Design and Analysis of Programming Platform for Accelerated GPU-Like Architectures

Houssam-Eddine ZAHAF
houssam-eddine.zahaf@univ-lille.fr
CRIStAL, Lille University
Lille, France

Giuseppe Lipari
giuseppe.lipari@univ-lille.fr
CRIStAL, Lille University
Lille, France

ABSTRACT

Many recent computing platforms combine CPUs with different types of accelerators such as Graphical Processing Units (*GPUs*), to cope with the increasing computation needs of complex real-time applications. However, most hardware accelerators have not been designed to provide predictable timing-behavior to support real-time tasks. Moreover, they do not provide efficient preemption mechanisms.

In this work, we present the design of a software library to program and execute real-time tasks onto hardware accelerators (e.g. GPUs) that exhibit limited-preemption capabilities with variables costs. The library provides: 1) parallel execution for real-time applications within the same accelerator; 2) the choice of different partitioned scheduling algorithms (FP, EDF, Gang-EDF); 3) support for (limited) task preemption. We describe a user space implementation of the library as a proof of concept. We also present a schedulability analysis for real-time tasks programmed using this platform, in particular for partitioned EDF and GANG-EDF.

The effectiveness of the proposed scheduling strategies and their analyses is demonstrated through 1) actual measurements on a GPU platform, and 2) through synthetic task sets.

ACM Reference Format:
Houssam-Eddine ZAHAF and Giuseppe Lipari. 2020. Design and Analysis of Programming Platform for Accelerated GPU-Like Architectures. In *28th International Conference on Real-Time Networks and Systems (RTNS 2020), June 9–10, 2020, Paris, France.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3394810.3394826

1 INTRODUCTION

Many real-time applications such as computer vision, surveillance systems, etc. are complex processing on a large amount of data, requiring teraflops computation capabilities. Since classical multiprocessor platforms combining only CPUs cannot satisfy the real-time requirements of such applications, they are usually implemented on special hardware computing units such as GPUs.

Recently, NVIDIA has proposed computing platforms combining CPUs with different types of specialized computing unit such as GPUs, Deep Learning Accelerating (nvDLA), etc., on the same chip. These platforms can offer suitable solution to meet deadlines for recent complex real-time applications. However, the complexity of the software combined with the complexity of the hardware make it difficult to predict the temporal behavior of such systems. Moreover, these accelerators are not designed to execute real-time tasks, as they do not provide proper hardware and software mechanisms to schedule real-time tasks.

Several authors [1, 8, 14] have been interested in providing support to real-time systems onto GPUs. NVIDIA GPU internal scheduling decisions have been studied in [1] by benchmarking the Jetson TX2 platform [17]. Several software mechanisms and libraries have been provided to control scheduling within the GPU [8, 13, 15]. However, most of them consider the GPU as a non-preemptive resource.

More recent NVIDA architectures, like Pascal, enable preemption at block level, whereas Volta-based architectures permit preemption at the instruction level. However, these preemption capabilities may be limited. For example in the Jetson TX2 platform, if task B preempts task C and task A wants to preempt task B, the latter is blocked until B finishes. Capodieci et al. in [4] have implemented preemptive scheduling algorithms (CBS/EDF, FP, etc.) for NVIDIA GPUs without these preemption limitations, however they use NVIDIA proprietary closed source internals.

In all the previously cited works, the GPU is seen as a single core platform: parallelism is exploited at the level of the *kernel* implementation (a kernel refers to the task code and is executed using several parallel threads on the GPU computing elements), but only one kernel at time is allowed to be resident within the GPU. This is a considerable limitation when several concurrent real-time applications need to be executed in the system. In fact, a GPU consists of many hundreds of computing SIMD and MIMD elements, and not every kernel needs to use all of them at the same time. As we will see later, the programmer can specify the amount of resources to execute a given task/kernel, and if enough free resources are available, it is possible to execute two or more tasks/kernels in parallel. Conversely, a kernel can demand all the resources of a GPU, and therefore must execute alone in the system.

To account for these different kind of kernels, the GPU can be modeled as a multi-core system, where tasks can be allocated to a single core, while others request all GPU resources *at the same time*. This execution model is similar to the *gang-scheduling* model. In gang scheduling, parallel tasks are run simultaneously on different processors at the same time. They start, finish and are preempted at the same time on all processors. Therefore it is the model that we will adopt in this paper.

RTNS 2020, June 9–10, 2020, Paris, France

ACM ISBN 978-1-4503-7593-1/20/06…$15.00
https://doi.org/10.1145/3394810.3394826

Contributions. In this paper, we present the design of a software library to program and execute real-time tasks onto hardware accelerators (e.g. GPUs) that exhibits limited preemption capabilities with variables costs. We describe one implementation of the library for the NVIDIA Jetson TX2 board and we provide schedulability analysis of the tasks programmed using our design on accelerators exhibiting the similar behaviors to GPUs.

The library provides the following functionalities:

- Parallel execution for real-time applications: every task is implemented by a CUDA kernel that executes on a number of *streaming processors* of the GPU. If enough streaming processors are available, two or more CUDA kernels can execute in parallel,
- Support for gang-scheduling: tasks may require more than one streaming processor can be scheduled along with tasks requiring only one streaming multiprocessor. Therefore, tasks are allowed to execute on all requested streaming multiprocessors at the same time. To support gang-scheduling, we combine resource reservations and hierarchical scheduling techniques (see Section 6).
- Choice of the scheduling algorithm: current implementation supports Fixed Priority (FP), Earliest Deadline First (EDF), and Gang-EDF;
- Support for (limited) task preemption.

To the best of our knowledge, this is the first library to supporting parallel and gang-scheduling on GPUs. The design of the library is generic and modular, and can be extended to other similar hardware platform and include other scheduling strategies.

The remainder of the paper is organized as follows. In the next section, we will briefly detail GPU architecture and its *known* internal scheduling mechanisms. In Section 3, we overview related works. Section 4 introduces the system model (task and computing architecture). The design of our programming platform is presented in Section 5. Schedulability analysis is described in Section 6. Results and experiments can be found in Section 7. We draw conclusions in Section 8.

2 OVERVIEW ON GPU ARCHITECTURE AND PROGRAMMING

Even if our platform design is generic enough to be applied to different accelerators, it has been currently implemented on NVIDIA GPUs using CUDA, therefore in this section we briefly overview the GPU architecture and programming.

A GPU is compound of one or several *streaming multiprocessors* (SMs) and one or several *copy engines* (CEs). Streaming multiprocessors are a collection of computational resources having the ability to execute GPU kernels, whereas copy engines perform memory copy operations between different address spaces. Programming the GPU requires dividing parallel computations into several *grids*, and each grid into several *blocks*. A block is a set of multiple threads. The number of blocks per grid and the number of threads per block is defined by the programmer. It represents the required amount of resources (threads) to properly execute the kernel.

A GPU can be programmed using generic platforms such OpenCL or proprietary APIs. We use CUDA, a NVIDIA proprietary platform, to have a tight control on SMs and CEs. Our library is implemented in C/C++ using the NVIDIA *nvcc* compiler.

Typical CUDA programs are organized in the same way. First, memory allocation operations are achieved both on CPU and GPU sides, for a real-time task this operation is achieved before the real-time task execution. Further, memory copies are operated between the main memory and the GPU visible memory. Later, the GPU kernel is launched (typically a very large number of parallel threads), and finally results are copied back to the main memory by memory copy operations. All threads of the same block are executed by one and only one SM, however different blocks of the same kernel may be executed on different SMs. The kernel execution order and resource management mechanisms are driven by internal closed-source NVIDIA drivers (in our case of study). Therefore, our library limits as much as possible scheduling decisions, taken by the NVIDIA internals, and implements scheduling strategies on the top of CUDA.

3 RELATED WORK

Analyzing the behavior of real-time tasks accelerated using hardware specific architectures has been widely studied [1, 4, 7, 8, 13, 14, 18, 21]. Particularly, the GPU has received more attention. Kato et al. have proposed different platforms (e.g. *TimeGraph* and *RGEM*) for non-preemptive scheduling for graphical tasks in the GPU [13, 14]. Another platform, called GPUsync, has been provided in [8]. It is a set of lock mechanisms for GPU engines (Compute and Copy). GPU engines are seen as mutually-exclusive resources that can be accessed only by real-time locking protocols. GPU kernels are scheduled with a non-preemptable FIFO algorithm. All these works consider the GPU as non-preemptable accelerator. Capodieci et al. in [4] modified the proprietary NVIDIA driver to implement different preemptive scheduling policies.

Many works have focused on other accelerators than GPU. An implementation of the *FRED* framework for the Linux over the Zynq-7000 platform produced by Xilinx has been presented in [18]. It discusses the design solutions for managing hardware accelerators in general. Then, a software architecture for Linux is presented. Danne and Platzner [7] proposed schedulability analysis for periodic real-time tasks on FPGA based accelerations. Burgio et al. in [3] in have proposed real-time analysis for many-core architectures embedding accelerators by focusing mainly on memory scheduling. In fact, memory copy operation between main memory and accelerators memory is a challenging issue in accelerated environment. Our platform handles memory copy operations, however memory copy operations are considered to be included in the task WCET in the schedulability analysis. This assumption can be easily removed and adapted according to the work in [3].

These works does not consider modeling the computations within the accelerator itself and consider them as a single computing resource. In this work, we consider accelerators as multiple resources. That is, several tasks may be accelerated at the same time on the same accelerator. Parallelisation mechanisms can be classified into two categories: *multithreaded* and *gang-scheduling*. In the first, parallel threads of the same task are scheduled independently from each other [2, 6, 11], whereas in the latter they are all scheduled at the same time. Goossens and Berten in [10] consider the scheduling

of periodic and GANG-tasks. They provide an exact schedulability test for Fixed Task Priority.

4 SYSTEM MODEL

We consider hardware platforms exhibiting the same characteristics as those described in Section 2. It is modeled as m identical cores (a core is equivalent to a streaming multiprocessor in a GPU).

Let $\mathcal{T} = \{\tau_1, \tau_2, \cdots, \tau_n\}$ be a set of n periodic tasks. Each task τ_i is an infinite sequence of jobs. It is characterized by:

- $\mathsf{C}(\tau_i)$ is the worst case execution time. It is the necessary time for the task to complete when all its resource requirements are met (the number of blocks for the GPU).
- $\mathsf{T}(\tau_i)$ is the task period. It is the exact time between the release of two consecutive jobs of task τ_i in the case of periodic tasks and is the minimum inter-arrival time between consecutive activation's for sporadic tasks;
- $\mathsf{D}(\tau_i)$ is the task relative deadline;
- $\mathsf{type}(\tau_i)$ is the task execution behavior. It must be one of two values:
  - single: the task requires only one core to be executed.
  - all: the task requires all cores to execute. Therefore tasks are executed for $\mathsf{C}(\tau_i)$ time units on all cores at the same time.

Preempting a task can be a costly operation, compared to more traditional real-time systems where preemption costs can be ignored. Therefore, we characterize a task by its worst case preemption cost, denoted by $\mathsf{PC}(\tau_i)$. For example, in the Jetson TX2 platform, we consider preemption at block level, therefore the preemption cost can be considered as the block execution time.

We denote by $\mathcal{T}^B$ the set of tasks that require all cores at the same time and by $\mathcal{T}^L$ the set of all tasks that do not belong to $\mathcal{T}^B$. In this paper, partitioned scheduling is considered. Tasks are allocated to cores and are not allowed to migrate at run-time. We assume that task allocation is known in prior. We denote by $\mathcal{T}_p$ the set of tasks allocated to **only** core p (i.e. $\mathcal{T}_p$ does not include tasks in $\mathcal{T}^B$), therefore:

$$\mathcal{T} = \{\bigcup_{p=1}^{m} \mathcal{T}_p\} \cup \mathcal{T}^B \tag{1}$$

We denote by $u(\tau)$ the utilization of task τ, it can be computed as the ratio of the task execution time by its period. We denote by $U(\mathcal{T})$ the total utilization of task set $\mathcal{T}$, it can be computed as follows:

$$U(\mathcal{T}) = \sum_{\tau \in \mathcal{T}} u(\tau) \tag{2}$$

$\mathcal{J}_i^j$ denotes the j^{th} instance (job) of task τ_i. In the case of periodic task behavior, $\mathcal{J}_i^j$ arrives exactly at time $a_i^j = j \cdot \mathsf{T}(\tau_i)$, whereas in the sporadic task model it must hold that $a_i^j \geq a_i^{j-1} + \mathsf{T}(\tau_i)$. The j^{th} job must complete no later than $d_i^j = a_i^j + \mathsf{D}(\tau_i)$.

Example 1. *We consider a set of 4 tasks, $\mathcal{T} = \{\tau_1, \tau_2, \tau_3, \tau_4\}$. Task characteristics and allocation are summarized in Table 1.*

In this example, task τ_1 is allocated on core 1 as well as task τ_2, whereas task τ_4 is allocated on core 2. Therefore task τ_1 and τ_2 can be run in parallel along with τ_4. When task τ_3 starts its execution, none of the other tasks can execute in parallel as long as τ_3 executes.

τ_i	C_i	$\mathsf{D}_i = \mathsf{T}_i$	alloc
τ_1	3	10	1
τ_2	5	15	1
τ_3	4	12	1 & 2
τ_4	1	10	2

Table 1: Task set example with allocation

5 GPU REAL-TIME PROGRAMMING USING OUR PLATFORM

In this section, we present the design of our programming platform. It is a set of functions to program real-time tasks on NVIDIA GPUs and take scheduling decisions.

The library uses *CUDA streams* to enforce scheduling and concurrent execution. CUDA streams are an abstraction provided by the *NVIDIA CUDA API* that allow to execute GPU kernels in a FIFO order. Between different streams, a fixed priority can be set. That is, kernel in a stream of high priority are meant to run before kernels of a lower priority stream. If a kernel in a low priority stream is executing, and a kernel of a higher stream is submitted, the GPU might preempt the current kernel, to execute the kernel of resident in the high priority stream. Fine-grained preemption capabilities are available in NVIDIA GPUs starting from the PASCAL architecture. In the PASCAL architecture a preemption is possible at the block level, i.e. preemption is achieved when all threads of a given block finish their execution. Recent VOLTA-based GPUs provide even finer-preemption levels, at instruction level.

While the number of available CUDA streams is theoretically unlimited, the number of priorities is limited. For example, in Jetson TX2 platform, the number of priorities for CUDA streams is limited to 2 levels. Although several streams allow asynchronous and concurrent execution between the different kernels, the kernels of the same stream are executed in FIFO order.

NVIDIA GPU resource management is hidden by proprietary internals. NVIDIA abstracts GPU computation resources through the notion of *streaming multiprocessors* (SMs) as a group of numerous parallel computing resources. In the scope of our programming platform, a SM can be assimilated to an independent core, and a GPU to a multi-core platform. We provide function allocate_to_sm(int sm_id) to allocate a kernel to a specific SM, where the sm_id is the id of the target streaming multiprocessor. Implementation details of our platform and its usage can be found in its repository (https://gitlab.cristal.univ-lille.fr/ptask/rtgpgpu/tree/master). For space reasons, we do not exhaustively describe all the functionalities of our platform. For further details, please refer to platform documentation available on its repository.

Our platform integrates three strategies to implement scheduling decisions, according to the user needs and goals. These strategies have different performances and overheads. In the first and the second strategy, the GPU is abstracted as a single core platform, while in the third strategy as a multi-core platform. An overview of our platform architecture is provided in Figure 1. The platform implement event-based schedulers. Therefore the scheduler is invoked

at activation and completion events. The different strategies have a common design. All tasks scheduled using our platform are stored in a task-queue called tq. When a task is activated, its priority is computed and it is inserted accordingly to the active run-queue denoted by rq, by invoking subscribe function. Later, according to the selected strategy, tasks are consumed from rq, when resched function is invoked. It moves the task from the active run-queue to CUDA streams. Once, the tasks are in the CUDA streams, the GPU internals consume the submitted kernels. Therefore, our platform controls task scheduling before their submission to the GPU. Tasks allocated to a dedicated core are inserted into the core queue (in Figure 1 SM0-q or SM1-q). In the rest of this section, we overview the different strategies and how they operate.

5.1 Single-stream schedulers

The first strategy, called *single-stream*, uses only **one** CUDA stream (h-sq in Figure 1) to enforce all the scheduling decisions. When scheduler is invoked and the stream queue is empty, the highest priority job in rq, according to the selected scheduling policy, is moved to h-sq .

As only one CUDA stream is used, once the task is executing, it cannot be preempted by another higher priority task. Therefore, only non-preemptive scheduling algorithms can be implemented. It is the simplest, and it provides an implicit synchronization between active tasks. However, it involves reserving all the GPU resources (both SMs in the case of Jetson TX2) to a single task at a time, even if it may not occupy all the resources.

5.2 Multiple stream schedulers: preemption enabling

In the second strategy, called *multiple streams*, the platform creates as much priority levels as streams to enforce scheduling decisions. Therefore, in the platform general design (Figure 1), only two streams are created, one with high and one low priority, h-sq and l-sq respectively) as the platform is instantiated for the Jetson TX2. This allows concurrent kernel execution, and preemption.

At each scheduling event, the scheduler checks one of the following cases:

(1) h-sq= $\emptyset \wedge$ l-sq= $\emptyset$: the scheduler will move the higher priority task from rq to $\mathsf{l-sq}$. The task will be submitted *immediately* to execution.
(2) h-sq = $\emptyset \wedge$ l-sq $\neq \emptyset$: the scheduler checks if the highest priority in rq is greater than the priority of the task in l-sq. If yes, the task is inserted into the high priority queue h-sq. Therefore, it preempts the task in l-sq, when *possible* (at block level for Jetson TX2).
(3) Otherwise, no scheduling decision are taken.

According to this scheduling rules, only one preemption is allowed at a time, when having two preemption levels, as in the case of Jetson TX2. These scheduling rules can be easily extended to more than two preemption levels. In the case of having PL preemption levels, $\mathsf{PL}-1$ nested preemption can be achieved. We will describe the analysis using an arbitrary number of preemption levels. Although this strategy allows preemption, it still uses the platform as a single core.

5.3 SMs as cores strategy

The third strategy uses the same stream configuration as the previous one. However, tasks may invoke allocate_to_sm($\cdots$), thus they are allocated to only one core of the platform. This allows more than one task to run in parallel at the same time. Tasks not invoking allocate_to_sm($\cdots$) are considered as requiring all GPU resources, hence are executed with no parallelism.

In addition to the scheduling structures described for the previous strategy, this strategy uses one queue-per-core (i.e. SM0-q and SM1-q in the case of two streaming multiprocessors). When a task is activated, the task type is checked. If it uses all GPU resources, 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. If it uses one core (i.e. SM), it is assigned to the corresponding core queue. Later, the jobs having the highest priority in all core queues (SM0-q and SM1-q) are scheduled by being inserted in l-sq or h-sq using the same scheduling rules as in the previous strategy.

We highlight that every strategy management is transparent to the platform user. Furthermore, it does not require any modification in the CUDA programming style. In fact, the design of the platform is generic and modular and permits to easily add new strategies.

6 SCHEDULABILITY ANALYSIS

In this section, we provide schedulability analysis for a set of tasks programmed using our platform for all the strategies. For sake of simplicity, we assume that memory copy operations are achieved at each kernel activation. The timing requirement of memory operations are included in the task worst case execution time, therefore they are implicitly included in analysis. Scheduling the memory copy operations independently for the task computation is going to be considered in future work.

We consider constrained deadlines, that is $\mathsf{D}(\tau_i) \leq \mathsf{T}(\tau_i)$ for the first and second strategy, whereas implicit deadlines for the last strategy.

6.1 Single-stream schedulers: EDF analysis

The first strategy allows using the platform as non-preemptive single resource. Therefore, classical non-preemptive analysis can be applied here. For completeness, the analysis for non-preemptive EDF is reported in Theorem 1.

Theorem 1. *(Jeffay et al. [12])*

Let $\mathcal{T} = \{\tau_1, \tau_2, \cdots, \tau_n\}$ be a set of sporadic tasks ordered in non decreasing order of their relative deadlines, that is $\tau_i \geq \tau_j \implies \mathsf{D}(\tau_i) \geq \mathsf{D}(\tau_j)$.

$\mathcal{T}$ is schedulable, under non-preemptive EDF using the first strategy, if :

$$\forall \tau_i \in \mathcal{T}, \forall t \leq t^*, \sum_{j=1}^{i} \left\lfloor \frac{\mathsf{C}(\tau_i) - \mathsf{D}(\tau_j) + \mathsf{T}(\tau_i)}{\mathsf{T}(\tau_j)} \right\rfloor \mathsf{C}(\tau_j) + \mathsf{B}_i \leq t \quad (3)$$

$$\mathsf{B}_i = \max_l \{C(\tau_l),\ i+1 \leq l \leq n\}$$

Where B_i is the maximum blocking time of task τ_i and t^* is the task set hyper period.

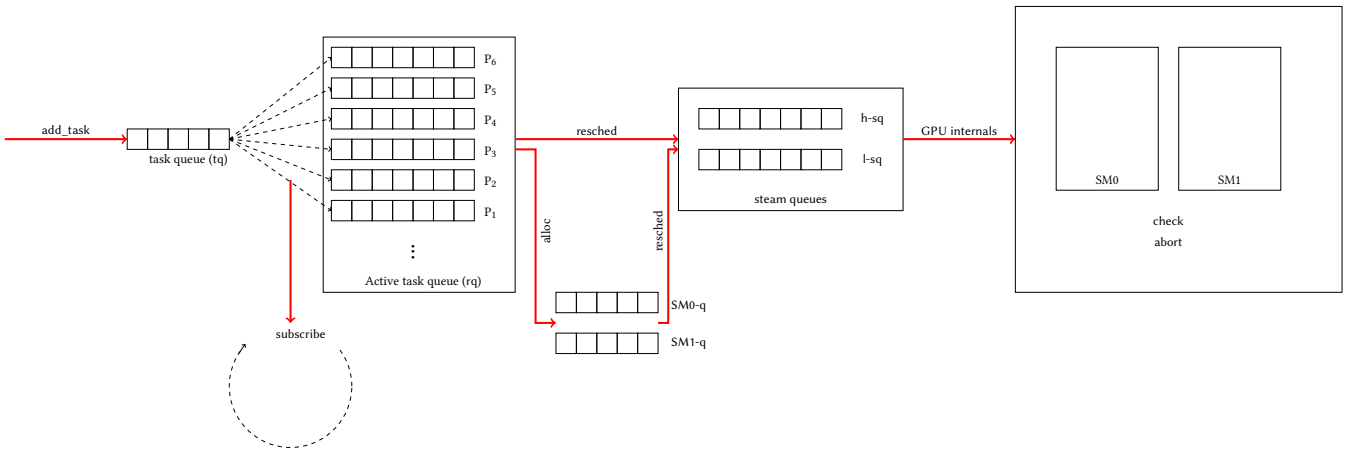

Figure 1: Global overview of our platform

6.2 Multiple stream schedulers: EDF analysis

In contrast to the previous strategy, the multiple stream strategy allows preemption. We denote by $PC(\tau_i)^o$ the preemption cost that task τ_i must consider when preempting other tasks. When PL preemption level is available, only PL − 1 nested preemption can be achieved. This limitation must be taken into account when analyzing the behavior of real-time tasks under this strategy.

Lemma 1. *Let $\mathcal{T} = \{\tau_1, \tau_2, \cdots, \tau_n\}$ be a set of sporadic tasks ordered in non decreasing order of their relative deadlines, that is*

$$\tau_i \geq \tau_j \implies D(\tau_i) > D(\tau_j)$$

$\mathcal{T}$ is schedulable, under EDF using the second strategy, if :

$$\forall i, \forall t \leq t^*, \sum_{j=0}^{i} \left\lfloor \frac{C(\tau_j) - D(\tau_j) + T(\tau_j)}{T(\tau_j)} \right\rfloor (C(\tau) + PC^o(\tau_j)) + B_i \leq t$$

$$B_i = \max_l \{C(\tau_l),\ i+1 \leq l \leq n - PL - 1\} \quad (4)$$

$$PC^o(\tau_j) = \max_l \{PC(\tau_l),\ i+1 \leq l \leq n\} \quad (5)$$

Proof. Let τ_i be the just-active task, and τ_j the running task. τ_i can preempt any task τ_j if $D(\tau_i) < D(\tau_j)$, except if all stream queues are not empty, according to scheduling rules of the second strategy. In such case, τ_i is blocked. Therefore τ_i is blocked if PL lower priority tasks are active. Therefore, if PL − 1 lower priority tasks are active, τ_i is not blocked.

Therefore task τ_i can not be blocked by tasks τ_j such that $j \geq PL - 1$. Thus, these tasks must not be considered for τ_i blocking time (Equation (4))

Task τ_i can preempt any other task having a lower priority, therefore we account for preemption cost for all tasks having a priority lower than τ_i as mentioned in Equation (5). □

6.3 SMs as cores : EDF-GANG analysis

In this strategy, the hardware architecture is considered as a multi-core platform where tasks may be either allocated to a single core or to all cores at the same time. Such scheduling problem is a special case of the well-known gang scheduling problem in the literature of real-time systems. For sake of simplicity, we remove preemption level constraints, as well as preemption costs. We will show how such parameters can be taken into account later in this section. In this section, we assume tasks with implicit deadlines, that its $D(\tau_i) = T(\tau_i)$.

We use a hybrid solution. It uses two time triggered reservations. The first reservation $\mathcal{R}^L$ is used to execute tasks that are allocated to only one core, whereas reservation $\mathcal{R}^B$ is used to schedule the tasks that are allocated to all m cores at the same time. All tasks allocated to a single core, even those executing on different cores, are executed on reservation $\mathcal{R}^L$. Within the same reservation, tasks are scheduled according to EDF. Reservations are periodic. Each reservation $\mathcal{R}$ is characterized by :

- $O(\mathcal{R})$: is the reservation offset. It is the time from which reservation starts.
- $Q(\mathcal{R})$: is the reservation budget. It is the time of service that the reservation ensures.
- $T(\mathcal{R})$: is the reservation period. It is the exact time between the beginning of two consecutive reservation intervals.

Example 2. *Let consider the task set in Table 1, using reservations $\mathcal{R}^L(0, 8, 12)$ and $\mathcal{R}^B(8, 4, 12)$. τ_3 executes within $\mathcal{R}^B$ service time, while τ_1, τ_2 and τ_4 within $\mathcal{R}^L$. Even if τ_1, τ_2 and τ_4 execute within $\mathcal{R}^L$, τ_4 executes in parallel with τ_1, τ_2 as it executes on a different core, τ_1, τ_2 are run concurrently on core 1.*

We focus on scheduling in core 1. The first reservation service time is the first 8 time units for each period of 12, whereas task τ_3 executes within the last 4 time units of the reservation period. The scheduling on core 1 is reported in Figure 2. Task τ_3 is allowed to run only in the blue hashed space, whereas the other tasks are able to run elsewhere. The last instance of task τ_1 misses its deadline.

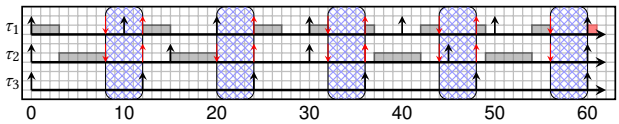

Figure 2: Example of time triggered reservation

Lemma 2. *(necessary schedulability test) Let $\mathcal{T}$ be a task set, $\mathcal{T} = \mathcal{T}^L \cup \mathcal{T}^B$.*
If $\mathcal{T}$ is schedulable, then :

$$\sum_{p=1}^{m} U(\mathcal{T}_p) + m \cdot U(\mathcal{T}^B) \leq m \tag{6}$$

Proof. The proof is trivial. Tasks of $\mathcal{T}^B$ require the processor for $m \cdot U(\mathcal{T}^B)$ as they execute on all cores at the same time, added to that the tasks of $\mathcal{T}^L$ are allocated to single cores, require at least their utilization to be schedulable. □

Theorem 2. *Let $\mathcal{T}$ be a task set scheduled on a platform compound of m cores using reservations $\mathcal{R}^L$ and $\mathcal{R}^B$ having the same period, and complementary offsets/budgets, therefore reservations do not overlap.*
$\mathcal{T}$ is schedulable if and only if :

- *$\forall p \in \{1, \cdots, m\}, \mathcal{T}_p$ is schedulable within reservation $\mathcal{R}^L$*
- *$\mathcal{T}^B$ is schedulable using reservation $\mathcal{R}^B$.*

Proof. The proof is simple. We assume that $\mathcal{T}^B$ is schedulable using reservation $\mathcal{R}^B$, therefore no task will miss its deadline. Similarly, tasks of $\mathcal{T}^L$ are allocated to a single core, and do not miss their deadlines. Proving the schedule feasible requires proving that both schedules do not overlap. This is ensured by reservation parameters, in fact $\mathcal{R}^L$ and $\mathcal{R}^B$ are complementary according to their offset, budget and periods, therefore they do not overlap, proving the theorem. □

According to Theorem 2, analyzing the schedulability of a task set implies: (i) analyzing the behavior of $\mathcal{T}^B$ using single core analysis, within reservation $\mathcal{R}^B$, and (ii) analyzing the schedulability on every core separately within reservation $\mathcal{R}^L$, and (iii) ensuring that both reservations do not overlap. Thus, converting the problem of scheduling gang tasks to a set of single core scheduling problems, for which well known scheduling techniques exist. In the rest of this section, we present schedulability analysis for single core platforms with time-triggered reservation. For sake of simplicity, we omit the reservation indexes.

Theorem 3. *(Single core schedulability using* sbf *[19])*
Let $\mathcal{T}$ be a set of sporadic tasks, executing within reservation $\mathcal{R}$.
$\mathcal{T}$ is schedulable under EDF if:

$$\forall t \leq t^*, \mathrm{dbf}(\mathcal{T}, t) \leq \mathrm{sbf}(\mathcal{R}, t) \tag{7}$$

such that :

$$\mathrm{dbf}(\mathcal{T}, t) = \sum_{\tau \in \mathcal{T}} \left\lfloor \frac{t - \mathsf{D}(\tau) - \mathsf{T}(\tau)}{\mathsf{T}(\tau)} \right\rfloor \cdot \mathsf{C}(\tau) \tag{8}$$

Where $\mathrm{sbf}(\mathcal{R}, t)$ is the supply bound function, i.e. the available processor time for reservation $\mathcal{R}$ within any interval of time of length t. It is computed as follows :

$$\mathrm{sbf}(\mathcal{R}, t) = \begin{cases} t - (k+1)(\mathsf{T}(\mathcal{R}) - \mathsf{Q}(\mathcal{R})) \\ \text{if } t > k \cdot \mathsf{T}(\mathcal{R}) + (\mathsf{T}(\mathcal{R}) - \mathsf{Q}(\mathcal{R}))) \\ k \cdot \mathsf{Q}(\mathcal{R}) \text{ else.} (k = \left\lfloor \frac{t}{\mathsf{T}(\mathcal{R})} \right\rfloor) \end{cases} \tag{9}$$

To reduce the complexity of computing the supply bound function, a lower bound approximation can be computed as follows:

$$\mathrm{sbf}(\mathcal{R}, t) = \begin{cases} 0 & \text{if } t \leq \mathsf{O}(\mathcal{R}) \\ \frac{\mathsf{Q}(\mathcal{R})}{\mathsf{T}(\mathcal{R})}(t - (\mathsf{T}(\mathcal{R}) - \mathsf{Q}(\mathcal{R}))) & \text{else.} \end{cases} \tag{10}$$

The test in Theorem 3 has been proposed for sporadic tasks, therefore it is pessimistic for periodic task sets. In this section we propose a schedulability test for periodic tasks. We prove the test not comparable with Theorem 3. Therefore, we will consider in the rest of this section, only periodic tasks.

The main idea of schedulability test for periodic tasks is to account for non-available time in the task demands. Therefore, moving task arrival times and deadlines, such that the task avoids being executed out of its available reservations. Let τ_i^* be the execution time demands of task τ_i including the time where it is not allowed to execute. Each job of τ_i has its own execution demand, deadlines, and arrival times as task periods may not match the reservation periods. We model the task execution requirement using a pipeline of chunks of jobs by unrolling the task period onto the task set hyper-period. We will show later that unrolling tasks is not necessary to derive our sufficient schedulability test. The task conversion is disclosed in Algorithm 1. Its complexity is exactly equal to $\frac{H}{\min\{\mathsf{T}(\tau_i), \mathsf{T}(\mathcal{R}))\}}$. [1]

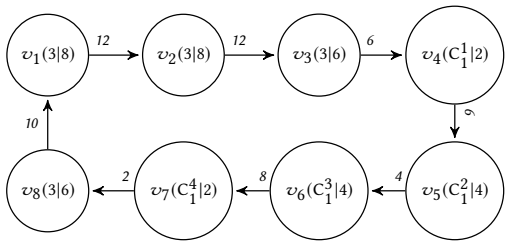

Figure 3: The requirement task for τ_1 of the Example in Table

Example 3. *Let consider task τ_1 of the example in Figure 2. Task requirements τ_1^* are reported in Figure 3, where each vertex is a chunk of the task, represented by the vertex identifier, the chunk execution time and the deadline, the inter-arrival time between task chunks are represented on the edges between vertices. The first job $\mathcal{J}_1^1$ will require to execute for* 3 *time units and must finish its execution before its* new artificial *deadline which is equal to* 8 *time units as shown in vertex v_1 in Figure 3. The second job, represented in v_2, must start its execution exactly at* 12 *time units from the arrival of v_1, and finish no later than* 8 *time units from its arrival. The $\mathcal{J}_1^4$ must execute within distinct intervals of the two reservation instances, as shown in Figure 2. Therefore it is split to* 2 *chunks, represented by v_4 and v_5, having deadlines of* 2 *and* 4 *respectively. The distribution of the task execution time on v_4 and v_5 is not known, therefore vertices execution times are denoted by C_1^1 and C_1^2 for vertices v_4 and v_5 respectively. The full conversion of task τ_1 is described in Figure 3. The sum of chunks execution times for the same job is bounded by the task worst case execution time, therefore the $\mathsf{C}_1^1 + \mathsf{C}_1^2 = 3$, under the constraint that*

[1] d(A, t) (resp. a(A, t)) allows to compute the deadline (resp. the arrival) of the t^{th} instance of A.

Algorithm 1 Conversion to digraph

```
1: Input: Task τ, Reservation R:
2: Output: requirement task τ*
3: k = 0, z = 0
4: j_l = create_chunks_list()
5: while (k · T(τ) < H or · T(R)) < H) do        ▷ H: Hyperperiod
6:     [a_eff, f_eff] = [a(τ, k), d(τ, k)] ∩ [a(R, z)), d(R, z)];
7:     if (a_eff < f_eff) then
8:         last = get_last_job(j_l);
9:         if (last sub-job is different instance) then
10:            C = C(τ);
11:        else
12:            C = undefined
13:            C(last) = undefined
14:        end if
15:        s = create_new_subjob(id, C, f_eff − a_eff, 0, a_eff, k)
16:        add_subjob_list(j_l, s)
17:    end if
18:    if (d(τ, k) == d(R, z)) then
19:        z++; k++;
20:    else
21:        if (a(τ, k + 1) > d(R, z)) then
22:            z++
23:        else
24:            if (d(τ, k) < d(R, z)) then
25:                k++
26:            end if
27:        end if
28:    end if
29: end while
30: update_arrival_times(j_l);
31: return j_l;
```

$C_1^1 \leq 2$ *for* v_2 *as its deadline is equal to* 2. *The artificial deadline and arrivals times are reported using red arrowns in Figure 2.*

Lemma 3. *Let* $\mathcal{T} = \{\tau_1, \tau_2, \cdots, \tau_n\}$ *be a set of periodic tasks executing within reservation* $\mathcal{R}$. *Let* $\mathcal{T}^*$ *be the set of requirements of every task in* $\mathcal{T}$.

$\mathcal{T}_k$ *is schedulable if and only if:*

$$\mathrm{dbf}(\mathcal{T}_k^*, t) \leq t, \forall t \leq t^* \tag{11}$$

such that :

$$\mathrm{dbf}(\tau^*, t) = \max_{v \in \tau} \sum_{v' \in \tau} \left\lfloor \frac{t - \tilde{O}(v') - \mathsf{D}(v') + \mathsf{T}(\tau)}{\mathsf{T}(\tau)} \right\rfloor \mathsf{C}(v') \tag{12}$$

where[2]*:* $\tilde{O}(v') = (\mathsf{O}(v') - \mathsf{O}(v)) \bmod \mathsf{T}(\tau)$

Proof. The proof comes straightforward from the work of Stigge et al. [20] as our model is a special case of the digraph tasks. In fact, each task requirement is a pipeline of jobs. Therefore, the number of paths to evaluate is exactly equal to the number of vertices. Moreover, t^* can be bounded to task set hyper-period rather than large bounds presented in [20] as the task periods are known. □

[2]We remind that the remainder of a/b is by definition a positive number r such that $a = kb + r$.

The task demand bound (dbf) function can be computed using dynamic programming. The dbf is therefore pre-calculated, and used during the analysis. Exploring solution space requires fast schedulability analysis. In fact, the test in Lemma 3 requires combining all possible values for non determined execution times of all chunks in all tasks. The dbf computation can be very time consuming, especially if task and reservation periods are prime. We provide now a dbf approximation to avoid exploring all possible values of chunk execution times.

Definition 1. *Let* τ^* *be the requirement for task* τ.
τ^{**} *denotes an approximation of* τ^* *obtained as follows:*

(1) *Select all the chunks within the same job*
(2) *Select the chuck having the largest deadline*
(3) *Eliminate other chunks and update vertices arrival times*
(4) *Repeat 1,2,3 for all job*

Example 4. *The approximation task* τ_1^{**} *for task* τ_1^* *presented in Figure 3 can be found in Figure 4. Please notice that ,* v_4 *and the* v_7 *has been eliminated.*

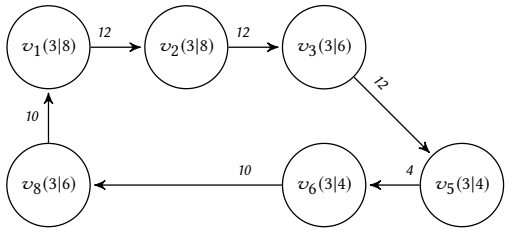

Figure 4: The τ_1^{**} conversion for τ_1^*

It is easy to prove that the even if the dbf of τ^{**}, is not necessarily an upper approximation of the dbf of τ^*, because it assumes that the execution time of all chunks that does not have the maximum deadline, are equal to zero, it is an upper bound of the dbf of the original task τ.

Definition 2. *Let* τ^{**} *be the task requirement of* τ *obtained by applying the transformation on Definition 1.*
Let τ^{***} *be a periodic task characterized by* $(\mathsf{C}, \mathsf{D}', \mathsf{T}')$ *such that :*

- D' *is the minimum deadline among all vertices in* τ^{**}
- T' *is the minimum period between any consecutive vertices in* τ^{**}

Theorem 4. *If task set* $\mathcal{T}^{***}$ *is schedulable, task set* $\mathcal{T}$ *is schedulable under reservation R,*

Proof. To prove the theorem, it is sufficient to prove the following Equation is true : :

$$\mathrm{dbf}(\tau^*, t) \leq \mathrm{dbf}(\tau^{***}, t), \forall t.$$

Let $\mathsf{L}^*(\tau, t)$ be the set of vertices that can be released by task τ^* at instance time t

$$\mathsf{L}^*(\tau, t) = \{v_1, v_2, \cdots, v_p\}$$

Consider a new sequence

$$\mathsf{L}^{***}(\tau^{***}, t) = \{v_1', v_2', \cdots, v_{p'}'\}$$

where each vertex $v'_{i'}$ has an execution time exactly equal to $C(\tau_i)$ which is the upper bound of every vertex v of the task τ^*.

The deadline of $T(\tau^{***})$ is shorter than the shortest deadline of task τ^* (respectively periods). Thus, It is clear that sequence L^{***} contains more jobs, with larger execution times than L^*. Therefore $dbf(\tau^*, t) \leq dbf(\tau^{***}, t)$, proving the theorem. □

Theorem 5. *The schedulability test presented in Theorem 3 and the schedulability test of Theorem 4 are not comparable.*

Proof. Proof by counter example.

Let $\tau_1(C = 4, D = 5, T = 10)$ and $\tau_2((C = 6, D = 40, T = 40)$ be two tasks served by reservation $\mathcal{R}(O = 0, Q = 16, T = 20)$.

By converting the task τ_1 to τ_1^{***}, we have $\tau_1^{***}(C = 4, D = 5, T = 10)$ and $\tau_2^{***}(C = 6, D = 16, T)$.

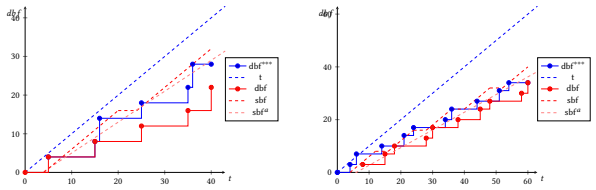

Figure 5: counter examples

The analysis results are represented in the left sub-figure of Figure 5. Task set compound of τ_1 and τ_2 is schedulable according to the test in Theorem 3, while it is not using schedulable according to Theorem 4.

Let now consider the task set compound of $\tau_1(C(\tau_1) = 3, T(\tau_1) = D(\tau_1) = 10)$ and $\tau_2((C(\tau_1) = 4, T(\tau_1) = D(\tau_1) = 15)$ running under reservation $\mathcal{R}(O(\mathcal{R}) = 0, Q(\mathcal{R}) = 8, T(\mathcal{R}) = 12)$. The results of schedulability analysis of these tasks are reported in the right sub-figure in Figure 5. It is clear that using the test of Theorem 3 the task set is schedulable, however when using the test in Theorem 4 it is not. Thus, the theorem is proven.

□

6.4 Analysis in the presence of harmonic periods

In this section, we consider harmonic periods, that is the reservations period are either multiplier or divider of each task period.

Definition 3. *Let τ be a task running using reservation $\mathcal{R}$ such that $T(\mathcal{R})$ and $T(\tau)$ are harmonic.*

τ' is a the harmonic conversion of task τ such that :

$$T(\tau') = \min\{T(\mathcal{R}), T(\tau)\} \tag{13}$$

$$D(\tau') = \max\{0, T(\tau') - (T(\mathcal{R}) - Q(\mathcal{R})\} \tag{14}$$

$$C(\tau)' = C(\tau) \cdot \frac{1}{\left\lceil \frac{T(\tau)}{T(\mathcal{R})} \right\rceil} \tag{15}$$

Theorem 6. *Let $\mathcal{T}$ be a set of tasks executing within reservation $\mathcal{R}$.*

Let $\mathcal{T}'$ be a task set obtaining by converting every task of $\mathcal{T}$ using Definition 3.

if $\mathcal{T}'$ is schedulable, $\mathcal{T}$ is schedulable.

Proof. The proof of this theorem is descendant from the proof on Theorem 4. It will be reported in under two assumptions.

Assumption 1. Reservations period are greater than the task period. As task period and reservation periods are harmonic, there will be no reservation that does not have arrival time or period matching one of the task arrival times. Therefore, none of the task instances will have more than one chunk. The most constrained job of these instances is either the first or the last (according to the offsets of reservations). When considering task conversion in Definition 3, the most constrained job is considered, therefore the analysis is safe.

Assumption 2. If the task period is greater than the reservation periods, the job will be split to a set of exactly identical chunks (having the same period, deadline) according to the conversion in Definition 3. Therefore the task execution time can be split, to several parts having the same length for every chunk, as defined in Equation (15). Therefore, the sum of the execution time of all chunks of any instance is equal to the task execution time.

From the assumptions (1) and (2) convers both case where reservation period is devider or multiplier of task periods, hence theorem is proved. □

7 RESULTS AND DISCUSSIONS

In this section, we evaluate the performances of the proposed programming platform and schedulability tests on real and synthetic task sets.

7.1 Synthetic experiments

We apply the different schedulability tests presented in the paper on a large set of randomly generated task sets onto an architecture composed of 2 SMs, similar to the GPU of the Jetson TX2. Task allocation is known in prior. We denote by n_1 the number of tasks allocated to core 1, respectively n_2 for core 2. n_b denotes the number of tasks that are allocated to both cores at the same time. Each point in all graphs is the average value of 100 execution.

7.1.1 Task set generation. The task generation algorithm starts by applying the UUniFast algorithm [9] to generate n utilization. For every task, the period is randomly selected from a predefined period list. We multiply the task period by its utilization to obtain the task execution time. The task deadline is randomly generated in the interval $[0.75 * T(\tau), T(\tau)]$.

This algorithm is repeated to generate tasks for $\mathcal{T}_1$, the tasks allocated to core 1 and $\mathcal{T}_2$ the tasks allocated to core 2, and $\mathcal{T}^B$ the tasks allocated to both cores. We vary total utilization of $\mathcal{T}^B$ from 0 to 1 by steps of 0.1. The utilization of the task sets $\mathcal{T}_1$ and $\mathcal{T}^B$ is bounded according to the values of $U(\mathcal{T}^B)$ using the necessary test in Equation (6).

The reservation period is selected also from the periods list. The reservation parameters are computed as follows: $Q(\mathcal{R}^L) = \frac{U(\mathcal{T}_1)}{U(\mathcal{T}_1)+U(\mathcal{T}^B)}$, $O(\mathcal{R}^L) = 0, O(\mathcal{R}^B) = Q(\mathcal{R}^L)$ and $Q(\mathcal{R}^B) = T(\mathcal{R}^L) - Q(\mathcal{R}^L)$.

7.1.2 Simulations and discussions. In Figures 6 and 7, we report schedulability rates of randomly generated task set where $U(\mathcal{T}^B)$ is fixed to 0.2. $U(\mathcal{T}_l)$ is varied from 0.2 to 1 by step of 0.1. We can see that all schedulability tests achieve high schedulability

rates when the workload is very low. When the load increases our test outperforms the sbf based test as the first fails at each time reservation budget is greater than the worst case execution time of any task (sporadic behavior), while our test keeps the task set schedulable as it considers only periodic task sets. We refer by SBF,SBF* and OUR to schedulability tests using Equation (9), Equation (10) and Theorem 4.

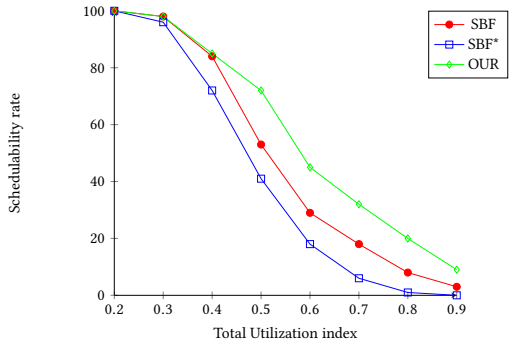

Figure 6: Schedulability rates when U($\mathcal{T}^B$)=0.2

In Figure 7, the $U(\mathcal{T}^B)$ utilization is set to 0.5. Now the reservation utilization are fairly shared between $\mathcal{R}^l$ and $\mathcal{R}^B$, Our dbf approximation becomes more pessimistic compared to the results in Figure 6, however it still outperforms the sbf based tests. The sbf approximation test fails drastically as the reservation utilization for $U(\mathcal{T}^B)$ is bigger.

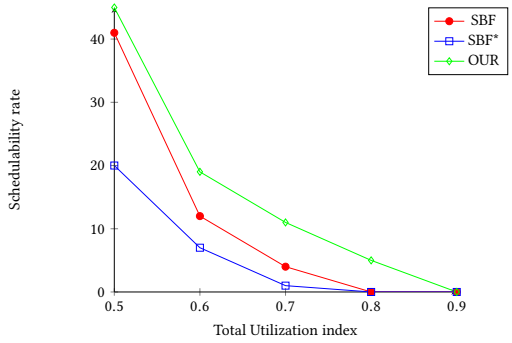

Figure 7: Schedulability rates when U($\mathcal{T}^B$)=0.5

In Figure 8, we report results for the same configuration as in Figure 7 and Figure 6. However, task periods are either multipliers or dividers of the reservations period. Our test is very efficient compared to the other reported tests. In fact, as reservation periods matches task periods, our dbf approximation is less pessimistic, therefore our test more efficient compared to sbf based tests. Moreover, our test complexity is very low compared to sbf based test.

7.2 Real implementation implementations

The experiments reported in this section has been achieved using our platform on Jetson TX2, running on Ubuntu 18.04.

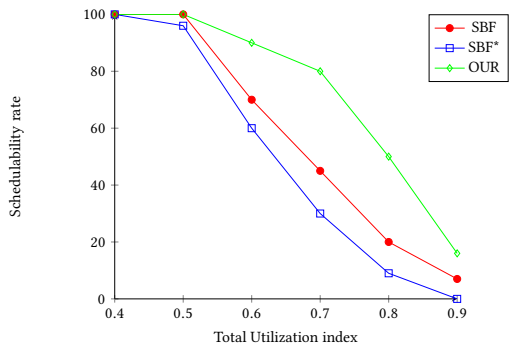

Figure 8: Schedulability test for harmonic periods

In Figure 9, we have run 3 different tasks running the same GPU kernel Compute_Hessian that can be found in Autoware [16] NDT package. Each task has a different priority. We have instantiated the number of blocks so that each task uses all the GPU resources (both SMs). We report the response time measurements for the most priority task when using our platform and when launching the kernels periodically without any control.

Our platform allows more predictability for the response time bounds compared to the default execution (a stream per task). In fact, in the absence of any control mechanism, tasks are all submitted to the GPU without any ordering in a concurrent way, that is the priority order is related to the task activation time only and available resource in the GPU. The task response time is very variable and can be very large, compared to the average case where the task executes in isolation on the GPU. Our platform is more deterministic and can provide *safe* execution behavior.

However, our platform present some anomalies due to the implementation in the user-space. Our response time, in few cases, can be high compared to the average value when using our platform. In fact, this is due to the completion notification mechanism. Once the kernel is submitted, our platform may call cudaStreamSynchronize primitive to notify the end of that job (it is called only when active job list does not contain the predicted highest priority job). The last is an NVIDIA-internal closed-source primitive, that may have unpredictable timing behavior. As response times are measured after the end of that primitive, they may be large, (the response times include the needed time to execute cudaStreamSynchronize primitive to avoid including it in analysis).

In Figure 10, we execute two tasks one of a high priority and the other of low priority, always the compute_Hessian processing of Autoware. We modified the input kernel size to enlarge the execution times so preemption can be more easy to emulate, the high priority task executes exactly 64 blocks, sufficiently to utilize all GPU resources, while the other contains variables number of blocks. The low priority task is run first in the we reproduce some of experiments in [5] using our platform. We report here the average execution time as it is reported by the NVIDIA profiler. Both tasks have been run using our platform. We recall that the NVIDIA profiler does not profile preemption, it reports only the start and the end execution time of GPU tasks. In the circle-mark plot, the low priority task is run in isolation (using all GPU resources), while

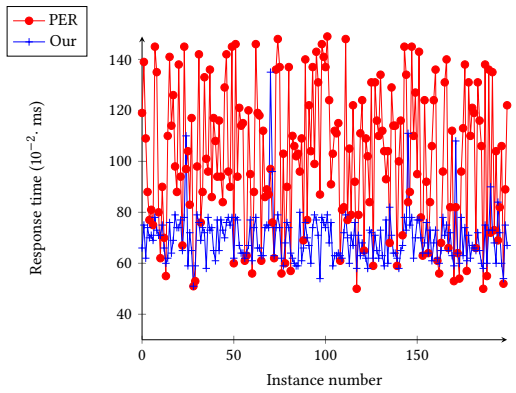

Figure 9: Execution response time with and without our platform

in the +-mark plot the low priority task is run first, later the high priority task is submitted. Both tasks have the same period. As you can see, when both tasks run only 64 blocks. The high priority task does not have time to preempt the low priority task as we dispose of preemption at block level in the Jetson TX2 platform as both tasks have the same execution time. For all all other scenarios where low priority task have more than 64 blocs, the low priority task execution time is increased in average by the high priority task execution time, the task has been preempted as expected at block level.

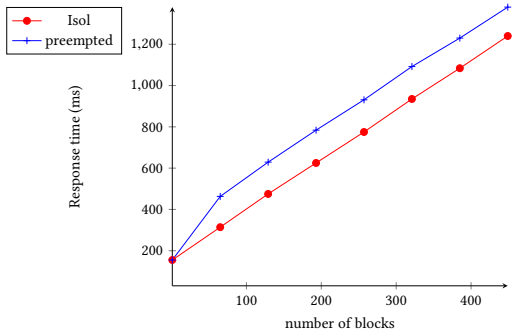

Figure 10: response time in the presence of preemptions

8 CONCLUSION

In this paper, we have presented the design of a programming platform for GPU-like architectures, and its implementation in user-space using CUDA for NVIDIA Jetson TX2. We developed analysis for our platform for EDF, and a special case of GANG-EDF.

We have also provided a large set of synthetic experiments and real implementations using our platform.

In future work, we plan to study memory co-scheduling for accelerated environments and provide more precise schedulability tests. We plan also to implement the solution that we provide in this paper on the nouveau GPU driver.

ACKNOWLEDGEMENT

This research has been funded in part By PHC-CURIEN TASSILI 19MDU213 grant, PRIMA WATERMED 4.0 and SECOURT ALSAT projects.

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] James H Anderson and John M Calandrino. Parallel real-time task scheduling on multicore platforms. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 89–100. IEEE, 2006.

[3] P. Burgio, A. Marongiu, P. Valente, and M. Bertogna. A memory-centric approach to enable timing-predictability within embedded many-core accelerators. In *2015 CSI Symposium on Real-Time and Embedded Systems and Technologies (RTEST)*, pages 1–8, Oct 2015.

[4] 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.

[5] Nicola Capodieci, Roberto Cavicchioli, Paolo Valente, and Marko Bertogna. Sigamma: Server based integrated gpu arbitration mechanism for memory accesses. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, pages 48–57. ACM, 2017.

[6] Sébastien Collette, Liliana Cucu, and Joël Goossens. Integrating job parallelism in real-time scheduling theory. *Information Processing Letters*, 106(5):180–187, 2008.

[7] K. Danne and M. Platzner. Periodic real-time scheduling for fpga computers. In *Third International Workshop on Intelligent Solutions in Embedded Systems, 2005.*, pages 117–127, May 2005.

[8] 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.

[9] Paul Emberson, Roger Stafford, and Robert I Davis. Techniques for the synthesis of multiprocessor tasksets. In *WATERS*, 2010.

[10] Joël Goossens and Pascal Richard. Optimal scheduling of periodic gang tasks. *Leibniz transactions on embedded systems*, 3(1):04–1, 2016.

[11] ZAHAF Houssam-Eddine, Giuseppe Lipari, Marko Bertogna, and Pierre Boulet. The parallel multi-mode digraph task model for energy-aware real-time heterogeneous multi-core systems. *IEEE Transactions on Computers*, 2019.

[12] Kevin Jeffay, Donald F Stanat, and Charles U Martel. On non-preemptive scheduling of period and sporadic tasks. In *[1991] Proceedings Twelfth Real-Time Systems Symposium*, pages 129–139. IEEE, 1991.

[13] 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.

[14] 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.

[15] Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott Brandt. Gdev: Gpu resource management in the operating system. In *USENIX'12*, pages 401–412, 2012.

[16] Shinpei Kato, Eijiro Takeuchi, Yoshio Ishiguro, Yoshiki Ninomiya, Kazuya Takeda, and Tsuyoshi Hamada. An open approach to autonomous vehicles. *IEEE Micro*, 35(6):60–68, 2015.

[17] NVIDA. Jetson TX2 Kit Developer. Technical report, NVIDIA Corporation, 10 2017.

[18] M. Pagani, A. Balsini, A. Biondi, M. Marinoni, and G. Buttazzo. A linux-based support for developing real-time applications on heterogeneous platforms with dynamic fpga reconfiguration. In *2017 30th IEEE International System-on-Chip Conference (SOCC)*, pages 96–101, Sep. 2017.

[19] Insik Shin and Insup Lee. Periodic resource model for compositional real-time guarantees. In *RTSS 2003. 24th IEEE Real-Time Systems Symposium, 2003*, pages 2–13. IEEE, 2003.

[20] Martin Stigge, Pontus Ekberg, Nan Guan, and Wang Yi. The digraph real-time task model. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 71–80. IEEE, 2011.

[21] Reyyan Tekin, Houssam-Eddine Zahaf, and Giuseppe Lipari. Pruda: An api for time and space predictible programming in nvdia gpus using cuda. In *Junior Workshop: JRWRTC-Real-Time Networks and Systems 2019*, 2019.