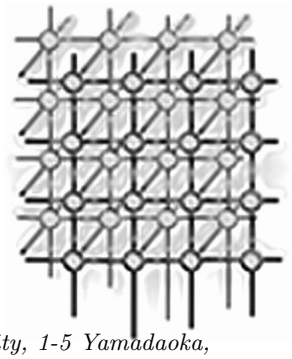
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2011; **23**:1–15 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*

Cooperative Multitasking for GPU-Accelerated Grid Systems



Fumihiko Ino[1,*,†], Akihiro Ogita[2], Kentaro Oita[1] and Kenichi Hagihara[1]

[1] *Graduate School of Information Science and Technology, Osaka University, 1-5 Yamadaoka, Suita, Osaka 565-0871, Japan*
[2] *NEC SOFT, Ltd., 1-18-7 Shinkiba, Koto-ku, Tokyo 136-0082, Japan*

SUMMARY

This paper presents a cooperative multitasking method for concurrent execution of scientific and graphics applications on the graphics processing unit (GPU). Our method is designed to accelerate compute unified device architecture (CUDA) based applications using idle GPU cycles in the office. To prevent significant slow-down of graphics applications, the method divides scientific tasks into smaller pieces, which are then sequentially executed at the appropriate intervals. The method also has flexibility in finding the best trade-off point between scientific applications and graphics applications. 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, biological sequence alignment can be processed at approximately 40% of the dedicated throughput while achieving interactive rendering at 58 frames per second. We also show that matrix multiplication can be efficiently processed at 60% of the dedicated throughput during word processing and web browsing. Copyright © 2011 John Wiley & Sons, Ltd.

KEY WORDS: multitasking; GPU; CUDA; grid computing

1. INTRODUCTION

The graphics processing unit (GPU) [1] is a microprocessor originally designed to accelerate graphics applications with offloading rendering tasks from the CPU. To satisfy increasing demand for rendering performance, the GPU has high memory bandwidth and high floating-

*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 1 September 2010
Revised xx xx 20xx

point performance with hundreds of single-instruction, multiple-data (SIMD) capable cores. In addition to these rich resources, the NVIDIA GPU has a flexible development framework, called compute unified device architecture (CUDA) [2], for running general-purpose applications. The CUDA framework allows us to write C-like code that typically achieves a 10-fold speedup over CPU-based implementations [3]. In this framework, the compute-intensive code is implemented as a *kernel*, namely a function executed on the GPU in a SIMD fashion.

The GPU also has emerged 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 60% of the entire performance is provided by idle GPUs, which account for only 6% 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, we denote *guests* as grid users who ask the system to run their scientific jobs on the donated resources.

One problem in GPU-accelerated grid systems [5, 6] is the lack of efficient multitasking mechanisms. Therefore, 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]. For example, hosts will suffer from frequent system freezes when their GPUs compute large matrix multiplication for guests. This is due to the current GPU architecture, which (1) sequentially executes kernels if they are invoked from different application contexts [9] and (2) blocks other kernels until the running kernel completes its execution. Therefore, the frame rate will significantly drop if the guest kernel occupies the entire resource for long time. In this sense, multitasking mechanisms must have flexibility in controlling guest application performance and host application performance.

Note that the slow-down mentioned above occurs not only on previous architectures but also on the latest Fermi architecture [9]. Though Fermi supports concurrent kernel execution, this capability is designed for small kernels invoked from the same application context. That is, it simultaneously runs multiple kernels if each of them cannot utilize all cores in the GPU [9]. In contrast, the problem we tackle in this paper is that of concurrent application execution, where large kernels are invoked from different application contexts. In this case, each of the kernels usually uses all cores in the GPU.

To solve the problem mentioned above, we develop a cooperative multitasking method capable of concurrent execution of a graphics application and a CUDA-based scientific application. Using this method, hosts can 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 appropriate period. Thus, hosts are allowed to find the best trade-off point between host application performance and guest application performance. The method assumes that guest applications are implemented using the CUDA platform while host applications are implemented using a periodical rendering model with graphics libraries such as DirectX [10] and OpenGL [11].

The rest of this paper is organized as follows. Section 2 introduces related work. Section 3 presents preliminaries including an overview of the CUDA and the periodical rendering model. Section 4 then describes the details of our multitasking method and Section 5 shows experimental results. Finally, Section 6 summarizes the paper with future work.

2. Related Work

As far as we know, current GPU-accelerated grid systems [4, 5, 12] use a screensaver approach to prevent slow-down of host applications. In these systems, screensavers detect fully idle resources for acceleration of guest applications. For example, Kotani *et al.* [5] present a screensaver-based system that detects idle GPUs according to the usage of video memory. The Folding@home project [4] uses another approach that suspends guest applications whenever a host application requests exclusive DirectX mode. A similar volunteer computing project, called GPUGRID.net [12], also constructs a screensaver-based system using the BOINC grid middleware [13]. Thus, both guest and host applications are exclusively executed in current systems. Therefore, most of the GPUs are dedicated to guest applications only when they are fully idle.

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. Thus, screensavers are not required to ensure exclusive execution of guest and host applications.

With respect to Windows systems, the GPU, the graphics driver and the operating system have to support Windows Display Driver Model (WDDM) 2.1 [14] in order to realize preemptive multitasking on the GPU. Although Fermi has reduced the overhead of context switching to below 25 μs [9], it supports only WDDM 1.1, where multitasking is done in a cooperative manner [14]. Thus, the GPU currently does not support preemptive multitasking because a large number of states must be efficiently saved for this highly threaded architecture. Furthermore, there is no way to suspend or cancel kernels after they are invoked from the CPU. Accordingly, the running kernel occupies the resources and blocks other kernels until it finishes execution. Since this blocking time increases with the execution time of a kernel, the slow-down of host applications usually appears more clearly as we increase the problem size of guest applications. For example, we cannot notice disturbances if small matrices are multiplied in background on a Fermi card. However, the desktop screen freezes when we iteratively multiply large matrices of more than 4096×4096 elements.

3. PRELIMINARIES

This section presents preliminaries needed to understand our method.

3.1. Compute Unified Device Architecture (CUDA)

The CUDA [2] 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 CPU code and a series of kernels. The CPU code invokes kernels, which typically generate millions of threads on the GPU. These threads then accelerate heavy computation by SIMD instructions on streaming multiprocessors (SMs) in the GPU. Currently,

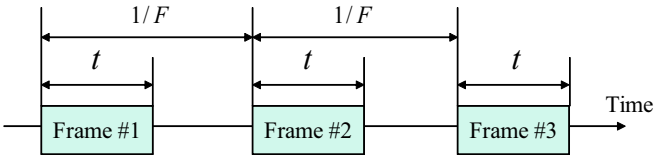


Figure 1. Periodical rendering model. Rendering tasks are executed at the regular intervals to produce a series of frames. Parameters F and t represent the frame rate of the graphics application and the rendering time for a single frame, respectively.

the number M of SMs ranges from 15 to 30 and each SM has 8 or 32 cores, depending on the architecture.

In CUDA programs, threads are classified into thousands of groups, each called as a thread block (TB). 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 prohibited in the kernel. Therefore, we have to separate the kernel into multiple pieces if such dependencies cannot be eliminated from the kernel code. Separated kernels are then sequentially executed to obtain the same results as the original kernel. Note that an implicit global synchronization is performed between successive kernel invocations.

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

3.2. 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 whatever graphics card we use. This unchanging performance is required for PC games and movie players, which have to avoid extremely fast forwarding of scenes on high-end cards. 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 an instance of kernel invocation. As we mentioned in Section 1, our method assumes that host applications are implemented using the periodical model.

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

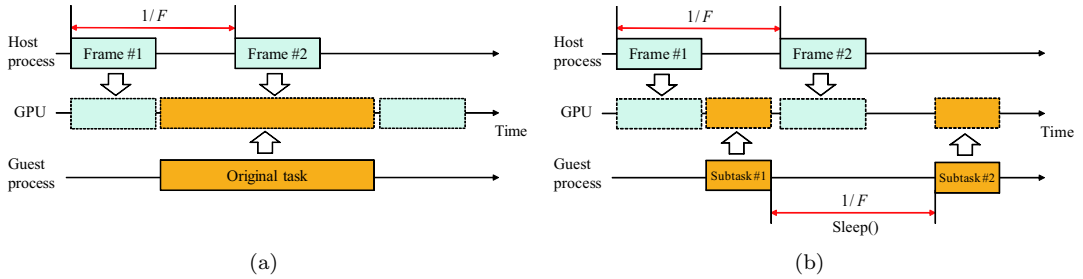


Figure 2. Timeline view of multitasking execution on the GPU. (a) A naive method drops the frame rate because the guest kernel blocks the host kernel until it completes its execution. (b) Our method prevents this performance degradation by dividing each of scientific tasks into smaller subtasks, which are then sequentially processed at regular intervals.

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 will be decreased if $W < 0$. One problem here is that the time t cannot be directly measured in grid environments because it is not easy for us to obtain and modify the source code of arbitrary host applications. Taking this constraint into account is important to develop grid middleware.

4. PROPOSED MULTITASKING METHOD

Figure 2 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. 2(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 2(b) illustrates how our method realize efficient cooperative 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 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 execution in grid environments.

4.1. Task Division

We now explain how task division can be done for CUDA-based applications. As we mentioned in Section 3.1, 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 sequentially 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 3.1 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 3 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 [2] in Fig. 3(a) generates (WC/BLOCK_SIZE) × (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. 3(b) reduces this number to (WC/BLOCK_SIZE) × 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 a SM. 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 SMs can run multiple TBs at a time.

4.2. 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.

```
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 3. Example of guest code modifications. (a) The original code [2] 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 granularity of 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. 3(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. 2(b). Note that we must call cudaThreadSynchronize() before calling the sleep

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

function because CUDA kernels are currently launched in an asynchronous, non-blocking mode [2]. Otherwise, CUDA kernels can be continuously executed between successive frames.

For the sleep function, we currently use Sleep() provided by Windows API [15]. This function 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 frames per second (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.

5. EXPERIMENTAL RESULTS

We now show experimental results to understand the effects of the proposed method. In order to make it clear the impact of exploiting idle GPU cycles during word processing and web browsing, we extend our preliminary results [16] to the case of non-periodical applications, which dynamically vary the frame rate.

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 architecture) 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: matrix multiplication [2] and biological sequence alignment [17]. 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 [18] to perform sequence

Table II. Execution time of matrix multiplication on machine #1 with different numbers of task divisions. Entire performance takes waiting time into account. The original task generates 128×128 TBs.

d: # of task divisions	Estimated time (ms)	k: Measured time (ms)	Kernel performance (GFLOPS)	Entire performance (GFLOPS)
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	18.7	18.8	57.1	28.9
32	9.4	9.4	57.1	31.6
64	4.7	4.8	57.1	15.8
128	2.3	2.4	55.9	7.9

alignment between a database of 250,143 entries and a query sequence of length 512. The kernel generates $n = 250,143$ TBs with a TB size of 128. Both guest applications are manually modified for the proposed method.

On the other hand, we used two host applications: a phong shader [19] as a periodical application and a game program, called Lost Planet Extreme Condition Benchmark [20], as a non-periodical application. The former is implemented using the OpenGL library [11]. The sleeping time $1/F$ is set as 17 ms in experiments because the shader originally runs at $F = 60$ fps. The latter is implemented using the DirectX library [10]. Since this program is based on the non-periodical rendering model, the frame rate ranges from 40 fps to 92 fps depending on the complexity of rendered scenes. The frame rate is measured using a tool [21].

5.1. Task Division Overhead

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 k 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. Actually, the estimated time K/d is close to the measured time k.

With respect to the overhead of task division, the overhead reveals when $d = 128$. In this case, the effective performance slightly reduces from 57.3 GFLOPS to 55.9 GFLOPS, which is 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 due to the sleeping time $1/F$, as shown in Table II. In

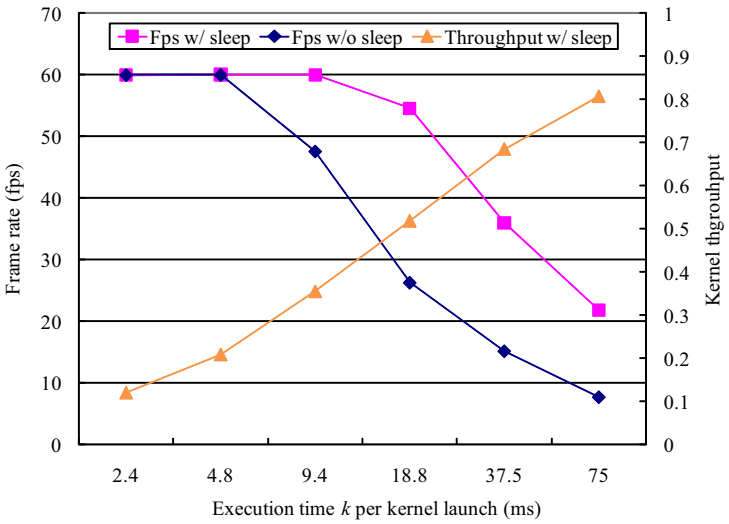


Figure 4. Frame rate of phong shader and throughput of matrix multiplication on machine #1 with different task granularities. Results are shown in average.

this table, the entire performance is given by $2N^3/d(k+w)$, 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 achieves low overhead but the waiting overhead dw will reduce the entire performance of guest applications.

5.2. Multitasking Performance

Figure 4 shows the frame rate of the phong shader and the relative throughput of matrix multiplication, explaining how the host and guest application performance varies 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 trade-off relation between the host performance and the guest throughput. For example, the host performance will be maximized when the task is decomposed into 128 subtasks ($k = 2.4$). However, it should be noted that a throughput of 0.35 is achieved without degrading the rendering performance. Furthermore, the throughput reaches 0.5 if hosts accept 10% performance loss (54 fps, in this case). Thus, we can obtain 50% of dedicated performance in a non-dedicated environment.

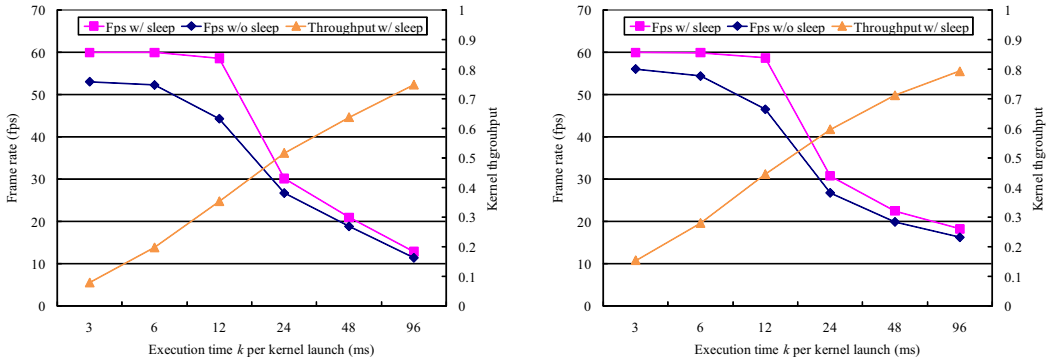


Figure 5. Frame rate of phong shader and throughput of guest applications on machine #2 with different task granularities. Results are shown in average for (a) matrix multiplication and for (b) biological sequence alignment.

We also investigated the effect of the alternative execution strategy, as shown in Fig. 4. 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 sleep function plays an important role in achieving smooth rendering for host applications.

Figure 5 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. 5(a), as compared with results on machine #1 (Fig. 4). 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. 5(a), which is 44% lower than that at $k = 18.8$ in Fig. 4. Secondly, the effects of the sleep function is reduced when $k \geq 24$. 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 [10]. Therefore, there are two host applications running on machine #2: the phong shader and the Windows Aero. In contrast, machine #1 has a single host application that runs on the GPU.

We also find similar results for biological sequence alignment, as shown in Fig. 5(b). As compared with matrix multiplication, we obtain slightly higher frame rates when running this 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 applications have different performance bottlenecks.

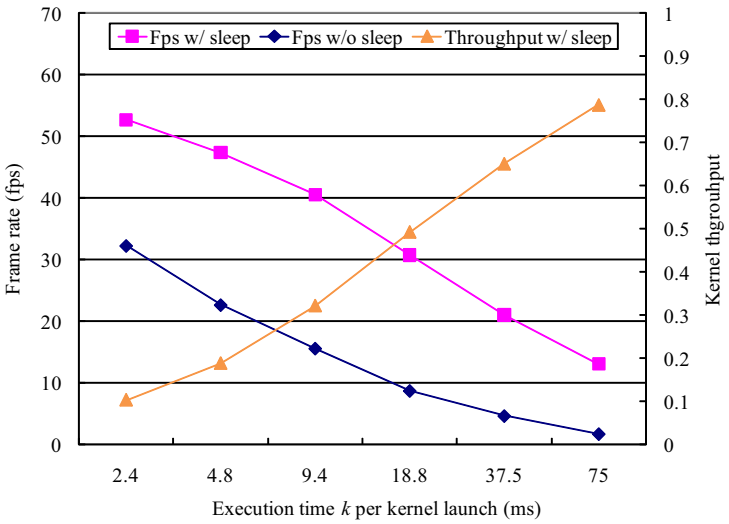


Figure 6. Frame rate of a game program [20] and throughput of matrix multiplication on machine #1 with different task granularities. Results are shown in average.

5.3. Evaluation with Non-Periodical Applications

Though our method is designed for periodical applications, we applied the method to non-periodical applications in order to investigate the effectiveness of the method. Using machine #1 with the sleeping time $1/F = 17$, we executed matrix multiplication while executing the game program in a demo mode. Figure 6 shows the relationship between the frame rate of the game and the throughput of matrix multiplication. Both measured values are presented in average. Similar to previous results obtained with periodical applications, there is a trade-off relation between host application performance and guest application performance. We also can see that the sleep function contributes to increase the frame rate of the game. The difference to the previous results is that frame rate proportionally decreases as the execution time k increases. This is due to the non-periodical model, which invokes kernels as soon as possible. Thus, host applications must be implemented using the periodical rendering model to make it possible to increase the guest performance without dropping the host performance.

Figure 7 shows how the frame rate varies for a series of successive 30 scenes with different execution time k per kernel launch. In this figure, the horizontal axis corresponds to the elapsed time. The frame rate ranges from 40 fps to 92 fps if we execute the game without guest applications. Similar to periodical applications, the frame rate decreases from the original rate when we execute both of host and guest applications. For example, we observe at most 18 fps when using the sleep function with $k = 75$. However, we achieve at least 26 fps when

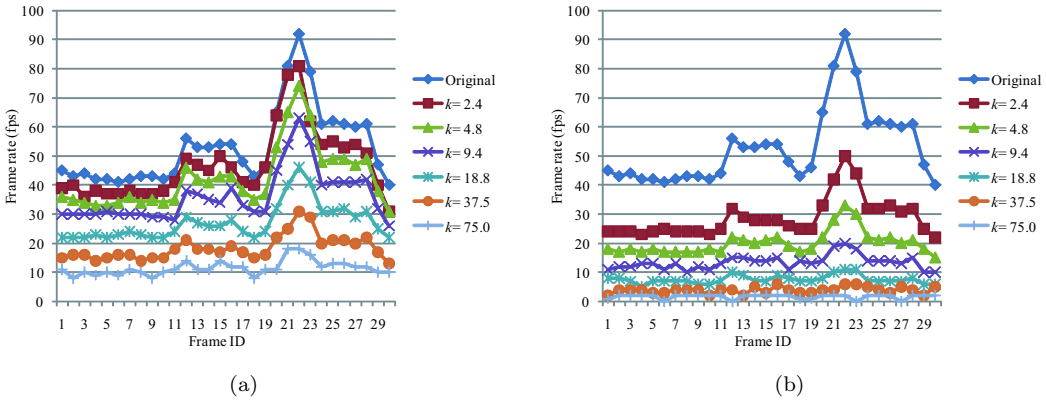


Figure 7. Sequence of frame rate measured using a game program [20] (a) with and (b) without a sleep function. Parameter k represents the execution time per kernel launch in ms. Frame IDs in (a) and (b) are identical and correspond to elapsed time.

$k = 9.4$. Thus, our task division strategy contributes to prevent significant slow-down not only for periodical applications but also for non-periodical applications.

The results in Fig. 7 also indicate that it is better to dynamically change the number d of task divisions according to the complexity of scenes. For example, the frame rate sharply increases around frame #20. In this case, we are allowed to process more guest subtasks without significant disturbances to hosts. We think that a dynamic mechanism will be useful to increase the execution time k from 9.4 ms to 18.8 ms without dropping the frame rate below 30 fps.

Finally, we investigate the effectiveness of our method with interactive applications. Figure 8 shows the results obtained using three host applications: (1) Microsoft Word running as a word processor, (2) a web browser playing a movie at YouTube, and (3) the phong shader mentioned before. With respect to the word processor, we recorded keyboard and mouse events while writing a document at approximately 12 fps. We then replayed the events while executing guest applications. This ensures the same behavior to perform fair comparison between multitasking execution and exclusive (single-task) execution. Note that the execution time k per kernel launch determines the maximum frame rate in Fig. 8. For example, the frame rates in Fig. 8(b) are bounded by approximately 30 fps because we use $k = 24$ for experiments (see also Fig. 6).

In Fig. 8, we can see that the kernel throughput reaches 0.6 and the frame rate is slightly reduced when using the word processor as a host application. Thus, our method minimizes disturbances to hosts, so that they probably cannot be aware of the execution of guest applications. Similar results are obtained for the case of web browsing. In summary, matrix multiplication and biological sequence alignment can be efficiently processed at 60% of the dedicated throughput during word processing and web browsing.

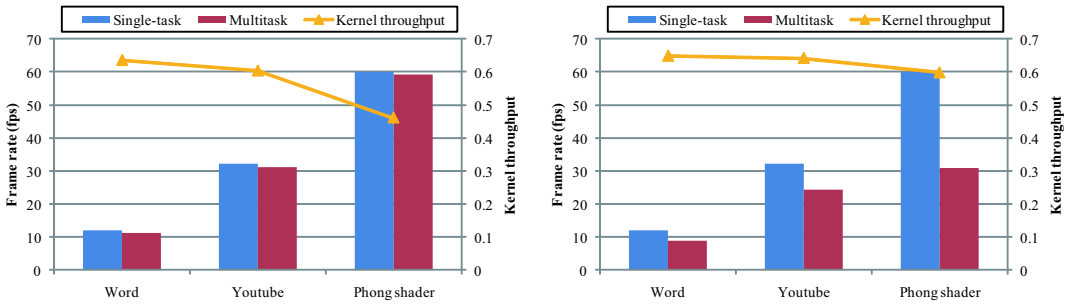


Figure 8. Frame rate of host applications and throughput of guest applications with different host applications. (a) Matrix multiplication running on machine #1 with $k = 18.8$ and (b) biological sequence alignment running on machine #2 with $k = 24$.

6. CONCLUSION

We have presented a cooperative multitasking method capable of concurrently running a graphics kernel and a CUDA kernel on a single GPU. Our method is designed for GPU-accelerated grid systems, where guests use idle GPU cycles donated by hosts. In order to control the frame rate of host applications and the throughput of guest applications, the method divides CUDA-based guest tasks into smaller subtasks such that each subtask can be completed within an idle period. Furthermore, it calls a sleep function for every kernel invocation to avoid 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 approximately 35% of guest throughput as compared with exclusive execution. This throughput is achieved without dropping the original frame rate of 60 fps. We also have shown that matrix multiplication and biological sequence alignment can be efficiently processed at 60% of the dedicated throughput while allowing hosts to continue their office work such as word processing and web browsing.

One future work is to extend the method for non-periodical applications, which dynamically vary the frame rate. For example, we think that the runtime selection of the task granularity is useful to maximize the guest application performance.

ACKNOWLEDGEMENTS

This work was partly supported by the Research Grant 09-05 from the Okawa Foundation for Information and Telecommunications, JSPS Grant-in-Aid for Scientific Research (A)(20240002), and the Global COE Program “in silico medicine” at Osaka University.

REFERENCES

1. Lindholm E, Nickolls J, Oberman S, Montrym J. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro* 2008; **28**(2):39–55.
2. NVIDIA Corporation. CUDA Programming Guide Version 2.3. Available at: http://developer.nvidia.com/cuda/ [31 August 2010].
3. Owens JD, Houston M, Luebke D, Green S, Stone JE, Phillips JC. GPU computing. *Proceedings of the IEEE* 2008; **96**(5):879–899.
4. The Folding@Home Project. Folding@home distributed computing. Available at: http://folding.stanford.edu/ [31 August 2010].
5. Kotani Y, Ino F, Hagihara K. A resource selection system for cycle stealing in GPU grids. *Journal of Grid Computing* 2008; **6**(4):399–416.
6. Ino F, Kotani Y, Munekawa Y, Hagihara K. Harnessing the power of idle GPUs for acceleration of biological sequence alignment. *Parallel Processing Letters* 2009; **19**(4):513–533.
7. Kotani Y, Ino F, Hagihara K. A resource selection method for cycle stealing in the GPU grid. *Proceedings of the 4th International Symposium on Parallel and Distributed Processing and Applications Workshops (ISPA'06 Workshops)*, Sorrento, Italy, December 2006; 939–950.
8. Yamagiwa S, Wada K. Performance study of interference on sharing GPU and CPU resources with multiple applications. *Proceedings of the 11th Workshop Advances in Parallel and Distributed Computational Models (APDCM'09)*, Rome, Italy, May 2009.
9. NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. Available at: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf [31 August 2010].
10. Microsoft Corporation. DirectX. Available at: http://www.microsoft.com/directx/ [31 August 2010].
11. Shreiner D, Woo M, Neider J, Davis T, *OpenGL Programming Guide* (5th edn). Addison-Wesley: Reading, MA, U.S.A., 2005.
12. Giupponi G, Harvey MJ, Fabritiis GD. The impact of accelerator processors for high-throughput molecular modeling and simulation. *Drug Discovery Today* 2008; **13**(23/24):1052–1058.
13. Anderson DP. BOINC: A system for public-resource computing and storage. *Proceedings of 5th IEEE/ACM International Workshop on Grid Computing (GRID'04)*, Pittsburgh, PA, U.S.A., November 2004; 4–10.
14. Pronovost S, Moreton H, Kelley T. Windows display driver model (WDDM) v2 and beyond. *Windows Hardware Engineering Conference on (WinHEC'06)*, May 2006. Available at: http://www.microsoft.com/whdc/winhec/Pres06.mspx [31 August 2010].
15. Microsoft Corporation. Windows API. Available at: http://msdn.microsoft.com/en-us/library/cc433218(VS.85).aspx [31 August 2010].
16. Ino F, Ogita A, Oita K, Hagihara K. Cooperative multitasking for GPU-accelerated grid systems. *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid'10)*, Melbourne, Australia, May 2010; 774–779.
17. Munekawa Y, Ino F, Hagihara K. Accelerating Smith-Waterman algorithm for biological database search on CUDA-compatible GPUs. *IEICE Transactions on Information and Systems* 2010; **E93-D**(6):1479–1488.
18. Smith TF, Waterman MS. Identification of common molecular subsequences. *Journal of Molecular Biology* 1981; **147**:195–197.
19. Teranishi T. Phong shading sample program. Available at: http://www.asahi-net.or.jp/%7Eyw3t-trns/opengl/samples/fshphong/ [31 August 2010].
20. CAPCOM Corporation. LostPlanet Extreme Condition. Available at: http://www.capcom.co.jp/pc/lostplanet/ [31 August 2010].
21. Beepa Pty Ltd. Fraps: real-time video capture & benchmarking. Available at: http://www.fraps.com/ [31 August 2010].