Towards Automating Multi-dimensional Data Decomposition for Executing a Single-GPU Code on a Multi-GPU System

Ryotaro Sakai, Fumihiko Ino, and Kenichi Hagihara
Graduate School of Information Science and Technology, Osaka University
1-5 Yamadaoka, Suita, Osaka 565-0871, Japan
Email: {r-sakai, ino}@ist.osaka-u.ac.jp

***Abstract*—In this paper, we present a data decomposition method for multi-dimensional data, aiming at realizing multi graphics processing unit (GPU) acceleration of a compute unified device architecture (CUDA) code written for a single GPU. Our multi-dimensional method extends a previous method that deals with one-dimensional (1-D) data. The method performs a sample run of selected GPU threads to decompose large data into small segments, which avoid exhaustion of GPU memory. As compared with the previous method, our multi-dimensional method produces smaller segments, so that it saves GPU memory consumption and reduces the amount of CPU-GPU data transfer. As a result of experiments using matrix multiplication, the presented method consumed less GPU memory compared with that of the previous method, and thereby successfully processed 29 times larger matrices as long as the matrices fit into CPU memory. However, we found that index transformation needed for multi-dimensional decomposition dropped the effective performance by 28%.**

1. Introduction

The graphics processing unit (GPU) [1], which is capable of running millions of parallel threads, is an accelerator device for graphics applications. This emerging device typically achieves a 10 times speedup over the CPU not only for traditional graphics applications [2] but also for general-purpose applications [3] in the fields of fluid dynamics, machine learning, and bioinformatics [4]. One drawback of this accelerator device is that the capacity of GPU memory is at most 16 GB, which restricts the problem size to be handled on the device. Consequently, multi-GPU systems are usually deployed to tackle the same problem size as that solved by CPU-based implementations.

In general, multi-GPU applications require more development efforts than single-GPU applications. For example, the compute unified device architecture (CUDA) [5], or a widely used framework for the NVIDIA GPU [1], requires application developers to largely rewrite their single-GPU code [6]; they must find out the best data decomposition scheme that maximizes the performance on the device. Moreover, developers have to transform the indexing scheme between the original data and the decomposed data, which usually have different indexes due to data decomposition. Thus, multi-GPU programming is more complicated and time consuming than single-GPU programming.

To facilitate multi-GPU programming, Kim *et al.* [7] presented a runtime framework that realized a single compute image in OpenCL [8] for multiple GPUs. Their framework assumes that the input single-GPU code satisfies a constraint but there is no need to modify the code for multi-GPU execution. To automate data decomposition, their runtime performs a sample run of the GPU code (i.e., the kernel function), which estimates the memory region that will be accessed by GPU threads. This sample run is conducted on the CPU before invoking the GPU code; the constraint mentioned above reduces the number of GPU threads to be simulated for estimating the memory access behavior of the entire GPU thread. According to this estimation, the data are decomposed into smaller segments.

Although this estimation is useful for automating data decomposition, the memory access region is specified with a one-dimensional (1-D) addressing scheme, as depicted in Fig. 1. With this 1-D scheme, segments for multi-dimensional data can include redundant elements that will not be accessed during execution. For example, the segment in Fig. 1 includes unaccessed elements in several columns because its region is specified only along the vertical axis. A 2-D addressing scheme, which restricts columns and rows, is necessary to exclude such unaccessed columns from the segment.

In this paper, we present a data decomposition method for multi-dimensional data that cannot be entirely stored in the GPU memory. The presented method minimizes redundant elements in produced segments. To achieve this, the method extends the previous method [7] such that it specifies the memory region with a multi-dimensional addressing scheme where the necessary region is restricted with every dimension that composes the data. We also show that a naive extension of the previous method fails to correctly decompose multi-dimensional data. The key idea for obtaining correct decomposition is precomputation of specific operations, which temporarily replaces variables with constant values before a sample run of the GPU code. Similar to the previous method [7], our method is applicable to arbitrary kernels in which memory references are given as affine functions of the thread and thread block indexes.

The paper is structured as follows. Section 2 introduces

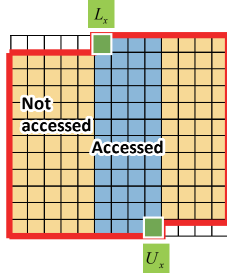

Figure 1. Memory access region estimated by the previous method [7], which deploys a 1-D addressing scheme. Blue columns in a 2-D array are accessed during execution, whereas the estimated region is given by elements enclosed with red lines. In this case, the estimated region, specified with two 1-D addresses L_x and U_x, includes redundant elements that will not be accessed.

related studies on data decomposition methods for GPU codes. Section 3 summarizes the previous method [7] and some issues that must be solved for multi-dimensional data decomposition. Section 4 then presents our method and Section 5 shows some experimental results. Finally, Section 6 summarizes the paper with future work.

2. Related Work

Müller *et al.* presented the CUDA system architecture (CUDASA) framework [9], capable of assisting application developers in developing multi-node, multi-GPU systems. This framework extended the hierarchies of CUDA threads and memories to allow a CUDA-like code to run on a multi-node, multi-GPU system. Kim *et al.* developed the SnuCL runtime system [10] capable of running multi-GPU code on a multi-node system with hiding inter-node communication. These frameworks significantly reduced development efforts needed for load balancing over multiple devices. However, application developers have to explicitly decompose large data in their application code.

As for automated data decomposition, Luk *et al.* proposed the Qilin system [11], which automated data decomposition by applying data dependence analysis to the GPU code. Similarly, Ji *et al.* [12] developed a runtime library, named Region-based Software Virtual Memory (RSVM), which hid data decomposition from application developers. These studies automated data decomposition but enforced application developers to rewrite their code with unique programming models and application programming interfaces (APIs). Such unique description increases programming efforts because code modification is needed to (1) transfer data between the CPU and GPU, (2) invoke the GPU code, and (3) access variables in the GPU code.

To hide data decomposition from application developers, Lee *et al.* [13], [14] presented a programming framework that can run a single-GPU code on a multi-GPU system. Similar to Kim's decomposition method [7], their framework analyzed the memory access behavior to automate data decomposition. The difference over Kim's method is that the entire thread is examined, so that various applications can be applied without any constraint; however, the analysis overhead, which is larger than Kim's method, is an issue that must be resolved for large-scale applications.

Some researchers [15], [16] tried static approaches to run a single-GPU code on a multi-GPU system. For example, Cabezas *et al.* [15] presented a compiler analysis approach that automated parallelization of kernels in shared-memory multi-GPU nodes. As compared with Lee's framework [14], their approach increased the maximum problem size that can be executed on a multi-GPU system. However, these static approaches assume that the data size is smaller than the total capacity of GPU memories. In contrast, our sampling approach can deal with out-of-core data, which cannot be entirely stored in GPU memories.

The CUDA provides some functionalities to facilitate large-scale computation on GPUs. Unified memory [5] provides a shared memory space to hide data transfer between the CPU and GPU. Mapped memory [5] allows GPU threads to directly refer data stored on the CPU memory. These functionalities are useful to write an application code without explicit data decomposition. However, the achieved performance is usually not satisfactory due to the limitation of static analysis; on-demand data transfer between the CPU and GPU degrades the performance.

Directive-based programming, such as OpenMP [17] and OpenACC [18], is an attractive approach to port CPU code onto a parallel machine with low efforts. In this approach, application developers have to add some compiler directives into their sequential code, which is then compiled with a parallelization compiler. Anonymous *et al.* [19] extended OpenACC directives to realize GPU-based out-of-core stencil computation accelerated with a software pipeline. OpenMPC [20] extended the OpenMP specification such that an OpenMP-like code can be accelerated on a CUDA-compatible GPU. Sabne *et al.* then automated pipelined execution of large data for an OpenMPC code. These directive-based approaches minimize programming efforts but some device-specific optimization cannot be applied due to their high-level description. By contrast, our CUDA-based approach allows device-specific optimization to be explicitly described in the code.

3. Kim's Previous Method

Given an OpenCL code [8] written for a single-GPU system, the previous method [7] accelerates the code on a multi-GPU system. The method automates data decomposition by assuming that the input code satisfies a certain constraint. Figure 2 shows a processing flow of the previous method. The method performs a sample run of a GPU code, namely a kernel function, before executing the code on the GPU. This sample run on the CPU estimates the memory access region for every task. A task here consists of computation assigned to a set of work-groups (i.e., thread blocks in the CUDA context), and different tasks are data-independent based on the constraint of the OpenCL programming model. According to this estimated region, large data are decomposed into small segments, each necessary for an execution

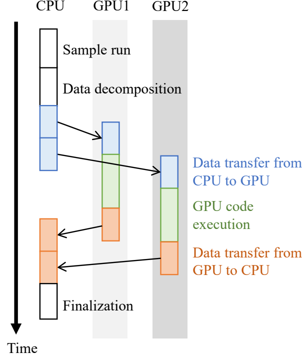

Figure 2. Processing flow of the previous system [7].

of a task. Thus, the segment size depends on the number of work-groups associated with a task. The generated segments are then transferred from CPU memory to GPU memory to accelerate their corresponding tasks on the GPU. Data-independent tasks are assigned to devices with a flexible task scheduling mechanism.

Note that the constraint assumed above is necessary to reduce the number of GPU threads to be sampled before data decomposition. Suppose here that array `A` is to be decomposed into segments, and its n-th element `A[n]` is accessed in the GPU code, as shown in Fig. 3. Then, the constraint is that the index `n` of the target array is given by an affine function of the work-item ID (l_1, l_2, l_3) and the work-group ID (l_4, l_5, l_6) (i.e., the thread index and the thread block index, respectively, in the CUDA context). That is,

$$f(l_1, l_2, \ldots, l_6) = \sum_{i=0}^{6} (a_i \times l_i), \tag{1}$$

where a_i $(0 \leq i \leq 6)$ is a constant value common to all GPU threads, and $l_0 = 1$ is a dummy variable that simplifies description. For an example of Fig. 3, we have $a_0 = 0$, $a_1 = 1$, $a_2 = s$, $a_3 = w$, $a_4 = sw$, $a_5 = wh$, and $a_6 = swh$, each confirmed as a constant value when invoking the GPU code. In this case, Theorem 1 can be applied to Eq. (1).

Theorem 1. *Given affine function $f(l_1, l_2, \ldots, l_n)$ of n variables, its least upper bound (greatest lower bound) can be obtained when all $l_1, l_2, \ldots, l_n$ get the maximum values (minimum values, respectively).*

Therefore, for the GPU code that satisfies the constraint mentioned above, the memory access region of the entire thread can be estimated by performing a sample run of selected threads that have the minimum/maximum ID $l_1, l_2, \ldots, l_6$; there is no need to sample all threads.

Figure 4 shows a sampling code generated for the GPU code presented in Fig. 3. This code runs on a CPU to estimate the memory access region for every task. The sampling code records the memory access region of the threads specified with their work-item ID and the work-group ID, `_l1`, `_l2`, ..., `_l6`.

```
1  __kernel void func(__global float* A, __global float* B,
2                     float d, int s, int w, int h)
3  {
4      int i = get_local_id(0) + get_group_id(0)*s;
5      int j = get_local_id(1) + get_group_id(1)*s;
6      int k = get_local_id(2) + get_group_id(2)*s;
7      int n = i + j*w + k*w*h;
8
9      B[n] = d * A[n];
10 }
```

Figure 3. An example of GPU code.

```
1  void _samp_func(int _l1, int _l2, int _l3,
2                  int _l4, int _l5, int _l6,
3                  float* A, float* B,
4                  float d, int s, int w, int h)
5  {
6      int i = _l1 + _l4*s;
7      int j = _l2 + _l5*s;
8      int k = _l3 + _l6*s;
9      int n = i + j*w + k*w*h;
10
11     ACCESS_READ(A, n); ACCESS_WRITE(B, n);
12 }
```

Figure 4. An example of sampling code generated for the GPU code presented in Fig. 3.

As shown in Fig. 1, data segments of the previous method can include redundant elements for multi-dimensional data because it specifies the memory region with a 1-D addressing scheme. Such redundant elements not only waste the GPU memory but also increase the CPU-GPU data transfer time, which can degrade the entire performance.

3.1. Issues for Multi-Dimensional Decomposition

One can easily apply the previous method to multi-dimensional data by mapping a 1-D memory address, namely the output of the previous method, to a multi-dimensional memory address. That is, as shown in Fig. 5(a), a data segment can be produced as a bounding rectangle whose corners are specified with the least upper bounds, U_x and U_y, and the greatest lower bounds, L_x and L_y, for each dimension (x and y).

Given an element in a 3-D array of size $X \times Y \times Z$, its 1-D address $f(l_1, l_2, \ldots, l_6)$, where $0 \leq f(l_1, l_2, \ldots, l_6) < XYZ$, can be mapped to a 3-D address (g_x, g_y, g_z), where $0 \leq g_x < X$, $0 \leq g_y < Y$, and $0 \leq g_z < Z$, as follows.

$$g_x(l_1, l_2, \ldots, l_6) = f(l_1, l_2, \ldots, l_6) \,\%\, X, \tag{2}$$

$$g_y(l_1, l_2, \ldots, l_6) = f(l_1, l_2, \ldots, l_6) \,/\, X \,\%\, Y, \tag{3}$$

$$g_z(l_1, l_2, \ldots, l_6) = f(l_1, l_2, \ldots, l_6) \,/\, X \,/\, Y, \tag{4}$$

where g_x, g_y, and g_z represent the address in the x-axis, that in the y-axis, and that in the z-axis, respectively, and $\%$ is a modulo operator.

Although this mapping approach specifies the memory region with a 3-D address, the modulo operations in Eqs. (2) and (3) do not satisfy the constraint required for Theorem 1. That is, neither g_x nor g_y is an affine function of

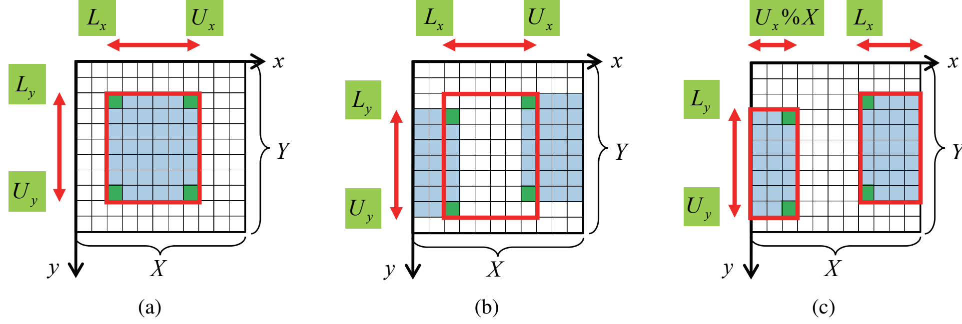

Figure 5. Estimating the memory access region for multi-dimensional data. (a) The estimated region is specified with four corners, corresponding to the least upper bounds, U_x and U_y, and the greatest lower bounds, L_x and L_y, for each dimension (x and y). (b) A naive extension results in wrong estimation, whereas (c) the proposed extension produces correct estimation.

$l_1, l_2, \ldots, l_6$. Owing to this failure, the mapping approach results in an incorrect decomposition, as depicted in Fig. 5(b); in this example, the accessed region consists of two pieces, one existing in the left-most columns and the other existing in the right-most columns, but the estimated region excludes them. Thus, another approach is necessary to obtain a correct decomposition for multi-dimensional data.

4. Proposed Data Decomposition Method

Our data decomposition method takes three inputs to produce data segments for multi-GPU execution: (1) a single-GPU code, (2) the size $X \times Y \times Z$ of the array to be decomposed, and (3) the number of decompositions (i.e., segments). Similar to the previous method [7], we assume that the input code satisfies the constraint mentioned in Section 3.

4.1. Estimating Memory Access Region for Multi-Dimensional Data

The key idea for correct estimation is to regard sizes X and Y in Eqs. (2) and (3) as constant values in the GPU code. In general, these sizes are coded as variables that can be changed at runtime, but they are usually fixed when invoking the GPU code from the CPU code. This interpretation eliminates the modulo operations by precomputation, which can be conducted before invoking the GPU code. Consequently, Eqs. (2)–(4) can be regarded as affine functions as follows.

$$g_x(l_1, l_2, \ldots, l_6) = \sum_{i=0}^{6} ((a_i \,\%\, X) \times l_i)\,, \tag{5}$$

$$g_y(l_1, l_2, \ldots, l_6) = \sum_{i=0}^{6} ((a_i \,/\, X \,\%\, Y) \times l_i)\,, \tag{6}$$

$$g_z(l_1, l_2, \ldots, l_6) = \sum_{i=0}^{6} ((a_i \,/\, X \,/\, Y) \times l_i)\,. \tag{7}$$

For example, our precomputation replaces $a_i \,\%\, X$ in Eq. (5) with a constant value, namely a coefficient of an affine

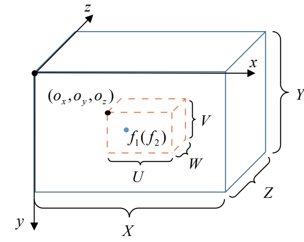

Figure 6. The coordinate systems for a 3-D array before and after decomposition. An element at f_1 in the xyz-coordinate system corresponds to that at f_2 in the uvw-coordinate system.

function. Notice that Eqs. (5) and (6) are not equivalent to Eqs. (2) and (3), respectively; for example, Eq. (2) implies $0 \le g_x < X$, which is not always true in Eq. (5). When $g_x \le X$, our method detects multiple pieces in a segment, which we discuss below.

For an example of Fig. 5(c), where the memory access region is separated into two pieces, the proposed method detects multiple pieces in a segment. This detection can be done as follows. Let U_x and L_x be the least upper bound and the greatest lower bound along the x-axis, respectively. The accessed region then consists of a single piece if $U_x < X$; elements in range $[L_x, U_x]$ are regarded as a segment. Otherwise, two pieces existing in ranges $[0, U_x \% X]$ and $[L_x, X-1]$ can be merged into a segment. Thus, the proposed method successfully estimates the correct region for arbitrary cases that satisfy the constraint.

4.2. Index Transformation

In general, data decomposition reduces the memory usage but it changes the indexing scheme after decomposition. Let $\mathcal{C}_1$ be the original indexing scheme defined over the entire data. Let $\mathcal{C}_2$ also be the indexing scheme defined for a separated segment. The GPU code then must be correctly translated with mapping $\mathcal{C}_1$ to $\mathcal{C}_2$.

Figure 6 shows a geometric relation between the original 3-D array and its separated segment. A rectangular enclosed with lines represents the original array of size $X \times Y \times Z$,

TABLE 1. EXPERIMENTAL ENVIRONMENT.

Item	Specification
CPU	Intel Xeon E5-2680v2
CPU memory capacity	512 GB
GPU	Two NVIDIA Tesla K40
GPU memory capacity	12 GB per GPU
OS	Ubuntu 14.04
Compiler	gcc 4.8.4
CUDA	6.5

whereas that enclosed with dashed lines represents a segmented array of size $U \times V \times W$. A 1-D memory address f_1 in the original scheme $\mathcal{C}_1$, where $0 \leq f_1 < XYZ$, then can be transformed to a 1-D memory address f_2 in the decomposed scheme $\mathcal{C}_2$, where $0 \leq f_2 < UVW$, as follows.

$$\begin{aligned} f_2 &= (f_1 \,\%\, X - o_x) \\ &\quad + (f_1 \,/\, X \,\%\, Y - o_y) \times U \\ &\quad + (f_1 \,/\, X \,/\, Y - o_z) \times U \times V, \end{aligned} \tag{8}$$

where (o_x, o_y, o_z) $(0 \leq o_x < X, 0 \leq o_y < Y, 0 \leq o_z < Z)$ is a 3-D coordinate of the original array that corresponds to the origin of the segment (i.e., decomposed array).

5. Experimental Results

We compared the proposed method with the previous method [7] in terms of the GPU memory usage and the effective performance. To do this, we applied both methods to matrix multiplication code of the CUDA software development kit (SDK) and measured their performance on a dual-GPU system (see Table 1). The comparable codes are summarized as follows.

1) The original single-GPU code. This code computes matrix multiplication $C = A \times B$ for matrix size $N \times N$.
2) The previous multi-GPU code. This code was generated by the previous method, which deployed a 1-D addressing scheme. As depicted in Fig. 7(a), matrices A and C were segmented with a 1-D block scheme. Note that matrix B cannot be segmented because columns in B are necessary to compute a row block in C.
3) The proposed multi-GPU code. This code was generated by the proposed method. All matrices A, B, and C were segmented with a 2-D block scheme (Fig. 7(b)).

Note that the original code can multiply matrices as long as the matrices can be stored entirely on GPU memory. By contrast, this limitation is relaxed by data decomposition, which allows matrices to be larger than the capacity of GPU memory. However, both the proposed and previous methods assume that the entire matrix is stored in CPU memory. In the following discussion, let M be the number of data decompositions (i.e., segments). We assume that M is a square of an arbitrary integer.

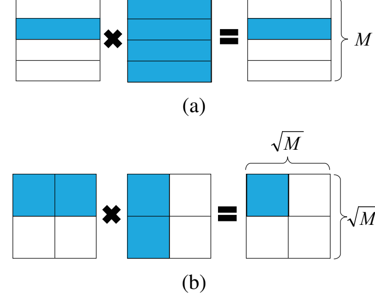

Figure 7. Data decomposition for matrix multiplication by the previous method and our method. (a) 1-D block scheme and (b) 2-D block scheme. M represents the number of decompositions.

The proposed and previous codes were accelerated with a software pipeline that overlapped CPU-GPU data transfer with GPU code execution. To realize this, both codes were implemented with an asynchronous CUDA APIs and double buffers; the buffer size was set to 6 GB per GPU, namely half of the capacity of GPU memory. Generated tasks were assigned to devices in a cyclic manner to achieve load balancing.

5.1. GPU Memory Usage

We analytically evaluated the proposed method with respect to the GPU memory usage estimated from the segment size. Recall here that the previous method fully included matrix B into segments. For a matrix of size $N \times N$, the segment size of the previous method is given by

$$4 \times N^2 \times \left(\frac{1}{M} + 1 + \frac{1}{M} \right) = 4N^2 \frac{M+2}{M}, \tag{9}$$

while that of the proposed method is given by

$$4 \times N^2 \times \left(\frac{\sqrt{M}}{M} + \frac{\sqrt{M}}{M} + \frac{1}{M} \right) = 4N^2 \frac{2\sqrt{M}+1}{M}. \tag{10}$$

Note that matrix elements were stored in `float` arrays and the segment sizes were given in bytes.

According to Eqs. (9) and (10), we investigated how the number M of decompositions affected the segment size. Figure 8 shows the relation between M and the segment size for a fixed matrix size of $40,000 \times 40,000$. Without data decomposition, the matrices required 19.2 GB of memory space, so that the original CUDA SDK code failed to run due to the exhaustion of GPU memory.

By contrast, the proposed method successfully reduced the segment size from 19.2 GB to 0.8 GB by increasing M with a 2-D block scheme. Consequently, our method allows large matrices to be multiplied correctly as long as they are small enough for CPU memory. For example, for large matrices of size $40,000 \times 40,000$, $M = 9$ segments are sufficient to reduce the segment size to approximately 5 GB, which can be stored in a double buffer.

Similarly, the previous method reduced the segment size, but it consumed 6.4 GB of memory space. As we mentioned

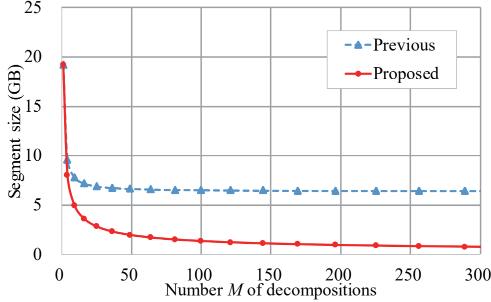

Figure 8. Segment size for different numbers of decompositions. The matrix size was set to $N \times N = 40,000 \times 40,000$.

TABLE 2. MATRIX SIZE $N \times N$ AND ITS MEMORY REQUIREMENT WITHOUT DATA DECOMPOSITION.

Code	Maximum matrix size N	Memory space (GB)
Original	31,584	11
Previous [7]	38,496	17
Proposed	207,680	482

above, the 1-D block scheme requires the entire matrix B, and thereby the segment size is at least $4N^2$ in bytes. Therefore, for large matrices of size $40,000 \times 40,000$, the segment size was larger than the buffer size of 6 GB. In this case, multi-GPU execution failed for arbitrary M.

5.2. Maximum Matrix Size

We next investigated the maximum matrix size that could be successfully processed on the experimental machine (Table 2). Table 2 clearly shows that the reduced segment size increased the maximum matrix size. The original single-GPU code resulted in an execution failure if the matrices A, B, and C consumed approximately 11 GB of memory space, which was close to the capacity of GPU memory; the maximum matrix size reached $31,584 \times 31,584$. The proposed method relaxed this limitation to the capacity of CPU memory (instead of that of GPU memory), and thereby the maximum matrix size reached $207,680 \times 207,680$, which was 44 times larger than the original size; a memory space of 482 GB, which was close to the capacity of CPU memory, was consumed for this problem size.

By contrast, the previous method increased the maximum matrix size to $38,496 \times 38,496$, which was 49% larger than the original size but was 1/29 of that achieved by the proposed method. According to Eq. (9), the segment size was approximately 6 GB for this size, which was equivalent to the buffer size. Thus, the 2-D block scheme rather than the 1-D block scheme is necessary to take advantage of data decomposed matrix multiplication.

Recall that compiler-based methods [15], [16] assume that the entire data can fit into the total capacity of GPU memories. For example, AMGE [15] running on four Tesla K40 GPUs processed the maximum matrix size of $48,900 \times 48,900$, which was 1/18 of that processed with our method.

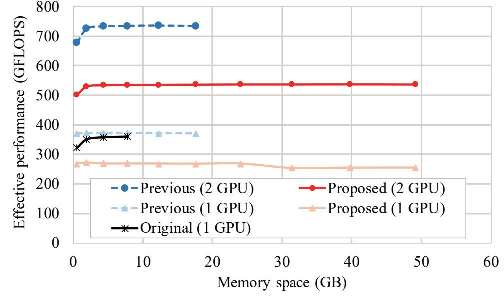

Figure 9. Effective performance of matrix multiplication.

5.3. Effective Performance

We measured the effective performance of each code with varying the matrix size $N \times N$ from 6400×6400 to $64,000 \times 64,000$. Figure 9 shows the effective performance in giga floating point number operations per second (GFLOPS); we varied the number M of decompositions from 1^2 to 16^2 to find the highest performance given by $2N^3/T$, where T is the best execution time, including CPU-GPU data transfer time.

By comparing single-GPU performance, the previous method was slightly faster than the original method. By contrast, the proposed method was 28% slower than the previous method for relatively small matrices. A similar performance behavior was observed for dual-GPU performance (27% slowdown). This performance degradation was due to the overhead needed for index transformation. As we mentioned in Section 4.2, the proposed method transforms indexes at every memory access in the GPU code. A similar transformation is needed for the previous method, but only one subtraction operation is sufficient for the 1-D block scheme. By contrast, the 2-D block scheme requires 13 operations, containing modulo, division, multiplication, and subtraction operations, as presented in Eq. (8). These operations increased the kernel execution time as compared with that of the previous method.

Figure 9 also shows that the effective performance was stable for different matrix sizes even though the required memory space exceeded the capacity of GPU memory. As compared with the proposed method, the previous method consumed more GPU memory and spent longer time to transfer segments between the CPU and GPU, but this drawback was not clearly observed; the software pipeline successfully overlapped CPU-GPU data transfer with kernel execution, and moreover, the kernel execution rather than CPU-GPU data transfer was the performance bottleneck, which determined the entire performance. Therefore, the drawback of the previous method was not a critical issue for matrix multiplication.

Note that the effective performance slightly dropped for small matrices of size 6400×6400; a load imbalance issue occurred for such small matrices, which limit the number M of segments. For example, both the proposed and previous methods dropped the performance by 10% when $M = 3$ on

dual-GPU execution. Thus, finding the appropriate number M of decompositions is the key to maximize the effective performance.

The speedup of dual-GPU execution over single-GPU execution was approximately a factor of two for both the proposed and previous methods. In general, the overhead of data decomposition increases with the number M of decompositions, but such performance degradation could not be observed in Fig. 9. For example, for matrices of size $32,000 \times 32,000$, the proposed method achieved the maximum performance of 535 GFLOPS on two GPUs when $M = 6$, which was close to 531 GFLOPS obtained when $M = 16$. Consequently, we think that matrix multiplication can scale its performance with the number of GPUs as long as the effective bandwidth of the PCIe bus increases with the number of GPUs.

6. Conclusion

In this paper, we presented a data decomposition method for multi-dimensional data, aiming at accelerating a single-GPU code on a multi-GPU system. Our method extends a previous method [7], namely a data decomposition method for 1-D data, such that multi-dimensional data can be decomposed into small segments. Similar to the previous method, our method produces data segments by performing a sample run of selected GPU threads, assuming that the indexes of accessed arrays are given by affine functions of thread and thread block indexes. To realize this, our method precomputes and eliminates modulo operations, which avoid array indexes from being interpreted as affine functions. This precomputation correctly produces smaller segments, which not only save the GPU memory usage but also reduce CPU-GPU data transfer time.

In experiments, we evaluated the presented and previous methods in terms of the memory usage and the effective performance on a dual-GPU system. To do so, we manually applied our method to matrix multiplication code of the CUDA SDK. As compared with the previous method, our method successfully processed 29 times larger matrices, which can be entirely stored in CPU memory but cannot fit into the GPU memory. However, our multi-dimensional method made index transformation more complicated than the previous 1-D method, and thus, the transformation overhead dropped the entire performance by 28%. In summary, our method is useful for dealing with large data that cannot be naively processed due to the exhaustion of GPU memory.

Future work includes automated code generation.

Acknowledgments

This study was supported in part by the Japan Society for the Promotion of Science KAKENHI Grant Numbers 15K12008, 15H01687, 16H02801, and the Japan Science and Technology Agency CREST program, “An Evolutionary Approach to Construction of a Software Development Environment for Massively-Parallel Computing Systems.” We are also grateful to the anonymous reviewers for their valuable comments.

References

[1] NVIDIA Corporation, “NVIDIA GeForce GTX 980,” Nov. 2014. [Online]. Available: http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF

[2] Y. Sugimoto, F. Ino, and K. Hagihara, “Improving cache locality for GPU-based volume rendering,” *Parallel Computing*, vol. 40, no. 5/6, pp. 59–69, May 2014.

[3] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, “GPU computing,” *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[4] D. Okada, F. Ino, and K. Hagihara, “Accelerating the Smith-Waterman algorithm with an interpair pruning method for all-pairs comparison of base sequences,” *BMC Bioinformatics*, vol. 16, no. 321, Oct. 2015, 15 pages.

[5] NVIDIA Corporation, “CUDA C Programming Guide Version 6.5,” Aug. 2014. [Online]. Available: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[6] F. Ino, S. Nakagawa, and K. Hagihara, “GPU-Chariot: A programming framework for stream applications running on multi-GPU systems,” *IEICE Trans. Information and Systems*, vol. E96-D, no. 12, pp. 2604–2616, Dec. 2013.

[7] J. Kim, H. Kim, J. H. Lee, and J. Lee, “Achieving a single compute device image in OpenCL for multiple GPUs,” in *Proc. PPoPP’11*, Feb. 2011, pp. 277–288.

[8] Khronos OpenCL Working Group, “The OpenCL specification version 2.2,” Mar. 2016, http://www.khronos.org/registry/cl/.

[9] C. Müller, S. Frey, M. Strengert, C. Dachsbacher, and T. Ertl, “A compute unified system architecture for graphics clusters incorporating data locality,” *IEEE Trans. Visualization and Computer Graphics*, vol. 15, no. 4, pp. 605–617, Jul. 2009.

[10] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, “SnuCL: An OpenCL framework for heterogeneous CPU/GPU clusters,” in *Proc. ICS’12*, Jun. 2012, pp. 341–352.

[11] C.-K. Luk, S. Hong, and H. Kim, “Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping,” in *Proc. MICRO’09*, Dec. 2009, pp. 45–55.

[12] F. Ji, H. Lin, and X. Ma, “RSVM: a region-based software virtual memory for GPU,” in *Proc. PACT’13*, Sep. 2013, pp. 269–278.

[13] J. Lee, M. Samadi, Y. Park, and S. Mahlke, “Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems,” in *Proc. PACT’13*, Sep. 2013, pp. 245–255.

[14] J. Lee, M. Samadi, and S. Mahlke, “VAST: The illusion of a large memory space for GPUs,” in *Proc. PACT’14*, Aug. 2014, pp. 443–454.

[15] J. Cabezas, L. Vilanova, I. Gelado, T. B. Jablin, N. Navarro, and W. mei W. Hwu, “Automatic parallelization of kernels in shared-memory multi-GPU nodes,” in *Proc. ICS’15*, Jun. 2015, pp. 3–13.

[16] L. D. Solano-Quinde, B. M. Bode, and A. K. Somani, “Automatic parallelization of GPU applications using OpenCL,” in *Proc. APCASE’15*, Jul. 2015, pp. 276–283.

[17] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel Programming in OpenMP*. San Mateo, CA: Morgan Kaufmann, Oct. 2000.

[18] OpenACC-Standard.org, “The OpenACC application programming interface, version 2.5,” Oct. 2015.

[19] T. Kato, F. Ino, and K. Hagihara, “PACC: An extension of OpenACC for pipelined processing of large data on a GPU,” in *Poster of SC’14*, Nov. 2014.

[20] S. Lee and R. Eigenmann, “OpenMPC: extended OpenMP for efficient programming and tuning on GPUs,” *Int’l J. Computational Science and Engineering*, vol. 8, no. 1, pp. 4–20, 2013.