PRUDA: An API for Time and Space Predictible Programming in NVDIA GPUs using CUDA

Reyyan Tekin, Houssam-Eddine ZAHAF, Giuseppe Lipari
Univ. Lille, CNRS, Centrale Lille, UMR 9189 - CRIStAL, Lille, France
{firstname.familyname}@univ-lille.fr

ABSTRACT

Recent computing platforms combine CPUs with different types of accelerators such as Graphical Processing Units (*GPUs*) to cope with the increasing computation power needed by complex real-time applications. NVIDIA GPUs are compound of hundreds of computing elements called *CUDA cores*, to achieve fast computations for parallel applications.

However, GPUs are not designed to support real-time execution, as their main goal is to achieve maximum throughput for their resources. Supporting real-time execution on NVIDIA GPUs involves not only achieving timely predictable calculations but also to optimize the CUDA cores usage.

In this work, we present the design and the implementation of *PRUDA* (Predictable Real-time CUDA), a programming platform to manage the GPU resources, therefore decide when and where a real-time task is executed. PRUDA is written in C and provides different mechanisms to manage the task priorities and allocation on the GPU. It provides tools to help a designer to properly implement real-time schedulers on the top of CUDA.

1. INTRODUCTION

Many real-time applications such as computer vision, surveillance systems, etc. demand complex processing on a large amount of data. Classical multiprocessor platforms combining only CPUs are not able to satisfy the real-time requirements of such systems as they require computing capabilities in the order of teraflops.

Recently, NVIDIA have provided computing platforms combining CPUs with different types of specialized computing unit such as GPUs, Deep Learning Accelerating (DLA), etc., on the same chip. These platforms can offer suitable solutions to meet deadlines for emerging complex real-time applications. However, the complexity of the software, combined with the complexity of the hardware architecture, makes it difficult to analyse the temporal behavior of such systems. Moreover, these accelerators are not fundamentally designed to execute real-time tasks. Therefore, they do not provide proper hardware and software mechanisms to schedule real-time tasks.

Several works [1, 3, 5] have attacked the problem of providing support to real-time systems onto GPUs from different perspectives. Kato et al. have proposed platforms (TimeGraph and RGEM) for non-preemptive scheduling for graphical tasks in GPU [5], [4]. Authors in [1] tried to study how a GPU takes scheduling decisions based on benchmarking of the Jetson TX2 platform. Capodieci et al. in [2] modified the proprietary NVIDIA driver to implement an event-driven scheduler allowing to use fine grain preemption levels provided by recent GPUs under different policies such as EDF and fixed priority. GPUSync [3] is a platform able to control scheduling within the GPU using locks. The work in [2] has closed sources whereas GPUSync [3] platform does not provide tools to freely implement real-time schedulers. In both works, GPU is used as a single core platform.

Contributions..

In this work, we develop *PRUDA*, a platform that implements different strategies to control real-time execution within a GPU using CUDA. PRUDA provides a control over priorities and task allocation and parallel execution within the same GPU at the same time. The different primitives of PRUDA allows implementing several real-time scheduling policies using different strategies. The platform is currently under active development: we are working on implementing special verison of EDF (GRUB) and Fixed priority scheduling policies with PRUDA.

The remainder of this paper is structured as follows. GPU architecture and its *known* scheduling mechanisms are details in Section 2. Section 3 presents the task and architecture models. We reserve Section 4 to define how priorities and allocations are controlled within a GPU using our platform. In Section 5, we overview the implementation of real-time schedulers using PRUDA. We draw our conclusions in Section 7.

2. GPU PROGRAMMING AND PRUDA PRIMITIVES

A GPU is compound of one or more *streaming multiprocessors* (SMs) and one or more *copy engines* (CEs). Streaming multiprocessors are able to achieve computations (*kernels*), whereas copy engines execute memory copy operations between different memory spaces. Programming the GPU requires dividing parallel computations into several grids, and each grid to several blocks. A block is a set of multiple threads. A GPU can be programmed using generic platforms such OpenCL or proprietary independent APIs. We use CUDA, a NVIDIA proprietary platform, to have a tight control on SMs and CEs in the C programming language and using the NVIDIA compiler.

When a kernel is invoked by CPU code, it submits commands to the GPU. How and when commands are executed, is hidden by constructors for intellectual property concerns. Authors in [1] have tried to reveal some *GPU scheduling secrets* by benchmarking a Jetson TX2 (abbreviated TX2 in the rest of this paper). It is compound of 6 ARM-based CPU

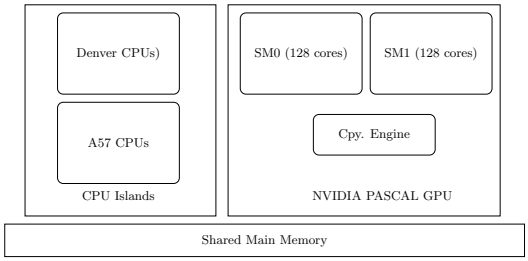

Figure 1: Jetson TX2 Architecture

cores, along with an integrated NVIDIA PASCAL-based GPU as shown in Figure 1, all running onto Ubuntu. The GPU in the TX2 is compound of 256 Cuda cores, divided into two SMs and one copy engine. CPUs and GPU share the same memory module. From a programming perspective, one may either allocate two separate memory spaces for CPU and GPU using malloc and CudaMalloc primitives respectively. The programmer may use a memory space visible logically by the CPU and the GPU called CUDA unified memory (even for discrete GPUs), therefore no memory copies are needed between CPU and GPU tasks such memory spaces (buffers) allocated using the CudaMallocManaged primitive. The current version of PRUDA supports CUDA unified memory to avoid dealing with memory copy operations, as it will be shown in PRUDA architecture. An extension to separate memory spaces is under development and will be soon available.

Typical Cuda programs are organized in the same way. First, memory is allocated both on CPU and GPU. Further, memory copies are operated between CPU and GPU. Then, the GPU kernel is launched, and finally results are copied back to the CPU by memory copy operations.

Regarding kernel execution within the GPU, authors in [1] affirm that all threads of any block are executed by only one SM, however different blocks of the same kernel may be executed on different SMs. In Figure 2, the green kernel is executed on both SM0 and SM1, the red SM is executed only on SM0. The kernel execution order and mechanisms are driven by internal closed-source NVIDIA drivers (in our case of study). A PRUDA user may obtain the SM where a given block/thread is executing by using the pruda_get_sm() function. PRUDA allows also enforcing the allocation of a given kernel to a specific SM by using PRUDA function pruda_allocate_to_sm(int sm_id), where the sm_id is the id of the target streaming multiprocessor. Implementation details about how these functions work can be found in the PRUDA description section.

To enforce an execution order between different kernels, we use a specific data structure, called *Cuda Stream*. A cuda

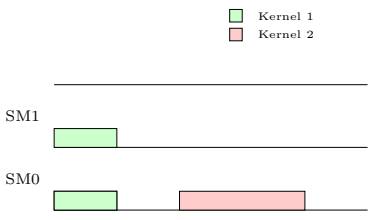

Figure 2: Example of Kernel scheduling in GPU

```
void *pruda_task(void * arg) {
  struct timespec_t next;
  p_kernel_t *pk = (p_kernel_t *)(arg);
  while (1) {
    // memory copy operation
    clock\_gettime(CLOCK\_REALTIME, &next);

    pruda_subscribe(pk->kernel,p->priority)

    timespec_addto(next, pk->T);
    clock_nanosleep(CLOCK_REALTIME, 0, &next, 0);
  }
}
```

Figure 3: Pseudo-code of PRUDA task

stream has a FIFO behavior. Therefore, kernels submitted to a Cuda stream are executed one after the other in a **sequential** fashion. Therefore, synchronization between two consecutive kernels is implicitly achieved. This property will be used later to implement non preemptive EDF and fixed priority real-time scheduling policies.

In Cuda, the user may define several streams. A priority might be set between different streams. Therefore, if a stream A has a higher priority than stream B, all kernels of A are meant to execute before kernels that are submitted to B. If a kernel in B is executing, and a kernel is activated on A, the GPU might preempt the kernel of B, to execute the kernel of A according to the GPU preemption level (we will show this behaviour in our benchmarks). We highlight that fine-grain preemption capabilities are available in NVIDIA GPUs starting from the PASCAL architecture. For example, if a preemption is set at a block level, preemption will be achieved when all already executing blocks finish their execution. Recent VOLTA GPUs allow even finer preemption levels.

Even if it is possible to create more than 2 streams, only two levels of priority are available in the Jetson TX2 platform. These properties will be used later to approximate EDF and fixed priority preemptive scheduling policies.

Other PRUDA functions will be detailed later.

3. SYSTEM MODEL

In this paper, we are only interested in GPU programming and scheduling. While this paper provides real-time support to GPUs, we do not provide any schedulability analysis yet, the analysis is work in progress.

We assume that all tasks in the system are programmed used PRUDA, therefore only PRUDA tasks are in concurrence in GPU. Each task τ_i is characterized by its deadline D_i and its period T_i. Tasks are strictly periodic, therefore the exact time between two successive activations of task τ_i is equal to T_i. The j^{th} instance of task τ_i must finish it execution no later than $\mathsf{T}_i \times j + \mathsf{D}_i$, otherwise it *misses* its deadline. The task may be scheduled using fixed priority, therefore it may be characterized by parameter priority P_i. From the implementation perspective, each PRUDA tasks is a instance of a periodic CPU thread as shown in the algorithm of Figure 3.

The PRUDA task starts by parsing the kernel parameters which are the kernel code, priority, deadline and period.

Further it starts the periodic task behavior. The task get the current time and computes the next instance activation time next. Later, the GPU job is registered in the correct (according to the desired scheduling policy) GPU run-queue (see PRUDA architecture in Figure 4). Once the PRUDA CPU thread launched the kernel, it sleeps until the next activation. Another scheduling entity checks the run-queue state and schedule the highest priority tasks first according to (i) one of the strategies details into the next section and (ii) to the desired scheduling policy.

The memory copy operation line achieves memory copies. This operation may need to copy several buffers from CPU to GPU and vice-versa. The current version of the platform use Cuda unified memory, therefore memory coherency is achieved automatically by the NVIDIA Driver.

The GPU may be scheduled as a single core platform or a parallel platform where each streaming multiprocessor is an independent core by the mean of the PRUDA __allocate_to_sm$(\cdots)$ function. The allocation to a given SM is achieved by testing if the task is in the correct SM, if yes, the computation is achieved, otherwise the thread on the *wrong* SM is killed.

4. TEMPORAL AND SPATIAL CONTROL OF PRUDA TASKS ON GPU

Our platform integrates several strategies to implement scheduling decisions. These strategies have different performances and overheads.

4.1 Single-stream strategy

The first strategy, called *single-stream* , uses one Cuda stream to enforce kernel scheduling decision. The scheduler uses three queues: a task queue (tq) which contains all PRUDA tasks list; an active kernels queue rq which contains the active PRUDA jobs; and the stream queue sq, which contains kernels that will be submitted to GPU. When a kernel is activated, it is added to the *correct* active kernel queue rq via the pruda_subscribe$(\cdots)$ function. Further, if Cuda stream queue sq is empty, it is moved from the rq to sq if it is the highest priority job according to the given scheduling policy using pruda_resched function.

As only one Cuda stream is used, once the pruda task is executing, it can not be preempted by another higher priority task, therefore only non preemptive scheduling algorithms can be implemented using this strategy. However, we would like to highlight that we allow pruda user to abort the current kernel under execution by calling pruda_abort() function.

This strategy is simple and easy to implement. It provides an implicit synchronization between active tasks, i.e. if task B is in the stream queue while A is running, B will wait until A finishes its execution before starting without overlapping. However, the use of this strategy involves reserving all the GPU resources (both SMs) for a single pruda task at a time, even if this task is not using all GPU cores, therefore resource are wasted. In the next strategies, we will show how to overcome these limitations.

4.2 Multiple stream: preemption enabling

In the second strategy, called *multiple streams*, PRUDA creates multiple streams to take scheduling decisions, allowing concurrent kernel execution on GPUs and preemption.

First, we recall that the TX2 allows only two priority levels. Therefore, we create only two streams: one with high priority and the other with low priority. The queue of the high priority stream is denoted by h-sq, the second stream queue is denoted by l-sq. We recall that using several streams allow asynchronous and concurrent execution between the two streams, however within the same stream the execution is always FIFO.

When a task is active, it is added to the correct ready-task queue rq. Further, the scheduler checks one of the following situations:

1. h-sq $= \emptyset \wedge$ l-sq $= \emptyset$: the scheduler will allocate the task to the l-sq queue, therefore the task will be submitted *immediately* to the GPU.

2. h-sq $= \emptyset \wedge$ l-sq $\neq \emptyset$: the scheduler checks that the activated task has a higher priority than the task in l-sq. If yes, the task is inserted into the high priority queue h-sq, therefore it preempts the task in the l-sq if possible. Otherwise, no scheduling decision are taken.

According to the scheduling decisions mechanism described in the text above, only one preemption is allowed when a task is already in execution. For example, if a task C arrives after B has preempted A, task C must wait until B finishes even if it is the highest priority active job. We are currently developing schedulability analysis for such limited preemption system. We would like also to highlight that preempted tasks, will continue to use GPU resources if the high priority task is not using *all* of the GPU resources.

Even if this strategy solves preemption limitations of the previous one, it is more complex. It uses also a GPU as a single core. In the next section, we use each SM in the GPU as a single processor allowing parallel execution within the GPU. We highlight also that the preemption at instruction level can not be guaranteed as the later is decided by the NVIDIA closed internals. However, we ensure that the preemption can be achieved at block boundaries, therefore the worst preemption cost is in the order of the block execution.

4.3 SMs as cores strategy

The third strategy uses the GPU in similar way as the previous one; therefore two streams are created and with the same queue configuration. However, we allow tasks to call the function pruda_allocate_to_sm$(\cdots)$, thus using a GPU as a multiprocessor rather than a single core. We consider two types of pruda tasks : the ones that are allocated to a given SM and the other that are not (we consider that the PRUDA tasks, not calling the allocation function as tasks requiring the GPU exclusively).

In addition to the scheduling structures described for the previous strategy, this strategy uses one queue per SM : sm0-q and sm1-q. When a task is active, if it uses both SMs, no other task will be scheduled at the same time, therefore it will be added to l-sq or h-sq similarly as in the previous strategy. Otherwise, it uses a single SM and it is assigned to the correct SM queue. Later, the two job having the highest priority in sm0-q and sm1-q are scheduled first by being inserted in l-sq and h-sq. This allows parallel execution on both streaming multiprocessor. This strategy allows using the GPU of TX2 as a 2-core platform.

In fact, the allocation function tests if a given block-/thread is in the correct SM: if yes, it continues onward

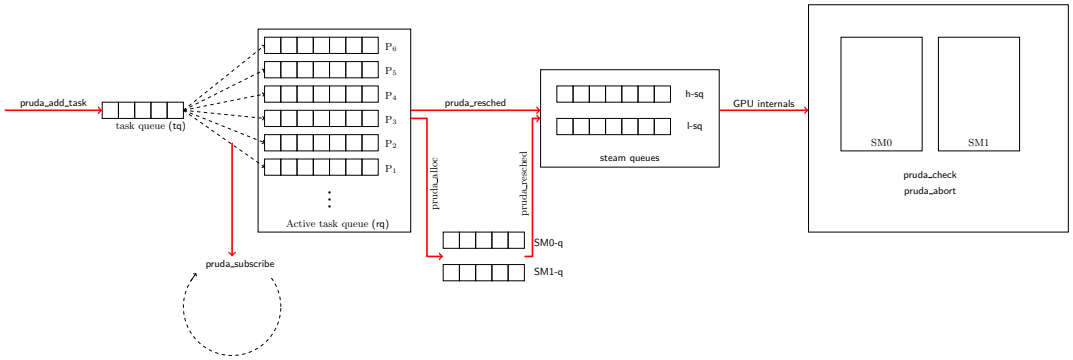

Figure 4: PRUDA global overview

execution, otherwise it exits. Therefore, the user has either to take that into account when using the block and thread indexes, or he/she must use new functions we provide to calculate indexes. The thread and block indexing mechanism we provide is simple but effective. The user is free to use the Cuda indexes, or our platform indexes, as long as there is no conflict. We highlight here that both of the previous strategies do not require any modification in the kernel code nor in the programming fashion (indexing). Although this method is more complex to implement than the two previous ones, it provides both temporal and spatial tasks execution control on GPUs. Analyzing the behavior of this final strategy is a challenging theoretical question, that is considered for future work.

5. REAL-TIME POLICIES USING PRUDA

Implementing real-time schedulers using PRUDA is simple. In fact, it requires implementing the pruda_subscribe function and the pruda_resched function. The goal of the first is to put the active task in the correct queue according to its priority. If the scheduling algorithm is fixed priority, it has to put it directly in the corresponding priority queue. If the algorithm is EDF, it requires calculating the priority and further inserting the task into the correct queue. The goal of the second function is to select which active task to select and in which Cuda stream queue it should be inserted, therefore to be submitted to the GPU. The user is also able to call pruda_abort to exit the execution of a given kernel to mix real-time with non real-time tasks if desired. The description of PRUDA provided in the current and the previous section is described in Figure 4. We highlight that pruda functions (except subscribe and resched) can be used even for non pruda tasks.

6. PRUDA API

All three strategies are integrated into the Pruda C++ API. We also have implemented for the single stream strategy both EDF and Fixed priority algorithms.

First of all, the API (Figure 4) requires the user to implement its kernel using CUDA. Further, the first step is to initialize the pruda scheduler by invoking function:

pruda_init_sched(method_t method, policy p);

where method is either SINGLESTREAM for the first strategy, MULTIPLESTREAMS for the second and for the MULTIPROC third. The policy P is the scheduling policy. The current version supports EDF or FP.

Once the scheduler has been initialized, we add kernels to the task queue tq by invoking function:

pruda_add_kernel(p_kernel_t kern, int gs, int bs, int p);

where kern is the a pointer to the kernel function, gs is the grid size, bs is the block size and p is the task priority if fixed priority policy is selected.

Once all pruda kernels have been added, the function pruda_start is invoked to start all periodic threads. Memory operations are implicitly achieved by the mean of Cuda unified memory, however explicit memory copies are under development to be soon supported.

7. CONCLUSION

In this paper, we have presented PRUDA, a programming interface to develop real-time scheduler on the top of Cuda. PRUDA provides different strategies to control temporal and space behavior of real-time tasks on the GPU. In future work, we plan to provide tools to analyze the real-time behavior of PRUDA tasks. In fact, scheduling real-time tasks does not allow free preemption and has a very limited number of priorities. These limitations has to be taken into account in the analysis of PRUDA tasks behavior to ensure the respect of timing constraints. We are also planing to develop a tool for tracing pruda tasks along with the NVIDIA nvprof profiling tool.

8. REFERENCES

[1] Tanya Amert, Nathan Otterness, Ming Yang, James H Anderson, and F Donelson Smith. Gpu scheduling on the nvidia tx2: Hidden details revealed. In *RTSS'2017*, pages 104–115. IEEE, 2017.

[2] Nicola Capodieci, Roberto Cavicchioli, Marko Bertogna, and Aingara Paramakuru. Deadline-based scheduling for gpu with preemption support. In *RTSS'2018*, pages 119–130. IEEE, 2018.

[3] Glenn A Elliott, Bryan C Ward, and James H Anderson. Gpusync: A framework for real-time gpu management. In *RTSS'2013*, pages 33–44. IEEE, 2013.

[4] Shinpei Kato, Karthik Lakshmanan, Aman Kumar, Mihir Kelkar, Yutaka Ishikawa, and Ragunathan Rajkumar. Rgem: A responsive gpgpu execution model for runtime engines. In *2011 IEEE 32nd Real-Time Systems Symposium*, pages 57–66. IEEE, 2011.

[5] Shinpei Kato, Karthik Lakshmanan, Raj Rajkumar, and Yutaka Ishikawa. Timegraph: Gpu scheduling for real-time multi-tasking environments. In *Proc. USENIX ATC*, pages 17–30, 2011.