A Resource Selection Method for Cycle Stealing in the GPU Grid*

Yuki Kotani, Fumihiko Ino, and Kenichi Hagihara

Graduate School of Information Science and Technology, Osaka University
1-3 Machikaneyama, Toyonaka, Osaka 560-8531, Japan
{y-kotani, ino, hagihara}@ist.osaka-u.ac.jp

Abstract. Modern programmable graphics processing units (GPUs) provide increasingly higher performance, motivating us to perform general-purpose computation on the GPU (GPGPU) beyond graphics applications. In this paper, we address the problem of resource selection in the GPU grid. The GPU grid here consists of desktop computers at home and the office, utilizing idle GPUs and CPUs as computational engines for compute-intensive applications. Our method tackles this challenging problem (1) by defining idle resources and (2) by developing a resource selection method based on a screensaver approach with low-overhead sensors. The sensors detect idle GPUs by checking video random access memory (VRAM) usage and CPU usage on each computer. Detected resources are then selected according to a matchmaking framework and benchmark results obtained when the screensaver is installed on the machines. The experimental results show that our method achieves a low overhead of at most 262 ms, minimizing interference to resource owners with at most 10% performance drop.

1 Introduction

Grid technology [1] has emerged as a new paradigm in computational science. It allows us to share hardware and software resources across multiple organizations, providing us a virtual supercomputer through the Internet. There are many types of grids such as server grids, desktop grids, and data grids [2]. In this paper, we use the term grid to refer to a desktop grid, namely a cycle stealing system that utilizes idle computers at home and the office.

Another emerging paradigm is GPGPU [3], which stands for general-purpose computation on the graphics processing unit (GPU) [4, 5]. The GPU is a single chip processor designed for acceleration of compute-intensive graphics tasks, such as three-dimensional (3-D) rendering applications. Modern GPUs are increasing in computational performance at greater than Moore's law [6]. For example, an nVIDIA GeForce 6800 card achieves a peak performance of 120 GFLOPS for single precision data [7]. In addition to their attractive performance, GPUs are becoming more flexible in programmability with supporting branching. Consequently, many researchers are trying to apply the GPU to non-graphics problems [8] as well as typical graphics problems.

* This work was partly supported by JSPS Grant-in-Aid for Scientific Research for Scientific Research (B)(2)(18300009) and on Priority Areas (17032007).

In this paper, we focus on enabling GPGPU applications to execute on desktop grids. This type of grids, named GPU grids, aims at exploiting idle GPUs as well as idle CPUs at home and the office. Thus, we think that desktop grids will become a more attractive computing platform if GPUs are explicitly managed and used as general-purpose resources as well as graphics accelerators.

Since GPUs have been used as dedicated display cards, many technical problems arise if these cards are shared between resource owners and grid users. A resource owner here is the person who contributes resources to the grid while a grid user means the person who desires to run grid applications on donated resources. One typical problem is resource conflicts between owners and users. In particular, the following critical problems must be resolved to achieve our goal of building GPU grids.

P1. The lack of definition of idle resources. GPUs are originally designed to serve their owners who directly see the display output. Therefore, the definition of idle resources has not been considered from the grid users point of view. We must define this to select appropriate resources for grid users. The definition here should be considered from both the owner side and the user side in order to (1) minimize interference to owners and (2) maximize application performance provided to users.

P2. The lack of external monitors for the GPU. Most operating systems are capable of providing CPU performance information such as load average and memory usage. However, current operating systems do not have information on GPU performance. Although modern GPUs have performance counters inside their chips, these internal counters are accessible only from instrumented programs running with an instrumented device driver [9]. Therefore, we need external monitors to minimize modifications to resource configurations and application code.

P3. The lack of efficient multitasking on the GPU. Current GPUs do not support context switching in hardware, so that preemptive multitasking of GPU applications is not available even in Windows XP, namely the most popular system [10]. Instead, multitasking is cooperatively done by software, which results in lower performance. Thus, GPUs are still not virtualized enough to allow multiple applications to be run effectively.

Although problem P3 is critical, it is not easy for non-vendors to give a direct solution to this problem. Therefore, assuming that the GPU grid consists of cooperative multitasking systems, we tackle the remaining problems P1 and P2 to select idle GPUs appropriately from grid resources.

To address problem P1, we experimentally define the idle state of the GPU. For problem P2, on the other hand, we develop a resource selection method based on a screensaver approach with low-overhead sensors. The sensors detect idle GPUs according to video random access memory (VRAM) usage and CPU usage on each computer. Once idle GPUs are detected, they are selected according to a matchmaking framework [11] and benchmark results. The benchmark results here are obtained when the screensaver is installed on each of the resources. Our method is currently implemented on Windows systems, which support the latest GPUs for entertainment use.

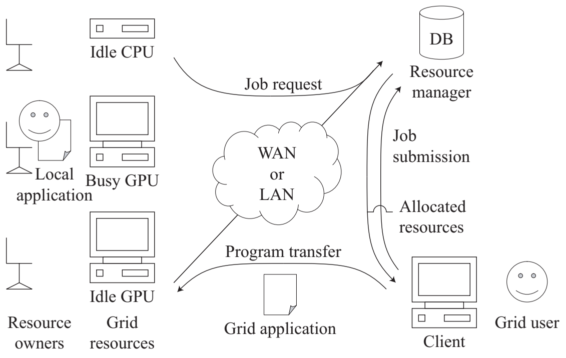


Fig. 1. Overview of the GPU Grid.

2 GPU Grid

The GPU grid has almost the same structure as existing desktop grids. The only difference is that the GPU grid explicitly manages the GPU as general-purpose resources. We think that this little difference allows us to easily integrate our resource selection method into existing desktop grid systems.

2.1 System Overview

Figure 1 shows an overview of the GPU grid, which consists of three main components as follows.

- Grid resources. Grid resources are desktop computers at home and the office connecting to the Internet. Ordinarily, these resources are used by resource owners. However, they are donated for job execution if they are in the idle state. Arbitrary computers are considered as grid resources regardless of having the programmable GPU or not.
- The resource manager. The resource manager takes the responsibility for monitoring and selection of registered resources. It also acts as a job manager, which receives jobs from grid users. For each job, the manager returns a list of available resources. This list contains idle resources waiting for job allocation, and thus matchmaking [11] is done using this list (see Section 3.2). We accept arbitrary jobs consisting of GPGPU, GPU, and CPU applications.
- Clients. Clients are front-end computers for grid users, who want to submit jobs to the grid. Clients can also be grid resources. Once the list of available resources is sent from the resource manager, the user program is sent to the resources for job execution.

Thus, the GPU grid is a wider concept of the desktop grid. Therefore, a desktop grid without GPU-equipped computers also can be regarded as the GPU grid.

In the following discussion, we use the term grid application to denote a program submitted by grid users. We also use the term local application to denote a program executed by resource owners using their resources.

Table 1. Classification of owner's activities.

Situation	CPU	GPU	Owner's activity
S1	Idle	Idle	Nothing
S2	Busy	Idle	Web browsing, movie seeing, music listening
S3	Idle	Busy	(unrealistic)
S4	Busy	Busy	Video gaming

2.2 Definition of Idle Resources

Since a grid resource have a CPU and possibly a GPU, the resource state can be roughly classified into four groups depending on the state of each unit. Table 1 presents this classification with owner's typical activities. In the following we show the definition of idle resources using this classification.

As we mentioned before, the definition must satisfy the following requirements.

R1. It minimizes interference to resource owners.
R2. It maximizes application performance provided to grid users.

To satisfy the above requirements, we define an idle resource such that it satisfies all of the following three conditions.

$D1$**.** The resource owner does not interactively operate the resource.
$D2$**.** The GPU does not execute any local application.
$D3$**.** The CPU is idle enough to provide the full performance of the GPU to grid users.

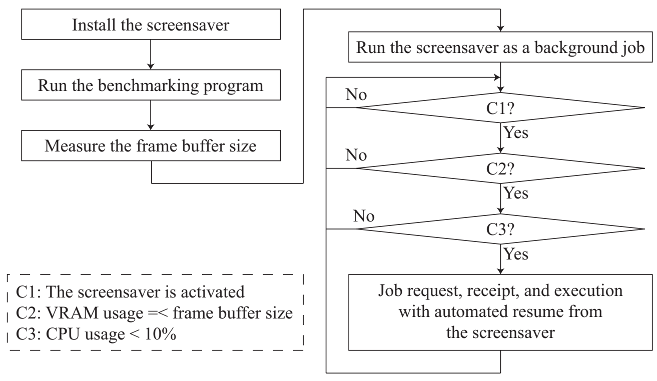
Firstly, condition $D2$ is essential to satisfy requirement R1, because the GPU does not support preemptive multitasking. Otherwise, some uncooperative applications will significantly drop the frame rate of the display, making resource owners nervous. Consequently, resource owners are interfered by grid applications if condition $D2$ is not satisfied. Thus, R1 excludes situations S3 and S4 from the idle state (See Table 1).

Secondly, due to the same reason, R1 also excludes situation S2 if the resource owner interactively operates their computer through the display output. We also have experimentally confirmed that the grid application suffers from lower performance if the resource owner gives a window focus to the operating window (see Section 4.1). Therefore, situation S2 does not satisfy requirement R2. Thus, condition $D1$ is needed.

Finally, condition $D3$ is essential to satisfy requirement R2. We have experimentally confirmed this (see Section 4.1). GPU applications generally make the CPU usage go to 100%, because they usually require CPU intervention during GPU execution. Note here that this condition might be eliminated in the future, because Windows Graphics Foundation (WGF) 2.0 will enable GPU processing without CPU intervention [12].

3 Resource Selection Method

In this section, we describe how idle resources are detected and how resources are selected from detected resources.



Install the screensaver
Run the benchmarking program
Measure the frame buffer size
Run the screensaver as a background job
C1?
C2?
C3?
No
No
No
Yes
Yes
Yes
Job request, receipt, and execution with automated resume from the screensaver
C1: The screensaver is activated
C2: VRAM usage =< frame buffer size
C3: CPU usage < 10%

Fig. 2. Resource detection procedure. Steps in the left-hand side are processed only once when the screensaver is installed on the resource.

3.1 Detection of Idle Resources

Figure 2 shows the procedure of resource detection. To detect an idle resource that satisfies conditions $D1$–$D3$, our method checks the resource in the following steps.

C1. The screensaver is activated.
C2. VRAM usage $\leq$ frame buffer size.
C3. CPU usage $< 10\%$.

Steps C1, C2, and C3 here aim at checking conditions $D1$, $D2$, and $D3$, respectively.

The first condition $D1$ is checked by a screensaver approach. The screensaver is currently activated after five minutes of owner's inactivity. This screensaver approach allows us to detect inactivity at a lower overhead. It also allows owners to rapidly resume their activity, as compared with polling-based methods [13].

Due to the lack of preemptive multitasking supports, our screensaver avoids updating the display output. Instead, the display is drawn only when the screensaver is turned on. This intends to avoid increasing the workload of the CPU and the GPU during the screensaver mode, delivering full GPU performance to grid users. The screensaver is implemented using a library scrnsave.lib, which is distributed as a part of Microsoft Visual Studio.

The remaining conditions $D2$ and $D3$ are checked by a sensor program. This program is implemented as a screensaver function ScreenSaverProc(), which is called when the screensaver is activated. Thus, we minimize the monitoring overhead by minimizing the invocation of the sensor program.

The key idea to evaluate condition $D2$ is the VRAM usage check. This idea assumes that the GPU always consumes VRAM for the frame buffer and further VRAM if it executes any GPU programs. Under this assumption, we can evaluate condition $D2$ by comparing the current usage and the default usage, namely the frame buffer size. The default usage here is measured only once when installing the screensaver on the resource. Our VRAM-based monitoring method has two advantages as follows.

- No modification. The VRAM usage can be easily investigated using a Direct Draw function GetCaps(), which is initially available in Windows computers. Thus, we do not need any special libraries and hardware at grid resources.
- Lower overhead. The function GetCaps() obtains the VRAM usage from the device driver. Therefore, this information is obtained without GPU intervention, leading to a low-overhead sensor.

Note here that GetCaps() does not directly give the VRAM usage. This function returns the capacity and the amount of free space, so we subtract them to estimate the usage.

The assumption mentioned above is valid in the current GPU, which allocates VRAM in advance of an execution. Furthermore, the GPU always consumes VRAM for the frame buffer size to refresh the display. Although the amount of this consumption might be computed according to the screen resolution and its color depth, we have found that it varies depending on hardware and software environments, such as the device driver version. Therefore, we directly measure the default usage at screensaver installation.

Finally, condition $D3$ can be evaluated by accessing performance information provided by operating systems. According to preliminary experiments (see Section 4.1), we currently use the CPU usage with a threshold of 10%. As well as the VRAM usage, this information does not require GPU intervention. Our implementation calls PdhCollectQueryData() to access a performance counter in the Windows operating system.

3.2 Selection of Idle Resources

Once idle resources are detected by the screensaver approach, the next issue is the resource selection problem. We resolve this issue by combining two different approaches: a benchmarking approach [14] and a matchmaking approach [11].

The benchmarking approach [14] takes the responsibility for measuring the actual performance for GPGPU applications. The reason why we perform benchmarking is due to the fact that the specification of the GPU does not always represent the actual performance for GPGPU applications. Actually, we have found that a high-end card is outperformed by a commodity card and that the device driver version significantly affects the performance. Therefore, we run a benchmark program at screensaver installation to obtain the actual performance under the idle state. These benchmark results are then collected at the resource manager to give priorities to detected resources.

On the other hand, the matchmaking approach [11] is responsible for providing a flexible and general framework of resource selection. For example, this framework allows grid users to select only nVIDIA GeForce 7800 cards or select only GPUs with having a fill rate of at least 3 Gpixels/s, according to the benchmark results. We think that this flexible framework is essential to run GPGPU applications in grid environments, because the GPU is still not a matured computing environment. We have experienced that some applications running on a GPU do not successfully run on different GPUs, due to architectural differences and driver version differences. Therefore, we think that the framework should allow users to select appropriate resources.

Table 2. Specification of experimental machines.

	PC1	PC2	PC3
CPU	Pentium 4 3.4 GHz	Pentium 4 3.0 GHz	Pentium 4 2.8 GHz
GPU	nVIDIA GeForce 7800 GTX	nVIDIA GeForce 6800 GTO	nVIDIA Quadro FX 3400
Core speed (MHz)	430	350	350
Memory speed (MHz)	1200	900	900
Memory bandwidth (GB/s)	38.4	28.8	28.8
Fill rate (Gpixels/s)	6.88	4.2	5.6
Pipeline engines	24	12	16
Graphics bus	PCI Express		
Driver version	79.70	78.01	66.93

4 Experimental Results

Table 2 shows the specification of experimental machines. We use three machines PC1, PC2, and PC3, each with different CPUs and GPUs. PC1 and PC3 provide the highest and the lowest performance, respectively.

For experiments, we use three GPGPU applications: LU decomposition [15], conjugate gradients (CG) [16], and 2-D/3-D rigid registration (RR) [17]. Due to the space limitation, we briefly summarize each characteristic.

- LU decomposition of a 2048×2048 matrix. In this implementation [15], the matrix data is stored as textures in the VRAM. Textures are then repeatedly rendered by the hardware components in the GPU, such as SIMD and vector processing units. The CPU takes the responsibility for computing the working area in textures where the GPU operates.
- CG for solving linear systems with a coefficient matrix of size 64×64. Similar to LU decomposition, this implementation also repeats rendering against textures. From the viewpoint of the workload characteristic, it has less CPU workload than LU decomposition because the GPU is responsible for computing the working area.
- RR for alignment between 2-D images and a 3-D volume. This implementation [17] has the lowest workload of the CPU among the three applications. In contrast, computation at the GPU side is the heaviest because it operates a volume data in addition to 2-D images.

4.1 Evaluating Definition of Idle Resources

To verify the definition of idle resources, we compare the performance of local and grid applications between idle resources and busy resources. The performance is presented by application throughput given by the number of executions per second.

According to the definition presented in Section 2.2, we measure the throughput on busy resources. Busy states here are given by negations of each of conditions $D1$, $D2$, and $D3$ as follows.

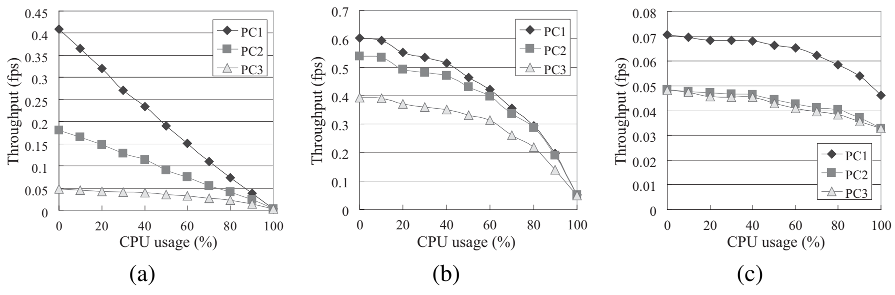


Fig. 3. Measured throughput with different CPU usages for three different GPGPU applications: (a) LU decomposition, (b) conjugate gradients (CG), and (c) rigid registration (RR). These applications are executed as grid applications. Throughput is presented in frames per second (fps).

$\overline{D1}$: The resource owner interactively operates the resource. We measure the throughput of local and grid applications while executing the PCMark05 benchmark [18]. PCMark05 here renders various web pages, and thus this experiment measures interference to owners assuming that they are browsing web pages during grid job execution.

$\overline{D2}$: The GPU executes local applications. We measure the throughput of local applications while executing a grid application. We use LU, CG, or RR application for each side. This experiment also intends to measure the interference to GPU applications instead of CPU applications (web browsing as mentioned above).

$\overline{D3}$: The CPU is not idle enough to provide the full performance of the GPU to grid users. We measure the throughput of grid applications with different CPU usages, ranging from 0% to 100%.

To obtain accurate throughputs, both local and grid applications are executed in an infinite loop during measurement.

Figure 3 gives the results for condition $\overline{D3}$. It shows the throughput of grid applications with different CPU usages. For all applications, we can see that the throughput decreases as the CPU usage increases. One remarkable point here is that LU linearly drops the performance while RR slowly decreases the performance. This is due to the difference of workload characteristics inherent in applications. As compared with CG and RR, LU frequently switches textures, and thus requires more CPU interventions during execution. It also requires more data transfer between the CPU and the GPU. Therefore, LU sharply drops the performance as compared with CG and RR. According to these results, we have determined that idle resources must have a CPU usage of at most 10%.

Figure 4 shows the results for conditions $\overline{D1}$ and $\overline{D2}$. It shows the throughput of grid applications with different activities: web browsing (PCMark05), LU, CG, and RR execution. LU and CG in Figs 4(a) and 4(b) indicate that grid applications significantly drop their performance if owners are seeing web pages. In contrast, the performance drop in RR is not so serious. We think that this is due to CPU interventions required

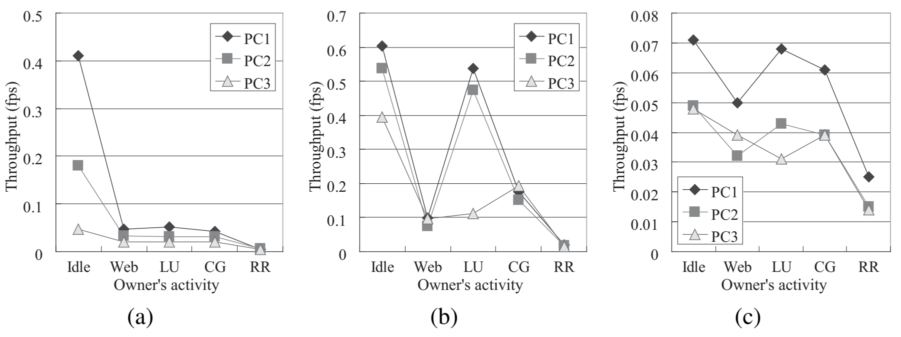


Fig. 4. Measured throughput with different owner's activities for three GPGPU applications: (a) LU decomposition, (b) conjugate gradients (CG), and (c) rigid registration (RR). Applications in the horizontal axis are local applications.

during GPU execution. As we mentioned earlier, LU and CG require more interventions than RR. We also think that the window focus is critical in these cases. Since the focus is given to owner's operating window, PCMark05 in this case, the operating system gives a lower priority to the background job, namely the grid application. This increases the overhead of CPU interventions, making the GPU in the idle state. Thus, resources in condition $\overline{D1}$ should not be used for job execution. We also confirmed this from the owner side. The rendering performance of web pages is decreased from approximately 2 to 0.5 pages/s.

Finally, we investigate condition $\overline{D2}$. In Fig. 4, we can see that the throughput of grid applications also significantly decreases if the owner executes a GPU application. In particular, RR seems to be an uncooperative application, because it significantly decreases the performance of LU and CG, as shown in Figs 4(a) and (b), respectively. Furthermore, if RR is executed as a grid application, it provides almost the same performance, whether it is executed on an idle resource or a busy resource. Thus, since both the grid users and resource owners can execute uncooperative applications on resources, we think that condition $D2$ is needed to define idle resources.

In summary, we think that the definition is reasonable with minimizing interference to resource owners while maximizing application performance provided to grid users.

4.2 Evaluating Overhead of Resource Selection

We now evaluate our resource selection method in terms of the monitoring overhead. We also investigate how local applications are interfered by the method. In experiments, we use LU, CG, and RR as local applications.

Table 3 shows the execution time of local applications, explaining how local applications are perturbated by the resource monitoring overhead. We first measured the original time T_1 with disabling resource monitoring, and then time T_2 with enabling

Table 3. Perturbation effects measured using three local applications. T_1 and T_2 represent the execution time without resource monitoring and that with monitoring, respectively. $T_2 - T_1$ represents the perturbation time increased by monitoring. σ denotes the perturbation ratio, where $\sigma = (T_2 - T_1)/T_1 * 100$. Times are presented in seconds.

Local	PC1 (s)			PC2 (s)			PC3 (s)		
application	T_1	T_2	$T_2 - T_1$ (σ)	T_1	T_2	$T_2 - T_1$ (σ)	T_1	T_2	$T_2 - T_1$ (σ)
LU	2.4	2.5	0.1 (4%)	5.6	5.7	0.1 (2%)	21.1	21.2	0.1 (1%)
CG	1.7	1.8	0.1 (6%)	1.9	2.1	0.2 (10%)	2.5	2.6	0.1 (4%)
RR	14.2	14.5	0.3 (2%)	18.6	18.8	0.2 (1%)	20.7	21.0	0.3 (1%)

monitoring. Therefore, the perturbation time $T_2 - T_1$ explains how applications are perturbated by monitoring.

We observe the highest perturbation time $T_2 - T_1$ of 300 ms when executing RR on computer PC1. The perturbation ratio σ also indicates that this time is short enough as compared with the original execution time T_1. Furthermore, our monitoring program is implemented as a CPU program, and thus it avoids making GPU programs slow down.

The perturbation time $T_2 - T_1$ of 300 ms is due to the monitoring overhead of 262 ms: 190 ms for activating the screensaver; 2 ms for checking the VRAM usage; and 70 ms for the CPU usage. Although the screensaver activation takes 190 ms at the GPU side, it does not cause critical interference to the resource owner, because the activation guarantee the owner's inactivity. One concern is the interference to GPGPU applications that do not require interaction between owners. However, this is not so critical because our screensaver avoids refreshing the display. The remaining time for checking the VRAM and CPU usages is also a low overhead, because it requires only references to performance information, which is processed at the CPU side. Thus, we think that our method achieves a low overhead monitoring with minimum interference.

5 Related Work

To the best of our knowledge, there is no work on utilizing the GPU as a general-purpose resource in the grid. However, some grid projects use the GPU as a graphics accelerator to achieve large-scale visualization in server grid environments [19, 20], in which resources are dedicated to grid users. Due to this dedication, server grids do not have resource conflicts between resource owners and grid users. Therefore, resources can be easily managed by a job management server that receives jobs from grid users. A similar work is presented by Fan et al. [21] who build a cluster of GPUs for fluid simulation and visualization.

There are many projects related to desktop grids. Condor [13] is an earlier system that explores using idle time in networked workstations. This system has a central server that polls every two minutes for available CPUs and jobs waiting. Each workstation has a local scheduler that checks every 30 seconds to see if the running job should be preempted because the owner has resumed using the workstation. Thus, owners are interfered for 30 seconds at the worst case. This interfering time is too long for cooperative multitasking systems, which can significantly drop the frame rate of the display.

BOINC [22] is a middleware system of the SETI@home project [23], which demonstrates the practical use of desktop grids. This system has a screensaver mode that shows the graphics of running applications. Although this mode is useful to know that resource owners currently do not operate their computers, it is not sufficient to decide if the GPU is not being used. Thus, some additional monitors are needed for the GPU.

NVPerfKit [9] is a monitoring tool that allows us to probe performance counters in the GPU. This tool gives us important performance information such as the ratio of the idle time to the total measured time. However, it requires modern nVIDIA GPUs with an instrumented version of the device driver to probe the counters. Therefore, this vendor-specific tool is not a realistic solution to our problem, where various GPUs should be monitored without system or code modifications.

Benchmarking tools provide us effective performance information based on direct execution of some small code. For example, 3DMark06 [18] measures GPU performance using a set of 3-D graphics applications. On the other hand, gpubench [14] aims at capturing GPU performance for GPGPU applications. Thus, benchmarking tools might be useful to detect idle GPUs. However, they require a couple of time to finish benchmarking. This high overhead is critical if they are executed every time the resource is checked for availability.

With respect to multitasking of GPU applications, Windows Vista will support preemptive multitasking [10]. As compared with cooperative multitasking, preemptive multitasking provides more stable, reliable performance when multiple applications are executed simultaneously. Therefore, our assumption of cooperative multitasking might lead to a strict definition of the idle GPU in the future. However, we think that this assumption is compatible with future systems, because we only have to relax the definition to gather more resources for such preemptive multitasking systems.

6 Conclusion

We have presented a resource selection method for the GPU grid, which aims at executing GPGPU applications on a desktop grid. We also have shown a definition of idle resources in the GPU grid. Both the method and definition works on cooperative (non-preemptive) multitasking systems. The method employs a screensaver-based approach with low-overhead monitors. The monitors are designed to detect idle GPUs with minimum program invocations.

The experimental results show that the definition is reasonable with minimizing interference to resource owners while maximizing application performance provided to grid users. We also find that the method achieves a low overhead of at most 262 ms, which is short enough as compared to the execution time of local applications.

References

1. Foster, I., Kesselman, C., eds.: The Grid: Blueprint of a New Computing Infrastructure. Morgan Kaufmann, San Mateo, CA (1998)
2. Chien, A., Calder, B., Elbert, S., Bhatia, K.: Entropia: architecture and performance of an enterprise desktop grid system. J. Parallel and Distributed Computing **63**(5) (2003) 597–610

3. GPGPU: General-Purpose Computation Using Graphics Hardware (2005) `http://www.gpgpu.org/`.
4. Fernando, R., ed.: GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics. Addison-Wesley, Reading, MA (2004)
5. Pharr, M., Fernando, R., eds.: GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation. Addison-Wesley, Reading, MA (2005)
6. Moore, G.E.: Cramming more components onto integrated circuits. Electronics **38**(8) (1965) 114–117
7. Montrym, J., Moreton, H.: The GeForce 6800. IEEE Micro **25**(2) (2005) 41–51
8. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A.E., Purcell, T.J.: A survey of general-purpose computation on graphics hardware. In: EUROGRAPHICS 2005, State of the Art Report. (2005) 21–51
9. nVIDIA Corporation: NVPerfKit 2 User Guide (2006) `http://developer.nvidia.com/NVPerfKit/`.
10. Pronovost, S., Moreton, H., Kelley, T.: Windows display driver model (WDDM) v2 and beyond. In: Windows Hardware Engineering Conf. (WinHEC'06). (2006) `http://www.microsoft.com/whdc/winhec/trackdetail06.mspx?track=11`.
11. Raman, R., Livny, M., Solomon, M.: Resource management through multilateral matchmaking. In: Proc. 9th IEEE Int'l Symp. High Performance Distributed Computing (HPDC'00). (2000) 290–291
12. Blythe, D.: Windows graphics overview. In: Windows Hardware Engineering Conf. (WinHEC'05). (2005) `http://www.microsoft.com/whdc/winhec/Pres05.mspx`.
13. Litzkow, M.J., Livny, M., Mutka, M.W.: Condor - a hunter of idle workstations. In: Proc. 8th Int'l Conf. Distributed Computing Systems (ICDCS'88). (1988) 104–111
14. Buck, I., Fatahalian, K., Hanrahan, P.: GPUBench: Evaluating GPU performance for numerical and scientific application. In: Proc. 1st ACM Workshop General-Purpose Computing on Graphics Processors (GP^2'04). (2004) C-20
15. Ino, F., Matsui, M., Hagihara, K.: Performance study of LU decomposition on the programmable GPU. In: Proc. 12th IEEE Int'l Conf. High Performance Computing (HiPC'05). (2005) 83–94
16. Corrigan, A.: Implementation of conjugate gradients (CG) on programmable graphics hardware (GPU) (2005) `http://www.cs.stevens.edu/~quynh/student-work/acorrigan_gpu.htm`.
17. Ino, F., Gomita, J., Kawasaki, Y., Hagihara, K.: A GPGPU approach for accelerating 2-D/3-D rigid registration of medical images. (In: Proc. 4th Int'l Symp. Parallel and Distributed Processing and Applications (ISPA'06). (2006)
18. Futuremark Corporation: Products (2006) `http://www.futuremark.com/products/3dmark06/`.
19. Jankun-Kelly, T., Kreylos, O., Ma, K.L., Hamann, B., Joy, K.I., Shalf, J., Bethel, E.W.: Deploying web-based visual exploration tools on the grid. IEEE Computer Graphics and Applications **23**(2) (2003) 40–50
20. Grimstead, I.J., Avis, N.J., Walker, D.W.: Automatic distribution of rendering workloads in a grid enabled collaborative visualization environment. In: Proc. SC'04. (2004) 10 pages (CD-ROM).
21. Fan, Z., Qiu, F., Kaufman, A., Yoakum-Stover, S.: GPU cluster for high performance computing. In: Proc. SC'04. (2004) 12 pages (CD-ROM).
22. Anderson, D.P.: BOINC: A system for public-resource computing and storage. In: Proc. 5th IEEE/ACM Int'l Conf. Grid Computing (GRID'04). (2004) 4–10
23. Sullivan, W.T., Werthimer, D., Bowyer, S., Cobb, J., Gedye, D., Anderson, D.: A new major SETI project based on project serendip data and 100,000 personal computers. In: Proc. 5th Int'l Conf. Bioastronomy. (1997) 729