Communication Library to Overlap Computation and Communication for OpenCL Application

Toshiya Komoda*, Shinobu Miwa*, Hiroshi Nakamura*
*Graduate School of Information Science and Technology
The University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113-8656, Japan
Email: komoda,miwa,nakamura@hal.ipc.i.u-tokyo.ac.jp

***Abstract*—User-friendly parallel programming environments, such as CUDA and OpenCL are widely used for accelerators. They provide programmers with useful APIs, but the APIs are still low level primitives. Therefore, in order to apply communication optimization techniques, such as double buffering techniques, programmers have to manually write the programs with the primitives. Manual communication optimization requires programmers to have significant knowledge of both application characteristics and CPU-accelerator architecture. This prevents many application developers from effective utilization of accelerators. In addition, managing communication is a tedious and error-prone task even for expert programmers. Thus, it is necessary to develop a communication system which is highly abstracted but still capable of optimization. For this purpose, this paper proposes an OpenCL based communication library. To maximize performance improvement, the proposed library provides a simple but effective programming interface based on Stream Graph in order to specify an applications communication pattern. We have implemented a prototype system on OpenCL platform and applied it to several image processing applications. Our evaluation shows that the library successfully masks the details of accelerator memory management while it can achieve comparable speedup to manual optimization in which we use existing low level interfaces.**

***Keywords*-Accelerators; OpenCL; Double Buffering; Stream Graph**

I. Introduction

For data parallel applications, effective use of accelerators is indispensable for higher system performance and energy efficiency. Therefore, application developers are required to have deep knowledge of both application characteristics and CPU and accelerator architecture, which is a big programming challenge. To overcome this difficulty, a lot of effort has been made so far. Communication management between multiple heterogeneous devices is one of the most challenging problems, as host CPUs and accelerators generally have their own disjoint memory spaces.

Currently, the most popular accelerator programming environment is CUDA [1]or OpenCL [2]. They provide programmers with a low level application interface to manually manage the communication between CPU and accelerator memory. Such low level primitives are helpful for programmers to optimize communication and to maximize performance by taking application behaviors into account. However, this optimization is complex and error-prone, as the memory spaces of CPUs and accelerators are disjoint. This difficulty is one of the major reasons to prevent many application developers from effective utilization of accelerators.

To overcome this programmability problem, previous work has investigated techniques to hide accelerator memory management from programmers [3][4]. However, programmers still cannot automate advanced communication optimizations that consider communication pattern of application. This includes possible optimizations such as the well-known double buffering technique, which hides communication latency. As a consequence, they still have to write their applications with low level APIs. Considering this situation, it is necessary to ease the difficulty of accelerator memory management and to allow high-quality optimization of memory communication. In this paper, we investigate such a communication system where programmers can get the maximum benefit from accelerators without being aware of the complexity of communication management.

Strategy: In this paper, we focus on automating double buffering techniques which overlap computation and communication. In order to manage such a complex communication, an automatic communication system must be aware of an applications communication pattern. To get the information, several approaches exist. A first approach is the use of static compiler analysis. However this approach has the disadvantage that static analysis may only reveal limited insight due to complex usage of pointers in real applications. Another approach is the use of runtime memory management system that predicts an applications communication pattern for prefetching. However, this dynamic mechanism requires dedicated hardware support in order to determine whether the prefetched data is dirty or not. Thus such a system would be limited to specific hardware.

Our strategy is that programmers describe high level hint information through a simple user interface.The system automates the communication optimization by analyzing this information. One of the problems of this strategy is the design of a user interface for programmers to describe an applications communication pattern. Here, we revisit the useful high level abstraction Stream Graph [5] that

expresses pipeline parallelism. Stream Graph is a well studied abstraction for expressing pipeline parallelism. There exists previous work on how to extract these graphs from a sequential program by using a profiling approach [6]. Thus we assume that it is easily obtained during application development.

The contributions of our work over prior work are summarized as follows.

- We revisit the stream graph abstraction as a simple interface for describing an applications communication pattern in order to enable the underlying system to tune CPU-accelerator communication.
- We propose a task and memory management library to overlap computation and communication according to stream graphs. The library can be implemented on top of the OpenCL platform. Thus, it is highly portable to any accelerator environment that supports OpenCL and not limited to GPUs.
- We implement and evaluate the proposed library by using the OpenCL runtime API on a CPU-GPU desktop platform. The result shows that our library can achieve performance improvements that are comparable to hand optimized programs, while it hides complex CPU-accelerator communication management from the programmer.

While it is beyond the scope of this paper, we believe that the library can be integrated with previous tools to extract stream graphs [6] in order to realize fully automatic double buffering in heterogeneous platforms.

The paper is organized as follows. Section 2 gives a brief overview of heterogeneous programming model for accelerators and the parallel programming development challenges in such a platform. Section 3 details our proposed library, including the application interface and internal mechanism. We evaluate the prototype system in section 4. Finally, we will refer to related work in section 5 and give our conclusions and future work in section 6.

II. Background

The potential benefit of heterogeneous system for performance and energy efficiency can be large but programmability wall prevents us from fully utilizing such a heterogeneous system in real applications.

Recently, OpenCL [2] is defined as a standard programming model for heterogeneous computing platforms which include GPU-like data-parallel accelerators. In this paper, we assume OpenCL as the underlying programming model for heterogeneous systems because it is the first standard for parallel programming in heterogeneous computing platforms. Here, we give a brief overview of the parallel programming with OpenCL. In addition, we illustrate OpenCL application development flow and describe the communication optimization problem.

```
1   float h_mem1[100],h_mem2[100];
2   cl_mem d_mem1, d_mem2;
3   initialize(h_mem1);
4
5   // Allocate Device Memory
6   d_mem1 = clCreateBuffer(...);
7   d_mem2 = clCreateBuffer(...);
8
9   // Blocking Host -> Device Transfer
10  clEnqueueWriteBuffer(..,d_mem1,..,h_mem1,..);
11
12  // set input and output device memory.
13  clSetKernelArg(kernel,..,d_mem1,..);
14  clSetKernelArg(kernel,..,d_mem2,..);
15
16  // Execute on Device
17  clEnqueueNDRangeKernel(..,kernel,...);
18  clFinish(cmdQueue);
19
20  // Blocking Device -> Host Transfer
21  clEnqueueReadBuffer(..,d_mem2,..,h_mem2,...);
22
23  for(i = 0; i< num;i++) printf("%d\n" ,h_mem2);
```

Figure 1. OpenCL Program Example

A. Heterogeneous Programming Model for Accelerators

Fig. 1 illustrates a simple example of an OpenCL program. First, program code in the line 1-7 initializes both host and device memories for communication between CPU and accelerators. Next, code in the line 10 executes an explicit data transfer from host memory to device memory. Then, code in the line 13-18 sets up and invokes a device kernel task. Finally, code in the line 21 executes an explicit data transfer from device memory to host memory. The results are outputted to a standard output in the line 23.

An OpenCL program is divided into two parts. One is kernel program, written in OpenCL C: a data-parallel programming language for writing tasks executed on OpenCL devices. The other is host program, written in Standard C or C++ with OpenCL Runtime API. It manages kernel program execution and communication between host CPUs and accelerator devices. Kernel program and host program have different memory spaces. Therefore, programmers must handle communication between host and accelerators manually.

B. Parallel Programming Development

Parallel programming is well known for its difficulty. Many researchers have investigated techniques to automate the parallelization process. Although fully automatic parallelization system has not been established, several researches propose techniques to automate a certain part of the parallelization process. Here we divide the parallelization process into three steps and describe each step with related software technique to ease the programming difficulty.

We roughly divide the parallelization process in heterogeneous platforms into three steps. First step is a step to discover parallelization in a program. Usually, input of parallelization process is a sequential program, which is

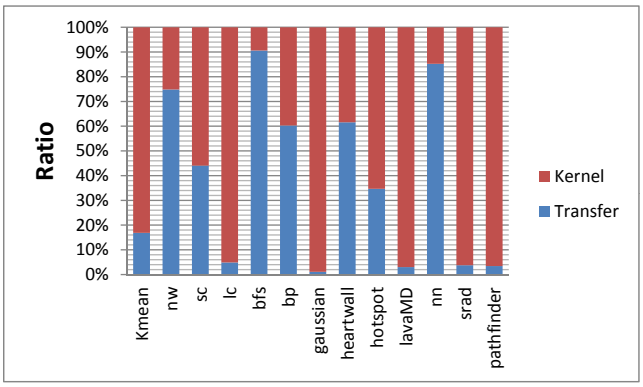

Figure 2. Kernel Execution time vs Communication Time (Rodinia Benchmark Suite)

often a legacy program. We have to identify data-parallel tasks in a sequential program and they should be offloaded to accelerators. Recent work proposes an intelligent profiling tools to discover parallelism in a sequential program [7]. Next step is a step to develop actual parallel programs for the discovered parallel tasks. To develop parallel programs, programmers can utilize a low level API such as OpenCL C, or can make use of automatic parallelizer [4] in some simple cases. Third step is called communication management and optimization [3]. In this step, programmers must manage and optimize the communication between CPUs and accelerators manually. The step introduces another difficulty in accelerator programming.

C. Communication Optimization

Communication overhead between CPUs and accelerators tends to be a performance bottleneck in accelerator programming. Fig. 2 shows breakdown of the execution time in several OpenCL applications in Rodinia Benchmark Suite [8]. In the figure, *Kernel* and *Transfer* denote total execution time of kernel tasks and total data transfer time between CPU and accelerator respectively. We measure the time with the same machine described in chapter 4. The result shows that communication overhead is comparable to execution time of accelerated tasks in many applications. Thus, in order to maximize application performance, programmers must tune communication by taking application behavior into account [9][10]. However, this optimization tends to be complex and error-prone. This difficulty is one of the major reasons to prevent many application developers from effective utilization of accelerators. Considering these situation, it is necessary to ease the difficulty of accelerator memory management and to allow high-quality optimization of memory communication.

III. Communication Library to Overlap Computation and Communication

A. Overview

Our design goal of proposed communication library is summarized as below.

- An API of the library hides details of accelerators memory management from programmers.
- Runtime system of the library overlaps accelerator tasks and CPU-accelerator communication in order to hide the communication latency.

To achieve the goal, our library provides a simple interface for programmers to describe an applications communication pattern among host memories and accelerator tasks. Programmers can utilize this API to write communication management in OpenCL applications.

Fig. 3 illustrates an overview of the system. Applications which use our library consist of kernel program and host program as same as usual OpenCL programs. The kernel program is written in OpenCL C and the host program is written in C/C++ with the library API, which is explained in the following subsections. In the runtime system of our library, the graph analyzer analyzes kernel programs and a stream graph which is described by the library API. It detects required CPU-accelerator communication and the memory manager allocates device memories for the communication. Then, the execution engine executes the kernel programs and the communication in parallel by considering an applications communication pattern. The kernels are compiled by the underlying OpenCL compiler as usual.

The following subsections describe details of our library, including the API to describe stream graphs, and internal mechanism of the runtime system which manages execution of tasks and device memory.

B. Extracting Stream Graph of Kernels

Our library utilizes a stream graph as abstraction of an applications communication pattern. Information of the graph enables the runtime system to overlap computation and communication correctly. Thus, programmers have to extract a stream graph of kernel programs in order to utilize our library. Fortunately, previous work reports that we can easily extract stream graph with a dynamic profiling technique [6]. Although we manually extract a stream graph from programs in the evaluation (section4), it is possible to utilize the system which extracts stream graph from sequential programs as a front-end of our library.

C. API to describe Stream Graph

To tell an applications communication pattern to the library, we prepare a simple API for programmers to express a stream graph of kernel programs. The API is based on the previous streaming programming language [5]. To use the API, programmers register kernel programs and I/O stream

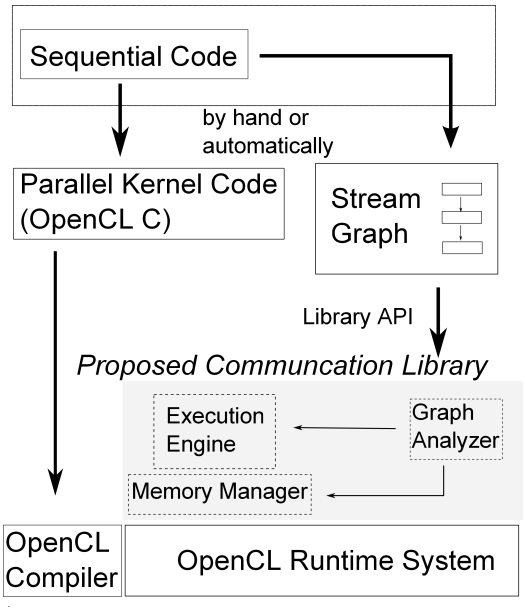

Figure 3. High-level Overview of the Proposed Library

```
1  __kernel void ckMedian(
2    struct p_Median p,
3    __global uchar4* input,
4    __global unsigned int* output,
5    __local uchar4* lmem
6                          )
7  {
8  /* Written in OpenCL C */
9  }
10
11 ClibObject clib;
12 clib.addDevice("NVIDIA CUDA");
13
14 // declare and register I/O Streams
15 IODesc is("is",size*NUM,size,ip);
16 IODesc os("os",size*NUM,size,op);
17 clib.addIStream(is);
18 clib.addOStream(os);
19
20 // set function parameters and register kernel
21 struct p_Median pm;
22 int num_dim = 2;
23 size_t lws[2];
24 size_t gws[2];
25 int lms;
26 pm.pitch  = iBlockDimX +2;
27 pm.width  = WIDTH;
28 pm.height = HEIGHT;
29 lws[0]=iBlockDimX;
30 lws[1]=iBlockDimY;
31 gws[0]=RoundUp(lws[0],WIDTH);
32 gws[1]=RoundUp(lws[1], HEIGHT);
33 lms=(pm.pitch*(iBlockDimY + 2)*4);
34
35 clib.addKernel("ckMedian",sizeof(pm),&pm,
36         num_dim, gws, lws, 0, lms);
37
38 // connect I/O Stream to kernel
39 clib.connectNodes("is","ckMedian",size);
40 clib.connectNodes("ckMedian","os",size);
41
42 // execution
43 clib.execute(cSourceCL,true);
```

Figure 4. An Example of Host Program with the API (Median Filter)

at first. Then, they connect them according to the application communication pattern and execute the application. Fig. 4 shows an example of a host program which uses the API.

Kernels and I/O streams are the nodes of the stream graph. Therefore, the API includes functions to register them to the list of nodes in the graph. To register kernel programs, the API includes *addKernel()* function. The function argument receives execution information about the kernel program whose name is given in a first argument. To register host I/O streams, *IODesc* object is used. Currently our library only supports I/O stream from continuous memory regions.

Edges in stream graphs are communication path among kernels and I/O streams. To describe edges in a stream graph, the API provides *connect()* function. Our library can handle a stream graph whose topology is DAG and not limited to linear. For example, stream graphs with multiple kernels with multiple host I/O Streams are supported. Programmers must specify size of data passed through the communication path.

Finally, *execute()* function is called to execute tasks and then our library receives input data from input stream, executes tasks on input data, and outputs results into output stream.

To use our library, small restrictions are needed to write kernel programs. Fig. 4 also illustrates a kernel function interface for our library (line 1-9). The first argument is a user custom structure. Device memory pointers for input memories and output memories are passed through the second and following arguments. Multiple inputs and outputs are supported. The underlying system automatically passes pointers to proper memory positions on devices according to the stream graph and the execution time step. The final argument can be used to make use of device local memory which is used to optimize kernel programs in OpenCL. This restriction is not fundamental restriction so that we can eliminate the restriction by combining parsing and code generation techniques.

D. Internal Mechanism

The runtime system executes kernel tasks with software pipelining engine. Its core logic is shown as pseudo code in Algorithm 1. In the pseudo code, the outermost loop denotes a single time step of pipelined execution. Each operations select the input data index according to the time $time$ and its pipeline stage index. Operations in the kth stage in pipeline select the $time - k$th input data(in the pseudo code the time $time - k$ is expressed as $time_{eff}$ for each operations). We can eliminate sequential dependency between operations by this mechanism and safely execute each operation in parallel. Synchronization must be done only at the bottom of the outermost loop.

Fig. 5 illustrates flow of processing an applications stream graph. Because the library API hides the device memory management from programmers, the runtime system must detect host-device communication in a stream graph. Then, a stream graph is divided into three pipeline stage in order to overlap communication and computation. First stage includes all host-to-device communications, second stage includes all kernel tasks executed on accelerator, and third stage includes all device-to-host communications. Note that each stage can include multiple operations. The execution engine must take care of the dependency between multiple kernel tasks because they must be executed in correct order in a single time step. Next, memory manager automatically allocates device memories. At the device memories connected to host I/O streams, we must duplicate communication buffers to avoid race condition between computation tasks and communication tasks (double buffering). Communication buffer is implemented as a circular buffer and its correct data location is indexed by the effective time for each operation. On the other hand, device memories which are not connected to host I/O streams do not have to be duplicated because multiple kernels are executed sequentially in one time step.

Mechanism of the execution engine and the memory management will be much more complex when we extend our system for multiple accelerator environment. In this paper, we assume a single accelerator environment and the extension for multiple accelerator environment is our future work.

Algorithm 1 Execution Engine

```
for time = 0; time < time_max; time++ do
  for all k ∈ Kernels do
    time_eff ← time − stage[k]
    if time_eff >= 0 && time_eff < time_max then
      asyncExecution(k, time_eff);
    end if
  end for
  for all m ∈ DeviceMemories do
    time_eff ← time − stage[m]
    if time_eff >= 0 && time_eff < time_max then
      asyncDataTransfer(m, time_eff);
    end if
  end for
  globalsynchronization
end for
```

IV. Experiment

A. Evaluation Setup

We have implemented and evaluate a proposed system as a user library running on OpenCL platform. To evaluate effectiveness of the system, we compare execution time of

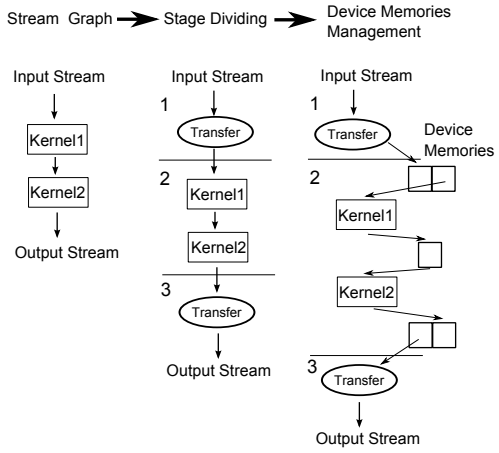

Figure 5. Processing Flow of Stream Graph in the library

OpenCL applications which uses proposed library (*lib*) and OpenCL applications which are written in OpenCL manually with and without double buffering techniques (*handopt* and *original*). In addition to performance evaluation, we compare the number of code lines of host programs in the three versions of programs. We only count the code lines related to memory management and task execution except for initialization and finalization of OpenCL runtime system. In the experiment, we use 4 applications from OpenCL sample applications provided by NVIDIA Corp [11]. All are image processing applications which are often used for streaming data in real time video systems. Table I shows the benchmark names and its input image sizes. These benchmarks are rewritten in the *lib* version, but the rewriting process is simple, finding input stream and output stream and rewrite the parts, so that we believe these step can be automated by existing techniques to exploit pipeline parallelism [6]. Note that when we develop parallel programs from sequential programs, then the rewriting process is not needed. Also, we rewrite the programs in the *handopt* version in order to make use of double buffering techniques which is not used in the original OpenCL programs. All benchmarks are running on the desktop machine, which has 1 CPU and 1 GPU. The detail of the machine is shown in table II.

B. Performance Result

Fig. 6 shows the performance result for each benchmark application. The x-axis is a execution time in microsecond; thus lower is better. In the figure, *lib*, *original*

Table I
Benchmark Applications

Application Name	Image Size
RecursiveGaussian(RGF)	1920x1080
BoxFilter(BF)	1024x1024
SobelFilter(SF)	1920x1080
MedianFilter(MF)	1920x1080

and *handopt* are the execution time of our library version, OpenCL program without double buffering and with manual double buffering respectively. The *handopt* version program achieves performance improvement up to 68% compared to the *original* programs. On the other hand, *lib* programs achieve comparable, sometimes more performance gain to *handopt* programs. The performance gain of *lib* version is up to 91% compared to the *original* programs.

Table III shows the breakdown of execution time of the applications. In the table, *Kernel* column shows the total time for tasks executed on the accelerator, *Transfer* shows the total time for communication between CPU and accelerators, *Lib* shows the time spent in our proposed library code. *Transfer* includes both host-to-device communication and device-to-host communication. The total execution times of *lib* version programs are almost equal to the time of *Kernel* when *Kernel* time is larger than *Transfer* time and vice verse. It indicates that our library realizes almost ideal overlapping of computation and communication. Also, the performance overhead of our library is negligible compared to the execution time of *Kernel* and *Transfer* in these applications. The overhead time is within 2% of total execution time in all applications.

Fig. 7 shows the variation of execution time when we vary the function parameter iRadius in kernel programs of BoxFilter. The parameter controls the window size of image filter operation. Therefore, the bigger value of the parameter means that the amount of computation in the kernel task is larger. In the figure, the x-axis is the execution time and the y-axis is the value of iRadius. The result shows that our library achieves as same level speed up as the *handopt* version code regardless of the amount of computation in the kernel task.

C. Programming Complexity

Although the smaller number of code lines does not directly mean the programmability is higher, counting the number of code lines is convenient way to evaluate programming complexity. We evaluate the number of code lines required to task execution and communication management for each version of programs. In the Table IV, we show the number of code lines for each benchmark application and each version. Although our library hides miscellaneous code related to initialization and finalization of OpenCL runtime system, we do not count the code for fair evaluation. The result shows that we can reduce the number of code lines by half to two-thirds by using our library.

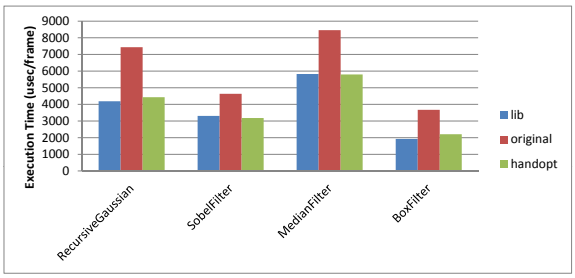

Figure 6. The Execution Time per Frame (microseconds)

Table III
THE BREAKDOWN OF EXECUTION TIME (MICRO SECONDS)

Bench Name	Kernel	Transfer	Lib	Total	Speed Up
RGF(orig)	4015.1	2911.5	-	7436.3	-
RGF(hand)	4026.4	2952.0	-	4432.7	67.8 %
RGF(lib)	4115.7	3186.1	42.5	4187.7	77.6 %
SF(orig)	1404.2	2846.8	-	4637.2	-
SF(hand)	1403.6	2849.6	-	3180.6	45.8 %
SF(lib)	1400.4	2907.6	19.7	3307.6	40.2 %
MF(orig)	5762.9	2639.2	-	8457.0	-
MF(hand)	5725.4	26321.2	-	5795.0	45.9 %
MF(lib)	5779.3	2906.0	19.7	5825.7	45.2 %
BF(orig)	1878.0	1409.2	-	3670.4	-
BF(hand)	1879.0	1431.3	-	2208.8	66.2 %
BF(lib)	1880.8	1441.8	20.5	1924.5	90.7 %

V. RELATED WORK

Many previous researches have investigated techniques and systems to ease the programming complexity in accelerator programming environment. In CGCM [3], static compiler transformation and runtime memory manager work cooperatively to completely hide the memory management of GPU. Also, they propose three optimization techniques to reduce useless communication. However, computation and communication overlapping is not considered in the system. Several works investigate semi-automatic parallelization from annotated programs into accelerator programs. "OpenMP to GPGPU" [4] proposes translation strategy and optimization technique targeting the GPU, but it does not consider communication optimization. Other research investigates Domain Specific Language for accelerators.

Table II
MACHINE SETUP FOR THE EVALUATION

OpenCL Host	Intel(R) Core(TM) i7 CPU X 990
OpenCL Device	Nvidia GeForce GTX 570
CL Driver Version	280.13
OpenCL Platform Name	NVIDIA CUDA (OpenCL 1.1 CUDA 4.0.1)
Operating System	Linux 2.6

Table IV
CODE LINES FOR TASK AND COMMUNICATION MANAGEMENT(LINES)

Bench Name	lib	original	handopt
RGF	53	64	78
SF	25	33	47
MF(lib)	24	32	47
BF(lib)	37	41	54

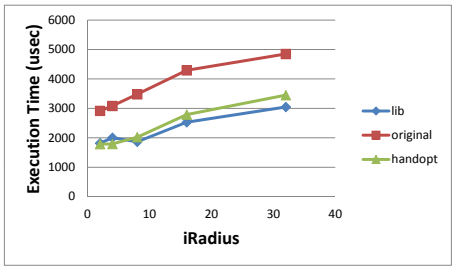

Figure 7. The Execution Time and the Amount of Kernel Computation

Physis [12] is a DSL for stencil computation in HPC applications. It provides an API to express memory access pattern of stencil computation in order to fully automate parallelization and communication optimization on super computers equipped with GPUs.

Stream graph is studied in StreamIt [5] Project. StreamIt is a programming language designed for utilizing pipeline parallelism in applications which have regular data flow. The semi-automatic technique to extract stream graph from applications are proposed by Thies [6] and they demonstrate the effectiveness of pipeline parallelism. They targeted homogeneous multi-core platforms and task parallelism. On the other hand, our library applies pipelining techniques in order to overlap computation and communication in heterogeneous platforms.

VI. Conclusion and Future Work

In this paper, we propose an optimized communication library which hides the complex accelerator memory management from programmers. It provides a small set of interface to describe application communication pattern with stream graph abstraction in order to realize optimized communication between host CPU and accelerators. As the proposed library is implemented as a user library on OpenCL platform, it is highly portable to any other type of accelerator environment. Our evaluation shows that the library successfully hides the details of accelerator memory management from programmers while it achieves comparable speedup to manually optimized program.

Our future work includes an extension for communication between multi-accelerator devices. Under this situation, communication management becomes harder than those under a single accelerator environment. An integration of our library and previous tools which extract a stream graph from a sequential program is also our future work. Through this integration, we can establish a seamless programming support system which brings both high productivity and performance improvement from accelerators.

Acknowledgment

This work was supported by the Grant-in-Aid for JSPS Fellows (23 8062).

References

[1] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, pp. 40–53, March 2008.

[2] *OpenCL Specification*, Khronous OpenCL Working Group Std., Rev. 1.2, 2011. [Online]. Available: http://www.khronos.org/opencl/

[3] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August, "Automatic CPU-GPU communication management and optimization," in *Proc. the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation(PLDI'11)*, New York, NY, USA, 2011, pp. 142–151.

[4] S. Lee, S.-J. Min, and R. Eigenmann, "OpenMP to GPGPU: a compiler framework for automatic translation and optimization," in *Proc. the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP'09)*, New York, NY, USA, 2009, pp. 101–110.

[5] W. Thies, M. Karczmarek, and S. Amarasinghe, "Streamit: A language for streaming applications," in *Proc. International Conference on Compiler Construction(CC'02)*, Grenoble, France, Apr 2002.

[6] W. Thies, V. Chandrasekhar, and S. Amarasinghe, "A practical approach to exploiting coarse-grained pipeline parallelism in C programs," in *Proc. the 40th Annual IEEE/ACM International Symposium on Microarchitecture(MICRO'07)*, Chicago,Illinois, USA, dec. 2007, pp. 356 –369.

[7] S. Garcia, D. Jeon, C. M. Louie, and M. B. Taylor, "Kremlin: rethinking and rebooting gprof for the multicore age," in *Proc. the 32nd ACM SIGPLAN conference on Programming language design and implementation (PLDI '11)*, New York, NY, USA, 2011, pp. 458–469.

[8] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. IEEE International Symposium on Workload Characterization (IISWC '09)*, Washington, DC, USA, 2009, pp. 44–54.

[9] T. Shimokawabe, T. Aoki, T. Takaki, T. Endo, A. Yamanaka, N. Maruyama, A. Nukada, and S. Matsuoka, "Peta-scale phase-field simulation for dendritic solidification on the tsubame 2.0 supercomputer," in *Proc. the 2011 ACM/IEEE conference on Supercomputing (SC'11)*, New York, NY, USA, 2011, pp. 3:1–3:11.

[10] H. K.-H. So, J. Chen, B. Y. Yiu, and A. C. Yu, "Medical ultrasound imaging: To GPU or not to GPU?" *IEEE Micro*, vol. 31, pp. 54–65, 2011.

[11] Cuda toolkit 4.0. NVIDIA Corporation. [Online]. Available: http://developer.nvidia.com/cuda-toolkit-40

[12] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka, "Physis: an implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers." in *Proc. the 2011 ACM/IEEE conference on Supercomputing (SC'11)*, New York, NY, USA, 2011, pp. 1–12.