Towards Efficient Parallel GPU Scheduling: Interference Awareness with Schedule Abstraction

Nordine Feddal, Houssam- Eddine Zahaf, Giuseppe Lipari

Towards Efficient Parallel GPU Scheduling: Interference Awareness with Schedule Abstraction

Nordine Feddal
Univ. Lille, CNRS, Inria
Centrale Lille, UMR 9189 CRIStAL
F-59000 Lille, France

Houssam-Eddine Zahaf
Nantes Université, CNRS, INRIA
Centrale Nantes, IMT Atlantique
LS2N, UMR 6004
F-44000 Nantes, France

Giuseppe Lipari
Univ. Lille, CNRS, Inria
Centrale Lille, UMR 9189 CRIStAL
F-59000 Lille, France

ABSTRACT

GPUs are powerful computing architectures that are increasingly used in embedded systems for implementing complex intelligent applications. Unfortunately, it is difficult to predict their temporal behavior, especially when multiple parallel tasks are concurrently executed. Running one single task at a time may results in severe underutilization of the resources; on the other hand, running multiple tasks concurrently may introduce mutual interference.

In this work, we introduce Parallel Batch Scheduler (PBS) to enable parallel execution of a set of real-time tasks on GPUs. PBS avoids concurrent execution when it might jeopardize schedulability, and it identifies scenarios where parallel flows might enhance platform utilization and therefore schedulability. To find the feasible scenarios, we propose a scheduling analysis based on a scheduling graph, in which all possible concurrent and serialized scenarios are evaluated for schedulability. To mitigate the explosion in the state space, we propose a technique to reduce the size of the graph.

Through an extensive set of experiments, we demonstrate that PBS outperforms both serialized and fully parallel execution approaches, highlighting its effectiveness in maximizing GPU utilization while maintaining schedulability. We illustrate the usefulness of our approach through the development of a tool that takes the trace of the execution generated by our schedulability analysis and manages GPU workload submissions for GPU tasks.

CCS CONCEPTS

• **Computer systems organization → Real-time system specification**.

KEYWORDS

Real-time, scheduling, GPU, interference, scheduling graphs

1 INTRODUCTION

Current trends in embedded system design and development consist in integrating machine learning and artificial intelligence into complex embedded applications. Due to the heavy computational requirements, designers are moving towards heterogeneous platforms consisting of multi-core CPUs coupled with massively parallel accelerators such as Graphical Processing Units (GPUs). The adoption of these platforms represents a difficult challenge for developers of safety-critical applications, e.g. avionics and automotive systems, where real-time predictability is of utmost importance.

One of the main challenges for the design of such systems lies in ensuring predictable temporal behavior of the GPUs. The compute capacity of GPUs comprises hundreds of Arithmetic Logical Units grouped into streaming multiprocessors (SMs), and each GPU can consist of multiple SMs. The number of SMs has been increasing across different generations of GPUs, ranging from 16 for average-sized GPUs to over 100 for the largest GPUs. These GPUs can be programmed using various APIs, such as Vulkan, OpenCL, etc. They enable the offloading of parallel tasks (called *kernels*) to the GPUs. Programmers have the flexibility to submit GPU kernels either sequentially, in a First-In-First-Out (FIFO) manner, or concurrently. In concurrent execution, tasks compete for GPU resources, which are arbitrated by the GPU internal scheduling mechanisms. For NVIDIA GPUs, this feature is known as Simultaneous Kernels (SMK), and it is only possible when the involved kernels satisfy specific runtime conditions [14].

Using the GPU as a single resource, by sequentially submitting the kernels to execute, can lead to underutilization of its computing capacity, especially for legacy code that has been designed for smaller GPUs compared to modern ones. On the other hand, experimental results on NVIDIA architectures have uncovered counter-intuitive behavior when concurrently executing multiple independent kernels. Parallel execution flows might result in extra latency compared to the baseline FIFO execution, as discussed in [14]. This outcome is attributed to contention for memory and computing resources among independent CUDA kernels. The work conducted by [25] has studied the runtime conditions under which the runtime costs can be important. They proposed techniques to classify kernels as memory-bound or compute-bound and they have shown how they might impact the temporal behavior of each other. Without proper mechanisms to control parallel execution within the GPU and the timing behavior of different parallel kernels, it is impossible to guarantee the respect of the system's timing constraints.

In this paper, we propose the *Parallel Batch Scheduler*, a novel GPU scheduler approach for periodic real-time parallel GPU kernels, and its corresponding schedulability analysis. The analysis precomputes a parallel schedule that ensure that all jobs will meet their deadlines by exploring possible parallel executions that do not jeopardize the system schedulability.

The contributions of this paper include: (i) an abstract execution model of GPU kernels that considers interference, and (ii) a scheduler for GPU kernels through the construction of a sequence

RTNS 2024, November 7–8, 2024, Porto, Portugal

ACM ISBN 979-8-4007-1724-6/24/11
https://doi.org/10.1145/3696355.3696361

of schedulable jobs featuring parallel execution. We evaluate the performance of our scheduler against FIFO schedulers and completely parallel schedulers through an extensive set of synthetic experiments. Finally, we present the design of a tool that is able to play the generated schedules online on the GPU.

2 SYSTEM MODEL

2.1 GPU Architecture and programming model

A General Purpose Graphics Computing Unit (GPGPU, or simply GPU) comprises computation resources and memory copy resources. Computation elements consist of hundreds of small Arithmetic Logic Units (ALUs) arranged in one or multiple streaming multiprocessors (SMs), which, in turn, are grouped into Graphic Processing Clusters (GPCs). Memory transfers are handled by a copy engine, a coprocessor responsible for executing memory copy operations between different address spaces. Programming the GPU can be done using special APIs such as the OpenCL standard, Vulkan, the Nvidia CUDA proprietary API, and others. From the programmer's perspective, these APIs expose a similar set of functionalities. Typically, GPU programs follow a specific programming pattern: first, memory allocation is performed on both the CPU (host) and GPU (device) sides, usually during the initialization stage because it can be a time-consuming operation. Next, memory copy operations are executed between the main memory and the GPU-accessible memory. One or more programs (also called kernels in the NVidia terminology) are then launched and executed on the GPU; and finally, the results are copied back to the main memory.

Programming a GPU kernel involves a specific task decomposition. Initially, the parallel task is defined as a compute grid, which is further composed of several *blocks*. A *thread block* is a set of threads that can be executed either serially or in parallel. Different blocks can be allocated on a single SM or dispatched to different SMs, but all the threads of the same block are executed within the same SM. The programmer defines the number of thread blocks per grid and the number of threads per block. When a single GPU kernel is used at a time, the NVIDIA runtime, through its block scheduler, dispatches the blocks to different SMs in a round-robin fashion [14]. The programmer can submit multiple CUDA kernels using *CUDA streams*. A *CUDA stream* is an abstraction of a queue containing compute and copy operations: all operations within the same stream are executed in the order of their submission, whereas operations from different streams are dispatched independently, allowing operations belonging to different streams to overlap in time. In this scenario, the GPU block scheduler dispatches thread blocks from different kernels among the SMs, taking into account the occupancy of memory and compute resources requested by the submitted kernels [15]. The simultaneous submission of different kernels to the GPU can introduce interference, which might have high runtime costs, as shown in [25].

2.2 Task model

We consider a set of n periodic real-time tasks, denoted as $\mathcal{T} = \{\tau_1, \tau_2, \cdots, \tau_n\}$. Each task τ_i represents a single GPU kernel and is characterized by the following parameters:

- $\mathsf{T}(\tau_i)$: The task period, which is the exact time interval between two consecutive activations of task τ_i.
- $\mathsf{D}(\tau_i)$: The task relative deadline. Task τ_i must complete no later than $\mathsf{D}(\tau_i)$ from it activation.
- $\mathsf{C}(\tau_i)$: Worst-case execution time of task τ_i when all its blocks execute in parallel on the GPU without interference.

A *job* J is an instance of the periodic task. We denote as $\tau(J)$ the task to which job J belongs. For brevity, we overload our previous notation to jobs: for example, the worst-case execution time of job J will be denoted as $\mathsf{C}(J)$, which is the same as $\mathsf{C}(\tau(J))$.

A job is further characterised by an arrival time $\mathsf{a}(J)$ and an absolute deadline $\mathsf{d}(J)$. Let J be the k-th job of task τ: then $\mathsf{a}(J) = k \cdot \mathsf{T}(J)$ and $\mathsf{d}(J) = \mathsf{a}(J) + \mathsf{D}(J)$. We denote by $f(J)$ the finishing time of the job in a specific schedule. A job is *ready* if it is arrived but it has not yet completed. The system maintains a queue of ready jobs to be scheduled. The hyperperiod $\mathcal{H}$ represents the least common multiple of the periods of all tasks in the task set.

We define as $\mathcal{A}(t_1, t_2)$ the set of jobs with arrival time between t_1 and t_2:

$$\mathcal{A}(t_1, t_2) = \{J | t_1 \leq \mathsf{a}(J) \leq t_2\}$$

2.3 Block allocation

When a job is submitted to the GPU, the block scheduler allocates the kernel blocks to the SMs in a round-robin fashion based on their occupancy. Occupancy is computed according to the number of blocks $\mathsf{B}(\tau_i)$, the number of threads $\mathsf{TH}(\tau_i)$, the number of registers per thread $\mathsf{Regs}(\tau_i)$, and the size of shared memory $\mathsf{Smem}(\tau_i)$. These parameters are crucial in determining if parallel execution is allowed. A job may *fit* or not on the GPU. The *order* of submission of the parallel jobs has an impact on their allocation. We give an example below to illustrate these concepts.

Example 2.1. Consider two jobs J_1 and J_2: J_1 contains 4 blocks of 64 threads each, and J_2 only 1 block of 192 threads. Suppose we want to execute these two jobs in parallel on a GPU with only two SMs of 256 threads each.

Suppose we submit $\{J_1, J_2\}$ to the block scheduler, in this order. The hardware block scheduler will first allocate the blocks of J_1 in round-robin order (see paper [14] for a model of the block scheduler of NVidia): therefore, it will allocate two blocks of J_1 on the first SM, using 128 threads, and the other two blocks on the second SM, again using 128 threads. When it tries to allocate the single large block of J_2, it will not fit in any of the SMs, and J_2 will not execute in parallel with J_1.

Conversely, suppose that the jobs are submitted in the reverse order $\{J_2, J_1\}$: the large block of J_2 is allocated first filling 3/4 of the first SM, and all the other blocks of J_1 will easily fit on the second SM. In this case, J_1 and J_2 will execute in parallel. □

The previous example shows that the order in which jobs are submitted to the hardware block scheduler plays an important role in their execution order, and therefore it impacts the schedulability of the system.

Definition 2.2. (batch of jobs) A *batch of jobs* $\beta = \langle J_1, J_2, \ldots, J_n \rangle$ is an ordered sequence of jobs to be submitted to the GPU, every job will be submitted using a different CUDA stream.

Definition 2.3 (Eligible Jobs batch). Let $\mathcal{R}(t)$ be a set of ready jobs at time t and let $\beta = \langle J_1, \ldots, J_n \rangle$ be a batch of jobs such that $\forall i = 1, \ldots, n \;\; J_i \in \mathcal{R}(t)$.

β is *eligible*, if all its jobs can be immediately executed in parallel when they are submitted to the GPU block scheduler in the same order.

To check if a job batch is eligible, we simulate the allocation algorithm described in [14]. The simulation algorithm will return *true* if the batch is eligible, and *false* otherwise.

Please, remark that the graph analysis described in Section 3 does not depend on the specific details of the block allocation algorithm, and it can be easily generalized by using any other method to assess the *eligibility* of a batch of jobs. For the rest of the paper, we will abstract away from the details of the specific allocation algorithm: we will instead assume that an *oracle* will tell us if a batch of job is eligible or not and return its completion time. Since the order of submission has an impact on eligibility, we need to check all possible permutations of any subset of $R(t)$ for eligibility.

2.4 Scheduling model

In this section, we define a novel off-line scheduler that selects a batch of the ready jobs to be submitted to the GPU, ensuring that all jobs will complete before their deadlines, called *Parallel Batch Scheduler*.

Let $\mathcal{R}(t)$ be the set of *ready* jobs at time t; that is, the set of jobs whose arrival time is less than or equal to t and that have not yet completed at time t. When the scheduler is invoked, it selects a batch of jobs from $\mathcal{R}(t)$ that is *eligible* (see Definition 2.3) and submits them to the GPU using a different CUDA stream per job. Then, it waits for the last job to finish before selecting a new batch. We call the interval of time between two invocations of the scheduler a *scheduling frame*. During a scheduling frame, jobs execute in parallel. The scheduler is therefore invoked at scheduling frames boundaries.

The *Parallel Batch Scheduler* (PBS) is *parallel*, because it can submit multiple jobs to the GPU that will be executed in parallel; it is *non-preemptive*, because it takes a new scheduling decision only when the previous job batch has finished. PBS is *non-work conserving at the job level*: in fact, some GPU resource can be freed before the end of the scheduling frame, and nothing else is submitted to the GPU even if a new job arrives before the end of the scheduling frame. On the other hand, PBS is *work-conserving at the batch level*: the scheduler takes a decision based on the ready jobs present at the beginning of the new scheduling frame, and it is not allowed to keep the GPU idle waiting for more jobs to arrive.

2.5 Interference

When a job batch is composed of two or more kernels (tasks), interference may occur, leading to increased total execution times. This interference primarily arises from bus contention and/or data cache loading and eviction.

Three different main approaches have been proposed in the literature to upper bound the interference. The first approach in [9] involves analyzing the binary code (PTX and SASS) to determine the conditions under which interference might occur. The second approach is measurement-based [10], and consists in pushing the GPU to extreme behaviors through benchmarking and modeling interference profiles using linear and non-linear regressions. Another approach involves leveraging AI techniques to infer the potential presence of interference [11].

	τ_1	τ_2	τ_3
$T(\tau_i)$	4	5	10
$D(\tau_i)$	4	5	10
$C(\tau_i)$	1	3	3

batches	completion
$\langle\tau_1, \tau_2\rangle$	4
$\langle\tau_1, \tau_3\rangle$	4
$\langle\tau_2, \tau_3\rangle$	4
$\langle\tau_1, \tau_2, \tau_3\rangle$	6

Figure 1: Parameters for the task set of the example.

In this paper, we abstract away this problem by considering that we know the completion time of every possible batch of tasks that may be executed in parallel, that is the completion time of the last job to complete in the batch. We are aware that deriving this information may be difficult; we are currently experimenting with different models for computing worst-case execution times and interference in realistic case studies. In Section 4.2 we will describe how we derived execution bounds for our experiments.

3 THE SCHEDULING GRAPH

Our approach is based on the generation of a *scheduling graph* representing all possible valid sequences of execution of the jobs in the hyperperiod. The number of possible scheduler configurations grows exponentially with the number of tasks and jobs to schedule. Therefore, one of the main challenges is to reduce as much as it is possible the size of the state-space by removing unnecessary nodes from the graph. In this section, we first describe the core idea that motivates our approach with an example, then we discuss the scheduling graph generation and the reduction strategies in details.

3.1 Example of the impact of interference on schedulability

For our motivating example, let us consider 3 tasks, τ_1, τ_2, and τ_3 having the parameters described in Figure 1. Consider the set of jobs generated by these tasks from time instant 0 until the hyperperiod $\mathcal{H} = 20$, which includes: 5 jobs of task τ_1 (ranging from J_1 to J_5), 4 jobs of task τ_2 (ranging from J_6 to J_9), and 2 jobs of task τ_3 (J_{10} and J_{11}). For simplicity, we consider that the GPU has enough resources to execute all tasks in parallel, so we do not report the other task parameters such as the number of blocks, threads, and registers. We use the Parallel Batch Scheduler described in the previous section as a scheduler. Figure 2 illustrates 4 different scheduling scenarios for the same job set.

In Figure 2a, jobs are executed one after the other, as we do not allow parallel execution of multiple kernels[1]. Scheduling is non-preemptive, and the jobs are submitted according to the Earliest Deadline First order. Therefore, J_1 is executed first, followed by J_6, etc. Although none of the tasks suffers from interference due to parallel execution, job J_7 misses its deadline.

In Figure 2b, we show the case in which the scheduler always dispatches all ready jobs at once. That is, at time instant 0, jobs J_1,

[1] On Nvidia GPUs, this can be obtained simply by submitting all jobs using the same CUDA stream.

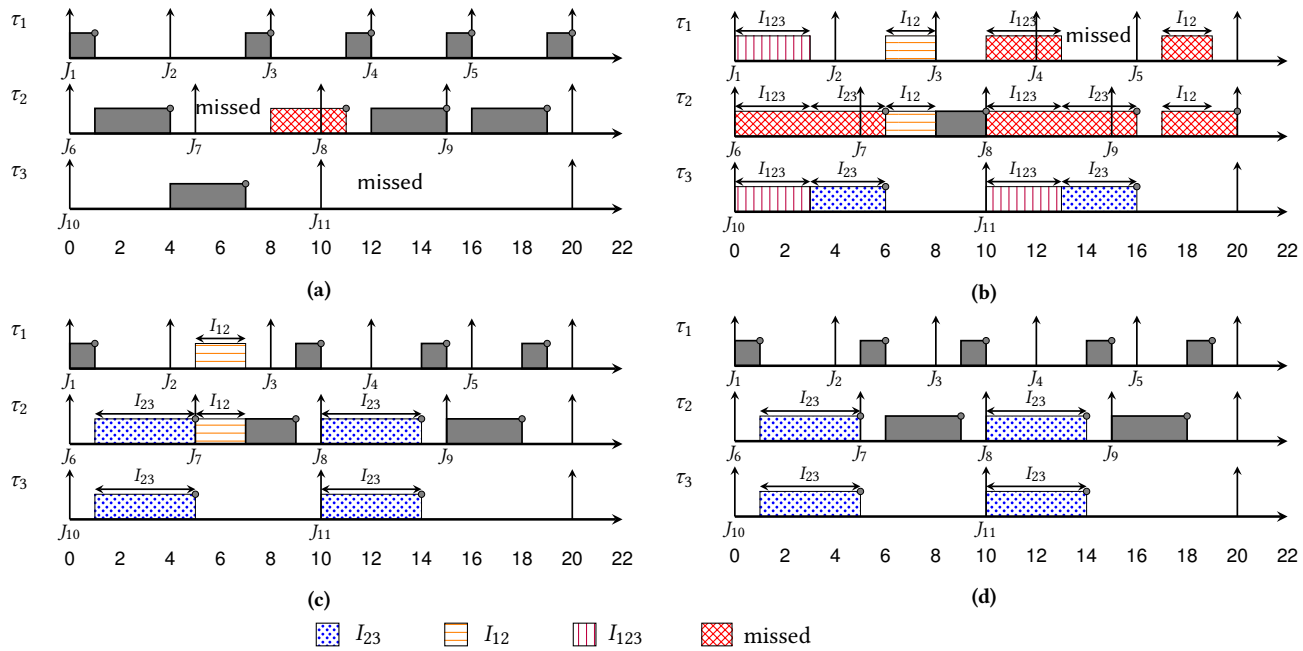


Figure 2: Four different schedules for the example task set.

J_6, and J_{10} are all submitted to the GPU[2]. As kernels execute in parallel, they suffer from mutual interference, and their execution slows down. At the completion of the first batch of jobs (J_1, J_6, J_{10}) at time 6, a new batch is submitted composed of J_2 and J_7. Due to the interference between jobs, J_3, J_4, J_5, J_6, and J_7 miss their respective deadlines.

Fig. 2c and Fig. 2d represent two of the schedules generated by our schedule exploration algorithm. The algorithm evaluates the impact of the interference due to the parallel execution of the active kernels, and selects one of the eligible batches to execute on the GPU. Therefore, scenarios that might jeopardize schedulability due to excessive interference (Figure 2b), or the serialized execution (Figure 2a) are discarded by our algorithm.

In the above example, J_1 is executed without concurrency in the first scheduling frame. At its completion, a batch composed of J_6 and J_{10} is submitted to the GPU for parallel execution. Further, J_2 and J_7 are executed in parallel in the same scheduling frame in Figure 2c, while their execution is serialized in Figure 2d. In these two cases, no deadline is missed, and the GPU resources are utilized in a *better* way than in both the previous situations. Both are schedulable, but Figure 2d uses fewer GPU resources.

In general, it is hard to determine the best schedule order to minimize the GPU resources utilization without jeopardizing the system predictability. Our schedulability analysis, called GPUSched, is based on the exploration of all possible job batches to be submitted to PBS.

In the previous example of Figure 2, at time 0 three jobs are ready to be executed: J_1, J_6 and J_{10}. The number of possible eligible job sets is the number of permutations of $n \leq 3$ jobs, which means that we need to analyse $\sum_{i=1}^{3} \frac{3!}{(3-i)!} = 15$ possible batches of jobs. By progressing in the schedule, the number of combination can grow very rapidly.

In this work, we represent all possible parallel and serialized schedules with a Direct Acyclic Graph called a *scheduling graph*. Each vertex in the graph represents the state of the GPU at the beginning of a *scheduling frame*, and each outgoing edge is labeled with the next batch of jobs to be scheduled. The main contribution of this paper is a method to perform the graph exploration within a reasonable amount of time.

In the remainder of this section, we will first formally describe a scheduling graph, and related notion. Further, we will outline the different approaches that allow us to build our scheduling graph, namely expansion and reduction. In the expansion phase, the graph considers new scheduling states according to task activations and scheduling decisions. During the reduction phase (which includes merging and splitting), we reduce the graph size to control the scheduling state design space.

3.2 Structure of the scheduling graph

Definition 3.1 (Scheduling Graph). We denote $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ the scheduling graph, where $\mathcal{V}$ is the set of vertices and $\mathcal{E}$ is the set of edges. A vertex $v_i \in \mathcal{V}$ represents the state of the system at the beginning of a scheduling frame, that is the state of each job (not yet arrived, ready, executed, etc.).

More formally, a vertex v_i is characterized by:

- $t(v_i)$ is a time instant;

[2]On Nvidia GPUs, this can be achieved by submitting every task to a separate stream.

- ready(v_i) is the set of jobs ready to be executed in the system at time $t(v_i)$, i.e. $\forall J \in \text{ready}(v_i)$ $\text{a}(J) \leq t(v_i)$ and J has not yet started execution;
- exe(v_i) is the set of jobs already completed at time $t(v_i)$;
- dom(v_i) is a set of time interval lengths that contains the time differences between the current vertex and all dominated vertices (see next section).

An edge $e_{i,j}$ from v_i to v_j is identified by the batch $\beta \subseteq \text{ready}(v_i)$[3] that is executed in the scheduling frame between $t(v_i)$ and $t(v_j)$. By definition, $\text{exe}(v_j) \equiv \text{exe}(v_i) \cup \beta$ and $\text{ready}(v_j) \equiv \text{ready}(v_i) \setminus \beta \cup \mathcal{A}(t(v_i), t(v_j))$.

Each path in the graph represents a schedule.

Definition 3.2 (Execution order path). A path $\pi = \{v_0, \ldots, v_n\}$ is a sequence of vertices connected by edges: $\forall i \in [0, n-1] \;\; \exists\, e_{i,i+1} \in \mathcal{E}$. We associate each path π with the list of jobs executed along the path π denoted $\text{exe}(\pi) = \text{exe}(v_n)$.

A vertex v_j is an *immediate successor* of vertex v_i if there exist an edge $e_{i,j}$. A vertex v_i is a *successor* of v_i if it exists a path between v_i and v_j. We denote by $\text{succ}(v_i)$ the set of immediate successors of vertex v_i.

A job set is schedulable if during the schedule graph generation we find a path for which all jobs have been scheduled and all of them respect their timing constraints.

Definition 3.3 (Schedulable Path). A path π is a schedulable path if and only if the current time of its last vertex v_i is $t(v_i) \geq H$ and all jobs in all vertices along the path finish their executions before their deadlines.

3.3 Expansion phase

The scheduling graph generation in described in Algorithm 1. It maintains an initially empty list of schedulable vertices $\mathcal{S}$, and a list $\mathcal{L}$ of vertices to be expanded. At each iteration it computes the successors of one vertex of this list, thus building the graph.

At the beginning, only one vertex v_0 is present, with $t(v_0) = 0$, $\text{exe}(v_0) = \emptyset$, and $\text{ready}(v_0) = \mathcal{A}(0, 0)$ (line 2). This initial vertex is also added to the list $\mathcal{L}$ of vertices to be expanded.

The algorithm selects a new vertex from the list, let it be v_i (line 5). First it checks for domination propagation. This allows to reduce the graph size under certain conditions(see next section for an explanation of the notion of the domination relation).

Then, in case the set of ready jobs of v_i is empty (i.e $\text{ready}(v_i) \equiv \emptyset$), we skip the idle time by *fast-forwarding* the time $t(v_i)$ to the next job arrival time (line 13, see next section for an explanation of this procedure).

Further, it computes all eligible batches of $\text{ready}(v_i)$ (see Definition 2.3), denoted as $\mathcal{P}(v_i)$ (line 15): it does this by invoking an *oracle* on each permutation of each distinct subset of $\text{ready}(v_i)$. The oracle tells us if a batch can execute in parallel on the GPU.

For each eligible batch $\beta \in \mathcal{P}(v_i)$ the algorithm generates a new vertex v_s having:

- $t(v_s)$ is the finishing time of the batch β, which is also the finishing time of the scheduling frame.

[3]With an abuse of notation, we extend classical operators on sets to batches with the natural semantics.

Algorithm 1 GPUSched

```
1:  S ← ∅                                          ▷ The list of schedulable vertices
2:  L ← {v_0}                                      ▷ The list of vertices to expand
3:  G ← {{v_0}, ∅}                                 ▷ Init the graph G
4:  while L ≠ ∅ do
5:      Extract v_i from L                         ▷ Vertex to expand
6:      for all Δ ∈ dom(v_i) do                    ▷ Check that domination holds
7:          if A(t(v_i), t(v_i) + Δ) ≠ ∅ then      ▷ Check Lemma 3.10
8:              Remove Δ from dom(v_i)
9:              Split(v_i, Δ)
10:         end if
11:     end for
12:     if ready(v_i) ≡ ∅ then                     ▷ If no ready jobs
13:         FastForward(v_i)                       ▷ Skip idle time
14:     end if
15:     Compute set of eligible batches P(v_i)
16:     for all β ∈ P(v_i) do
17:         create vertex(v_s)
18:             t(v_s) ← t(v_i) + C(β)             ▷ Compute frame end
19:             ready(v_s) ← ready(v_i) \ β ∪ A(t(v_i), t(v_s))
20:             exe(v_s) ← exe(v_i) ∪ β
21:             dom(v_s) ← dom(v_i)
22:         end create
23:         if no deadline miss ∈ [t(v_i), t(v_s)] then
24:             dominated ← false
25:             for all v_j ∈ V do                 ▷ Check dominance
26:                 if v_s ⊴ v_j then              ▷ If v_s is dominated
27:                     Merge(v_j, t(v_s) − t(v_j))
28:                     dominated ← true
29:                     break                      ▷ Stop checking domination
30:                 else
31:                     if v_j ⊴ v_s then          ▷ If v_s dominates v_j
32:                         Merge(v_s, t(v_j) − t(v_s))
33:                         discard v_j and its successors in G, L
34:                     end if
35:                 end if
36:             end for
37:             if not dominated then
38:                 Add v_s to G as successor of v_i
39:                 if t(v_s) < H then
40:                     Add v_s to L
41:                 else
42:                     Add v_s to S
43:                 end if
44:             end if
45:         end if
46:     end for
47: end while
```

- $\text{exe}(v_s) = \text{exe}(v_s) \cup \beta$;
- $\text{ready}(v_s) = \text{ready}(v_i) \setminus \beta_k \cup \mathcal{A}(t(v_i), t(v_s),)$;
- $\text{dom}(v_s) = \text{dom}(v_i)$ (see next section for an explanation of this part).

If the finishing time of the scheduling frame is beyond the deadline of any of the executed jobs, then a job has missed its deadline. The vertex is then *pruned* and will not be added to the graph (line 23).

The new vertex v_s is then analyzed in the *merging* phase to see if we really need to add it to the graph (Lines 25-37). If this new vertex is not *dominated*, it is added to the graph as a successor of vertex v_i and it is added to the list $\mathcal{L}$ of vertices that need to be further expanded.

3.4 Domination relation

To reduce the number of vertices that need to be explored, GPUSched introduces a domination relation between vertices.

Definition 3.4 (Domination relation.). A vertex v_j is dominated by vertex v_i, and we write $v_j \trianglelefteq v_i$, if and only if the following conditions are verified:

$$\mathsf{exe}(v_j) \equiv \mathsf{exe}(v_i) \ \wedge \ \mathsf{ready}(v_j) \equiv \mathsf{ready}(v_i) \ \wedge \ t(v_j) \geq t(v_i)$$

The domination relation tells us that the two vertices, v_j and v_i, represent two different states of the system in which exactly the same set of jobs has been processed ($\mathsf{exe}(v_j) \equiv \mathsf{exe}(v_i)$) and the same set of jobs is ready to be executed ($\mathsf{ready}(v_j) \equiv \mathsf{ready}(v_i)$). However, $t(v_i)$ is no later than $t(v_j)$, therefore the schedule represented by v_i is *no worse* than the schedule represented by v_j, as stated by the following lemmas.

Lemma 3.5. *Let us assume two vertices v_i and v_j such that $v_j \trianglelefteq v_i$. If a job's deadline is missed in the scheduling frame ending at v_j, then a deadline is also missed in the scheduling frame ending at v_i. Conversely, if no deadline is missed in v_i, then no deadline is missed in v_j.*

Proof. Descends directly from the definition of domination relation and scheduling frame. □

Lemma 3.6. *Let us assume two vertices v_i and v_j such that $v_j \trianglelefteq v_i$. Then, no jobs arrives between $t(v_j)$ and $t(v_i)$:*

$$\mathcal{A}(t(v_i), t(v_j)) \equiv \emptyset$$

Proof. By contradiction: a job arriving between $t(v_j)$ and $t(v_i)$ would be in $\mathsf{ready}(v_i)$ but not in $\mathsf{ready}(v_j)$ and this contradicts the hypothesis that $v_j \trianglelefteq v_i$. □

As a consequence of Lemma 3.6, the possible eligible batches in v_j are the same as those in v_i and therefore the immediate successors of v_j are *similar* to the immediate successors of v_i, in the sense that they execute the same jobs, however with a possibly earlier finishing time.

Definition 3.7 (Delta). Let us assume two vertices v_i and v_j such that $v_j \trianglelefteq v_i$. We denote by $\Delta_{j,i} = t(v_j) - t(v_i) \geq 0$ the interval of time between the two vertices.

Lemma 3.8. *Let v_i and v_j be two vertices of the scheduling graph, such that $v_j \trianglelefteq v_i$.*

Consider a vertex v'_j immediate successor of v_j, and let the edge $e(v_j, v'_j)$ be annotated by job batch $\beta_x \in \mathcal{P}(v_j)$ (we remind the reader that $\mathcal{P}(v_j)$ is the set of all eligible job batches in v_j).

Then, it exists a vertex $v'_i \in \mathsf{succ}(v_i)$ such that:

- *the edge $e(v_i, v'_i)$ is annotated with $\beta_x \in \mathcal{P}(v_i) \equiv \mathcal{P}(v_j)$;*
- *$t(v'_i) = t(v'_j) + \Delta_{j,i}$.*

Proof. Since $v_j \trianglelefteq v_i$, by definition $\mathsf{ready}(v_j) \equiv \mathsf{ready}(v_i)$, hence $\mathcal{P}(v_j) \equiv \mathcal{P}(v_i)$. Therefore, every successor of v_j has a corresponding vertex successor of v_i for which the same batch of jobs is executed. Since the same batch of jobs is executed in the two edges, their execution time must be the same, let it be x. Then, by definition of $\Delta_{j,i}$,

$$t(v'_i) = t(v_i) + x = t(v_j) + \Delta_{j,i} + x = t(v'_j) + \Delta_{j,i}.$$

□

Notice that, if the domination relation holds for two vertices $v_j \trianglelefteq v_i$, it may not hold for all of their successors. In fact, the set of their ready jobs can be different, because the respective scheduling frames end at different instants, as explained in the following example.

Example 3.9. Consider vertices v_i and v_j with $v_j \trianglelefteq v_i$, $\mathsf{ready}(v_i) = \mathsf{ready}(v_j) = \{J_1, J_2, J_3\}$, $t(v_i) = 200$, $t(v_j) = 220$, $\Delta_{j,i} = 20$.

Let us consider the job batch $\beta_x = \langle J_1, J_2 \rangle$, whose execution time is 50. Further, suppose that job J_4 arrives at time $\mathrm{a}(J_4) = 260$.

From v_i we can generate vertex v'_i with $t(v'_i) = 250$ and $\mathsf{ready}(v'_i) = \{J_3\}$ (because J_4 has not arrived yet). The set of eligible batches in v'_i is $\mathcal{P}(v'_i) = \{\langle J_3 \rangle\}$.

From v_j we can generate vertex v'_j with $t(v'_j) = 270$ and $\mathsf{ready}(v'_j) = \{J_3, J_4\}$, and the set of eligible batches is $\mathcal{P}(v'_j) = \{\langle J_3 \rangle, \langle J_4 \rangle, \langle J_3, J_4 \rangle\}$.

Notice that, v'_i does not dominate v'_j because job J_4 arrived between $t(v'_i)$ and $t(v'_j)$.

This means that, starting from v'_j we can generate states that cannot be generated from v'_i. Hence, we cannot eliminate node v'_j from the analysis, otherwise the exploration of the task graph will be incomplete. □

To avoid this kind of situations, we keep track of the difference in completion time between the dominated vertices and the dominating one with the list $\mathsf{dom}(v_i)$. This will allow us to check if a timing anomaly might occur in the future, and ensure that the domination relationship can be propagated to the successors.

The following lemma gives us the condition for the domination of the successors.

Lemma 3.10. *Let v_i and v_j be two vertices of the scheduling graph, such that $v_j \trianglelefteq v_i$. Let v'_j be an immediate successor of v_j, and let β_x be the batch of jobs executed in the scheduling frame between v_j and v'_j. Let v'_i be the immediate successor of v_i such that the same batch of jobs β_x is executed from v_i to v'_i.*

Then, $v'_j \trianglelefteq v'_i$ if and only if $\mathcal{A}(t(v'_i), t(v'_i) + \Delta_{j,i}) \equiv \emptyset$.

Proof. Let $C(\beta_x)$ denote the execution time of the batch of jobs β_x. By definition of the domination relation we have that $v'_j \trianglelefteq v'_i$ iff $\mathsf{ready}(v'_j) \equiv \mathsf{ready}(v'_i)$. The latter is true only if the set of jobs $\mathcal{A}(t(v_j), t(v_j) + C(\beta_x)) \equiv \mathcal{A}(t(v_i), t(v_i) + C(\beta_x))$. The interval $[t(v_i), t(v_j) + C(\beta_x)]$ can be split in three parts:

- Since $v_j \trianglelefteq v_i$, no job arrives in interval $[t(v_j), t(v_i)]$.
- All jobs arriving in $[t(v_j), t(v'_i)]$ belong to $\mathsf{ready}(v'_j)$ and to $\mathsf{ready}(v'_i)$; in fact, they arrive after the two scheduling frames have been started and they must wait for their completion before being considered for execution.

Algorithm 2 Merging two vertices

```
1: procedure MERGE(v_i, Δ)                    ▷ v_i is the dominating vertex
2:     dom(v_i) ← dom(v_i) ∪ {Δ}
3:     for all v_k ∈ succ(v_i) do              ▷ Propagate to successors
4:         if A(t(v_k), t(v_k) + Δ) ≠ ∅ then   ▷ If it is not dominant
5:             SPLIT(v_k, Δ)                   ▷ Split again the node
6:         else
7:             MERGE(v_k, Δ)                   ▷ Else, recursively propagate domination
8:         end if
9:     end for
10: end procedure
```

Algorithm 3 Split vertex

```
1: procedure SPLIT(v_k, Δ)                    ▷ v_k is the vertex to be split
2:     create vertex(v_s)
3:         t(v_s) ← t(v_k) + Δ
4:         ready(v_s) ← ready(v_k) ∪ A(t(v_k), t(v_k) + Δ)
5:         exe(v_s) ← exe(v_k)
6:         dom(v_s) ← dom(v_k)                 ▷ The other deltas are still respected
7:     end create
8:     add v_s to the successors of v_k
9:     add v_s to L
10: end procedure
```

- If a job arrives in $[t(v'_i), t(v'_j)]$, then it will be in $\text{ready}(v'_j)$ but it will not be in $\text{ready}(v'_i)$; therefore, v'_i does not dominate v'_j.
- If no job arrives in $[t(v'_i), t(v'_j)]$, then $\mathcal{A}(t(v'_i), t(v'_i)+\Delta_{j,i}) \equiv \emptyset$, and $\text{ready}(v'_j) \equiv \text{ready}(v'_i)$, hence $v'_j \trianglelefteq v'_i$.

□

3.5 Merging and splitting

Following Lemma 3.10, when merging a vertex v_j into a dominating vertex v_i, we need to store their time difference $\Delta_{j,i}$ in the list $\text{dom}(v_i)$. Then, vertex v_j is not added to the list of vertices to be explored (or it is removed if it was in that list). In the special case in which $\Delta_{j,i} = 0$, the two vertices are equivalent so we can simply remove v_j and skip the merging phase.

The merging procedure is shown in Algorithm 2. We first add Δ to the list of domination intervals $\text{dom}(v_i)$. There are two possibilities: either v_i is a new vertex with no successors yet, in which case the procedure completes immediately. If v_i is an already expanded vertex, then we try to recursively propagate the domination relation to all successors. For every successor of v_i, we first check if the condition of Lemma 3.10 is verified; if so, we merge this vertex as well. If not, we invoke procedure Split (described in Algorithm 3) which creates a new vertex that reflects the different state: in particular, the new vertex will contain a ready set $\text{ready}(v_s)$ which includes the newly arrived jobs.

Let us go back to Algorithm 1 to explain how the merge and split procedures are used in the graph generation. When generating the outgoing vertices from v_i in the exploration phase, for every

Algorithm 4 Fast-Forward procedure

```
1: procedure FASTFORWARD(v_i) ▷ v_i is the vertex to be advanced
2:     δ ← min{a(J) | J ∈ A(t(v_i), H)} − t(v_i)
3:     dom(v_i) ← ∅
4:     t(v_i) ← t(v_i) + δ
5:     ready(v_i) ← A(t(v_i), t(v_i))
6: end procedure
```

$\Delta \in \text{dom}(v_i)$ we check if there is a job arrival in the interval of length Δ after the finishing time (line 6); if a job arrival exist, we *split* again the schedule and generate one more vertex because the domination relation does not hold anymore between the successor of v_i and the corresponding dominated vertex.

After that, the algorithm checks for idle time. Suppose a vertex has $\text{ready}(v_i) \equiv \emptyset$ (line 13), and let $t(v_i) + \delta$ be the earliest arrival time. We know that for every $\Delta \in \text{dom}(v_i)$ there is no arrival in $t(v_i) + \Delta$ (because this check comes after lines 6-11), hence $\Delta \leq \delta$. Therefore, all vertices dominated by v_i experience the same idle time; as a consequence, we can skip this idle time in v_i and in all dominated vertices by advancing $t(v_i) \leftarrow t(v_i) + \delta$ and by setting all $\Delta \in \text{dom}(v_i)$ to 0 (see Algorithm 4). Notice that, when $\Delta \equiv 0$, the dominated vertices are equivalent to the dominating one, so we can simply discard them from the list: $\text{dom}(v_i) \leftarrow \emptyset$.

After creating the vertex v_s, and after ensuring that no deadline miss happens in the scheduling frame, we compare v_s against all existing vertices in the graph, looking for a domination (line 25). If v_s is dominated, we merge it into the dominating vertex (line 26). If v_s dominates an existing vertex v_j, we discard the entire subgraph starting from v_j and we merge it into v_s.

Before continuing the explanation of GPUSched, we present the following lemma that resumes the properties of the merge and split operations.

LEMMA 3.11 (ALL SUCCESSORS ARE DOMINATED). *Let v_i and v_j be two vertices of the scheduling graph, such that $v_j \trianglelefteq v_i$. Let us suppose that v_j is not discarded by the graph, and let v'_j be a successor of v_j. Then, there is a successor v'_i of v_i in the graph such that $v'_j \trianglelefteq v'_i$.*

PROOF. Since $v_j \trianglelefteq v_i$, GPUSched keeps the difference $\Delta = t(v_j) - t(v_i)$ stored in the list $\text{dom}(v_i)$. By induction. In the base step, let us consider the immediate successors. Suppose that v'_j is an *immediate* successor of v_j and let β be the job batch scheduled between v_j and v'_j. By Lemma 3.8, there is an immediate successor v'_i of v_i such that the same job batch β is scheduled between the two. By Lemma 3.10, there are 2 cases to consider:

- Case A: the interval $[t(v'_i), t(v'_i) + \Delta]$ does not contain any new job arrival. Then $v'_j \trianglelefteq v'_i$;
- Case B: the interval $[t(v'_i), t(v'_i) + \Delta]$ contains a new job arrival; in this case, GPUSched invokes the Split procedure which generates a new vertex v_s, successor of v_i, with:
  - $t(v_s) = t(v'_i) + \Delta = t(v'_j)$ (by construction of Δ);
  - $\text{ready}(v_s) = \text{ready}(v'_j)$;
  - $\text{exe}(v_s) = \text{exe}(v'_j) = \text{exe}(v'_i)$

  Hence, v_s is a successor of v_i and $v'_j \trianglelefteq v_s$.

By induction, we can apply the same reasoning to any non-immediate successors of v_i and v_j. We can conclude that, for any vertex that would have been generated directly or indirectly by v_j, we can find a successor of v_i that dominates it. □

We now analyse Algorithm 1, starting at line 37. If v_s is not dominated and it finishes within the hyperperiod, it is added to list $\mathcal{L}$, otherwise it is the last vertex of a schedulable path and it is added to the list of vertices $\mathcal{S}$. The algorithm terminates when the expansion list $\mathcal{L}$ is empty and the schedulable solutions are in $\mathcal{S}$.

Each schedulable solution produced by GPUSched is one feasible schedule that respects all the job deadlines when executed by the *Parallel Batch Scheduler* described in Section 2.4. To implement such schedule, we select one of the vertices from $\mathcal{S}$, let it be v_s and we reconstruct the path leading to v_s from the graph. Since every vertex in the path represents the state of the schedule at the beginning of a scheduling frame, and each edge represents the batch of jobs to be executed, we can build a *scheduling table* that will be given as input to the *Parallel Batch Scheduler* for submitting the batches in the given order and at the given time to the GPU block scheduler. We now describe the algorithm through an example.

Example 3.12. Let us consider the example illustrated in Figure 3 that shows the expansion and merging operations on the graph. We consider the following sequence of jobs: $J_3(a = 6, C = 5), J_4(a = 6, C = 4), J_5(a = 17, C = 6), J_6(a = 18, C = 5)$. For the sake of simplicity, we assume that execution time of parallel tasks is equal to their execution time when executing in isolation.

For vertices v_1 and v_2, we suppose that $t(v_1) = 10$, $t(v_2) = 7$ (finishing times are shown in parenthesis in the figure). Also, suppose that $\text{exe}(v_1) \equiv \text{exe}(v_1)$ and $\text{ready}(v_1) \equiv \text{ready}(v_2) \equiv \{J_3, J_4\}$. Therefore $v_2 \trianglelefteq v_1$ and v_1 is merged into v_2 with $\Delta_{2,1} = 3$.

The algorithm GPUSched explores the eligible batches for the ready jobs J_3 and J_4 starting from v_2. When J_3 is executed first, it creates vertex v_3, finishing at time instant $t(v_3) = 12$. When J_4 is executed first, it creates v_4, finishing at time instant $t(v_4) = 11$. It is also possible to submit J_3 and J_4 in parallel with submission J_3/J_4 and J_4/J_3, creating v_5 and v_6, both finishing at time instant 15, so v_5 is merged with v_6. The time difference $\Delta_{2,1}$ stored in $\text{dom}(v_2)$ is propagated to all of its successors. In vertex v_3, we have only J_4 as a ready job; therefore, it is executed, creating vertex v_7, ending at time instant $t(v_7) = 16$. Similarly, from v_4, only J_3 is ready to execute, leading to the creation of vertex v_8, finishing at time instant $t(v_8) = 16$. Vertices v_7 and v_8 are therefore merged.

When GPUSched analyses v_6, it finds an empty ready job set, therefore it skip the idle time until $t = 17$ and $\text{dom}(v_6) \leftarrow \emptyset$. From there, only J_5 is ready, so it generates vertex v_9 with $t(v_9) = 23$.

Then, it analyses v_8. It notices that $t(v_8) + \Delta_{2,1} = 18 > 17$, therefore the domination with v_2 successors is not valid anymore. It performs a split to reflect this generating vertex v'_8 with $t(v'_8) = 16 + \Delta_{2,1} = 19$. Then, like v_6, it skips the idle time from 16 to 17, and generates vertex v_{16} which is immediately merged with v_9.

v'_8 has both J_5 and J_6 in its ready job list, and it can explore their different possible submission profiles, generating vertices v_{11}, v_{12} and v_{14}. The first one is merged into v_9 with a $\Delta_{9,11} = 2$; from v_9, the last job J_6 is executed to create v_{13} with $t(v_{13}) = 28$.

Starting from v'_8, it is also possible to submit J_6 and further J_5, creating v_{14} and v_{15} with completion times of 24 and 30. The parallel submission of J_5 and J_6 from v'_8 allows having the shortest completion time at 25 in vertex v_{12}. This vertex dominates all others as it executes all jobs and completes at time instant 25.

Theorem 3.13. *Consider a set of periodic tasks to be executed on a GPU, with hyperperiod H, which produces the set of jobs* J *in* $[0, H]$.

Let $\mathcal{S}$ be the set of schedules produced by Algorithm GPUSched. If $\mathcal{S} \equiv \emptyset$ then no schedulable solution can be found for PBS.

Proof. By contradiction. Suppose a schedule exists for the PBS, but GPUSched is unable to find it. First, notice that all possible combinations and permutations of jobs that can execute completely in parallel or serialized (as required by the *Parallel Batch Scheduler*) are generated by GPUSched. However, some of these combinations may results in vertices that are discarded from the graph, either because they are *pruned*, or because they are *merged*: any successor from these vertices will not end up in $\mathcal{S}$.

A vertex v_j is pruned if a deadline is missed: any path containing v_j is not schedulable, and any successor of v_j will not be part of a feasible schedule. A vertex v_j is merged if $\exists v_i, v_j \trianglelefteq v_i$. No successor of v_j will be in S. However, from Lemma 3.11, for any successor v'_j of v_j (immediate or not), there is a successor v'_i of v_i that dominates it. From Lemma 3.5, if no deadline is missed in v'_j, then no deadline is missed in v'_i. Hence, if there is a feasible successor of v_j, there is a successor of $v_i \in \mathcal{S}$, which is a contradiction. □

One consequence of Theorem 3.13 is that GPUSched is **optimal** for PBS: it analyses all possible schedules that can be generated by PBS, and selects the schedulable ones.

4 IMPLEMENTABILITY OF PBS SCHEDULER

4.1 PBS Scheduler over GPU Internals

We illustrate the usefulness of our approach by presenting a configuration tool that takes as input the schedule generated by GPUSched, builds the scheduling table, and submits the kernels according to the order established in the table.

To achieve this, we assign a CUDA stream to every task for all jobs submissions related to that task. All CUDA streams have the same priority. During the initialization phase, our PBS scheduler parses a trace file containing the activation time for every batch of jobs and the list of considered kernels.

Our PBS scheduler then loops over the different table entries. At each iteration, it submits the corresponding jobs (kernels) to the GPU, with each job in its own CUDA stream. Further, it invokes `cudaDeviceSynchronize()` to enforce their synchronization, and waits for the batch completion. There are two possible strategies at this point. In the first strategy, a timer sleeps until the next entry activation, strictly adhering to the prebuilt schedule. Alternatively, the scheduler might wait for the latest submission within the batch to be released, allowing it to start earlier and reclaim the unused execution time for the next batches. This tool and our schedulability analysis tool are available on https://gitlab.cristal.univ-lille.fr/gpusched.

4.2 The oracle

In this work, we assume there exists an oracle capable of determining whether kernels can be executed in parallel or not, and if so,

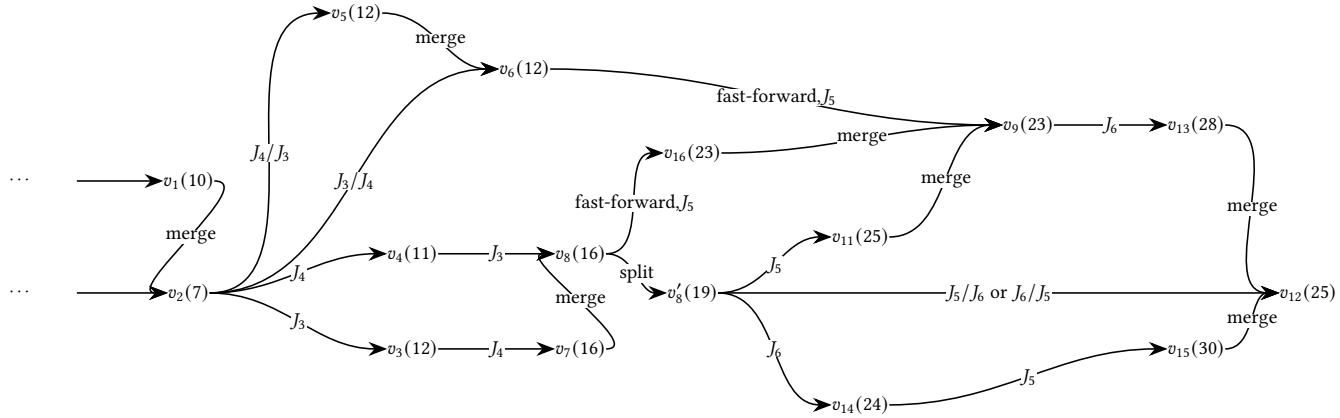


Figure 3: Example of merge

to define their completion time. For the first part, we take into account the kernel submission parameters (number of blocks, threads, registers, and shared memory). We use the "revealed hidden details" reported in the related work. This part of the oracle is integrated into our schedulability analysis tool available online.

Furthermore, for every combination of *parallelizable kernels*, we build benchmarking scenarios. In each scenario, kernels are submitted to the GPU, each in its own stream. All streams have the same priority, similar to our runtime. We then invoke `cudaDeviceSynchronize` and measure the elapsed time between kernel submissions and the return of `cudaDeviceSynchronize`. We repeat this operation multiple times. From this, one can extract either worst-case, typical, or the average scenario according to the required target guarantees.

5 STATE OF THE ART

In this work, we focus on scheduling issues within GPUs. This has been a hot topic in the real-time systems community during the last few years, with multiple efforts aimed at providing real-time guarantees for parallel application running on GPUs. We can classify related work on GPU scheduling into three classes: papers that focus on (1) reverse engineering the GPU to extract internal features, (2) works to estimate execution time profiles, and (3) works interested in the scheduling within the GPU itself.

The internal design of many GPUs (NVIDIA, AMD) is closed-source due to intellectual property concerns. Multiple efforts have focused on reverse engineering their internal scheduling mechanisms. For Nvidia GPUs, a set of experiments have been carried out in [2] and the authors have revealed important aspects of a GPU scheduler. The authors of [14] continued in these efforts to model the hierarchical scheduling structure of NVIDIA GPUs. AMD GPUs are presented as a viable alternative to execute safety critical applications in [16]. The authors of [17] identified the different points that might affect the performance of a real-time task executing on AMD GPUs, and they reported their internal scheduling rules.

With regards to estimating the execution time within the GPU, multiple efforts are being made using different techniques. The work in [9] focuses on building Control Flow Graphs for a single GPU warp. The authors of [11] employ a different approach to estimate the execution time of a kernel using machine learning methodologies. Authors of [6] investigate the interference between CPU cores and integrated accross different Integrated GPU. In [5], an exhaustive set of experiments is conducted across multiple generations of NVIDIA Jetson boards to analyze the evolution of memory contention. In [22], authors provide a method for producing *stress* programs that intentionally contend for GPU resources in order to enable more confident measurement-based WCET estimations.

In order to ensure the schedulability of real-time applications, it is required to precisely control their execution order, and this often means to bypass the closed-source internal mechanisms of the GPU. Several works proposed building real-time schedulers on top of the GPU internal schedulers [4, 7, 23]. The GPU is treated as a single resource scheduled using non-preemptive algorithms via software locks [7], or as a preemptive resource by modifying the NVIDIA proprietary drivers [4]. With the continuous increase in GPU compute power, it becomes urgent to consider parallel execution within the GPU to improve its utilization. This can be achieved through partitioning, or through parallel workload submissions techniques such as Spatial Multikernel or Spatial Partitioning [1] and Simultaneous Multikernel (SMK) [21]. Some works [1, 19, 24, 26] partition the GPU based on workload characteristics, such as compute-bound, memory/interconnect-bound, and problem-size-bound. They evaluate various compile-time spatial partitioning schemes. Recent works [18, 27], SM-level scheduling is used to improve GPUs resources utilization. This can be even more important as recent work allows to partiton recent NVIDIA GPUs' SMs transparently using library titled *libsmctrl* [3].

Another crucial aspect of our paper is the reduction of schedulability analysis complexity for a set of real-time tasks executing in parallel on the GPU. Schedulability analysis for a set of independent non-preemptive tasks has always been a hot topic in real-time systems. However, the nature of the target platforms can significantly influence the analysis methodologies. Classical schedulability analysis for multicores is typically classified to those for partitioned and global schedulers. Various schedulability tests have been proposed

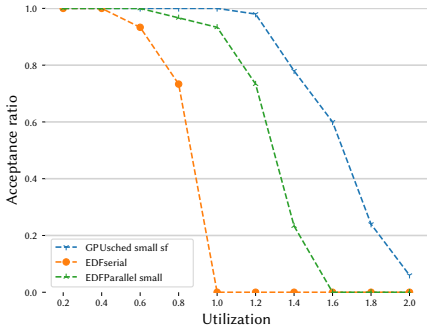
for both partitioned and global scheduling strategies. A recent review and pointers to these analysis can be found in [20].

Recently, a novel schedulability worst-case response time analysis technique based on exploring all possible execution states has been proposed in [12, 13], called schedule abstraction graphs (SAG). SAG is a promising approach that balances schedulability analysis pessimism and temporal complexity. It uses graphs to explore the space of all possible schedules, utilizing state abstraction. The goal is to build possible execution states, to derive some real-time properties such as WCRT. It has been proven efficient compared to related work.

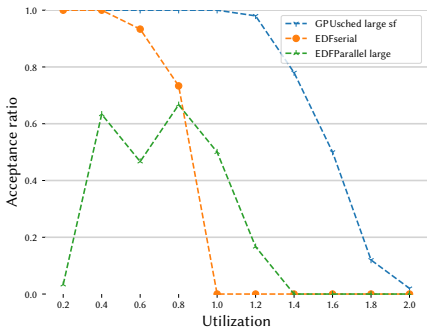
Our work employs a similar approach but with a completely different methodology and objectives. Instead of finding a real-time property by identifying the worst scenario, our scheduling graph enforces scheduling decisions to find the most suitable sequence that allows the system to be schedulable. These decisions are taken during the construction of the scheduling graph, unlike SAG, which *simulates* a scheduler. In our scheduling graph, the process stops as soon as the system finds a single feasible execution trace from time 0 to the hyper-period, while SAG must explore all system states before deriving any property. Additionally, scheduling within a GPU is significantly different from the related work of SAG. GPU scheduling is more akin to GANG scheduling as a task might execute in parallel on different SMs compared to global scheduling, where a task always execute on a single core. Henceforth, the core state does not depend only on the local core execution state, but on the state on all cores at the same time. Moreover, the exploration space in SAG depends only on task execution, whereas our approach also accounts for interference. SAG produces the a single property, while our approach generates a schedule. Our approach and SAG shares the idea of pruning and merging to avoid combinatorial state space explosion, while these two operations are still very different.

6 RESULTS AND DISCUSSIONS

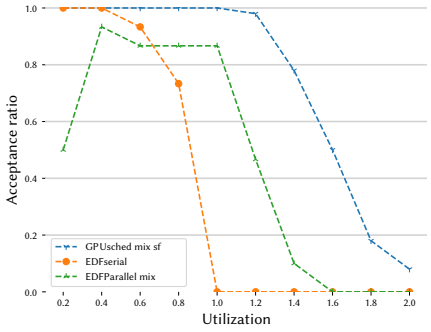
In this section, we evaluate the efficacy of PBS's schedulability analysis in comparison to exclusive-serialized and completely parallel executions. In the first approach (EDFserial) each task uses the GPU as a single resource, and jobs are scheduled in EDF order. In contrast, in the completely parallel approach (EDFParallel) all eligible jobs are executed in the EDF order: the algorithm checks the eligibility of each job in the EDF order, and if a job is eligible it is included in the batch. This algorithm is completely deterministic as it always selects the same batch from the same ready set.



(a) Schedulability for small slowdown factors.



(b) Schedulability for large slowdown factors.

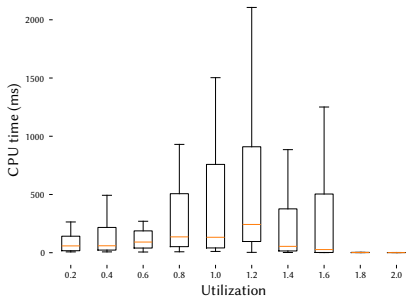


(c) Schedulability for mixed slowdown factors.

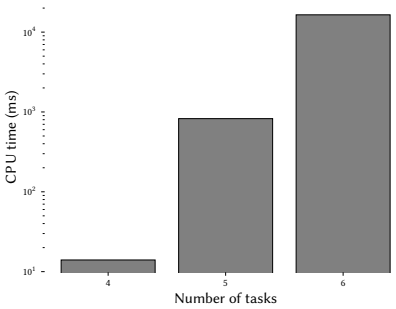
Figure 4: Schedulability ratio: varying utilization.

6.1 Parameters of the experiments

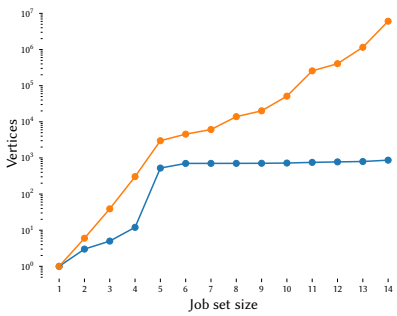
We consider a GPU consisting of 8 SMs, each SM may contain up to 1024 threads and 32 blocks. The task set generation process begins with two inputs: the target utilization U and the number of tasks n. For each task set, we generate n tasks with their parameters to meet the target total utilization. Task periods are randomly selected from a predefined list spanning from 400 to 1600. This strategy mitigates the generation of excessively large hyper-periods and subsequently reduces the overall number of jobs. We employ the UUniFast-Discard algorithm [8] to generate the utilizations for n tasks. The execution time for each task is computed as the multiplication of its utilization and period, representing the time required for task execution without any interference. Consequently, the effectively scheduled workload may be larger than the baseline utilization, as tasks encounter interference when executed in parallel. Within the uniform approach, the occupancy vector for each task is generated by multiplying the corresponding GPU capacity by the task utilization. In terms of interference, we explore three scenarios. Firstly, we simulate relatively small interference by inflating execution time by slowdown factors in $[1, 1.4]$. Secondly, we consider larger slowdown impact, randomly chosing slowdown factor in $[1.7, 1.9]$. Finally, we introduce completely random slowdown factors in $[1.1, 1.9]$.



(a) Running time of GPUSched for task sets of 5 tasks.



(b) Running time as a function of the number of tasks.



(c) Size of the graph with and without vertex merging.

Figure 5: Running time and graph size.

6.2 Simulations and discussions

In Figure 4 we show the schedulability ratio obtained by analysing tasks sets consisting of 5 tasks, with varying utilizations from 0.2 to 2.0, for different interference scenarios. For each point in the graphs we randomly generated 50 sets, and we run GPUSched on each of them. On the Y axis we report the percentage of schedulable task sets with GPUSched, EDFserial (where no parallelism is permitted) and EDFParallel. Clearly, GPUSched largely dominates the others as the utilization increases. We notice that EDFParallel is a good solution when the interference is small; however, when the interference increases, its behavior becomes much more unpredictable.

In Figure 5a we show the execution time distribution of the GPUSched algorithm running on task sets of 5 tasks each in the large slowdown scenarios. The execution time is in the order of 1 second, with a variability due to the random parameters of the task set. We also notice that the variability is much smaller for low and for high utilization: this is due to the fact that, for small utilizations, a lot of vertices are merged together due to the idle time; and for large utilizations, many vertices as pruned as non-schedulable. The distribution is very similar for the other scenarios (not shown here).

In Figure 5b we show the average analysis time of GPUSched as a function of the number of tasks, for an utilization of $U = 1$. The Y axis is in logarithmic scale (in ms): the running time of the algorithm is highly dependent in the number of tasks. In fact, the algorithm must generate all possible eligible batches at the beginning of the scheduling frame, and this means testing all permutations of all subsets of the ready job set, whose maximum size is equal to the number of tasks. For practical purposes, current use of modern GPU consists in executing a few large tasks consisting of many parallel threads each. We believe that a limit of 6 tasks is still reasonable for many modern applications. It is currently impossible to use our algorithm for task sets with more than 6 or 7 tasks.

In Figure 5c, we show how the number of vertices in the graph grows during the analysis for 5 tasks and utilization equal to 1, when we merge the vertices (blue line) and when we do not merge them (red line). On the X axis, we show the job number as analysis progresses; please notice that the Y-axis is in logarithmic scale. The number of vertices generated very quickly reaches 10^7 with only 14 jobs when we do not merge vertices; with the merge operation enabled, the number of vertices remains in the order of 10^3 as the analysis progresses. This practically demonstrates the effectiveness of the merge operation for reducing the size of the graph: in fact, our algorithm is able to analyse tasks sets with up to 6 tasks thanks to vertex merging. We stopped the analysis after job 14 to avoid an excessive large number of vertices in the no-merge configuration.

7 CONCLUSION

We proposed a novel approach for executing real-time tasks in parallel within the GPU, aiming to enhance GPU resource utilization while guaranteeing the respect of real-time constraints and mitigating interference through the use of scheduling graphs. Our work addresses the challenges of abstracting GPU execution and explores the different scheduling states through our scheduling graph. A key challenge in our proposed techniques is the reduction of the size of the state graphs, without eliminating feasible schedules.

We are working on ways to improve our PBS scheduler. In our proposed approach, once a batch of jobs is submitted to the GPU the scheduler has to wait for the completion of the last job before submitting new jobs to the GPU. This may lead to a waste of resources, because new arriving jobs are blocked until the completion of the batch even if enough resources are available. However, submitting new jobs while a batch is executing may add interference and modify the completion time; therefore, our execution model is not valid anymore. We are currently working toward extending the graph representation and the domination relationship to take into account this new scheduling model.

REFERENCES

[1] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte. The case for gpgpu spatial multitasking. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–12, 2012. doi:10.1109/HPCA.2012.6168946.

[2] Tanya Amert, Nathan Otterness, Ming Yang, James H. Anderson, and F. Donelson Smith. Gpu scheduling on the nvidia tx2: Hidden details revealed. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 104–115, 2017. doi:10.1109/RTSS.2017.00017.

[3] Joshua Bakita and James H. Anderson. Hardware compute partitioning on nvidia gpus*. In *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 54–66, 2023. doi:10.1109/RTAS58335.2023.00012.

[4] Nicola Capodieci, Roberto Cavicchioli, Marko Bertogna, and Aingara Paramakuru. Deadline-based scheduling for gpu with preemption support. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 119–130, 2018. doi:10.1109/RTSS.2018.00021.

[5] Nicola Capodieci, Roberto Cavicchioli, Ignacio Sañudo Olmedo, Marco Solieri, and Marko Bertogna. Contending memory in heterogeneous socs: Evolution in nvidia tegra embedded platforms. In *2020 IEEE 26th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–10, 2020. doi:10.1109/RTCSA50079.2020.9203722.

[6] Roberto Cavicchioli, Nicola Capodieci, and Marko Bertogna. Memory interference characterization between cpu cores and integrated gpus in mixed-criticality platforms. In *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–10, 2017. doi:10.1109/ETFA.2017.8247615.

[7] Glenn A Elliott, Bryan C Ward, and James H Anderson. Gpusync: A framework for real-time gpu management. In *2013 IEEE 34th Real-Time Systems Symposium*, pages 33–44. IEEE, 2013.

[8] P. Emberson, R. Stafford, and R.I. Davis. Techniques for the synthesis of multiprocessor tasksets. In *1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, pages 6–11, July 2010.

[9] Louison Jeanmougin, Thomas Carle, Pascal Sotin, and Christine Rochange. Warp-Level CFG Construction for GPU Kernel WCET Analysis. In Peter Wägemann, editor, *21st International Workshop on Worst-Case Execution Time Analysis (WCET 2023)*, volume 114, pages 1:1–1:13, Vienne, Austria, July 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://hal.science/hal-04171474, doi:10.4230/OASIcs.WCET.2023.1.

[10] Alessio Masola, Nicola Capodieci, Roberto Cavicchioli, Ignacio Sanudo Olmedo, and Benjamin Rouxel. Memory-aware latency prediction model for concurrent kernels in partitionable gpus: Simulations and experiments. In Dalibor Klusáček, Julita Corbalán, and Gonzalo P. Rodrigo, editors, *Job Scheduling Strategies for Parallel Processing*, pages 46–73, Cham, 2023. Springer Nature Switzerland.

[11] Alessio Masola, Nicola Capodieci, Benjamin Rouxel, Giorgia Franchini, and Roberto Cavicchioli. Machine learning techniques for understanding and predicting memory interference in cpu-gpu embedded systems. In *2023 IEEE 29th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 147–156, 2023. doi:10.1109/RTCSA58653.2023.00026.

[12] Mitra Nasri and Bjorn B. Brandenburg. An exact and sustainable analysis of non-preemptive scheduling. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 12–23, 2017. doi:10.1109/RTSS.2017.00009.

[13] Mitra Nasri, Geoffrey Nelissen, and Björn B. Brandenburg. A response-time analysis for non-preemptive job sets under global scheduling. In *Euromicro Conference on Real-Time Systems*, 2018. URL: https://api.semanticscholar.org/CorpusID:21708009.

[14] I. S. Olmedo, N. Capodieci, J. L. Martinez, A. Marongiu, and M. Bertogna. Dissecting the cuda scheduling hierarchy: a performance and predictability perspective. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 213–225, 2020. doi:10.1109/RTAS48715.2020.000-5.

[15] Ignacio Sañudo Olmedo, Nicola Capodieci, and Roberto Cavicchioli. A perspective on safety and real-time issues for gpu accelerated adas. In *IECON 2018 - 44th Annual Conference of the IEEE Industrial Electronics Society*, pages 4071–4077, 2018. doi:10.1109/IECON.2018.8591540.

[16] Nathan Otterness and James H. Anderson. AMD GPUs as an Alternative to NVIDIA for Supporting Real-Time Workloads. In Marcus Völp, editor, *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, volume 165 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:23, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECRTS.2020.10, doi:10.4230/LIPIcs.ECRTS.2020.10.

[17] Nathan Otterness and James H. Anderson. Exploring amd gpu scheduling details by experimenting with "worst practices". In *Proceedings of the 29th International Conference on Real-Time Networks and Systems*, RTNS '21, page 24–34, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3453417.3453432.

[18] Sujan Kumar Saha, Yecheng Xiang, and Hyoseung Kim. Stgm: Spatio-temporal gpu management for real-time tasks. In *2019 IEEE 25th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–6, 2019. doi:10.1109/RTCSA.2019.8864564.

[19] 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.

[20] Micaela VERUCCHI and Marko BERTOGNA. A comprehensive analysis of dag tasks: solutions for modern real-time embedded systems.

[21] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo. Simultaneous multikernel gpu: Multi-tasking throughput processors via fine-grained sharing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 358–369. IEEE, 2016. doi:10.1109/HPCA.2016.7446078.

[22] Tyler Yandrofski, Jingyuan Chen, Nathan Otterness, James H. Anderson, and F. Donelson Smith. Making powerful enemies on nvidia gpus. In *2022 IEEE Real-Time Systems Symposium (RTSS)*, pages 383–395, 2022. doi:10.1109/RTSS55097.2022.00040.

[23] Houssam-Eddine Zahaf and Giuseppe Lipari. Design and analysis of programming platform for accelerated gpu-like architectures. In *Proceedings of the 28th International Conference on Real-Time Networks and Systems*, RTNS '20, page 1–10, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3394810.3394826.

[24] Houssam-Eddine Zahaf and Giuseppe Lipari. Design and analysis of programming platform for accelerated gpu-like architectures. In *Proceedings of the 28th International Conference on Real-Time Networks and Systems*, RTNS 2020, page 1–10, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3394810.3394826.

[25] Houssam-Eddine Zahaf, Ignacio Sanudo Olmedo, Jayati Singh, Nicola Capodieci, and Sebastien Faucou. Contention-aware gpu partitioning and task-to-partition allocation for real-time workloads. In *Proceedings of the 29th International Conference on Real-Time Networks and Systems*, RTNS '21, page 226–236, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3453417.3453439.

[26] Houssam-Eddine Zahaf, Ignacio Sanudo Olmedo, Jayati Singh, Nicola Capodieci, and Sebastien Faucou. Contention-aware gpu partitioning and task-to-partition allocation for real-time workloads. In *Proceedings of the 29th International Conference on Real-Time Networks and Systems*, RTNS '21, page 226–236, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3453417.3453439.

[27] An Zou, Jing Li, Christopher D. Gill, and Xuan Zhang. Rtgpu: Real-time gpu scheduling of hard deadline parallel tasks with fine-grain utilization. *IEEE Transactions on Parallel and Distributed Systems*, 34(5):1450–1465, 2023. doi:10.1109/TPDS.2023.3235439.