A Dynamic Offload Scheduler for spatial multitasking on Intel Xeon Phi Coprocessor

Takamichi Miyamoto, Kazuhisa Ishizaka, Takeo Hosomi

Green Platform Research Laboratories
NEC Corporation
Shimonumabe 1753, Kawasaki, Kanagawa, JAPAN
{t-miyamoto@ap, k-ishizaka@ay, hosomi@ah}.jp.nec.com

Abstract— Intel Xeon Phi Coprocessor appears and it fully supports multitasking, but it does not automatically ensure high performance in this case. A conventional task level resource allocation scheduler could be used, but a processor utilization of the Xeon Phi is low because of idle time on the Xeon Phi. In this paper, we propose a dynamic offload scheduler which assigns processor resources of the Xeon Phi to tasks by an offload level. We describe an effectiveness of the proposed method with evaluations.

I. Introduction

Many-core accelerators have become widely used as a powerful and power efficient computing coprocessor for high performance systems. A range of applications, from scientific computing to multimedia, are well suited for the many-core accelerators to achieve large speedups [1, 2]. However, General-Purpose Computing on Graphics Processing Unit (GPGPUs) [3, 4], which were the dominant many-core accelerators, have very limited support for multitasking. Few works have been done for multitasking on many-core accelerators [5].

Intel's many-core accelerator, Xeon Phi Coprocessor (Xeon Phi) has appeared [4, 6, 7]. It supports x86 ISA and Linux OS just like commodity Xeon processors. Thus the Xeon Phi fully supports multitasking. Unfortunately, it does not automatically ensure that multiple tasks will run on the Xeon Phi with high performance. In this paper, we discuss the performance of the Xeon Phi in multitasking environments.

In order to use the Xeon Phi, programmers write their programs with "offload model". The main part of the code runs on the host processor. The regions specified to be offloaded by pragmas runs on the Xeon Phi. The offload model enables to utilize the strong points of the host processor and the Xeon Phi. The host processor is suitable to execute a sequential part, since its single-threaded performance is higher than that of the Xeon Phi. The Xeon Phi is suitable to execute a parallelized part, since its multi-threaded performance is higher than that of the host processor.

For sharing the Xeon Phi by multiple tasks which are written with the offload model, a task level scheduler could be used. The scheduler manages the Xeon Phi's cores as a resource, and allocates parts of the resources to tasks when the tasks are invoked. The resources are spatially partitioned and assigned to tasks while they run. This method will work well when the tasks run only at the Xeon Phi. However, a low resource utilization issue could happen when the tasks are written with the offload model. Some regions of the tasks run only at the host processor and the assigned Xeon Phis resource are not used. Therefore, new scheduling method will be required to fully utilize the Xeon Phi's resource for the offload tasks in the multitasking environments.

The contributions of this paper are as follows:

- We propose a dynamic offload scheduler which assigns processor resources of the Xeon Phi to tasks by an offload level. It enables to allocate the resources for the offload regions that run on the Xeon Phi, and not to the other regions that run on the host processor. Hence, the proposed scheduler can efficiently utilize the resource.

- We show that the proposed scheduler works well across our workloads and obtains a maximum of 1.62 times performance against the task level scheduler.

The rest of this paper is structured as follows. Section II describes background and motivation in this paper. Section III highlights our proposed dynamic offload scheduler. Section IV validates the dynamic offload scheduler by an evaluation through our workloads. Section V describes the related work. Finally, Section VI concludes this work.

II. Background and Motivation

This section describes hardware specs and software environment of the Xeon Phi, and a problem on the Xeon Phi shared by multiple offload model tasks (offload tasks).

A. Intel Xeon Phi Coprocessor

The Xeon Phi 5110P has 60 x86 processor cores. Each processor core supports four hardware threads. The core has 512-bit vector unit and 512KB L2 cache. The core is clocked at 1.053GHz. The core uses in-order execution. A peak single-precision performance of the Xeon Phi is more than 2 TFLOPS. Compared these specs with Xeon processor (Xeon) which is usually adopted as a host processor, a single-threaded performance of the Xeon Phi is lower than that of the Xeon because the latest Xeon is clocked at more than 2.0 GHz and it uses out-of-order execution. However, a multi-threaded performance of the Xeon Phi is higher than that of the Xeon by using thread parallelism of programs.

As a software environment, Manycore Platform Software Stack, called MPSS, is provided by Intel. In this environment, Linux kernel runs on the Xeon Phi. Hence, it is easy to run multiple tasks on the Xeon Phi.

B. Multitasking for Offload Model

In order to use the Xeon Phi, programmers write their programs with "offload model". Figure 1 illustrates an example code of an offload program. The code includes an offload part to the Xeon Phi. First, the program is executed on the host processor. Next, it is executed on the Xeon Phi. Finally, it is executed on the host processor. The offload part is indicated with a directive as "#pragma offload target (mic)". In the offload part, a loop is parallelized by OpenMP for example, so that the offload part is executed in parallel on the Xeon Phi.

When multiple offload tasks share the Xeon Phi, we can use a conventional task level scheduler for utilizing the Xeon Phi. The task level scheduler statically splits processor resources of the Xeon Phi into the number of tasks. And, it allocates each split processor resource to each task. For an example which the task level scheduler manages two tasks, it statically splits 59 cores into 30 cores and 29 cores, and it statically allocates 30 cores to one task and allocates 29 cores to the other. Figure 2 illustrates a problem of the task level scheduler. The task level scheduler allocates each split processor resource of the Xeon Phi to each task while each task runs. It causes idle time on the Xeon Phi because an execution time of the task includes a host processor execution time. Hence, it has a problem which a processor utilization of the Xeon Phi is low.

There is a method to minimize the idle time on the Xeon Phi with a programming effort in the offload task. The programmer makes the offload region asynchronous,

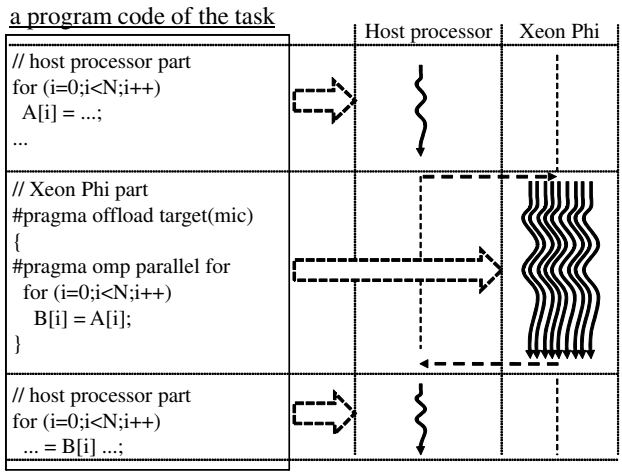


Fig. 1. An example of offloading program code for the Xeon Phi.

and overlaps the execution of the host part and the Xeon Phi part. However, the method is not applicable to applications which have a data dependency between the host part and the Xeon Phi part. Moreover, the method cant not utilize the Xeon Phi resources for applications whose execution time of the host part is much longer than taht of the Xeon Phi part.

III. Proposed Method

This section describes our proposed method for sharing the Xeon Phi by multiple offload tasks. The proposed method "dynamic offload scheduler" is designed to improve a processor utilization of the Xeon Phi. It runs on the host processor. It cooperates with multiple tasks. It dynamically allocates processor resources of the Xeon Phi to each task per offload, so that each task occupies a part of the processor resources only while each offload runs. It minimizes idle time of the Xeon Phi.

Figure 3 illustrates a program code of a task which cooperates with the dynamic offload scheduler. The task has two kinds of Application Program Interfaces (APIs) to the dynamic offload scheduler. It calls $AcquireProcessorResource()$ API to acquire a part of the processor resources of the Xeon Phi before the offload part is executed. It calls $ReleaseProcessorResources()$ API to release the acquired part of the processor resources after the offload part is executed. Each task is able to cooperate with the dynamic offload scheduler by calling these APIs.

Figure 4 shows a flowchart of the dynamic offload scheduler which cooperates with the task. The flowchart of the dynamic offload scheduler consists of four steps. Through step S1 to step S3, the dynamic offload scheduler allocates a part of idle processor resources to the task when the task acquires processor resources. The dynamic of-

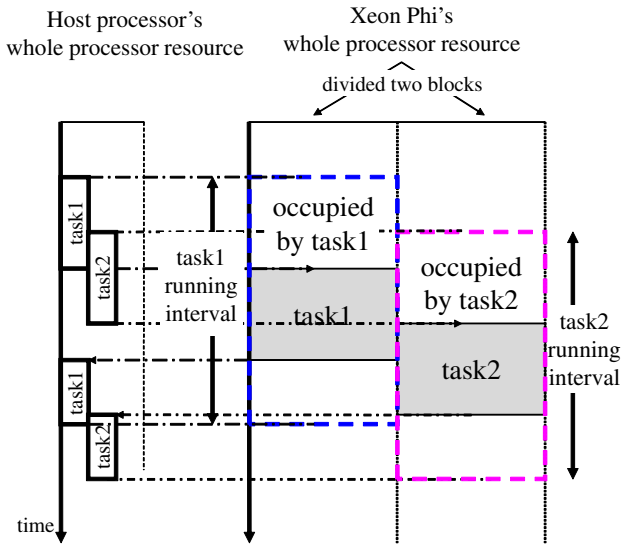


Fig. 2. Low processor utilization problem scheduled with partitioning resources per task statically.

```
1  // host processor part
2  for (i=0;i<N;i++)
3      A[i] = ...;
4
5  AcquireProcessorResource();
6
7  // Xeon Phi part
8  #pragma offload target(mic)
9  {
10 #pragma omp parallel for
11     for (i=0;i<N;i++)
12         B[i] = A[i];
13 }
14
15 ReleaseProcessorResource();
16
17 // host processor part
18 for (i=0;i<N;i++)
19     ... = B[i];
```

Fig. 3. An example of an offloading program code which cooperates with the dynamic offload scheduler.

fload scheduler is also able to give thread affinity to the task. And the task is able to set the thread affinity for the offload. On step S4, the dynamic offload scheduler marks idle processor resources when the task releases the acquired processor resources. After that, the idle processor resources can be allocated to the other task. Therefore, the dynamic offload scheduler is able to control a processor resource allocation per offload.

These steps are described as follows.

S1 The dynamic offload scheduler searches *idle* processor resources from the available processor resource table on demand from each task's acquirement API. If it is able to find them, it goes step S2. If it is not able to find them, it repeatedly searches again.

S2 The dynamic offload scheduler changes states of the processor resources into *busy* in the available processor resource table.

S3 The dynamic offload scheduler returns the processor resources to the task. Furthermore, it is able to return a thread affinity too.

S4 The dynamic offload scheduler changes states of processor resources into *idle* in the available processor resource table on demand from each task's release API.

Figure 5 illustrates that the proposed method improves a processor utilization of the Xeon Phi against the task level scheduler. Figure 5(a) illustrates a processor utilization of the Xeon Phi by the task level scheduler. Figure 5(b) illustrates a processor utilization of the Xeon Phi by the proposed method. In Figure 5(a), the task level scheduler allocates the processor resources to each task while each task runs, so that there are idle processor resources. In Figure 5(b), the proposed method allocates the processor resources to each task while each offload runs, so that it is able to minimize the idle processor resources. The proposed method increases the number of tasks which is able to share the Xeon Phi. Thus, the proposed method is able to improve the processor utilization of the Xeon Phi against the task level scheduler.

IV. Evaluations

This section describes evaluation methodology and the evaluation result.

A. Methodology

We create micro benchmarks that represent the offload task and enable to focus the goal of evaluating performance improvement by our proposed scheduler. The task repeatedly executes a host part and an offload part. Execution times of the host part and the offload part are the same. There are no overlaps between two executions. The host part just waits for offload executions with sleep() call. We use two kernels, sGEMM and sGESV, from Intel Math Kernel Library[8] for the offload part. Both kernels conduct matrix operations. Matrix sizes of them are 4096x4096. The offload part runs on the Xeon Phi with 32 threads. Utilizing processor resources of the Xeon Phi increases the performance. However, CPU overutilization like thread oversubscription incurs performance down because it leads context switches and cache misses. Thus, we measure the total performance of multiple tasks in order to considering CPU overutilization. The performance of each task is measured by FLOPS, where execution time

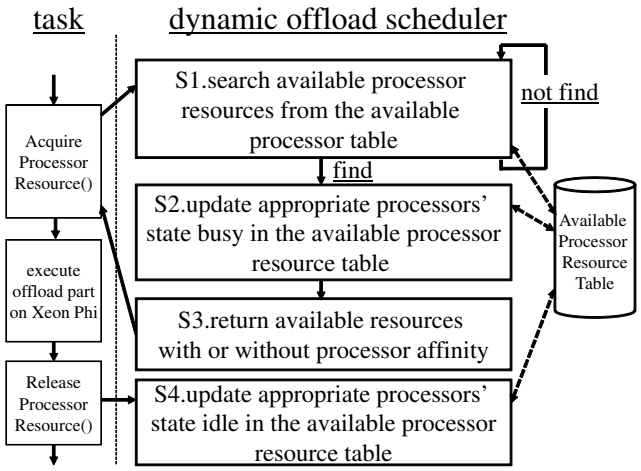


Fig. 4. A flowchart of the dynamic offload scheduler which cooperates with a task.

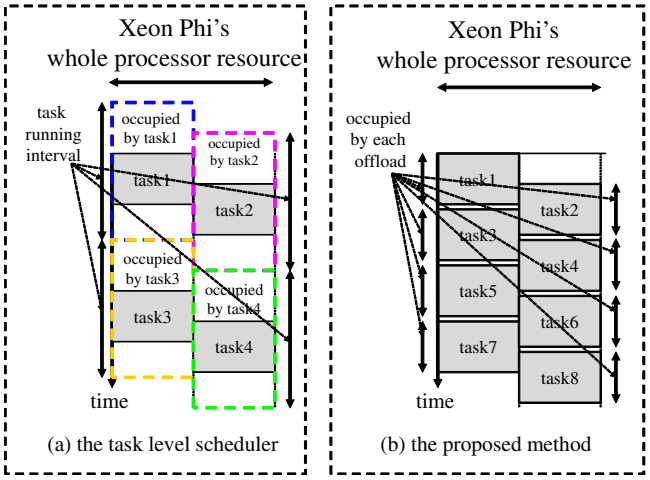


Fig. 5. A comparison between the task level scheduler and the dynamic offload scheduler for a processor utilization of Xeon Phi.

includes both the host part and the offload part, and the number of floating operations at the kernel, sGEMM and sGESV. The performance of the system can be calculated by summing the FLOPS of every running task.

We evaluate two conventional methods, "task level scheduler" and "Linux scheduler", and three proposed methods, "proposed(none)", "proposed(scatter)" and "proposed(compact)". Table I summarizes the difference of these five methods.

- "task level scheduler" : This method assigns spatially partitioned resources to tasks. This method divides Xeon Phi's 59 cores to seven 8-cores, and one 3-cores. Therefore, we run eight tasks, where seven tasks run with 32 threads and one with 12 threads.
- "Linux scheduler" : This method leaves threads scheduling to Linux OS on the Xeon Phi. Since the Linux OS can handle multitasking and has a threads scheduling mechanism, it is possible to run many offloads without any scheduling middleware. We run 15 tasks, and each tasks run with 32 threads.
- "proposed(none)" : The dynamic offload scheduler only manages the number of threads executed on the Xeon Phi in order not to oversubscribe threads more than the hardware threads. It still leaves the thread scheduling to the Linux OS on the Xeon Phi. We run 15 tasks.
- "proposed(scatter)" : The dynamic offload scheduler manages both the number of threads and the thread scheduling. In addition to the "proposed(none)", the scheduler assigns appropriate hardware threads for offload. The scheduler uses *scatter* affinity. Thus, the scheduler avoids more than one thread run on the same hardware thread. We run 15 tasks.
- "proposed(compact)" : The dynamic offload scheduler manages both the number of threads and the thread scheduling. Unlike "proposed(scatter)", this method uses *compact* affinity. We run 15 tasks.

Figure 6 illustrates how to allocate tasks on a processor by each method. In this example, the processor has two cores, and each core supports two hardware threads. Each task uses two threads. The "task level scheduler" and "proposed (compact)" allocate hardware thread 0 and hardware thread 1 to a task, and allocate hardware thread 2 and hardware thread 3 to the other. The *compact* affinity sets thread $\langle n \rangle + 1$ of a task to a free thread context as close as possible to the thread context where the $\langle n \rangle$ thread of a task was placed. The "Linux scheduler" and "proposed(none)" allocate all hardware threads to all tasks without an affinity setting. Thus, Linux kernel dynamically allocates each execution thread onto hardware thread. The "proposed(scatter)" allocates hardware thread 0 and hardware thread 2 to a task, and allocates hardware thread 1 and hardware thread 3 to the other. The *scatter* affinity distributes the threads of a task as evenly as possible across the entire system.

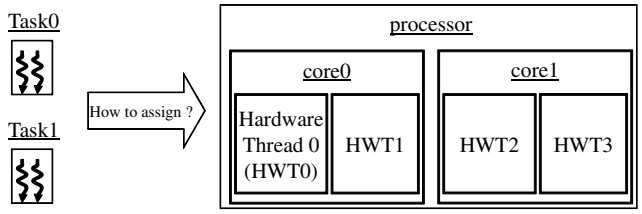
B. Results

Figure 7 illustrates evaluation results. X-axis shows sGEMM and sGESV. Y-axis shows a normalized performance against that of "task level scheduler". Bars show "task level scheduler", "Linux scheduler", "proposed(none)", "proposed(scatter)" and "proposed(compact)" from left to right.

In sGEMM and sGESV, our proposed method "proposed(compact)" shows 1.54 and 1.62 times performance improvements compared to the "task level scheduler". It

TABLE I
EVALUATED SCHEDULING METHODS.

	Resource assignments level	Thread oversubscribe control	Thread affinity	# of tasks
task level scheduler	Task level	yes	Compact	8
Linux scheduler	Offload level	no	None	15
proposed (none)	Offload level	yes	None	15
proposed (scatter)	Offload level	yes	Scatter	15
proposed (compact)	Offload level	yes	Compact	15



	Task0	Task1
task level scheduler	HWT0, HWT1	HWT2, HWT3
Linux scheduler	HWT0-3	
proposed (none)	HWT0-3	
proposed (scatter)	HWT0, HWT2	HWT1, HWT3
proposed (compact)	HWT0, HWT1	HWT2, HWT3

Fig. 6. How to execute two tasks which use two threads on two processor cores which support two hardware threads by each method.

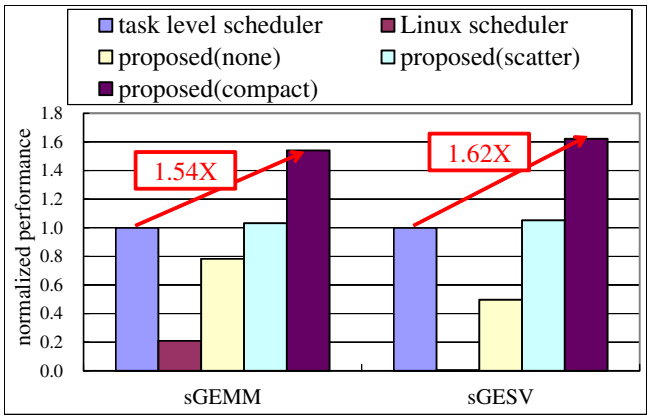


Fig. 7. Evaluation result using sGEMM and sGESV.

clearly shows that our proposed method improves the utilization of the Xeon Phi's resource.

On "Linux scheduler", huge performance drops happens. Since it does not control the number of total threads, it causes the thread oversubscription; the number of invoked threads exceeds the number of hardware threads. Such oversubscription results in context switches and cache conflicts, which are more expensive on the Xeon Phi than Xeon. We can avoid this oversubscription by "proposed(none)", and it shows some performance improvement. From these results, it becomes clear that avoiding the thread oversubscription is important on the Xeon Phi.

There are performance gaps among "proposed(none)", "proposed(scatter)" and "proposed(compact)". The difference among those three is thread affinity settings. On "proposed(none)", the dynamic offload scheduler does not set thread affinity. The Linux OS on the Xeon Phi allocates offload threads to the hardware threads. On the other hand, the dynamic offload scheduler specifies the thread affinity on "proposed(scatter)" and "proposed(compact)". Since the performance of "proposed(none)" is lower than those of other two, we think reducing an overhead of OS scheduling by thread affinity is also important on the Xeon Phi.

Moreover, "proposed(compact)" shows better performance than "proposed(scatter)". When using *scatter*, four threads from different tasks run on the same core, and it causes the cache conflicts. On the other hand, four threads from the same task run on the same core when using *compact*, and it reduces such cache conflicts and improves cache efficiency. Hence, we think it is important to set thread affinity that fits to the multitasking environments.

From the above discussions, we can conclude that the dynamic offload scheduler with "proposed(compact)" works better than the task level scheduler because the former can pack more offload processing from multiple tasks to the Xeon Phi, and improve the performance. It also works better than the scheduler of Linux OS on the Xeon Phi because of following three reasons; 1) it prevents the thread oversubscription, 2) reduces the overhead of OS scheduling, and 3) set the suitable affinity for multitasking.

V. RELATED WORKS

Multitasking on GPGPU has been studied [5, 9, 10, 11]. Peters and et.al [5] proposes a method to realize sharing Tesla generations GPGPU among multiple tasks by way of running one kernel included some functions on

GPGPU and transferring data for each function. Sun and et.al [9] proposes a sharing GPGPU method by way of merging different data from some same programs and offloading process with merged some different data. Li and et.al [10] proposes a sharing GPGPU method by way of merging some kernels of multiple tasks with considering I/O, computations and data transfer information. Ino and et.al [11] proposes a cooperative multitasking method by way of dividing offloading task into small pieces. These studies realize sharing GPGPU with one kernel by way of merging data or kernels, and share GPGPU by time-sharing. Thus, these studies do not present to schedule multiple offloads spatially.

Resource management in multiple tasks environment is studied [12, 13]. Mok and et.al [12] proposes a scheduling method by way of using two partitioning methods and scheduling tasks into these partitions. Adriaens and et.al [13] proposes a scheduling kernel on Streaming Processors (SMs) of GPGPU by way of distributing SMs statically and scheduling a kernel on distributed SM dynamically with considering profile results of each application. These studies targets to schedule tasks on statically distributed resources, so that these are different from our dynamically distributing and dynamically scheduling method.

VI. Conclusions

In this paper, we propose a dynamic offload scheduler which assigns processor resources of the Xeon Phi to tasks by an offload level. By preventing the thread oversubscription and settings the appropriate thread affinity, the dynamic offload scheduler works better than the task level scheduler because the former can pack more offload processing from multiple tasks to the Xeon Phi, and improve the performance. It also works better than the scheduler of Linux OS on the Xeon Phi because of following three reasons; 1) it prevents threads oversubscription, 2) reduces the overhead of OS scheduling, and 3) set the suitable affinity for multitasking.

References

[1] Gang Chen, Guobo Li, Songwen Pei, and Baifeng Wu. High performance computing via a gpu. In *Information Science and Engineering (ICISE), 2009 1st International Conference on*, pages 238 –241, dec. 2009.

[2] Mian Lu, Jiuxin Zhao, Qiong Luo, Bingqiang Wang, Shaohua Fu, and Zhe Lin. Gsnp: A dna single-nucleotide polymorphism detection system with gpu acceleration. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 592 –601, sept. 2011.

[3] NVIDIA Corp. Kepler gk110 white paper. http://www.nvidia.com/content/PDF/ kepler/NVIDIA-Kepler-GK110- Architecture-Whitepaper.pdf.

[4] A. Heinecke, M. Klemm, and H. Bungartz. From gpgpu to many-core: Nvidia fermi and intel many integrated core architecture. *Computing in Science Engineering*, 14(2):78 –83, march-april 2012.

[5] H. Peters, M. Koper, and N. Luttenberger. Efficiently using a cuda-enabled gpu as shared resource. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 1122–1127, 2010.

[6] Intel Corp. Intel many integrated core architecture. http://www.intel.com/ content/ www/ us/ en/ architecture-and- technology/ many- integrated- core/ intel- many-integrated- core- architecture.html.

[7] Intel. The intel®xeon phi^{TM}product family. http://www.intel.com/ content/ dam/ www/ public /us/ en/ documents/ product- briefs/ high- performance- xeon-phi- coprocessor- brief.pdf.

[8] Intel®math kernel library. http://software. intel. com/ en-us/ sites/ default/ files/ Intel_ Math_ Kernel_ Library_ v12_ PB-1.pdf.

[9] Siqi Sun, Zhuo Zhang, Liang Wang, Wenfeng Shen, Weimin Xu, and Yanheng Zheng. A study of the single-program multiple-task model on gpu computing. In *Automatic Control and Artificial Intelligence (ACAI 2012), International Conference on*, pages 296–300, 2012.

[10] Teng Li, V.K. Narayana, E. El-Araby, and T. El-Ghazawi. Gpu resource sharing and virtualization on high performance computing systems. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 733–742, 2011.

[11] F. Ino, A. Ogita, K. Oita, and K. Hagihara. Cooperative multitasking for gpu-accelerated grid systems. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 774–779, 2010.

[12] A.K. Mok, Xiang Feng, and Deji Chen. Resource partition for real-time systems. In *Real-Time Technology and Applications Symposium, 2001. Proceedings. Seventh IEEE*, pages 75–84, 2001.

[13] J.T. Adriaens, K. Compton, Nam Sung Kim, and M.J. Schulte. The case for gpgpu spatial multitasking. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1 –12, feb. 2012.