Integrating Multi-GPU Execution in an OpenACC Compiler

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

Naoya Maruyama†
† Advanced Institute for Computational Science
RIKEN
7-1-26, Minatojima-minami-machi,
Chuo-ku, Kobe, Hyogo, 650-0047 Japan
Email: nmaruyama@riken.jp

***Abstract*—GPUs have become promising computing devices in current and future computer systems due to its high performance, high energy efficiency, and low price. However, lack of high level GPU programming models hinders the wide spread of GPU applications. To resolve this issue, OpenACC is developed as the first industry standard of a directive-based GPU programming model and several implementations are now available. Although early evaluations of the OpenACC systems showed significant performance improvement with modest programming efforts, they also revealed the limitations of the systems. One of the biggest limitations is that the current OpenACC compilers do not automate the utilization of multiple GPUs.**

In this paper, we present an OpenACC compiler with the capability to execute single GPU OpenACC programs on multiple GPUs. By orchestrating the compiler and the runtime system, the proposed system can efficiently manage the necessary data movements among multiple GPUs memories. To enable advanced communication optimizations in the proposed system, we propose a small set of directives as extensions of OpenACC API. The directives allow programmers to express the patterns of memory accesses in the parallel loops to be offloaded. Inserting a few directives into an OpenACC program can reduce a large amount of unnecessary data movements and thus helps the proposed system drawing great performance from multi-GPU systems. We implemented and evaluated the prototype system on top of CUDA with three data parallel applications. The proposed system achieves up to 6.75x of the performance compared to OpenMP in the 1CPU with 2GPU machine, and up to 2.95x of the performance compared to OpenMP in the 2CPU with 3GPU machine. In addition, in two of the three applications, the multi-GPU OpenACC compiler outperforms the single GPU system where hand-written CUDA programs run.

***Keywords*-OpenACC; Multi-GPU**

I. Introduction

Much effort has been made to studies on the potential benefits of the heterogeneous computing platform with GPUs, which has specialized architecture for data-parallel applications. Previous studies showed that GPUs can boost the performance of data-parallel applications [15]. At the same time, they also showed that the programming process with the existing GPU programming environments (CUDA or OpenCL) is much more complicated and time-consuming than that with the parallel programming environments for conventional multiprocessor systems. CUDA and OpenCL can be accepted for expert programmers while not for usual programmers. For wide acceptance of GPU computing, it is essential to develop a more productive programming environment whose level of system abstraction is higher than CUDA and OpenCL.

The industry and a lot of academic researchers have been studying high-level programming models for GPUs to ease the programming complexity [4], [17], [19]. Among those, directive-based GPU programming models [5], [9], [13], [20], [23] have become the center of attention because of their simplicity and their similarity to OpenMP, which is popular in developing parallel applications for multiprocessor systems. OpenACC is the first industry standard of such a directive based GPU programming model, developed in 2011 [2]. Previous work reported promising results of the OpenACC programming model [14], [16], [18], [22].

However, the previous work also pointed out some limitations of the current OpenACC compilers. One of the biggest limitations is that the current OpenACC compilers do not automate the utilization of multiple GPUs. In the application development with CUDA or OpenCL, utilization of multiple GPUs is a popular technique to increase the performance [21]. Utilization of multiple GPUs increases not only the number of cores but also the total amount of GPU memories, so some applications which have large input data are benefited by utilizing multiple GPUs. Thus, for wide acceptance of OpenACC platforms, it is necessary to integrate the multi-GPU execution into the compilers.

To run OpenACC programs written for single GPU on multiple GPUs, the system must manage distributions of tasks and data among multiple GPUs. Also, the system must manage inter-GPU communications, which are necessary in order to keep consistency between replicated data and to handle write access to data on the remote GPU memory. The data movement among the CPU memory and the multiple GPU memories tends to be the performance bottleneck because the performance of the communication bus is limited in the current hardware architectures. In the programming process with CUDA or OpenCL for the multi-GPU system, it is often the case that programmers must optimize the data movement according to the application memory access patterns. Although the current OpenACC API provides a

variety of directives for tuning fine-grained task mapping and memory accesses on single GPU architecture, it lacks directives for enabling the system to optimize communications in the distributed GPU memories.

In this paper, we propose a new OpenACC compiler which can run single GPU OpenACC programs on multiple GPUs. In addition to proposing a system design, we propose a small set of new directives as extensions of the OpenACC API for the multi-GPU system. The newly introduced directives are designed based on the memory access patterns observed in typical GPU friendly applications. The directives allow programmers to express the memory access patterns specific to the tasks in the parallel loops. With the information given by the directives, the compiler and the runtime system can cooperatively optimize the data movement among the distributed memories and can avoid performance degradation from inefficient communications.

We implement and evaluate the prototype system on top of the CUDA platform. The experimental result shows that the prototype system can improve the performance of single GPU OpenACC programs by utilizing multiple GPUs with a small number of the newly introduced directives. The proposed system achieves up to 6.75x performance improvement compared to OpenMP applications in the 1CPU-2GPU machine, and achieves up to 2.95x performance improvement compared to OpenMP applications in the 2CPU-3GPU machine. In almost all cases, the performances of the proposed system are higher than those of hand-written CUDA programs running on single GPU.

The paper is organized as follows. Section II gives the background of the problem. Section III details the basic system design and the proposed extensions of the OpenACC API. We give a description of implementation and optimization techniques to build the proposed system in section IV. We evaluate the system by using three data parallel applications in section V. We discuss the limitation and the future work of the proposed system in section VI. We refer to the related work in section VII and give the conclusion in section VIII.

II. Background

A. OpenACC

The OpenACC API is designed to delegate the low-level GPU programming process to the compiler. It assumes the similar execution model as CUDA or OpenCL. That is the main program runs on the CPU and the data parallel tasks are offloaded to the GPUs. Unlike CUDA and OpenCL, which provide the unique languages to describe parallel tasks for GPUs, the OpenACC API provides OpenMP-like directives to use GPUs. The directives allow programmers to run parallel tasks written in the C or Fortran languages on GPUs only with the several lines of additional codes. It can be thought as a GPU extension of OpenMP, which proves to be effective

```
1  #pragma acc data \
2  copy(x[0:size], c[0:c_size], cIndex[0:size])
3  { while (error < eps)   {
4    error = 0.0;
5    #pragma acc parallel\
6    present(c[0:size], cIndex[0:size])
7    {
8      #pragma acc loop reduction(+:error)
9      for (i=0; i < n; ++i) {
10       for(j=cIndex[i]; j < cIndex[i+1]; ++j){
11         x[i] *= c[j];
12       }
13       error += x[i];
14     }
15   }
16 } }
```

Figure 1. A simple example code with the OpenACC directives.

to improve the productivity of the parallel programming process in homogeneous multiprocessor systems.

Fig. 1 illustrates an example of the OpenACC directives in a simple code example. We can use the *parallel* directive (or the *kernels* directives) (line 5) to identify the code regions to be offloaded on GPUs. Loops annotated with the *loop* directives (line 9) becomes the actual candidates to be transformed into the kernel programs running on GPUs. In order to tune fine-grained task mapping on GPUs, the *gang* clause, the *worker* clause and the *vector* clause can be used with the *loop* directive. Also, like OpenMP, the *reduction* clause can be used for scalar variables (line 7).

In addition to the directives to express parallelism, the *data* directive is important in the OpenACC API. In the current GPU architecture, the GPU memories are physically separated from the system memory and the data movement between the system memory and the GPU memories tends to be the performance bottleneck. Thus, the OpenACC compiler is responsible to optimize the data movement. To avoid the unnecessary data movement, programmers can give the hint information to the OpenACC compilers through the *data* directives (line 1, 2, 6).

B. Using Multiple GPUs

In this paper, we focus on the applications running on the single compute node equipped with multiple GPUs. Computer systems equipped with multiple GPUs have become popular in order to increase the performance and the amount of memory available on GPUs [21]. Fig. 2 illustrates the architecture of such systems. The systems consist of CPUs, GPUs and communication bus. Typically, the number of the GPUs is 1 to 4. The GPUs are connected to the CPUs and other GPUs through the PCIe bus. Due to the limited performance of the communication bus, data movement among the CPUs and the GPUs often becomes the performance bottleneck for the applications running on the multiple GPUs.

In the application development with CUDA or OpenCL, we have to manually manage the multiple GPUs. Because

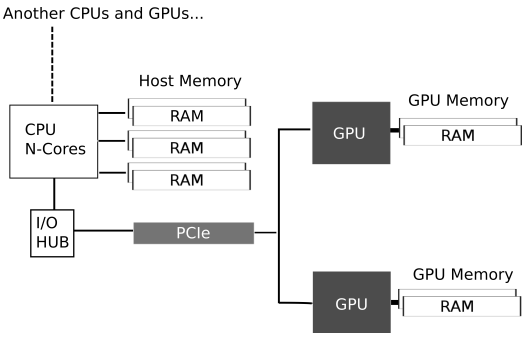


Figure 2. Compute node equipped with multiple GPUs

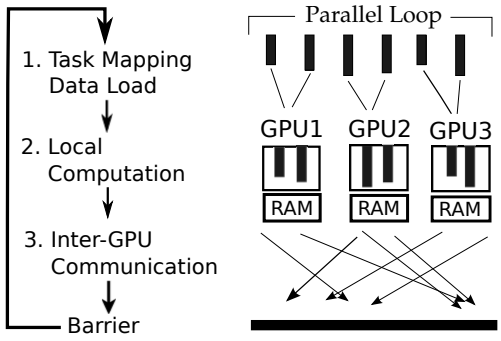


Figure 3. Execution steps of parallel loops in the proposed system

the memory of each GPU is physically separated from each other and the architecture is exposed in CUDA and OpenCL, we can see the system equipped with multiple GPUs as a distributed memory machine. To build and optimize parallel programs on distributed memory machines is well known to be complicated and time-consuming. Programmers have to manually write program code for distributing data among the different memories and have to keep consistency of the data replicated on the multiple memories. It is often the case that the complicated optimization is required to avoid the performance bottleneck at the data movement among the distributed memories.

In terms of the utilization of the multiple GPUs, the current OpenACC compilers do not provide the solution to ease the programming complexity. The programming process to utilize multiple GPUs with OpenACC is almost the same as that with CUDA or OpenCL. Programmers have to explicitly distribute parallel tasks for each GPU and have to manually manage the data movement and synchronization between the GPUs. This limitation is pointed out to be one of the most important challenges for the future OpenACC systems [14], [22]. It is necessary to integrate the multi-GPU execution into OpenACC compilers for wide adoption of OpenACC systems.

III. Basic Design and Directive Extensions

A. Basic Execution Steps on Multi-GPU Environment

We design our multi-GPU OpenACC compiler based on the *bulk synchronous parallel* (BSP) model. Because inter-loop dependencies and critical sections must not exist in the loops annotated with the OpenACC *loop* directive, the loops annotated with the *loop* directive can be executed efficiently on the BSP based systems.

In the proposed system, the parallel loops are executed on the multiple GPUs with three steps. Fig. 3 illustrates the steps. First, the system maps tasks and data to the multiple GPUs. The iteration space of the parallel loop is divided into tasks. The system also determines the mapping of the tasks to the multiple GPUs. At the same time, all the data potentially read by the tasks mapped to each GPU are loaded into the corresponding GPU memory. Next, the actual computations are executed in parallel on the multiple GPUs. Each task is executed by a single GPU thread with the data which are already loaded into the GPU memory. Finally, the system handles the necessary inter-GPU communications, which include the handling of writes to the replicated data, and include the handling of irregular writes whose destinations do not present in the local GPU memory. Then, the global barrier among GPUs occurs and the next parallel loop will be executed.

B. Requirements on directives for multi-GPU environments

Previous researches on directive-based GPU programming models have focused on how to control fine-grained optimizations on a single GPU [12], [20]. They are reflected and adopted in the current OpenACC API. For example, the OpenACC API provides the *gang* clause, the *worker* clause, and the *vector* clause in order to tune the fine-grained task mapping on a single GPU. Also, it provides the *cache* clause in order to make use of the on-chip scratch pad memory.

On the other hand, in order to extend the directive based GPU programming model for the system equipped with multiple GPUs, it is necessary to support directives to control optimizations on the data movement among the distributed memories. Although the current OpenACC provides the *data* directives in order to control the data movement occurred outside the execution of parallel loops, it lacks the interface to control the data movement among the multiple GPU memories inside the parallel loops.

A wide variety of GPU friendly applications exhibits a lot of spatial and temporal locality in the memory access pattern. For example, in computing kernels categorized into MapReduce dwarf [3], which includes typical GPU friendly applications (linear algebra, data mining, monte carlo simulations, and graph processing [10]), each parallel task is executed on its private data elements in arrays iteratively. In the application development with CUDA for the multi-GPU environment, expert programmers make use of the spatial and the temporal locality in order to avoid unnecessary data movement among the distributed memories. The compiler

can do the similar optimizations if it can identify the region of arrays which is used by each iteration of the parallel loop. However, it is not always easy for the compiler to safely analyze the memory access pattern in the application. Thus, it is necessary to provide programming interfaces to express the detailed memory access pattern inside the parallel loops.

In addition, it is necessary to add support for complicated reduction operations inside parallel loops. Here, the complicated reduction operations means the reduction operations whose destinations are the array elements and the access indices are dynamically determined. In the current OpenACC, reduction operations inside parallel loops are supported only for scalar variables. Under this situation, we have to move the complicated reduction operations outside the loop and then execute them on GPUs sequentially or on CPUs. Although this is a problem even in a single GPU environment, the situation gets worse in the system with multiple GPUs. This is because, by using more GPUs, the computation time of the parallel tasks gets shorter but the time required for the reduction operations will not decrease.

In CUDA, expert programmers manually implement the parallel reduction algorithms customized for the memory hierarchy in the multi-GPU environments to avoid the performance bottleneck. The compiler can use the optimized reduction algorithm if it can identify the pattern of the reduction operations. Thus, it is necessary to provide programming interfaces to allow the complicated reduction operations inside the parallel loops.

C. The Directive Extensions

We propose two new directives as an OpenACC extension for multi-GPU environments. One is the *localaccess* directive, which is used to describe the region of data elements read by each parallel task inside the parallel loop. The other is the *reductiontoarray* directive, which is used to tell the compiler about complicated reduction operations to array elements inside the parallel loops.

The details of the directives are as below.

- **localaccess [clause]**
  The directive allows programmers to specify the range of indices for a certain array which can be read in i-th iteration of the loop. When the directive is given, the compiler can aggressively optimize the generated code with the assumption that the i-th iteration of the loop does not read any part of the array outside the specified range of indices. The range of indices must be consecutive. Therefore, it is specified by the pair of the lower bound of the indices and the upper bound of the indices. To express the constant stride read access, the directive supports the *stride(ArrayName[*$stride$*:*$left$*:*$right$*])* clause. This means that the i-th iteration of the loop use the array elements whose indices are from the $(stride * i - left)$ to $\{stride * (i + 1) - 1 + right\}$. To support non-uniform stride access, programmers can specify an index array which contains the lower bounds of the access indices for each iterations of the loop. The *indirect(ArrayName[LowerBounds])* clause can be used for this purpose. We can specify the index array at $LowerBounds$ and can specify the actual array to be read in the loops at $ArrayName$. The $LowerBounds$ array contains the lower bound of the indices used for the $ArrayName$ in the i-th iteration of the loop. The upper bound is given by the lower bound on the $i+1$-th iteration of the loop like the compressed sparse row format used in the field of sparse matrix multiplication.
- **reductiontoarray [clause]**
  We extend the scalar reduction clause for arrays. Unlike the conventional *reduction* clause, the directive is used inside the parallel loops to annotate single reduction statement directly. We can specify the name of the destination array and the range of the indices in the clause. The access index of the destination array can be dynamically determined. The compiler generates the optimized reduction codes which can run efficiently on the multi-GPU environment.

```
1  #pragma acc loop
2  #pragma acc localaccess \
3          stride(x[1:0:0]) stride(b[1:0:0]) \
4          indirect(c[cIndex])
5  for (i=0; i < n; ++i) {
6      for(j=cIndex[i]; j < cIndex[i+1]; ++j){
7          x[i] *= c[j];
8      }
9      #pragma acc reductiontoarray (+:errors[0:e_size]) {
10         errors[b[i]] += x[i];
11     }
12     }
13 }
```

Figure 4. An simple example of the proposed directives.

Example: Fig. 4 illustrates an example of the C program annotated with the proposed directives. In the code, the read access patterns for the array x, the array b and the array c are passed to the compilter through the *localaccess* directive (line 2). On the other hand, the $errors$ array does not have the *localaccess* directive. In this case, the compiler does not aggressively optimize the data movements for the array (detailed in Section IV). Also, we use the *reductiontoarray* directive to tell the compiler that the statement at line 10 must be treated as the reduction operations whose destinations are the elements in the array $errors$. Note that programmers do not have to consider the existence of the multiple GPUs because no task mapping and no data transfer between the multiple GPUs are manually commanded.

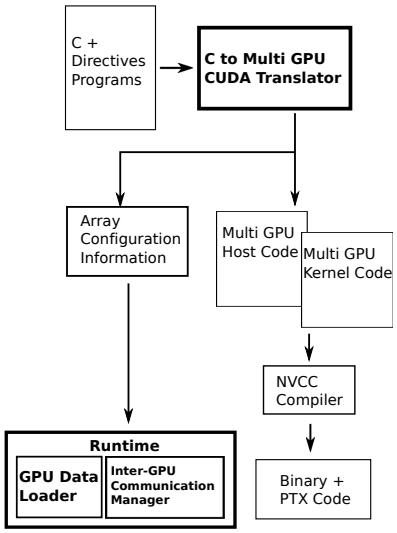


Figure 5. An overview of the prototype system

IV. Compiler and Runtime System

A. System Overview

Fig. 5 illustrates the overview of the prototype system for the proposed multi-GPU OpneACC compiler. The system consists of two components, the OpenACC to CUDA translator and the runtime system. The runtime system is further divided into two parts, which are the data loader and the inter-GPU communication manager.

The translator is designed as a source to source translator. The translator generates CUDA kernel codes and host codes from the C programs annotated with the OpenACC directives and the proposed extension. In addition to generating the CUDA codes, the translator also generates the information which summarizes the memory access patterns for arrays. The information is used in the runtime system in order to optimize the data movement among the distributed memories. The data loader manages the necessary data movement between the CPU memory and the multiple GPU memories according to the OpenACC semantics. To make sure that the execution is correct, all the data which are potentially read by the kernel running on each GPU must be loaded into the corresponding GPU memory before the kernel execution. The inter-GPU communication manager is responsible for handling the inter-GPU communication. It is called just after the kernels executed on every GPUs. It checks the write operation done by the kernel on each GPU and updates the remote GPU memories.

B. Translator

The translator generates the CUDA kernel and host codes which utilize the multiple GPUs. In the current implementation, the parallel loop annotated with the OpenACC *loop* directive is transformed into a CUDA kernel function. The translator replaces the original loop with the call statement for the kernel function. Also, the translator generates the CUDA host code which includes the control codes to initialize the devices, to call the kernel functions, and to control the data movement among the distributed memories.

1) Handling Data Movement: The actual operations for the data movement among the distributed memories are delegated to the runtime system. The translator just inserts the statements to call the runtime functions at the program points where the data movements are required.

2) Task Mapping and Thread Generation: In the current implementation, the tasks in the parallel loop are equally divided among the GPUs. Before the call of the kernel function on each GPU, the host program sets up the number of the thread blocks and the number of the CUDA threads per blocks for the GPU according to the number of the assigned tasks to the GPU.

3) Organizing Array Accesses on GPUs: In the kernel functions, all the indices in the array accesses must be recalculated by considering the data layout on the GPU memory because all the elements of the arrays are not always loaded into the memory. To do so, the host program includes the codes to ask the runtime system about the layout of the arrays among the GPU memories and the codes to pass the information to the arguments of the kernel functions. The translator rewrites the indices of the array accesses in the kernel codes by using the argument. In addition, the translator inserts the additional codes for the inter-GPU communication manager to identify which parts of the arrays are written by the kernel (detailed in Subsection IV-D).

4) Optimizing Kernel Functions: The translator applies two GPU specific optimizations to kernel functions. One is the data layout transformation of two-dimensional arrays for enhancing the coalesced memory accesses. The transformation is applied to device arrays which satisfy the following three conditions: read-only, the access indices are all in affine forms, and the array has the *localaccess* directive. Also, to avoid the performance bottleneck at reduction operations, the translator uses the hierarchical reduction algorithm for the reductions in the kernel functions. At first level, the reduction is done on the shared memory for each thread block. Next, the values are collected among the thread blocks on the same GPU. Finally, the values are transferred and merged between multiple GPUs.

5) Generating Array Configuration Information: The translator generates the array configuration information, which is used by the data loader and the inter-GPU communication manager. The information summarizes the memory access patterns of the arrays. It is generated for every parallel loops and for every device arrays used in the loop. The information contains several attributes of the array, including whether the array is read-only or write-only, the range of the access indices in each loop iteration (if the array has the *localaccess* directive), and the array is the destination of the complicated reduction operations.

C. Data Loader

In OpenACC, the compiler manages the data movement between the system memory and the device memory. The data loader is responsible for guaranteeing the semantics of the GPU memory management while it transparently manages the multiple GPU memories.

The data loader is called at the entrance and the exit points of the *parallel* regions, the *kernels* regions and the *data* regions. In these regions, the data loader is called where the programmers command the data movement by inserting the OpenACC directives such as the *update* directives. The data loader is also called before every kernel calls to load the necessary data into the GPU memories because the necessary data can be changed during the different kernel calls.

In order to avoid unnecessary data movement among the multiple GPU memories, the data loader makes use of two different policies to load the arrays into the GPU memories. One policy is the replica-based policy. With the policy, all the data elements in the array are replicated to all the GPU memories. This is the default placement policy and the data loader place the arrays without the *localaccess* directives according to the replica-based policy. The other policy is the distribution-based policy. In the policy, the array is divided into sub-arrays and only the sub-array actually accessed by the GPU is loaded into the corresponding GPU memory.

With the distribution-based policy, the arrays require less amount of data movement and less amount of device memory footprints than arrays with the replica-based policy. However, because the runtime system must know the precise memory access patterns to safely distribute the arrays, only the arrays with the *localaccess* directives can be managed with the distribution-based policy.

Note that the data loader can avoid additional data movement before the kernel calls when the read memory access pattern in the next kernel call is the same to that in the previous kernel call. This is common in iterative algorithms, the same parallel loop is executed many times, as seen in the applications evaluated in Section V.

D. Inter-GPU Communication Manager

The inter-GPU communication manager is called just after the kernel functions executed on the GPUs. It handles the necessary data exchanges between the multiple GPU memories, including the update operations at the writes to the replicated data and the remote write operations to the data which exist only in the remote GPU memory. To maximize the performance at the communication step, the communication manager directly exchanges the data between the GPU memories and the communications are executed asynchronously.

1) Replicated Array: When the write accesses occur in the replicated arrays, the inter-GPU communication manager must update other copies on the different GPU memories to keep consistency.

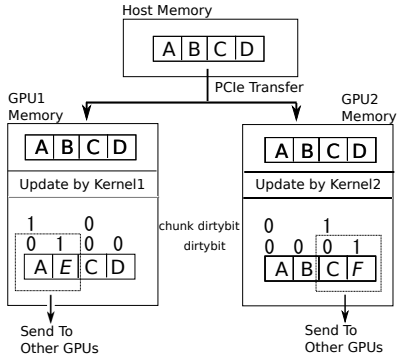


Figure 6. Two-level dirty-bit mechanism to keep consistency of the replicated arrays.

To do this, the manager prepares the dirty bit arrays on the GPU memories for the replicated arrays. In order to identify which elements of the array are written in the kernel execution, the translator inserts the additional operations to turn on the dirty bits for every writes accesses to the replicated arrays. However, with the single level dirty bits, the manager has to transfer all the array data, including the clean elements and the dirty bits, to other GPU memories because the manager cannot check the contents of dirty bits efficiently on the sender GPU. It degrades the performance of the inter-GPU communications.

To solve the problem, we use the two-level dirty-bit mechanism. It is illustrated in Fig. 6. A dirty bit array is subdivided into chunks whose sizes are constant. Each chunk also maintains single bit, which indicates all the dirty bits in the chunk are clean. The bit is used as the second level dirty bit. The translator can add the codes to turn on the second level dirty bit in the kernel codes. With the second level dirty bits, the inter-GPU communication manager avoids unnecessary data transfers at the chunks which have no dirty data. The optimal size of the chunks is dependent both on the applications and the hardware characteristics. In the evaluation at Section V, we experimentally choose 1MB to the chunk size of the second level dirty bit arrays.

2) Distributed Array: In the case of the arrays which are subdivided and distributed among multiple GPUs, we have to handle irregular writes to the data which do not exist in the local GPU memory. To correctly handle the writes to the data on the remote GPU memories, the manager must know that which write accesses misses on the local GPU memory in the previous kernel execution. To tell the manager about the write miss, the translator insert the check codes for every write accesses on the distributed arrays to identify the write misses. When the write access causes write miss, the pair of the written data and the destination address are temporarily buffered into the system buffers on the local GPU. After the kernel execution, the inter-GPU communication manager transfers the records of the write misses in the system buffers to the remote GPU memories where the destination

Table I
THE MACHINE SETUPS FOR THE EVALUATION

Desktop Machine	
CPU	Intel Core i7 x 1 (6core, Hyper Threading)
GPUs	Nvidia Tesla C2075 x2
Supercomputer Node	
CPU	Intel Xeon x 2 (12=2x6core, Hyper Threading)
GPUs	Nvidia Tesla M2050 x3

exists. Then, the communication manager calls the CUDA kernels to complete the write access to the remote GPU. If the compiler can statically analyze that the write address is always within the range described by the *localaccess* directive, we can eliminate the check code to avoid the additional performance overhead.

V. EVALUATION

A. Methodology

We implemented the prototype system of the proposed compiler. The translator is implemented by using ROSE compiler infrastructures developed at Lawrence Livermore National Laboratory [1]. It also uses parts of open-source C to CUDA translators for a single GPU machine [20]. The runtime system is implemented with C++ on top of the CUDA 4.0 platform. Although we implemented the proposed system on top of the CUDA platform, we can build the same system on top of the OpenCL platform as well.

We evaluated the proposed compiler system in two different platforms. One is the desktop machine equipped with two GPUs. The other is the thin-node of TSUBAME2.0 supercomputer at Tokyo Institute of Technology. Each platform differs by the type and number of CPUs and GPUs. Also, the performances of communication buses are different. The details of the platforms are shown in the Table I.

We use three benchmark applications *BFS*, *MD*, and *KMEANS* selected from rodinia [6] and shoc [7] benchmark suites. They exhibit different inter-GPU communication characteristics. *BFS* is highly memory intensive with a lot of irregular writes. It is one of the most difficult applications to be efficiently executed in multiple GPU environments. On the contrary, *MD* requires no inter-GPU communications. *KMEANS* is in the middle of these two applications. It requires small amont of inter-GPU communications due to the existence of reduction operations whose destination is the array which is used in all the GPUs. We summarize the details of the applications in Table II.

We compare the performance of the several versions of the applications. The versions are as follows.

- OpenMP: The programs are written with OpenMP. They are compiled by the gcc compiler with the O2 optimization flag. The number of running threads is set to 12 in the Desktop Machine and 24 in the supercomputer node.
- PGI OpenACC Compiler: The programs are written with OpenACC and are compiled by the PGI OpenACC compiler with the O2 optimization flag.
- CUDA: The programs are written in CUDA. They are compiled by the nvcc compiler with the O2 optimization flag. They can be executed only in a single GPU.
- Proposal: The programs are written with the OpenACC API with the proposed extensions (the *localaccess* directive and the *reductiontoarray* directive). The generated CUDA codes are compiled by the nvcc compiler with the O2 optimization flag. They are executed in one, two, and three GPUs.

B. Results

1) Overview: Fig. 7 shows the performance of the proposed system compared to other versions of the programs. The y-axis is the relative performance normalized to the pefromance of the OpneMP versions. In the figure, the number of GPUs used in the program is denoted within the parentheses in each label. We measures the execution time spent in the parallel regions, including the time spent on the CPU-GPU communications and the GPU-GPU communications. We divide the execution times of the OpenMP versions by the execution time of each version in order to get the relative performance.

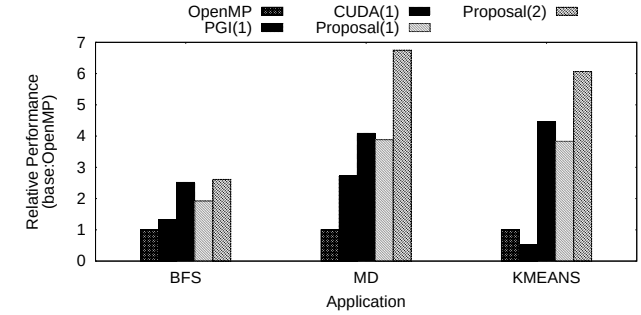
First, by using GPUs, the proposed system achieves higher performance than the baseline OpenMP versions in all cases except for *bfs* in the supercomputer node. Plus, by using multiple GPUs, the proposed system achieves higher performance than the hand-written CUDA version with a single GPU. Even in the case of *bfs* in the supercomputer node, in which the proposed system does not exhibit performance improvement, the applications with the proposed system can benefit from the larger amount of GPU memory by using multiple GPUs. Considering that much less porting effort is paid in the proposal versions than in hand-written CUDA versions, the results demonstrate the effectiveness of the proposed system.

2) Analysis: Fig. 8 shows the breakdown of the execution time in the proposed system. Each execution time is divided into the time spent on the data transfer between GPUs and GPUs (*GPU-GPU*), the time spent on the data transfer between CPU and GPUs (*CPU-GPU*), and the actual execution time of the GPU kernels (*KERNELS*). Each execution time is normalized to the total execution time in the single GPU execution.

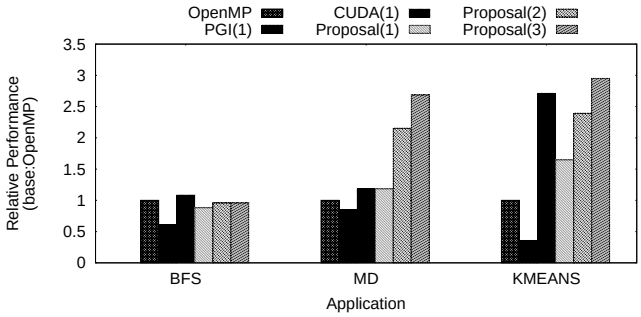
In Fig. 8, we can see that the times spent on the data transfer between CPUs and GPUs (*CPU-GPU*) are the main reason that prevents us from achieving linear speed up to the number of GPUs. Because the proposed system incurs negligible overheads in the data transfer between the CPU and the GPUs (will be shown in Fig. 9), the limitation is originated in the application characteristics. Thus, the data transfer between CPUs and GPUs still prevents the linear

Table II
A: TOTAL DEVICE MEMORY USAGE IN SINGLE GPU EXECUTION, B: # OF PARALLEL LOOPS, C: # OF KERNEL EXECUTIONS, D: # OF ARRAYS WITH LOCALACCESS DIRECTIVE / # OF ARRAYS USED IN PARALLEL LOOPS

Application	Source	Description	Input	A	B	C	D
MD	SHOC	Simulation	73728 Atom	39.8MB	1	1	2/3
KMEANS	Rodinia	Clustering	kdd_cup	69.2MB	2	74	2/5
BFS	SHOC	Graph Traversal	5M node	444.9MB	1	10	2/3

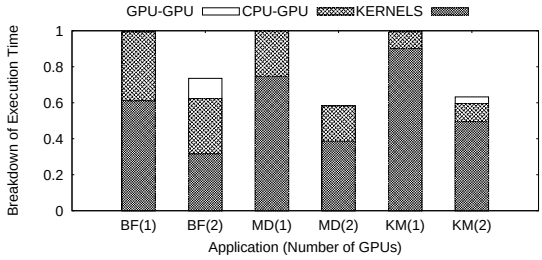


(a) Desktop

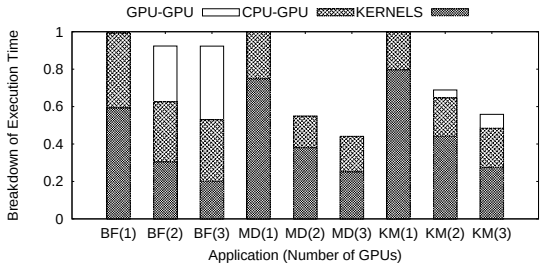


(b) Supercomputer node

Figure 7. Performance Comparison. Normalized to the OpenMP versions.



(a) Desktop



(b) Supercomputer node

Figure 8. The breakdown of the execution time. Normalized to the total execution time in single GPU execution.

speed up even when we manually make use of multiple GPUs with CUDA or OpenCL. To solve the problem, we have to rewrite the applications by redesigning the algorithm. However, this is beyond the scope of this work.

In *bfs*, which yields little speedup by using the multiple GPUs in the supercomputer node, we can observe that the time for inter-GPU communication become the performance bottleneck in two or three GPU executions. To reduce the amount of the communications, we can apply the advanced communication optimization based on graph splitting in the manual development in CUDA or OpenCL. However, the proposed system does not support such an optimization. To integrate the optimization into the proposed system is our future work.

Next, we show the memory overhead caused by the data replication and the temporary buffers used by the runtime system. Fig. 9 shows the device memory usages in the applications with the proposed system. The bars in the figure indicate the amount of memory used to store the user data (*User*) and the amount of memory used in the runtime system (*System*). The values are normalized to the total device memory usage in the single GPU execution, where no system memory or no replicated data exist. Note that the amount of the data transfer between the CPU memory and the multiple GPU memories is roughly proportional to the size of the *User* memory in the evaluated applications because we avoid CPU-GPU communication during the parallel regions in the implementations.

In Fig. 9, we can see that the amount of the *User* memory, which includes the memory for the replicated data, does not increase significantly even with the multi-GPU executions. If the data loader replicates all the data on every GPU memory, it would increase in proportion to the number of GPUs. However, with the memory access patterns provided through the *localaccess* directives, the proposed system can avoid this situation by making use of the access locality in the arrays. On the other hand, the runtime system consumes some amount of the device memory in order to handle the inter-GPU communications. The amount of device memory used in the runtime system is larger in the applications in which more inter-GPU communications are needed, such as

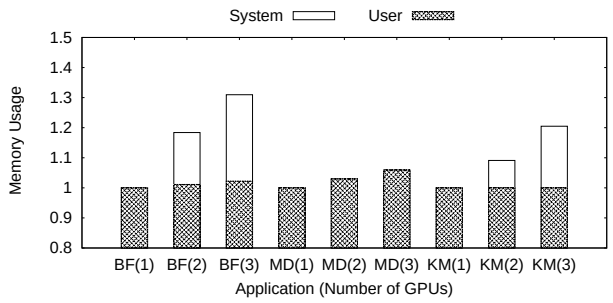


Figure 9. The usage of the device memory. Normalized to the amount of the total device memory used in single GPU execution.

bfs. However, the overhead is less than 30% even in *bfs*. Thus, the extra memory overheads in the proposed system do not decrease the benefits of larger available memory with the multi-GPU execution.

VI. Limitations and Future Work

The most important limitation of the current prototype system is that the applicability of the communication optimization is limited to one dimensional arrays though it is possible to execute the applications which uses multi-dimensional arrays. Due to the limitation, the performance improvement from the current system is not large for some important applications, e.g. stencil computations. In the stencil computations, it is important to make use of the nested parallelism and optimize the data movement in the multidimensional arrays. However, the limitation is not originated in the system design. Theoretically, we can extend the *localaccess* directives to support multidimensional arrays and we can apply these kind of optimizations in the proposed system. This is our future plan to enhance the proposed system. Also, the current prototype does not have the capability to make use of inter-node multiple GPUs. Again, the limitation is not originated in the system design. To extend the system for inter-node multiple GPUs is also our future work.

VII. Related Work

Previous studies evaluate the various aspects of the current OpenACC compilers [16] [18] [22]. Wienke et al. [22] evaluate and compare the performance of the realistic GPU applications written in OpenACC and OpenCL. They reported that OpenACC can achieve good performance gains with modest programming cost compared to OpenCL. Levesque et al. [16] investigate the hybrid implementation with MPI and OpenACC in the GPU cluster system. They use OpenACC for the intra-node application development and use MPI to make use of the multiple nodes in the clusters. Reyes et al. [18] have developed an open sourced OpenACC compiler. Lee et al. [14] compared the current OpenACC compiler to the previous directive based GPU compilers in performance, functionality, and programmability. They pointed out the limitations in the current directive based GPU compilers, which includes the limited scalability to the available number of GPUs.

Kim et al. [11] have developed the OpenCL system which can make use of multiple GPUs from single GPU OpenCL programs. The concept is similar to ours. That is building the illusion of the single GPU memory on top of the multiple GPUs. We integrate the concept with an OpenACC compiler and propose a small set of directives to control the communication optimizations. The integration is essential because we can design compiler and runtime system in a holistic way in order to make use of the application memory characteristics. OmpSs has been presented as a directive based programing model to execute single GPU CUDA programs on GPU clusters [8]. OmpSs provides directives to express data dependency between parallel tasks. The directives allow programmers to express the detailed memory access pattern, so they are similar to the *localaccess* directives proposed in this paper. However, the OmpSs system does not include compilers to generate programs running on the GPUs. Hence the programmers must manually write the CUDA codes and divide the applications into parallel tasks. Also, their inter-GPU memory manager relies on inefficient software cache mechanisms. StarPU [4] provides another high level programming interface for the multi-GPU environments. In the programming model, programs must be rewritten with a data-flow based API, so we cannot incrementally port the existing programs to the multi-GPU environments.

VIII. Conclusion

In this paper, we present an OpenACC compiler system with the capability to execute single GPU OpenACC programs on multiple GPUs. By orchestrating the compiler and the runtime system, we could build an efficient multi-GPU directive-based compiler on top of the current platforms. In order to enable the advanced communication optimizations in the proposed system, we also propose a small set of directives as an extension to the current OpenACC standards. The directives allow programmers to express the memory access patterns in the parallel loops with a few lines of additional codes.

We implemented and evaluated the prototype system. The prototype system achieves up to 6.75x of the performance improvement compared to OpenMP in a machine with one CPU and two GPUs, and achieves up to 2.95x of the performance improvement to OpenMP in a machine with two CPUs and three GPUs. Supporting the optimizations on multidimensional arrays is our future work to widen the applicability of the proposed system. To explore the possibility to extend the proposed system on clusters is another future work.

IX. Acknowledgment

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

References

[1] ROSE Compiler Infrastructures. http://rosecompiler.org/.

[2] OpenACC Directives for Accelerators. http://www.openacc-standard.org/, 2011.

[3] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[4] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198, Feb. 2011.

[5] F. Bodin and S. Bihan. Heterogeneous Multicore Parallel Programming for Graphics Processing Units. *Scientific Programming.*, 17(4):325–336, Dec. 2009.

[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the IEEE International Symposium on Workload Characterization, 2009. IISWC 2009.*, pages 44–54, Oct.

[7] A. Danalis, G. Marin, C. Mccurdy, J. S. Meredith, P. C. Roth, K. Spafford, and J. S. Vetter. The Scalable HeterOgeneous Computing (SHOC) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU3)*, pages 63–74, 2010.

[8] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. OmpSs: A Proposal for Programming Heterogeneous Multi-Core Architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[9] T. D. Han and T. S. Abdelrahman. hiCUDA: a High-level Directive-based Language for GPU Programming. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 52–61, 2009.

[10] W. W. Hwu. *GPU Computing Gems Emerald Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011.

[11] J. Kim, H. Kim, J. H. Lee, and J. Lee. Achieving a Single Compute Device Image in OpenCL for Multiple GPUs. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 277–288, 2011.

[12] S. Lee and R. Eigenmann. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2010*, pages 1–11, Nov.

[13] S. Lee, S. Min, and R. Eigenmann. OpenMP to GPGPU: a Compiler Framework for Automatic Translation and Optimization. *ACM SIGPLAN Notices*, 44(4):101–110, Feb. 2009.

[14] S. Lee and J. S. Vetter. Early Evaluation of Directive-based GPU Programming Models for Productive Exascale Computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 23:1–23:11, 2012.

[15] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100X GPU vs. CPU Myth: an Evaluation of Throughput Computing on CPU and GPU. *ACM SIGARCH Computer Architecture News*, 38(3):451–460, June 2010.

[16] J. M. Levesque, R. Sankaran, and R. Grout. Hybridizing S3D into an Exascale Application using OpenACC: an Approach for Moving to Multi-petaflops and Beyond. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 15:1–15:11, 2012.

[17] 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 *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2011*, pages 1 –12, nov. 2011.

[18] R. Reyes, I. Lpez-Rodrguez, J. Fumero, and F. Sande. accULL: An OpenACC Implementation with CUDA and OpenCL Support. In *Proceedings of the Euro-Par 2012 Parallel Processing*, volume 7484 of *Lecture Notes in Computer Science*, pages 871–882. Springer Berlin Heidelberg, 2012.

[19] A. Sidelnik, S. Maleki, B. Chamberlain, M. Garzaran, and D. Padua. Performance Portability with the Chapel Language. In *Proceedings of the IEEE 26th International Parallel Distributed Processing Symposium (IPDPS), 2012*, pages 582–594, May.

[20] D. Unat, X. Cai, and S. B. Baden. Mint: Realizing CUDA Performance in 3D Stencil Methods with Annotated C. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 214–224, 2011.

[21] J. Vetter, R. Glassbrook, J. Dongarra, K. Schwan, B. Loftis, S. McNally, J. Meredith, J. Rogers, P. Roth, K. Spafford, and S. Yalamanchili. Keeneland: Bringing Heterogeneous GPU Computing to the Computational Science Community. *Computing in Science Engineering*, 13(5):90–95, Sept.-Oct.

[22] S. Wienke, P. Springer, C. Terboven, and D. Mey. OpenACC: First Experiences with Real-World Applications. In *Proceedings of the Euro-Par 2012 Parallel Processing*, volume 7484 of *Lecture Notes in Computer Science*, pages 859–870. Springer Berlin Heidelberg, 2012.

[23] M. Wolfe. Implementing the PGI Accelerator model. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 43–50, 2010.