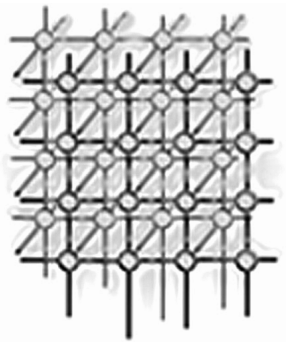
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2013; **24**:1–15 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*



A Parallel Scheme for Accelerating Parameter Sweep Applications on a GPU

Fumihiko Ino[1,*,†], Kentaro Shigeoka[1], Tomohiro Okuyama[1], Masaya Motokubota[2] and Kenichi Hagihara[1]

[1] *Graduate School of Information Science and Technology, Osaka University, 1-5 Yamadaoka, Suita, Osaka 565-0871, Japan*
[2] *Corporate Manufacturing Engineering Center, Toshiba Corporation, 33 Shin-Isogo-Cho, Isogo-ku, Yokohama, Kanagawa 235-0017, Japan*

SUMMARY

This paper proposes a parallel scheme for accelerating parameter sweep applications on a graphics processing unit (GPU). Using hundreds of cores on the GPU, our scheme simultaneously processes multiple parameters rather than a single parameter. The simultaneous sweeps exploit the similarity of computing behaviors shared by different parameters, thus allowing memory accesses to be coalesced into a single access if similar irregularities appear among the parameters' computational tasks. In addition, our scheme reduces the amount of off-chip memory access by unifying the data that are commonly referenced by multiple parameters and by placing the unified data in the fast on-chip memory. In several experiments, we applied our scheme to practical applications, and found that our scheme can perform up to 8.5 times faster than a naive scheme that processes a single parameter at a time. We also include a discussion on application characteristics that are required for our scheme to outperform the naive scheme. Copyright © 2013 John Wiley & Sons, Ltd.

KEY WORDS: parameter sweep; acceleration; GPU; CUDA

*Correspondence to: Fumihiko Ino, Graduate School of Information Science and Technology, Osaka University, 1-5 Yamadaoka, Suita, Osaka 565-0871, Japan
†E-mail: ino@ist.osaka-u.ac.jp

Received 14 September 2012
Revised 12 February 2013

1. INTRODUCTION

Parameter sweep (PS) is a well-known strategy for solving combinatorial optimization problems. PS applications typically process the same sequence of operations with different parameters in order to find the best solution in parametric space. Because the number of combinations is usually large in practical situations, PS applications have been accelerated using high-performance computing (HPC) systems. In such systems, computational grids typically exploit the coarse-grained parallelism inherent in PS computation [1, 2]. As different parameters do not have data dependencies between their operations, PS applications can be efficiently parallelized by a master-worker paradigm where the master node assigns a set of parameters to idle worker nodes in a round-robin fashion.

Another key HPC platform is the GPU [3], for which NVIDIA designed a programming framework called compute unified device architecture (CUDA) [4]. Using this flexible framework, the GPU can serve as a powerful accelerator for not only graphics applications but also general purpose applications that require a significant memory bandwidth to perform arithmetic intensive operations. The GPU accelerator exploits fine-grained parallelism with millions of threads, achieving a typical speedup that is ten times faster than CPU-based implementations [5].

Consequently, many grid computing systems [6, 7] utilize the GPU as a computational engine to attain greater acceleration. For example, the Folding@home project [6] accelerates simulations of protein folding and other molecular dynamics using more than 20,000 GPUs. On each GPU, millions of threads exploit fine-grained data parallelism in single-parameter computations. Owing to this highly parallel computation, GPU-equipped nodes provide 70% of the system throughput although they account for only 10% of all available nodes in the system. Thus, the GPU increases its contribution to the grid system performance. This trend inspired us to develop an efficient parallel scheme for PS applications running on a GPU.

In this paper, we present a parallel scheme for accelerating PS applications on a CUDA-compatible GPU. Similar to the previously mentioned fine-grained scheme, which we will call the *naive scheme*, our scheme focuses on the data parallelism in PS computations. The key difference from the naive scheme is that our scheme simultaneously processes multiple parameters rather than a single parameter. This task organization strategy allows the GPU to exploit similar computing behaviors among the parameters. For instance, irregular data accesses for a single parameter can be efficiently mapped onto the GPU if they are transformed into regular data accesses for multiple parameters. In addition, our scheme saves the off-chip memory bandwidth by unifying the data commonly referenced by the multiple parameters and by using on-chip memory for the unified data. We extend our preliminary results [8] with a detailed case study and guidelines that are useful for selecting the appropriate scheme for a target PS application.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 presents preliminaries, including the naive scheme that maps PS applications onto the CUDA-compatible GPU. Section 4 describes our parallel scheme along with guidelines for selecting a scheme, and Section 5 presents our experimental results. Finally, Section 6 presents our conclusions.

2. RELATED WORK

The GPU has been used to accelerate PS applications in various fields including chemistry [6], physics [9], and bioinformatics [10]. Although these studies have achieved significant acceleration compared with CPU-based implementations, their parallel schemes are specific to the target problems they tackle. On the other hand, our study focuses on developing a general scheme for GPU-accelerated PS applications and clarifying the key application characteristics required for achieving acceleration with our proposed scheme.

Several previous studies have proposed optimization strategies for CUDA-based applications. Meng et al. [11] introduced an optimization technique called dynamic warp subdivision, which allows threads to interleave the computations of paths along different branches in order to hide memory latency. Zhang et al. [12] presented runtime optimizations that can eliminate thread divergence with a CPU-GPU pipelining scheme. They subsequently enhanced their scheme's performance by eliminating dynamic irregularities in memory references and control flows [13]. Che et al. [14] proposed a simple application program interface (API) that optimizes memory efficiency on the basis of some hints about memory access patterns. Although all these previous studies are effective for irregular applications, their optimization strategies focus on a single fixed task. In contrast, our scheme organizes multiple tasks so that the threads can eliminate irregular behaviors. To the best of our knowledge, our study is the first that tackles the issue of irregular memory references by an appropriate organization of computational tasks.

There are many projects that support parametric studies on CPU-based systems. For example, Condor [15] presented one of the first grid middleware systems to focus on idle machines for accelerating scientific applications. It provides a software framework [16] that allows users to easily parallelize applications on the grid using the master-worker paradigm. Nimrod/G [17] is a grid middleware that supports executing large-scale distributed PS. This system is equipped with a computational economic framework [18] for regulating supply and demand in order to address complex resource management issues. The AppLeS parameter sweep template [19] is another grid middleware system designed to efficiently and adaptively use computational resources managed by multiple grid environments. Our scheme can be integrated into these systems because there is no overlap between their CPU-related accomplishments and our GPU-related contribution.

3. PARAMETER SWEEP WITH CUDA

Suppose that a PS application has n parameters to be swept. We assume that each parameter involves a *task* to be processed. Let $\mathcal{T}_i$ $(1 \leq i \leq n)$ denote the task associated with parameter P_i, and let $\mathcal{I}_i$ and $\mathcal{O}_i$ $(1 \leq i \leq n)$ be the sets of input data and output data, respectively, for parameter P_i. To simplify our presentation, we assume that all the sets $\mathcal{I}_i$ and $\mathcal{O}_i$ $(1 \leq i \leq n)$ have the same number m of elements. The i-th input and output datasets can then be denoted as

$$\mathcal{I}_i = \{\ e_{i,j} \mid 1 \leq j \leq m\ \}, \tag{1}$$

$$\mathcal{O}_i = \{\ f_{i,j} \mid 1 \leq j \leq m\ \}, \tag{2}$$

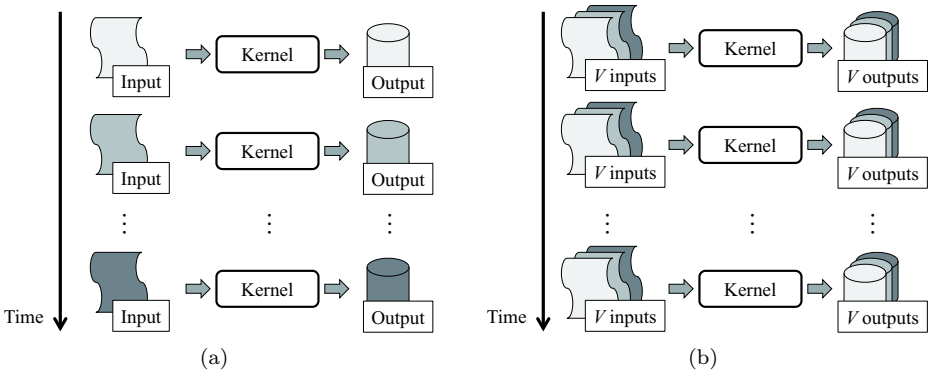


Figure 1. Parallel schemes for PS applications. (a) In the naive scheme, threads process a single parameter. (b) In our scheme, threads process multiple parameters. V denotes the number of parameters simultaneously handled by each SIMT unit. In this example, $V = 3$.

where $e_{i,j}$ and $f_{i,j}$ represent the j-th elements of the i-th input and output sets, respectively. In the case of an image processing application, $\mathcal{I}_i$ and $\mathcal{O}_i$ correspond to the input and output images, respectively, while $e_{i,j}$ and $f_{i,j}$ correspond to the pixels in the images. Using the previous notation, the subset $\mathcal{C}$ of the input data that are commonly referenced by all parameters can be identified as

$$\mathcal{C} = \bigcap_{1 \le i \le n} \mathcal{I}_i. \tag{3}$$

Given the data independence mentioned in Section 1, the master-worker paradigm can exploit coarse-grained parallelism by assigning arbitrary tasks to the worker nodes. The naive parallel scheme then attempts to accelerate single-parameter computation with single-instruction multiple-thread (SIMT) units [4] on the GPU (Fig. 1(a)). In this scheme, the worker nodes send the input data $\mathcal{I}_i$ from the main memory to the video memory, launch *kernel functions* [4] to process the corresponding task $\mathcal{T}_i$ on the GPU, and then send the output data $\mathcal{O}_i$ back to the main memory. These three processing steps are iteratively processed for $i = 1, 2, \ldots, n$.

It is possible to achieve efficient acceleration of the kernel functions' performance through the following three design components:

1. Hiding off-chip memory latency by *coalescing memory accesses*,
2. Reducing off-chip memory access by small on-chip memory, and
3. Running more resident threads [4] on the GPU by saving per-thread resource consumption.

The first design component serves to increase the effective bandwidth of the off-chip memory. The GPU is equipped with a memory coalescing technique for off-chip memory access so

that multiple transactions can be coalesced into a single transaction if the memory access patterns satisfy certain conditions [4] that depend on the compute capability of the underlying architecture. Currently available architecture can accomplish perfect coalescing if the following two conditions are satisfied: (1) a series of 32 threads (i.e., a *warp* [4]) accesses a contiguous region of memory and (2) the initial address to be accessed is aligned to the memory boundary.

The second design component saves the off-chip memory bandwidth, which is essential because the memory bandwidth rather than arithmetic performance usually determines the efficiency of kernel execution. This can be clearly seen on recent architectures, which have a low byte per floating point operation (Byte/Flop) ratio. To make matters worse, the latency of the off-chip memory is hundred times longer than the latency of the on-chip memory [4]. Off-chip memory access can be reduced by having threads belonging to the same group (the same *thread block* [4]) reuse data, because such threads are allowed to exchange data through the on-chip memory (called *shared memory*). However, the shared memory is on the order of KBs, while the off-chip memory is on the order of GBs. Tiling techniques [20] can be used to compensate for this difference in capacity by partitioning data into multiple small tiles, thus allowing data reuse in the shared memory.

Finally, the third design component is useful for hiding memory access latency with data-independent computations. Saving per-thread resource consumption allows more resident threads on the GPU, given that each thread consumes GPU resources such as register files and shared memory. Increasing the number of resident threads provides a warp scheduler with more computation-ready threads, thus facilitating warp switching for overlapping execution. The CUDA Occupancy Calculator [4] can examine the degree of resource consumption and calculate the *occupancy* of a kernel function, which is the ratio of the actual number of resident warps to the maximum number possible.

4. OUR PROPOSED PARALLEL SCHEME

To efficiently map PS applications onto the GPU, we focus on two characteristics of PS computation: all parameters $P_1, P_2, \ldots, P_n$ (1) involve the same series of operations and (2) access a common subset $\mathcal{C}$ of input data. Based on characteristic (1), our scheme exploits the SIMT units on the GPU by concurrently processing a series of multiple tasks $\mathcal{T}_i, \mathcal{T}_{i+1}, \ldots, \mathcal{T}_{i+V-1}$, where V represents the number of parameters that can be simultaneously handled by each SIMT unit (Fig. 1(b)). Currently, we set V to the warp size (so that $V = 32$), because the warps simultaneously execute each SIMT instruction. Based on characteristic (2), our scheme reduces the amount of memory consumption by combining the common data subset $\mathcal{C}$ for the V tasks so that it can be stored in the shared memory.

Figure 2 shows an overview of our parallel scheme. Unlike the naive scheme, which handles parameters one at a time, our scheme requires arranging the input and output data to enable coalesced memory access for V parameters. Although this arrangement incurs overhead, a stream processing technique [21] can be used to overlap the overhead with kernel execution if the CPU is responsible for the data arrangement. Another difference between our proposed scheme and the naive scheme is an increase in kernel execution time. Kernels running under our proposed scheme can spend V times longer than kernels running under the naive scheme to

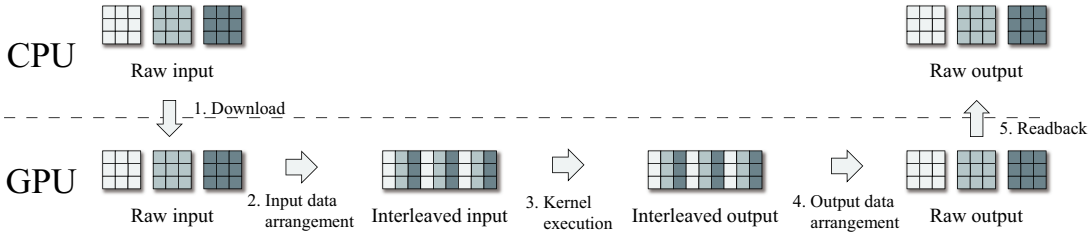


Figure 2. Overview of our parallel scheme. Our scheme arranges data before and after kernel execution so that data are stored in an interleaved manner. This data arrangement can be skipped for the common data $\mathcal{C}$.

complete their execution, because they process V parameters rather than a single parameter. However, the total number of kernel launches is decreased by a factor of roughly V, and therefore, the increased execution time is insignificant if the number of parameters to be swept is large.

In the remainder of this section, we introduce our memory coalescing and data unification techniques in order to explain how kernels can be written for our parallel scheme.

4.1. Memory Coalescing for Hiding Off-chip Memory Latency

As shown in Fig. 3, our key idea for acceleration is to arrange the input and output data $\mathcal{I}_i$ and $\mathcal{O}_i$ so that separate memory transactions can be coalesced into a single transaction. We believe that this arrangement can improve kernel performance when parameters with irregular memory access patterns have similar access strides. Because the irregular access patterns prohibit coalescing memory transactions within a single task, the naive scheme performs poorly. Therefore, we propose that the SIMT nature of PS computations can be usefully exploited to provide coalesced memory access for multiple parameters.

For the state-of-the-art architecture, memory coalescing can be achieved if the target application satisfies the following two conditions:

1. Task assignment condition: threads in the same warp are responsible for V distinct tasks $\mathcal{T}_i, \mathcal{T}_{i+1}, \ldots, \mathcal{T}_{i+V-1}$.
2. Data structure condition: input and output data for tasks $\mathcal{T}_i, \mathcal{T}_{i+1}, \ldots, \mathcal{T}_{i+V-1}$ are stored in an interleaved manner.

To satisfy the first condition, we assign tasks to warps so that each warp is responsible for V consecutive tasks $\mathcal{T}_i, \mathcal{T}_{i+1}, \ldots, \mathcal{T}_{i+V-1}$. Note that this condition cannot be achieved under the naive scheme, which assigns a single task to warps. Similar to the naive scheme, our scheme processes each task in parallel with thousands of warps.

In addition to the task assignment condition, the second condition must be satisfied for memory coalescing. Suppose that task $\mathcal{T}_i$ accesses an input element $e_{i,j}$ (where $1 \leq i \leq n, 1 \leq j \leq m$). Then the other tasks $\mathcal{T}_{i+1}, \mathcal{T}_{i+2}, \ldots, \mathcal{T}_{i+V-1}$ will probably access the input elements

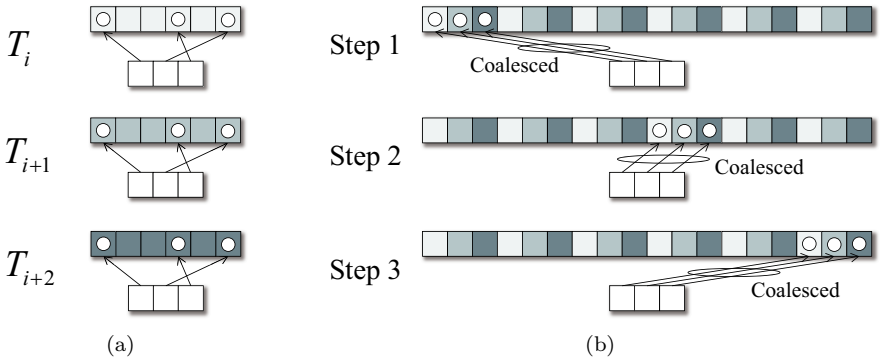


Figure 3. Data arrangement for memory coalescing. Each arrow represents a memory reference from a thread to the array element denoted by a circle. (a) Each of the tasks T_i, T_{i+1}, and T_{i+2} are degraded by irregular memory accesses that cannot be coalesced into a single transaction. (b) After arranging the data to be stored in an interleaved manner, each set of strided accesses can be coalesced into a single access if the irregularities among tasks that are simultaneously processed are similar.

$e_{i+1,j}, e_{i+2,j}, \ldots, e_{i+V-1,j}$ at the same relative address j, because PS applications apply the same sequence of operations to different parameters (see Fig. 3(a)). Therefore, we arrange the input and output data in an interleaved manner, as shown in Fig. 3(b). That is, we store the elements $e_{i,1}, e_{i,2}, \ldots, e_{i,m}$ in the sequence $e_{i,j}, e_{i+1,j}, \ldots, e_{i+V-1,j}$ rather than in the original order. This arrangement of data elements is intended to allow contiguous address spaces to be accessed by different tasks during the same clock cycle.

Note that strided accesses are essential for interleaving the data elements, and thus, the data arrangement can result in poor performance. However, similar to matrix transposition samples in the CUDA software development kit (SDK) [22], the input/output data arrangement can be efficiently implemented on the GPU by using the shared memory to avoid strided accesses to the off-chip memory. Figure 4 shows our arrangement code, which assumes that arrays for `V` tasks are contiguously stored from the `src` pointer and each array consists of `size` elements. The arrangement function `arrangement()` calls the transposition kernel `transpose()`, which generates a transposed matrix of `V` $\times$ `size` elements. Each thread block uses a shared buffer of `V` $\times$ (`V+1`) elements to transpose the input matrix of `size` $\times$ `V` elements.

Note also that the common data $\mathcal{C}$ do not require this arrangement. All tasks share the same memory region for the common data through the data unification technique presented below. Consequently, it is not necessary to arrange any input data if $\forall 1 \leq i \leq n,\ \mathcal{I}_i = \mathcal{C}$.

Given the naive code with at most two-dimensional (2-D) thread blocks, our proposed task assignment can be achieved by *thread block extension*, which increases the dimension of the thread blocks. For example, 2-D thread blocks of size $X \times Y$ can be extended to 3-D thread blocks of size $V \times X \times Y$ to allow each warp to process a series of V tasks. If the thread blocks of the naive code have 3-D indexes, which is the maximum permissible dimension in CUDA, we must reduce the dimension of the indexes before applying our technique. This reduction

```
#define V 32  // Number of tasks processed simultaneously

template<class T>
__global__ void transpose(T *dst, T *src, int size)
{
    __shared__ T buf_s[V][V + 1];  // Padding avoids bank conflicts

    // read the matrix tile into shared memory
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = threadIdx.y;
    if (x < size)
    {
        buf_s[threadIdx.x][threadIdx.y] = src[y * size + x];
    }
    __syncthreads();

    // write the transposed matrix tile to global memory
    x = threadIdx.x;
    y = blockIdx.x * blockDim.x + threadIdx.y;
    if (y < size)
    {
        dst[y * V + x] = buf_s[threadIdx.y][threadIdx.x];
    }
}

template<class T>
void arrangement(T *dst, T *src, int size)
{
    dim3 grid(size / V);
    dim3 block(V, V);

    transpose<T><<<grid, block>>>(dst, src, size);
}
```

Figure 4. A partial code of data arrangement. This code is based on a tile-based algorithm that transposes a matrix of `size` × `V` elements into that of `V` × `size` elements.

can be accomplished with 3-D to 2-D index translation, which maps the 3-D thread blocks to 2-D thread blocks. We note that the maximum size of the thread blocks (i.e., the maximum number of threads in a thread block) is limited by the underlying architecture. If the size of the translated thread blocks exceeds this maximum size, we have to reduce the number of threads per task in order to adapt the thread block organization to the limitations of the architecture. This reduction can be accomplished by *thread block decomposition* if the threads do not share data dependencies. Otherwise, if the threads are interdependent, the reduction

can be achieved by *thread aggregation*, which assigns a larger workload to each thread. Thread aggregation can also be used to adjust the number of thread blocks to the maximum permissible number. However, this procedure must be carefully applied to the kernel code so that it does not interfere with memory access coalescence.

4.2. Data Unification for Reducing Off-chip Memory Accesses

Because our scheme processes V parameters at a time, the common data $\mathcal{C}$ can be unified to reduce the amount of memory consumption for V tasks. The data unification reduces not only the size of the input data by $(V-1)|\mathcal{C}|$ but also the number of off-chip memory accesses if the common data set $\mathcal{C}$ is stored in the shared memory. The number of off-chip memory accesses can be reduced by at least a factor of V because the threads in the same warp are allowed to access $1/V$ amounts of data. In the best case scenario, our data unification technique further saves the off-chip memory bandwidth. In this case, the common data $\mathcal{C}$ can be shared among threads in the same thread block.

In our scheme, the threads in the same thread block first copy their data from the off-chip memory to the shared memory. This copy operation must be performed cooperatively to prevent redundant memory transactions. The threads can then perform their computations using the shared memory rather than the off-chip memory. Finally, they must copy their results back to the off-chip memory before the kernel completes execution.

4.3. Scheme Selection Guidelines

Our scheme has both strengths and weaknesses compared with the naive scheme. Consequently, our scheme will not always run faster than the naive scheme. Because it is not easy to predict the best scheme for an arbitrary application, we present guidelines to help developers choose the scheme for their applications. Our guidelines are based on the three design components presented in Section 3.

First, the effect of memory access coalescing depends on the memory access patterns inherent in the target algorithm; therefore, it is necessary to examine the algorithm's memory access strides. Our scheme can realize coalesced access if the algorithm produces similar access patterns (similar access strides) for V tasks. In contrast, the naive scheme can realize coalesced access if the algorithm refers to contiguous memory locations for a single task. Thus, the *similarity* and *continuity* of memory accesses determine the effectiveness of memory access coalescing. If a target algorithm exhibits both similarity and continuity or if it exhibits neither, the remaining design components must be investigated to select the better scheme. Note that our concept of continuity differs slightly from that of locality insofar as it excludes situations where there are intensive accesses to the same memory address. These situations require atomic operations, which cannot be efficiently handled under either scheme.

Second, the number of off-chip memory accesses can be reduced by not only the shared memory but also data unification. As we noted in Section 4.2, our data unification technique allows a common set of data $\mathcal{C}$ to be reused by V tasks through the shared memory. In contrast, the naive scheme uses the shared memory to perform data reuse within a single task. Because our scheme processes V tasks at a time, each task is allowed to access $1/V$ of the memory

space for data reuse. Thus, assuming that the shared memory has a limited capacity, there is a tradeoff between data reuse within a single task and data reuse among V tasks.

Third, our scheme can consume V times more memory space than the naive scheme owing to the simultaneous execution of V tasks. Because the shared memory has a limited capacity, our scheme can have fewer resident threads than the naive scheme, and consequently, our scheme's occupancy might be lower in certain cases. A similar issue concerns the possible exhaustion of register files. Thus, our scheme may be disadvantageous in terms of resource consumption. Possible solutions include reducing the number of threads or decomposing the kernel code into smaller pieces so that each thread can run with limited resources. Obviously, our scheme cannot be used directly if V tasks require memory space beyond the capacity of the off-chip memory.

These three design components characterize the kernel's performance. However, we also have to consider the overhead for data arrangement because this overhead can be a pitfall in our scheme. In particular, the overhead for arranging data must be analyzed for applications whose memory accesses exhibit both similarity and continuity and for those with neither.

5. EXPERIMENTAL RESULTS

In this section, we present the results of our experimental evaluations of our scheme's performance. The experiments were performed on a Windows 7 machine that had an NVIDIA GeForce GTX 580 card with 1.5 GB of off-chip memory with CUDA 4.2 [4] and driver version 301.32. The GPU card was attached through a peripheral component interconnect express 2.0 $\times$16 bus. The Windows machine had a 3.3 GHz Intel Core i7 2500K CPU and an 8 GB main memory. The reason why we selected a Windows machine is that it is the most familiar computing system that can be utilized as a grid resource. We also think that GPUs are primarily designed for PC games, which mainly run on Windows systems.

We applied our scheme to the practical applications described in Table I: a neural network (NN) [23], an all-pairs shortest path (APSP) algorithm [24], a joint histogram (JH) [25], and a Gaussian filter (GF) [22]. These applications were originally implemented using the naive scheme, which processes a single parameter at a time. The application code was compiled using the Visual Studio 2008 compiler at optimization level O2. Each application was executed 10 times in every experiment, and the average values were used.

Note that the naive version of GF used a thread block size of 32×8 rather than 16×8 in order to avoid bank conflicts [4] on the Fermi architecture [26], which has 32 banks in the shared memory. Note also that a task in APSP corresponds to the computation of a single-source shortest path in the given graph. Consequently, the APSP tasks require a single graph as their common input, that is, $\forall 1 \leq i \leq n, I_i = \mathcal{C}$. This implies (as noted in Section 4.1) that APSP does not require input data arrangement. In contrast, the remaining applications do require input and output data arrangement. APSP processed 32K tasks and the remaining applications processed 32 tasks during the performance measurement.

Table I. Description of experimental applications.

Application	Similarity	Continuity	Input/output arrangement	Description	Parameter
Neural network (NN) [23]	yes	no	yes/yes	Implements an artificial NN for recognition of handwritten digits. A task corresponds to recognition of a 29×29 pixel image of a digit.	32 images in float format
All-pairs shortest path (APSP) [24]	yes	no	no/yes	Computation of APSPs for a weighted graph with 32K vertices. A task computes a single-source shortest path in the graph.	32K vertices in integer format
Joint histogram (JH) [25]	no	no	yes/yes	Computation of a joint histogram for medical image registration. A task corresponds to processing a pair of $512 \times 512 \times 16$ voxel volume data.	32 floating volume data in unsigned short format
Gaussian filter (GF) [22]	yes	yes	yes/yes	A separable convolution filter of a 2-D image in the CUDA SDK. A task filters an image of 1024×1024 pixels.	32 images in float format

5.1. Code Modification

We first examined the application code to determine whether it exhibited similarity and/or continuity of memory accesses. Using the results of the analysis presented in Table II, we then modified the code so that it could process $V = 32$ tasks at a time. For all applications, we used the same transposition function to perform data arrangement (see Fig. 4). All kernels except SSSPKernel1 (for APSP) retained the same occupancy as the original kernels. Details of the code modification are as follows.

NN implements an artificial NN and consists of four kernels, each corresponding to a single layer of the artificial NN. Because the tasks share the same network, their weight coefficients can be regarded as the common data $\mathcal{C}$. All kernels were simply modified using the thread block extension described in Section 4.1. However, because the size of the thread blocks reaches the maximum permissible size in the FirstLayer kernel, we aggregated threads in order to reduce the size of the thread blocks. As a result, the original 13×13 thread block size was changed to 13×1 by thread aggregation and then to $32 \times 13 \times 1$ by thread block extension.

As shown in Table II, APSP implements an iterative algorithm with two kernels. The inputs to the first kernel are the common data $\mathcal{C}$; therefore, we stored them in the shared memory. The original 256×1 thread block size was changed to 4×1 by thread block decomposition and then to 32×4 by thread block extension. Similarly, the second kernel used 64×4 as the thread block size instead of the original 256×1 size. We chose 64 instead of 32 because the thread block size 32×4 causes partition camping in this kernel, which can degrade the kernel performance by as much as sevenfold [27].

JH computes a joint histogram for a pair consisting of the reference and floating data. The reference data are fixed whereas the floating data are changed during PS computation. Consequently, the reference data can be regarded as the common data $\mathcal{C}$ and thus can be

Table II. Kernel code analysis. Data sizes are presented as per-parameter values so that they must be multiplied by $V = 32$ for our scheme. The location column indicates the location where the original code stores the corresponding variable. G, S, and C in this column stand for global memory, shared memory, and constant memory.

App.	Kernel	Variable	Classification	Similarity	Continuity	Unified	Size (B)	Location
NN	FirstLayer	Layer1_Neurons	$\mathcal{I}$	yes	no	no	3.6 K	G
		Layer1_Weights	$\mathcal{C}$	yes	yes	yes	0.6 K	G
		Layer2_Neurons	$\mathcal{O}$	yes	yes	no	4.0 K	G
	SecondLayer	Layer2_Neurons	$\mathcal{I}$	yes	no	no	4.0 K	G
		Layer2_Weights	$\mathcal{C}$	yes	yes	yes	30.5 K	G
		Layer3_Neurons	$\mathcal{O}$	yes	yes	no	4.9 K	G
	ThirdLayer	Layer3_Neurons	$\mathcal{I}$	yes	no	no	4.9 K	G
		Layer3_Weights	$\mathcal{C}$	yes	no	yes	488.7 K	G
		Layer4_Neurons	$\mathcal{O}$	yes	no	no	0.4 K	G
	ForthLayer	Layer4_Neurons	$\mathcal{I}$	yes	no	no	0.4 K	G
		Layer4_Weights	$\mathcal{C}$	yes	no	yes	4.0 K	G
		Layer5_Neurons	$\mathcal{O}$	yes	no	no	40.0	G
APSP	SSSPKernel1	Va	$\mathcal{C}$	yes	yes	yes	128 K	G
		Ea	$\mathcal{C}$	yes	no	yes	512 K	G
		Wa	$\mathcal{C}$	yes	no	yes	512 K	G
		Ma	$\mathcal{O}$	yes	yes	no	128 K	G
		Ca	$\mathcal{O}$	yes	yes	no	128 K	G
		Ua	$\mathcal{O}$	yes	no	no	128 K	G
	SSSPKernel2	Ma	$\mathcal{I}$	yes	yes	yes	128 K	G
		Ca	$\mathcal{I}$	yes	yes	no	128 K	G
		Ua	$\mathcal{I}$	yes	yes	no	128 K	G
		F	$\mathcal{O}$	yes	yes	yes	4	G
JH	Histogram	fdat	$\mathcal{I}$	yes	yes	no	8 M	G
		rdat	$\mathcal{C}$	yes	yes	yes	8 M	G
		hist2d	$\mathcal{O}$	no	no	no	256 K	G
GF	Rows	d_Dst	$\mathcal{I}$	yes	yes	no	4 M	G+S
		c_Kernel	$\mathcal{C}$	yes	yes	yes	68	C
		d_Src	$\mathcal{O}$	yes	yes	no	4 M	G
	Columns	d_Dst	$\mathcal{I}$	yes	yes	no	4 M	G+S
		c_Kernel	$\mathcal{C}$	yes	yes	yes	68	C
		d_Src	$\mathcal{O}$	yes	yes	no	4 M	G

unified accordingly. The original 32×32 thread block size was changed to 32×1 by thread block decomposition and then to $32 \times 32 \times 1$ by thread block extension.

GF applies a separable convolution to a 2-D signal with a Gaussian function. This filter is implemented by two kernels, one responsible for 1-D convolution in the row direction and the other responsible for that in the column direction. Both kernels store Gaussian coefficients in the constant memory that can be reused for the different tasks. The kernels are also accelerated using a tiling technique, which allows the reuse of temporal data through the shared memory. The 32×8 thread block size was changed to 8×1 by thread block decomposition and then to $32 \times 8 \times 1$ by thread block extension.

Figure 5 shows a simplified version of the naive JH code and Fig. 6 shows the corresponding part of our parallel JH code. By applying thread block decomposition to the naive code, the

```
1  #define BLOCK_XSIZE 32
2  #define BLOCK_YSIZE 32
3
4  __global__
5  void Histogram(short *rdat, short *fdat, dim3 size, int *hist2d)
6  {
7      // (x,y,*): responsible coordinates
8      int x = blockIdx.x * blockDim.x + threadIdx.x;
9      int y = blockIdx.y * blockDim.y + threadIdx.y;
10
11     int pos = size.x * y + x;
12     int base = 0;
13
14     for (int z=0; z<size.z; z++)
15     {
16         int f = fdat[base + pos];
17         int r = rdat[base + pos];
18         atomicAdd(&hist2d[WIDTH * f + r], 1); // histogram width
19         base += size.x * size.y;
20     }
21 }
22
23 void MakeHistogram(short *rdat, short *fdat, dim3 size, int *hist2d)
24 {
25     dim3 block(BLOCK_XSIZE, BLOCK_YSIZE);
26     dim3 grid(size.x / BLOCK_XSIZE, size.y / BLOCK_YSIZE);
27
28     Histogram<<<grid, block>>>(rdat, fdat, size, hist2d);
29 }
```

Figure 5. A simplified code of naive JH. A joint histogram `hist2d` is produced from a pair of `size.x`×`size.y`×`size.z` voxel images `rdat` and `fdat`. Each thread is responsible for `size.z` voxels.

thread block size `block` is decreased from `(BLOCK_XSIZE, BLOCK_YSIZE)` to `(BLOCK_XSIZE, 1)` while the grid size `grid` is increased from `(size.x / block.x, size.y / block.y)` to `(size.x / block.x, size.y)` accordingly. Thread block extension then further changes the thread block size to `(V, BLOCK_XSIZE, 1)`, as shown in Fig. 6. Furthermore, the coordinate `(x,y)` at lines 7 and 8 is adapted accordingly. Finally, memory offsets are adapted to access interleaved data correctly. For interleaved data, the original offset `p` is replaced with `p * V + threadIdx.x`. In contrast, the offset `p` for unified data is kept as is.

```
#define V 32
#define BLOCK_XSIZE 32

__global__
void HistogramPS(short *rdat, short *fdat, dim3 size, int *hist2d)
{
    // (x,y,*): responsible coordinates
    int x = blockIdx.x * blockDim.y + threadIdx.y;
    int y = blockIdx.y;

    int pos = size.x * y + x;
    int base = 0;

    for (int z=0; z<size.z; z++)
    {
        int f = fdat[(base + pos) * V + threadIdx.x]; // interleaved
        int r = rdat[base + pos]; // unified
        atomicAdd(&hist2d[(WIDTH * f + r) * V + threadIdx.x], 1);
        base += size.x * size.y;
    }
}

void MakeHistogramPS(short *rdat, short *fdat, dim3 size, int *hist2d)
{
    dim3 block(V, BLOCK_XSIZE, 1);
    dim3 grid(size.x / BLOCK_XSIZE, size.y);

    HistogramPS<<<grid, block>>>(rdat, fdat, size, hist2d);
}
```

Figure 6. A simplified code of our parallel JH. The original 32×32 thread block size was changed to 32×1 by thread block decomposition and then to $32 \times 32 \times 1$ by thread block extension. `fdat` and `hist2d` are interleaved for `V` tasks, whereas `rdat` is unified.

5.2. Performance Comparison

We compared our scheme's performance with that of the naive scheme. Figure 7 shows the execution times for our scheme and the naive scheme. Figure 8 also shows the breakdowns of the execution times, including the data readback and download times, the input and output arrangement times, and the kernel execution time.

For all applications, our scheme runs faster than the naive scheme. In particular, the speedups of our scheme over the naive scheme reach a factor of 7.7 for NN and 8.5 for APSP. Comparison of kernel performance revealed that our NN and APSP kernels run 6.4 to 21.8 times faster than the naive kernels. As shown in Table I, both applications exhibit similarity

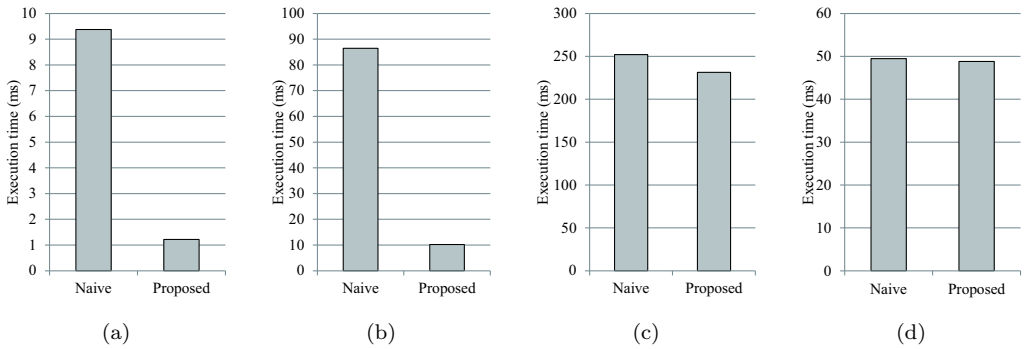


Figure 7. Performance comparisons for (a) NN, (b) APSP, (c) JH, and (d) GF.

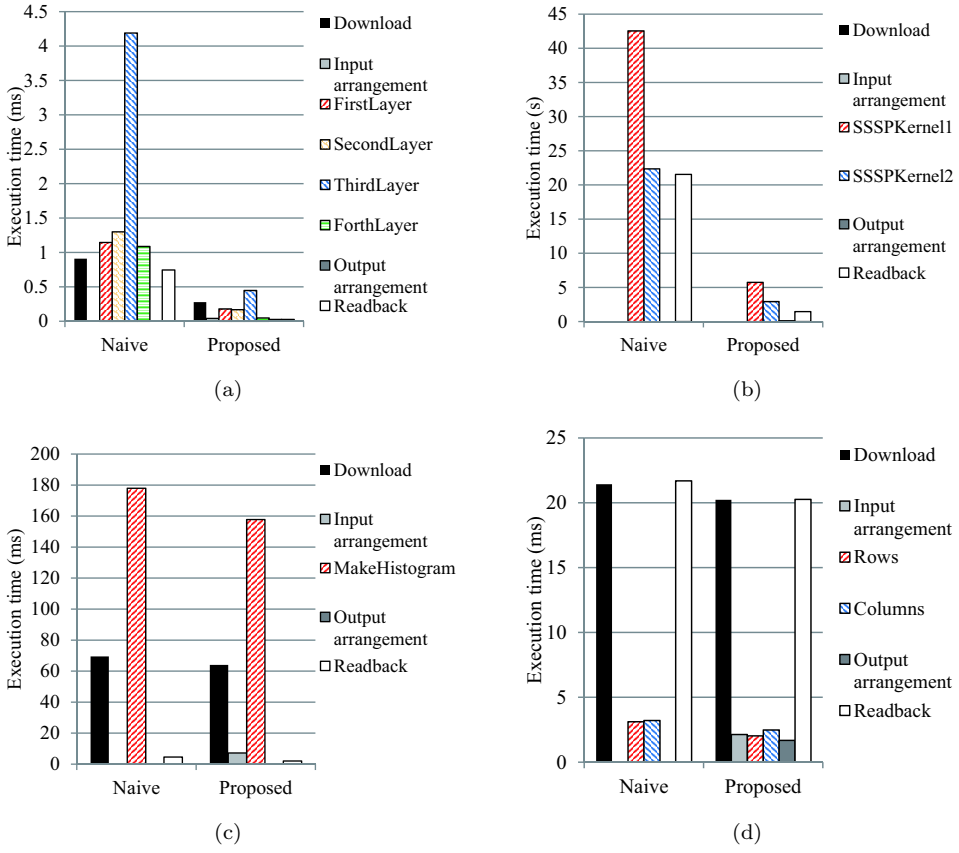


Figure 8. Breakdowns of execution times for (a) NN, (b) APSP, (c) JH, and (d) GF.

of memory accesses but lack continuity. In these cases, our scheme takes advantage of memory coalescing as described in Section 4.3. For example, the original version of NN suffers from noncoalesced accesses because irregular connections between the neurons make it difficult to realize coalesced access on the GPU. Our scheme eliminates such irregular access patterns by simultaneously running multiple NNs. The irregularity can be found in not only the memory operations but also the instructions. With the help of the CUDA Profiler, we found that our scheme reduced the number of branches in the four kernels from approximately 292K to 15K. We also found that the overhead for data arrangement is acceptably small in this application: 5.7% of the execution time was used to interleave the data for 32 tasks.

Similar to the NN kernels, the APSP kernels achieve significant speedups, ranging from a factor of 7.4 to a factor of 7.7. The main reason for this acceleration is a reduction in the number of divergent branches. This number is reduced from 90M to 25M in SSSPKernel1 and from 6M to 2M in SSSPKernel2. Moreover, our kernels use the shared memory to store the common data $\mathcal{C}$ presented in Table II. Consequently, SSSPKernel1 replaces 64% of the global memory accesses with shared memory accesses. Thus, the simultaneous processing of multiple tasks serves to eliminate irregular instruction patterns within warps (i.e., divergent warps) and to exploit shared memory, neither of which can be accomplished by a single task in this application.

Note that the readback times of NN and APSP are reduced by factors of 25 and 15, respectively. These reductions are primarily due to our task organization, which coalesces V readback operations into a single operation. For example, 32 readbacks of 40 B blocks of data are coalesced into a single readback of a 1280 B block of data in NN. In particular, our scheme efficiently reduces readback time if each readback operation transfers a small amount of data. In this case, the transfer latency rather than the transfer bandwidth dominates the readback time. In contrast, there is no significant reduction in the corresponding results for JH and GF because their readback times are determined by the transfer bandwidth.

In contrast to NN and APSP, our scheme runs the remaining applications slightly faster than the naive scheme does. JH exhibits neither similarity nor continuity of memory accesses and thus neither scheme can achieve coalesced access for JH. However, our kernel runs 1.1 times faster than the naive kernel because it increases the L1 cache hit rate from 0% to 73%. Thus, this improvement is data dependent, meaning that our scheme will not always run faster than the naive scheme. However, JH deals with clinical datasets, which have similar distributions in terms of X-ray beam intensities. In this case, the similarity of intensities between multiple datasets can be higher than that within a single dataset. Thus, the similarity of datasets explains our scheme's performance improvement.

In contrast, both schemes achieve coalesced access for GF, which exhibits both similarity and continuity of memory accesses. However, our row and column kernels run 1.8 and 1.5 times faster, respectively, than the naive kernels. This acceleration is due to an increase in the L2 cache hits: we found that the number of these hits increased by factors of 2.3 and 1.4, respectively, for the row and column kernels. The increases can be explained as follows. The GTX 580 card has 768 KB of L2 cache and the input image consists of 1024×1024 pixels, each containing 4 bytes of data. With the naive scheme, the L2 cache can store 192 rows of the image at any time. On the other hand, it can store 6 rows of the interleaved image for our scheme. The number of thread blocks that access these rows during row convolution reaches

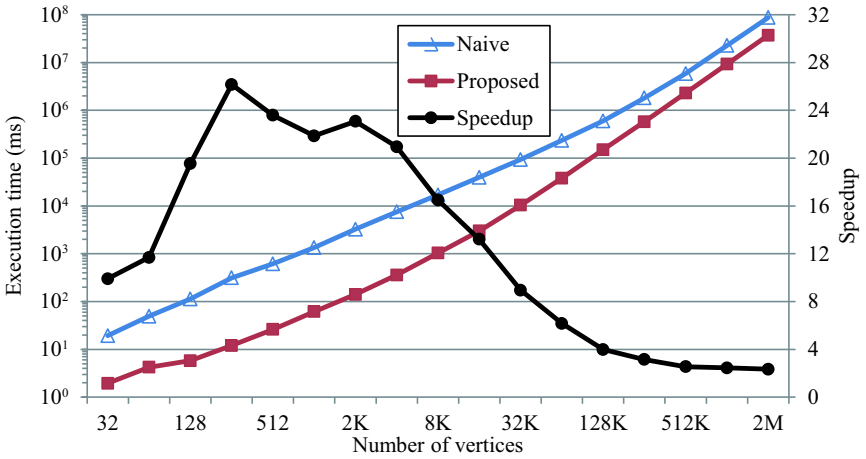


Figure 9. Execution times and speedups of APSP for different numbers of vertices. Execution times are presented in the log scale.

96 ($= 4 \times 24$) in the naive scheme and 256 ($= 128 \times 2$) in our scheme, because both schemes use the thread block size 32×8 and each thread is responsible for 8 columns. Therefore, our scheme allows more than twice the number of thread blocks to access data in the L2 cache, producing higher locality. Although our scheme produces efficient kernels, this advantage is canceled by the arrangement overhead.

5.3. Scalability Analysis

Finally, we investigated how our scheme's performance scales with respect to the problem size. This scalability analysis intends to increase the data size of elements rather than the number of elements (i.e., the granularity of tasks rather than the number of tasks), because the performance behavior of the latter case can be easily estimated by multiplying execution time by the number of kernel invocations needed to process all elements. To perform the scalability analysis, we executed APSP and JH with different problem sizes. NN was not investigated for this analysis, because this application required network construction for each image size, and the network was hardcoded inside the kernel functions.

Figure 9 shows the execution times and speedups of APSP with varying numbers of vertices ranging from 32 to 2M. As we increased the number of vertices from 32 to 256, the speedup increased from a factor of 9.9 to a factor of 26.2. Then, the speedup decreased and converged to a factor of approximately 2.5 as we increased the number of vertices. A shortage of resident threads explains this behavior. The naive scheme creates N threads for N vertices, whereas our scheme runs $32N$ threads for N vertices. Consequently, when $N \leq 2048$, the naive scheme performs poorly owing to a lack of a sufficient number of resident threads to hide memory access latency. The performance improves when $N > 2048$, but the use of shared memory makes our scheme run faster than the naive scheme.

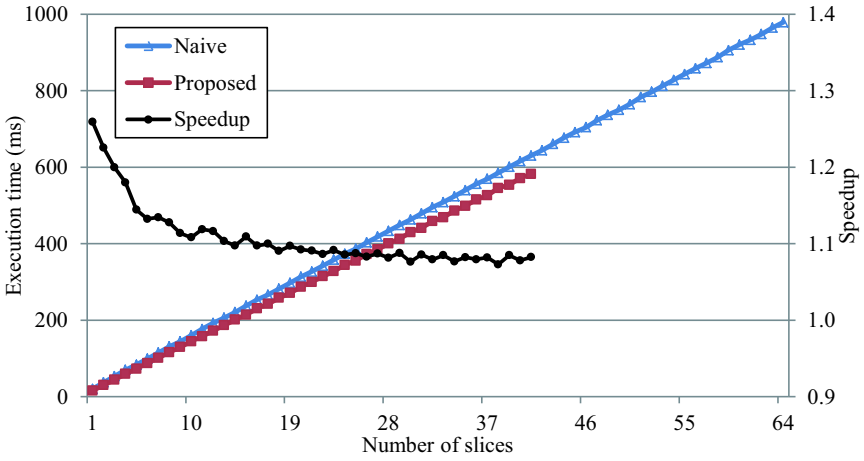


Figure 10. Execution times and speedups of JH for different numbers of slices. Our scheme cannot simultaneously execute more than 41 slices.

Figure 10 shows the results of JH for different numbers of 512×512 pixel slices, ranging from 1 to 64. As this figure shows, our scheme cannot execute more than 41 slices owing to the lack of off-chip memory. The bottleneck in the code is the data arrangement procedure, which consumes 64 times more memory space than the naive scheme: 32 times more because there are 32 tasks and then 2 times more for the input and output data. Thus, our scheme can fail to process applications that use large datasets. However, JH does not have complex data dependencies within its computation. Therefore, the entire volume of data can be decomposed into small blocks that can be processed sequentially. Consequently, we can process datasets with more than 41 slices by launching the kernel iteratively, each time for 41 slices.

Figure 11 shows the results of GF for different image sizes ranging from 256×256 pixels to 2048×2048 pixels. Similar to the results of JH, our scheme cannot execute more than 1280×1280 pixels. The speedup for 256×256 pixels is higher than that for other pixels. For 256×256 pixel images, the naive scheme performs poorly owing to a lack of a sufficient number of thread blocks to utilize all CUDA cores. In contrast, our scheme avoids such idle cores by processing V tasks simultaneously.

5.4. Discussion

Our parallel scheme takes advantages of a memory coalescing technique and an explicitly-managed cache (i.e., shared memory), which are available on recent NVIDIA cards. Consequently, our scheme is efficient also for Tesla and Quadro cards. However, the execution configuration must be tuned to maximize the effective performance on each card, because GPUs exhibit different performance characteristics with different amounts of resources [28]. For example, we applied this kind of optimization to GF in order to avoid bank conflicts, as we mentioned in Section 5.

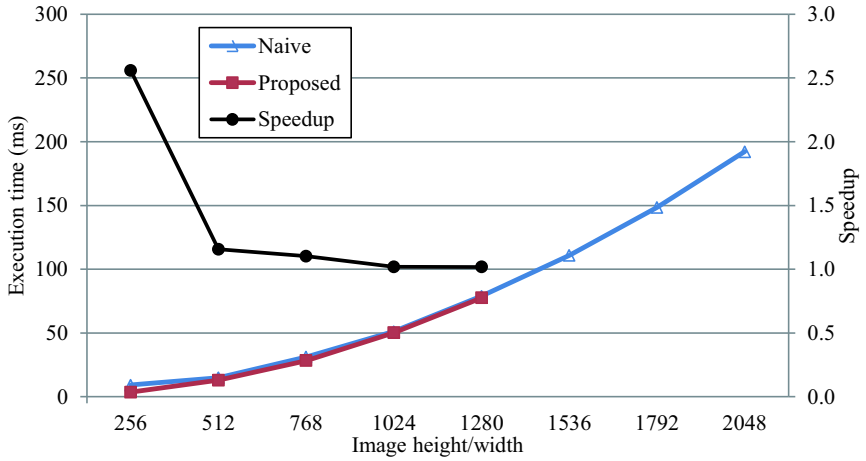


Figure 11. Execution times and speedups of GF for different image sizes. Our scheme cannot simultaneously execute more than 1280 × 1280 pixels.

Because our parallel scheme assumes the CUDA memory architecture, it cannot be directly applied to CPU-based HPC systems. However, our data arrangement procedure may increase the cache hit rate if it improves the locality of reference. The procedure may also be useful to minimize the memory access stride for single-instruction, multiple-data (SIMD) instructions, such as streaming SIMD extensions (SSE) [29].

We think that our data arrangement process is generic for regular data but not for irregular data. For example, the transposition function in Fig. 4 cannot generate an efficient data structure for graphs with different topologies and sizes, because our parallel scheme probably fails to access them in a coalesced manner. With respect to APSP, its input graph can be regarded as irregular data, but it does not require data arrangement, as we mentioned in Section 5.1. Because our transposition function separates the input data from the output data, developing an in-place transposition algorithm is left as a future work.

6. CONCLUSION

We have presented a parallel scheme for accelerating PS applications with CUDA. Our scheme improves performance by simultaneously processing multiple parameters rather than a single parameter. It interleaves the input and output data to coalesce memory access for multiple parameters, and it saves the off-chip memory bandwidth by using the shared on-chip memory to store common data that can be accessed by the multiple parameters.

We conducted experiments in which we applied our scheme to four practical applications. Our scheme performed 8.5 times better for a graph application than the naive scheme, which processes a single parameter at a time. In particular, our kernels run faster than naive kernels if they have irregular access patterns within a single task but have similar access strides

across multiple tasks (we characterized such access patterns on the basis of the concept of similarity and continuity of memory accesses). Therefore, we believe that our scheme is useful for addressing the problem of irregular memory accesses if the memory accesses cannot be coalesced for a single parameter. The regularization effect of our task organization scheme can be found in not only memory operations but also instructions.

In addition, we identified some disadvantages of our scheme. Similarity of memory accesses can be successfully exploited if the performance benefit of memory coalescing is large enough to compensate for the overhead incurred by the data arrangement. For some kernels, we need to reduce the thread block size and the usage of shared memory because our scheme consumes roughly $V = 32$ times more resources than the naive scheme.

ACKNOWLEDGEMENTS

This study was partially supported by JSPS KAKENHI Grants 23300007 and 23700057 and by the JST CREST program "An Evolutionary Approach to Construction of a Software Development Environment for Massively-Parallel Computing Systems." The authors would like to thank the anonymous reviewers for helpful comments to improve their paper.

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. C. Youn and T. Kaiser, "Management of a parameter sweep for scientific applications on cluster environments," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 18, pp. 2381–2400, Dec. 2010.
3. 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.
4. NVIDIA Corporation, "CUDA Programming Guide Version 4.2," Apr. 2012. [Online]. Available: http://developer.nvidia.com/cuda/
5. 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.
6. The Folding@Home Project, "Folding@home distributed computing," 2010, [Online]. Available: http://folding.stanford.edu/
7. F. Ino, Y. Munekawa, and K. Hagihara, "Sequence homology search using fine grained cycle sharing of idle GPUs," *IEEE Trans. Parallel and Distributed Systems*, vol. 23, no. 4, pp. 751–759, Apr. 2012.
8. M. Motokubota, F. Ino, and K. Hagihara, "Accelerating parameter sweep applications using CUDA," in *Proc. 19th Euromicro Int'l Conf. Parallel, Distributed and Network-Based Computing (PDP'11)*, Feb. 2011, pp. 111–118.
9. F. Varray, C. Cachard, A. Ramalli, P. Tortoli, and O. Basset, "Simulation of ultrasound nonlinear propagation on GPU using a generalized angular spectrum method," *EURASIP J. Image and Video Processing*, vol. 2011, no. 17, Nov. 2011, 6 pages.
10. Y. Munekawa, F. Ino, and K. Hagihara, "Accelerating Smith-Waterman algorithm for biological database search on CUDA-compatible GPUs," *IEICE Trans. Information and Systems*, vol. E93-D, no. 6, pp. 1479–1488, Jun. 2010.
11. J. Meng, D. Tarjan, and K. Skadron, "Dynamic warp subdivision for integrated branch and memory divergence tolerance," in *Proc. 37th Annual Int'l Symp. Computer Architecture (ISCA'10)*, Jun. 2010, pp. 235–246.
12. E. Z. Zhang, Y. Jiang, Z. Guo, and X. Shen, "Streamlining GPU applications on the fly," in *Proc. 24th ACM Int'l Conf. Supercomputing (ICS'10)*, Jun. 2010, pp. 115–125.

13. E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen, "On-the-fly elimination of dynamic irregularities for GPU computing," in *Proc. 16th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*, Mar. 2011, pp. 369–380.
14. S. Che, J. W. Sheaffer, and K. Skadron, "Dymaxion: Optimizing memory access patterns for heterogeneous systems," in *Proc. Int'l Conf. High Performance Computing, Networking, Storage and Analysis (SC'11)*, Nov. 2011, 11 pages (CD-ROM).
15. 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.
16. 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.
17. R. Buyya, D. Abramson, and J. Giddy, "Nimrod/G: An architecture for a resource management and scheduling system in a global computational grid," in *Proc. 4th Int'l Conf. High Performance Computing in Asia-Pacific Region (HPC ASIA'00)*, May 2000.
18. R. Buyya, D. Abramson, J. Giddy, and H. Stockinger, "Economic models for resource management and scheduling in Grid computing," *Concurrency and Computation: Practice and Experience*, vol. 14, no. 13/15, pp. 1507–1542, Dec. 2002.
19. 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.
20. 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).
21. 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.
22. NVIDIA Corporation, "GPU Computing SDK," 2012. [Online]. Available: http://developer.nvidia.com/gpu-computing-sdk/
23. billconan and kavinguy, "A neural network on GPU," 2008. [Online]. Available: http://www.codeproject.com/Articles/24361/A-Neural-Network-on-GPU
24. P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in *Proc. 14th Int'l Conf. High Performance Computing (HiPC'07)*, Dec. 2007, pp. 197–208.
25. K. Ikeda, F. Ino, and K. Hagihara, "Accelerating joint histogram computation for image registration on the GPU," in *Proc. Computer Assisted Radiology and Surgery: 26th Int'l Congress and Exhibition (CARS'12)*, Jun. 2012, pp. S72–S73.
26. NVIDIA Corporation, "NVIDIA's Next Generation CUDA Compute Architecture: Fermi," Nov. 2009, [Online]. Available: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf
27. A. M. Aji, M. Daga, and W. chun Feng, "Bounding the effect of partition camping in GPU kernels," in *Proc. 8th Int'l Conf. Computing Frontiers (CF'11)*, May 2011, 10 pages.
28. A. Nukada and S. Matsuoka, "Auto-tuning 3-d FFT library for CUDA GPUs," in *Proc. Int'l Conf. High Performance Computing, Networking, Storage and Analysis (SC'09)*, Nov. 2009, 10 pages (CD-ROM).
29. A. Klimovitski, "Using SSE and SSE2: Misconceptions and reality," in *Intel Developer Update Magazine*, Mar. 2001.