Tasks Fairness Scheduler for GPU

B. López-Albelda, J.M. González-Linares y N. Guil[1]

Abstract— **Nowadays GPU clusters are available in almost every data processing center. Their GPUs are typically shared by different applications that might have different processing needs and/or different levels of priority. As current GPUs do not support hardware-based preemption mechanisms, it is not possible to ensure the required Quality of Service (QoS) when application kernels are offloaded to devices. In this work, we present an efficient software preemption mechanism with low overhead that evicts and relaunches GPU kernels to provide support to different preemptive scheduling policies. We also propose a new fairness-based scheduler named Fair and Responsive Scheduler, (FRS), that takes into account the current value of the kernels slowdown to both select the new kernel to be launched and establish the time interval it is going to run (quantum). Nine applications selected from different benchmark suites have been used to evaluate the computing cost and the eviction delay of our preemption mechanism, showing our implementation has a very low overhead. The proposed scheduler has also been compared with Shortest Job First Scheduler (SJF), Shortest Remaining Time Scheduler (SRT), Round Robin Scheduler (RR) and Completely Fairness Scheduler (CFS). Comparison was carried out using different metrics like average normalized turnaround time (ANTT), deviation normalized turnaround time (DNTT) or system overall throughput (STP). DNTT metric shows that FRS consistently obtains better fairness scheduling results than other schedulers, specifically FRS is 1.5 times lower (better) than SRT (the second best fair scheduler).**

Keywords— **GPU preemption, Scheduling, CUDA Streams.**

I. Introduction

GPUs are broadly used in multitask environments, such as data centers, where applications running on CPUs offload specific functions to GPUs in order to take advantage of the device high performance. In these environments, it is likely to have several independent kernels ready to run concurrently on a GPU.

In this context, several works have been published that try to improve the way kernels are scheduled on GPUs. They pursue different aims like reducing the makespan of a set of kernels by taking advantage of concurrent kernel execution capabilities available in devices [1], [2], or providing priority-based kernel execution by developing soft-real time schedulers [3]. Implementing more complex scheduling policies which provide quality of service (QoS), fairness, and support for different priorities requires the ability to preempt running kernels, i.e., evict a kernel before finishing execution and schedule a new kernel. Unfortunately, although NVIDIA GPUs support some sort of compute preemption since Pascal architecture [4], it is restricted to a few uses such as interactive graphics tasks and debuggers, and it is not exposed to programmers. There are some research proposals of hardware-based preemption mechanisms, [5], [6], but they are not available in real GPUs, and there is no guarantee that they would be more efficient than software-based approaches. The implementation of a preemption mechanism on GPUs, similar to those employed on CPU, would require to save the state of all active threads (e.g., the contents of the entire register file) in the running kernel. As thousands of threads can be active in a standard GPU, the cost of collecting and saving all these states would be very high and incompatible with the implementation of an efficient preemptive scheduler, where kernels should be evicted and relaunched with minimum overhead.

More recently, other works have proposed the use of software-based preemption mechanisms to improve the scheduling policies [7], [8]. Thus, these methods only need to keep track of the number of pending thread blocks. As a result, they avoid the execution of expensive operations to save kernel state. Our preemptive scheme also follows this thread block-based approach. The mechanisms proposed by [7], [8] are based on zero-copy memory transactions between GPU and CPU memories. In addition, they require to run a proxy kernel on GPU to reduce the PCI data traffic generated during preemption. Our scheduler will not use a GPU proxy. Therefore, our scheduler sends scheduling commands to the running kernel by waiting GPU global memory variables. This can reduce the traffic through the interconnecting bus and use all the GPU resources only for kernel execution.

Regarding scheduling policies, previous works have implemented systems based on simple priority, priority queues or round robin schemes, [3], [9], [8] among others. However, not many works have studied fairness based policies to build GPU schedulers. We think fairness-oriented policies are necessary in GPUs servers where users demand a fair distribution of the available resources.

With the aim of improving current techniques for GPU kernel scheduling, we use explicit transfers to deploy our preemption mechanism We also develop a Fair and Responsive scheduler, FRS, that ensures, in a holistic way, fairness across different GPU applications. Thus, we make two main contributions:

- An efficient preemption mechanism that directly sends scheduling commands to the running kernel using GPU global memory variables. This way, we avoid the use of a proxy kernel, which allows application kernels to have more GPU resources available and, at the same time, eliminates the constant PCI data traffic produced by zero-copy operations.
- A new scheduler that guarantees fairness in the use of GPU resources. It monitors the pending

[1]University of Málaga, Andalucía Tech, Dept. of Computer Architecture, Spain, e-mail: {blopeza, jgl, nguil}@uma.es.

work of each kernel to minimize the kernel slowdown, and implements a policy to balance the global slowdown.

The rest of the article is organized as follows: Section II gives an overview of the preemption mechanism, while the implementation is deeply described in Section III. Section IV introduces our fairness-oriented scheduler along with other popular schedulers used for comparison purpose. Metrics used in our experiments are presented in Section V. The experimental setup and evaluation of our approach are described in Section VI. Related works are discussed in Section VII and, finally, Section VIII concludes the article.

II. Preemption overview

Our scheduler uses the preemption mechanism to implement different QoS-aware scheduling policies. This scheduler, which runs on a CPU thread, is continuously monitoring a queue of eligible kernels. Each kernel is inserted in this queue by the corresponding application. A particular kernel can be declared *eligible* when its required device memory allocation and host-to-device data transfers are done. The application is responsible for memory allocation and the data transfers.

When a condition for eviction is fulfilled, the preemption mechanism is initiated by the scheduler. The scheduler may issue three commands: 1) The *eviction* command indicates that the currently running kernel should be evicted. It is implemented through a data transfer that updates a *State* variable in device memory, which indicates to the running kernel that it has to evict. 2) The *pending work* command is intended to collect information about the kernel remaining work, i.e. the number of pending tasks, by reading a variable located on device memory. 3) The *kernel (re-)launch* command updates the *State* variable associated to the re-launched kernel to set the corresponding value. This way, it guarantees a low delay in the preemption mechanism.

The global memory region used by an evicted kernel is kept during the execution of other kernels. Several works propose different migration techniques to free space in global memory [10]. They are orthogonal to our mechanism, and could be implemented if needed.

Section III gives a detailed description of the implementation of our preemption mechanism while Section IV-E describes our Fair and Responsive Scheduler (FRS), that uses this preemption mechanism to implement a new fairness-aware based scheduling policy.

III. Implementation Details

In this section the two main elements of our preemption mechanism are explained. First, we show the modifications that must be applied to original kernels to support preemption. Second, the core of the preemption mechanism is presented, that is, all the operations involved in the eviction mechanism.

A. Kernel Transformation

The implementation of the preemption mechanism requires the modification of the original kernels, although this modification can be easily automatized. First, the original kernel grid must be modified so that it is executed using persistent thread blocks. Thus, our scheme launches just the number of thread blocks that fit into the available SMs ($maximum\_number\_of\_blocks\_per\_SM \times numSMs$). This transformation can be done automatically by a compiler, as a previous work has shown [7].

The proposed transformation has two advantages. First, persistent thread blocks typically execute more iterations in comparison with no persistent thread blocks and eviction could be performed at the end of each iteration. Thus, when a previously evicted kernel is relaunched, each thread block would just resume the execution from the last executed iteration. Second, this eviction mechanism works faster as only tenths (at most a few hundredths in modern Volta architectures) of persistent thread blocks are active instead of several thousandths blocks in many kernels.

```
1  /************* GPU code *************/
2  Kernel_func (list_of_original_params,
       int MaxNumTask, int *State, int *
       TaskId) {
3    while(true) {
4    // Check state (only one thread)
5      if (threadIdx.x == 0) {
6        if (*State == EVICTED)
7          blockId = -1;
8        else
9          blockId = atomicAdd(TaskId, 1);
10     }
11     // synchronize block threads
12     _sync_threads();
13     // end checking: no more tasks or
           evict
14     if (blockId>=MaxNumTask || blockId
           ==-1)
15       return;
16     // Original kernel code follows
           here
17   }
18 }
19 /********* CPU calling code *********/
20 Kernel_func<persist_grid_size> (
       list_of_original_params, MaxNumTask
       , State, TaskId);
```

Listing 1
Original kernel code modification to support preemption. New code is highlighted in bold type.

A flexible mechanism for load distribution among thread blocks is also proposed. Instead of assigning a specific computation to each thread block with a fixed mapping [11], thread blocks obtain load information by atomically updating a common global memory variable at the start of each iteration. This global memory variable acts as a counter (starting from zero) that increments a task index. These tasks ordering do not affect original kernel execution and benefits the preemption mechanism as only the index of the last executed task must be saved when

a kernel is evicted. In this paper we use the term *task* to name the basic unit of work and it is given by the amount of computation done by one thread block during one iteration.

In Listing 1 main kernel changes are indicated. As it can be observed, two new global memory variables are required per kernel (line 2). One of these variables, called *State*, keeps the state of the GPU kernel. It can take two possible values: Running or Evicted. Just before a kernel is launched it is set to Running. The other memory variable is called *TaskId* and contains the index of the next available task. Initially it is set to 0. In addition there is a new parameter, *MaxNumTask* (also in line 2), that contains the total number of tasks to execute. It is checked by GPU threads to acknowledge when all tasks have been computed and the kernel execution is done.

Listing 1 also shows the code for the eviction mechanism and the tasks distribution strategy. The original thread block computation is enclosed within a while loop (line 4) that finishes when either an eviction command is sent by the CPU scheduler thread or no more pending tasks are available (line 14). At the beginning of each iteration the thread with id 0 is in charge of reading the *State* variable (line 6). If the state has changed to Evicted, a variable mapped in shared memory, called *blockId*, is set to -1 (line 7). This variable is also used to store the task id when the kernel is running (line 9). As it can also be observed in line 9, new values of task id are obtained by thread 0 at each thread block iteration by executing an AtomicAdd instruction on *TaskId* variable.

A block synchronization instruction is also added (line 12) so that the rest of the block threads wait for thread 0. After this barrier, all block threads read *blockId* and check (line 14) finishing condition (all kernel tasks have been computed or *State* has been changed to Evicted). If condition is not fulfilled, thread block executes a new task (with the current *blockId* value). An additional modification must also be applied to the original code to change the indexation employed during computation. In original kernels this indexation is typically implemented using thread block indexes while in the modified kernel the *blockId* variable should be used.

B. Eviction and launching implementation

Unlike previous approaches [7], our preemption mechanism does not require the use of a proxy kernel in the GPU. Therefore, all GPU resources can be devoted to workload execution. In addition, the scheduler can directly access device global memory positions by sending eviction commands and reducing greatly the bus traffic with respect to recent works, [7], [8], as it was explained in Section I.

In previous section we explained that the eviction mechanism requires to distinguish between two kernel states: Running and Evicted. There is a global memory variable, *State*, that contains one of this two possible values. The scheduler is in charge of changing the variable content using a Host to Device (HtD)

Fig. 1. A profiler capture showing the eviction (and re-launching) of five kernels. Three transfers are required to evict a running kernel and launch a new one.

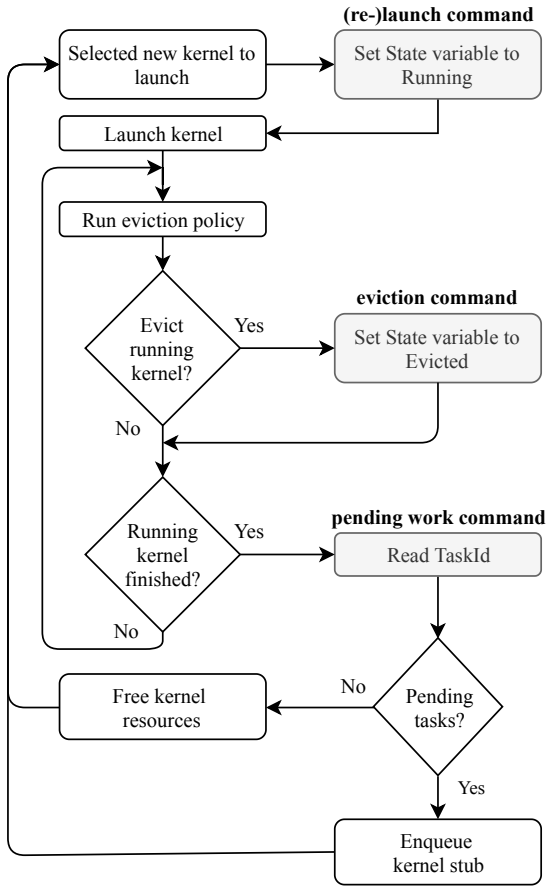

Fig. 2. Flow diagram of the scheduler that illustrates the three transfer commands (grey boxes) involved in the eviction and launching procedures. All commands are sent by the scheduler to TM queues for further processing.

transfer. Thus, as it is shown in Figure 2, before a kernel is launched (or re-launched after a previous eviction) this value is set to Running (*launch command*). When the scheduler decides to evict the running kernel it changes the state to Evict using a new HtD transfer (*eviction command*). Then, when thread blocks start a new iteration, their thread 0 consult the state of this variable and exit if the value is set to Evicted. When all the kernel thread blocks have exited, the kernel is considered evicted. Meanwhile, the scheduler is using a non-blocking synchronization call to detect kernel termination. Kernel termination can happen for two different reasons: 1) all tasks have been executed, or 2) an eviction command has been issued. Thus, the complete eviction process also requires the scheduler checks the number of pending tasks of the just finished kernel. This way the scheduler can know if kernel has finished (no pending tasks) or if it needs to be relaunched later. Thus, when the scheduler detects that the current executing kernel has finished it issues a *pending work*

command to read the current value of the *TaskId* variable.

When a new kernel is selected, the three commands involved in the preemption mechanism are launched again. All these commands are submitted (gray boxes in Figure 2) using specific streams in order to avoid false dependencies with other commands using the same streams.

Figure 1 shows a capture of the NVIDIA profiler, *nvpp*, that illustrates the transfers involved in the preemption mechanism. There are three timelines in the figure. Two of the timelines correspond to HtD and DtH transfers since the GPU used in the experiment has two DMA engines. The other timeline, called Compute, is used to indicate kernel execution. Thus, we can appreciate the execution of several kernels (bars with different colors). In the gap between two kernels execution, a group of three short transfers appears (in brown color). From left to right, the first transfer (HtD) changes the kernel status to Evicted. Then, when kernel finishes, a DtH transfer reads the number of kernel pending tasks. Finally, when a new kernel is selected to run, its status is changed to Running with a HtD transfer.

IV. Schedulers

In this section the schedulers used in our study are presented. All of these schedulers support preemptive scheduling. Thus, when a priority kernel arrives the running kernel is preempted.

A. SJF: Shortest Job First

In Shortest Job First (SJF) the priority of a kernel is based on its duration. The longer its duration, the lower its priority. SJF minimizes the total waiting time of a set of jobs, giving superior responsiveness. For applications running several kernels, their priority will be computed using the total time of their kernels.

B. SRT: Shortest Remaining Time

Shortest Remaining Time (SRT) is similar to SJF but, in this case, the priority is dynamically updated. When a new kernel arrives, the remaining time of each kernel/application is estimated, and a new priority is assigned to each of them. The remaining time of applications running several kernels is calculated using the sum of the remaining times of their kernels.

C. RR: Round Robin

In a Round Robin scheduler (RR) all kernels have the same priority and a fair execution time (*quantum*) is assigned to each kernel. In this way, all kernels will be executed during its *quantum* on a first-come, first-served manner. If a kernel does not finish during its *quantum*, the kernel is preempted and added to the end of the waiting queue.

D. CFS: Completely Fairness Scheduler

The Completely Fair Scheduler (CFS) is a well-known implementation of a fairness policy and it was used as the default scheduler in the Linux kernel (2007 release). CFS defines an *epoch* in order to assign the same amount of computation time at each ready kernel. Thus, the *quantum* assigned to a specific kernel is given by the result of dividing the epoch value by the number of active kernels. Within an epoch, execution ordering is calculated using the time a kernel has been waiting for execution. Thus, kernels are ordered in decreasing order attending to their waiting time values, and scheduled following that ordering.

E. FRS: Fair and Responsive Scheduler

One of the aims of this work is the study of fair policies for scheduling kernels on GPUs. These policies are built using some fairness measure and, in this paper, we have selected the slowdown variance. In this context a fair scheduling policy should try to obtain similar slowdowns for all running jobs, that is, it should implement a *proportional fairness* policy [12]. We consider this type of fairness is interesting for a scheduler that arranges jobs from different applications on a GPU as the assignment of resources among different kernels should tend to be balanced. Because of that, we propose a scheduler, named Fair and Responsive Scheduler (FRS), that uses a kernel slowdown metric to take scheduling decisions.

The design of FRS is based on the instantaneous slowdown, IS, of a kernel. The IS of a GPU kernel that has executed N_t tasks from a total of NT tasks, after spending t seconds since it became ready, is given by

$$IS = (t + (NT - N_t) * tpt)/(N_t * tpt) \qquad (1)$$

being tpt the time, in seconds, required to execute a task. The expression $(NT - N_t)$ is the number of tasks that remains to be executed. Thus, analyzing the expression for IS, we can see that it calculates the rate between the predicted execution time, assuming that all the remaining tasks will be executed in a row, and the shortest execution time, assuming the entire kernel was executed with no interruption. The value of tpt in the expression could be extracted from a brief previous execution of the kernel in the case the kernel has a regular behavior, or it could be updated after each kernel eviction. Notice that the use of the *pending work* command in our preemption mechanism allows the scheduler to know N_t and to calculate tpt at each kernel eviction.

When a kernel is evicted, the scheduler updates the IS values of all the ready kernels and chooses the one with highest IS, IS_{max}. A *quantum* value is assigned to the kernel using the following formula

$$quantum = IS_{max} * NT * tpt - (NT - N_t) * tpt - t \quad (2)$$

where the values of NT, tpt, N_t and t are given by the task with minimum IS value. This way, after executing the chosen kernel for this *quantum*, the new IS value of the task with minimum IS will be the same as the launched task, that is, IS_{max}.

V. Metrics

In this section, we introduce the metrics we have used to measure the effectiveness of each scheduler attending to different criteria as performance, responsiveness and fairness given a set of n tasks.

A. Average normalized turnaround time (ANTT)

The NTT of tasks i is defined as follows:

$$NTT_i = T_i^{MP}/T_i^{SP} \quad (3)$$

where T_i^{MP} and T_i^{SP} are the execution times of the task in its co-run and its standalone run respectively. NTT_i is usually greater than 1, the smaller the more responsive the application is. $ANTT$ is the average of $NTTs$ for all the executed applications.

$$ANTT = \overline{NTT} \quad (4)$$

where NTT is a random variable that takes a specific value for each evaluated task.

B. System overall throughput (STP)

It is defined as follows:

$$STP = \sum_{i=1}^{n} T_i^{SP}/T_i^{MP} \quad (5)$$

STP varies from 0 to n (the number of applications); the higher, the better.

C. Deviation normalized turnaround time (DNTT)

We employ this metric to evaluate how fair is the execution of a set of taks. It is based on the slowdown variance value [12] and considers that a fair scheduling of a set of tasks should obtain low slowdown variance, that is, the lower, the better. Thus,

$$DNTT = \sigma(NTT) \quad (6)$$

VI. Experiments

Experiments have been conducted using different applications with kernels belonging to CUDA SDK [13], Rodinia [14] and Chai [15] benchmark suites. With these applications we pursuit to build a real workload where several applications (up to nine) share the GPU. All experiments have been run on a server with two Xeon(R) E5-2620 CPUs and one NVIDIA TITAN X Pascal. The interconnecting bus is a PCIe 3.0.

Table I shows the list of application used. Most of them have only one kernel, but two of them, namely Separable Convolution and Canny, are composed by two (RCONV and CCONV) and four kernels (GCEDD, SCEDD, NCEDD and HCEDD), respectively. Kernels of both applications are executed in a pipeline fashion since the output of one kernel is the input to the next kernel.

In order to evaluate our approach, several experiments have been designed. Thus, first two experiments focus on analyzing both the overhead of the required kernel modifications and the performance of the preemption mechanism. The third experiment evaluates different fairness scheduling policies.

TABLE I

Applications used in the experiments along with time, in milliseconds, taken by their execution commands. Most of the applications consist of one kernel although Separable Convolution and Canny are composed by two and four kernels, respectively.

Application	Source	Description	Execution Time (ms)
CEDD	Chai	Canny	13.57
SPMV	Rodinia	Sparse MV Mult.	8.36
VA	CUDA SDK	Vector Addition	3.71
CONV	CUDA SDK	Separable Convolution	3.27
BS	CUDA SDK	Black Scholes	2.36
PF	Rodinia	Path Finder	1.65
MM	CUDA SDK	Matrix Mult.	1.43
HST256	CUDA SDK	Histogram	1.29
RED	CUDA SDK	Reduction	0.87

A. Kernel transformation overhead

Implementation of the preemption mechanism requires the transformation of the original kernel code. In this experiment, original and modified kernels are executed until they finish. This way we can measure the overhead incurred by kernel transformation, O_t. The value for this overhead is given by the expression $O_t = \frac{T_t}{T_o}$, where T_t and T_o indicate the execution times of the transformed and original kernels, respectively. Experimental results are shown in the second column of Table II.

Attending to the expression of the overhead, values higher than one are expected since the kernel transformation explained in Section III-A increases the number of instructions executed by the kernel. However, there are cases where values are lower than one. These results can be explained by the fact that kernel transformation also implies a modification in the number and granularity of thread blocks. For instance, original kernel of HST256 uses a small number of blocks with coarse granularity. After our transformation, the number of thread blocks of HST256 are increased and, consequently, the granularity is decreased. With these modifications the kernel runs faster on our device. Focusing on kernels with overhead higher than one we can see that the execution time increases, at most, by 5%, with the exception of MM. Thread blocks of these kernels employ all the available shared memory and the $TaskId$ variable must be mapped in global memory in order to keep the occupancy of the original kernel. Then, the overhead is increased due to longer memory latency. Nevertheless, the average overhead is 0.98, that can be considered almost negligible.

B. Eviction delay

The eviction mechanism is based on asynchronously updating the *State* variable located on GPU global memory (see Sec. III-A) by the scheduler running on the CPU. GPU thread blocks read this variable and, depending on its content, either finish or keep running. Consequently, a delay can be expected between the submission of the transfer command that changes the variable content and the termination of the running kernel. This delay

TABLE II

SECOND COLUMN INDICATES THE OVERHEAD INCURRED BY KERNEL MODIFICATION. THIRD COLUMN SHOWS THE DELAY (IN MICROSECONDS) OF THE PREEMPTION MECHANISM FOR EACH KERNEL.

Kernel	Transformation Overhead	Eviction Delay (μs)
GCEDD	1.02	45
SCEDD	0.98	40
NCEDD	1.02	38
HCEDD	1.04	37
SPMV	0.73	229
VA	1.00	95
BS	0.92	83
RCONV	0.91	53
CCONV	0.97	46
PF	1.00	112
MM	1.11	86
HST256	0.85	94
RED	1.05	32

is directly related to how frequently a thread block reads the variable. Thus, a reduction of the delay requires more frequent memory reads which, however, increase the overhead of the preemption mechanism (as it was discussed in previous subsection). As we are interested in keeping the code modification as simple as possible, the minimum granularity is given by the computation enclosed within a thread block of the original kernel (see Listing 1). If necessary, this granularity could be increased by applying a coarsening technique to the thread block code.

The third column of Table II shows the eviction delay for each kernel. Most of the kernels have an eviction delay lower that 100 μs. Only SPMV has high values for eviction delay. In this kernel, matrix rows are distributed among thread blocks. When the size of the row is large (more precisely, the number of elements different from zero), as in our data set, the computation performed by the thread block is high and the response to changes in the State variable is slow. The average delay for all kernels is around 76μs, a value that permits the development of schedulers with short eviction intervals.

C. Fair scheduling

We have conducted a third set of experiments to compare SJF, SRT, RR, CFS and FRS. Experiments are run by executing all the applications concurrently. Thus, one CPU thread per application is created. All the HtD and DtH commands are launched by these threads while the kernel execution commands are enqueued in a scheduling queue that is managed by the scheduler. Similar to the experiments of other previous works [7], kernels arrive orderly and with 1 ms between them. One hundred of kernels combinations have been run ten times, using each scheduling policy, and their execution times have been averaged by each application and scheduler. In order to obtain more consistent results, all the initial HtD transfers required by the kernels are completed before the scheduler starts.

Several parameters have been also fixed. Thus, the minimum *quantum* size for FRS is established in 1 ms. The epoch size for CFS is fixed to 4 ms (several values were tested and the best value was chosen). The *quantum* size for RR is set to 1 ms and it was chosen to reduce the waiting time without penalizing the turnaround time of short kernels. Table I shows that PF, MM, HST256 and RED have an execution time near to 1 ms; then, if we had used a larger *quantum*, the turnaround time would be affected for these applications.

Although the main objective of this experiment is to compare fairness of different scheduling policies using the DNTT metric, we have also computed the other metrics introduced in Section V to extract complementary information regarding the performance of the evaluated scheduling policies. Thus, Figure 3 shows the normalized turnaround time obtained for each kernel. This metric is used to measure the slowdown of each application. CFS gets the worst NTT, with RR as the second worst scheduler. As we mentioned in Section IV-D, the *quamtum* assigned to each kernel in CFS depends on the active kernels in each epoch. Therefore, the same *quantum* is assigned to short and large kernels, and this increases the turnaround time of the short kernels. RR obtains a better value for *Reduction* because the assigned *quantum* is higher that its execution time thus, although it must wait a long time, it executes completely when it is scheduled. SJF and SRT obtain the best results, followed closely by FRS. Short kernels obtain better values because all these schedulers give them higher priority.

All these NTT values can be used to compute the $DNTT$, $ANTT$ and STP metrics. $DNTT$ metric results are shown in the middle column of Figure 4. A high value means normalized turnaround times of the scheduled applications have a high variability, thus a fair scheduler should obtain low values. CFS and RR get the worst results, with 2.99 and 1.41 respectively, mainly because NTT in short kernels is much larger than in long kernels. SJF and SRT obtain a slowdown variance of less than 1.0, more precisely a value of 0.7 and 0.63 respectively. Finally, FRS obtains the best value (0.42), which is around 1.5, 1.66, 3.35 and 7.11 times lower (better) than SRT, SJF, RR and CFS respectively.

Average values, obtained with the $ANTT$ metric (Figure 4 first column), are related to the responsiveness of the scheduling policy. CFS gets once again the worst value (6.0), followed by RR with a value of 4.87. SJF, SRT and FRS have a similar average normalized response time, close to 2.0 of their standalone application execution time (1.95, 1.96 and 2.44 respectively). These scheduling policies are very responsive, to both short and long kernels.

Finally, we have studied the system overall throughput (STP), showed in Figure 4 third col-

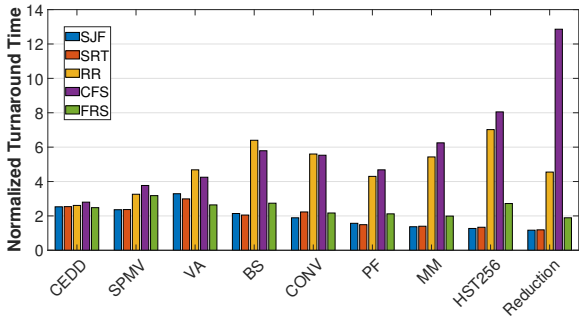

Fig. 3. Normalized turnaround time; SJF, SRT, RR, CFS and FRS priorities are used. The benchmarks are ordered in decreasing order of their simple execution time (from left to right).

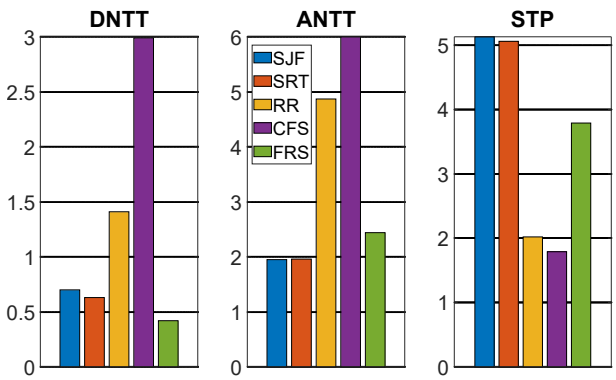

Fig. 4. Average normalized turnaround time ($ANTT$ the lower, the better), deviation normalized turnaround time ($DNTT$ the lower, the better) and system overall throughput (STP the higher, the better) under SJF, SRT, RR, CFS and FRS.

umn. This metric is used to measure the throughput of the system when all the applications are running together with respect to their standalone execution. Thus a higher STP value means a better value. CFS and RR obtain the worst results because the applications need a long time to be executed (1.79 and 2.02 respectively). On the other hand, SFT and SRT get the better STP value with 5.13 and 5.06, respectively, while FRS stands in the middle with a satisfactory result of 3.79.

VII. Related work

Early works on GPU tasks scheduling have tried to increase the responsiveness of the system by minimizing the impact of the non-preemptive nature of both DMA transfers and GPU kernels execution. Thus, Kato et al. [16] studied how to divide memory transfers in chunks to increase the concurrency with kernel launches. Other works, like [3] and [17], have proposed to profile GPU resources to schedule high priority kernels to meet soft real-time requirements. On the other side, some authors have analyzed techniques like elastic kernels [18] or kernel slicing [19], [20], to improve multiple kernels' concurrent execution. Furthermore, some works use the idea of persistent threads [21] for dynamic load balancing of several kernels [22], [23]. Neither of these works have a preemption mechanism that could be used to implement a more sophisticated scheduler or to avoid priority inversion problems.

Recent works have proposed software based preemption mechanisms. Thus, Chen et al. [7] presented a software framework that enables temporal preemption at thread block level. Compared to our work, their mechanism needs a special kernel running in the GPU for each application to serve as a proxy for their host runtime, and lengthy host to/from device transfers can produce priority inversion problems. Wu et al. [8] introduced a spatio-temporal preemption mechanism at thread block level, where thread blocks running in some SMs can be evicted while the others continue to run. Yun et al [24] proposed a similar preemption mechanism, where GPU resources assigned to each kernel are dynamically adjusted by evicting thread blocks. Once again, there is no support for a data transfers control mechanism that could avoid unnecessary delays.

There have been some other works that have studied hardware mechanisms to enable preemptive multiprogramming on GPUs. Tanasic et al. [5] proposed two mechanisms: a context switch that needs to save the execution context of each running thread block, and a SM draining mechanism that waits for each currently running thread block to finish. Park et al. [6] added a third mechanism, SM flushing, to instantly preempt idempotent kernels, i.e., kernels that can be safely restarted. All these works are orthogonal to ours, as an efficient preemption hardware mechanism would benefit our scheduling algorithm.

There are not many works that include QoS or fairness scheduling strategies. Chen et al. [25] presented Prophet, a QoS scheduler that predicts performance of co-located applications. Kerbl et al. [9] proposed bucket queues to schedule tasks using some simple strategies, like FIFO or priorities, or more complex strategies by assigning quotas to the queues. Nevertheless, they do not consider any preemption mechanism thus they are vulnerable to priority inversion problems.

VIII. Conclusions

In this work we have presented a software-based preemption mechanism that can be used to design schedulers for current GPU systems. Experiments with a workload composed by nine applications have shown that the overhead of the kernel modifications required by the preemption mechanism is negligible.

Nevertheless, the main advantage of our scheme, which marks the difference with other recent works, is that a GPU proxy is not needed for the preemption mechanism. Only three transfers are used in our preemption mechanism: 1) an *eviction* command from CPU to GPU, 2) a *pending work* command of the running kernel from GPU to CPU, and 3) a kernel *(re−)launch* command, that updates the *State* variable, from CPU to GPU. Our implementation obtains low delays (less that 0.1 ms) and allows the design of efficient schedulers.

Furthermore, the preemption mechanism also permits to know the pending work of running kernels.

We have used this information to develop a new Fair and Responsive Scheduler, FRS, that tries to balance the instantaneous slowdown of the active kernels. Comparison results with other schedulers, Shortest Job First, Shortest Remaining Time, Round Robin and Completely Fair Scheduler, show that our scheduler obtains the best fairness values, 1.5 times lower (better) than the second best scheduler, using the $DNTT$ metric. Furthermore, FRS get a $ANTT$ of 2.44 close to the most responsive scheduling policies, SJF and SRT. Finally, STP metric shows that FRS gets a satisfactory result of 3.79.

IX. Acknowledgements

This work has been funded by project TIN2016-80920R (Spanish Ministry of Science and Technology) and University of Malaga (Campus de Excelencia Internacional Andalucía Tech). We gratefully acknowledge the support of NVIDIA Corporation with the donation of the TITAN X Pascal GPU used for this research.

Referencias

[1] A.J. J. Lázaro-Muñoz, J.M. González-Linares, J. Gómez-Luna, and N. Guil, "A tasks reordering model to reduce transfers overhead on GPUs," *Journal of Parallel and Distributed Computing*, vol. 109, pp. 258–271, nov 2017.

[2] Florian Wende, Frank Cordes, and Thomas Steinke, "On improving the performance of multi-threaded CUDA applications with concurrent kernel execution by kernel reordering," *Symposium on Application Accelerators in High-Performance Computing*, pp. 74–83, 2012.

[3] Shinpei Kato, Karthik Lakshmanan, Ragunathan Rajkumar, and Yutaka Ishikawa, "Timegraph: Gpu scheduling for real-time multi-tasking environments," in *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, Berkeley, CA, USA, 2011, USENIXATC'11, pp. 2–2, USENIX Association.

[4] NVIDIA, "NVIDIA Tesla P100," 2016.

[5] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling preemptive multiprogramming on gpus," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, June 2014, pp. 193–204.

[6] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke, "Chimera: Collaborative preemption for multitasking on a shared gpu," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 2015, ASPLOS '15, pp. 593–606, ACM.

[7] Guoyang Chen, Yue Zhao, Xipeng Shen, and Huiyang Zhou, "Effisha: A software framework for enabling efficient preemptive scheduling of gpu," in *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, New York, NY, USA, 2017, PPoPP '17, pp. 3–16, ACM.

[8] Bo Wu, Xu Liu, Xiaobo Zhou, and Changjun Jiang, "FLEP: Enabling Flexible and Efficient Preemption on GPUs," in *ASPLOS '17*, New York, New York, USA, 2017, pp. 483–496, ACM Press.

[9] Bernhard Kerbl, Michael Kenzel, Dieter Schmalstieg, Hans Peter Seidel, and Markus Steinberger, "Hierarchical Bucket Queuing for Fine-Grained Priority Scheduling on the GPU," *Computer Graphics Forum*, vol. 36, no. 8, pp. 232–246, 2017.

[10] Taichiro Suzuki, Akira Nukada, and Satoshi Matsuoka, *Euro-Par 2015: Parallel Processing: 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings*, chapter Efficient Execution of Multiple CUDA Applications Using Transparent Suspend, Resume and Migration, pp. 687–699, Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[11] Y. Liang, H. P. Huynh, K. Rupnow, R. S. M. Goh, and D. Chen, "Efficient GPU Spatial-Temporal Multitasking," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 3, pp. 748–760, March 2015.

[12] Vincent J. Maccio, Jenell Hogg, and Douglas G. Down, "On slowdown variance as a measure of fairness," *Operations Research Perspectives*, vol. 5, pp. 133 – 144, 2018.

[13] NVIDIA, "Cuda sdk code samples," May 2018.

[14] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, Oct 2009, pp. 44–54.

[15] J. Gómez-Luna, I. E. Hajj, L. Chang, V. Garca-Floreszx, S. G. de Gonzalo, T. B. Jablin, A. J. Pea, and W. Hwu, "Chai: Collaborative heterogeneous applications for integrated-architectures," in *ISPASS*, April 2017, pp. 43–54.

[16] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar, "RGEM: A Responsive GPGPU Execution Model for Runtime Engines," in *2011 IEEE 32nd Real-Time Systems Symposium*. nov 2011, pp. 57–66, IEEE.

[17] Haeseung Lee and Mohammad Abdullah Al Faruque, "Gpu-evr: Run-time event based real-time scheduling framework on gpgpu platform," in *Proceedings of the Conference on Design, Automation & Test in Europe*, 3001 Leuven, Belgium, Belgium, 2014, DATE '14, pp. 220:1–220:6, European Design and Automation Association.

[18] Sreepathi Pai, Matthew J Thazhuthaveetil, and R Govindarajan, "Improving GPGPU concurrency with elastic kernels," *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems - ASPLOS '13*, p. 407, 2013.

[19] Qiumin Xu, Hyeran Jeon, Keunsoo Kim, Won Woo Ro, and Murali Annavaram, "Warped-slicer: Efficient intra-sm slicing through dynamic resource partitioning for gpu multiprogramming," in *Proceedings of the 43rd International Symposium on Computer Architecture*, Piscataway, NJ, USA, 2016, ISCA '16, pp. 230–242, IEEE Press.

[20] J. Zhong and B. He, "Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1522–1532, June 2014.

[21] Timo Aila and Samuli Laine, "Understanding the efficiency of ray traversal on GPUs," *Proceedings of the 1st ACM conference on High Performance Graphics HPG 09*, p. 145, 2009.

[22] Sanjay Chatterjee, Max Grossman, Alina Sbîrlea, and Vivek Sarkar, "Dynamic task parallelism with a GPU work-stealing runtime system," in *7146 LNCS*, 2013, pp. 203–217.

[23] Stanley Tzeng, Brandon Lloyd, and John D. Owens, "A GPU Task-Parallel Model with Dependency Resolution," *Computer*, vol. 45, no. 8, pp. 34–41, aug 2012.

[24] Chao Yu, Yuebin Bai, Hailong Yang, Kun Cheng, Yuhao Gu, Zhongzhi Luan, and Depei Qian, "SMGuard: A Flexible and fine-grained resource management framework for GPUs," *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[25] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang, "Prophet: Precise QoS Prediction on Non-Preemptive Accelerators to Improve Utilization in Warehouse-Scale Computers," *ACM SIGOPS Operating Systems Review*, vol. 51, no. 2, pp. 17–32, apr 2017.