International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 4, Number 2, pages 236–250, July 2014

A Fine Grained Cycle Sharing System with Cooperative Multitasking on GPUs

Fumihiko Ino
Graduate School of Information Science and Technology
Osaka University
1-5 Yamadaoka, Suita, Osaka 565-0871, Japan

Yosuke Oka
Ricoh Company, Ltd
1-3-6 Naka-magome, Ohta-ku, Tokyo 143-8555, Japan

and

Kenichi Hagihara
Graduate School of Information Science and Technology
Osaka University
1-5 Yamadaoka, Suita, Osaka 565-0871, Japan

Received: April 2, 2014
Revised: May 15, 2014
Accepted: May 16, 2014
Communicated by Koji Nakano

Abstract

The emergence of compute unified device architecture (CUDA), which has relieved application developers from having to understand complex graphics pipelines, has made the graphics processing unit (GPU) useful not only for graphics applications but also for general applications. In this paper, we present a cycle sharing system named GPU grid, which exploits idle GPU cycles to accelerate scientific applications. Our cycle sharing system implements a cooperative multitasking technique, which is useful for remotely executing a guest application on a donated host machine without causing a significant slowdown on the host. In addition, our system estimates whether a GPU is busy, partially idle, or fully idle, to accordingly maximize guest application throughput. Experimental results show that our system not only avoids frame rate degradation but also achieves a 91% higher guest application throughput in comparison to a previous system that estimates GPU load by monitoring mouse and keyboard activities.

Keywords: GPGPU, Cooperative multitasking, Cycle sharing, Grid computing, Volunteer computing

1 Introduction

A graphics processing unit (GPU) [14, 18] is a hardware component mainly designed for the acceleration of graphics tasks such as real-time rendering of three-dimensional (3D) scenes. To satisfy the demand for real-time complex scene rendering, the GPU has higher arithmetic performance and memory bandwidth than the CPU. The emergence of compute unified device architecture (CUDA) [19] has allowed application developers

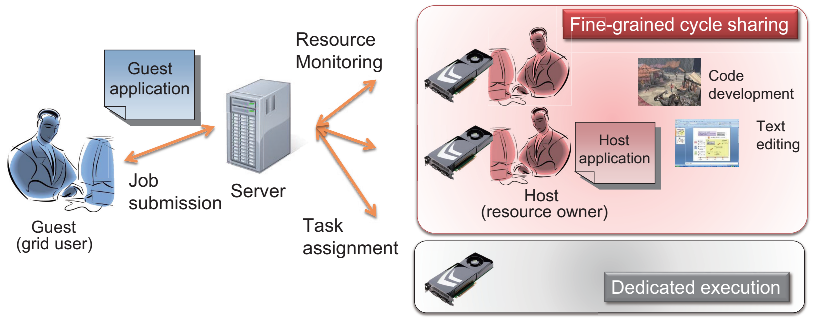


Figure 1: Overview of GPU grid.

to easily utilize the GPU as an accelerator not only for graphics applications but also for general applications. Using the CUDA, an application hotspot can be eliminated by implementing the corresponding code as *a kernel function*, which runs on the GPU in parallel. Therefore, many research studies use the GPU as an accelerator for compute- and memory-intensive applications [5, 16, 21, 22].

One such study, the Folding@home project [2, 28], employed 20,000 idle GPUs to accelerate protein folding simulations on a grid computing system. Although there are several types of grid systems, a grid system in this study it is a volunteer computing system that shares network-connected computational resources to accelerate scientific applications. We denote *a host* as a user who donates a computational resource, and *a guest* as a user who uses the donated resource for acceleration (Fig. 1). A host task corresponds to a local task generated by daily operations on a resource, and a guest task corresponds to a grid task to be accelerated remotely on the donated resource.

Host and guest tasks can be simultaneously executed on a donated resource because the resource is shared between hosts and guests. However, current GPU architectures do not support preemptive multitasking, so that a guest task can intensively occupy the resource until its completion. Therefore, simultaneous execution of multiple GPU programs significantly decreases the frame rate of the host machine. To make the matter worse, the impact of this performance degradation increases with kernel execution time. For example, our preliminary results [12] show that a guest task running on a donated machine causes the machine to hang and reduces its frame rate to less than 1 frame per second (fps). Accordingly, GPU-accelerated grid systems have to not only minimize host perturbation (i.e., frame rate degradation) but also maximize guest application performance.

In this paper, we present a GPU-accelerated grid system capable of exploiting short idle times such as hundreds of milliseconds. Our cycle sharing system extends a previous system [9] to estimate GPU load without relying on an event monitor that detects mouse and keyboard activities. According to this estimation on a donated host machine, our system executes a guest application remotely without causing a significant slowdown on the machine. Our system employs a cooperative multitasking technique [10] as a basic mechanism for finding the best tradeoff point between guest application throughput and host frame rate. We extend our preliminary results [7] with code examples and detailed evaluation results on host perturbation.

The remainder of this paper is structured as follows. Section 2 introduces previous studies on GPU-accelerated grid systems. Section 3 presents a brief summary of our basic multitasking technique [10]. Section 4 describes how our cooperative system estimates GPU load to maximize guest application throughput. Section 5 shows experimental results obtained in our laboratory. Finally, Section 6 concludes this paper with future directions.

2 Related Work

2.1 Pre-CUDA Era

Before the release of CUDA, the only way to implement GPU applications was to use a graphics application programming interface (API) such as DirectX [4] or OpenGL [26]. Because these APIs are designed primarily for graphics applications, it is not easy to implement general applications using them. Despite this low programmability for general applications, some grid systems have tried to accelerate their computation using the GPU. The Folding@home and GPUGRID.net systems [6, 28] are based on Berkeley Open Infrastructure for Network Computing (BOINC) [1], which employs a screensaver to avoid the simultaneous execution of multiple GPU programs on a host machine. These systems detect an idle machine according to screensaver activation. A running guest task can be suspended (1) if the screensaver turns off due to the host's activity or (2) if the host machine executes DirectX-based software with exclusive mode. Here, the exclusive mode is useful to avoid a significant slowdown on the host machine if both guest and host applications are implemented using DirectX.

In contrast to this DirectX-based approach, Kotani *et al.* [12] presented a screensaver-based system that monitors video memory usage in addition to the host's activity. By monitoring this usage, the system can avoid the simultaneous execution of host and guest applications, though the host applications are not executed with exclusive mode. Screensaver-based systems are useful for detecting long idle periods spanning a few minutes. However, short idle periods, such as a few seconds, cannot be detected because of the limitation of timeout length. Their system was applied to a biological application to evaluate the impact of utilizing idle GPUs in a laboratory environment [8].

Caravela [29] is a stream-based distributed computing environment that encapsulates a program for execution in local or remote resources. This environment focuses on the encapsulation and assumes that resources are dedicated to guests. The perturbation issue, which must be solved for non-dedicated systems, is not addressed.

2.2 CUDA Era

To detect short idle time spanning a few seconds, Ino *et al.* [9] presented an event-based system that monitors mouse and keyboard activities, video memory usage, and CPU usage. Similar to screensaver-based systems, they assume that idle resources do not have mouse and keyboard events for one second. Furthermore, they divide guest tasks into small pieces to minimize host perturbation by completing each piece within 100 milliseconds. Because of this task division, their system realizes a minimum frame rate of around 10 fps.

One drawback of this system is that the GPU is not always busy when the mouse or keyboard is operated interactively by the host. Furthermore, mouse and keyboard events are usually recorded at short intervals such as a few seconds. Consequently, resources can frequently alternate between idle and busy states. This alternation can mean that guest tasks are frequently canceled immediately after their assignment, because idle host machines turn out to be busy before task completion. Furthermore, the job management server can suffer from frequent communication, because a state transition on a resource causes an interaction between the resource and the server.

Some research projects have developed GPU virtualization technologies to realize GPU resource sharing. To the best of our knowledge, NVIDIA GRID [20] and Gdev [11] are the only systems that virtualize a physical GPU into multiple logical GPUs and achieve a prioritization, isolation, and fairness scheme. Gdev currently supports Linux systems. Although virtualization technologies are useful for dealing with the host perturbation issue, they require system modification on host machines. We think that the host perturbation issue should be solved at the application layer to minimize mandatory modification at the system level.

rCUDA [24] is a programming framework that enables remote execution of CUDA programs with a small overhead. A runtime system and a CUDA-to-rCUDA transformation framework are provided to intercept CUDA function calls and redirect them to remote GPUs. Because rCUDA focuses on dedicated clusters rather than shared grids, the host perturbation issue is not solved. A similar virtualization technology was implemented as a grid-enabled programming toolkit called GridCuda [13].

vCUDA [25] allows CUDA applications executing within virtual machines to leverage hardware acceleration. Similar to rCUDA, it implements interception and redirection of CUDA function calls so that CUDA applications in virtual machines can access a graphics device of the host operating system. The host perturbation issue is not tackled.

```
1  // Execution configuration setup
2  dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
3  dim3 grid(WC / threads.x, HC / threads.y);
4  // Kernel invocation
5  matrixMul<<<grid, threads>>>(d_C, d_A, d_B, WA, WB);
```

(a)

```
1  // Object creation of multitasking class
2  MultitaskingMethod* mm =
3      MultitaskingMethod::createMultitaskingMethod(argc, argv);
4  // Execution configuration setup
5  dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
6  // Set the original number of thread blocks
7  mm->setTask(HC / threads.y);
8  while (mm->subtaskExists()) {
9      // Set a reduced number of thread blocks
10     dim3 grid(WC / threads.x, mm->getGrid());
11     // Kernel execution with the appropriate offset
12     matrixMul<<<grid, threads>>>(d_C, d_A, d_B, WA, WB, mm->getOffset());
13 }
14 mm->exitTask();
```

(b)

Figure 2: An example of guest CPU code (a) before and (b) after task division. This example is a part of matrix multiplication distributed as the CUDA software development kit (SDK). See [17] for more detailed information.

3 Cooperative Multitasking

Our cooperative multitasking technique [10] prevents frame rate degradation while maximizing the guest application throughput. This basic technique allows hosts to declare their minimum frame rate F acceptable during guest application execution. According to this declaration, the multitasking technique attempts to make the host frame rate higher than the minimum desired frame rate F. The basic process to realize this cooperation is guest task division that finds the best trade-off point between guest application performance and host frame rate. The guest application throughput increases as we increase the task granularity, but the host frame rate decreases at the same time. Hereafter, we call *a subtask* a small divided piece of a guest task.

To realize task division for CUDA programs, we take advantage of the assumption of CUDA: different thread blocks do not have data dependence between them. Owing to this data independence, we can divide a task into subtasks by simply reducing the number of thread blocks per kernel invocation. To realize this, we have to modify both the CPU code and the GPU code of the guest application, as shown in Figs. 2 and 3. The mandatory modification can be summarized as follows:

- Reducing the number of thread blocks given to the guest kernel (line 10 of Fig. 2(b)).
- Constructing an iterative structure. After task division, a kernel invocation processes a part of the original task, so that we have to iteratively invoke the guest kernel to process all subtasks (line 8 of Fig. 2(b)).
- Correcting the references related to thread block IDs. Because each kernel invocation creates fewer thread blocks, thread blocks have different IDs compared to the original execution. Consequently, we have to adjust the indexing scheme to obtain the same computational results after task division. For example, the outputting address, which is usually specified using a thread block ID, must be corrected using an offset (line 12 of Fig. 2(b) and line 4 of Fig. 3(b)).

```
__global__ void matrixMul(float* A, B, C, int wA, wB) {
    // Block index
    int bx = blockIdx.x;
    int by = blockIdx.y;

    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;

    // Omitted: multiply-adds using shared memory

    // Write the block sub-matrix to device memory;
    // each thread writes one element
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = cSub;
}
```

(a)

```
__global__ void matrixMul(float* A, B, C, int wA, wB, offset) {
    // Block index
    int bx = blockIdx.x;
    int by = blockIdx.y + offset;

    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;

    // Omitted: multiply-adds using shared memory

    // Write the block sub-matrix to device memory;
    // each thread writes one element
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = cSub;
}
```

(b)

Figure 3: An example of guest GPU code (a) before and (b) after task division. This example is a part of matrix multiplication distributed as the CUDA SDK. See [17] for more detailed information.

The appropriate number of task divisions, which maximizes both guest and host performance, can be determined according to the kernel execution time. To estimate the appropriate number, our technique assumes that (1) the kernel execution time T per invocation increases linearly with the number of thread blocks and (2) the original execution time T on the donated resource is given to the technique. Let t be the kernel execution time after task division, and let H be the execution time spent in generating a single frame for the host application. Frames then can be rendered at regular intervals $1/F$ (i.e., without delays) if the following equation is satisfied:

$$t \leq 1/F - H. \tag{1}$$

As a rule of thumb, small subtasks can suffer from low execution efficiency. Consequently, it is better to maximize t to obtain a higher guest application throughput. However, the execution time H cannot be measured in a practical situation because it depends on the host application. Therefore, given T and F, the appropriate number $\lceil T/t \rceil$ of task divisions should be experimentally determined using Eq. (1).

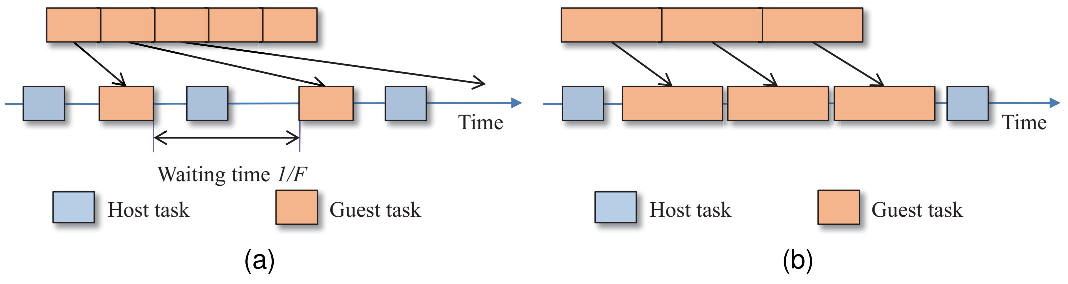


Figure 4: Our cooperative multitasking technique based on two execution modes, for partially and fully idle resources. (a) Periodical execution mode executes guest tasks at regular intervals $1/F$, where F is the minimum desired frame rate. (b) Continuous execution mode intensively executes guest tasks.

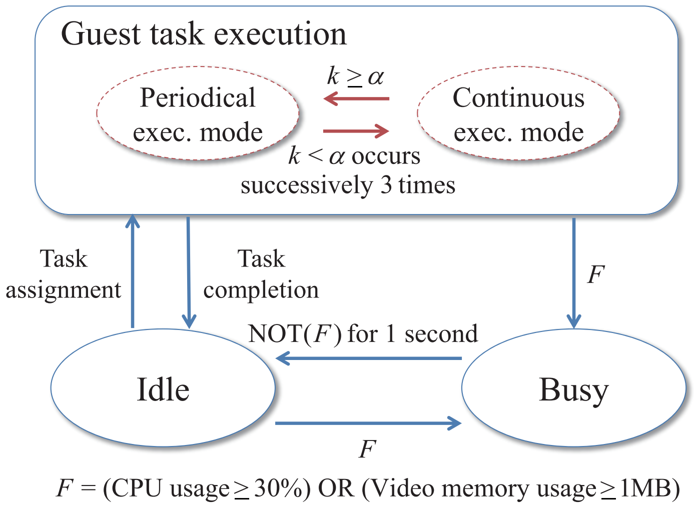


Figure 5: State transition diagram for cooperative multitasking.

4 Our Cycle Sharing System

Our cycle sharing system is capable of exploiting short idle time of hundreds of milliseconds without dropping the frame rate of donated resources. To realize this, our system extends a previous cycle sharing system [9] to avoid mouse and keyboard monitoring. Similar to [10], our system divides a guest task into small subtasks to complete each subtask within tens of milliseconds. Our extension can be summarized in three parts: (1) a relaxed definition of an idle state, (2) GPU load estimation based on an empty kernel, and (3) two execution modes, for partially and fully idle resources (Fig. 4).

The relaxed definition relies only on CPU and video memory usages. Consequently, there is no need to monitor mouse and keyboard activities. A resource is assumed to be busy if both CPU and video memory usages exceed 30% and 1 MB, respectively (Fig. 5). Instead of monitoring mouse and keyboard activities, we integrate two execution modes into our system. For idle resources, our system locally selects the appropriate execution mode for guest tasks. Consequently, most state transitions can be processed locally, avoiding frequent communication between resources and the resource management server.

To determine whether a resource is partially or fully idle, our system estimates GPU load while keeping the frame rate as high as possible. GPU load information can be obtained from the video driver. However, the provided information is not fresh because the time resolution is not sufficiently high to deal with short idle periods such as ten milliseconds. Furthermore, the load detection process itself can cause a significant perturbation on non-preemptive GPUs.

To realize such low-overhead estimation, our system executes an empty kernel before guest task execution and measures its execution time k. An empty kernel is a device function that immediately returns after its

Algorithm 1 Multitasking execution algorithm.

Input: The pre-measured time α, the minimum desired frame rate F, the number N of thread blocks given to the original guest kernel, the number N_1 of thread blocks for periodical execution, the number N_2 of thread blocks for continuous execution.
Output: Multitasking execution.

```
1:  $count := 0$; $k := 0$;
2:  while $N > 0$ do                                        ▷ Guest subtasks remained
3:      $start := clock()$;                                 ▷ clock() returns current time
4:      Empty kernel invocation;
5:      $k := clock() - start$;
6:      if $\alpha > k$ then
7:          $count := count + 1$;
8:      else
9:          $count := 0$;
10:     end if
11:     if $count < 3$ then
12:         Kernel invocation with $\min(N_1, N)$ thread blocks;   ▷ Periodical mode for partially idle state
13:         $N := N - N_1$;
14:         Sleep($1/F$);
15:     else
16:         Kernel invocation with $\min(N_2, N)$ thread blocks;   ▷ Continuous mode for fully idle state
17:         $N := N - N_2$;
18:     end if
19: end while
```

function call. The measured time k is then compared to the pre-measured time α, obtained by dedicated execution on the same resource. We assume that the resource is partially idle if $k \geq \alpha$ and is fully idle if $k < \alpha$ occurs successively three times. Similar to the kernel execution time t, which determines the number of task divisions, the value for α is experimentally determined.

The two execution modes are as follows:

1. A periodical execution mode for partially idle resources. Here, our system uses the periodical mode with tiny subtasks. Each subtask here can be processed within a few tens of milliseconds, and a series of subtasks are processed at regular intervals $1/F$ to keep the frame rate around F fps. In other words, F is the minimum frame rate desired by the host, because at least one frame can be rendered during $1/F$ time. This periodical execution can be realized by calling the Sleep function [15] after every kernel invocation.

2. A continuous execution mode for fully idle resources. Here, our system switches its execution mode to the continuous mode with small subtasks. A series of subtasks is continuously processed on the GPU to achieve higher guest throughput than in periodical execution mode. The continuous execution mode allows guests to execute their tasks on lightly-loaded resources that are interactively operated by hosts.

Algorithm 1 summarizes our multitasking execution algorithm, which runs on host machines as part of a guest application. As inputs, the algorithm requires the pre-measured time α, the minimum desired frame rate F, the number N of thread blocks given to the original guest kernel, the numbers N_1 and N_2 of thread blocks for periodical execution mode, and continuous execution mode, respectively. The numbers N_1 and N_2 are experimentally determined, as mentioned in Section 3. In general, the continuous execution mode must process more thread blocks than the periodical mode (i.e., $N_1 < N_2$) to increase the guest application throughput. Currently, we set N_1 and N_2 such that the kernel completes within approximately 8.5 ms and 17.0 ms, respectively.

False positive and false negative cases can occur when switching to continuous execution mode. The former leads to excessive execution of guest tasks, failing to keep the original frame rate obtained without guest task execution. On the other hand, the latter fails to maximize guest task throughput, but the frame

Table 1: Specification of experimental machines.

Item	Specification
OS	Windows 7 Professional 64 bit
CPU	Intel Core i7-3770K
Main memory capacity	16 GB
GPU	NVIDIA GTX 680
Video memory capacity	2 GB
Core clock	1006 MHz
Memory clock	1502 MHz
CUDA	5.0
Video driver	310.90

Table 2: System uptime in hour.

Host machine	#1	#2	#3	#4
Uptime	135.1	15.9	197.0	81.6

rate can be kept. We think that the latter issue is not critical for our system, because our first priority is minimization of host perturbation. In contrast, we prevent the former case by confirming $k < \alpha$ three times, which avoids immediate transition to continuous execution mode.

5 Experimental Results

We conducted experiments to evaluate our system in terms of guest application throughput. Table 1 shows the specification of our experimental machines. Each machine was equipped with an NVIDIA GTX 680, which could dynamically boost up the clock speed. Four machines were used by graduate students and were monitored for a month. The students mainly used their machines to write CPU/GPU programs, edit documents, and browse websites. Table 2 shows total system uptimes observed on the machines.

To compare our system with a previous system [9] in a fair manner, we simulated the behavior of the previous system by using logs obtained on the experimental machines. The logs contained a time series of CPU and video memory usages, and mouse and keyboard events.

We did not use a resource management server, so the host machines immediately executed a guest task when they turned to idle. Similarly, guest tasks were iteratively executed without communicating with a resource management server. Here, a guest task contained 50 multiplications of 3072×3072 matrices. A cooperative multitasking version of matrix multiplication was developed by modifying the CUDA SDK sample code.

5.1 Guest Application Throughput

Figure 6 shows the measured throughputs of guest task execution. Compared with the previous system, our system achieved a 91% higher throughput on host machine #4. This increase can be explained by the increase in detected idle time length. As shown in Fig. 7, our system detected longer idle times than the previous system, which depends on mouse and keyboard monitoring. This monitoring process prevents short idle periods from being exploited for guest task execution. In contrast, according to the relaxed definition of the idle state, our system eliminates such a monitoring process.

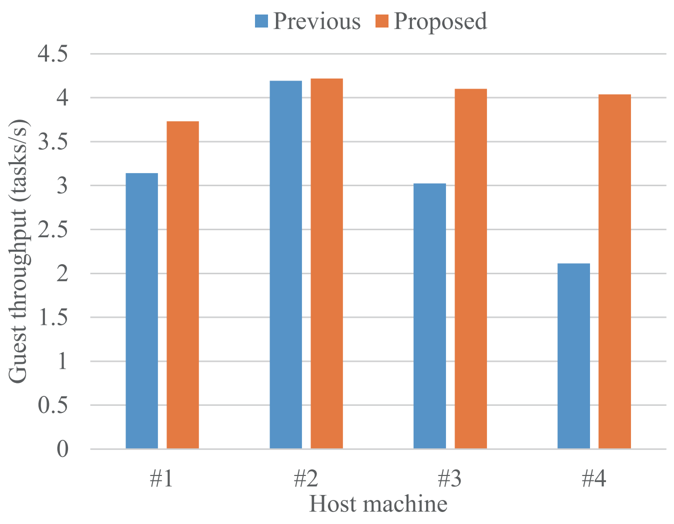


Figure 6: Measured throughput of guest tasks.

Compared to dedicated execution, multitasking execution cannot achieve high efficiency for guest tasks. This might decrease the guest application throughput, but our system covers this drawback by increasing the detected idle time. In fact, Fig. 7 shows that our system detected 1.04–1.67 times longer idle periods than the previous system, which cannot detect short idle times such as hundreds of milliseconds.

In Fig. 7, our detected idle time occupies 99% of system uptime. This indicates that hosts usually use their resources for interactive applications, which do not use GPU resources intensively. Such interactive cases include document editing and web browsing. These cases cause mouse and keyboard events, so that interactively operated resources are considered busy in previous systems. In contrast, our system regards them as partially idle resources, owing to the relaxed definition of the idle state.

Figure 8 shows the number of occurrences of idle period length detected on each host machine. In the previous system, idle period lengths follow a power law distribution. That is, the shorter the idle period is, the more we find an occurrence. We think that this distribution is because of the resource owner's interaction, which frequently operates the mouse and keyboard. However, our system reduces the number of short idle periods spanning over a few seconds but increases the number of long idle periods. This implies that our relaxed definition of an idle state successfully merges short idle periods into a single large period, which is useful for completing a long guest task without cancelation. In Figs. 7 and 8, host machine #2 shows a different behavior compared with the other machines. Our system slightly increases the detected idle period length, which in turn results in a slight increase in terms of guest application throughput. We find that the mouse and keyboard of this machine were infrequently operated by the owner. Consequently, features of the owner's activity determines the effectiveness of our system.

Finally, we measured the number of state transitions on host machines. Figure 9 shows the measured number per minute. Owing to the relaxed definition, our system achieved fewer transitions than the previous system. The numbers were reduced by 40%–96%, so our system will allow the resource management server to register more host machines than the previous system.

5.2 Host Frame Rate

Next, we evaluate host frame rates of our system using the same guest application (i.e., matrix multiplication). We used a web browser and an OpenGL-based shader [27] as host applications. The web browser, Internet Explorer 11, iteratively replays a specific movie file provided by YouTube, whereas the OpenGL-based shader renders multiple spheres using the Phong reflection model [23]. The movie file was replayed using GPU acceleration.

Table 3 shows the frame rate measured during the simultaneous execution of guest and host applications.

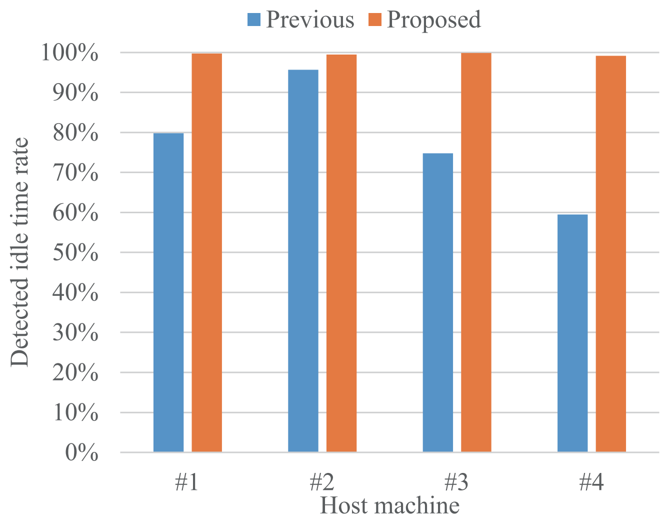


Figure 7: Detected idle time rate over system uptime.

Table 3: Measured frame rate in fps.

Host application	Original	Naive	Multitasking
Browsing	28.7	4.0	28.6
Rendering	60.0	0.3	60.0

The frame rate was measured using Fraps [3]. For both host applications, the frame rate of at least 28.7 fps is significantly reduced to at most 4.0 fps. In particular, the OpenGL-based shader resulted in only 0.3 fps because of the running guest application. In contrast, the frame rate observed with our multitasking technique is almost the same as the original rate. This means that our system successfully minimizes host perturbation. Actually, hosts were not aware of guest tasks running in the background.

Table 4 shows the throughput ratio $P = p_1/p_2$ of matrix multiplication, where p_1 is the execution time obtained under dedicated execution (i.e., the maximum performance of the guest application) and p_2 is that obtained under simultaneous execution (i.e., the shared performance). In addition, it shows the continuous ratio $Q = q_1/(q_1 + q_2)$ of the selected execution mode, where q_1 is the number of subtasks executed in continuous mode and q_2 is that of subtasks executed in periodical mode.

Without host applications, the throughput ratio P reaches 93.7%, implying that the multitasking overhead reduces the guest application throughput by 6.3%. At the same time, our multitasking system appropriately selects continuous execution mode with a ratio of 92.8%. The remaining 7.2% was executed in periodical execution mode. In contrast, with highly-loaded host applications, such as a web browser and an OpenGL renderer, our system selects periodical mode to execute guest tasks. The continuous ratio was 78% and 100% for the web browser and the OpenGL-based renderer, respectively. Consequently, the guest application throughput reduces to less than half, but the original frame rate is maintained as presented in Table 3.

Finally, we measured the minimum and maximum of core and memory clock frequencies. Figure 10 shows the results observed using the web browser and matrix multiplication as host and guest applications, respectively. Similar results were obtained using the OpenGL-based shader. According to the scenes to be rendered, our device dynamically changed both the core and memory clock frequencies. The minimum frequencies were less than the base lines (1006 MHz for the core clock and 1502 MHz for the memory clock) if only the host application was executed on the device. In contrast, the device boosted up the clock speed beyond the baselines if it executed matrix multiplication. This behavior was observed whether using our multitasking technique or not. Consequently, our performance results were affected by the boost, but the throughput ratio P was not biased toward upward.

Table 4: Analysis of guest application throughput and selected execution mode.

Host application	Throughput ratio P (%)	Continuous ratio Q (%)
Nothing	93.7	92.8
Browsing	45.9	21.8
Rendering	35.0	0.0

6 Conclusion and Future

We have presented a GPU-accelerated grid system capable of utilizing short idle time spanning hundreds of milliseconds. Our cooperative multitasking technique realizes the concurrent execution of host and guest applications, minimizing host perturbation. Our technique eliminates the mouse and keyboard monitoring process required in previous systems. Our monitoring process checks only CPU and video memory usages, according to a relaxed definition of an idle resource. This relaxation reduces not only the number of state transitions, but also of communication messages between resources and the resource management server.

We performed a case study in which our system was applied to four desktop machines in our laboratory. Compared to a previous screensaver-based system, our cooperative system detected 1.7 times longer idle time. Consequently, our system not only avoided frame rate degradation but also achieved a 91% higher guest throughput, realizing the efficient utilization of idle resources. Furthermore, our system reduced the server workload by reducing the number of state transitions by 96%.

Future work includes detailed evaluation using more practical applications in a large-scale environment. We plan to apply our system to a homology search problem [16]. NVIDIA has announced that their next-generation GPU architectures, Maxwell and Volta, will support preemption and unified virtual memory. Such preemptive architectures will require a task scheduler to find the best tradeoff point between the frame rate of host machines and the throughput of guest tasks.

Acknowledgement

This study was supported in part by the Japan Society for the Promotion of Science KAKENHI Grant Numbers 23700057 and 23300007 and the Japan Science and Technology Agency 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] David P. Anderson. BOINC: A system for public-resource computing and storage. In *Proc. 5th IEEE/ACM Int’l Workshop Grid Computing (GRID’04)*, pages 4–10, November 2004.

[2] Adam L. Beberg, Daniel L. Ensign, Guha Jayachandran, Siraj Khaliq, and Vijay S. Pande. Folding@home: Lessons from eight years of volunteer distributed computing. In *Proc. 26th IEEE Int’l Parallel and Distributed Processing Symp. (IPDPS’09)*, April 2009. 8 pages (CD-ROM).

[3] Beepa Pty Ltd. Fraps: real-time video capture & benchmarking, 2014. `http://www.fraps.com/`.

[4] David Blythe. The Direct3D 10 system. *ACM Trans. Graphics*, 25(3):724–734, July 2006.

[5] Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, and Vasily Volkov. Parallel computing experiences with CUDA. *IEEE Micro*, 28(4):13–27, July 2008.

[6] GPUGRID.net, 2014. `http://www.gpugrid.net/`.

[7] Fumihiko Ino. The past, present, and future of GPU-accelerated grid computing. In *Proc. 1st Int'l Symp. Computing and Networking (CANDAR'13)*, pages 17–21, December 2013.

[8] Fumihiko Ino, Yuki Kotani, Yuma Munekawa, and Kenichi Hagihara. Harnessing the power of idle GPUs for acceleration of biological sequence alignment. *Parallel Processing Letters*, 19(4):513–533, December 2009.

[9] Fumihiko Ino, Yuma Munekawa, and Kenichi Hagihara. Sequence homology search using fine grained cycle sharing of idle GPUs. *IEEE Trans. Parallel and Distributed Systems*, 23(4):751–759, April 2012.

[10] Fumihiko Ino, Akihiro Ogita, Kentaro Oita, and Kenichi Hagihara. Cooperative multitasking for GPU-accelerated grid systems. *Concurrency and Computation: Practice and Experience*, 24(1):96–107, January 2012.

[11] Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott Brandt. Gdev: First-class GPU resource management in the operating system. In *Proc. 2012 USENIX Ann. Technical Conf. (ATC'12)*, June 2012. 12 pages (CD-ROM).

[12] Yuki Kotani, Fumihiko Ino, and Kenichi Hagihara. A resource selection system for cycle stealing in GPU grids. *J. Grid Computing*, 6(4):399–416, December 2008.

[13] Tyng-Yeu Liang, Yu-Wei Chang, and Hung-Fu Li. A CUDA programming toolkit on grids. *Int'l J. Grid and Utility Computing*, 3(2/3):97–111, May 2012.

[14] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, March 2008.

[15] Microsoft Corporation. Windows API, 2014. `http://msdn.microsoft.com/en-us/library/cc433218(VS.85).aspx`.

[16] Yuma Munekawa, Fumihiko Ino, and Kenichi Hagihara. Accelerating Smith-Waterman algorithm for biological database search on CUDA-compatible GPUs. *IEICE Trans. Information and Systems*, E93-D(6):1479–1488, June 2010.

[17] NVIDIA Corporation. GPU Computing SDK, 2012. `http://developer.nvidia.com/gpu-computing-sdk/`.

[18] NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110, May 2012. `http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`.

[19] NVIDIA Corporation. CUDA C Programming Guide Version 5.0, October 2012. `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`.

[20] NVIDIA Corporation. NVIDIA GRID, 2014. `http://www.nvidia.com/object/cloud-gaming.html`.

[21] Yusuke Okitsu, Fumihiko Ino, and Kenichi Hagihara. High-performance cone beam reconstruction using CUDA compatible GPUs. *Parallel Computing*, 36(2/3):129–141, February 2010.

[22] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, March 2007.

[23] Bui Tuong Phong. Illumination for computer generated pictures. *Communication of the ACM*, 18(6):311–317, June 1975.

[24] C. Reaño, A. J. Peña, F. Silla, J. Duato, R. Mayo, and E. S. Quintana-Ortí. CU2rCU: towards the complete rCUDA remote GPU virtualization and sharing solution. In *Proc. 19th Int'l Conf. High Performance Computing (HiPC'12)*, December 2012. 10 pages (CD-ROM).

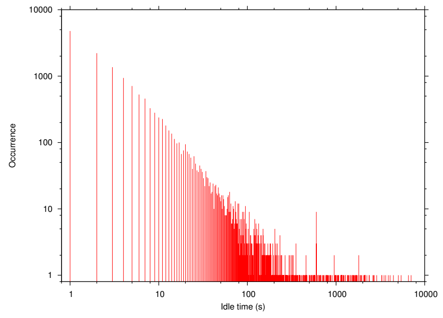
[25] Lin Shi, Hao Chen, and Jianhua Sun. vCUDA: GPU-accelerated high-performance computing in virtual machines. *IEEE Trans. Computers*, 61(6):804–816, June 2012.

[26] Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. *OpenGL Programming Guide*. Addison-Wesley, Reading, MA, fifth edition, August 2005.

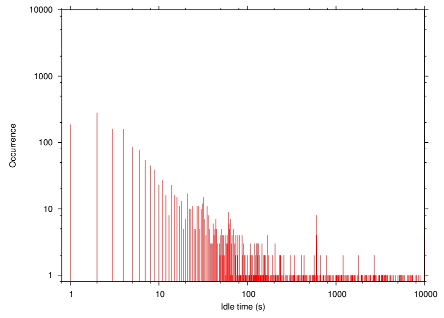
[27] Tadamasa Teranishi. Phong shading sample program, 2003. `http://www.asahi-net.or.jp/~yw3t-trns/opengl/samples/fshphong/`.

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

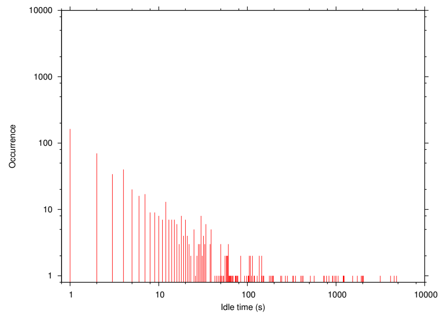
[29] Shinichi Yamagiwa and Leonel Sousa. Caravela: A novel stream-based distributed computing. *IEEE Computer*, 40(5):70–77, May 2007.



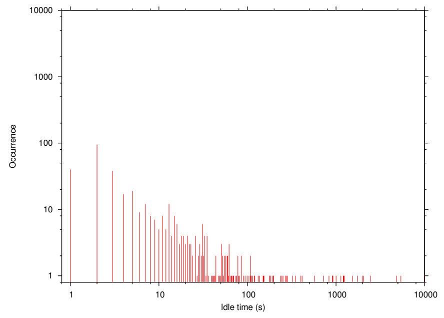
(a)



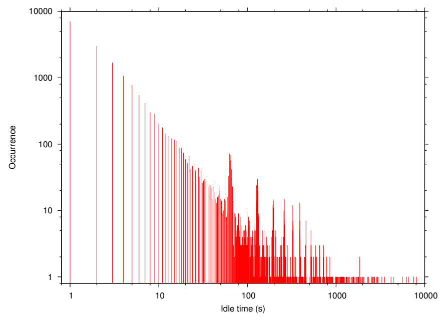
(b)



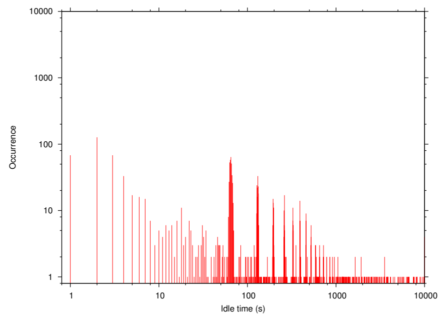
(c)



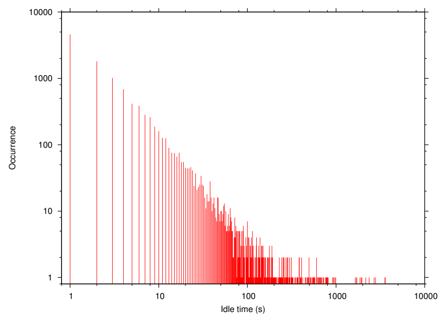
(d)



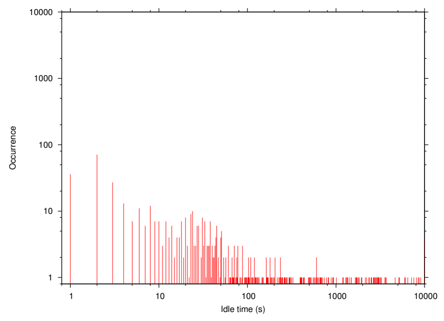
(e)



(f)



(g)



(h)

Figure 8: Distribution of idle period lengths. Results on host machine #1 using (a) the previous system and (b) our system, those on host machine #2 using (c) the previous system and (d) our system, those on host machine #2 using (e) the previous system and (f) our system, and those on host machine #2 using (g) the previous system and (h) our system,

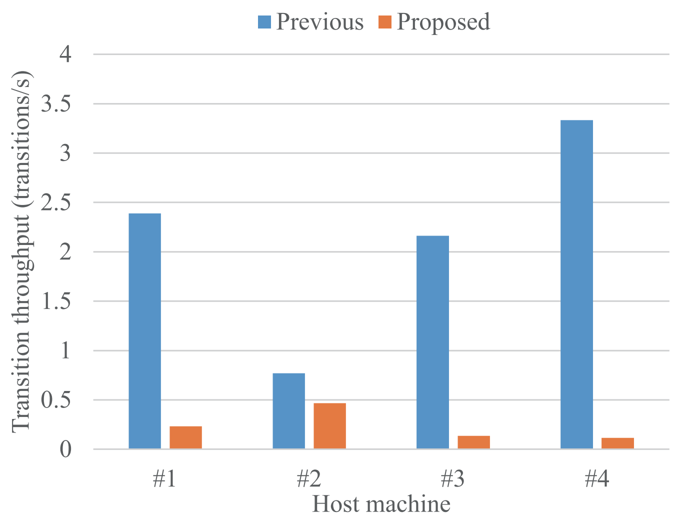


Figure 9: Number of state transitions per minute.

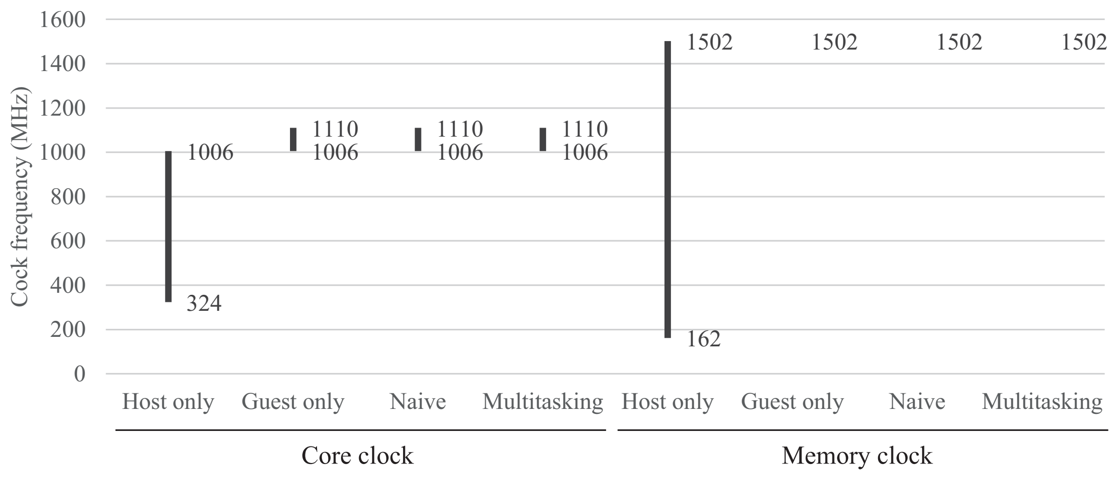


Figure 10: Minimum and maximum of core and memory clock frequencies.