2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing

Accelerating Parameter Sweep Applications Using CUDA

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

***Abstract*—This paper proposes a parallelization scheme for parameter sweep (PS) applications using the compute unified device architecture (CUDA). Our scheme focuses on PS applications with irregular access patterns, which usually result in lower performance on the GPU. The key idea to resolve this irregularity is to exploit the similarity of data accesses between different parameters. That is, the scheme simultaneously processes multiple parameters instead of a single parameter. This simultaneous sweep allows data accesses to be coalesced into a single access if the irregularity appears similarly at every parameter. It also reduces the amount of off-chip memory access by using fast on-chip memory for the data commonly accessed for multiple parameters. As a result, the scheme achieves up to 4.5 times higher performance than a naive scheme that processes a single parameter by a kernel invocation.**

***Keywords*-parameter sweep; acceleration; GPU; CUDA;**

I. Introduction

The parameter sweep (PS) is a well-known strategy for solving combinational optimization problems in parallel. For example, PS applications typically apply the same operations to different parameters in order to find the best parameter from the parametric space. Since the number of combinations is usually enormous, many researchers are trying to accelerate PS applications using high-performance computing systems. For example, grid systems provide us a powerful solution with a distributed computing environment [1]. Such systems typically exploit the coarse-grained parallelism inherent in the PS computation because different parameters usually do not have data dependence between their computations. Therefore, parallelization can be efficiently achieved by a master-worker scheme, which simply assigns a set of independent parameters to computing nodes in a round-robin fashion. Thus, the PS application is one of the killer applications for grid systems.

One remarkable feature of recent grid systems is that the graphics processing unit (GPU) [2], [3] is emerging as a powerful accelerator not only for graphics applications but also for general, non-graphics applications. For example, the Folding@home project [4] accelerates simulations of protein folding and other molecular dynamics using more than 15,000 GPUs. Their simulator is implemented using the computer unified device architecture (CUDA) [5], which is a development framework for the NVIDIA GPU [2]. In their system, the GPU provides 60% of the entire throughput though it accounts only for 5% of all available resources, including the CPU and the Cell Broadband Engine. Thus, the GPU is increasing their contribution to the grid system performance. Therefore, we need a general parallelization scheme to efficiently run PS applications on the GPU.

In this paper, we propose a parallelization scheme for accelerating PS applications on the CUDA-compatible GPU, aiming at enhancing grid systems with GPU-based acceleration. Similar to the master-worker scheme, we focus on the data parallelism in the PS computation. The main difference to this previous scheme is that the proposed scheme simultaneously processes multiple parameters as a fine-grained task on the GPU. Such a multiple sweep allows the GPU to access data in a coalesced manner [5], which is useful to maximize the effective bandwidth of off-chip video memory. In particular, our scheme will efficiently run if each parameter has irregular access patterns but with a similar behavior between different parameters. Furthermore, we can save the bandwidth of video memory by using on-chip memory for the common data that can be accessed for multiple parameters.

The remainder of the paper is structured as follows. Section II introduces related work. Section III presents preliminaries including an overview of CUDA and a computational model of PS applications. Section IV describes our scheme and then Section V shows some experimental results. Finally, Section VI concludes the paper.

II. Related Work

To the best of our knowledge, there is no work that aims at providing a general framework for accelerating PS applications on the GPU. However, some specific applications are successfully accelerated on the GPU. For example, Liu *e*t al. [6] present a GPU-based method that implements a biological alignment algorithm. Their method parallelizes the algorithm by processing multiple pairs of data at a time, which is similar to our parallelization scheme. Since their work is done using the OpenGL graphics library, which differs from the CUDA in terms of programming model and architecture, it is not clear whether the scheme is useful for CUDA-compatible GPUs and for general applications.

1066-6192/11 $26.00 © 2011 IEEE
DOI 10.1109/PDP.2011.19

In contrast, there are many projects that support parameter study on CPU-based grid systems. For example, Condor [7] is one of the first grid middleware that focuses on idle machines for acceleration of scientific applications. It provides a software framework [8] that allows users to easily parallelize applications using the master-worker paradigm on the grid. The AppLeS parameter sweep template (APST) [9] is also a grid middleware designed to efficiently and adaptively use resources managed by multiple grid environments. Our scheme can be integrated into these previous systems because there is no overlap between their CPU-related achievements and our GPU-related contribution.

With respect to optimization strategies for the GPU, Ryoo *e*t al. [10] propose an optimization procedure called program optimization carving. Their procedure is designed to reduce the optimization search space. According to the procedure, we can easily find the best configuration parameters such as the thread block size and the loop unroll factor. In contrast, our scheme tackles the problem of memory coalescing. We demonstrate that memory coalescing can be achieved more easily if there are many parameters that can be processed at the same time.

III. Preliminaries

This section presents preliminaries needed to understand our parallelization scheme.

A. Compute Unified Device Architecture (CUDA)

The CUDA [5] is a development framework for writing and running parallel applications on the NVIDIA GPU. Using this framework, parallel applications can be written in the C-like language. As shown in Fig. 1, we can regard the GPU as hierarchical multiprocessors (MPs), where each MP has streaming processors (SPs) capable of processing single instruction, multiple data (SIMD) computation with millions of threads.

In the CUDA framework, the GPU program is implemented as a kernel function, which will be called from the CPU program. Since the GPU is based on the SIMD architecture, every thread processes an instance of the same kernel. Similar to the hardware architecture, threads have a hierarchical organization, where they are grouped into thread blocks (TBs) such that different TBs do not have data dependence between themselves. TBs are then assigned to one of MPs for SIMD execution on SPs. Note here that it is important to reduce per-thread resource consumption. This reduction will lead to have more active TBs on MPs because TBs are assigned to MPs as far as there are resources. Such multiple assignments allow MPs to switch active TBs to overlap the memory access latency with data-independent computation for another TB.

As shown in Fig. 1, the CUDA-compatible GPU has two different physical memories: on-chip memory and off-chip memory. The former contains shared memory and register files while the latter includes global memory, texture memory and constant memory. Shared memory has the same latency (4 clocks) as registers if bank conflicts [5] are not occurred during memory access. Since each MP has its own on-chip memory, shared memory can be used to exchange data between threads in the same TB. In other words, such data exchanges are not available for threads in different TBs because they can be assigned to different MPs.

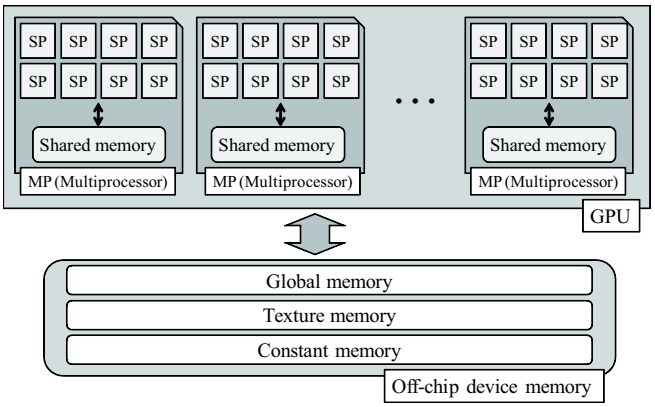

Figure 1. Structure of the CUDA-compatible GPU. SP denotes streaming processor.

On the other hand, off-chip memory has 400 clocks of latency. To deal with this high latency issue, memory transactions to global memory can be coalesced into a single transaction if a series of 16 threads, called a half warp, accesses the same segment on global memory [5]. Such coalesced accesses will increase the effective bandwidth of off-chip memory because the memory latency is associated with a memory transaction. Constant and texture memory resolve the latency problem by a cache but they are not writable from the GPU.

Summarizing the above discussion, memory-bound applications can be efficiently accelerated according to the following guidelines.

1) Hiding global memory latency by memory access coalescing.
2) Reducing off-chip memory access by on-chip shared memory.
3) Running more threads concurrently on each MP by saving per-thread resource consumption.

B. Parameter Sweep Model

Let n be the number of parameters to be swept. To simplify the description, our model assumes that every parameter P_k requires a single set $\mathcal{I}_k$ of input data to generate a single set $\mathcal{O}_k$ of output data, where $1 \leq k \leq n$. Despite of this assumption, our design scheme can be easily extended to deal with multiple sets. The model also assumes that each set has the same number m of elements: $|\mathcal{I}_k| = |\mathcal{O}_k| = m$. The input data set and the output data set then can be given

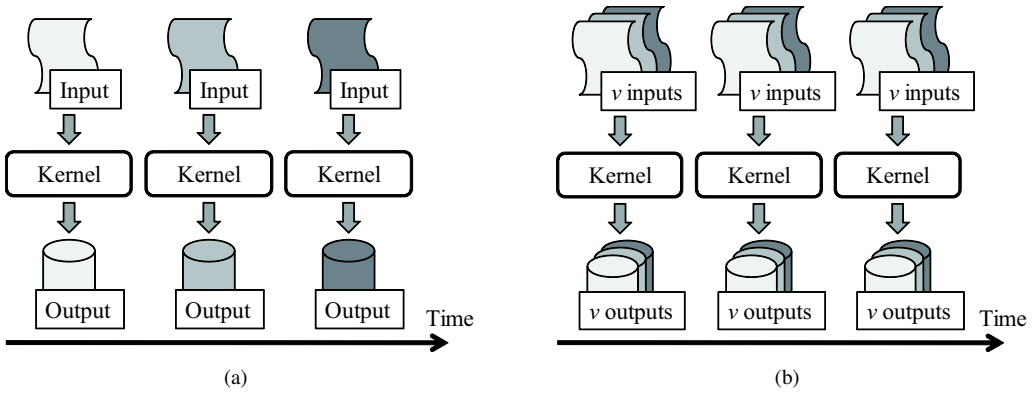

Figure 2. Parallelization schemes for PS applications. (a) A kernel invocation in a naive scheme is responsible for a single parameter while (b) that in the proposed scheme is responsible for multiple parameters. Notation v denotes the number of parameters processed at the same time ($v = 3$, in this example).

by

$$\mathcal{I}_k = \{ e_{k,s} \mid 1 \le s \le m \}, \quad (1)$$

$$\mathcal{O}_k = \{ f_{k,t} \mid 1 \le t \le m \}, \quad (2)$$

where $1 \le k \le n$, and $e_{k,s}$ and $f_{k,t}$ represent an element of the input set and that of the output set, respectively. For example, $\mathcal{I}_k$ and $\mathcal{O}_k$ correspond to the k-th input and output images in image processing applications, respectively, while $e_{k,s}$ and $f_{k,t}$ correspond to pixels in the images. Using the representation mentioned above, the subset $\mathcal{D}$ of input data commonly accessed for all parameters can be written as

$$\mathcal{D} = \bigcap_{1 \le k \le n} \mathcal{I}_k. \quad (3)$$

Let $\mathcal{T}_k$ denote the task associated with parameter P_k, where $1 \le k \le n$. Since there is no data dependence between different tasks, the master-worker scheme organizes a set of arbitrary tasks to exploit the coarse-grained parallelism on distributed systems.

IV. Proposed Scheme

The proposed scheme focuses on the following two characteristics of PS applications.

- The same operations are processed for every parameter P_k, where $1 \le k \le n$.
- Different parameters can commonly access the same subset $\mathcal{D}$ of input data.

According to the characteristics mentioned above, our scheme simultaneously processes multiple tasks $\mathcal{T}_k, \mathcal{T}_{k+1}, \ldots, \mathcal{T}_{k+v-1}$, where v represents the number of parameters processed by a kernel invocation (see Fig. 2(b)). We currently use the size of a half-warp ($v = 16$) for the GT200 architecture in order to coalesce memory transactions. The details will be presented later.

Figure 3 illustrates an overview of our scheme. The scheme requires reorganization of input/output data before and after kernel invocation in order to achieve coalesced access on the GPU. Furthermore, it takes longer time to execute a single kernel because the kernel processes multiple parameters instead of a single parameter. However, this increased time is a trivial problem if there is a large number n of parameters to be swept. Thus, the total number of kernel invocations is decreased by a factor of v in our scheme.

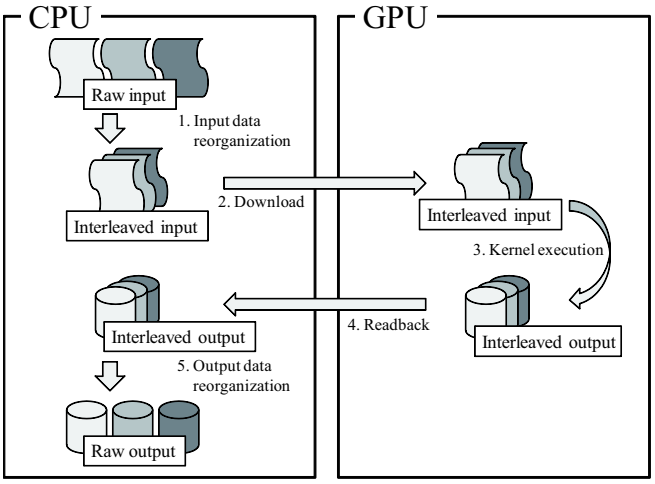

Figure 3. Overview of the proposed scheme. Our scheme involves data reorganization before and after kernel invocation to store data in an interleaved manner.

A. Hiding Global Memory Latency

As shown in Fig. 4, the key idea for acceleration is to reorganize the input/output data $\mathcal{I}_k$ and $\mathcal{O}_k$ such that memory transactions can be coalesced into a single transaction. We think that this reorganization can improve the kernel performance especially if each parameter has irregular accesses but with similar irregularity between different parameters. Such irregular accesses prevent us from coalescing memory transactions, usually resulting in lower performance in the previous scheme. Thus, we think that the SIMD nature of PS applications makes it easier to achieve coalesced access

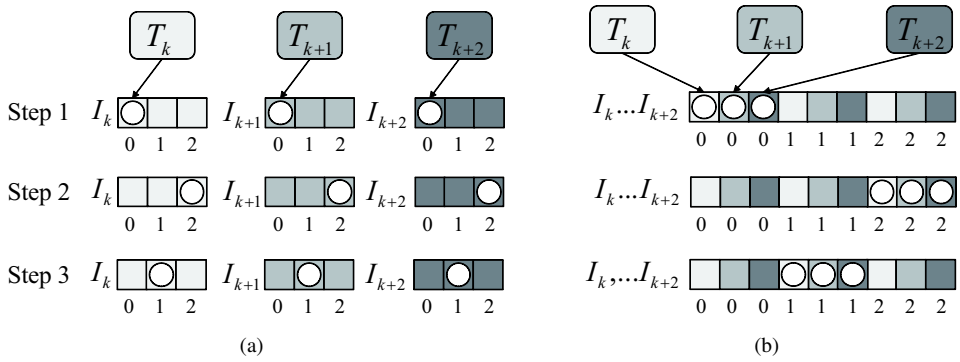

Figure 4. Data reorganization in the proposed scheme. Array elements with circles are accessed at each step. (a) Each parameter has irregular accesses that cannot be coalesced into a single transaction if data is not stored in an interleaved manner. After reorganization, (b) such accesses can be coalesced if the irregularity appears similarly at every parameter.

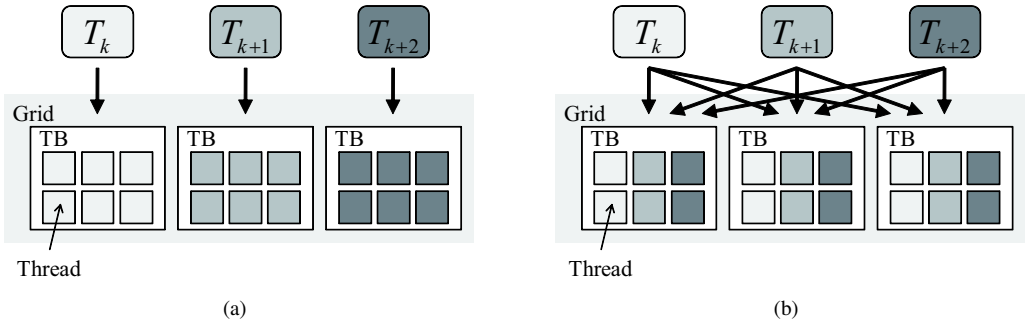

Figure 5. Task assignment. (a) Each TB in a naive scheme is responsible for a single task. In contrast, (b) each TB in our scheme is responsible for multiple tasks.

for multiple parameters rather than for a single parameter.

With respect to the GT200 architecture, memory coalescing can be achieved if the PS application satisfies the following conditions.

1) Threads in a half-warp are responsible for v different tasks $\mathcal{T}_k, \mathcal{T}_{k+1}, \ldots, \mathcal{T}_{k+v-1}$.
2) Input and output data required for tasks $\mathcal{T}_k, \mathcal{T}_{k+1}, \ldots, \mathcal{T}_{k+v-1}$ is stored in an interleaved manner.

In order to satisfy condition 1), every TB in our scheme is responsible for v consecutive tasks $\mathcal{T}_k, \mathcal{T}_{k+1}, \ldots, \mathcal{T}_{k+v-1}$, as shown in Fig. 5(b). Note that this condition cannot be achieved if we use a naive scheme where each TB is responsible for a single task, as shown in Fig. 5(a).

Using the task assignment scheme mentioned above, condition 2) is further needed to achieve memory coalescing. Suppose that task $\mathcal{T}_k$ accesses an input element $e_{k,s}$, where $1 \leq k \leq n$ and $1 \leq s \leq m$. Since the PS application applies the same operations to different parameters, other tasks $\mathcal{T}_{k+1}, \mathcal{T}_{k+2}, \ldots, \mathcal{T}_{k+v-1}$ probably access their input elements $e_{k+1,s}, e_{k+2,s}, \ldots, e_{k+v-1,s}$ located at the same address s, respectively (see Fig. 4(a)). Therefore, as shown in Fig. 4(b), we reorganize the input/output data in an interleaved manner. This reorganization stores data elements such that different tasks can access contiguous address at the same cycle. That is, it stores elements $e_{k,s}, e_{k+1,s}, \ldots, e_{k+v-1,s}$ in a sequence, instead of the original order: $e_{k,1}, e_{k,2}, \ldots, e_{k,m}$.

B. Reducing Off-chip Memory Access

As we mentioned in Section III-A, shared memory is useful to save the bandwidth of off-chip memory. For this purpose, our scheme stores the subset $\mathcal{D}$ of input data in shared memory if possible. To do this, threads in the same TB firstly copy the data from off-chip memory to shared memory. This copy operation must be done in a cooperative manner to prevent redundant memory transactions. Threads are then allowed to perform computation by accessing shared memory instead of off-chip memory. However, they have to copy results to off-chip memory before kernel completion. Since our scheme processes v tasks at the same time, the amount of off-chip memory access can be roughly reduced by a factor of v if we store $\mathcal{D}$ in shared memory.

Note that this strategy is available only if the naive scheme leaves room in shared memory. Otherwise, our scheme cannot store $\mathcal{D}$ in shared memory, so that it has the same memory allocation as the naive scheme. Another drawback of our scheme is that it can consume v times more shared memory due to simultaneous execution of v tasks. Since the capacity of shared memory is limited by 16 KB in the GT200 architecture, this drawback can reduce the number of active

Table I
OVERVIEW OF EXPERIMENTAL APPLICATIONS. DATA SIZE IS PRESENTED AS PER-PARAMETER VALUE. RESOURCE USAGE IS SHOWN AS PER THREAD BLOCK.

Application	Kernel	Resource usage			Input/output data		
		Grid size	TB size	Shared memory per TB (B)	Classification	Location	Size (B)
Optical simulator	MAD	(256,1)	(64,1,1)	56	$\mathcal{I}$: 2-D images	Global	32 K
					$\mathcal{D}$: coefficients	Shared	24 M
					$\mathcal{O}$: 1-D array	Global	12 M
	Reduction	(256,1)	(4,1,1)	56	$\mathcal{I}$: 1-D array	Global	12 M
					$\mathcal{D}$: none	—	—
					$\mathcal{O}$: 2-D images	Global	32 K
Gaussian filter	Row	(16,2048)	(152,1,1)	616	$\mathcal{I}$: 2-D signal	Global	16 M
					$\mathcal{D}$: coefficients	Constant	68
					$\mathcal{O}$: 2-D signal	Global	16 M
	Column	(128,43)	(16,8,1)	4144	$\mathcal{I}$: 2-D signal	Global	16 M
					$\mathcal{D}$: coefficients	Constant	68
					$\mathcal{O}$: 2-D signal	Global	16 M

threads as compared with the naive scheme. Similarly, the lack of registers can cause the same issue. Thus, the scheme has a disadvantage with respect to guideline 3) mentioned in Section III-A. A possible solution to this issue is to decompose the kernel code into smaller pieces such that each of the pieces can run under limited resources.

V. EXPERIMENTAL RESULTS

We now show some experimental results to evaluate the proposed scheme in terms of performance. We applied our scheme to two practical applications: an optical propagation simulator and a Gaussian filter [5]. Table I shows an overview of the applications with their input/output data. The data sizes are presented as per-parameter values, so that they should be multiplied with $v = 16$ for our scheme.

The simulator performs convolution for a large number of 64×64-pixel image pairs. It is used for optimizing circuit patterns and optical conditions to accelerate development of NAND memory. Therefore, there are a large number of patterns that must be investigated by the simulation. The convolution operation is implemented by the multiply-add (MAD) kernel and the reduction kernel. Since the MAD kernel accesses the same weight coefficients for every pair, this constant data can be regarded as the common data $\mathcal{D}$. Thus, the coefficients can be reused between different pairs, so that they are sent to the GPU only once before the first kernel execution. One problem here is that the coefficients take 24 MB of memory space, which exceeds the capacity of shared memory. To deal with this capacity problem, the MAD kernel performs computation in a step-by-step manner using at most 16 KB of coefficients at each step. In contrast, there is no such a common data in the reduction kernel.

The Gaussian filter applies a separable convolution to two-dimensional (2-D) signal with a Gaussian function. The signal here is given by an array of 2048×2048 elements, which is larger than the images processed by the simulator. Therefore, the filtering performance can be limited by the data transfer between the CPU and the GPU rather than the kernel execution. The filter is implemented by two kernels, each responsible for 1-D convolution in the row direction and that in the column direction. Both kernels store Gaussian coefficients in constant memory because they are small enough to store in the memory and every thread iteratively access them. The kernels are also accelerated using a tiling technique [11], which partitions the image domain into multiple tiles. Similar to cache blocking techniques on the CPU, this technique contributes to reduce the amount of computation by performing data reuse within TBs. Thus, the filter uses shared memory for complexity reduction.

As shown in Table I, all kernels except the reduction kernel have a TB size of at least 64 threads for a single parameter. Therefore, the TB size can exceed the maximum available size of 512 threads [5] if 16 parameters are processed at the same time. To avoid this problem, we reduced the TB size of the original code and then have applied our scheme to the modified code. For the MAD, row and column kernels, the TB sizes in our scheme are reduced to 16×32, 16×32 and 16×2 threads (per 16 parameters), respectively. Similarly, we have to use smaller tiles for the column kernel to reduce the usage of shared memory. We reduced the usage to 50% by using tiles with 33% smaller height.

The experiments were performed using a Windows XP machine having a GeForce GTX 285 card with CUDA 2.3 [5] and driver version 195.62. This machine has a Xeon X5160 CPU running at 3 GHz clock speed. A Linux machine was also used for experiments but we show here the results on the Windows machine because there is no significant difference between them.

A. Performance Comparison with Previous Scheme

We first compare the scheme with a naive scheme that simply processes a single parameter by a kernel invocation. Figure 6 shows the breakdown of the execution time for 16 parameters, containing the data reorganization time, the kernel execution time and the data transfer time between the CPU and the GPU. Note that the data transfer time in Fig. 6(a) does not include the time needed for coefficients

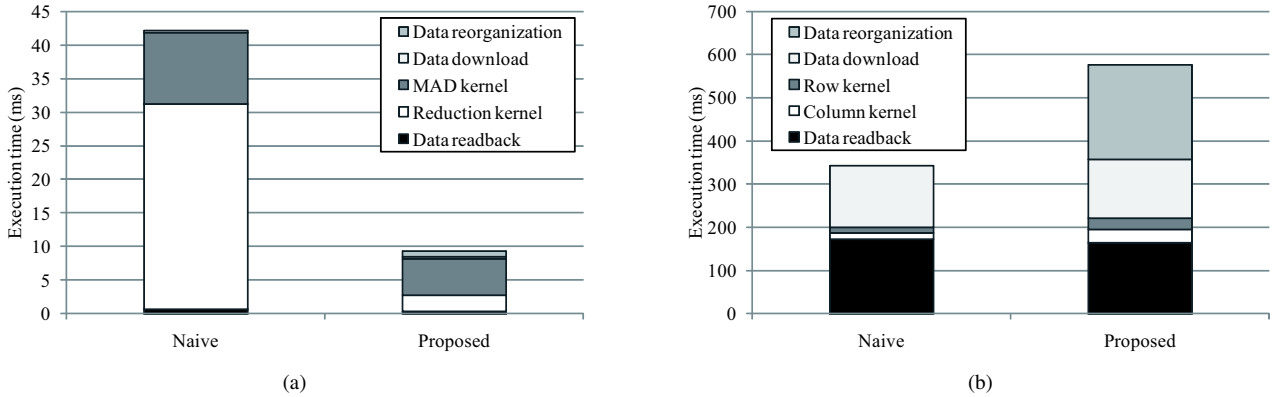

Figure 6. Performance comparison with a naive scheme. Breakdown of execution time (a) for the simulator and (b) for the filter. Data download represents the data transfer from the CPU to the GPU and data readback represents that in the opposite direction.

because they are sent only once before the first kernel invocation, as we mentioned in Section V.

With respect to the simulator, the proposed scheme reduces the execution time from 42.1 ms to 9.3 ms, achieving 4.5 times higher performance than the naive scheme. In particular, we find that the reduction kernel runs 12 times faster than the naive scheme. This is due to the non-coalesced accesses occurred in the naive scheme. In this kernel, threads in a half-warp access different memory segments at every step. Since each parameter has irregular access patterns, it is not easy to eliminate such inefficient accesses in the naive scheme. Therefore, our scheme eliminates them by coalescing accesses between different parameters. Actually, we find that the proposed scheme reduces the number of 32-byte memory load transactions (glb_32b) from 146,525 to 408 according to the CUDA profiler.

We also find that the overhead of data reorganization is small enough in this application. It takes 0.8 ms to reorganize data for every 16 parameters. This low overhead contributes to increase the speedup over a CPU version from a factor of 6 to that of 28. The CPU version here is a single-core implementation but accelerated using streaming SIMD extensions (SSE) instructions [12].

In contrast, our scheme fails to improve the performance of the Gaussian filter, as shown in Fig. 6(b). We observe 68% performance degradation as compared with the naive scheme. This is mainly due to the overhead of data reorganization. Unlike the simulator mentioned before, the filter processes a large array of 2048×2048 elements. Therefore, the input data size reaches 256 MB when $v = 16$. Since this overhead occurs on the CPU, we think that the overhead can be overlapped with kernel execution by using a stream processing technique [5], [13]. In addition to this overhead, both of the kernels reduce the performance to approximately 50%. A detailed analysis will be presented later in Section V-B.

B. Efficiency Analysis

We next investigate the efficiency of our scheme in terms of effective memory bandwidth. We also evaluate the effects of using shared memory. For this purpose, we develop an implementation that uses our scheme but without storing $\mathcal{D}$ in shared memory.

Figure 7 shows the effective bandwidth of each implementation. In the naive scheme, the original MAD kernel processes 80% of memory transactions in a coalesced manner because it has high memory locality. Therefore, the effective memory bandwidth reaches 78 GB/s, which corresponds to approximately 50% of the theoretical memory bandwidth (159 GB/s). On the other hand, the reduction kernel results in 6 GB/s, which is extremely lower than the MAD kernel due to irregular accesses. By applying our scheme to both kernels, the MAD kernel and the reduction kernel increase the effective bandwidth to 120 GB/s and 79 GB/s, respectively. Thus, our scheme significantly increases the reduction kernel performance by coalescing irregular accesses for multiple parameters.

The MAD kernel further increases the effective bandwidth to 138 GB/s by using shared memory for the common data $\mathcal{D}$, namely the coefficients. In this kernel, every thread fetches 12 array elements to obtain their results. Since 4 out of 12 elements are coefficients stored in shared memory, the amount of off-chip memory accesses is reduced by roughly 33% in the shared memory version. Actually, the proposed scheme achieves 37% higher kernel performance by storing $\mathcal{D}$ in shared memory. Note that the reduction kernel has the same bandwidth whether we use shared memory or not, because it lacks data that can be stored in shared memory.

With respect to the Gaussian filter, the naive scheme processes all memory transactions in a coalesced manner. Furthermore, both kernels efficiently use shared memory, achieving the effective bandwidth of more than 500 GB/s. In contrast, the proposed scheme increases the effective bandwidth from 100 GB/s to around 300 GB/s by using

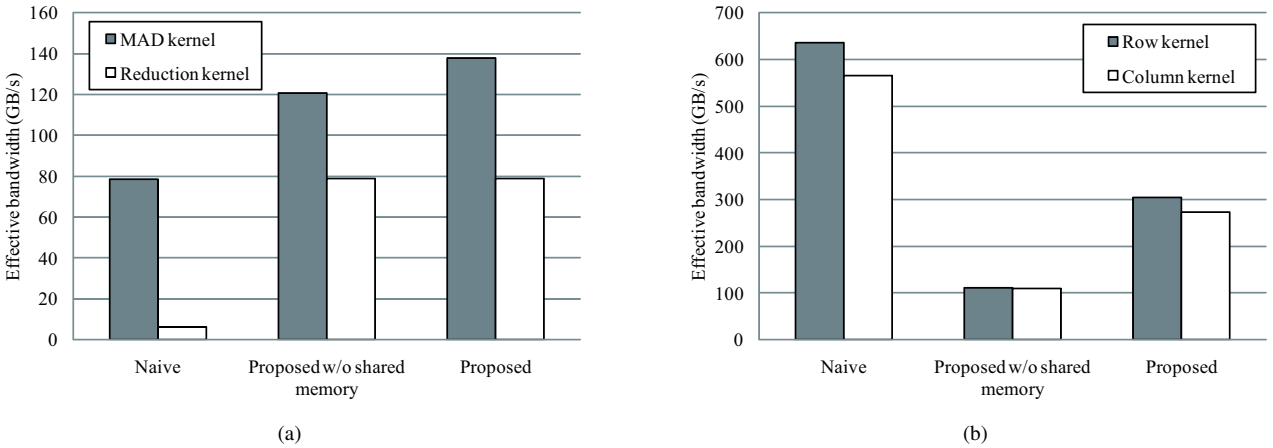

Figure 7. Effective bandwidth of kernels composing (a) the simulator and (b) the filter.

shared memory. Thus, using shared memory for $\mathcal{D}$ is also effective in this application. However, both results are lower than the original bandwidth of 500 GB/s.

In order to investigate why our scheme fails to improve the filtering performance, we develop another naive implementation that uses the same TB size as the proposed scheme. Figure 8 shows the kernel execution time of all implementations. We find that the naive scheme drops the performance of both kernels if it uses the same TB size as the proposed scheme. This performance degradation is due to 33% smaller TBs employed in our scheme. Due to these smaller TBs, the tiling technique reduces the degree of data reuse. Consequently, the row kernel has 1.9 times more instructions and 1.8 times more amount of off-chip memory accesses, taking approximately three times longer execution time as compared with the original version. Thus, the proposed scheme reduces the execution time from 34.7 ms to 27.0 ms but this improvement is achieved without using the optimal TB size.

On the other hand, the column kernel drops the kernel performance due to smaller tiles, which also reduce the efficiency of data reuse. Similar to the row kernel, the column kernel in our scheme has 1.5 times more accesses to off-chip memory though it reduces the usage of shared memory by 50%. In summary, our scheme requires us to modify the kernel code to run with smaller TBs and less amount of shared memory. It can result in low performance if the modified code drops the efficiency such as the degree of data reuse.

Finally, we investigate the effects of constant memory. Using constant memory instead of shared memory, the proposed scheme achieves 16% and 13% higher performance for the row kernel and for the column kernel, respectively (Fig. 7). This improvement is achieved by constant caches, which reduce the amount of off-chip memory accesses. Constant memory differs from shared memory in that different

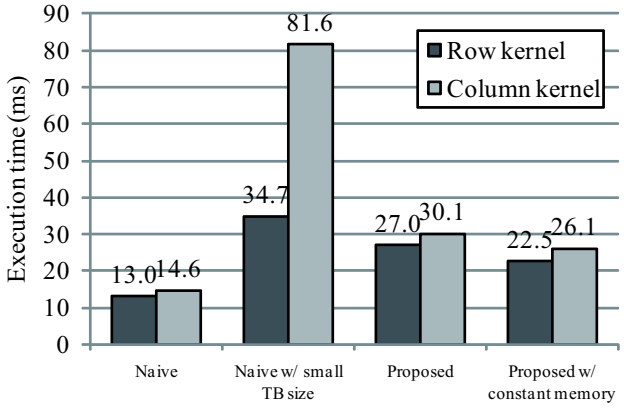

Figure 8. Kernel execution time of Gaussian filter. Naive scheme with small TB size uses the same TB size as the proposed scheme.

TBs can share on-cache data. In the shared memory version, every TB has to copy data from global memory to shared memory because shared memory is a per-TB memory space. In contrast, TBs allocated to the same MP can prevent off-chip accesses after one of them has copied data from global memory. Thus, the number of off-chip memory accesses equals to the number of MPs rather than that of TBs if the constant data is smaller than the cache capacity. Actually, the CUDA profiler shows that the constant memory version allows every TB to have 17 less memory accesses than the shared memory version. Therefore, we think that it is better to store $\mathcal{D}$ in constant memory if it has smaller size than the capacity of constant caches.

VI. Conclusion

We have presented a parallelization scheme for PS applications accelerated using CUDA. Our scheme increases the entire throughput by simultaneously processing multiple parameters instead of a single parameter. It stores the in-

put/output data in an interleaved manner to realize coalesced memory access during kernel execution. Furthermore, the scheme tries to save the bandwidth of off-chip memory by using shared on-chip memory for the common data that can be accessed for multiple parameters.

In experiments, our scheme achieves 4.5 times higher performance for an optical propagation simulator. In particular, the modified reduction kernel runs 11 times faster than the original version because it has irregular access patterns. However, it fails to improve the performance of the Gaussian filter optimized already with memory coalescing. Therefore, we think that the scheme is useful to resolve the issue of irregular accesses if memory coalescing cannot be achieved with a single parameter. We also find a drawback of our scheme. Since the scheme roughly consumes $v = 16$ times more resources, some kernels have to reduce the TB size and the usage of shared memory. Such modifications can result in a lower efficiency in terms of data reuse.

One future work is to develop an automated tool that minimizes the efforts to apply our scheme to the kernel code. We are also planning to evaluate our scheme using the new Fermi architecture [14].

ACKNOWLEDGMENT

This work was partly supported by JSPS Grant-in-Aid for Scientific Research (A)(20240002) and the Global COE Program "in silico medicine" at Osaka University. We are grateful to the anonymous reviewers for their valuable comments.

REFERENCES

[1] S. D. Olabarriaga, A. J. Nederveen, and B. O. Nualláin, "Parameter sweeps for functional MRI research in the "virtual laboratory for e-science" project," in *Proc. 17th IEEE Int'l Symp. Cluster Computing and the Grid (CCGrid'07)*, May 2007, pp. 685–690.

[2] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, Mar. 2008.

[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] The Folding@Home Project, "Folding@home distributed computing," 2010, http://folding.stanford.edu/.

[5] NVIDIA Corporation, "CUDA Programming Guide Version 2.3," Jul. 2009, http://developer.nvidia.com/cuda/.

[6] W. Liu, B. Schmidt, G. Voss, and W. Müller-Wittig, "Streaming algorithms for biological sequence alignment on GPUs," *IEEE Trans. Parallel and Distributed Systems*, vol. 18, no. 9, pp. 1270–1281, Sep. 2007.

[7] M. J. Litzkow, M. Livny, and M. W. Mutka, "Condor - a hunter of idle workstations," in *Proc. 8th Int'l Conf. Distributed Computing Systems (ICDCS'88)*, Jun. 1988, pp. 104–111.

[8] J.-P. Goux, S. Kulkarni, M. Yoder, and J. Linderoth, "Master-worker: An enabling framework for applications on the computational grid," *Cluster Computing*, vol. 4, no. 1, pp. 63–70, Mar. 2001.

[9] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, and D. Zagorodnov, "Adaptive computing on the grid using AppLeS," *IEEE Trans. Parallel and Distributed Systems*, vol. 14, no. 4, pp. 369–382, Apr. 2003.

[10] S. Ryoo, C. I. Rodrigues, S. S. Stone, J. A. Stratton, S.-Z. Ueng, S. S. Baghsorkhi, and W. mei W. Hwu, "Program optimization carving for GPU computing," *J. Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1389–1401, Oct. 2008.

[11] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *Proc. Int'l Conf. High Performance Computing, Networking, Storage and Analysis (SC'08)*, Nov. 2008, 11 pages (CD-ROM).

[12] A. Klimovitski, "Using SSE and SSE2: Misconceptions and reality," in *Intel Developer Update Magazine*, Mar. 2001.

[13] S. Nakagawa, F. Ino, and K. Hagihara, "A middleware for efficient stream processing in CUDA," *Computer Science - Research and Development*, vol. 25, no. 1/2, pp. 41–49, May 2010.

[14] NVIDIA Corporation, "NVIDIA's Next Generation CUDA Compute Architecture: Fermi," Nov. 2009, http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.