The Journal of Supercomputing (2022) 78:43–71
https://doi.org/10.1007/s11227-021-03819-z

FlexSched: Efficient scheduling techniques for concurrent kernel execution on GPUs

Bernabé López-Albelda, · Francisco M. Castro[1] · José M. Gonzalez-Linares[1] · Nicolás Guil[1]

Accepted: 15 April 2021 / Published online: 19 May 2021
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2021

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. In this scenario, concurrent kernel execution can leverage the use of devices by co-executing kernels having a different or complementary resource utilization profile. A paramount issue in concurrent kernel execution on GPU is to obtain a suitable distribution of streaming multiproccessor (SM) resources among co-executing kernels to fulfill different scheduling aims. In this work, we present a software scheduler, named *FlexSched*, that employs a run-time mechanism with low overhead to perform intra-SM cooperative thread arrays (a.k.a. thread block) allocation of co-executing kernels. It also implements a productive online profiling mechanism that allows dynamically changing kernels resource assignation attending to the instant performance achieved for co-running kernels. An important characteristic of our approach is that off-line kernel analysis to establish the best resource assignment of co-located kernels is not required. Thus, it can run in any system where new applications must be immediately scheduled. Using a set of nine applications (13 kernels), we show our approach improves the co-execution performance of recent slicing methods. Moreover, our approach obtains a co-execution speedup of 1.40× while slicing method just achieves 1.29×. In addition, we test *FlexSched* in a real scheduling scenario where new applications are launched as soon as GPU resources become available. In this scenario, *FlexSched* reduces the average overall execution time by a factor of 1.25× with respect to the time obtained when proprietary hardware (HyperQ) is employed. Finally, *FlexSched* is also used to implement scheduling policies that guarantee maximum turnaround time for latency sensitive applications while achieving high resource use through kernel co-execution.

Keywords GPU scheduling · Concurrent kernel execution · Online profiling · Simultaneous multikernel

✉ Nicolás Guil
nguil@uma.es

Extended author information available on the last page of the article

1 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 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 such as reducing the makespan of a set of kernels taking advantage of concurrent kernel execution capabilities available in devices [1, 2], improving time-sharing execution [3] or providing priority-based kernel execution developing soft-real time schedulers [4]. As it has been pointed out by several previous works [5–10], many GPU kernels do not scale well because some hardware resources become saturated. This way, they have tried to increase GPU utilization applying a concurrent kernel execution scheme, running simultaneously in the device kernels with complementary resource needs.

Concurrent kernel execution (CKE) can be exploited using software queues (called *streams* in the Compute Unified Device Architecture Software Development Kit (CUDA SDK) [11]) which allow programmers to launch independent kernels. These software queues are mapped onto hardware queues from which kernels are scheduled. The kernel scheduling unit in these devices is called Cooperative Thread Array (CTA), also known as thread block, and the CTA hardware scheduler allocates CTAs to streaming multiprocessors (SM). However, no API is supplied to perform a precise CTA allocation. In fact, the hardware scheduler follows a FIFO order [12] to schedule CTAs. In the case the number of CTAs of a kernel exceeds the capability of the GPU (a very common situation), CTAs of the following kernel has to wait until previous kernel start to free resources. Consequently, most of the time the first kernel run alone and only at the end of its execution some CTAs of the second kernel can be concurrently executed. Another drawback of the CTA hardware scheduler is it does not guarantee a specific mapping of CTAs onto SMs. Thus, although two o more kernels with a small number of CTAs could be concurrently executed, the attained performance could be low as it strongly depends on the CTA allocation mapping [3, 8]. Therefore, all general software solutions for CKE require to add some software support into the kernel code, typically some simple modifications in either source or PTX (Parallel Thread Execution) code [3, 6, 8, 9], to allow concurrent execution even when kernels launch many CTAs, and to set a specific CTA allocation on SMs.

Schedulers can benefit of a CKE implementation to, for example, deal with different types of kernels like batch or priority kernels. For batch kernels, the scheduling policy aim could be to improve GPU utilization to increase kernel execution rate [5, 7], while for priority kernels, more sophisticated scheduling policies could be implemented to simultaneously guarantee Quality of Service (QoS) and GPU utilization. Besides, a preemption technique should be included to provide QoS. There are some research proposals of hardware-based preemption mechanisms, [13, 14], but they are not available in current GPUs. More recently, other

works have proposed the use of software-based preemption mechanisms like [10, 15]. The former proposes a preemption mechanism that requires the modification of kernel source code and it is restricted to the execution of just one kernel in the GPU. The latter also introduces a co-execution scheme where CTAs on a SM can be partially preempted. However, optimum resource assignment to kernels is not discussed, which limits the performance achieved by the scheduler.

These scheduling policies could benefit of a CKE model that provided some guidance on how GPU resources should be assigned to the concurrent kernels because, as it was previously indicated, the performance achieved depends largely on the distribution of these resources and the interference among co-executing kernels. Some works have proposed performance models that requires exhaustive off-line profiling before kernel co-location [6, 8, 16, 17], but this approach has a limited utility in a data center where new applications are constantly scheduled. Other works schedule kernels taking into account a resources reservation policy [9] but, without a correspondence between assigned resources and achieved throughput, the scheduler cannot take performance-oriented decisions.

In this work, we propose a flexible mechanism to co-execute kernels on GPUs. Our mechanism can carry out a specific allocation of CTAs to SMs and can take online decisions to modify current assignation in order to increase the global performance of batch applications and/or fulfill QoS requirements of latency sensitive (LS) applications co-located with batch kernels. Thus, the main contributions of this study are the following:

- A kernel launcher that performs an efficient co-location scheme of CTAs belonging to different kernels on SMs. We have called it *rtSMK* as it is a run-time mechanism inspired in the simultaneous multikernel (SMK) approach [18], which proposes a hardware based fine-grain dynamic sharing mechanism that fully utilizes resources within a SM by exploiting heterogeneity of different kernels.
- A performance model based on a productive online profiler that finds the best resource assignment for any pair of concurrent kernels without the need for an off-line kernel analysis.
- A scheduler, named *FlexSched*, that employs the performance model to dynamically change GPU resources assignation with low overhead. The scheduler is able to manage an arbitrary number of applications and takes CTA re-mapping decisions each time a new kernel must be launched.

Our approach is tested using 13 kernels belonging to 9 applications. Firstly, *rtSMK* is favorably compared to *cCUDA*, the most recent slicing method for kernel co-execution [17]. Secondly, *FlexSched* is tested in two scenarios.

In the first scenario, we consider that all applications are part of the same batch and they must be scheduled to obtain the best kernel throughput, that is, the scheduler policy is aimed to obtain the lowest overall execution time. As all applications belong to the same batch, they can be executed concurrently and we can test the behavior of the scheduler when new kernels must be launched and co-executed with kernels already running on the GPU. In the second scenario, there are some LS

applications running concurrently with an arbitrary number of batch applications. In this case, the objective of the scheduler is to fulfill the execution time requirements of the LS applications, while the remaining available resources are used efficiently to concurrently execute other batch applications. The rest of the paper is organized as follows. Sect. 2 introduces basic CKE features in modern GPUs and presents a motivating example with the foremost features of our scheduler. Section 3 presents the main modules that constitute our software approach. Then, Sect. 4 focuses on the necessary implementation details to develop an efficient software scheduler that takes advantage of kernel co-location. Extensive experiments of our proposal are conducted in Sect. 5, which show the role of the profiling process in the achieved performance. In Sect. 6, the key elements of previous works that propose both software and hardware kernel co-execution techniques are reviewed, and the main differences with our proposal are also pointed out. Finally, the main conclusions are presented in Sect. 7.

2 Background and motivation

The execution of kernels on a GPU may not scale well due to some architecture constraints. For instance, memory bound kernels can saturate the global memory bandwidth long before all possible CTAs are allocated to SMs. There are other causes for resources saturation, like pipeline stalls on Read After Write (RAW) dependencies (in compute bound kernels) or L1-cache trashing (in L1-cache sensitive kernels) [19]. One way to deal with this problem is to reduce the number of CTAs of the saturating kernels assigned to SMs and, to maintain high GPU utilization, schedule concurrently CTAs from another kernel with different resource needs.

Independent kernels can be scheduled to execute concurrently on a GPU using current APIs. The hardware CTA scheduler is responsible for launching the kernels but is not very flexible because it follows a *leftover policy* [5]. Thus, if a kernel creates more CTAs than those that can be simultaneously allocated on the GPU, no concurrent execution can be performed with the next scheduled kernel until the last finishing CTAs of the running kernel free up GPU resources. On the other hand, if two kernels have a small number of CTAs, the hardware scheduler is able to co-execute them, but the programmer cannot control their allocation mapping, which makes it difficult to develop fine-grained policies that exploit the available heterogeneity of the kernels. Alternatives to the current hardware scheduler policy have been studied employing simulators [18–22]. These works provide more precise control for CTA allocation, but cannot be used in a real system.

Some previous works have implemented CKE using a software approach, but this requires to modify the kernel original source code [5–10]. Nevertheless, they have also shown that the overhead introduced by these modifications is low and is compensated by the performance increase obtained by co-execution. More modern implementations also include a preemption mechanism that can be used for resource re-assignment. However, they typically model the co-execution performance carrying out an off-line profiling of the isolated execution of kernels [6, 8], predicting kernels Instructions Per Cycle (IPC) using demanding Markov chain models [23],

or just taking into account the number of CTAs assigned to each co-located kernel [9], but ignoring the impact in performance that the interference between co-running kernels competing for similar resources can produce. This way, all these approaches cannot be efficiently applied on a GPU server, where new and unknown applications are constantly scheduled.

In contrast, we have developed a CKE scheduler that can be employed on a GPU server where independent kernels are simultaneously ready to be launched. Our scheduler includes an online productive profiling method that helps to establish the best SM partitioning when concurrent execution starts, using a flexible mechanism that is able to manage partial kernel eviction and launching. For the sake of clarity, most frequent terminology employed in the paper is shown in Table 1.

Figure 1 shows a motivating example of our proposal where kernel co-execution is exploited by a scheduler implementing policies oriented to both high throughput and low latency. The figure displays the temporal schedule of four kernels and the relative distribution of GPU resources among them, i.e., the number of allocated CTAs for each kernel. Three of them are kernels belonging to a batch, that is, we assume they have the same priority and the objective is to co-execute them as fast as possible to obtain a high overall throughput, and the remaining one is a LS kernel that must be executed with some specific temporal constraints on turnaround time. Initially, only batch kernels are ready to run. The scheduler selects $K1$ and $K2$ kernels for co-execution and tries to maximize the co-execution performance to achieve a high kernels execution rate. The scheduler could have some basic kernel information, e.g., if kernels are either memory or compute bound, to help it to choose a pair of kernels but also a random selection could be carried out. Either way, the scheduler starts a profiling phase to obtain a good GPU resources partitioning between the two kernels. This phase is shown with a vertical shaded box starting a time t_1. During this phase, the performance of the co-execution is evaluated and, if required,

Table 1 Meaning of terms frequently used in the paper

Term	Meaning
Cooperative thread array (*CTA*)	Any group of threads that concurrently execute a task (also known as Thread Block)
Streaming multiprocessor (*SM*)	GPU physical component where CTAs are allocated for being executed
Basic scheduling unit (*BSU*)	Smaller launching unit managed by *FlexSched* consisting of persistent CTAs
Task	Computation performed by original CTA before kernel transformation
Simultaneous multikernel (*SMK*)	Co-execution mapping where CTAs of different kernels are co-located in the same SM. The same co-location scheme is replicated in all SMs
Co-execution configuration space (*CCS*)	All possible CTA co-location schemes of two co-running kernels in *FlexSched*. Each scheme exhausts at least one SM resource so that no more CTAs can be co-located
Co-execution configuration (*CC*)	Any of the possible co-location schemes of a CCS

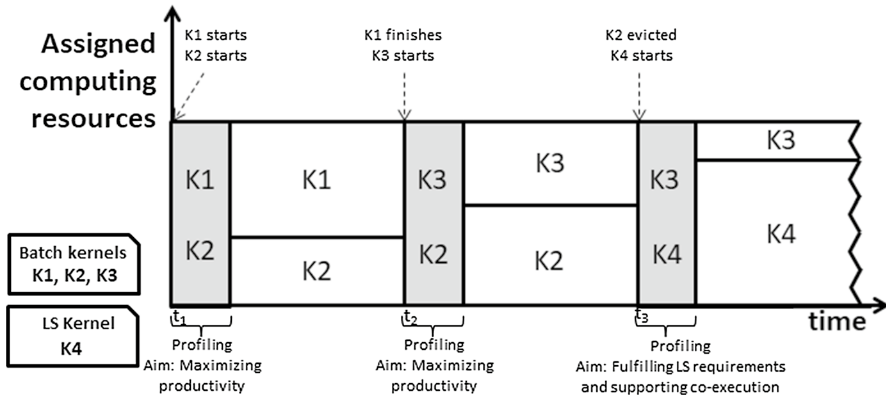


Fig. 1 Timeline showing an overview of the proposed approach for kernel co-execution. When a new kernel is scheduled, a phase of productive profiling starts to establish the best SM partition (shadow boxes). As it can be seen, the method is applied on the fly to an arbitrary number of new kernels that arrive with different scheduling aims

the resources distribution is changed. Notice that no specific CTA assignation is depicted during this phase because it varies over time. Once a good distribution is found, the profiling phase finishes and kernels continue their execution until one of them finishes. The height of the box representing each kernel indicates the relative proportion of assigned resources. At time t_2, $K1$ ends and the scheduler selects and launches $K3$. Once again, a profiling phase starts to find a good resource partitioning between $k2$ and $k3$. Then, $k2$ and $k3$ run until, at time t_3, a LS kernel, $K4$, becomes ready. Due to this event, the scheduler evicts one of the running kernels, $K2$ in this example, and launches $K4$. A new profiling phase starts but now the scheduler aim is to fulfill QoS requirements for $K4$ and, as a side effect, less resources are assigned to $K3$. Once an adequate assignment is found, the profiling phase ends and kernels continue running until next event.

As far as we know, our proposal is the first GPU kernel scheduler that implements an online productive profiling to find suitable co-execution configuration according to the required scheduling policy. In addition, as the profiling phase can be performed at any time, it can be also employed to change the current CTA allocation when new kernels are ready to be executed. This way, an arbitrary number of kernels can be efficiently scheduled by adapting CTA allocation to the requirements of running kernels.

3 Overview of FlexSched

FlexSched is a software approach, running on a host, that can schedule a set of GPU applications applying both performance and QoS oriented criteria. From simple modifications in the kernel original source code, it develops flexible mechanisms to run concurrent kernels including partial kernel eviction and launching. Also, an online productive profiler is implemented to obtain information of

co-executing kernels on the fly. This way, the scheduler can establish a suitable SM partitioning for concurrent kernels.

Profiling analysis and scheduling decisions are based on information collected during co-execution. In order to provide a flexible CKE scheme, the original kernel grid is transformed into a new grid formed by persistent CTAs [8]. Additionally, these CTAs are further grouped into Basic Scheduling Units (*BSU*), which are the smallest scheduling entities managed by our scheduler. The use of *BSUs* allows controlling precise CTA distribution on SMs during profiling and avoids costly kernel eviction and relaunching operations involving all pending CTAs as the number of running CTAS on a BSU is limited to one per SM. Figure 2 depicts a scheme of this approach. It can be observed that several *BSUs* of two kernels are running on the GPU, while *rtSMK*, scheduler and profiler modules are running on the CPU. The main features of these modules are the following:

- Profiler module: during profiling, the number of executed tasks of running kernels is constantly sampled through the bus connecting the host to the device, and an instantaneous task execution rate is calculated. Notice that *BSUs* belonging to the same kernel collaborate to increment the number of executed tasks of this kernel. Then, the calculated value is sent to the scheduling module so that changes in the current *BSUs* mapping can take place, if required.
- Scheduler module: using information from the performance model, which could be different depending on the scheduling aim, the scheduler instructs *rtSMK* to launch *BSUs* belonging to co-located kernels, monitors their progress and, in case of deviation with respect to the model prediction, makes a decision on CTA re-assignment. When a co-executing kernel finishes, this module also selects the new kernel to be launched and performs resource re-assignment if necessary. Notice that re-assignment is performed by evicting and (re-)launching *BSUs*.

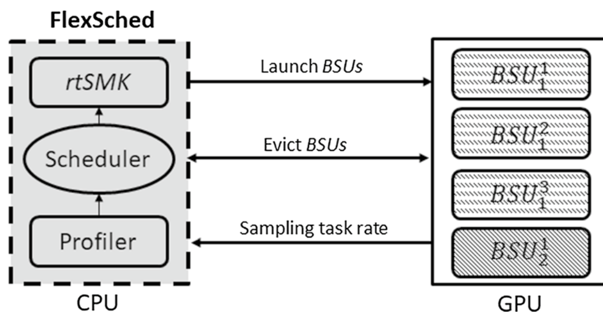


Fig. 2 Overview of the proposal. FlexSched consists of a *BSU* launcher (*rtSMK*), a Scheduler and a Profiler, which run on the CPU. An example of three BSUs (BSU_1^1, BSU_1^2 and BSU_1^3) from kernel 1 and 1 BSU (BSU_2^1) from kernel 2, running on the GPU, is also shown. The arrows connecting the CPU with the GPU represent launch and evict commands initiated by the launcher and the scheduler, respectively, while data transfer commands are managed by the profiler to obtain the instantaneous number of tasks executed per kernel. The profiler passes this information to the scheduler which assesses whether the current *BSU*s configuration is suitable

Thus, any running kernel is always progressing, except when all kernel *BSUs* are evicted.
- *rtSMK* organizes the kernel computation using *BSUs* and carries out *BSUs* launching according to scheduler commands.

4 Implementation details

In this section the four key elements of *FlexSched* implementation are explained. First, we show the modifications that must be applied to kernel original source code before it can be used by our scheduler. Next, the mechanism used to allocate CTAs to SMs , that is *rtSMK*, is introduced. Then, the set of possible CTA allocation configurations for two co-located kernels is discussed and, finally, the metric used during profiling phase is presented.

4.1 Kernel transformation

The implementation of the preemption mechanism requires a source to source transformation of original kernels, but this modification can be easily automatized as explained in [15, 16]. Firstly, the original kernel grid is modified to execute the kernel using persistent CTAs. Thus, our scheme can launch a number of CTAs that is equal or lower than $\max(CTA_{SM}) \cdot numSMs$, where $\max(CTA_{SM})$ is the maximum number of CTAs that can be simultaneous allocated in a SM and *numSMs* is the total number of SMs in the device. This transformation can be done automatically by a compiler, as a previous work [10] has shown.

The proposed transformation has two advantages regarding the preemption mechanism. On one hand, persistent CTAs typically execute several iterations where, in each iteration, the work of a single CTA of the original grid is computed. Each iteration is called a **task**. Thus, eviction can take place at the end of each iteration (task) and, when a previously evicted kernel is relaunched, each CTA would just resume the execution from the last completed iteration. On the other hand, the eviction mechanism works faster as, for most common grid sizes, only tens (at most a few hundreds in a modern Volta architecture) of persistent CTAs are active instead of several thousands.

A flexible mechanism for load distribution among CTAs has also been developed. Instead of assigning a specific computation to each CTA with a fixed mapping (like in [6]), CTAs obtain workload by atomically updating a common global memory variable at the start of each iteration, as it is also done in [8]. This global memory variable acts as a counter (starting from zero) that increments a tasks index and plays an important role during online profiling. This tasks execution ordering does not affect original kernel execution and benefits the preemption mechanism because 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 CTA during one iteration.

```
/************* GPU code *************/
Kernel_func( list_of_original_params , int MaxNumTask, int *State ,
    int *TaskCont) {
  while( true ) {
  // Check state (only one thread)
    if ( threadIdx.x == 0 ) {
      if ( *State == EVICTED )
        blockId = -1;
      else
        blockId = atomicAdd( TaskCont, 1 );
    }
    // synchronize block threads
    _sync_threads();
    // end checking: no more tasks or evict
    if ( blockId>=MaxNumTask || blockId==-1 )
      return;
    // Original kernel code follows here
  }
}
/********* CPU calling code *********/
Kernel_func<persist_grid_size> ( list_of_original_params ,
    MaxNumTask, State , TaskCont);
```

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

In Listing 1, the key kernel changes are indicated. As it can be observed, two new global memory variables are required for each kernel (line 2). One of these variables, named **State**, keeps the state of the GPU kernel and it can take three possible values: Ready, Running or Evicted. When a kernel has solved all its data dependencies, and its input data has been transferred from the host to the device, it takes the Ready state. Just before a kernel is launched, it is set to the Running state. The other memory variable, named **TaskCont**, contains the index of the next available task (initially set to 0). In addition, there is a new parameter, **MaxNumTask** (also in line 2), that contains the total number of tasks to execute and it is checked by GPU threads to terminate when all tasks have been computed.

Listing 1 also shows the code for the eviction mechanism and the task distribution strategy. The original CTA computation is enclosed within a while loop (line 3) 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 while loop iteration the CTA thread with id 0 reads the **State** variable (line 6). If the state has changed to Evicted, then 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 be also observed in line 9, new values of task id are obtained by thread 0 at each CTA iteration by executing an AtomicAdd instruction on **TaskCont** variable while the remaining block threads wait at a block synchronization instruction (line 12). After this barrier, all block threads read blockId and check (line 14) termination condition (all kernel tasks have been computed or **State** content has changed to Evicted). If condition is not fulfilled, CTA computes a new

task with the current blockId value. An additional modification must be also applied to the original code to change the indexation employed during computation. In kernels source code this indexation is typically implemented using CTA indexes while now the blockId content must be used. Anyway, these transformations can be automatically applied as shown in [15, 16].

4.2 Basic scheduling units

rtSMK is the *FlexSched* module in charge of scheduling kernels to execute concurrently using a software-based mechanism to distribute available GPU resources. Since our scheduler is forced to employ the CTA hardware scheduler, we take advantage of the round-robin CTA allocation policy it follows [6] to implement an intra-SM CTA allocation scheme, where CTAs of different kernels are allocated in the same SM. Thus, our approach defines a Basic Scheduling Unit (*BSU*) as a set of as many CTAs as the number of SMs in the GPU. This way, when a *BSU* is launched on an idle GPU, one CTA will be assigned to each SM. If another *BSU* is launched, a new CTA will be running in each SM provided there are enough resources for them. As a result, two CTAs, which belong to different BSUs, will be executing at each SM. Notice the use of *BSU* leads to a CTA distribution scheme similar to SMK [7, 18], which exploits kernel heterogeneity allowing fine-grain sharing by multiple kernels within each SM.

Figure 3 shows an example where three *BSUs* belonging to two kernels are scheduled on a GPU with four SMs. In that figure, BSU_k identifies a *BSU* of kernel *k*. *BSUs* that belong to the same kernel collaborate with each other to perform kernel computation, since they share the counter **TasksId** (see Listing 1). This is the case for the two instances of BSU_1 in Fig. 3. However, they can be evicted independently because each *BSU* has its own variable **State**. Therefore, an efficient and responsive dynamic CTA allocation can be carried out since an evicted *BSU* can leave room for *BSUs* from another kernel, as long as the new *BSUs* can be executed with resources released by the just evicted *BSU*.

We define the granularity of a *BSU* as the number of CTAs that will be launched at each SM. Thus, a *BSU* with a granularity value of one, as those represented in Fig. 3, will launch *numSMs* CTAs, where *numSMs* is the number of streaming multiprocessors in the device. Nevertheless, granularity can be higher than one to reduce *BSUs*

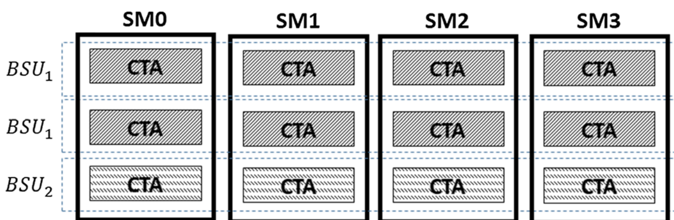


Fig. 3 Result of launching three *BSU*s on a GPU with four SMs. When a *BSU* is launched, a CTA is allocated at each SM as indicated by dotted box. BSU_1 and BSU_2 identifies *BSU*s belonging to kernel k1 and k2, respectively. Thus, two *BSU*s of k1 and one *BSU* of k2 are running

management overhead in kernels with a high number of persistent CTAs per SM. Thus, the granularity of a *BSU* can range from 1 to the maximum number of persistent CTAs per multiprocessor. Our scheduler is able to manage *BSUs* with different granularity attending to runtime decisions.

The use of BSU allows the distribution of kernel CTAs into independent scheduling units (streams in CUDA terminology). Thus, when kernel resource assignment must be reduced, not all the running CTAs of a kernel need to be evicted but just only those belonging to a BSU. This way, eviction delay is reduced. Similarly, when more resources must be assigned to a kernel, running BSUs can be kept and only the new required ones are launched.

The use of BSU allows the kernel CTAs to be distributed into independent scheduling units (streams in CUDA terminology). Thus, when kernel resource assignment needs to be reduced, it is not necessary to evict all running CTAs from a kernel, but just only those belonging to a BSU. This way, eviction delay is reduced. Similarly, when more resources must be allocated to a kernel, currently running BSUs can be kept and only the new ones that are required are launched.

4.3 Co-execution configuration space

A major issue that must be solved to obtain benefits from concurrent execution is to establish a good GPU resource partitioning for co-running kernels. As a result of this partitioning, CTAs belonging to different kernels are co-executed on the device. Our CTA allocation approach follows a SMK scheme, thus intra-SM co-execution is performed and all SMs have identical allocation layout. However, as CTAs of a scheduled *BSU* must be persistent, resource occupancy requirements must be checked before launching a new *BSU* to ensure new CTAs can be correctly assigned to the SMs. Valid numbers of scheduled *BSUs* from different kernels can be calculated in advance knowing the following features regarding CTAs of concurrent kernels: number of threads per CTA, number of required registers per thread and shared memory space per CTA. All these values are known at the code compilation stage. Thus, using these features and taking into account the GPU architecture capability, it is straightforward to establish a *valid co-execution configuration*, that is, the number of *BSUs* of co-running kernels that can be launched. More precisely, a valid configuration must fulfill two conditions: all CTAs must be persistent and no more CTAs can be allocated. The latter condition guarantees that a valid configuration must exhaust at least one of the SM resources so that no more CTAs can be allocated. We name the set of valid co-execution configurations as *co-execution configuration space*, *CCS*. This space is composed of an ordered set of tuples that indicates the number of CTAs that are co-executed on each SM for two kernels, as follows:

$$CCS(K_1, K_2) = \{(|BSU_1^1|, |BSU_2^1|), \dots, (|BSU_1^n|, |BSU_2^n|)\}, \tag{1}$$

where K_1 and K_2 are the concurrent kernels and $|BSU_i^j|$ indicates the number of launched BSUs of kernel K_i in the jth co-execution configuration. To facilitate the search of a good configuration, the set of configurations is sorted by increasing values of $|BSU_1^j|$, that is, $|BSU_1^j|$ takes the lowest value of concurrent CTAs per SM of

$K1$ for configuration 1. The number of CTAs increases as we move to higher configuration indexes. Thus, $|BSU_1^n|$ represents the highest number of CTAs per SM of kernel K_1 than can be executed concurrently with CTAs of K_2. It is easy to deduce that the values taken by $|BSU_2^j|$ have just the opposite behavior, that is, they decrease as the configuration index increases. An example is presented to illustrate all these concepts. Let us assume the resources needed by each kernel permit allocating a maximum of 8 CTAs per SM for each kernel. However, when they are executed concurrently, the SM resources must be shared, resulting in seven possible configurations, $CCS = \{(1,7),(2,6),(3,5),(4,4),(5,3),(6,2),(7,1)\}$. For instance, in the first configuration one *BSU* of the first kernel and seven *BSUs* of the second kernel can be executed concurrently.

4.4 Co-execution profiling and scheduling

The fourth key element in *FlexSched* implementation is the profiling phase. During this phase, the *CCS* of two kernels is explored to find the best co-execution configuration attending to some specific scheduling policy. More precisely, a profiler determines the task execution rate, *TER*, achieved by the running *BSUs* of a kernel. *TER* value of kernel i is calculated using the following expression:

$$TER_i = \frac{TaskCont_i}{T_i}, \tag{2}$$

where $TaskCont_i$ is the task counter shared by all *BSUs* of kernel K_i , and T_i is the time elapsed since kernel started to run.

Using *TER* value, the scheduler can apply a throughput-oriented policy to compare the performance achieved by two configurations of the *CCS*. This is done by calculating the weighted speedup, WS_{TER}, between *TER* values of kernels K_1 and K_2 for two different configurations, as follows:

$$WS_{TER} = \frac{1}{2} \cdot \left(\frac{TER(|BSU_1^j|)}{TER(|BSU_1^k|)|} + \frac{TER(|BSU_2^j|)}{TER(|BSU_2^k|)} \right), \tag{3}$$

where $TER(|BSU_1^j|)$ and $TER(|BSU_2^j|)$ indicate the task execution rates of kernel K_1 and K_2 when a specific number of *BSUs*, given by the expression $|BSU_1^j|$, of K_1 are concurrently executed with $|BSU_2^j|$ ones of K_2. Taking into account that the values in the numerator and denominator correspond to two different co-execution configurations, k and j, the previous expression defines a metric that compares the speedup between them. Therefore, if the resulting value for WS_{TER} is higher than one, configuration j achieves a better global *TER* than configuration k. Similarly, if $WS_{TER} < 1$, configuration k obtains a better tasks execution rate. Thus, different co-execution configurations of a *CCS* can be compared to find the best one using a *hill climbing approach* as it will be shown in next section.

In conclusion, our proposal uses an online productive profiling to find the best co-execution configuration instead of previous software-based CKE schedulers, which are based on co-execution models that only considers GPU resource partitions for

kernel co-location [5, 9]. Furthermore, these works ignore the interference between kernels produced during co-execution, thus they are not able to establish a good CTA mapping that meets performance optimization criteria or turnaround time constrains.

5 Experimental results

Experiments have been conducted using different applications containing one or several kernels belonging to CUDA SDK [11], Rodinia [24], and Chai [25] benchmark suites. With these applications, we pursuit to build a real workload where compute and memory bound kernels can be tested. 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 2 shows the list of applications that we have used. Most of the applications have only one kernel, but two of them, namely Separable Convolution and Canny, are composed by two and four kernels, respectively. Kernels of both applications are executed in a pipeline fashion since the output of one kernel is the input to the next kernel.

As it has been shown in previous works [3, 7], the co-execution of kernels obtains good performance results when co-executing kernels make use of complementary resources. This way, if several kernels are ready to be launched, it is advisable to conduct a study in order to find suitable kernel pairings. Several authors [17, 26] have proposed different approaches for kernel classification. In this work, we use the Kernel Mix Intensity (*KMI*) value to represent the operational intensity of a GPU kernel. This value is obtained by dividing the number of computation and memory instructions executed by a kernel. Then, attending to the resulting KMI value, a kernel is classified as compute-bound (CB) or memory-bound (MB). The CUPTI

Table 2 Applications used in the experiments. Most applications consist of one kernel, although Separable Convolution and Canny are composed of two and four kernels, respectively. The third column shows the execution time of each kernel. The fourth and fifth columns indicate the most utilized resource per kernel and the category to which it belongs, respectively

Kernel Acronym	Application	Exec time (ms)	Category
HST256	Histogram	4.18	MB
BS	Black Scholes	4.47	MB
VA	Vector Addition	7.24	MB
SPMV	Sparse MV Mult.	10.60	MB
RED	Reduction	5.05	MB
CCONV	Separable	1.60	MB
RCONV	Convolution	1.45	MB
GCEDD	Canny	5.26	CB
SCEDD		9.65	CB
NCEDD		7.10	CB
HCEDD		2.15	CB
PF	Path Finder	6.77	CB
MM	Matrix Mult.	10.38	CB

library [27] is employed to obtain the number of these instructions by profiling different metrics. Concretely, we have used the Metric API of this library to calculate the number of integer and floating point instructions (half, single and double precision). In addition, we also get the number of dram memory transactions for both read and write instructions. Thus, the final expression for *KMI* is given by:

$$KMI = \frac{\#integer + \#half_float + \#single_float + \#double_float}{\#dram_read_trans + \#dram_write_trans} \tag{4}$$

The right-most column of Table 2 shows the category each kernel belongs to.

5.1 Kernel transformation overhead

The implementation of *FlexSched* that provides both *BSU*-based kernel execution and software-based preemption support, requires the transformation of the kernel original source code as explained in Sect. 4.1. This transformation can lead to an increase in overhead that we will measure with the following experiment. Let T_t and T_o be the total execution time of the transformed and original kernels, respectively. Then, the overhead incurred by this transformation can be computed by the next expression, $O_t = \frac{T_t}{T_o}$. Second column in Table 3 shows the values of O_t obtained for the 13 kernels.

Attending to the expression of the overhead, values higher than one are expected since the kernel transformation, as explained in Sect. 4.1, 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 CTAs. For instance, original kernel of RED performs two different operations. First, each CTA thread

Table 3 Analysis of kernel code transformation. Second column indicates the overhead incurred by kernel modification, with respect to the original code. Third column shows the average (Avg) and standard deviation (SD) values, in microseconds, of the preemption mechanism delay for each kernel, assuming that data transfers do not occur concurrently with kernel execution

Kernel	Transformation Overhead	Eviction Delay Avg ± SD (μs)
MM	0.96	72 ± 1
BS	1.00	81 ± 1
VA	1.00	76 ± 3
SPMV	1.06	71 ± 10
RED	0.90	28 ± 1
PF	1.00	90 ± 7
RCONV	1.01	45 ± 2
CCONV	1.00	23 ± 1
GCEDD	1.05	45 ± 3
SCEDD	1.05	38 ± 1
NCEDD	0.97	33 ± 2
HCEDD	1.04	32 ± 1
HST256	0.91	85 ± 3
Average	0.99	51

accumulates data content read from global memory, and then a reduction at CTA level is performed. As the transformed kernel has a lower number of CTAs, the reduction takes less time. HST256 overhead value can be explained in a similar way since there is also a final reduction. Focusing on kernels with overhead higher than one we can see that the execution time increases, at most, by 5%. Finally, the average overhead is 0.99, indicating that, globally, the overhead is negligible.

5.2 Eviction delay

A key aspect of any preemption implementation is its responsiveness to eviction commands. In our implementation, the scheduler evicts a kernel by asynchronously writing in the variable **State** of the *BSU*. This variable is allocated on GPU global memory as it was explained in Sect. 4.1. Any CTA in the *BSU* reads this variable before the execution of a new task and, attending to its content, computes the whole task or finishes. Consequently, there is a delay between the submission of the eviction command and the termination of the running kernel. This delay is directly related to how frequently a CTA reads the variable **State**. Thus, a reduction of the eviction delay requires more frequent memory reads which, however, increase the overhead of the atomic operation to update the variable **TaskCont**. As we are interested in keeping code modification as simple as possible, the minimum granularity is given by the computation enclosed within a CTA of the original kernel (see Listing 1). Nevertheless, this granularity could be increased if necessary by applying a coarsening technique to the CTA code.

We have measured the eviction delay of each kernel as in [10]. Eviction commands are sent at different times of the kernel execution, and the elapsed time until kernel stops is measured. This experiment have been repeated 50 times, and average and standard deviation values have been computed. Third column in Table 3 shows these values. All kernels have an eviction delay lower that 100 μs and small standard deviation values. Finally, the average eviction delay for all kernels is around 51 μs. This value allows developing scheduling policies requiring short eviction intervals.

5.3 Performance of CTA allocation scheme

As it was indicated in Fig. 2, *FlexSched* consists of three modules. In this section we are interested in evaluating the performance achieved by just one of the modules, *rtSMK*, which focuses on allocating CTAs on SMs following a SMK scheme. As *rtSMK* is the mechanism used by our approach to launch CTAs belonging to different kernels on GPU SMs, it can be compared with other recent method with similar functionality as *cCUDA* [17]. *cCUDA* presents an approach to kernel co-execution based on kernel slicing [3]. They employ two streams to launch multiples slices of both co-located kernels. This way, slices of both kernels can be concurrently executed as long as no slice exhausts all SMs resources. Notice that this technique also requires kernel modification as CTA index rectification must be applied to ensure correct kernel behavior. Also, original dimension of kernel grid must be passed as

a parameter. Kernel slicing incurs in an overhead caused by multiple launching of kernel slices. This overhead can be reduced by applying CTA coarsening.

As computation time of each co-executing kernel differs, the concurrent execution time, T_{CO}, considers only computation performed when both kernels execute simultaneously. Thus, in *rtSMK*, the number of tasks executed by the longest lasting kernel is annotated when the shortest one finishes. Then, the sequential execution time, T^{SE}, is calculated by launching sequentially that pair of kernels using the same number of tasks, that is, all the tasks for the shortest kernel and the annotated number of tasks for the longest one. Similarly, for *cCUDA*, we count the number of CTAs executed by both kernels when they are co-running, and the sequential execution time is calculated launching all CTAs of the shortest kernel and the annotated number of CTAs for the longest one. Notice that sequential kernel execution is carried out using different streams for each kernel. This way, kernel overlapping can still take place. The speedup achieved by two kernels, $K1$ and $K2$, when they are executed concurrently is given by the following expression:

$$S(K1, K2) = \frac{T^{SE}(K1, K2)}{T^{CO}(K1, K2)} \tag{5}$$

In Fig. 4a, a graph bar is used to represent the best speedup values achieved by pairs of memory-bound and computed-bound kernels for both methods. Speedup values are calculated exploring all *CCS* for each pair of kernels using a brute-force

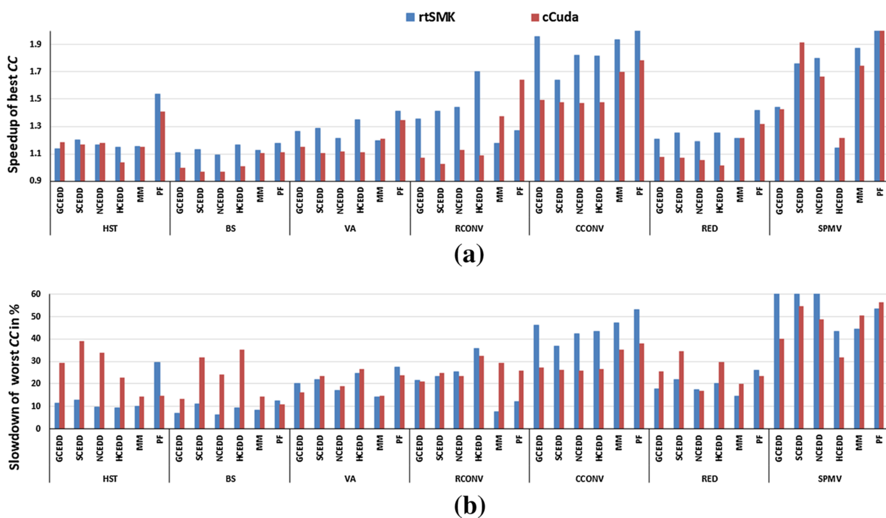


Fig. 4 Top graph shows the maximum speedups achieved by the best *CC* when CB and MB kernel pairs are co-executed using both our CTA co-location scheme, *rtSMK*, and the slicing method based on *cCUDA*. It can be observed that our technique is able to obtain better scores in most cases. Thus, the average speedup obtained by *rtSMK* is 1.40 while *cCUDA* just gets 1.29. The bottom graph displays the slowdown with respect to the best speedup when the *CC* configuration obtaining the lower speedup is used. The average slowdown for both methods is around 29% which indicates that the selection of a good *CC* is paramount to obtain a good performance

approach and the co-execution configuration (CC) with the highest score is taken. In order to perform a fair comparison, the same values for CTA coarsening are employed in both methods.

It can be observed from Fig. 4a that our proposed CTA allocation scheme, *rtSMK*, obtains speedups ranging from 1.1 to 2.1, which confirms the advantages of kernel co-execution. Comparing maximum speedup values of *rtSMK* and *cCUDA*, we can see that most times our approach improves *cCUDA* results. Three are the issues that explains this behavior:

- *cCUDA* incurs in an overhead each time a new slice is launched. Thus, if slice execution time is short and the number of launched slices is high, the overhead is not negligible. An example of this behavior can be observed during the co-execution of RED and HCEDD. The co-execution time of a RED slice is just 12μs while slice launching time is 7μs. Thus, the execution overhead of this kernel is high and just a speedup of 0.95 is achieved. Notice that *rtSMK* only launches *BSUs* once so it is not affected by this overhead, obtaining a speedup of 1.23.
- *cCUDA* also presents efficiency problems when slices finish. As successive slices of a kernel are launched into the same stream, a new slice has to wait for the total completion of the previous slice before starting to run. However, during slice finalization phase, GPU resources are wasted since some slice CTAs are still running while others have already finished.
- *cCUDA* dynamically assigns CTAs to SMs using a greedy approach that can lead to less efficient mappings while *rtSMK* constantly monitors and controls CTAs assignment.

As a result, if we calculate the average speedup for all kernels pairings, *rtSMK* obtains a score of 1.40 while *cCUDA* achieves 1.29. Thus, *rtSMK* is competitive when compared with techniques based on kernel slicing. However, the most important advantage of our method lies in the included preemption mechanism, which allows it to find the best co-execution configuration during a productive profiling phase, as already explained in Sect. 4.4. On the contrary, *cCUDA* launches the execution commands for kernel slices in a row and they are stored in a hardware command queue associated to the corresponding stream. In this queue, slices are sequentially executed until all commands have finished using a non-preemptive scheduling.

Figure 4b points out the importance of finding the best *CC*. In this graph, the slowdown of the worst *CC* with respect to the best one is calculated using the following expression:

$$SD(K1, K2) = \frac{S_M(K1, K2) - S_m(K1, K2)}{S_M(K1, K2)} \cdot 100, \tag{6}$$

where $S_M(K1, K2)$ and $S_m(K1, K2)$ are the maximum (M) and minimum (m) speedups obtained by the best and worst *CC*, respectively. Slowdown values range from 6% to 60% in *rtSMK* and from 10% to 40% in *cCUDA*. The average slowdown factor for *rtSMK* and *cCUDA* is around 29%. These results show that low performance values can be obtained if *CC* is not adequate. In the following section we show how

FlexSched is able to localize the best or close to the best *CC* for any co-executing kernels.

5.4 *FlexSched* performance-oriented scheduling

We have developed *FlexSched*, a scheduler that can execute concurrently a set of applications while increasing the throughput. In this section, we explain in detail the heuristics used by *FlexSched* during the profiling phase to find the co-execution configuration that achieves the best performance. Next, the heuristic is validated in a real scenario where several applications are executed, and the results are compared with those obtained by the GPU hardware scheduler (HyperQ).

5.4.1 Profiling phase

During the first stage of *FlexSched* profiling phase, the *CCS* of co-executing kernels is obtained. This information can be gathered off-line, since it only requires details about kernels resources usage which are setup during kernel compilation. More precisely, this information consists of the number of threads per CTA, the use of shared memory per CTA, and the number of registers used per thread. Therefore, any valid configuration in *CCS* is characterized by the maximum number of persistent blocks that can be scheduled concurrently on a SM for both co-executing kernels, taking into account the previous information.

Once the *CCS* is obtained, *FlexSched* can look for a suitable configuration as shown in the example in Fig. 5, where a VA/RCONV kernel pair is analyzed. The *CCS* of this pairing consists of seven co-execution configurations, specifically $CCS = \{(1,7),(2,6),(3,5),(4,4),(5,3),(6,2),(7,1)\}$, where the first and second element in each configuration indicate the number of VA and RCONV *BSUs*, respectively. *FlexSched* goes through this *CCS*, launching each configuration to obtain *TER* values, and compares consecutive configurations using the weighted speedup,

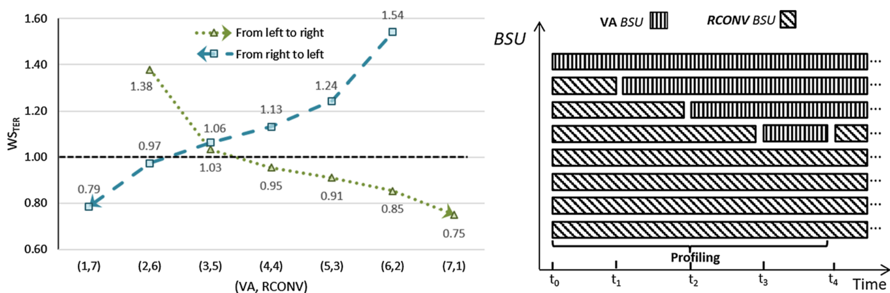


Fig. 5 Left plot shows weighted speedups for different co-execution configurations of VA and RCONV. Dotted green line corresponds to going through *CCS* from left to right, while dashed blue line represents going in the opposite direction. Right diagram shows the profiling phase until the best configuration is found. First, one VA *BSU* and seven RCONV *BSU*s are launched at time t_0. Then, a RCONV *BSU* is evicted and a new VA *BSU* is launched at time t_1. This procedure continues until the best configuration is found

WS_{TER}, calculated using Eq. 3. For clarity, all the WS_{TER} values going through *CCS* from left to right (green dotted line) and in the opposite direction (blue dashed line) are shown in the left plot in Fig. 5. For instance, the leftmost point in the green dotted line indicates that *TER* when going from configuration (1, 7) to configuration (2, 6) is 1.38. As explained in Sect. 4.4, a value of *TER* greater that one means that *TER* value for configuration (3, 5) is better than the corresponding one of configuration (2, 6), which in turn is better than *TER* of configuration (1, 7). However, *TER* of configuration (4, 4) is worse than *TER* of configuration (3, 5), and the following configurations also have WS_{TER} values below one, therefore the best configuration for VA/RCONV is (3, 5). Notice that the same conclusion can be obtained when going through *CCS* in the opposite direction. Thus, the best configuration can be found by looking for the last tuple with a WS_{TER} value higher that one, as it obtains the greatest *TER* value of all *CCS* tuples.

FlexSched does not need to obtain all the WS_{TER} values in both directions. Instead, it goes through *CCS* computing on the fly the values it needs until the *one* mark is crossed. In the example in Fig. 5, *FlexSched*, following the green dotted line, takes the first configuration in *CCS*, (1, 7), and launches one *BSU* of VA and seven *BSU*s of RCONV at time t_0. After a sampling time, *TER* values for each kernel are read and their WS_{TER} is computed. Then, *FlexSched* tests the performance of configuration (2, 6) by evicting one RCONV *BSU* and launching a new VA *BSU* at time t_1. This process is repeated until configuration (4, 4) is reached because WS_{TER} drops below 1. Afterward, *FlexSched* evicts a RCONV *BSU* and launches a VA *BSU* to go back to (3, 5) configuration at time t_4, terminating the profiling phase. This configuration is kept until one of the two kernels finishes. In the worst case for this simple search method, the profile should evaluate $nconf - 1$ configurations, being *nconf* the number of tuples within *CCS*. However, computational complexity could be further reduced using a *divide-and-conquer* approach for the search process.

Figure 6 shows the situation when RCONV finishes at instant t_5. When *FlexSched* detects the finalization of RCONV, it selects another kernel, MM in this example, and obtains the *CCS* for this pairing. This *CCS* already has a configuration with three *BSU*s of VA, (5, 3), so *FlexSched* launches five *BSU*s of MM at time t_5. Then, at time t_6, calculates *TER* values for this co-executing configuration, and launches the

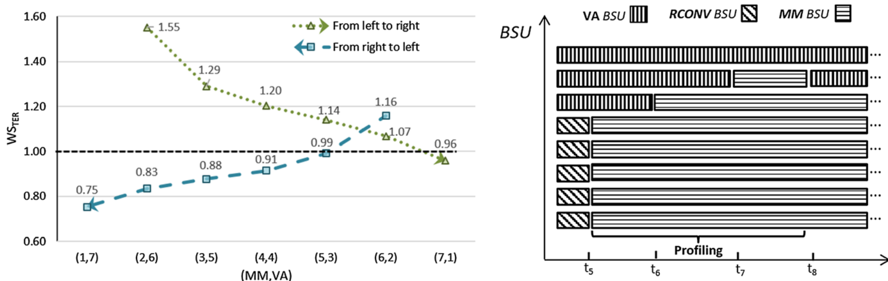


Fig. 6 Profiling phase when RCONV finishes. Our scheduler selects MM and launches 5 *BSUs* to fill the room left by RCONV. Then, the profiler goes through the new *CCS* to find the best co-execution configuration for VA and MM

next configuration. There are two options, configuration (4, 4) or configuration (6, 2). Let us suppose configuration (6, 2) is selected, thus one VA *BSU* is evicted and a new MM *BSU* is launched. *TER* values are computed at time t_7, and WS_{TER} between configurations (5, 3) and (6, 2) is calculated. It has a value greater than 1, concretely 1.07, thus (6, 2) is better and the next configuration, (7, 1), can be checked. At time t_8, WS_{TER} is computed, obtaining a value below 1 (0.96), so the previous configuration is re-established and maintained until one of the two kernels finishes. At time t_6 *FlexSched* could have chosen configuration (4, 4), but then at time t_7 it would have obtained $WS_{TER} = 0.91$. As it is below 1, *FlexSched* would discard this direction and select configuration (6, 2) (it evicts two VA *BSU*s and launches two MM *BSU*s). Then, the search process continues in a similar way as it was previously indicated.

For some kernel pairs the WS_{TER} plots can be very different from the previous examples, with values always above or below one. For instance, Fig. 7 shows the values obtained for PF/SPMV pair. From left to right they are always above 1 and, consequently, from right to left are below 1. Thus, if *FlexSched* starts searching from the left, configuration (1, 14); then, it will launch all configurations until reaching the last configuration. On the other hand, if *FlexSched* starts searching from the right, configuration (7, 2), then it will stop searching after checking that the next configuration, (6, 4), has a WS_{TER} value below 1 (0.69).

We have conducted an experiment to show that the configuration obtained with this heuristic is optimal or near-optimal in all cases. First, we have obtained the optimal configurations for all pairings using the brute-force method introduced in Sect. 5.3. Then, performance results for these optimal configurations have been compared with the performance results of the configurations selected by *FlexSched*. The average relative difference is below 1%, that is, *FlexSched* can find most of time the best, or close to the best, configuration. The highest differences are obtained for co-execution configurations with low performance. For example, the maximum error reaches 6% when RCONV and RED are co-located. Both kernels are memory bound and the best co-execution configuration obtains a speedup of only 1.0 w.r.t. sequential execution while *FlexSched* finds a configuration that reaches 0.94.

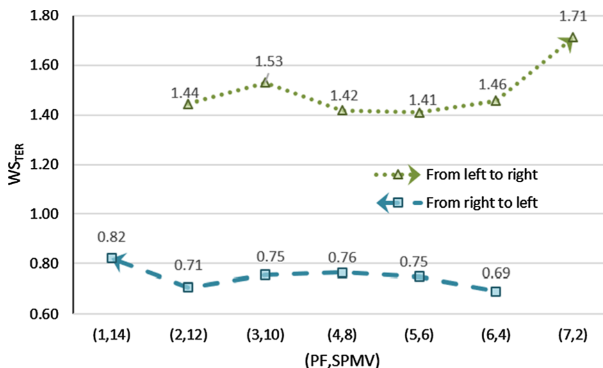


Fig. 7 WS_{TER} values taken for the *CCS* of PF/SPMV. All values in the left to right direction are higher that one

5.4.2 Profiling overhead

During the profiling phase, *FlexSched* dynamically changes the *BSU* mapping in order to find a co-execution configuration with the highest performance. However, the flexibility offered by the online profiling scheme has a temporal cost as it must be applied every time a *CC* must be evaluated. In this section, the overhead of this profiling phase is evaluated.

Let $T_{prof}(K_1, K_2)$ be the time taken by the profiling phase for a specific kernel pair (K_1, K_2). It depends on the number of configurations that must be evaluated before reaching the best one. In addition, the time taken for each configuration requires to evict several *BSUs* and launch new ones. Furthermore, as it was shown in Table 3, the eviction time is different for each kernel. Consequently, a precise evaluation of the profiling overhead requires to evaluate all possible kernel pairs.

Since this profiling phase is employed to find a good co-execution configuration that improves the kernels sequential execution, the overhead can be computed by comparing $T_{prof}(K_1, K_2)$ with the sequential execution of kernels, computing the same number of tasks per kernel as those executed during profiling. That is, the profiling overhead $\mathcal{O}_{prof}(K_1, K_2)$ is given by

$$\mathcal{O}_{prof}(K_1, K_2) = \frac{Tprof(K_1, K_2)}{T_s(K_1 \mid task_1) + T_s(K_2 \mid task_2)}, \tag{7}$$

where $T_s(K_i \mid task_i)$ is the time taken by kernel K_i to execute $task_i$ tasks sequentially, and $task_1$ and $task_2$ are the number of tasks executed during the profiling phase by kernels K_1 and K_2, respectively.

Left side in Fig. 8 shows, using a boxplot, the statistical distribution of $\mathcal{O}_{prof}$ for all possible pairs of kernels (78 values have been collected). It can be observed that, although some kernel pairs obtain values greater than one, most of them are below that value. Thus, in most cases, the time taken during the profiling phase is less than the time taken by the sequential execution of a similar number of tasks. In fact, the

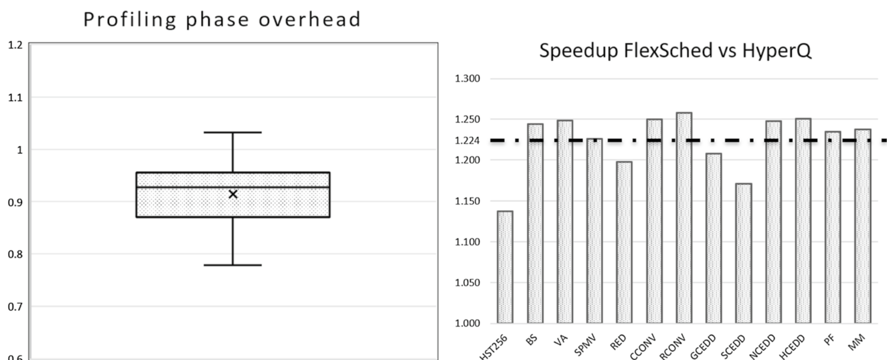


Fig. 8 Left: boxplot of profiling phase overhead for all possible pairs of kernels. Overhead is computed with respect to the sequential execution of the same number of tasks. Right: average speedups of the total execution time using *FlexSched* with respect to HyperQ, by first launching each of the 13 kernels

average and the median values for $\mathcal{O}_{prof}$ are 0.913 and 0.926, respectively. An important conclusion that can be extracted from this experiment is that, during the profiling phase, despite the partial *BSU*s eviction and launch operations, the overhead of these operations is hidden thanks to the concurrent execution of others *BSUs*, which results in a execution time that is typically shorter than the equivalent computation performed by the sequential execution of kernels.

5.4.3 Concurrent execution of applications

In the next experiment we compare *FlexSched* with the native hardware scheduling mechanism provided by Nvidia (HyperQ). As we are interested in analyzing the behavior of our scheduler in a real scenario, the experiments execute the 9 applications (13 kernels) indicated in Table 2. Notice that other works, as [8, 9], only run two kernels. We also assume all applications are ready to run (all required input data have been transferred from host to device) before scheduling starts.

In this experiment *FlexSched* begins by selecting one of the 13 kernels, then it examines its category as indicated in Table 2, and selects randomly a partner among the kernels in the other categories. Next, during the profiling phase, it looks for a good co-execution configuration as explained in Sect. 5.4.2. When either of the two kernels ends, a new kernel that belongs to another category is selected, if available. Otherwise, a kernel from the same category is chosen. Any new kernel schedule involves a profiling phase to look for a good configuration. This process continues until all applications are finished. The selection of co-executing kernels is random, therefore we run the experiment 1000 times to perform a statistical analysis of the scheduler behavior. Furthermore, we repeat this experiment using each of the 13 kernels as the first kernel to account for differences.

The same experiment has been executed using HyperQ, a feature in modern GPUs since Kepler architecture, where several hardware-managed queues can schedule kernel commands from different streams. Thus, each application is launched at the same time in a different stream. In order to make a fair comparison, each experiment is run 1000 times again, and the streams are launched in the same order in which the applications were executed in the corresponding experiment with *FlexSched*. For each run, we have computed the speedup of the total execution time using *FlexSched* with respect to HyperQ. Right side in Fig. 8 shows the average of these values when using each kernel as the first kernel, and a horizontal line is drawn to indicate the total average. The heuristic in *FlexSched* obtains different speedups depending on the first kernel as different co-execution pairs are chosen, but it exploits the concurrent execution capabilities of the GPU much better than HyperQ, achieving an average speedup of 1.224. HyperQ always obtains worst performance, since it can only overlap the execution of a new kernel when the previous one is finishing.

5.5 *FlexSched* latency-oriented scheduling

In a data processing center, applications can have different requirements; for example, there can be, at the same time, a batch of throughput-oriented applications

together with latency-sensitive (LS) applications which must be executed within a maximum response time. A naive strategy to schedule one of these applications is to evict all the currently running kernels and then launch the LS application. However, this approach will waste GPU resources that could be used to co-execute other kernels while the LS application temporal restrictions are fulfilled.

FlexSched can be extended to support co-execution of LS applications with a batch of throughput-oriented applications. Different from [9], that can only impose restrictions regarding the number of CTAs assigned to LS applications, we can employ a more useful temporal constraint based on the minimum value of *TER* allowed. Thus, when an LS application is co-executed with other kernels, *FlexSched* looks, during the profiling phase, for configurations that ensure this value of *TER*. More specifically, *FlexSched* begins by selecting the configuration in *CCS* which assigns more *BSU*s to the LS application. Then, it computes its *TER* value and, if it is greater than the minimum allowed, the next configuration in *CCS* is selected. When the task rate restriction is not fulfilled, the previous configuration is restored and the profiling phase finishes.

The experiment to test the behavior of *FlexSched* with LS applications consists in executing concurrently a kernel with temporal constraints with a batch kernel. We define several slowdown factors of the concurrent *TER* with respect to the sequential *TER* of that application when running alone. These factors vary from 0.5 to 0.1, in steps of 0.1. For instance, a slowdown factor of 0.5 means that the minimum allowed value of *TER* of the LS application must be at least half of value of *TER* when running alone. Furthermore, we have selected CEDD as the LS application, and we have executed it concurrently with other kernels. In case, the other kernels finish before the LS application, they are launched again. Table 4 shows the results when CEDD is executed concurrently with several memory-bound or compute-bound kernels. For each slowdown factor, we record the percentage of CTAs that belong to the other kernel that can be co-executed with the CEDD kernel while fulfilling the LS performance requirements. Thus, when MM is run concurrently with CEDD imposing a maximum slowdown factor of 0.5, the 31% of CTAs running on the GPU belong to MM. Average values for CTAs percentage are also shown for each slowdown factor. As expected, when the slowdown factor decreases, the number of the other kernel CTAs that can be co-executed with CEDD decreases since more

Table 4 Percentage of CTAs per SM used by non-latency sensitive kernels with respect to the total available CTAs for concurrent execution with CEDD kernel

		Slowdown factor				
		0.5	0.4	0.3	0.2	0.1
Kernel	MM	31	23	13	11	1
	VA	28	21	12	7	1
	BS	18	12	7	1	1
	RED	37	28	23	12	1
	PF	40	33	22	12	1
	HST256	33	22	18	12	1
	Average	31	23	16	9	1

resources must be assigned to the LS application. However, even for small slowdown values, our scheduler is able to assign resources to the other kernel. In conclusion, *FlexSched* can be used to schedule applications that must meet strict QoS requirements and, in addition, launch other kernels to achieve high use of resources.

6 Related work

Modern GPUs, starting with NVIDIA Fermi devices, have hardware support for concurrent kernel execution. This hardware support allows the use of software queues, called *streams* in CUDA terminology, to launch independent kernels that could be concurrently executed on the GPU. However, there is no software-level mechanism to select how kernel CTAs are distributed on the GPU. In fact, if a kernel requires to execute more CTAs than the maximum number of persistent thread blocks, it cannot run concurrently with other kernel except when it is finishing and resources become available (leftover policy). Thus, to take a real advantage, concurrent execution using a software approach typically requires to modify the PTX source code. Other authors have proposed CKE hardware-based approaches employing a simulator, typically GPUSim [28], to add hardware mechanisms to perform specific CTA allocation mappings and to support co-located kernel scheduling. As we are presenting a software approach, we will focus on previous works developing software techniques. However, we also show the most relevant papers that propose hardware modifications.

6.1 Software approaches for kernel co-execution

Many researches have proposed run-time support to take advantage of CKE for improving the GPU hardware utilization and to develop different co-scheduling polices. Thus, an early work, [5], proposed to transform kernels into elastics ones. This way, the number of CTAs required by the original kernel is reduced and kept below the maximum number of resident CTAs per SM. Consequently, GPU resources could be made available for CTAs of other elastic kernels and kernels could be spatially co-executed. One important problem of this technique is it cannot be applied to kernels using shared memory. In addition, several CTA assignation policies were defined but all of them are static and based on models that only takes into account the use of GPU resources extracted during kernel compilation instead of information obtained from kernel co-execution. Consequently, the real achieved performance could be low. Other approach, [3, 17] proposed a solution to improve time-sharing execution by slicing kernels. Each slice computes a small number of blocks and, this way, it can co-execute with a slice of other kernel. The main problem of this approach is that large kernels can lead to severe launch overhead as kernels must be divided into many slices.

The use of co-scheduling has been also explored when priority aware policies are implemented. Thus, in [7], they propose the co-execution of applications kernels taking into account priorities. Since this work does not implement kernel preemption

techniques, the scheduler only can schedule a higher priority kernel when a lower one has finished. Thus, this scheme can have slow response time when long kernels are executing.

Other authors have proposed the building of macrokernels to execute kernels concurrently [6]. This macrokernel combines the code of two kernels and requires the utilization of additional memory arrays that identify the schedule pattern to orchestrate kernel execution and the index for the identification of CTAs and threads. Authors also employ an heuristic to establish the best order to execute a set of kernels. This heuristic needs to know the execution time of kernels with exclusive GPU access and co-executes different number CTA kernels with dummy CTAs. Thus, the proposed scheme of kernels combination is not flexible and it cannot be applied on-the-fly because the profiling needed is very costly.

Another work [8], has used SM-Centric kernel code transformation so that two kernels can be co-scheduled on GPU. Those kernels are allocated in different SMs using a *filling-retreating* scheme. Nevertheless, the best SM partition is found during an off-line stage which consists of two phases where many co-execution combinations must be tried. In addition, the method can evict CTAs of co-running kernels but it does not implement any method to re-start evicted CTAs. Other paper [15], focuses on the development of an efficient preemption mechanism. Thus, applying spatial preemption, kernel co-location can take place. However, the developed scheduler only considers the execution of two kernels with different priorities. In [9], physical SM are conceptually divided into small capacity slices where kernel CTAs are executed. Slices are further grouped in a software entity called CapSM. Thus, a CapSM can group CTAs running on different SMs. The proposed scheduler is based on resource reservation alone, that is, the number of CTAs a kernel will launch. Furthermore, co-execution kernel interference is no taken into account, which hampers the development of precise dynamic scheduling policies based on either performance or QoS. In addition, due to the way CapSM are organized (CTAs of a CapSM could be located in different SMs) a precise co-executing model cannot be developed.

Other recent work, [16], implements QoS policy, without using a preemption mechanism, by developing a model to predict the performance degradation of latency sensitive applications on accelerators due to application co-location. The model requires exhaustive off-line profiling which restricts the utilization of the model in real situations.

6.2 Hardware approaches for kernel co-execution

One of the earliest works proposing hardware modifications to support CKE compares spatial multitasking, that is, kernel co-location, with cooperative multitasking, that is, temporal multitasking, and shows the advantages of the former one [29]. Later, [19] proposes an intra-SM CTA allocation policy for CKE. This work focus on reducing the resource fragmentation when CTAs of different kernels are allocated in a SM and also proposes a resource partitioning method that maximizes the throughput. However, it requires an off-line phase to take IPC values for each kernel

varying the number of CTAs per SM, which can take a long number of cycles. In order to save time, they propose to carry out these measurements with several kernels executing concurrently. In [18], a similar CTA allocation strategy is proposed but preemption is added. This way a new arriving kernel can be allocated by previously evicting CTAs belonging to the running kernel. Thanks to the preemption mechanism, the paper implements strategies for improving overall throughput while being fair to co-executing kernels. In [20], a productive mechanism to establish the best CTA mapping for two concurrent kernels is proposed. In order to explore in a fast way all possible configurations, it proposes a heuristic where different CTA mappings are assigned to different groups of SMs. Analyzing IPC values of SMs, a iterative process gradually finds the best co-location configuration. To reduce the overhead produced by the process of finding the best solution, in [22] a trained predictor of slowdown for co-located kernels is proposed. The predictor collects statistics of hardware events of two co-running kernels and estimates their slowdown. Also, other works [21, 30], have focused on the memory subsystem to increase the performance of co-located kernels. The former tries to avoid starvation of compute-intensive applications when they run together with memory-intensive ones. They observed that, in this configuration, the latency of global memory accesses of compute-intensive kernels grows as their memory requests are queued behind many previous requests emitted by memory-intensive kernels. Then, they develop methods to rein the memory accesses for memory-intensive kernels. The latter work increases system throughput and fairness using a metric that takes into account both DRAM bandwidth and cache miss rates. Finally, a recent work [31] employs an inter-SM (also called simultaneous multithreading) CTA mapping for kernel co-execution. The paper shows a method to classify kernels (memory bound or compute bound) while kernels are co-running using GPU counters for both the number of memory accesses and DRAM row buffer hits. In addition, an estimation of single kernel performance can be carried out which permits evaluating co-execution performance with respect to sequential execution.

7 Conclusion

In this work, we have presented *FlexSched*, a software scheduler for GPU applications that takes advantage of CKE to implement scheduling policies aimed at maximizing application execution throughput or meeting QoS application requirements such as maximum turnaround time. An important feature of *FlexSched*, which makes it different from software schedulers proposed in previous works, is the use of a productive on-line profiling. During profiling, our scheduler employs a heuristic that compares different co-execution configurations to find a suitable CTA allocation scheme that fulfills the scheduling requirements: throughput or QoS. Thanks to this flexible schedule scheme, *FlexSched* can be applied to real situations where unknown applications must be immediately executed on a GPU, in contrast with previous works that need to perform costly off-line profiling.

Extensive experiments have been conducted to test *FlexSched* using 9 applications with 13 kernels in total. First, to compare our proposal with other approaches

for kernel co-execution, we have tested *rtSMK*, the *FlexSched* CTA launcher, and a recent slicing method, *cCUDA*. Results show that our approach obtains a co-execution speedup of 1.40× while the slicing method just achieves 1.29. Also we have conducted an experiment that shows the speedup achieved by kernel co-execution strongly depends on the chosen co-execution configuration. Then, we have shown that the heuristic employed by the *FlexSched* scheduler is able to find co-execution configurations that achieve the highest performance in most cases. Moreover, the average performance loss with respect to the best possible configuration is less than 1%. *FlexSched* has been also compared with the hardware scheduler using HyperQ. The experiment, in contrast to other works where only a specific co-execution configuration was used, simultaneously considers all applications. Thus, during the experiment, *FlexSched* has looked for suitable co-execution configurations as new kernels were launched besides executing them. Nevertheless, *FlexSched* has improved kernel throughput by a factor of 1.22 with respect to HyperQ. In addition, experiments have been carried out with latency sensitive applications that require a strict turnaround time. In this case, *FlexSched* has calculated how many GPU resources could be assigned to batch applications while meeting the requirements of latency sensitive application subject to different slowdown factors. Thus, experiments with slowdown factors of 0.5×, 0.6×, 0.7× and 0.8× show that batch kernels have still available the 31%, 23%, 16% and 9%, respectively, of GPU resources for co-execution. Once again, we have demonstrated that the productive profiling process is crucial to leverage the use of resources in the GPU while QoS constrains are fulfilled.

In future work, we plan to add GPU hardware support to *FlexSched* to make unnecessary the current kernel transformation, to efficiently implement the requited CTA mapping, and to include mechanisms to automatically find the best co-execution configuration.

Acknowledgements This work has been supported by the Junta de Andalucía of Spain (P18-FR-3130) and the Ministry of Education of Spain (PID2019-105396RB-I00). We also thank Nvidia for hardware donations within its GPU Grant Program.

References

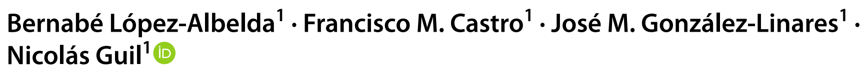
1. Lázaro-Muñoz AJ, González-Linares J, Gómez-Luna J, Guil N (2017) A tasks reordering model to reduce transfers overhead on GPUs. J Parallel Distrib Comput 109:258–271. https://doi.org/10.1016/j.jpdc.2017.06.015
2. Wende F, Cordes F, Steinke T (2012) On improving the performance of multi-threaded CUDA applications with concurrent kernel execution by kernel reordering. In: Symposium on Application Accelerators in High-Performance Computing, pp 74–83. https://doi.org/10.1109/SAAHPC.2012.12
3. Zhong J, He B (2014) Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling. IEEE Trans Parallel Distrib Syst 25(6):1522–1532. https://doi.org/10.1109/TPDS.2013.257
4. Kato S, Lakshmanan K, Rajkumar R, Ishikawa Y (2011) Timegraph: Gpu scheduling for real-time multi-tasking environments. In: Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference, pp 2–2
5. Pai S, Thazhuthaveetil MJ, Govindarajan R (2013) Improving GPGPU concurrency with elastic kernels, ASPLOS '13 407. https://doi.org/10.1145/2451116.2451160

6. Liang Y, Huynh HP, Rupnow K, Goh RSM, Chen D (2015) Efficient GPU spatial-temporal multitasking. IEEE Trans Parallel Distrib Syst 26(3):748–760. https://doi.org/10.1109/TPDS.2014.2313342
7. Lee H, Al Faruque MA (2014) Gpu-evr: Run-time event based real-time scheduling framework on gpgpu platform. In: Proceedings of the Conference on Design, Automation & Test in Europe, DATE '14, pp 220:1–220:6
8. Wu B, Chen G, Li D, Shen X, Vetter J (2015) Enabling and exploiting flexible task assignment on gpu through sm-centric program transformations. In: Proceedings of the 29th ACM on International Conference on Supercomputing, ICS '15, ACM, New York, NY, USA, pp 119–130. https://doi.org/10.1145/2751205.2751213
9. Yu C, Bai Y, Yang H, Cheng K, Gu Y, Luan Z, Qian D (2018) Smguard: a flexible and fine-grained resource management framework for gpus. IEEE Trans Parallel Distrib Syst 29(12):2849–2862
10. Chen G, Zhao Y, Shen X, Zhou H (2017) 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, PPoPP '17, pp 3–16. https://doi.org/10.1145/3018743.3018748
11. NVIDIA, Cuda sdk code samples (2018). https://www.nvidia.com/object/cuda_get_samples_3.html
12. Pai S, Govindarajan R, Thazhuthaveetil MJ (2014) Preemptive thread block scheduling with online structural runtime prediction for concurrent gpgpu kernels. In: 2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT), pp 483–484. https://doi.org/10.1145/2628071.2628117
13. Tanasic I, Gelado I, Cabezas J, Ramirez A, Navarro N, Valero M (2014) Enabling preemptive multiprogramming on gpus. In: 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA), pp 193–204. https://doi.org/10.1109/ISCA.2014.6853208
14. Park JJK, Park Y, Mahlke S (2015) Chimera: Collaborative preemption for multitasking on a shared gpu. In: ACM SIGARCH Computer Architecture News, ASPLOS '15, pp 593–606. https://doi.org/10.1145/2694344.2694346
15. Wu B, Liu X, Zhou X, Jiang C (2017) FLEP: Enabling flexible and efficient preemption on GPUs. ACM SIGPLAN Notices 52(4):483–496
16. Chen Q, Yang H, Guo M, Kannan RS, Mars J, Tang L (2017) Prophet: Precise QoS prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers. ACM SIGOPS Oper Syst Rev 51(2):17–32
17. Shekofteh SK, Noori H, Naghibzadeh M, Fröning H, Yazdi HS (2020) ccuda: Effective co-scheduling of concurrent kernels on gpus. IEEE Trans Parallel Distrib Syst 31(4):766–778. https://doi.org/10.1109/TPDS.2019.2944602
18. Wang Z, Yang J, Melhem R, Childers B, Zhang Y, Guo M (2016) Simultaneous multikernel gpu: Multi-tasking throughput processors via fine-grained sharing. In: 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp 358–369. https://doi.org/10.1109/HPCA.2016.7446078
19. Xu Q, Jeon H, Kim K, Ro WW, Annavaram M (2016) Warped-slicer: Efficient intra-sm slicing through dynamic resource partitioning for gpu multiprogramming. In: 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), ISCA '16, pp 230–242. https://doi.org/10.1109/ISCA.2016.29
20. Park JJK, Park Y, Mahlke S (2017) Dynamic resource management for efficient utilization of multitasking gpus. In: Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '17, ACM, New York, NY, USA, pp 527–540. https://doi.org/10.1145/3037697.3037707
21. Dai H, Lin Z, Li C, Zhao C, Wang F, Zheng N, Zhou H (2018) Accelerate gpu concurrent kernel execution by mitigating memory pipeline stalls. In: 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp 208–220. https://doi.org/10.1109/HPCA.2018.00027
22. Zhao W, Chen Q, Lin H, Zhang J, Leng J, Li C, Zheng W, Li L, Guo M (2019) Themis: Predicting and reining in application-level slowdown on spatial multitasking gpus. In: 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp 653–663. https://doi.org/10.1109/IPDPS.2019.00074
23. Zhong J, He B (2014) Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling. IEEE Trans Parallel Distrib Syst 25(6):1522–1532. https://doi.org/10.1109/TPDS.2013.257

24. Che S, Boyer M, Meng J, Tarjan D, Sheaffer JW, Lee SH, Skadron K (2009) Rodinia: A benchmark suite for heterogeneous computing. In: Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on, pp 44–54. https://doi.org/10.1109/IISWC.2009.5306797
25. Gómez-Luna J, Hajj IE, Chang L, García-Floreszx V, de Gonzalo SG, Jablin TB, Peña AJ, Hwu W (2017) Chai: Collaborative heterogeneous applications for integrated-architectures. In: ISPASS, pp 43–54. https://doi.org/10.1109/ISPASS.2017.7975269
26. Carvalho P, Quintanilla Cruz R, Drummond L, Bentes C, Clua E, Cataldo E, Marzulo L, Kernel concurrency opportunities based on gpu benchmarks characterization, Cluster Computing 23. https://doi.org/10.1007/s10586-018-02901-1
27. NVIDIA, CUPTI: User guide. Version: DA-05679-001_v11e.1 (2020)
28. Bakhoda A, Yuan GL, Fung WWL, Wong H, Aamodt TM (2009) Analyzing cuda workloads using a detailed gpu simulator. In: 2009 IEEE International Symposium on Performance Analysis of Systems and Software, pp 163–174. https://doi.org/10.1109/ISPASS.2009.4919648
29. Adriaens JT, Compton K, Kim NS, Schulte MJ (2012) The case for gpgpu spatial multitasking. In: IEEE International Symposium on High-Performance Comp Architecture, pp 1–12. https://doi.org/10.1109/HPCA.2012.6168946
30. Wang H, Luo F, Ibrahim M, Kayiran O, Jog A (2018) Efficient and fair multi-programming in gpus via effective bandwidth management. In: 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp 247–258. https://doi.org/10.1109/HPCA.2018.00030
31. Zhao X, Jahre M, Eeckhout L (2020) Hsm: A hybrid slowdown model for multitasking gpus. In: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '20, Association for Computing Machinery, New York, NY, USA, pp 1371–1385. https://doi.org/10.1145/3373376.3378457

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Authors and Affiliations



Bernabé López-Albelda[1] · Francisco M. Castro[1] · José M. González-Linares[1] · Nicolás Guil[1]

Bernabé López-Albelda
blopez@uma.es

Francisco M. Castro
fcastro@uma.es

José M. González-Linares
jgl@uma.es

[1] Department of Computer Architecture, University of Málaga, Campus de Teatinos, 29071 Málaga, Spain