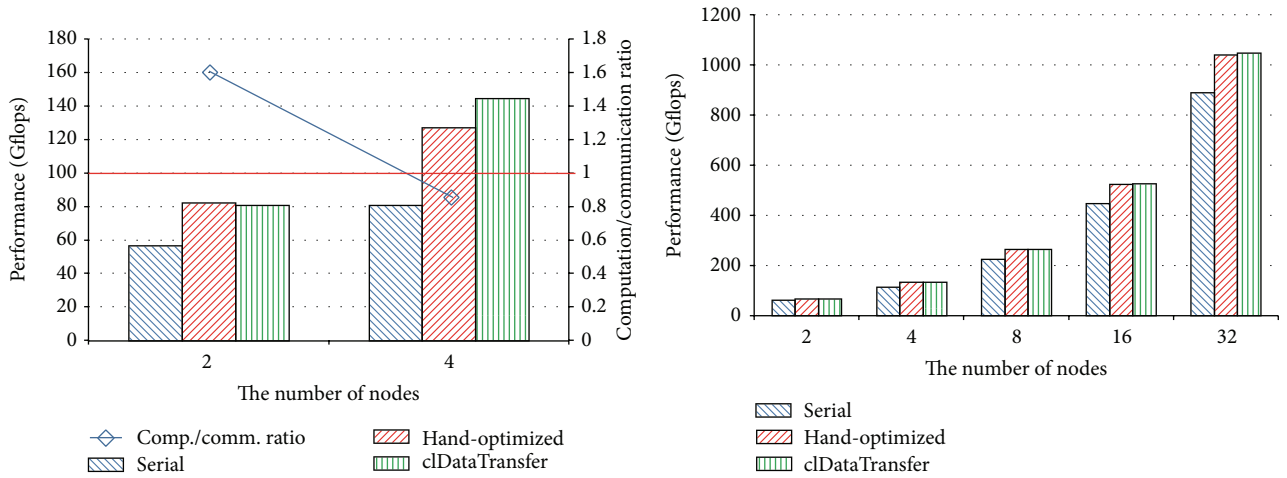
Figure 6 shows the sustained performances of the three implementations for the Himeno benchmark with M-size data. Since the hand-optimized implementation is well designed for overlapping the computations and communications, it can always achieve a higher performance than the serial implementation; the average speedup ratios are 51.2% and 15.2% for Cichlid and RICC, respectively. The performance of the clDataTransfer implementation is almost always comparable to that of the hand-optimized implementation because the communication times of both the hand-optimized and the clDataTransfer implementations are not exposed to their total execution times. Accordingly, the clDataTransfer extension allows an application programmer to easily overlap the communication and computation by simply sending internode communication commands to devices and utilizing OpenCL event objects to enforce the dependencies among OpenCL commands.

The results in Figure 6(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 is less than one, and hence 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 clDataTransfer implementation. The main reason of the performance difference is that the mapped data transfer behind the clDataTransfer 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 clDataTransfer 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 shown in Listing 4.

5.3.3. Evaluation with a Practical Application. The performance impact of the clDataTransfer extension is further discussed by taking the nanopowder growth simulation [14] 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 clDataTransfer 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 clDataTransfer extension. One is the baseline implementation that just uses `MPI_Isend` and `MPI_Recv` for coefficient data distribution. The other is the clDataTransfer implementation, which uses `MPI_Isend` with `MPI_CL_MEM` to send the coefficients in host memory buffers and `clEnqueueRecvBuffer` to receive them.

Figure 7 shows the results to compare the performances of the two implementations on RICC. Unlike the Himeno benchmark, the communication overheads are obviously



(a) Cichlid (b) RICC

Figure 6: The performance for the Himeno benchmark.

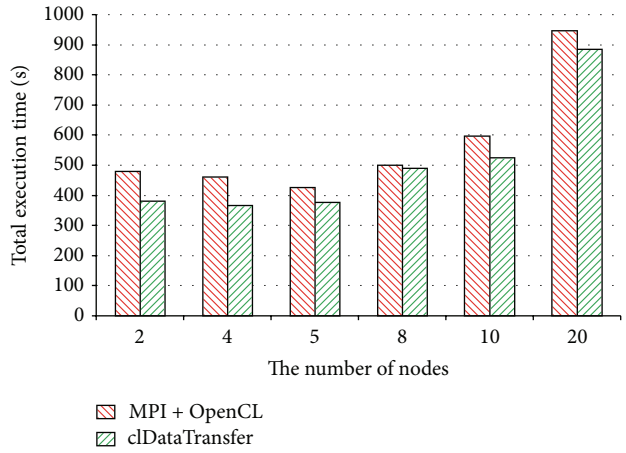


Figure 7: The execution time of the nanopowder growth simulation (700 simulation steps).

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. Because of the poor parallelism, the performance degrades when the number of nodes increases beyond 8.

As shown in Figure 7, the clDataTransfer outperforms the baseline implementation because it can exploit an optimized implementation that overlaps the host-device communication with the internode 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 clDataTransfer 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 clDataTransfer extension. This is very important because most of existing applications have been developed using MPI.

6. Conclusions

This paper has proposed an OpenCL extension, clDataTransfer, to allow OpenCL to perform data transfers that need collaboration between hosts and compute devices. In the clDataTransfer extension, additional OpenCL commands are defined for encapsulating common programming patterns in data transfers from/to the device memory, such as internode communications and file accesses. 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, data transfers indicated by the additional commands are incorporated into the OpenCL execution model in a natural manner.

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

The performance evaluation results clearly show that clDataTransfer can achieve efficient data transfers while hiding the complicated implementation details, resulting in higher performance and scalability. Moreover, using the clDataTransfer extension, the host thread of an application is not blocked to serialize dependent operations of data

transfers. As a result, the clDataTransfer extension allows an application programmer to easily use the opportunities to overlap communications and storage accesses with computations.

Although this work focuses on OpenCL, we believe that the idea itself could be applicable to other programming models such as CUDA. In the future, we will further improve the extension so that it can support other kinds of tasks that need help of host threads, such as system calls.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

The authors would like to thank Professor Mayasa Shigeta and Professor Fumihiko Ino of Osaka University for allowing them to use their simulation codes 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 is partially supported by JST CREST "An Evolutionary Approach to Construction of a Software Development Environment for Massively-Parallel Heterogeneous Systems" and Grants-in-Aid for Scientific Research (B) nos. 25280041 and 25280012. The work is also partly supported by DoE Vancouver Project (DE-SC0005515).

References

[1] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan Kaufmann Publishers, 2007.

[2] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa, *Heterogeneous Computing with OpenCL*, Morgan Kaufmann, Boston, Mass, USA, 2011.

[3] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface*, The MIT Press, 1999.

[4] O. S. Lawlor, "Message passing for GPGPU clusters: cudaMPI," in *Proceedings of the IEEE International Conference on Cluster Comptuing and Workshops (CLUSTER '09)*, pp. 1–8, 2009.

[5] A. M. Aji, J. Dinan, D. Buntinas et al., "MPI-ACC: An integrated and extensible approach to data movement in accelerator-based systems," in *Proceedings of the 14th IEEE International Conference on High Performance Computing and Communications (HPCC '12)*, pp. 647–654, Liverpool, UK, June 2012.

[6] H. Wang, S. Potluri, M. Luo, A. K. Singh, S. Sur, and D. K. Panda, "MVAPICH2-GPU: optimized GPU to GPU communication for InfiniBand clusters," *Computer Science: Research and Development*, vol. 26, no. 3-4, pp. 257–266, 2011.

[7] J. A. Stuart, P. Balaji, and J. D. Owens, "Extending MPI to accelerators," in *Proceedings of the 1st Workshop on Architectures and Systems for Big Data (ASBD '11)*, pp. 19–23, 2011.

[8] I. Gelado, J. Cabezas, N. Navarro, J. E. Stone, S. Patel, and W.-M. W. Hwu, "An asymmetric distributed shared memory model for heterogeneous parallel systems," in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '10)*, pp. 347–358, March 2010.

[9] J. A. Stuart and J. D. Owens, "Message passing on data-parallel architectures," in *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS '09)*, pp. 1–12, May 2009.

[10] A. Barak, T. Ben-Nun, E. Levy, and A. Shiloh, "A package for OpenCL based heterogeneous computing on clusters with many GPU devices," in *Proceedings of the IEEE International Conference on Cluster Computing Workshops and Posters*, pp. 1–7, September 2010.

[11] Y. Munekawa, F. Ino, and K. Hagihara, "Design and implementation of the Smith-Waterman algorithm on the CUDA-compatible GPU," in *Proceedings of the 8th IEEE International Conference on BioInformatics and BioEngineering (BIBE '08)*, pp. 1–6, October 2008.

[12] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, vol. 25, pp. 1097–1105, 2012.

[13] E. H. Phillips and M. Fatica, "Implementing the Himeno benchmark with CUDA on GPU clusters," in *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS '10)*, pp. 1–10, April 2010.

[14] M. Shigeta and T. Watanabe, "Growth model of binary alloy nanopowders for thermal plasma synthesis," *Journal of Applied Physics*, vol. 108, no. 4, Article ID 043306, 2010.

[15] The Open MPI Project, "Open MPI: open source high performance computing," http://www.open-mpi.org/.