TASK LEVEL PIPELINING ON MULTIPLE ACCELERATORS VIA FPGA SWITCH

Takaaki Miyajima †, Takuya Kuhara †, Toshihiro Hanawa ‡, Hideharu Amano †, Taisuke Boku ‡
† Graduate School of Science and Technology, Keio University,
3-14-1 Hiyoshi, Kouhoku-ku, Yokohama, Kanagawa, Japan
‡ Center for Computational Sciences, University of Tsukuba,
1-1-1, Tennodai Tsukuba-city, Ibaraki 305-8577, Japan
peach@am.ics.keio.ac.jp

ABSTRACT
We show a task level pipelining on multiple accelerators with PEACH2. PEACH2, which is implemented on FPGA, enables ultra low latency direct communication among multiple accelerators over computational nodes. By installing PEACH2, typical high performance computation nodes are tightly coupled. In this environment, application can be accelerated by exploiting not only data level parallelism, but also task level parallelism. Furthermore, we can process multiple task on multiple accelerators in a pipelined manner. In our evaluation, pipelined application which is implemented in a task level pipelined manner achieves 52% speed up compared to a single GPU.

KEY WORDS
Interconnect for accelerators, GPU cluster, Accelerator computing, FPGA Interconnect, Task Level Pipeline

1 Introduction
High performance computers consisting of multiple accelerators are well known to enormous speed up of data parallel applications. Such data parallel applications are implemented on multiple accelerators as follows. A user first exploits data level parallelism (DLP) and distributes them to each accelerator, then the same computation is applied to the data in all the accelerators. Finally, result data are sent back to the host CPU. If other computations are left, the result data are distributed and processed again. However, the bottle neck of this type of implementation is bandwidth to distribute the data. Since the different data are required by each accelerator, the required memory bandwidth linearly increases along with the number of accelerators. When we increase the number of GPUs, the problems of latency and data bandwidth between GPUs over the nodes become critical. Current CUDA API partly supports direct communication between GPUs within node, but it doesn’t support between nodes. To communicate with other accelerators in different nodes, multiple memory copies via the CPU memory are required. These multiple memory copies cause an increase of latency that severely degrades performance, especially in the case of a short message.

Task Level Pipelining is yet another way to implement applications. With regard to the bandwidth, it doesn’t increase required bandwidth along with the number of accelerators, since the same number of data are required and intermediate result data are constant. Task level pipeline utilizes inter task parallelism and runs tasks in a pipelined manner. Such implementation processes on multiple accelerators in the different manner from data parallel one. A user first exploits task parallelism and assigns tasks to each accelerator. Accelerators are basically connected in a line to realize the corresponding data flow. Then the data are processed along with the data flow on accelerators. Task level pipeline on multiple accelerators is very common in embedded application, but not so much in GPU. The problem is still on the latency and bandwidth between GPUs.

To deal with the problem of direct communication between nodes among multiple GPUs over nodes, Tightly Coupled Accelerators (TCA) and PEACH2 have been proposed[1]. TCA is capable of low latency communication between accelerators over different nodes. Compared to other non-direct communication, it can eliminate memory copy overhead. PEACH2 which provides that low latency communication is installed to ordinary supercomputer nodes to realize TCA. The key point of PEACH2 is that, it enables direct communication via the standard PCIe protocol, thus protocol conversion overhead is also eliminated. We implemented applications in task level pipeline manner on TCA so as to explore the capability of
pipeline implementation on multiple GPUs. In our evaluation, pipelined application which is implemented in a task level pipelined manner achieves 52% speed up compared to a single GPU.

This paper is organized as follows. In Section 2, we explain architecture of TCA and PEACH2. Then Section 3 describes the fundamental concept and implementation of task level pipelining on multiple GPUs. And then we evaluate our concept by using application in Section 4. Finally, we conclude the paper.

2 Architecture of PEACH2 and TCA

Reducing communication latency is necessary to fully utilize multiple accelerators, since the I/O bandwidth bottleneck causes serious performance degradation. PCI-Express Adaptive Communication Hub 2 (PEACH2) is an FPGA switch which provides an ultra low latency direct communication via PCI-Express[1]. In the current version, up to 32 GPUs within 16 nodes can communicate less than 2.5 µsec at minimum thanks to PEACH2. We call such accelerators Tightly Coupled Accelerators (TCA). This section describes an architecture of PEACH2 and TCA which are used in our proposal.

2.1 PCI-Express Adaptive Communication Hub 2 (PEACH2)

As shown in Figure 1, PEACH2 board uses Altera’s Stratix IV GX530NF45 FPGA as a switch core. It has four PCIe Gen2 x8 lane, DMA controller, NIOS soft core processor, and DDR3-SDRAM memory. Figure 2 depicts the block diagram of PEACH2. Four PCIe ports and a DMA controller are implemented as hard IPs and connected via Avalon Streaming (Avalon ST) interface. The DMA controller, the core of PEACH2, can deal with multiple descriptors in arbitrary position in order to realize continuous transfer. In addition, DDR3-SDRAM can be accessed in one clock cycle. FPGA logic utilization is less than 25%, and running frequency is 250MHz. Stratix IV GX has 32 GBX (Gigabit Transceiver Block) transceiver, and all of them are used for four PCIe lanes. Typically, north port is connected to accelerators in the host node, and east/west/south ports are connected to another PEACH2 in other nodes. PEACH2 can be used to build various network topologies. Available topologies are fully connected mesh topology with 4 TCA nodes, cube topology with 8 TCA

Figure 3: Ping-Pong Latency (left axis, solid line) and Bandwidth (right axis, dash line) between CPU and GPU: Bandwidth reaches to 93% of theoretical peak performance.

Figure 4: Latency (left axis, solid line) and Bandwidth (right axis, dash line) between CPU/GPU-DDR3 on PEACH2: Except for GPU to DDR3 on PEACH2, latency is less than 4 µsec. Bandwidth reaches to 78% of theoretical peak performance.
nodes, or ring topology with 16 TCA nodes.

Figure 3 shows ping-pong communication latency and bandwidth between two GPUs. Average ping-pong latency between two GPUs is around $2.5 \mu\text{sec}$ when the data size is less than 2048 bytes. It is 5 times faster than cudaMemcpy() with UVA (Unified Virtual Address), and 8 times faster than MVAPICH2. About bandwidth, 93% of theoretical peak performance is achieved when the data size is 256 KBytes. In particular, PEACH2 provides better performance over CUDA, MVAPICH2 in latency and bandwidth.

The latency and bandwidth of DDR3 on PEACH2 memory is also shown in Figure 4. Latency is less than $4\mu\text{sec}$ in all parts except for GPU to DDR3 on PEACH2. Besides, bandwidth reaches 78% of theoretical peak performance and 4.8Gbps when the data size is 128 KBytes. In particular, DDR3 on PEACH2 memory can be used when the data size is large. In section 4, we transfer the image data that the size is larger than 2MBytes. Note that latency and bandwidth from GPU to DDR3 on PEACH2 are much worse than the others. This is caused by smallness of PCIe buffer of Intel’s Sandy Bridge architecture. This problem will be solved on Intel’s Ivy Bridge architecture.

There are some researches on FPGA based network system. APEnet+[2] is one of the closest idea to PEACH2. Different from PEACH2 which is compatible with PCIe protocol, APEnet+ uses their own protocol to realize direct connection between GPUs and network interface. In the respect, PEACH2 has an advantage to APEnet+ since it isn’t required to convert protocols.

2.2 Tightly Coupled Accelerators (TCA)

TCA realizes direct communication among multiple accelerators over computation nodes thanks to PEACH2. In our environment, each computation node (TCA node) has two Ivy Bridge-EP processors, four GPUs, and one PEACH2 board. Specification of used TCA node is also shown in Table 1. By using PEACH2, each GPU is connected via PCIe Gen2 x8 with small input/output overhead. Figure 5 depicts an overview of TCA node. Taking advantage of PEACH2, on the left of the figure, up to 32 GPUs can directly communicate another GPU in another TCA node. This bunch of TCA nodes is called Node Cluster. Up to 16 TCA nodes can consist a node cluster, and multiple Node Clusters are connected via InfiniBand.

A heterogeneous supercomputer cluster which has CPU, GPU, and FPGA, has been researched. AXEL [3] from Imperial College London has one with multiple CPUs, GPUs, and FPGAs. These three computational units are conjoined via PCI. From the viewpoint of a CPU, FPGAs and GPUs are equivalent as an accelerator. Namely, the FPGA is just used for an accelerator, not a communication system. We discuss the difference of application implementation in Section 3.1.1. QP [4] from Illinois University is more similar to TCA node. It has two CPUs, four GPUs, and one FPGA. However, the FPGA is just used for an accelerator as well as AXEL. In this respect, TCA node with PEACH2 which can be used as communication system and accelerator is different from previous researches. Additionally, AXEL and QP must return the output data to CPU memory once, even if the user just wants to transfer the data to another accelerator. TCA realizes ultra low latency direct communication between accelerators, that is, the TCA is more suitable for stream computation than AXEL or QP.
Figure 6: Data Level Parallel on ordinary computer nodes: Data are divided and assigned to each GPU evenly, inter communications are done if needed.

Figure 7: Inter accelerator pipelining on ordinary computer nodes. Data are required to be copied to CPU to communicate with another nodes.

Figure 8: Inter accelerator pipelining on multiple TCA nodes: PEACH2 enables direct communication among GPUs over TCA nodes.

sends the data to each GPU, and then each GPU starts processing. Each processing step becomes sequential and scatter/gather process is required along with each step as well.

In the case of IAP on ordinary supercomputer nodes, which is shown in Figure 7, the host CPU sends the data to GPU in the node, then GPU starts processing. Finally, result data are sent to the next pipeline stage on another GPU via host CPU. Communication via host CPU still degrades performance. This is one of the reasons why task level pipeline on multiple GPUs is not easy.

Compared to IAP on ordinary supercomputer nodes, IAP on TCA nodes can eliminate extra copy to CPU. In the case of IAP on TCA which is shown in Figure 8, direct communicate is enabled thanks to PEACH2, and communication latency is dramatically reduced. From the point of view of the pipeline, reducing the communication latency corresponds to shorten the processing time of each stage. As a result, whole processing time of a pipeline is reduced. Additionally, some other benefits can be found when we implement stream application. For example, merging output data from multiple GPUs into one large output data in chronological order. When we implement such application in DLP style on multiple GPUs, extra merging process is needed. On the other hand, pipeline doesn’t need such process since it gets input data and generates output data in chronological order naturally.

3.1.1 Related Work

Little research has been done on this topic for multiple GPUs. AXEL[3], the heterogeneous supercomputer cluster from Imperial College, has Map-Reduce framework to run application efficiently over multiple AXEL nodes. In the paper, they implemented N-body simulation while exploiting data parallelism by using the framework. Although data parallel application works well in the framework and AXEL nodes, stream application is hard to apply this environment. Additionally, AXEL has an inter node communication network (PCI-X 2.0, 1.066Gbps at maximum [6]), but they didn’t use it in the paper. Huynh et.al. proposed a framework for implementing stream application on multiple GPUs [7]. They focused on programming model to exploit task parallelism and to describe stream application, since we propose an practical case.

In the area of high performance computing with multiple FPGAs, Sano et al. proposed a similar concept in multiple FPGAs environment[8]. They focused on time axis task parallelism, and achieved strong scaling on stencil computation of CFD while keeping memory bandwidth constant. Their benchmark application is relatively simple and target platform is FPGAs, that is, communication between accelerators, inter accelerator data, and pipelining, problems (we discuss this later) become simple.

In their case, accelerators are connected via two 1GB/s uni-direction wire (High-speed Terasic Connectors). It didn’t become bottleneck of this system. Storage which stores inter accelerator data is consisting of registers in order to fully utilize FPGA resource, and pipeline is fixed. TCA can deal with more various problem since it has CPU, GPU and FPGA. However, the above three problems are difficult to solve. We discuss these problems in the Section 3.2.

3.2 Implementation

When we construct an inter accelerator pipeline on multiple GPUs, there are three major problems. First is communication latency between accelerators, the second is storage which stores inter accelerator data, and the last is pipelining itself. We used PEACH2 to address first two problems, and used Intel Thread Building Blocks[9] for the last problem as well. First problem which is solved by PEACH2 is described in previous sections, so we talk about the other two problems in this section.
3.2.1 Pipelining on Multiple Accelerators

Implementation of task level pipeline is difficult since it requires many parallel programming technique, for example exclusive control. There are some open source libraries which realize task pipeline on CPU, for example pthread[10], Boost::thread[11], or Glib::thread[12]. Intel Thread Building Blocks (TBB) is a most flexible library. It provides software controlled task pipeline using multiple threads. The tbb::pipeline class is provided to implement task pipeline, and each stage has token to keep processing flow. A user fist writes each task which runs in each stage of pipeline. Then the user registers tasks and also defines a processing order of tasks to TBB. In the Section 4, we divide an image processing algorithm called Sobel operator into three tasks, and processed them in a simple straight forward pipelined manner.

Pipeline structure of TBB is based on [13], data and task assignment to threads is controlled by a master thread. Besides, each task runs on the slave thread. This type of pipeline makes it easy to reorder tasks and insert new tasks. Input/output data to/from tasks and storing are done when the master thread receives the finish token from slave threads. TBB assigns a task which is registered by the user to an idle slave thread, and also transfers input data. Then the slave thread runs task, and finally sends back output data and the finish token to master thread. TBB can deal with tasks which are written in CUDA. To our knowledge, the other libraries are more difficult than TBB to make task pipeline. TBB also realizes a double buffering when the two or more GPU tasks are registered. Figure 10 depicts an implementation of task pipeline using TBB on an ordinary supercomputer node. Dash line shows task token communication and solid line shows data communication. Both token and data are sent to the master thread in host CPU.

We conducted a preliminary evaluation of a scalability of TBB pipeline. Used application is applying 100 input images to Sobel operator 32 times. For it, 32 stages straight forward pipeline is implemented. The evaluation is done on TCA node and a CPU. When 13 stages run in parallel, it achieved 89% speed up compared to a single stage. Additionally, when 32 stages run in parallel, they achieved 92% speed up while the speed up is saturated. According to the evaluation, TBB can construct effective task pipeline. So we decided to use Intel TBB to construct task level pipeline.

3.2.2 Storage Which Stores Inter Accelerator Data

Task level pipeline often requires large size intermediate data between tasks. For example, image processing which applies multiple filter to input image. Same as task level pipeline, IAP requires intermediate data between accelerators as well. FIFO is a common way to store such intermediate data to keep the processing order. Once processing time becomes imbalance, FIFO depth increase and large size memory is also required. This storage problem should be carefully determined since transfer time degrades the total performance.

In the case of TCA, there are three large size memory modules, DDR3 on CPU, GDDR5 on GPU, and DDR3 on PEACH2. Considering the latency, bandwidth, and flexibility, we chose DDR3 on PEACH2 as a storage to store inter accelerator data. As we described in Section 2, DDR3 on PEACH2 is the best option since the latency is the same as DDR3 on CPU, and bandwidth is enough for large data. Furthermore, if we construct more complicated pipeline (e.g. feed back loop), mutex or data control mechanism are needed. Extra FPGA resource on PEACH2 can provide the mechanism. Another option is using DDR3 on CPU, however, an extra FPGA resource on PEACH2 will be used for other task such as reduction operation. Thus, this option may cause unnecessary data transfers when we use FPGA resource. So we decided to use DDR3 on PEACH2 to store inter accelerator data as shown in Figure 8.

To store inter accelerator data to DDR3 on PEACH2, we modified each task of TBB. TBB’s pipeline (tbb::pipeline class) uses simple token to maintain processing flow. By default, a task of TBB pipeline returns inter accelerator data to master thread (DDR3 on CPU), and also returns token. It is equal to use single entry array and limitation to use multiple entry. We changed each task not to return the inter accelerator data as follows. After finishing the processing, we store inter accelerator data to DDR3 on PEACH2 by using our data send API at the last of each task, then return the property of stored data instead of the data to master thread. The property is required to load the data from PEACH2 in the next stage. Currently, the property is just eight bytes, and includes the size and the bit depth of the inter accelerator data.

4 Evaluation

We show an evaluation of Inter Accelerator Pipelining (IAP) on TCA node in this section. Simple image processing application is implemented in IAP style, and execution time is evaluated. Evaluation environment is shown in Table 1. We also used an Equation 1 to measure speed up ratio.

<table>
<thead>
<tr>
<th>OS</th>
<th>Scientific Linux 6.3 (kernel 2.6.32)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Xeon E5-2670 @ 2.6GHz (Sandy Bridge-EP) × 2</td>
</tr>
<tr>
<td>PCI-express</td>
<td>Gen.3 × 80 Lane (40 Lane/CPU)</td>
</tr>
<tr>
<td>Memory</td>
<td>64 GB, DDR3 1866MHz</td>
</tr>
<tr>
<td>GPU</td>
<td>NVIDIA Tesla K20m (GK110)</td>
</tr>
<tr>
<td># of GPUs / Node</td>
<td>2</td>
</tr>
<tr>
<td>Single Precision</td>
<td>1.31 TFLOPS/GPU</td>
</tr>
<tr>
<td>Double Precision</td>
<td>3.59 TFLOPS/GPU</td>
</tr>
<tr>
<td>Global Memory</td>
<td>6 GByte/GPU</td>
</tr>
<tr>
<td>Others</td>
<td>PEACH2</td>
</tr>
</tbody>
</table>
Figure 9: Implementation of data level parallel. CPU runs one or two threads to control GPUs and each GPU runs the same processing. (DLP1, DLP2)

Figure 10: Implementation of inter accelerator pipeline without PEACH2. Data transfer time between GPUs degrade total performance. (IAP1, IAP2)

Figure 11: Implementation of inter accelerator pipeline with PEACH2. Stage tokens and data are sent to CPU and PEACH2 respectively. (IAP-P2)

\[
\text{Speedup}[\%] = \frac{T_{IAP}}{T_{DLP} \times 100} \quad (1)
\]

4.1 Application

An image filter called Sobel operator is used for the evaluation. It consists of three tasks, 50 times of x-axis Sobel operator (“xSobel”), 50 times of y-axis Sobel operator (“ySobel”), and generation of the output image in PNG format (“imout”). 100 images (1280x720 pixel) are processed. x-Sobel and y-Sobel run on GPU, and imout runs on CPU. Each task is implemented in OpenCV which is an open source library for image processing [14].

Six different versions of Sobel operator were implemented as described in Table 2. “CPU” running on only a CPU is the base line. CPU is implemented in task level pipeline style, and used three physical threads. “DLP1” and “DLP2” are typical implementation for GPU, data level parallelism is exploited as shown in figure 9. In the case of DLP1, on the CPU a thread to control one GPU runs. All 100 images are send to the GPU, then x-Sobel and y-Sobel are applied sequentially. In the case of DLP2, two threads to control two GPUs run on the CPU. 50 images are sent to each GPU, then x-Sobel and y-Sobel are applied sequentially. We used Intel TBB’s thread to run threads without using OpenMP or some other parallelise libraries.

“IAP1”, “IAP2” and “IAP2-P2” are implemented in IAP style. In these three versions, three tasks are applied to each input image in a pipelined manner. All tasks are registered to TBB and pipelined as we described in Section 3.2.1. Setting of TBB is decided to achieve the shortest execution time. Intermediate data of stages are stored in DDR3 on PEACH2 or DDR3 on CPU. IAP1 isn’t fully implemented in IAP style since only one GPU is used and two tasks are switched in the single GPU. For IAP2 and IAP2-P2, all tasks run in a pipelined manner including data communication on GPUs. IAP2 which is shown in Figure 10, is assumed running on an ordinary computer node. It is necessary to copy inter accelerator data to CPU to send another node. On the other hand, IAP2-P2 which uses PEACH2 is not required to copy the data to CPU. Additionally, the data are stored to DDR3 on PEACH2. Figure 10 depicts IAP-P2. Note that IAP1 sends back the intermediate date to CPU in order to make IAPs under fair condition.

### Table 2: Six different implementation of Sobel operator

<table>
<thead>
<tr>
<th>Style</th>
<th>Used GPUs</th>
<th>Used memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Task Pipeline</td>
<td>0</td>
</tr>
<tr>
<td>DLP1</td>
<td>Data Parallel</td>
<td>1</td>
</tr>
<tr>
<td>DLP2</td>
<td>Data Parallel</td>
<td>2</td>
</tr>
<tr>
<td>IAP1</td>
<td>Task Pipeline</td>
<td>1</td>
</tr>
<tr>
<td>IAP2</td>
<td>Task Pipeline</td>
<td>2</td>
</tr>
<tr>
<td>IAP2-P2</td>
<td>Task Pipeline</td>
<td>2</td>
</tr>
</tbody>
</table>

From the viewpoint of data movement, DLP and IAP are completely different. Main drawback of DLP implementation is that, imout cannot start process until all the images are finished to be applied two Sobel operators. On the other hand, IAP can start imout stage once after each image is finished to be processed in the former stages, since it is implemented in pipelined manner. Such sequential processing can be found in many applications.

4.2 Result

Figure 12 shows a processing time and a chronological processing order of five GPU implementations. CPU only implementation takes 293.1 [sec]. It is not included in the figure due to limitations of space. Implementations using GPU achieved at least 88.1% speed up compared to CPU. Processing time to apply three processes to each image is as follows, xSobel is 0.12 [sec], ySobel is 0.12 [sec], and imout is 0.04 [sec]. These numbers are the same in all implementations. The difference is processing order and the
Figure 12: Processing time comparison and chronological processing order. IAPs are faster than DLPs at least 33%. IAP2-P2 which stores data to PEACH2 is almost the same as IAP2 which stores data to CPU. DLP1/2 cannot start imout until x/y-Sobel are finished. IAP1/2 and IAP2-P2, pipelined implementations, can start imout once after each image is applied x/y-Sobel.

Processing time of each task for one image is as follows, xSobel is 0.12 [sec], ySobel is 0.12 [sec], and imout is 0.04 [sec].

The number of used GPUs. Note that, data transfer time is not shown in Figure 12 due to space limitations.

Processing time of this application is short, namely, data transfer time accounts for larger portion of total processing time. Longer data transfer time and short processing time usually cause negative effect to the total processing time, however IAPs show speed up. That is, we can expect that applications which has the opposite feature will show better result.

In the case of data level parallel implementations, processing time of x/y-Sobel of DLP2 takes only half of that of DLP1, since evaluation application has no task and data dependencies. By contrast, whole processing time of DLP2 doesn’t become a half of that of DLP1 since imout cannot start processing until two Sobel operators are finished.

In the case of inter accelerator pipeline, IAP1 achieved 33% speed up compared to DLP2. IAP1 which is implemented in pipelined manner successfully hides the data transfer time and the processing time of imout. Scheduler inside a GPU switches two Sobel operator tasks. IAP2 and IAP2-P2 achieved 52% speed up compared to DLP2, and also achieved 28% speed up compared to IAP1. These implementations do not require task switch. Specifically, advantage of task level pipelining on multiple accelerator is proved.

Difference between IAP2 and IAP2-P2 is the place to store 2MByte inter accelerator data. In the case of IAP2, round trip data transfer between GPU and CPU occurs twice as shown in Figure 10. It takes approximately 5,000 \( \mu \text{sec} \) in total. On the other hand, in the case of IAP2-P2, round trip between GPU and CPU and between GPU and DDR3 on PEACH2 occurs once respectively as shown in Figure 11. It takes about 6,400 \( \mu \text{sec} \) in total. Table 3 shows more specific time of data transfer to send 2MB to each DDR3.

Data transfer time of IAP2-P2 is a bit slow compare to IAP2. There are some reasons. First of all, PEACH2 uses PCIe Gen2 x8 and CPU-GPU uses PCIe Gen3 x8. Furthermore, Smallness of Intel Sandy Brige’s PCIe buffer. In addition, current our data sending/receiving API for PEACH2 is beta version. Burst transfer is not fully supported. More precisely large data is divided into 4 byte data and transferred. Note that, this number is inside the single node, data transfer time of CPU and GPU to another TCA nodes becomes larger. In contrast, transfer time of GPU and PEACH2 is constant even if GPU sends the data to another TCA nodes.

<table>
<thead>
<tr>
<th></th>
<th>CPU to GPU</th>
<th>GPU to CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time [( \mu \text{sec} )]</td>
<td>1,170</td>
<td>1,340</td>
</tr>
<tr>
<td></td>
<td>GPU to PEACH2</td>
<td>PEACH2 to GPU</td>
</tr>
<tr>
<td>Time [( \mu \text{sec} )]</td>
<td>2,200</td>
<td>1,600</td>
</tr>
</tbody>
</table>

Table 3: Transfer time of 2MB data to each memory (inside a node)

5 Conclusion
In this paper, we proposed a task level pipelining on multiple GPUs in TCA node. GPUs are directory connected via FPGA switch, called PEACH2 that realizes low latency direct communication among multiple accelerators over up to 16 TCA nodes. Inter Accelerator Pipelining (IAP) is an implementation concept that connects multiple accelerators as a form of a pipeline computation. We implemented a simple image processing application on single TCA node in IAP style to evaluate the concept. We also consider three problems associated with the pipeline implementation. The problems are communication between accelerators, storage which stores inter accelerator data, and pipelining itself. We solved them by using PEACH2 and Intel TBB. In the case of the application, IAP implementation achieved 52% speed up compared to the data parallel implementation.

In future work, we measure actual multiple TCA nodes performance and compare the performance with
APEnet+[2]. We will also try to modeling an IAP application while considering the size of intermediate data and data transfer time. We also implemented some other more complicated applications to confirm the scalability. The applications will include multiple entry FIFO on the DDR3 on PEACH2.

Acknowledgement

The present study is supported in part by the JST/CREST program entitled “Research and Development on Unified Environment of Accelerated Computing and Interconnection for Post-Petascale Era” in the research area of “Development of System Software Technologies for post-Peta Scale High Performance Computing”.

References


