Cluster Comput (2016) 19:2241–2250
DOI 10.1007/s10586-016-0656-8

A memory-driven scheduling scheme and optimization for concurrent execution in GPU

Bao-yu Xu[1] · **Wu Zhang**[1] · **Xian-he Sun**[2] · **Yang Wang**[1]

Received: 24 May 2016 / Revised: 16 August 2016 / Accepted: 17 September 2016 / Published online: 29 September 2016

Abstract Concurrent execution of GPU tasks is available in modern GPU device. However, limited device memory is an obvious bottleneck in executing many GPU tasks. And the task priority and system performance are often ignored. To address these, a real-time GPU scheduling scheme is proposed in this paper. A reservation algorithm based on device memory(RBDM) is adopted to provide more opportunity for the High-priority task in the scheme. high priority first wake (HPFW) and small memory HPFW (SM-HPFW) are employed in the scheduling of waiting tasks to improve the priority response time and system performance. A CPU-based monitor is developed to check the GPU task execution. Experiments show the RBDM can work effectively. Compared with FIFO, HPFW can decrease overall priority response time significantly. Overall task completion time can be reduced by 20 % using the SM-HPFW while the distribution of device memory requirement of GPU tasks is even.

Keywords GPU · Device memory · Reservation · Schedule

1 Introduction

Graphics Processing Unit (GPU) is often applied in general purpose computing such as no-graphics and scientific computing because it is extreme-scale, cost-effective and power-efficient. To effectively utilize the resources supplied by the GPU, the tasks which originate from a single application or/and many independent applications [15] can be executed concurrently in the mainstream GPU cards such as Nvidia kepler GPU.

GPU is a PCI device and has fixed memory, which is called Device memory(DM) in Nvidia product. While handling many tasks in GPU, the DM often becomes a bottleneck due to a mass of data. The bottleneck usually includes two aspects [13]. First, CPU and GPU have separate memories. The data required in GPU execution must be transferred between CPU and GPU memory, but the DM is limited in size. Second, the PCI-e link that connects the two memories has limited bandwidth.

To address above problems, many applications need to be optimized through data compression [17] and data mining [4], etc. Additional cache that comes from the host is supplied for asynchronous data copy between host and device to solve the lack of device memory [13]. In addition, some application-aware algorithms, such as FR-RR-FCFS [7], are used to control the execution of task. FR-RR-FCFS is adopted to optimize the I/O bottleneck of GPU, which has been realized in GPU simulator. Remarkably, these methods are tailored to be suited for some special applications. First come first serve (FCFS) is employed to schedule task in most of GPU cards. First-ready FCFS (FR-FCFS) [18] is certificated effective in improving the internal bandwidth. C-AMAT [20] is an accurate metric for analysis of memory wall in concurrent memory systems. These methods are suitable for real-time system since they ignore the implementation details of application.

Many tasks often must be executed in priority order because they potentially come from different users, which has been presented in existing GPU scheduling system. For example, Timegraph [9] is a time-based scheduling system, which allocates the executing time of GPU to each task

✉ Bao-yu Xu
xbybao@163.com

1 School of Computer Engineering and Science, Shanghai University, Shanghai, China

2 Department of Computer Science, Illinois Institute of Technology, Chicago, IL, USA

depending on their priority. However, these tasks can only be executed asynchronously in GPU. RCSM [8] is a core-based scheduling architecture and allows multiple tasks to execute simultaneously. Experimental result proves the RCSM validity in the mobile device. Timegraph and RCSM are all designed in operation-system level and to allocate limited resource reasonably. Executing time is restrained for single-task GPU in Timegraph and Core number is less in the mobile device in RCSM, so the DM is not considered in their design.

For DM importance mentioned above, this paper focuses on the DM of GPU and try to design memory-driven scheduling scheme, specific to multi-task GPU, to improve the execution performance . However, obtaining DM cost accurately in executing a task is difficult because some DM is used in extra operations such as thread synchronization. Though a few tools, for example *nvprof*, can check the size of DM used in the GPU task, extra memory overhead will be produced if these tools are launched. Additionally, since DM is limited, high-priority task may lose execution chance once the DM is all partitioned to low-priority task physically, which has occurred in some other real-time scheduling systems [11]. Remarkably, some existing GPU scheduling systems pay attention to priority response time but neglect entire system performance, such as RCSM. This paper makes the following contributions to address these problems.

First, combining the developed monitor, a modified reservation algorithm based on device memory is employed in the multi-task scheduling of GPU. Logical DM partition is employed in the algorithm according to the priority requirement of GPU applications. The monitor only runs in CPU. The DM cost is predicted by measuring the size of defined DM in the task code. Second, for the waiting tasks, HPFW is adopted to increase the priority reponse chance for high-priority task and SM-HPFW (that is the improving HPFW) can decrease the overall task completion time to improve system performance.

The rest of this paper is organized as follows. Section 2 gives a background including GPU and real-time scheduling. Section 3 introduces how to realize DM estimation and GPU monitor. In Sect. 4, we display the multi-tasks scheduling scheme based on DM, including scheduling architecture, reserve algorithm and wakeup policy. Performance metric and experiments are presented in Sects. 5 and 6. We close this paper with the conclusion.

2 Background

Multi-task GPU which can execute multiple tasks concurrently, such as NVIDIA K20 GPU, is adopted as experiment platform. The GPU application is developed with CUDA [16] that allows developer to use C as a high level programming language. Next, we will provide a brief description for them and display the relevant works in scheduling.

A GPU [16] is built around an array of Streaming Multiprocessors (SMs) that are designed to execute hundreds of threads concurrently. The threads may access data from multiple memory spaces during their execution. The CUDA programming model assumes a system composed of a host and a device, each with their own separate memory. Kernels operate out of device memory, so the runtime provides functions to allocate, de-allocate, and copy, as well as transfer data between host memory and device memory. Device memory can be allocated using function malloc such as cudaMalloc. Multi-task technology has been proposed in Fermi architecture and improved in Kepler architecture, as shown in the Fermi and Kepler whitepaper [14,15].

Operation-system level resource allocation and multi-task scheduling is, in essence, defining a scheduling policy consisting in creating a set of tasks and associating them with different resource [1]. For example, a simple FIFO is to array the tasks depending on their orders of entering scheduling system, and preferentially satisfying the resource demand of task of head of array.

Reservation [3,10] has been used to provide guaranteed and predictable memory access performance to real-time tasks. It allows a task to reserve a portion of the total system resource for its exclusive use. Reservation has also been used in scheduling based on priority, and the reserved object is the limited resource provided by system. The considerable parameters include the size of required resource and allocation operation. In Timegraph and RCSM, the reserved resource can never be stolen by other task to maintain the temporal isolation for each task, called physical reservation, so that the interference among tasks is decreased. Some reserve techniques are applied to partition the resource logically [5], called the logical reservation. In general, logical reservation needs a mechanism to avoid the interference.

3 DM estimation and GPU monitor

The DM costed in executing a task and the rest of GPU DM are prerequisite before scheduling GPU task, but it is difficult to acquire them by checking the GPU device because other existing DM overhead. Therefore, DM estimation is adopted in our design and a monitor is developed to detect GPU execution process.

3.1 DM estimation

In CUDA, computing data for the task must be sent to the GPUs DM from the host memory, through the PCI-E bus prior to invoking the task. Output data should be written to the GPUs DM first, then returned to the host memory. All DM used by the task should be pre-allocated [6].

CUDA has tailored special functions to allocate GPU DM to task. These functions include cudaMalloc, cudaMalloc3D, cudaMallocArray, etc. Keywords of these functions, such as cuda and Malloc, and corresponding parameters can be used to look for correct function and measure the allocated DM size. For example, the second parameter in function cudaMalloc defines the size of allocated DM, as shown in the following CUDA code.

```
#include lib.h
type kernel_name(parameter set)
{
      ...
 // Allocate host memory
 unsigned int mesize = com_data_size;
 float *h = (float *) malloc(memsize);
 // Allocate device memory
 cudaMalloc((void **) &d, memsize);
 // copy host memory to device
 cudaMemcpy(d, h, memsize, cudaMemcpyHostToDevice);
 // execution in GPU
    ...
 // Copy result from device to host
 cudaMemcpy(h, d, memsize, cudaMemcpyDeviceToHost);
 // Clean up memory
 free(h);
 cudaFree(d);
 cudaDeviceReset();
    ...
}
int main(int argc, char **argv)
{...}
```

The code presents some basic operations of GPU application, including DM allocation and data copy between host and device, which can be realized by special functions with parameter *memsize*. The value of *memsize* defines the requirement of DM in executing process for a GPU task. In our system, a developed script need to read the value of *memsize* existing in the source code while the scheduling system receives a execution request, so the source-code file which has the postfix *.cu* is necessary.

3.2 GPU monitor

In DM-based scheduling system, the rest of DM should be obtained before scheduling a task. Even though the return value of a program can discover whether the DM allocation is successful or not, developers tend to forget to add the return value in code. Some specific tools supplied by vendor can get the detail, but the tools are invisible to user and the additional DM overload is made as well if they are activated.

Considering GPU task is initiated by CPU process, we can save DM to trace the GPU task by detecting the related CPU process. In addition, Some execution imformation available in device file, such as the linux file $/dev/nividia$, can be captured in executing a GPU task. In our system, a GPU monitor is realized by the linux script according to above methods. As shown in the following code, the process ID can be obtained if a GPU task is submitted successfully and these codes only run in the host and acquire the return signal from CPU process.

```
# detect DM allocation
submit task &
my_pid=$!
count=0
sleep 10
if ps | grep "$my_pid"
then
count=0
else
count=1
fi
```

At present, the monitor is only used to trace the allocation process of DM. If the required DM is allocated for a task, the task is submitted successfully. In future work, more information will be captured from the device file to strengthen the monitor.

4 Multi-task scheduling based on DM

4.1 scheduling architecture overview

For the multi-task scheduling, the base architecture is composed of ready queue, waiting queue, shceduler and some policies designed for submission and wakeup, etc. In our work, the scheduling scheme is used in general-purpose GPU(GPGPU) computing, such as scientific application. Figure 1 shows the multi-task GPU scheduling architecture.

Interrupt may generate some wrong results because of some inconspicuous reasons, which should be avoid for the long-time GPGPU computing if possible. In addition, if interrupt is frequent, the GPU workload increases rapidly due to the data transmission and thread synchronization. Therefore, the interrupt is not considered in the architecture now.

The size of DM needed by each task can be obtained before the task is submitted by DM estimation, which is realized in the task analysis. While the task is submitted, the GPU monitor acquires a return signal to check the execution status of task and decides what to do next.

Scheduler adopts a scheduling policy depending on DM reservation to schedule the tasks existing in the ready queue. To improve the priority reponse time of task and system

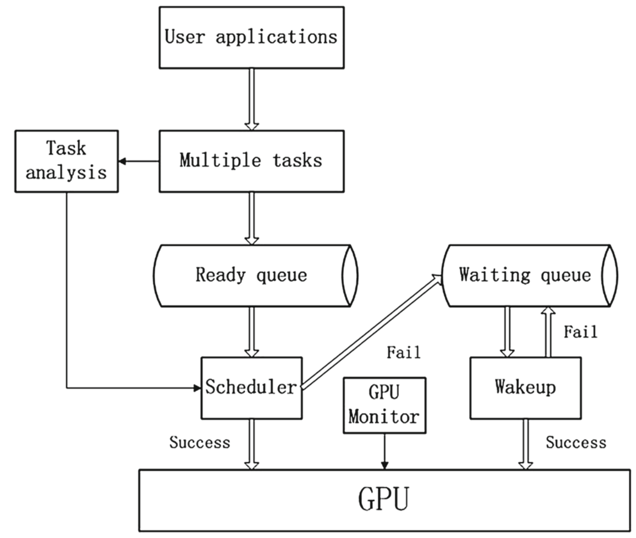

Fig. 1 Multi-task scheduling architecture

performance, two wakeup policies are proposed in Wakeup module. The details of these policies will be illuminated in the following subsection.

4.2 Scheduling plicy: RBDM

In RBDM algorithm, GPU DM is the reserved object and is allocated logically for each task. The reservation process is composed of several reservation operations, called RO, which is responsible for allocating specified amount of DM for the task which lies in the ready queue. The amount of DM of each task depends on its priority, some definitions are shown next.
Definition the reserved DM of task i, called RM_i, is a digital value which indicates the amount of DM that is assigned to task i in each reserve operation. RM_i can be obtained through Eq. 1, in which P_i is the quantified priority of task i, and M_i is the estimated DM size of task i.

$$RM_i = \frac{P_i M_i}{P_1 + P_2 + \cdots + P_n} \tag{1}$$

M_i can predict the required DM in execution, though the latter is often much bigger than the former in execution process. According to the value of RM_i, we can get how many reserve operations, called RO_i, can complete the entire reserve process for task i. How to get the RO_i is shown below.

$$RO_i = \frac{M_i}{RM_i} = \frac{P_1 + P_2 + \cdots + P_n}{P_i} \tag{2}$$

In Eqs. 1 and 2, the priorities can be quantified by any positive value, as long as the higher priority is quantified to the larger value. Suppose there are high priority, middle priority and low priority, they can be represented with integers 3, 2 and 1, respectively. Table 1 shows RO of tasks with different priorities.

Table 1 RO for different priority

Priority	RO
High	2
Middle	3
Low	6

Since the actual remaining size of GPU DM is unknown, the scheduling algorithm must combine with the GPU monitor to detect whether the scheduled task is executed successfully. the RBDM is stated in Algorithm 1.

Algorithm 1 RBDM

```
1: procedure R_A($\&Task_i, M_i, RM_i, cycle, \&w\_num$)
2:     $budget \leftarrow 0$
3:     $res\_signal \leftarrow fail$
4:     $sub\_signal \leftarrow fail$
5:     $time \leftarrow system\_time$
6:     while $budget < M_i$ do
7:         $budget \leftarrow budget + RM_i$
8:         if $budget < M_i$ then
9:             $time + cycle$                    ▷ reserved synchronization
10:        end if
11:        $res\_signal \leftarrow true$
12:    end while
13:    if $w\_num = 0$ then
14:        if $res\_signal = true$ then
15:            submit $Task_i$
16:            $sub\_signal \leftarrow$ monitor($Task_i$)
17:            if $sub\_signal = fail$ then       ▷ submission to fail
18:                $w\_num \leftarrow$ waitqueue($Task_i$)
19:            end if
20:        end if
21:    else
22:        $w\_num \leftarrow$ waitqueue($task$)
23:    end if
24:    return $sub\_signal$
25: end procedure
```

In Algorithm 1, the res_signal and sub_signal record whether the reservation and submittion are successful or not. The w_num indicates the number of waiting tasks. The $cycle$ is the time constant which is equal to a RO period. In a period of $cycle$, task i is allocated the specifed RM_i once. Each task is submitted if it acquires enough logical DM or sent to waiting queue otherwise.

By the algorithm, these tasks may be executed concurrently, but submitted orderly due to only one scheduling managerment. Especially, FIFO is only applied to schedule the waiting task (that is, the head of task in waiting queue is submitted first) to test the RBDM. FIFO focuses on the order of time but not on the demand of application. The scheduling improvement in waiting queue will be discussed below.

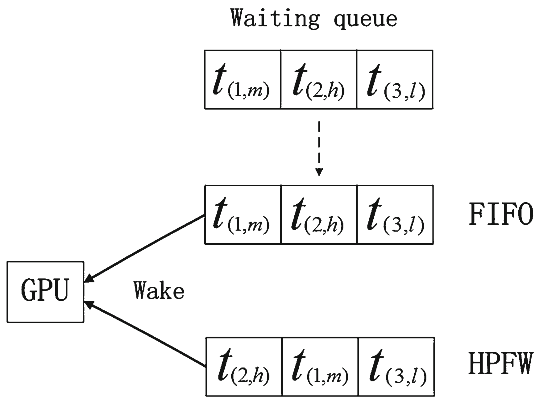
Waiting queue
$t_{(1,m)}$ $t_{(2,h)}$ $t_{(3,l)}$
FIFO
GPU
Wake
$t_{(2,h)}$ $t_{(1,m)}$ $t_{(3,l)}$
HPFW

Fig. 2 The task wakeup order in waiting queue by FIFO and HPFW

4.3 Scheduling improvement

The task with higher priority may not be submitted earlier by RBDM. A usual case is the GPU DM has been exhausted while the high-priority task is submitted. The unexecuted task will be pushed into waiting queue in disregard of its priority and often be awaken by the FIFO policy, i.e., the first entering task has the highest priority. In fact, the priorities of all tasks existing in waiting queue are reset by FIFO so that they can not be processed in line with their original priorities.

4.3.1 HPFW

By HPFW, the tasks that lie in waiting queue are submitted again according to their priorities from high to low. If the former is fail in submission, the rest will wait even if the unused DM is enough for one of them. Figure 2 shows the difference between FIFO and HPFW.

In Fig. 2, there are three tasks with respective priorities labeled as h, m and l, indicating high priority, middle priority and low priority, respectively. The order of entering the waiting queue is indicated by digital subscript. HPFW schedules the waiting tasks in the priority order($t_{(2,\mathbf{h})}$,$t_{(1,\mathbf{m})}$,$t_{(3,\mathbf{l})}$), compared with FIFO.

No matter FIFO or HPFW is adopted as scheduling policy of waiting queue, the utilization efficiency of DM tends to be neglected. For example, the execution block, which comes from the DM lack for the first task, can not be canceled even if there is a task with the same priority as the first and can be executed in remainning GPU DM. Additionally, the executing time of task(ETT) is often determined according to the required DM of the task(DMT). The exponential growing is often presented between ETT and DMT because of data transmission and synchronization, etc, which is also verified by some experiments.

We repeatedly execute the Matrix Multiplication (MM), the benchmark supplied by CUDA, by increasing matrix scale before each execution. As shown in Fig. 3, the abscissa

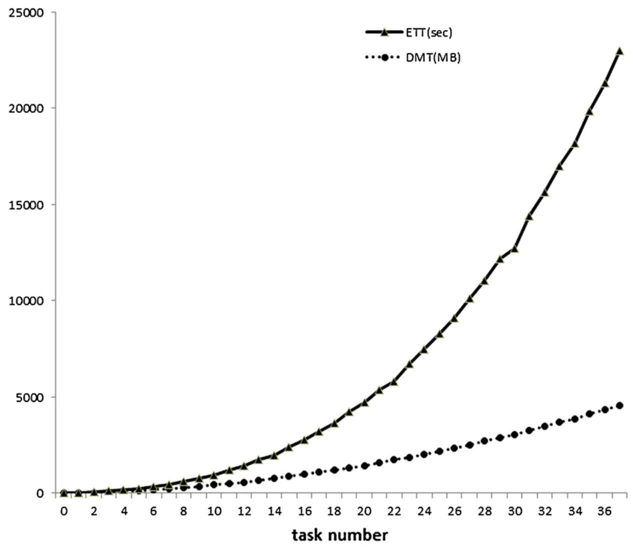

Fig. 3 ETT and DMT in MM test

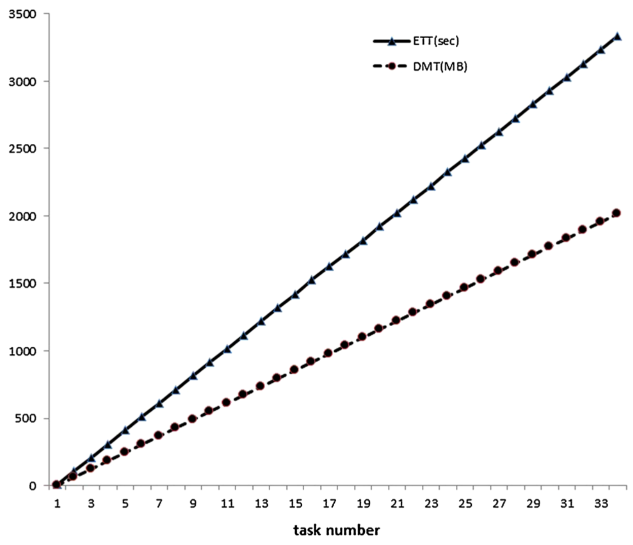

Fig. 4 ETT and DMT in FFT test

is the the serial number of these tasks in MM test, and the rising trend of DMT is presented as the dot line which is auxiliary. For example the number 16 respresents the 16th MM task in which the DMT is 867(MB) and the ETT is 1530(s). The result shows the ETT increases rapidly along with the DMT.

Similar to MM test, a simple application, which realizes some FFT operations including positive transform, inverse transform and convolution, is repeatedly executed by increasing the signal size. The library *cufft*, which is supplied by CUDA, is used in the application code because of its good scalability in GPU environment. As shown in Fig. 4, the ETT and DMT increase in value, but ETT has a faster speed than DMT.

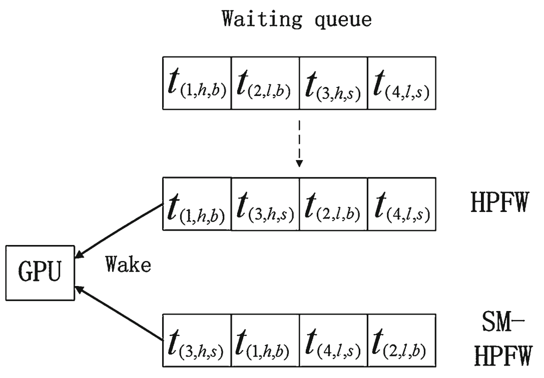
Waiting queue
$t_{(1,h,b)}$ $t_{(2,l,b)}$ $t_{(3,h,s)}$ $t_{(4,l,s)}$
HPFW
$t_{(1,h,b)}$ $t_{(3,h,s)}$ $t_{(2,l,b)}$ $t_{(4,l,s)}$
GPU
Wake
SM-HPFW
$t_{(3,h,s)}$ $t_{(1,h,b)}$ $t_{(4,l,s)}$ $t_{(2,l,b)}$

Fig. 5 The task wakeup order in waiting queue by HPFW and SM-HPFW

4.3.2 SM-HPFW

Combining the ETT effect with the priority requirement, Small-Memory HPFW (SM -HPFW) policy is employed to awake the waiting tasks. The wakeup prioritization order of SM-HPFW is: (1) high-priority wakeup over low-priority wakeup, (2) small-DMT wakeup over big-DMT wakeup. Specifically, the high-priority task can be awaken prior to the low-priority no matter HPFW or SM-HPFW is adopted as scheduling policy. For tasks with the same priority, the task with smallest DMT can be awaken first in SM-HPFW.

In SM-HPFW, each task has an attached subscript including the order of entering waiting queue, priority and DMT size. Two priority levels, high priority and low priority, are abbreviated respectively with the letters h and l. The letter s or b describes the small or big DM size which is needed for the awaken task.

Figure 5 shows the difference between HPFW and SM-HPFW. Though task $t_{(1,h,b)}$ and task $t_{(3,h,s)}$ have a same priority, the latter can be awaken prior to the former because of smaller DMT.

Remarkably, SM-HPFW is unfair for the big-DMT task because it is not considered preferentially, which will be improved in the future multi-GPU scheduling.

5 Performance metric

Response time or makespan has been one of the most important issues in the design of a computer system [2,22]. In our work, the metrics applied in performance evaluation of scheduling system are priority response time (that is the time between the arrival of the task and the beginning of its computation) and task completion time (i.e., the time between the arrival of the task and the completion of its computation). The priority response time is applied to determine whether the scheduling is effective for task priority or not. The task execution time can show the result of system improvement.

For evaluating the general perfomance, the following performance metric is more valid than the above. T_{ri}, the Normalized overall Priority response time (that is the sum of the priority response time of all tasks with priority i), is defined as

$$T_{ri} = SUM(TR_j),\ j = 1, 2, ..., n \qquad (3)$$

where n is the number of all tasks with priority i and TR_j is the priority response time of the jth task.

Another important metric is T_{ci}, the normalized overall task completion time (i.e., the sum of the task completion time of all tasks with priority i), can be written as

$$T_{ci} = SUM(TC_j),\ j = 1, 2, \ldots, n \qquad (4)$$

where n is the number of all tasks with priority i and TC_j is the task completion time of the jth task. The overall task completion time can discover whether the policy improves the system performance. Smaller overall task completion time indicates better performance of scheduling system in general.

6 Experiment

The experiments are carried out on a server with two Intel Xeon E5-2650 and 96 GB memory, and Nvidia Kepler K20M GPU card with 2600Mhz and 4.8 GB device memory. The server is running the GNU/Linux operation system with task version 2.6.32 and CUDA runtime version 5.5. The benchmark is from the CUDA 5.5 SDK and coded by C language.

We focus on the limited DM rather than other GPU resource in the following experiments. To avoid the extra influence from different computing process, we attempt to find a general GPU application to examine these policies. Matrix Multiplication (MM) is adopted in our experiment because the GPU is often used in the matrix-based computation [12] and MM is highly scalable on the GPU [19] and a typical benchmark in CUDA SDK [16,21]. The stream-operation code is added into the benchmark to execute concurrently. The executed tasks are generated in random. The difference of these tasks is the task priority and matrix scale.

First, twenty tasks are used to test the improvement in the order of obtainning DMT by RBDM. As shown in Table 2, the numbers 3, 2 and 1 represent the priority *high*, *middle* and *low*, respectively. Column RBDM or FCFS indicates the execution order of these tasks after they are scheduled by RBDM or FCFS. Compared with FCFS, RBDM can obviously accelerate the DMT obtainment for the high priority task.

Table 2 The scheduling by FCFS and RBDM

Task	Priority	DMT(MB)	RM(MB)	FCFS	RBDM
t_1	1	3421	570	1	3
t_2	3	3137	1568	2	1
t_3	1	630	105	3	4
t_4	2	1934	644	4	3
t_5	3	2262	1131	5	2
t_6	1	3170	528	6	6
t_7	3	412	206	7	4
t_8	3	1047	523	8	5
t_9	1	3621	603	9	8
t_{10}	1	2717	452	10	9
t_{11}	2	2102	700	11	7
t_{12}	2	1998	666	12	7
t_{13}	3	986	493	13	7
t_{14}	1	191	31	14	11
t_{15}	1	2743	457	15	12
t_{16}	3	2563	1281	16	10
t_{17}	1	1143	190	17	13
t_{18}	3	527	263	18	11
t_{19}	1	2130	355	19	14
t_{20}	1	727	121	20	15

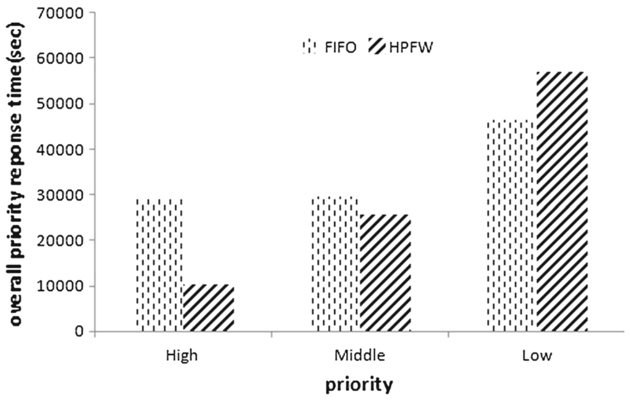

Fig. 6 The overall priority response time (test 1)

To examine the HPFW validity in runtime, the above tasks are rescheduled according to RBDM and awaken by HPFW and FIFO respectively if the task lies in waiting queue. The experiment is repeated with two more groups of tasks. The overall priority response time for each group is shown in Figs. 6, 7 and 8, respectively.

To show the holistic performance clearly, all results in three tests are corrected in Fig. 9. As shown in the figure, the HPFW can significantly shorten the overall priority response time of high-priority task versus the FIFO. The overall priority response time can be decreased down to 53.5 % of FIFO for high-priority tasks. So the execution of high-priority task is brought forward obviously by HPFW.

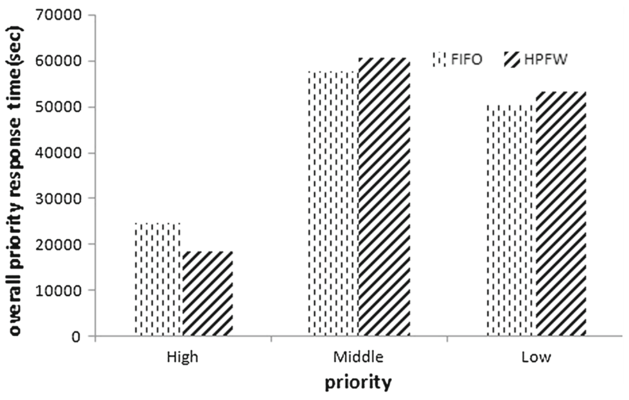

Fig. 7 The overall priority response time (test 2)

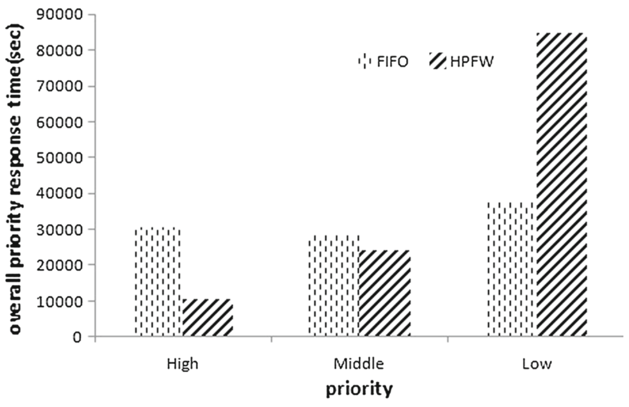

Fig. 8 The overall priority response time (test 3)

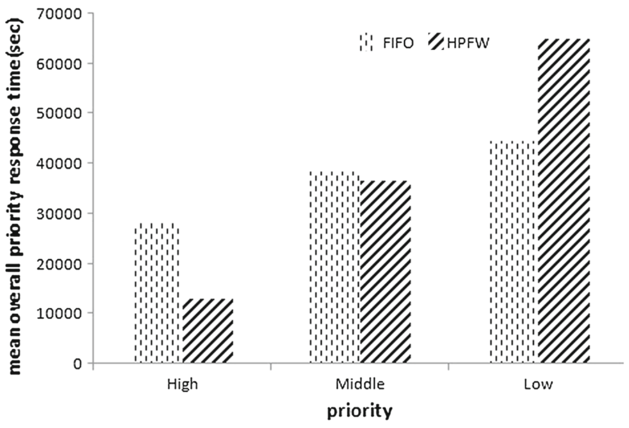

Fig. 9 The mean overall priority response time

In above tests, we record the overall task completion time respectively. As shown in Fig. 10, they are nearly equal between the HPFW and FIFO, which means the HPFW can not improve the system performance significantly.

To examine the SM-HPFW, we should first get a criteria to decide which task has small DM requirement. However, it is diffcult to set the criteria due to the diversity of the GPU hardware and application such as the read/write speed

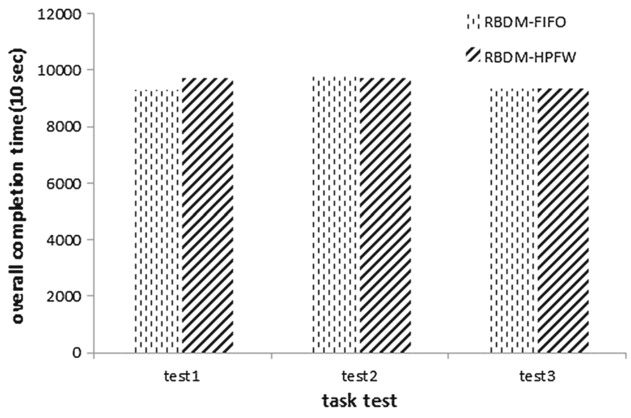

Fig. 10 The overall completion time with FIFO/HPFW

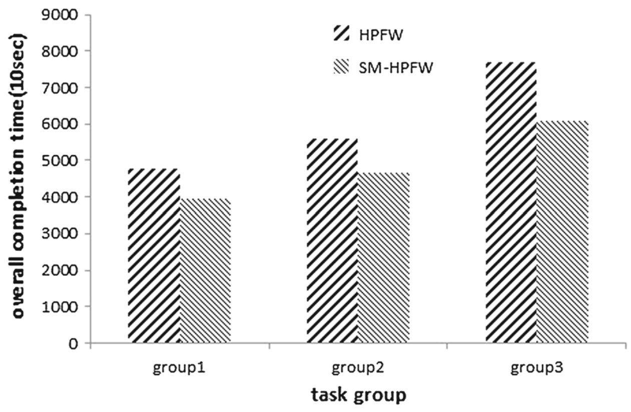

Fig. 11 The overall completion time with HPFW/SM-HPFW

of GPU memory, PCI bandwidth and data character of application. Unfortunately, a reasonable criteria also can not be found in other research. Therefore, we try to get an empirical criteria.

As shown in Fig. 3, the task completion time increases rapidly while its DMT size exceeds 1 GB, so the size 1GB is regarded as a boundary between small DMT and big DMT. The task whose DMT size exceeds 1GB is treated as big task in our experiment, and the other is treat as small task. The empirical value is not precise but useful for finding the difference between SM-HPFW and HPFW.

Noting that the probability proportionate to DMT size and priority is the same for each task in the real-time system, we generate three groups of tasks. The rules of generation tasks for each group are (1) the number of tasks with different priority is the same and (2) the distribution of tasks is even in two kinds of DM size (small and big).

Figure 11 depicts the overall task completion time where the SM-HPFW and HPFW is employed in three group of tasks respectively. Compared with HPFW, SM-HPFW can reduce the overall task completion time by 21 % for group 3, 16.7 % for group 2, and 17 % for group 1, respectively.

7 Conclusion

A DM-based and multi-task scheduling scheme is introduced for the real-time GPU system. The logical reserve is employed to facilitate task to obtain the GPU DM depending on priority. The required DM of task can be predicted in advance. In addition, the HPFW is proposed to reduce the priority response time. The SM-HPFW, a improving HPFW, is employed to improve the system performance. A CPU-based tool is used to monitor the GPU task execution. Experimental results show RBDM is effective and can obviously shorten the overall priority-response time associating with HPFW. The syetem performance can be improved if SM-HPFW is adopted in some cases.

The I/O bottleneck is not considered and the SM-HPFW is unfair for some data-intensive applications. In our future work, we intend to develop real-time scheduler and improved monitor for multiple GPUs since the distributed computing is an optional method for the data-intensive application. For some special applications, new algorithms will be attempted, such as data compressing and asynchronous transmission, to reduce the amount of I/O data.

Acknowledgements This research is supported by NSFC and Shanghai Municipal Education Commission. I would like to extend my sincere gatitude to my friends at Illinois Institute of Technology (IIT), who have provided selfless help for my work and life abroad during my visiting scholar career. I gratefully acknowledge IIT who has offered me a cosy work environment and my colleagues of HPCC, Shanghai university.

References

1. Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., Tallent, N.R.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurr. Comput. Pract. Exp. **22**, 685–701 (2010). doi:10.1002/cpe
2. Chong, E.K.P.: Performance for imprecise evaluation computer of scheduling systems algorithms. J. Syst. Softw. **15**, 261–277 (1991)
3. Eswaran, A., Rajkumar, R.: Energy-aware memory firewalling for QoS-sensitive application. Proc. Euromicro Conf. Real-Time Syst. **2005**, 11–20 (2005). doi:10.1109/ECRTS.2005.14
4. Fang, W., Lau, K.K., Lu, M., Xiao, X., Lam, C.K., Yang, P.Y., He, B., Luo, Q., Sander, P.V., Yang, K.: Parallel data mining on graphics processors. Ph.D. thesis, Hong Kong University (2008). http://gpuminer.googlecode.com/files/gpuminer.pdf
5. Hardy, D., Puaut, I.: Predictable code and data paging for real time systems. In: Proceedings—Euromicro Conference on Real-Time Systems, pp. 266–275 (2008). doi:10.1109/ECRTS.2008.16
6. Hung, C.L., Hua, G.J.: Local alignment tool based on Hadoop framework and GPU architecture. BioMed Res. Int. **2014**, 1–7 (2014). doi:10.1155/2014/541490
7. Jog, A., Bolotin, E., Guz, Z., Parker, M., Keckler, S.W., Kandermir, M.T., Das, C.R.: Application-aware memory system for fair and efficient execution of concurrent GPGPU applications. In: Workshop on General Purpose Processing Using GPUs(GPGPU-7), pp. 1–8 (2014). doi:10.1145/2576779.2576780
8. Joo, W., Shin, D.: Resource-constrained spatial multi-tasking for embedded GPU. In: 2014 IEEE International Conference on Consumer Electronics (ICCE), pp. 2010–2011 (2014)

9. Kato, S., Lakshmanan, K., Rajkumar, R.R., Ishikawa, Y.: TimeGraph: GPU scheduling for real-time multi-tasking environments. In: 2011 USENIX Annual Technical Conference (USENIX ATC11), p. 17 (2011)
10. Kim, H., Rajkumar, R.: Shared-page management for improving the temporal isolation of memory reservations in resource kernels. In: Proceedings—18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2012—2nd Workshop on Cyber-Physical Systems, Networks, and Applications, CPSNA, pp. 310–319 (2012). doi:10.1109/RTCSA.2012.50
11. Kim, H., Rajkumar, R.: Memory reservation and shared page management for real-time systems. J. Syst. Archit. **60**(2), 165–178 (2014). doi:10.1016/j.sysarc.2013.07.002
12. Lindholm, E.N.: Nvidia tesla:aunified graphics and computing architecture. Micro IEEE **28**(0272–1732), 39–55 (2008)
13. Mokhtari, R., Stumm, M.: BigKernel—high performance CPU-GPU communication pipelining for big data-style applications. In: Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium, pp. 819–828 (2014). doi:10.1109/IPDPS.2014.89
14. Nvidia: NVIDIA's Next Generation CUDA Compute Architecture:Kepler GK110. http://www.nvidia.com/content/PDF/kepler/NVIDIA-kepler-GK110-Architecture-Whitepaper.pdf
15. Nvidia: Whitepaper NVIDIAs Next Generation CUDA Compute Architecture:Fermi (2009). doi:10.1016/j.immuni.2005.11.006. http://www.nvidia.com
16. Nvidia: Cuda c programming guide (2013). http://docs.nvidia.com/cuda/cuda-c-programming-guide
17. O'Neil, M.a., Burtscher, M.: Floating-point data compression at 75 Gb/s on a GPU. In: Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units - GPGPU-4, pp. 1–7 (2011). doi:10.1145/1964179.1964189. http://portal.acm.org/citation.cfm?doid=1964179.1964189
18. Rixner, S., Dally, W.J., Kapasi, U.J., Mattson, P., Owens, J.D.: Memory access scheduling. In: Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201), vol. 27, pp. 1–11 (2000). :10.1145/342001.339668
19. Stuart, J.a., Owens, J.D.: Multi-GPU MapReduce on GPU clusters. In: Proceedings—25th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2011, pp. 1068–1079 (2011). doi:10.1109/IPDPS.2011.102
20. Sun, X.H., Wang, D.: Concurrent average memory access time. IEEE Comput. **47**(5), 74–80 (2014)
21. Volkov, V., Demmel, J., Berkeley, U.C.: Benchmarking g GPUs to Tune Dense Linear Algebra. In: Proceedings of the 2008 ACM/IEEE Conference on Superconducting (SC '08), pp. 1–11 (2008)
22. Yazdanpanah, H.: Evaluation performance of task scheduling algorithms in heterogeneous environments. Int. J. Comput. Appl. **138**(8), 1–9 (2016)

Bao-yu Xu was born in 1980. He graduated from Jiangxi Normal University in 2002, received his Master degree from the School of Computer Engineering and Science, Shanghai University, in 2008. He has been in the Scalable Computing Software laboratory at Illinois Institute of Technology (Chicago, 2014–2015) as a visiting scholar. Now, he is an Engineer at Shanghai University. His research interests include High Performance Computing, big data system and application.

Wu Zhang is a Professor of Shanghai University. He graduated from Nanjing Aeronautical Institute in 1980, obtained his Mater Degree of Computational Mathematics at Xi'an Jiaotong University in 1984 and Ph.D. of Aerodynamics at Northwestern Polytechnic University in 1988. He was postdoctoral research fellow of CFD at Peking University during 1989–1991 and Applied Mathematics at UNC Charlotte, USA, during 1996–1998, respectively. Before he joined the School of Computer Engineering and Science, Shanghai University, in 2002, he worked at Beijing University (1991–1993), Xi'an Jiaotong University (1993–2001), and visited, as Associate Professor, EE Department of UNC (Charlotte, 1995–1996), and, as visiting professor, CS Department of Illinois Institute of Technology (Chicago, 2000–2001). He has been working on the research areas of Numerical Solutions of PDEs, CFD, Parallel Algorithms and Applications.

Xian-he Sun is a Distinguished Professor of Computer Science at the Illinois Institute of Technology (IIT). He is the director of the Scalable Computing Software laboratory at IIT and a guest faculty in the Mathematics and Computer Science Division at the Argonne National Laboratory. Before joining IIT, he worked at DoE Ames National Laboratory, at ICASE, NASA Langley Research Center, at Louisiana State University, Baton Rouge, and was an ASEE fellow at Navy Research Laboratories. He is an IEEE fellow and is known for his memory-bounded speedup model, also called Sun–Ni's Law, for scalable computing. His research interests include parallel and distributed processing, memory and I/O systems, software systems for big data applications, and performance evaluation. He has over 200 publications and 5 patents in these areas. He is a former IEEE CS distinguished speaker, a former vice chair of the IEEE Technical Committee on Scalable Computing, the past chair of the Computer Science Department at IIT, and is serving and served on the editorial board of leading professional journals in the field of parallel processing.

Yang Wang, currently work as an experimentalist in School of Computer Engineering and Science, Shanghai University, received his M.Sc. degree from Department of Mechanics, Peking University in 2008, and B.Eng. degree from Department of Engineering Mechanics, Sichuan University in 2005. He has been in Temple University in USA from March 2013 to March 2014 as a visiting scholar. His research areas include High Performance Computing, Computational Fluid Mechanics, etc.