2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing

Cooperative Multitasking for GPU-Accelerated Grid Systems

Fumihiko Ino, Akihiro Ogita, Kentaro Oita and Kenichi Hagihara
Graduate School of Information Science and Technology
Osaka University
1-5 Yamadaoka, Suita, Osaka 565-0871, Japan
Email: ino@ist.osaka-u.ac.jp

***Abstract*—Exploiting the graphics processing unit (GPU) is useful to obtain higher performance with a less number of host machines in grid systems. One problem in GPU-accelerated grid systems is the lack of efficient multitasking mechanisms. In this paper, we propose a cooperative multitasking method capable of simultaneous execution of a graphics application and a CUDA-based scientific application on a single GPU. To prevent significant performance drop in frame rate, our method (1) divides scientific tasks into smaller subtasks and (2) serially executes them at the appropriate intervals. Experimental results show that the proposed method is useful to control the frame rate of the graphics application and the throughput of the scientific application. For example, matrix multiplication can be processed at 50% of the dedicated throughput while achieving interactive rendering at 54 frames per second.**

***Keywords*-multitasking; GPU; CUDA; grid;**

I. Introduction

The graphics processing unit (GPU) [1], [2] is an accelerator originally designed for graphics applications. It provides us high memory bandwidth and floating-point performance with a single-instruction, multiple-data (SIMD) capability. Furthermore, it now has a flexible development framework, called compute unified device architecture (CUDA) [3], demonstrating many acceleration results typically with a 10-fold speedup over CPU-based implementations. In CUDA, the compute-intensive code is usually implemented as a *kernel*, namely a function that runs on the GPU.

The GPU is also emerging as a powerful computational resource in grid environments, where distributed resources are virtually collected into an integrated system. For example, the Folding@home project [4] demonstrates that 70% of the entire performance is provided by idle GPUs, which account for only 10% of all resources available in the system. Thus, exploiting the GPU is useful to obtain higher performance with a less number of host machines in grid systems. In this paper, we denote *hosts* as users who donate their resources to the grid system. On the other hand, *guests* are grid users who submit jobs to the system.

One problem in GPU-accelerated grid systems [5], [6] is the lack of efficient multitasking mechanisms. For example, the frame rate of graphics applications and the throughput of scientific applications can significantly drop when they are simultaneously executed on the same GPU [7], [8]. This is due to the current GPU architecture, which (1) allows only a single kernel to run at a time and (2) switches the running kernel only when it completes its execution. Therefore, the frame rate will significantly drop if the guest kernel occupies the resources for long time.

To avoid this problem, current systems [4], [5] use screensavers capable of detecting fully idle GPUs for acceleration of guest applications. In other words, both guest and host applications are exclusively executed in the systems, where idle resources are typically dedicated to guest applications. In contrast to these dedicated systems, we are focusing on non-dedicated systems, which have a true resource sharing mechanism to harness the power of GPUs in the home and office. For example, such mechanisms allow us to run guest applications on lightly loaded GPUs, where host users operate their machines for office work with almost no workload on the GPU.

Our main goal is to achieve true resource sharing between hosts and guests for GPU-accelerated grid systems. To achieve this, we are developing a cooperative multitasking method capable of simultaneous execution of a graphics application and a CUDA-based scientific application. Using this method, screensavers are not needed to ensure exclusive execution of guest and host applications. Instead, hosts who accept scientific jobs from guests should specify the minimum frame rate they need. According to this requirement, the proposed method divides the workload of scientific applications such that each kernel can complete its execution within the desired period. The method assumes that guest applications are implemented using CUDA while host applications are implemented using a periodical rendering model with graphics libraries such as DirectX [9] and OpenGL [10].

The rest of this paper is organized as follows. Section II presents preliminaries including an overview of CUDA and the periodical rendering model. Section III then describes the details of our multitasking method and Section IV shows experimental results. Finally, Section V summarizes the paper with future work.

II. Preliminaries

This section presents preliminaries needed to understand our method.

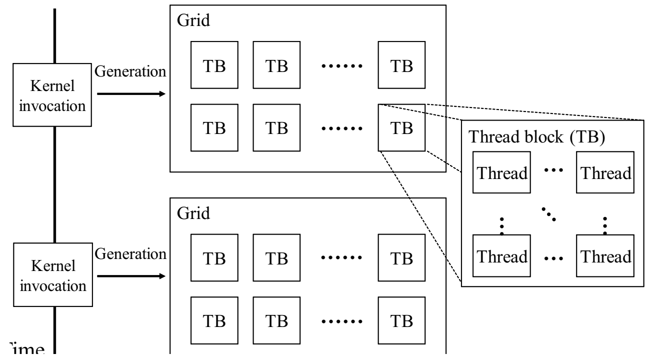


Figure 1. Hierarchical thread structure in CUDA. The CPU code invokes kernels, which generate thousands of threads to accelerate computation on the GPU. An implicit global synchronization is performed between successive invocations.

A. Compute Unified Device Architecture (CUDA)

CUDA [3] is a flexible development framework for the NVIDIA GPU. Since it is based on an extension of the C language, it allows us to easily implement GPU-accelerated applications, which consist of kernels and CPU code. CUDA-based kernels typically generate millions of threads on the GPU, which then accelerate heavy computation by SIMD instructions on multiprocessors (MPs). Currently, the number M of MPs ranges from 16 to 30 depending on the GPU.

Figure 1 illustrates an overview of the hierarchical thread structure in CUDA. In CUDA programs, threads are classified into thousands of groups, each called as thread blocks (TBs). Threads belonging to the same TB are allowed to synchronize each other, so that such threads can share data using fast on-chip memory, called shared memory. On the other hand, data dependencies between different TBs are not allowed in the kernel. Therefore, we have to separate the kernel into multiple pieces to deal with such dependencies. Separated kernels are then serially executed with global synchronization.

TB is the minimum unit allocated to MPs. Therefore, the number n of TBs should be a multiplier of M for load balancing between MPs. It also should be noted here that the GPU architecture is designed to hide the memory latency. This is achieved by concurrently running multiple TBs on each MP. Since there is no data dependence between different TBs, each MP is allowed to switch them to perform computation during a memory fetch operation. To maximize the effects of this latency hiding, MPs run more TBs as long as registers and shared memory are available.

B. Periodical Rendering Model

There are two typical rendering models that can be employed for graphics applications: a periodical model and a non-periodical model. The former is intended to provide the same frame rate on any graphics card. For example, PC games and movie players are required to ensure the same rate to avoid extremely high frame rates per second (fps). In contrast, the latter executes rendering tasks on the GPU as fast as possible. A rendering task here corresponds to the drawing code flushed by the glFlush() function, which produces a single frame in graphics applications. On the other hand, a CUDA task in scientific applications corresponds to a kernel invocation. As we mentioned in Section I, our method assumes that host applications are implemented using the periodical model.

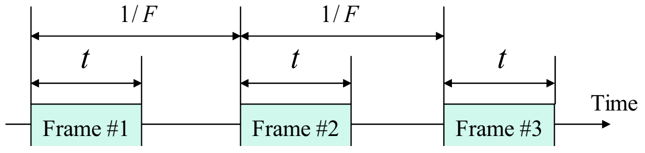


Figure 2. Periodical rendering model. Rendering tasks are executed at regular intervals to produce a series of frames.

Figure 2 shows how the periodical model produces a series of frames. In this model, rendering tasks are executed at regular intervals. The intervals are given by $1/F$, where F denotes the frame rate needed for the graphics application. Let t be the time spent for producing a frame. The idle period W between frames then can be represented by

$$W = 1/F - t. \tag{1}$$

According to the model mentioned above, the frame rate F can be decreased if $W < 0$. One problem here is that the time t cannot be directly measured in grid environments because it is not realistic to obtain and modify the source code of arbitrary host applications. It is important to take this constraint into account when developing grid middleware.

III. Proposed Multitasking Method

Figure 3 illustrates how guest and host applications are executed under the periodical rendering model. According to our preliminary experiments, we find that rendering tasks and CUDA tasks are always executed in exclusive mode. That is, the GPU switches tasks when the current kernel completes its execution. Therefore, rendering tasks can be delayed or cancelled if a scientific task occupies the GPU for long time, as shown in Fig. 3(a).

To solve this problem, we have to control the execution time of guest kernels such that the host application can process a rendering task at every time $1/F$. Figure 3(b) illustrates how our cooperative method realize efficient multitasking on the GPU. The proposed method consists of two strategies as follows.

1) Task division. The method divides guest tasks into smaller subtasks such that each kernel completes its execution within the idle period W.
2) Alternative execution. The method invokes the scientific kernel at almost the same intervals as the host

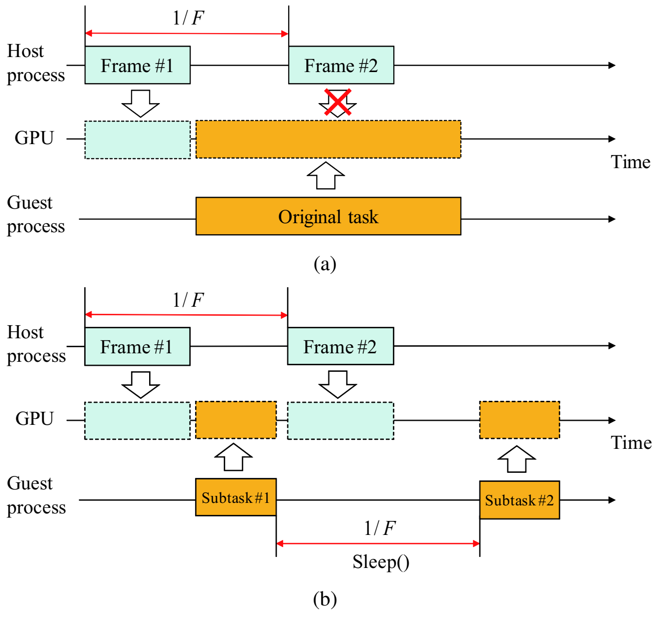


Figure 3. Timeline view of multi-task execution on the GPU. (a) A naive method will drop the frame rate but (b) our method prevents this performance degradation. The proposed method divides each of scientific tasks into smaller subtasks, which are then serially processed at regular intervals.

```
OriginalMatrixMultiplication()
 1: // setup execution parameters
 2: dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
 3: dim3 grid(WC / threads.x, HC / threads.y);
 4: // execute the kernel
 5: matrixMul<<< grid, threads >>>(d_C, d_A, d_B, WA, WB);
```

```
__global__ void
     marixMul(float *C, float *A, float *B, int wA, int wB) {
 6:      // Block index
 7:      int bx = blockIdx.x;
 8:      int by = blockIdx.y;
 9:      ... // omitted
10: }
```

(a)

```
ModifiedMatrixMultiplication(GRID_YSIZE)
 1: // setup execution parameters
 2: dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
 3: dim3 grid(WC / threads.x, GRID_YSIZE);
 4: // execute the kernel
 5: for (int i=0; i<HC / threads.y / GRID_YSIZE; i++) {
 6:      matrixMul<<< grid, threads >>>
              (d_C, d_A, d_B, WA, WB, i*GRID_YSIZE);
 7:      cudaThreadSynchronize();
 8:      Sleep(1/F);
 9: }
```

```
__global__ void
     marixMul(float *C, float *A, float *B, int wA, int wB,
     int offset) {
10:      // Block index
11:      int bx = blockIdx.x;
12:      int by = offset + blockIdx.y;
13:      ... // omitted
14: }
```

(b)

Figure 4. Example of guest code modifications. (a) The original code [3] for matrix multiplication $C = AB$ and (b) the modified code for cooperative multitasking. WC and HC represent the width and the height of matrix C. The input parameter GRID_YSIZE determines the number of TBs.

application. This prevents resource starvation because host and guest applications are given equal chances to use the GPU.

Note that our method requires code modifications of guest applications. In contrast, there is no need to modify the code of graphics applications that run as host applications on grid resources. This is essential to run the method in grid environments, where there can be a large number of host applications and their code is not allowed to edit. In this sense, the method provides a realistic solution to the problem of multitasking in grid environments.

A. *Task Division*

We now explain how task division can be done for CUDA-based applications. As we mentioned in Section II-A, there is no data dependence between different TBs. Therefore, our method divides the original task into smaller subtasks by simply reducing the number n of TBs given to the CUDA kernel. In addition to this task division, it serially invokes the kernel with changing TBs to obtain the same results as before.

Our division strategy has an advantage in code modification. Firstly, we use the same TB size as the original code. This means that kernel optimization inherent in TBs is kept as the original. For example, we do not have to concern about reducing the degree of parallelism available in each TB. One exception is that the memory latency hiding mentioned in Section II-A can be cancelled due to the reduced number of TBs. The second advantage is that almost all of the original kernel code can be reused after task division. The modification only needed is that we have to add an offset as a kernel argument and have to specify the appropriate address of output data by using the offset.

Figure 4 shows an example of code modifications. It explains how matrix multiplication $C = AB$ can be adapted to cooperative multitasking. In this example, the original code [3] in Fig. 4(a) generates (WC/BLOCK_SIZE) $\times$ (HC/BLOCK_SIZE) TBs, where BLOCK_SIZE is the size of TBs, and WC and HC represent the width and the height of matrix C. On the other hand, the modified code in Fig. 4(b) reduces this number to (WC/BLOCK_SIZE) $\times$ GRID_YSIZE TBs, as shown in line 3, where 1 $\leq$ GRID_YSIZE $\leq$ HC/BLOCK_SIZE. Parameter GRID_YSIZE here represents the number of TBs in y direction. This parameter can be given to the CPU code to change the granularity of subtasks at run-time.

Let K (> 0) and k $(\leq K)$ be the execution time of the original task and that of a divided subtask, respectively. In general, the kernel workload is proportional to the number n of TBs. Therefore, our method assumes that the execution time K can be represented by

$$K = B\lceil n/M \rceil, \tag{2}$$

where B represents the time needed for processing a single TB on an MP. In optimized kernels, we can assume that $n \gg M$ and $n \equiv 0 \pmod{M}$. Suppose that the original kernel takes long time such that $K > W$. In this case, we divide the original task into $\lceil K/W \rceil$ subtasks in order to satisfy $k \leq W$. Notice that this estimation is not precise because MPs can run multiple TBs at a time.

B. Alternative Execution

Although each of subtasks completes its execution within the idle period W, the frame rate of the graphics application can drop if guest subtasks are continuously executed on the GPU. To prevent guest subtasks from occupying the GPU, our method tries to ensure that at least a rendering task is processed between successive guest subtasks.

Such alternative execution requires synchronization between host and guest applications, because they are independently executed in grid environments. However, it is not realistic to develop a synchronization mechanism for arbitrary combinations of graphics applications and scientific applications. Therefore, our method invokes the guest kernel at almost the same intervals as the graphics application. This can be simply realized by calling a sleep function between guest kernel calls, as shown at line 8 in Fig. 4(b). The sleep function then sleeps time $1/F$, so that at least a frame will be produced before the next call of the guest kernel, as shown in Fig. 3(b). Note that we must call cudaThreadSynchronize() before calling the sleep function because CUDA kernels are currently launched in an asynchronous, non-blocking mode [3]. Otherwise, CUDA kernels can be continuously executed between successive frames.

For the sleep function, we currently use Sleep() provided by Windows API [11]. This has an advantage over a naive implementation that enters a busy loop because Sleep() allows the guest process to move to the waiting state. However, we need an accurate sleep mechanism with 1-ms resolution to deal with graphics applications with a higher frame rate F ranging from 30 fps to 60 fps. On the other hand, the resolution of Sleep() depends on that of hardware timer and the time slice of operating system. For example, Windows XP has the default value of 15 ms if it runs on multiple CPUs. To obtain an accurate sleep, our method increases the rate of context switches by altering the time quantum from 15 ms to 1 ms. This alternation can be done using timeBeginPeriod() and will be done if and only if guest applications are allocated to the GPU. Due to the same reason, some PC games might change the time quantum when they are executed as host applications.

Table I
SPECIFICATION OF EXPERIMENTAL MACHINES.

Item	Machine #1	Machine #2
Operating system	Windows XP	Windows 7
CPU	Core 2 Duo	Xeon W3520
GPU	GeForce 8800 GTS (G80)	GeForce GTX 280
CUDA	1.1	2.3
Driver	169.21	191.07
Desktop resolution	1280×1024 pixels	1920×1080 pixels

IV. EXPERIMENTAL RESULTS

We now show experimental results to understand the effects of the proposed method. For experiments, we used two desktop PCs, as shown in Table I. One is equipped with an Intel Core 2 Duo CPU running at 1.86 GHz. This machine has an NVIDIA GeForce 8800 GTS (G80) card with $M = 12$. We have installed Windows XP, CUDA 1.1, and graphics driver 169.21. The other one has an Intel Xeon W3520 CPU running at 2.66 GHz. This machine has an NVIDIA GeForce GTX 280 card with $M = 30$. We have installed Windows 7, CUDA 2.3, and graphics driver 191.07.

With respect to guest applications, we used two applications. One is matrix multiplication [3] and the other is biological sequence alignment [12]. The former solves the problem with the matrix size of 2048×2048. During execution, the kernel generates $n = 16,384$ TBs, each consisting of 16×16 threads. The latter implements the Smith-Waterman algorithm [13] to perform sequence alignment between a database of 250,143 entries and a query sequence of length 512. The kernel generates $n = 250,143$ TBs with thread block size 128. Both guest applications are manually modified for the proposed method.

On the other hand, a phong shader [14] is employed as a host application. This shader is implemented using the OpenGL library [10] with the periodical rendering model. Since the shader runs at $F = 60$ fps, the sleeping time $1/F$ is set as 17 ms in experiments.

A. Overhead of Task Division

We first confirm that our task division strategy controls the execution time k of a subtask. Table II shows the measured time k of matrix multiplication with varying the number d of task divisions. We can see that the time k varies according to the number d of task divisions, i.e., the number n of TBs. For example, the original kernel takes $K = 299.9$ ms to complete matrix multiplication but the execution time is reduced to 2.4 ms if the task is divided into 128 subtasks ($d = 128$). Furthermore, the time k is proportional to the number n, as we modeled in Eq. (2). Therefore, we can easily control the time k if the original time $K = 299.9$ is given to the grid system.

With respect to the overhead of task division, the overhead reveals when $d = 128$. In this case, the effective performance reduces from 57.3 GFLOPS to 55.9 GFLOPS, which is

Table II
EXECUTION TIME OF MATRIX MULTIPLICATION WITH DIFFERENT NUMBERS OF TASK DIVISIONS. PERFORMANCE IS MEASURED USING MACHINE #1.

d: # of task divisions	k: kernel time (ms)		Performance (GFLOPS)	
	Measured	Estimated	Kernel	Kernel w/ wait
1 (original)	299.9	299.9	57.3	57.3
4	75.0	75.0	57.3	44.5
8	37.5	37.5	57.3	37.9
16	18.8	18.7	57.1	28.9
32	9.4	9.4	57.1	31.6
64	4.8	4.7	57.1	15.8
128	2.4	2.3	55.9	7.9

equivalent to 2.4% performance drop. The effective performance here is given by $2N^3/dk$, where $2N^3$ represents the number of floating-point operations needed for matrix multiplication of size N. Thus, the overhead is small for matrix multiplication. Note here that the entire guest performance can be further reduced from this value due to the sleeping time $1/F$. Table II also shows another effective performance, $2N^3/d(k+w)$, which explains the impact of this waiting overhead. For example, the entire performance results in 7.9 GFLOPS when $d = 128$ though the kernel performance itself reaches 55.9 GFLOPS.

In summary, our task division strategy is useful to control the execution time k of a subtask. It has a lower overhead but the waiting overhead dw will reduce the entire performance of guest applications.

B. Performance of Multitasking

Figure 5 shows the frame rate of the phong shader and the relative throughput of matrix multiplication, explaining how the host and guest application performance vary according to the execution time k per kernel invocation, i.e., the number d of task divisions. The relative throughput of 1.0 here corresponds to the maximum performance measured on dedicated machine #1. The frame rate is shown in average. Obviously, there is a tradeoff relation between the host performance and the guest throughput. For example, the host performance will be maximized when the task is decomposed into many subtasks. However, we can see that a throughput of 0.35 can be achieved without degrading the rendering performance. Furthermore, the throughput reaches 0.5 if resource owners accept 10% performance loss (54 fps, in this case). Thus, we can obtain 50% of dedicated performance in a non-dedicated environment.

We also investigated the effect of the alternative execution strategy, as shown in Fig. 5. We obtain higher, stable frame rates by calling the sleep function. For example, the frame rate reaches 54 fps when $k = 18.8$ but it reduces to 26 fps if we do not call the sleep function. Accordingly, the kernel throughput increases from 0.5 but the frame rate becomes unstable. For example, the rate ranges from 51 fps to 57 fps if we call the sleep function. In contrast, it ranges from 23 fps to 39 fps if we do not call the function. In this sense, the

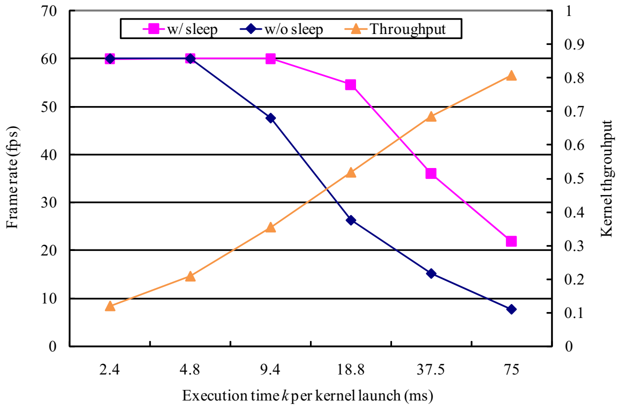


Figure 5. Frame rate of phong shader and throughput of matrix multiplication with different task granularities. Results are obtained on machine #1 and are shown in average.

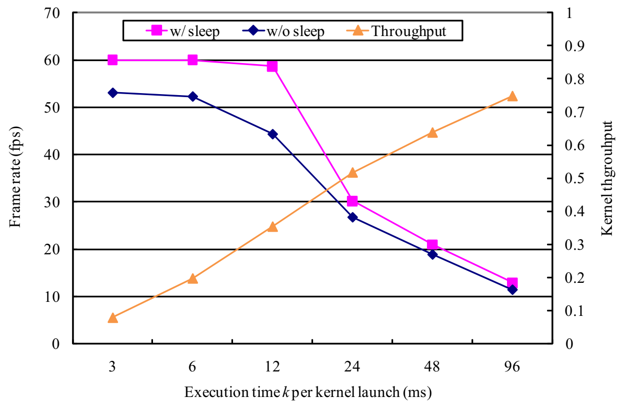
sleep function plays an important role in achieving smooth rendering for host applications.

Finally, Fig. 6 shows the frame rate of the phong shader, the relative throughput of matrix multiplication and that of biological sequence alignment. These results are measured on machine #2. There are two differences in Fig. 6(a), as compared with results on machine #1 (Fig. 5). Firstly, we observe lower frame rates on machine #2 though it has higher performance than machine #1. For example, the frame rate at $k = 24$ is approximately 30 fps in Fig. 6(a), which is 44% lower than that at $k = 18.8$ in Fig. 5. Secondly, the effects of the sleep function is reduced when $k \geq 24$. We think that these differences are due to the difference between Windows XP and Windows 7. The latter has a hardware-based graphical user interface (GUI) called Windows Aero. This GUI is implemented using the DirectX graphics library [9]. Therefore, there are two host applications on machine #2: the phong shader and the Windows Aero.

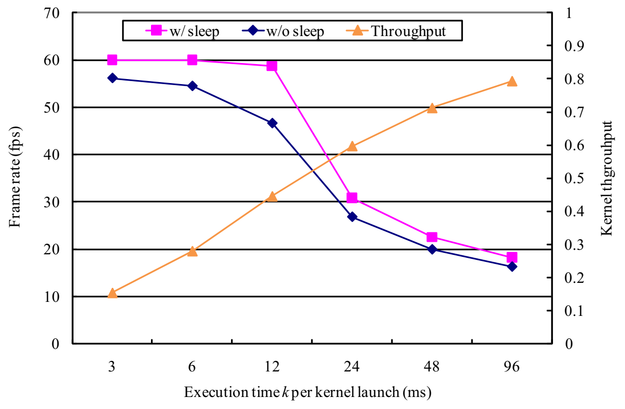
We also find similar results for biological sequence alignment, as shown in Fig. 6(b). As compared with matrix multiplication results, we obtain slightly higher frame rates when running this biological application. It differs from matrix multiplication in that the performance is limited by the instruction issue rate rather than the memory bandwidth. Therefore, the frame rate can be increased if host and guest application have different performance bottlenecks.

V. CONCLUSION

We have presented a cooperative multitasking method capable of simultaneously running a graphics kernel and a CUDA kernel on a single GPU. In order to control the frame rate of the graphics application, our method divides CUDA tasks into smaller subtasks such that each subtask can be completed within an idle period. Furthermore, the method calls a sleep function for every kernel invocation to avoid



(a)



(b)

Figure 6. Frame rate of phong shader and throughput of guest applications with different task granularities. (a) matrix multiplication and (b) biological sequence alignment. Results are obtained on machine #2 and are shown in average.

resource starvation due to continuous execution of CUDA kernels.

In experiments, we have shown that the method successfully controls the frame rate of host applications and the throughput of guest applications. Our multitasking execution achieves 35% of guest throughput as compared with exclusive execution. This throughput is achieved without dropping the original frame rate of 60 fps. The throughput increases to 50% if host users accept 10% frame loss.

One future work is to extend the method for non-periodical applications, which dynamically vary the frame rate.

Acknowledgment

This work was partly supported by JSPS Grant-in-Aid for Scientific Research (A)(20240002), Young Researchers (B)(19700061), and the Global COE Program "in silico medicine" at Osaka University.

References

[1] 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.

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

[4] The Folding@Home Project, "Folding@home distributed computing," 2008, http://folding.stanford.edu/.

[5] Y. Kotani, F. Ino, and K. Hagihara, "A resource selection system for cycle stealing in GPU grids," *J. Grid Computing*, vol. 6, no. 4, pp. 399–416, Dec. 2008.

[6] F. Ino, Y. Kotani, Y. Munekawa, and K. Hagihara, "Harnessing the power of idle GPUs for acceleration of biological sequence alignment," *Parallel Processing Letters*, vol. 19, no. 4, pp. 513–533, Dec. 2009.

[7] Y. Kotani, F. Ino, and K. Hagihara, "A resource selection method for cycle stealing in the GPU grid," in *Proc. 4th Int'l Symp. Parallel and Distributed Processing and Applications Workshops (ISPA'06 Workshops)*, Dec. 2006, pp. 939–950.

[8] S. Yamagiwa and K. Wada, "Performance study of interference on sharing GPU and CPU resources with multiple applications," in *Proc. 11th Workshop Advances in Parallel and Distributed Computational Models (APDCM'09)*, May 2009, 8 pages (CD-ROM).

[9] Microsoft Corporation, "DirectX," 2007, http://www.microsoft.com/directx/.

[10] D. Shreiner, M. Woo, J. Neider, and T. Davis, *OpenGL Programming Guide*, 5th ed. Reading, MA: Addison-Wesley, Aug. 2005.

[11] Microsoft Corporation, "Windows API," 2009. [Online]. Available: http://msdn.microsoft.com/en-us/library/cc433218(VS.85).aspx

[12] Y. Munekawa, F. Ino, and K. Hagihara, "Design and implementation of the Smith-Waterman algorithm on the CUDA-compatible GPU," in *Proc. 8th IEEE Int'l Conf. Bioinformatics and Bioengineering (BIBE'08)*, Oct. 2008, 6 pages (CD-ROM).

[13] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *J. Molecular Biology*, vol. 147, pp. 195–197, 1981.

[14] T. Teranishi, "Phong shading sample program," 2003, http://www.asahi-net.or.jp/~yw3t-trns/opengl/samples/fshphong/.