Independent tasks on 2 resources with co-scheduling effects

Lionel Eyraud-Dubois, Cristiana Bentes

Independent tasks on 2 resources with co-scheduling effects

Lionel Eyraud-Dubois[a,*], Cristiana Bentes[b]

[a] *Inria, University of Bordeaux, Bordeaux, France, 33000*
[b] *Systems Engineering Department, State University of Rio de Janeiro, Brazil, 20550-900*

Abstract

Concurrent kernel execution is a relatively new feature in modern GPUs, which was designed to improve hardware utilization and the overall system throughput. However, the decision on the simultaneous execution of tasks is performed by the hardware with a leftover policy, that assigns as many resources as possible for one task and then assigns the remaining resources to the next task. This can lead to unreasonable use of resources. In this work, we tackle the problem of co-scheduling for GPUs with and without preemption, with the focus on determining the kernels submission order to reduce the number of preemptions and the kernels makespan, respectively. We propose a graph-based theoretical model to build preemptive and non-preemptive schedules. We show that the optimal preemptive makespan can be computed by solving a Linear Program in polynomial time, and we propose an algorithm based on this solution which minimizes the number of preemptions. We also propose an algorithm that transforms a preemptive solution of optimal makespan into a non-preemptive solution with the smallest possible preemption overhead. We show, however, that finding the minimal amount of preemptions among all preemptive solutions of optimal makespan is a NP-hard problem, and computing the optimal non-preemptive schedule is also NP-hard. In addition, we study the non-preemptive problem, without searching first for a good preemptive solution, and present a Mixed Integer Linear Program solution to this problem. We performed experiments on real-world GPU applications and our approach can achieve optimal makespan by preempting 6 to 9% of the tasks. Our non-preemptive approach, on the other side, obtains makespan within 2.5% of the optimal preemptive schedules, while previous approaches exceed the preemptive makespan by 5 to 12%.

Keywords: GPU Concurrency, Scheduling, Preemption

1. Introduction

Graphics Processing Units (GPUs) are throughput-oriented co-processors that enable faster and energy-efficient execution for many classes of applications. Modern GPUs are witnessing exciting advances in their microarchitecture and a rapid increase in the amount of computing resources. With each new generation

*Corresponding author
Email addresses: lionel.eyraud-dubois@inria.fr (Lionel Eyraud-Dubois), cris@eng.uerj.br (Cristiana Bentes)

of GPUs, there are expressive growth in core count, shared memory, registers, and memory bandwidth. To avoid keeping these growing resources underutilized and improve performance, concurrent kernel execution (CKE) has been proposed and has recently received significant attention. CKE allows multiple tasks, called *kernels* in the GPU environment, to execute simultaneously on the GPU. Some previous approaches showed that CKE can improve GPU throughput and resource utilization [5, 2, 6, 1], however, CKE is quite a recent feature in GPU architectures.

Although the problem of co-scheduling applications on CPU multicore nodes has been well studied in the past [7, 8, 9], they are not directly applicable to the GPU environment. While the multicore resources can be spatially and fairly divided among the CPU tasks, the GPU resource allocation is performed by the hardware, that assigns as many resources as possible for one task and then assigns the remaining resources to the next task, if there are sufficient *leftover* resources [10]. This allocation policy can lead to an unreasonable use of the available resources, degrading the overall system throughput. For example, suppose that there are 4 kernels A, B, C and D, that arrive at the GPU at the same time, with execution times and resource consumption as described in the table of Figure 1. We can observe that the launching order of these kernels can impact their makespan. In Figure 1(a), the submission order is A, B, C and D, and according to the leftover scheduling, when B finishes, C is submitted, but D has to wait since its amount of requested resource is not yet available. In Figure 1(b), we consider a different order to submit the kernels, C, A, B and D, and we can observe a decrease in the kernels makespan from 170 to 140.

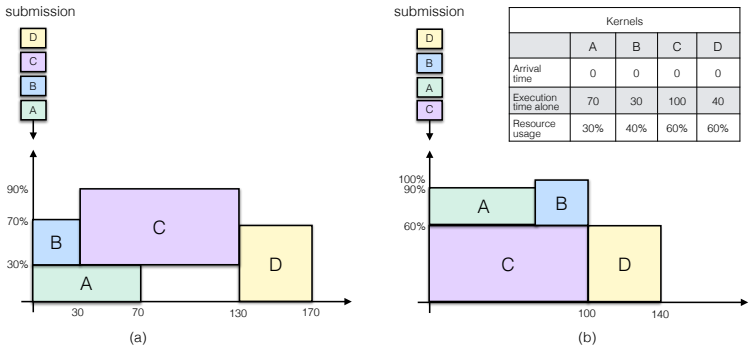


Figure 1: A set of four kernels is submitted to the GPU in two orders.

In this way, the launching of kernels to the GPU has to be wisely performed in order to avoid an unbalanced occupation of the GPU resources and its consequent negative effect on the system performance. Some previous works [11, 12] performed an extensive analysis of concurrent kernel execution on GPU and used machine learning techniques to understand and predict how the resource usage of the kernels impacts the interference in their co-execution. In this work, we exploit this interference profile and study the problem of co-scheduling for GPUs on high competition future scenarios where the GPU will behave as multiprogrammed devices, such as CPUs are in the present. We propose two graph-based co-scheduling solutions, a preemptive

and a non-preemptive one, that define the kernels submission order focusing respectively on reducing the number of preemptions performed and the kernels makespan. More formally, we deal with the following problem: given a co-scheduling speedup matrix S, where $S_{i,j}$ is the the speed at which kernel i makes progress when running with j, and the duration of each kernel, what is the best way to co-schedule them, in order to minimize their makespan. We first propose a Linear Programming model to generate an optimal preemptive schedule that minimizes the makespan of the kernels. The optimal solution then produces a co-execution graph G where each node i represents one kernel and an edge connecting i to j indicates that kernels i and j can co-execute. From G, we propose the algorithm CaterpillarSplit to generate the kernel submissions with the minimum number of preemptions. To generate non preemptive schedules, we also propose the use of a Weighted Path Cover algorithm to generate a non-preemptive kernel submission with the smallest possible overhead. However, we also show that the complete problems to minimize the number of preemptions or the makespan of a non-preemptive schedule directly from the speedup matrix S are NP-hard. These algorithmic contributions are summarized on Figure 2.

We used 60 kernels from real-world applications obtained from the main benchmark suites for evaluating GPUs, Rodinia [13], Parboil [14] and SHOC [15] and observed that our algorithms produce very efficient schedules. In the preemptive case, we obtain the optimal makespan by preempting between 5% and 10% of the kernels depending on the setting. In the non-preemptive case, we obtain schedules with an overhead of at most 2% over the preemptive makespan, whereas previous approaches yield an overhead between 5% and 12% depending on the context.

The rest of the paper is organised as follows. In Section 2, we provide additional context about the GPU execution model and review related works. In Section 3, we present the notations and the scheduling model used throughout the paper. Section 4 is focused on the preemptive case, where we describe how to obtain preemptive schedules with minimal makespan, and how to minimize the number of preemptions. Section 5 addresses the non-preemptive case, with a description and approximation analysis of the corresponding algorithms. Finally, Section 6 gives an experimental validation of our algorithms on realistic instances.

2. Context

2.1. GPU Execution Model and Preemption Possibilities

Along the paper we use NVIDIA and CUDA terminology, since they are the most popular architecture and programming model for GPUs. However, the architecture and programming model descriptions can also be applied to other GPU vendors. The GPU architecture comprises a number of Streaming Multiprocessors (SMs), where each SM contains a number of cores that share a scratchpad memory called shared memory, a register file, and a L1 cache. In the GPU programming model, kernels are launched from the CPU to execute on the GPU. Each kernel is composed by a number of threads. The threads are grouped into *thread*

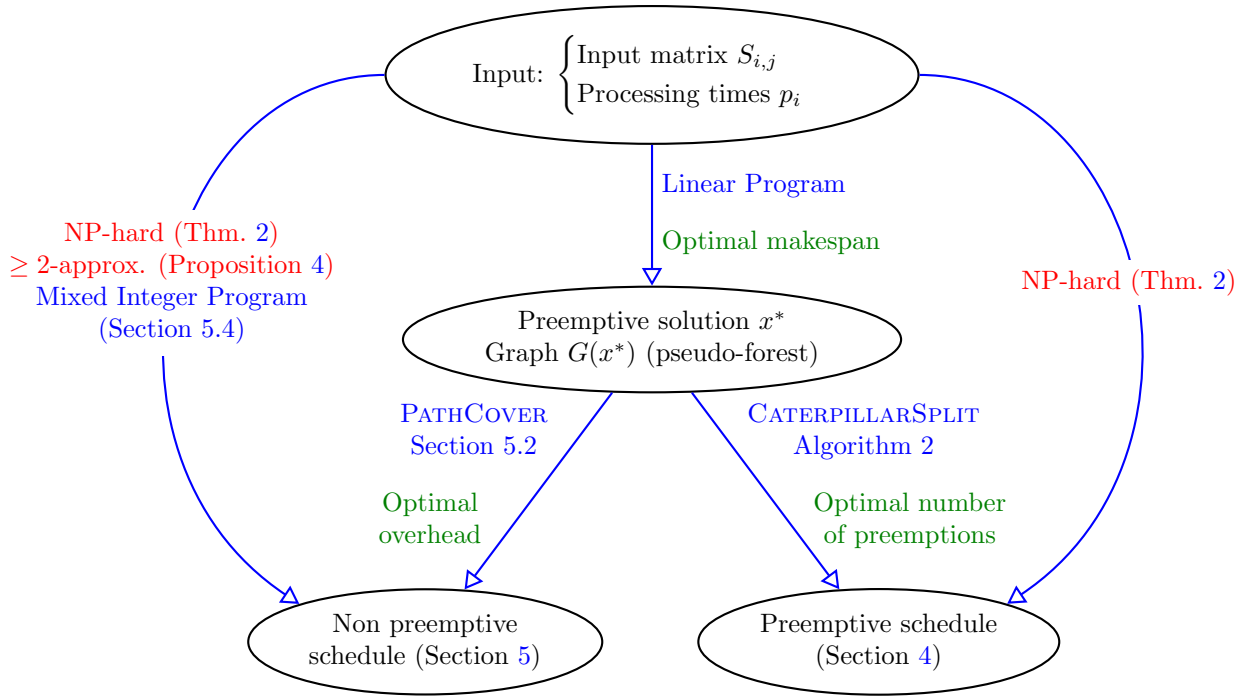


Figure 2: Summary of contributions. The picture uses blue text for algorithms, red text for negative results, and green text for positive results.

blocks, in a way that each block is dispatched by the hardware to be executed in one SM. The programmer, however, has no control over how the thread blocks are dispatched to execute on the SMs.

The GPU architecture provides multiple hardware work queues, called *streams*. Independent kernels can be associated to different streams and can be execute concurrently. By default, kernels are inserted into the single default stream, unless specified otherwise. In addition to the hardware work queues, NVIDIA also introduced a framework called Multi-Process Service (MPS). MPS enables kernels from different applications or contexts to share the GPU resources.

Current GPU architectures provide hardware preemption at coarse and fine-grain granularities: thread level and instruction level. In thread level preemption, the threads that are executing on the SMs have to be completed before the preemption actually occurs. This type of preemption reduces the amount of context to be saved on the preemption event, since the threads finished their work. According to NVIDIA, the switching of kernels in thread boundary can complete in less than 100 μs [16]. In instruction level preemption, however, all thread processing stops at the current instruction and the threads contexts have to be saved. This type of preemption involves substantially more state information, because all the registers of the executing threads must be saved. Modern GPUs can have up to 2048 threads executing on a SM. Each thread has its own registers and shares the scratchpad memory. So, the size of the context for one SM can be large, each SM has a register file of up to 256KB and 96KB of shared memory. Saving such a large context takes significant

memory bandwidth.

In this scenario, reducing the overhead of context switching has been a hot topic of research in recent years. Tanasic *et al.* [17] proposed some hardware extensions to allow preemption in early GPU architectures and two preemption mechanisms, one that resembles CPU preemption and saves the context of all the thread blocks running on a SM, and another one, called SM-draining, that preempts the execution of one block at a time in the SM until the whole SM is drained. Park *et al.* [18] designed Chimera that introduces the SM flushing technique. The idea is to detect idempotent kernels, which generate exactly the same result even when restarted in the middle of the execution, and to drop an execution of a thread block without context saving. Lin *et al.* [19] used compiler intervention to identify appropriate preemption points in the code, and a compression mechanism to reduce kernel switch overhead. Li *et al.* [20] proposed a mechanism that proactively prepares the context switching, reducing the amount of work during the preemption event. NVIDIA proposed the instruction level preemption quite recently, but since preemption is important for preventing long-running kernels to monopolize the GPU, future generation GPU will probably enhance this hardware preemption mechanism.

Despite the high overhead of context switching, the ability to preempt an execution not only improves responsiveness, but also allows a more sophisticated scheduling mechanism to increase resource utilization, as multitasking operating systems do with the CPUs. Figure 3 illustrates how preemption can be used to provide a more fair access of the GPU resources and increase the throughput. Suppose another group of 4 kernels A, B, C and D, that arrive at the GPU in different timings. On Figure 3 (a), the kernel scheduling is performed by the leftover policy. On Figure 3 (b), a scheduling policy with preemption is employed. What we can observe is that the scheduling with preemption provides more responsiveness for kernels B, C and D, and improve the system throughput.

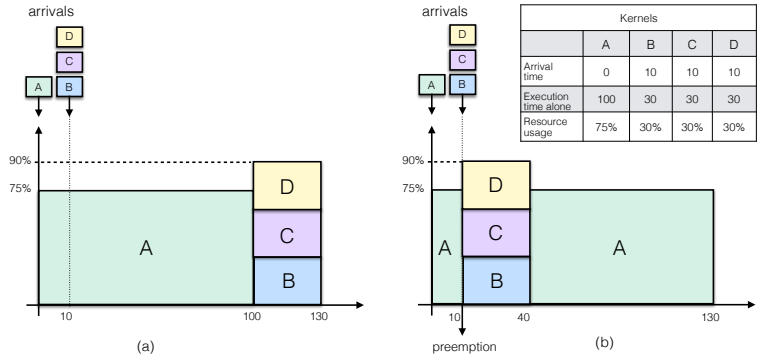


Figure 3: Example where the use of preemption can improve responsiveness and the system throughput

2.2. Related Work

2.2.1. Co-Scheduling

In [7], the authors consider the problem of scheduling independent moldable tasks on a multicore node. For each task and each number of cores, the processing time is assumed to be given, possibly using a model function. In general, due to intrinsic lack of parallelism in tasks, using all cores for a given task is usually not efficient, and it is better to schedule several tasks simultaneously. They consider that tasks will be processed 2 by 2 only, and that a given task can only be scheduled with another task. Then, under this assumption, the optimal solution can be found in polynomial time. It consists in building a graph where vertices are tasks and the edge between T_i and T_j denotes the minimal time to run T_i and T_j in parallel (to do this, you need to find the optimal number of cores to give to each task, and they have a simplified algorithm that always consist in achieving the optimal makespan for one of the tasks, what looks were arguable). Then, finding the optimal schedule consists in finding a minimal matching in this graph, what can be done in polynomial time.

This approach has been later extended by Aupy et al. in [8]. Instead of considering at most two parallel tasks running at the same instant on the multicore node, the authors propose to run a group of at most k tasks in parallel on the p cores (k-IN-p-COSCHEDULE optimization problem). Then, groups are run in sequence with no overlap, so that [7] corresponds to the case where $k = 2$. From the complexity point of view, it is proved that the problem becomes NP-Complete as soon as $k \geq 3$ and several approximation algorithms and heuristics are proposed to efficiently build the groups.

In both above works, groups of tasks that are allowed to run in parallel are defined and then the different groups are processed in sequence. This strongly differs with our GPU approach. We limit the level of parallelism to 2 tasks, as in [7], what better represents the capacities of GPUs. On the other hand, we want a long task to run in parallel to several short tasks, that are processed in sequence, since the strong heterogeneity in task durations makes group based solution not efficient in our context. A similar approach was advocated in [9], where a comparison between list and group based scheduling strategies is provided, where several approximation ratios are proposed for both classes of scheduling strategies. Nevertheless, this results do not apply directly to our context, since the restriction to list schedules is too strong. Indeed, with tasks that run very well or very bad in parallel, as it is the case in GPU scheduling but not in parallel tasks scheduling, introducing idle times is crucial to achieve good performance.

2.2.2. GPU Scheduling

Enabling efficient multiprogramming on GPUs is receiving a lot of attention from the research community in recent years. Software techniques, such as *reordering* [32, 33, 3] were proposed to find a good submission order to improve resource utilization. Other software techniques propose modifying the granularity of the kernels to create more concurrency opportunities. Zhong *et al.* [1] proposed *kernel slicing*, Ravi *et al.*

[34] proposed *molding* the dimensions of the grid and the thread blocks, and Pai *et al.* [2] improved this technique, proposing the *elastic kernel*, that elasticizes any kind of kernel. Hardware techniques were also proposed to divide the GPU resources among the concurrent kernels, called *spatial multitasking*. Adriens *et al.* [5] proposed some partition schemes to divide the GPU multiprocessors (SMs) among the applications. Liang *et al.* [35] proposed an heuristic that makes the spatial multitasking decisions based on the kernels behavior. Zhao *et al.* [6] proposed a dynamic spatial multitasking mechanism that classifies the workloads and searches for effective SM partition based on the workload characteristics.

The non-preemptive co-scheduling of kernels of different applications on the GPU have also been studied. The work of Margiolas and O'Boyle [21] presented accelOS, a modified OpenCL JIT compiler that analyzes the kernel behavior at the compilation time and transparently modifies the kernel code in terms of the thread blocks size in order to improve fairness in the assignment of the GPU resources among multiple kernels. Belviranli *et al.* [36] work focused on the data transfer overhead in the scheduling decisions. Kato *et al.* [22] proposed TimeGraph a GPU scheduler based on priorities to make high-priority tasks responsive on the GPU on real-time environments. Chen *et al.* [23] presented a runtime system called Baymax that comprises a task duration predictor and a task reordering mechanism based on the predictions to guarantee QoS. In the same direction, Prophet was proposed in [24] with a interference prediction model across different resources and a scheduling algorithm based on the predicted times to guarantee the QoS of latency-sensitive applications. Ukidave *et al.* [12] presented Mystic an interference-aware mechanism for co-scheduling on GPUs based on machine learning to predict whether kernels can share a GPU efficiently. The recent work of Wen *et al.* [25] proposed a graph-based algorithm to schedule kernels in pairs. Their approach models the co-execution of kernels as a graph, in which nodes represent kernels while an edge indicates that the co-execution of the two nodes can experience performance gain, and the edge weight can be labelled with the relative speedup of co-execution.

There are also some works in exploiting kernel preemption to provide fairness, responsiveness, and quality of service of applications running on GPUs. Xu *et al.* [27] proposed a deadline-aware dynamic GPU partitioning to improve the responsiveness when a GPU is shared by tasks with strict and non-strict deadlines. Wang *et al.* [26] proposed QoS mechanisms for fine-grained GPU sharing, where kernels with QoS requirements receive enough resources to reach its goals, and the remaining resources are assigned to kernels without QoS requirements. Wang *et al.* [28] proposed Simultaneous Multikernel (SMK) that allows fine-grain sharing within each SM, kernels with complimentary resource usage are co-scheduled in the same SM to achieve resource fairness and better utilization. Chen *et al.* [29] proposed a compiler-runtime software solution for priority-based preemptive scheduling on GPUs, that transforms the kernel for voluntary preemptions. Jin *et al.* [30] proposed a preemption-aware scheduler that use cache miss behavior to classify the kernels into compute-intensive and memory intensive, and predict the performance of complementary execution of pairs

of kernels based on their classification. They also proposed a preemption overhead model based on the context size and implement a greedy algorithm that schedule each time the pair of kernels with the best co-execution. Wu *et al.* [31] proposed FLEP, a compilation and a runtime system that transforms the input kernel to allow spatial preemption, where the running kernel only yields part of the SMs. FLEP comprises a kernel-specific performance model to predict the kernel durations and two scheduling algorithms, one to schedule with priorities and other to achieve fairness.

Table 1 shows a comparison of our proposed co-scheduling strategies with the previous preemptive and non-preemptive approaches. Compared to the other approaches, we propose both preemptive and non-preemptive algorithms that use optimization strategies to find a global optimal solution rather than local or greedy solutions. In addition, we are the first to propose an algorithm to reduce the number of preemptions in co-scheduling.

3. Model and Notations

In this paper, we consider the problem of scheduling a set $\mathcal{T}$of n independent tasks on a GPU with two streams. We assume that we know for each task i its execution time p_i, and for each pair of task (i, j), we know the speed $S_{i,j}$ at which task i makes progress when running at the same time as task j. For ease of notation, we will assume that $S_{i,i} = 1$.

Given this input data, we consider schedules σ defined as a succession of intervals I_l, with duration d_l, during which the set of running tasks is R_l, where $1 \leq |R_l| \leq 2$. If $i \in R_l$, its *companion* task in R_l is either j if $R_l = \{i, j\}$ or i if $R_l = \{i\}$, and is denoted $c(R_l, i)$. A schedule is *valid* if all tasks have progressed to completion, *i.e.* $\forall i \in \mathcal{T}, \sum_{l|i\in R_l} d_l S_{i,c(R_l,i)} = p_i$. The *makespan* of a schedule is its total duration, $\sum_l d_l$. We are interested in the following problems:

Problem 1 (MinMakespan)**.** Given input data p_i and $S_{i,j}$, find a valid schedule of minimal makespan.

Problem 2 (NonPreemptiveMinMakespan)**.** Given input data p_i and $S_{i,j}$, find a valid *non-preemptive* schedule of minimal makespan.

A schedule is *non-preemptive* if each task i runs without interruption, which means that $X_i = \{l|i \in R_l\}$ is an interval (a set of successive integers) for all i. In a preemptive schedule, we can define the number of preemptions of task i as $P(\sigma, i)$, the number of intervals required to partition X_i minus one. The total number of preemptions of a schedule σ is $P(\sigma) = \sum_{i\in\mathcal{T}} P(\sigma, i)$.

Remark.. If for a pair a task (i, j), we have $S_{i,j} + S_{j,i} \leq 1$, it is not worth it to schedule tasks i and j together. Indeed, it is never worse to replace a time interval of length d where they are together by a time of length $dS_{i,j}$ for task i alone followed by a time of length $dS_{j,i}$ for task j alone. Thus we can assume that $\forall i, j$, $S_{i,j} + S_{j,i}$ is either 0 or more than 1.

In the rest of the paper, we focus on these two problems successively: Problem MinMakespan in Section 4, then Problem NonPreemptiveMinMakespan in Section 5.

4. Preemptive Schedules

In this Section, we present results and algorithms to obtain preemptive schedules. We first show that the optimal preemptive makespan can be computed rather easily by solving a Linear Program in rational numbers. Then, we are interested in building a schedule which minimizes the number of preemptions. In Section 4.2 we present an algorithm which, from any optimal solution of the Linear Program, computes a schedule for this particular solution with the minimum number of preemptions. However, in Section 4.3, we show that finding an optimal solution of the Linear Program that will result in the smallest number of preemptions is a NP-hard problem.

4.1. Optimal Preemptive Schedule

The first remark from the previous definitions is that the makespan of a preemptive schedule σ does not depend on the ordering of its intervals, but only on their duration. It can thus be described with $\frac{n(n-1)}{2}$ variables x_e for $e \in \{\{i,j\} | i,j \in \mathcal{T}\}$ describing the duration of the interval during which the set of running tasks is e. For this reason, given a speed matrix S, we introduce the graph $G(S) = (\mathcal{T}, E(S))$ where the set of vertices is $\mathcal{T}$, and there is an edge between i and j if $S_{i,j} + S_{j,i} > 1$. Note that the graph $G(S)$ also contains loops (edges from i to i).

This description means that we can express the MinMakespan problem as the following Linear Program, called (PLP):

$$\begin{aligned}
\text{Minimize} \quad & \sum_{e \in E(S)} x_e \\
\text{subject to} \quad & \forall i \in \mathcal{T}, \quad \sum_{e \in E(S) | i \in e} x_e S_{i,c(e,i)} \geq p_i \\
& \forall c \in C, \quad x_c \geq 0
\end{aligned}$$

This Linear Program has $|E(S)|$ variables, and $|E(S)| + n$ constraints. In any optimal solution, at least $|E(S)|$ constraints are saturated, thus at most n inequalities are strict. This implies that at most n x_c variables are positive, and thus any optimal solution contains at most n different intervals. In the following, we denote by x^* an optimal solution of this Linear Program. This solution yields the optimal makespan for problem MinMakespan, and any ordering of the obtained intervals incurs at most n preemptions (the total number of task appearances is at most $2n$, and the first appearance of each task does not count as a preemption).

In the remainder of this Section, we present how to compute an ordering of these intervals to minimize the number of preemptions. To his end, we introduce the graph $G(x^*)$:

Definition 1 (Graph $G(x)$ associated with a solution x)**.** Given a solution x of the (PLP), we define $G(x)$ as the graph with one vertex per task, in which there is an edge between tasks i and j if and only if $x_{\{i,j\}} > 0$.

By definition, $G(x)$ is a spanning subgraph of $G(S)$. The above result ensures that $G(x^*)$ has at most n edges. But we can actually prove a stronger statement:

Proposition 1. *Any connected component of $G(x^*)$ with k vertices has at most k edges.*

Proof. If we rewrite (PLP) with only the variables x_e such that $x^*_e > 0$, we obtain exactly the same solution x^*. In this restricted (PLP), the connected components of $G(x^*)$ produce independent problems, and for each of them the above analysis on the number of constraints and variables is still correct. □

Definition 2. A connected graph with at most as many edges as vertices is called a *pseudo-tree*, and can have at most one cycle. A *pseudo-forest* is a graph in which each connected component is a *pseudo-tree*.

Theorem 1. *For any optimal solution x^*, $G(x^*)$ is a pseudo-forest.*

4.2. Minimizing the number of preemptions

In this Section, we describe how to minimize the number of preemptions of a preemptive solution. We start by characterizing the graphs $G(x^*)$ from which it is possible to build a non-preemptive schedule. As shown on Figure 4a, if the graph is a single path we can build a schedule without any preemption, just by following the path. Case 4b shows that this holds even if the graph is a caterpillar, *i.e.* has several nodes of degree 1 connected to a central path. In case 4b the path is $A - B - E - G - H$, with C, D and F (shown in lighter color) connected to it.

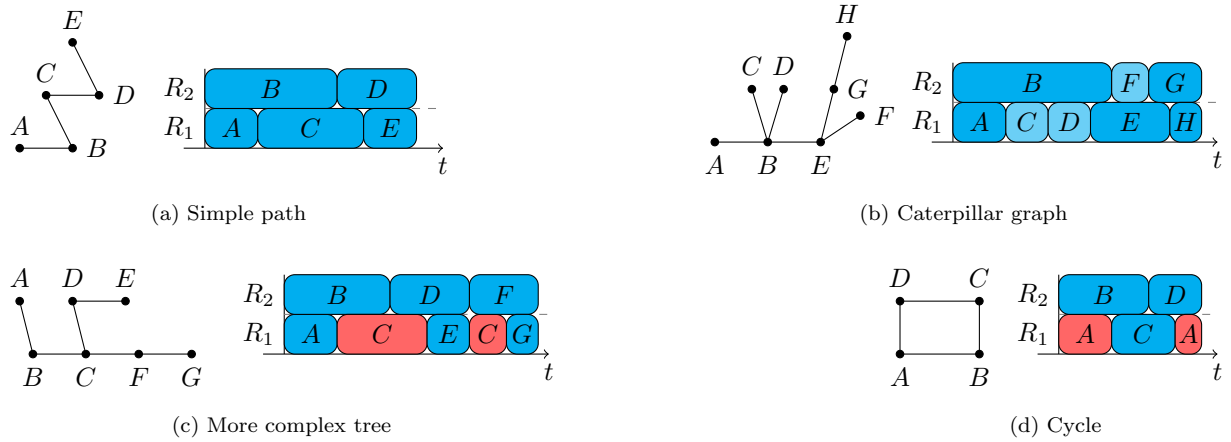
Case 4c shows that this no longer holds true when nodes connected to the path have a more complicated structure. Here the path is $A - B - C - F - G$, with $D - E$ connected to it, and it is necessary to have a preemption (here C is preempted). Case 4d shows that any cycle incurs at least one preemption.

This brings us to the following Lemma:

Lemma 1. *The graph G of a preemptive solution can be ordered in a non preemptive way if and only if it is a vertex-disjoint collection of combs.*

Definition 3 (Caterpillar)**.** A graph G is a *caterpillar* if it contains a path P, such that all nodes of G that do not belong to P are incident to only one edge, directly connected to a node of P.

We call the path P the *internal path*, and all nodes and edges of this path are called *internal*.



(a) Simple path

(b) Caterpillar graph

(c) More complex tree

(d) Cycle

Figure 4: Possible cases to turn the graph G into a schedule.

From the above discussion, it is natural to consider that preempting a task can be modeled by *splitting* the corresponding vertex of the graph, so that the task can appear at several places in the schedule. When a vertex is split, its neighbors are partitioned in two sets: the first set is attached to the first copy of the vertex, and the second set to the second copy.

The problem that we are interested in is:

Problem 3 (CaterpillarSplit)**.** Given a graph G, find a minimum number of node splitting operations to obtain a graph G' which is a collection of vertex-disjoint caterpillars.

Of course, this problem can be solved separately on each connected component of G. We start by analyzing problem CaterpillarSplit on trees, and will move to pseudo-trees later.

4.2.1. Solving the problem on trees

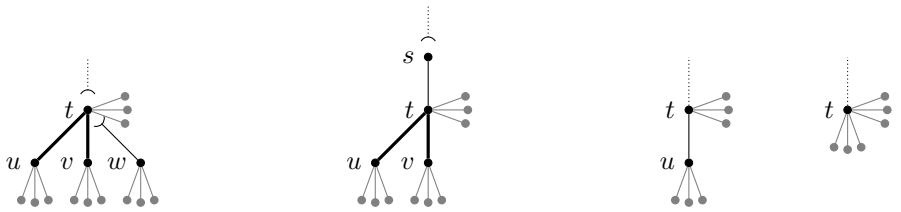
On trees, the problem CaterpillarSplit can be formulated this way:

Problem 4. Given a tree T, find a partition of the edges of T into a minimum number of edge-disjoint caterpillars.

Indeed, two different caterpillars in the result can share at most one node, and thus the number of caterpillars in a solution is exactly one plus the number of node splitting operations required.

To solve this problem, we make a number of observations in the following lemmas. We assume that the tree T is rooted at some arbitrary root r. We first define the *height* of a vertex in T.

Definition 4 (height of a vertex)**.** The *height* of a vertex v of a rooted tree T is the length of the longest downward path to a leaf from v.



(a) Lemma 2 (b) Lemma 3 (c) Lemma 4 (original tree T on the left, T' on the right)

Figure 5: Illustrations of Lemmas 2, 3 and 4. Leaves are in gray, internal paths of the caterpillars have thick edges, and non connected edges show where the splitting occurs.

By definition, leaves of T have height 0, and if the children of a vertex v are all leaves, then v has height 1. For any node v, we call *leaves* of v the children of v which are leaves in T.

Lemma 2 (Greedy for vertices of height 1)**.** *Assume that T contains a vertex t of height 2 connected to at least 3 vertices of height 1 u, v, w. There exists an optimal solution containing the caterpillar whose internal path is (u, t, v), with their leaves attached.*

Proof. We first show that there exists an optimal solution in which nodes u, v and w are all internal nodes of their respective caterpillars in which they are connected to t. Indeed, if any of them (say node u) is a leaf node in the caterpillar C_1 containing edge (u, t), then the solution necessarily contains another caterpillar C_2 with the edges connecting u to its leaves. We can thus disconnect edge (u, t) from caterpillar C_1 and add it to caterpillar C_2. Both C_1 and C_2 remain valid caterpillar, and both solutions contains the same number of caterpillars.

Let us consider such an optimal solution, in which u, v and w are internal nodes. Since no path connects all three of them, they can not be in the same caterpillars in this solution. Consider the caterpillar C_u which contains u, the caterpillar C_v which contains v, and the caterpillar C_w which contains w. If $C_u \neq C_v$, we can disconnect both caterpillars at node t and reconnect them pairwise, so that u and v belong to the same caterpillar, and the other parts are connected as well. If $C_u = C_v$ we do not need to do this step. Leaves attached at t can be freely attached to this (u, t, v) caterpillar. Any other node connected to t as a leaf of either C_u or C_v can be attached to C_w. The resulting solution has the same number of caterpillars, and contains the caterpillar (u, t, v) with their leaves attached, which proves our lemma. □

Lemma 3 (Special case for two vertices of height 1)**.** *Assume that T contains a vertex t of height 2 connected to exactly 2 vertices of height 1, u and v. Denote by s the father of t. There exists an optimal solution containing the caterpillar (u, t, v) with their leaves attached and with edge (t, s) also included.*

Proof. By definition, u and v each have at least one leaf, which we denote by u' and v'. Consider any optimal solution in which edges (u, u') and (v, v') are part of two different caterpillars C_u and C_v. One of these caterpillars contains edge (t, s) in its internal path, otherwise they can be merged together and the solution is not optimal. Assume without loss of generality that C_u contains (t, s). It is possible to split C_u at node s, and merge C_v with the resulting part. This yields a solution with the same number of caterpillars as an optimal solution, thus it is optimal.

Consider now any optimal solution which contains the caterpillar (u, t, v) without edge (t, s). We can assume that all leaves of t are connected to this (u, t, v) caterpillar. Denote C the caterpillar of this solution which contains (t, s). By assumption, the only edges connected to t which do not lead to leaves are (t, u), (t, v) and (t, s). This implies that edge (t, s) is an external edge of caterpillar C. It is thus possible to remove it from caterpillar C and attach it to caterpillar (u, t, v): the solution produced is still optimal. □

Lemma 4 (Contract lonely vertices of height 1)**.** *Assume T contains a vertex t of height 2 which has exactly child of height 1, u. Then in any optimal solution, node u belongs to only one caterpillar.*

This implies that solving CaterpillarSplit *on T is equivalent to solving* CaterpillarSplit *on T' where edge (t, u) is contracted (i.e., nodes t and u are merged).*

Proof. Consider any solution in which node u belongs to several caterpillars, and denote by C the caterpillar which contains edge (t, u). Since t has only one child of height 1, the internal path of this caterpillar is either empty or ends with the edge connecting t to its parent. In both cases, it is possible to add edge (t, u) to this path. All leaves of u can thus be added to caterpillar C, decreasing the total number of caterpillars of the solution, which implies that the considered solution is not optimal. □

These lemmas suggest the following algorithm to solve CaterpillarSplit on trees:

The optimality of this algorithm comes directly from the previous Lemmas: for each caterpillar added to S, we have a proof that there exists an optimal solution containing this caterpillar. Since a tree T that does not contain any vertex of height 2 is a caterpillar, the produced solution is valid and optimal.

4.2.2. Solving the problem on pseudo-trees

We now consider the general problem CaterpillarSplit on pseudo-trees. Consider a pseudo-tree T which is not a tree: it either contains a self loop (an edge (i, i)) or one proper cycle C. If T contains a self loop (i, i), then an equivalent problem is obtained by replacing this loop with an edge connecting i to a new node i' which represents the time during which task i must execute by itself. This replacement yields a tree, on which we can apply Algorithm 1.

If T contains a proper cycle C, it is easy to see with a similar argument as above that the number of caterpillars of any solution is equal to the number of split operations. We are thus again interested in minimizing the number of caterpillars used to partition the edges of T.

Algorithm 1: CaterpillarSplit(T, r)

Input: A tree T, rooted at r

Output: A partition of T in caterpillars

```
S ← ∅ ;
while T has vertices of height 2 do
    t ← a vertex of height 2 of T;
    if t has one child of height 1 u then
        Merge t with u ;
    else if t has two children of height 1 u and v then
        C ← caterpillar (u, t, v) with leaves attached, and the edge from t to its parent;
        S ← S ∪ {C};
        Remove edges of C from T;
    else if t has at least 3 children of height 1 then
        Pick u and v, two children of height 1 of t;
        C ← the caterpillar (u, t, v) with leaves attached;
        S ← S ∪ {C};
        Remove edges of C from T;
end
return S ∪ {T}
```

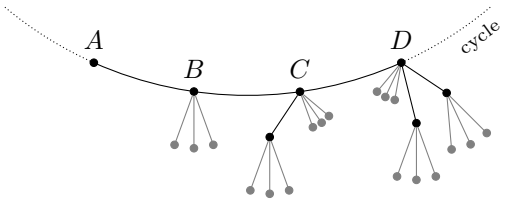
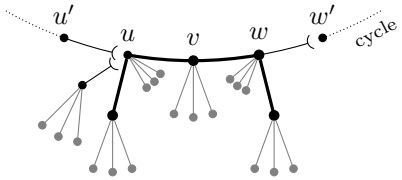
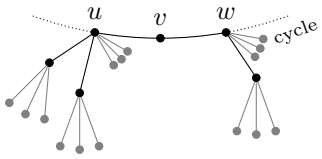
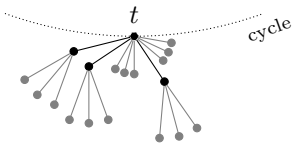


Figure 6: Four possible shapes for a node on the cycle.



(a) Lemma 5, where u has two legs and w has one. Thick edges show the internal path of the caterpillar, and non-connected edges show where the splitting occurs.





(b) Lemma 7. The original pseudo-tree T is on the left, T' is on the right.

Figure 7: Illustrations for Lemmas 5 and 7. Gray nodes are leaves.

For each $u \in C$, the subgraph of nodes reachable from u without going through C is a tree T_u. By rooting T_u at u, all the Lemmas proven above are still valid, with one exception: Lemma 3 cannot be applied if the vertex of height 2 t is u, since u has no father in T_u. This implies that iteratively applying the Lemmas as in Algorithm 1 yields a cycle C in which the subtrees T_u can have 4 possible shapes (see Figure 6):

A. $T_u = \{u\}$, *i.e.* u is a leaf in T_u;

B. u has height 1 in T_u, with at least one leaf;

C. u has height 2 in T_u, and exactly one child of height 1;

D. u has height 2 in T_u, and exactly two children of height 1.

If T_u is of shape C or D, we call the subtree corresponding to each vertex of height 1 in T_u a *leg* of u. Each $u \in C$ can thus have 0, 1 or 2 legs. Note that for shapes C and D, node u can have any number of leaves in T_u.

If two legs are separated on the cycle by at least two nodes, or by one node of shape B, we can reduce the problem with the following lemmas:

Lemma 5. *Assume the cycle C contains three consecutive nodes u, v, w, where u and w have at least one leg, and node v has shape B. Then there exists an optimal solution which contains the caterpillar made of*

one leg of u, T_v and one leg of w, with leaves of u and w attached. If u and/or w has only one leg, then this caterpillar also contains the next edge of the cycle in the corresponding direction, as an external edge.

Proof. We first prove that there exists an optimal solution in which edges (u, v) and (v, w) are part of the same caterpillar. Take any optimal solution S in which these edges are part of two different caterpillars C_u and C_w. Apart from these two edges, node v is only connected to leaves. Hence, if any of these edges is an internal edge of its caterpillar, it is necessarily the last edge of the internal path. Consider the three following cases:

- Both (u, v) and (v, w) are internal edges of their respective caterpillars. Both internal paths of these caterpillars finish at node v, we can thus fuse them together and obtain a better solution, in contradiction with the optimality of S.
- Only one edge, (u, v) for example, is an internal edge. It is possible to cut the internal path of C_w at node w, extend one of the parts with edge (v, w) and fuse it with the internal path of C_u, which yields a solution with the same number of caterpillars.
- Both edges (u, v) and (v, w) are external edges. This implies that S contains at least one caterpillar C_v which contains the edges from v to its leaves. Consider the solution S' obtained by cutting C_u at node u, cutting C_w at node w, and connecting one part of each caterpillar through node v. S' has the same number of caterpillars as S.

Let us now consider any optimal solution in which edges (u, v) and (v, w) are part of the same caterpillar C_v. Just like in the proof of Lemma 2, the height 1 nodes of the legs of u and w are internal nodes of their respective caterpillars C_u and C_w. If $C_u \neq C_v$, then it is possible to replace the end of the internal path of C_u which contains the leg of u with the end of the internal path of C_v which continues after edge (v, u). The same can be done with C_w.

Thus there exists an optimal solution in which one leg of u, T_v and one leg of v are in the same caterpillar C_v. All leaves of u and w can be attached to C_v. If the edge from u to the next node u' on the cycle belongs to C_v, it is an external edge. If u has another leg, we can attach this edge to the caterpillar containing this other leg. Conversely, if u has no other leg and the edge (u, u') is not part of C_v, it is also an external edge because v has no other edge where the path could continue. It can thus be attached to C_v. We can do the same analysis with node w for the edge leading to the next node on the cycle.

This concludes our proof. □

Lemma 6. *Assume the cycle C contains at least four consecutive nodes $(u, v_1, \ldots, v_k, w)$ for $k \geq 2$ where u and w have at least one leg, and nodes v_i have no leg. Then there exists an optimal solution containing the caterpillar made of one leg of u, $\cup_i T_{v_i}$, and one leg of w, with the leaves of u and w attached. If u and/or*

w has only one leg, then this caterpillar also contains the next cycle edge in the corresponding direction, as an external edge.

Proof. This Lemma is proven in the same way as the previous one, by stating that there exists an optimal solution in which edges of the path $(u, v_1, \ldots, v_k, w)$ are part of the same caterpillar. □

Node v in Lemma 5 and nodes $v_1, \ldots, v_k$ in Lemma 6 are called *forcing* nodes.

When two legs are close enough, however, the next Lemma shows that we can merge the corresponding nodes:

Lemma 7. *Assume that C contains three consecutive nodes (u, v, w) where u and w have at least one leg, and T_v has shape A (i.e., v is connected to no other edge). Consider T' obtained by contracting the edges (u, v) and (v, w): the three nodes are replaced by a node t, to which T_u and T_w are connected.*

Then solving CaterpillarSplit *on T is equivalent to solving* CaterpillarSplit *on T'.*

Proof. Consider any solution S' for T'. If one leg of u and one leg of w are part of the same caterpillar in S', the edges (u, v) and (v, w) can be added to this caterpillar in place of node t, and we obtain a solution for T. On the other hand, if there is no caterpillar with a leg of u and a leg of w, it means S' contains at least two caterpillars with node t. We can attach edge (u, v) to the caterpillar which contains a leg of u, and (v, w) to the caterpillar which contains a leg of w (both as external edges). Since v has no other edge, this yields a solution for T.

If two caterpillars of S' contain a leg of u and a leg of w, we can swap them to obtain a solution where one caterpillar has two legs of u and one caterpillar has two legs of w, and return to the second case mentioned above. □

Lemma 7 also holds if u and w are neighbors in the cycle: edge (u, w) can be contracted.

With these three lemmas, we can write algorithm 2 to solve CaterpillarSplit on pseudo-trees. On line 13, once C_v has been removed from T, the resulting graph is a tree. We can thus use Algorithm 1 from the previous Section to solve CaterpillarSplit in that case. On line 16, all legs of the cycle will be grouped by two, and we obtain $\lceil \frac{l}{2} \rceil$ caterpillars, where l is the number of legs.

4.3. Simultaneously minimizing makespan and preemptions

The algorithms presented above work in two steps (see the overview of results in Figure 2): we first compute a preemptive solution x^* with optimal makespan thanks to the Linear Program (PLP), and then we compute the optimal ordering for **this particular solution** which minimizes the number of preemptions. This does not guarantee that the resulting schedule has an optimal number of preemptions: there may exist another optimal solution of (PLP) which incurs fewer preemptions with a correct ordering.

Algorithm 2: CaterpillarSplit(T, C)

Input: A pseudo-tree T, containing one cycle C

Output: A partition of T in caterpillars

foreach $u \in C$ **do**
  Solve CaterpillarSplit on T_u, stop when u has at most two children of height 1;
end
if *no node of* C *has a leg* **then**
  Split any node of C to obtain a path;
  return the caterpillar made of this path and all their leaves attached;
else if *exactly one node* u *of* C *has one or two legs* **then**
  Split node u to obtain a path;
  return the caterpillar made of this path and all their leaves attached;
else if *there are at least two legs on cycle* C **then**
  if *there exists two legs separated by a forcing node* v **then**
    $C_v \leftarrow$ the caterpillar forced by v (Lemma 5 or 6);
    $S \leftarrow$ the solution of CaterpillarSplit on $T \setminus C_v$;
    return $S \cup \{C_v\}$;
  else
    Repeatedly merge two consecutive nodes on the cycle (Lemma 7) and apply Lemma 2;
    return the resulting caterpillars
  end

In order to strengthen the guarantee on the number of preemptions, one may want to study the following problem (we denote it with MinPreemptions): given a speed matrix $S_{i,j}$ and processing times p_i, output a schedule with the smallest number of preemptions among those schedules with optimal makespan. Unfortunately, we show in this Section that this problem is actually NP-complete. Furthermore, this same proof shows that the non preemptive version NonPreemptiveMinMakespan is also NP-complete.

Theorem 2. *The* MinPreemptions *and* NonPreemptiveMinMakespan *problems are NP-complete.*

Proof. Both problems clearly belong to NP: given a solution, it is easy to check in polynomial time that the solution is valid, and to compute its makespan and number of preemptions.

We prove their NP-hardness by reduction from the well-known problem 3-Partition, whose input is a set of $3n$ integers a_i, with $\sum_i a_i = nB$, and whose output is a partition of $[|1, 3n|]$ into n parts P_j such that $\forall j, \sum_{i \in P_j} a_i = B$.

From any 3-Partition instance with values a_i, we build the following instance for our scheduling problems:

- There are n groups of tasks A_j, B_j, C_j, X_j, Y_j with the following processing times:

	A	B	C	X	Y
p	$2B$	$3B$	$2B$	B	B

  The co-scheduling matrix S for the tasks of group j is as follows:

	A	B	C	X	Y
A	$\cdot$	1	0	1	0
B	1	$\cdot$	1	0	0
C	0	1	$\cdot$	0	1
X	1	0	0	$\cdot$	0
Y	0	0	1	0	$\cdot$

  For any tasks u and v in different groups, $S_{u,v} = 0$.

- In addition, there are $3n$ V_i tasks, with $p_{V_i} = a_i$ from the 3-Partition instance. The co-scheduling coefficients of these tasks are 1 for all the B_j tasks ($S_{V_i,B_j} = 1$ for all i and j), and 0 for all other tasks of any group j.

If the 3-Partition instance is positive, it is possible to schedule these tasks without preemption with makespan $5nB$ (see Figure 8): each group j is associated with one part P_j, with the following schedule: tasks A_j and X_j for B time, then A_j and B_j for B time, then B_j and the V_i tasks for $i \in P_j$ for B time, then B_j with C_j for B time, then C_j with Y_j for B time.

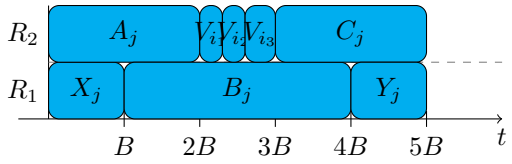


Figure 8: Valid schedule for group j with $P_j = \{i_1, i_2, i_3\}$ if the 3-PARTITION instance is positive.

Conversely, assume there exists a non preemptive schedule for these tasks in time $5nB$. Since the total execution time of all tasks is $9nB$ from tasks of all groups and nB from V_i tasks, this schedule must have two tasks executing at all times. In each group j, task X_j can only run together with task A_j for a duration B. The remaining duration of task A_j can only be together with task B_j. Similarly, task Y_j must execute with task C_j, and the remaining duration has to be with task B_j. The remaining duration of task B_j (for B time units) has to be together with some tasks V_i for some i. Let us denote by P_j the set of indices i such that task B_j runs with the tasks V_i in this schedule. Since no preemption is allowed, the sum of execution times of these V_i tasks is exactly B, and so we have a solution for the 3-PARTITION instance. □

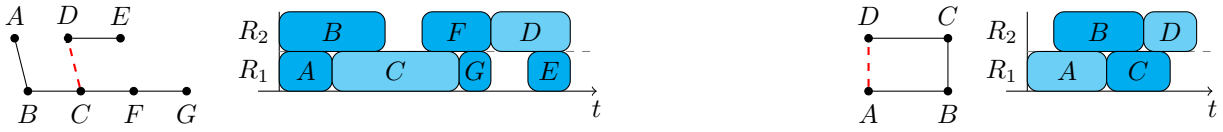
5. Non Preemptive Schedules

In this Section we present results about building non-preemptive schedules. Our methodology remains the same: solve the preemptive Linear Program described in Section 4.1, and modify the obtained solution to get a valid solution with the smallest cost. In Section 4.2, the cost was the number of preemptions; we are now interested in minimizing the makespan of the solution (since we will have to introduce idle times), under the constraint that it has zero preemptions.

Then, in Sections 5.3 and 5.4, we study the complete problem NONPREEMPTIVEMINMAKESPAN (*i.e.*, which does not take as input a preemptive solution from the Linear Program). We first give approximation results for our proposed algorithm, then present a Mixed Integer Linear Program to compute a more precise solution.

5.1. Problem definitions

We remember from the previous Section that a non preemptive schedule of a graph G is possible if it is a collection of vertex-disjoint caterpillars. However, instead of splitting nodes, we will now be allowed to *break* edges. In graph $G(x^*)$, the edge $\{i, j\}$ means that tasks i and j should run together during a time $x^*_{\{i,j\}}$. Breaking this edge means that we decide that these tasks no longer run together, but instead that each task runs alone (and this can happen at very different times in the schedule), thus introducing idle time on the other resource. Figure 9 shows how this can be done on the two examples of Figure 4 which contain



(a) More complex tree, breaking edge (CD)

(b) Cycle, breaking edge (AD)

Figure 9: Turning the graph G into a schedule, without preemptions.

preemptions. This replaces an interval of length $x^*_{\{i,j\}}$ by two intervals of respective lengths $x^*_{\{i,j\}}S_{i,j}$ and $x^*_{\{i,j\}}S_{j,i}$. The additional makespan cost of breaking edge $\{i,j\}$ is thus $x^*_{\{i,j\}}(S_{i,j}+S_{j,i}-1)$.

We are thus interested in solving the following problem:

Problem 5 (MinMakespanEdgeBreaking)**.** Given a solution x^* and its associated graph $G(x^*)$, find a non preemptive schedule of minimum makespan by breaking edges from $G(x^*)$.

Equivalently, find the minimum weight edge set B so that breaking all edges of B from G yields a collection of caterpillars, where the weight of edge (i,j) is $w_{\{i,j\}} = x^*_{\{i,j\}}(S_{i,j}+S_{j,i}-1)$. The weight of B represents the increase of makespan compared to the preemptive solution.

Unlike in the previous Section, this edge breaking operation has the following property: breaking an edge (u,v) in a graph G where u has degree at least 2 can not result in u having degree 1 afterwards. Indeed, breaking this edge results in two self-loops (u,u) and (v,v), and thus node u has to be an internal node of the caterpillar it belongs to. Furthermore, breaking an edge (u,v) if u is has degree 1 is not useful: either v has also degree 1 and $\{u,v\}$ is a caterpillar, or v has degree at least 2 and it is an internal node of its caterpillar in any solution, thus u can be part of the caterpillar of v without breaking the edge.

For any graph G, we denote as $\widehat{G}$ the graph obtained by removing all nodes of degree 1 from G. A graph G is a caterpillar if and only if $\widehat{G}$ is a path. The above properties implies that for any graph G, the two following statements are equivalent:

- Breaking edges of B from G yields a collection of caterpillars.
- Breaking edges of B from $\widehat{G}$ yields a collection of paths.

We are thus interested in solving the following problem:

Problem 6 (MinCostEdgeBreaking)**.** Given a weighted graph G with edge weights w_e, find the minimum weight edge set B so that removing all edges of B from G yields a collection of paths.

This MinCostEdgeBreaking problem is in turn equivalent to the PathCover problem:

Problem 7 (PathCover)**.** Given a weighted graph G with edge weights w_e, find a maximum weight covering of the vertices of G by vertex-disjoint paths.

5.2. *Solving the* PathCover *problem*

A previous study on PathCover has proven the following results [37]:

- It is a NP-hard problem on general graphs. Indeed, if all weights are one, searching for a covering of weight at least $|V| - 1$ is equivalement to the Hamiltonian path problem.
- A greedy algorithm (called *Algorithm A*) that iteratively adds the valid edge of maximum weight to the current solution is a $\frac{1}{2}$-approximation.
- It is possible to obtain a $\frac{2}{3}$-approximation by first computing a maximum weight subgraph of G of maximum degree 2, and then removing the minimum weight edge of each cycle in the subgraph. This algorithm is called *Algorithm B* in [37].

Obviously, Algorithm B when applied to trees is optimal, since the second phase of the algorithm is not needed. On trees, the first phase can be solved in polynomial time with the following linear program [38], where the variable p_e indicates that edge e is part of the solution:

$$\begin{aligned} \text{Maximize} \quad & \sum_{e \in E} w_e p_e \\ \text{subject to} \quad & \forall i \in \mathcal{T}, \quad \sum_{e \in E | i \in e} p_e \leq 2 \\ & \forall e \in E, \quad 0 \leq p_e \leq 1 \end{aligned}$$

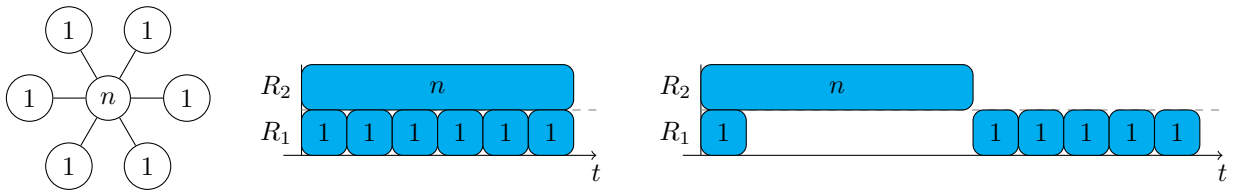
Another algorithm for the maximum weight degree constrained subgraph problem, based on matching techniques, has also been proposed by Shiloach [39].

If graph G is a pseudo-tree with one cycle C, any solution for G has to break at least one edge of C. A simple algorithm to solve PathCover on G is thus the following: for each edge e of C, break edge e to obtain a tree and solve PathCover on this tree. The best solution obtained over all edges e is an optimal solution for G.

5.3. *Approximation Analysis for* NonPreemptiveMinMakespan

We now consider approximation results for the complete NonPreemptiveMinMakespan problem, whose input is the scheduling matrix and not the graph. We have shown in Section 4.3 that this problem is NP-hard. We actually give a stronger result in the general case (where $S_{i,j}$ may be greater than 1, which is rather unrealistic): it is not possible to obtain an approximation algorithm in this case.

Proposition 2. *Problem* NonPreemptiveMinMakespan *in the general case does not have an constant approximation algorithm unless* $P = NP$.



(a) Graph $G(S)$, labeled with p_i

(b) Optimal schedule

(c) MaxPair schedule

Figure 10: Worst-case example for MaxPair.

Proof. Assume that some algorithm A has approximation ratio α. Consider the instance from the NP-hardness proof (Theorem 2), in which all $S_{i,j}$ values are multiplied by a factor $k = 10nB\alpha$. Remember that a solution for this instance exists in which no task is executed alone, with makespan $5nB$ in the original instance. Its makespan in the modified instance is thus $\frac{1}{2\alpha}$.

The approximation ratio of Algorithm A ensures that the solution it returns has makespan at most $\frac{1}{2}$. Since all tasks have processing times at most 1, it implies that in this solution, no task is executed alone. Algorithm A is thus actually optimal for NonPreemptiveMinMakespan, which implies that $P = NP$. □

For this reason, we focus on the *no speedup* case for the rest of the Section. This case is significantly easier to approximate: since we can only execute two tasks at a time, any solution has makespan at least $\frac{\sum_i p_i}{2}$. This implies that even the trivial Sequential algorithm (which executes all tasks sequentially) has approximation ratio 2.

We consider here two different algorithms. The first one is MaxPair from [25], described in Algorithm 3, which produces solutions in which each task is executed with at most one other task, and whose main ingredient is to compute a maximal matching in a weighted graph. The second one is our NonPreemptiveAlg algorithm, described in Algorithm 4, which computes an optimal solution to the (PLP) and solves the corresponding PathCover problem to optimality, as discussed in Section 5.2.

Unfortunately, the next results are negative results: both MaxPair and NonPreemptiveAlg have approximation ratio exactly 2, and thus provide no worst-case improvement over the trivial Sequential algorithm.

Proposition 3. *Algorithm* MaxPair *has approximation ratio at least* 2.

Proof. Let us consider an instance (depicted on Figure 10) where the graph $G(S)$ of matrix S is a star graph with n leaves (this means $S_{i,j} = 1$ for $i = j$ or $i = 1$ or $j = 1$, and 0 otherwise), and the processing times are given by $p_1 = n$ and $p_i = 1$ for $i > 1$. On this instance, there exists an optimal non preemptive schedule of

Algorithm 3: MaxPair (n, $S_{i,j}$, p_i)

Input: n tasks, a speed matrix $S_{i,j}$, processing times p_i

Output: A non-preemptive schedule for all n tasks

foreach *pair* (i, j) *of tasks with* $S_{i,j} + S_{j,i} - 1$ **do**
  $w_{i,j} = p_i + p_j -$ time if i and j are scheduled together;
end
Build graph G with n vertices and edge weights $w_{i,j}$;
$M \leftarrow$ maximum weight matching of G;
foreach *edge* $e = (i, j) \in M$ **do**
  Add interval l with $d_l = \min(\frac{p_i}{S_{i,j}}, \frac{p_j}{S_{j,i}})$ and $R_l = \{i, j\}$;
  Add an interval with i or j running alone for its remaining duration;
end
foreach *task* i *not incident to* M **do**
  Add interval l with duration $d_l = p_i$ and $R_l = \{i\}$;
end
return the produced schedule;

length n (all tasks run together with task 1), whereas the MaxPair algorithm returns a schedule of length $2n - 1$: task 1 is matched with one other task, and all other tasks run alone. □

Proposition 4. *Any algorithm based on breaking edges from a solution of (PLP), like Algorithm* NonPreemptiveAlg *for example, has approximation ratio at least* 2.

Proof. We consider the example given on Figure 11. The matrix S is a $\{0, 1\}$ matrix defined for an even number n and a given $\epsilon > 0$, and we describe its graph $G(S)$, where an edge between two nodes indicates that the corresponding tasks have $S_{i,j} = S_{j,i} = 1$, and no edge indicates that $S_{i,j} = S_{j,i} = 0$. The graph contains two wheels of size n (a cycle in which all nodes are connected to a common central node) whose centers are connected, and each node on the border of the wheels is connected to an additional exterior node of degree one. In total, this graph contains $4n + 2$ nodes (n regular nodes and n exterior nodes on each wheel, plus the two center nodes). Center nodes correspond to tasks of duration n, cycle nodes have duration $1 + \epsilon$, exterior nodes have duration ϵ. The preemptive Linear Program corresponding to this instance has at least two optimal solutions (depicted at the bottom of Figure 11). Both solutions include the edges between each cycle node and its corresponding exterior node. But in the first solution, the edge between the two center nodes is selected, as well as half the edges on both cycles. In the other solution, only the "star" edges between the center node and the cycle nodes on each wheel are selected. Both solutions have the

Algorithm 4: NonPreemptiveAlg $(n, S_{i,j}, p_i)$

Input: n tasks, a speed matrix $S_{i,j}$, processing times p_i

Output: A non-preemptive schedule for all n tasks

Solve the Linear Program (PLP) and obtain solution x^*, with graph $G(x^*)$;
Build $\widehat{G}(x^*)$, keep removed edges in R;
Assign weights $w_{i,j} = x^*_{\{i,j\}}(S_{i,j} + S_{j,i} - 1)$;
$\mathcal{P} \leftarrow$ the set of paths solution of PathCover for $\widehat{G}(x^*)$;
$B \leftarrow$ the set of broken edges in this solution;
Start an empty schedule;
foreach *path* P *of* $\mathcal{P}$ **do**
    foreach *node* u *along the path* P **do**
        Add interval l with $d_l = x^*_{u,v}$ and $R_l = \{u, v\}$ for each edge $(u, v) \in R$;
        Add interval l with $d_l = x^*_{u,v} S_{u,v}$ and $R_l = \{u\}$ for each edge $(u, v) \in B$;
        Add interval l with $d_l = x^*_{u,u'}$ and $R_l = \{u, u'\}$ where u' is the next node on P;
    end
end
return the produced schedule;

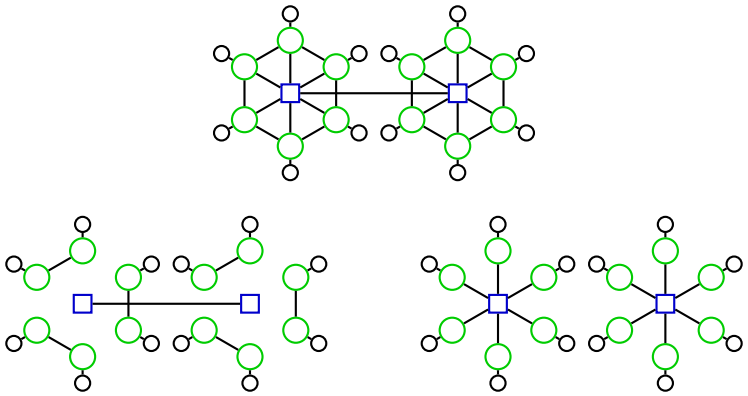


Figure 11: Worst case example for the Linear Program and Algorithm B when applied to the whole problem, for $n = 6$. Top: the graph $G(S)$. Bottom left: an optimal solution of the LP which is already non preemptive. Bottom right: another optimal solution of the LP which requires many preemptions.

same preemptive makespan ($2n + 2n\epsilon$), and we can obtain a non preemptive schedule directly from the first solution. However, any non preemptive schedule obtained from the second solution will have to break all edges from the star but 2 (in a similar way as for the MaxPair counter-example above), inducing a schedule of length $2(n+(n-2))+2n\epsilon = 4(n-1)+2n\epsilon$. This shows that we can not prove an approximation guarantee smaller than 2 for any algorithm based on breaking edges from a solution of the (PLP). □

5.4. Mixed Integer Programming approach for NonPreemptiveMinMakespan

We now try to directly solve the non-preemptive problem, without searching first for a good preemptive solution. Our chosen approach is to extend the LP with additional constraints to ensure that the resulting schedule is non preemptive. This requires to add integer variables to the formulation, and thus obtain a non polynomial algorithm. However, since ILP solvers are nowadays very efficient, this can result in a reasonable algorithm for many cases. In addition, this may provide good comparison points for our polynomial heuristics.

Let us again consider the graph where each task is a vertex, and there is an edge between two vertices if the corresponding tasks run together at some point in the schedule. From our previous analysis, we know that a non preemptive schedule corresponds to a graph which can be decomposed in disjoint paths, after all vertices of degree one have been removed. We will view this in the following way: if node i has degree one in $G(x^*)$, we say that it *pays* for the edge that connects it to its neighbor. This edge will be *free* for the neighbor, in the sense that it does not count towards its degree limit. In order to constrain the graph resulting from the LP to have this property, we introduce additional integer variables:

- y_e for $e \in E(S)$ is a $\{0,1\}$-variable indicating whether e is an edge in the graph $\widehat{G}(x^*)$,
- $f_{(i,j)}$ for $i \neq j$ where $\{i,j\} \in E(S)$ is a $\{0,1\}$-variable indicating that i is a degree one vertex in $G(x^*)$, and that it pays for the edge $\{i,j\}$. Note that both $f_{(i,j)}$ and the symmetric variable $f_{(j,i)}$ are present in the formulation. We note $A(S)$ the set of arcs (i,j) such that $f_{(i,j)}$ is a valid variable.

The complete mixed linear program is the following ($E'(S) = E(S)\backslash\{(i,i) \,|\, i \in \mathcal{T}\}$ denotes the set of configurations which execute exactly two tasks at a time):

$$\text{Minimize} \quad \sum_{e \in E(S)} x_e$$

$$\text{subject to} \quad \forall i \in \mathcal{T}, \quad \sum_{e \in E(S) | i \in e} x_e S_{i,c(e,i)} \geq p_i$$

$$\forall i \in \mathcal{T}, \quad \sum_{e \in E'(S) | i \in e} y_e + 2 \sum_{(i,j) \in A(S)} f_{(i,j)} \leq 2 \tag{1}$$

$$\forall i \in \mathcal{T}, \quad y_{(i,i)} + \sum_{(i,j) \in A(S)} f_{(i,j)} \leq 1 \tag{2}$$

$$\forall i \in \mathcal{T}, \quad \sum_{(i,j) \in A(S)} d(i) \cdot f_{(i,j)} + f_{(j,i)} \leq d(i) \tag{3}$$

$$\forall \{i,j\} \in E(S), \quad x_{\{i,j\}} \leq (y_{\{i,j\}} + f_{(i,j)} + f_{(j,i)}) \cdot \max(\frac{p_i}{S_{i,j}}, \frac{p_j}{S_{j,i}}) \tag{4}$$

$$\forall e \in E(S), \quad x_e \geq 0, \quad y_e \in \{0,1\}$$

$$\forall (i,j) \in A(S), \quad f_{(i,j)} \in \{0,1\}$$

Constraint (1) ensures that each node has degree at most 2 in the graph G' defined by the y_e variables, and additionally makes sure that if node i pays for a free arc, then it should not have any other incident edge in G'. Constraint (2) enforces this condition also on the (i,i) loop, which does not count for the degree 2 constraint, but node i paying for an edge prevents the loop (i,i) to be present in the graph $G(x^*)$. Constraint (3) ensures that if node i pays for a free arc, no other free arc incident to i can be paid by one of its neighbors. Finally, constraint (4) links the variables x to the integer variables to make sure that an edge is only used if at least one of the corresponding integer variables is equal to one.

Since any schedule can be seen as a collection of paths with additional degree-one vertices, it is clear that any schedule corresponds to a feasible integer solution of this linear program. However, some solutions can not be directly converted to a schedule. Indeed, the constraints only express that the graph G' has maximum degree 2, but there is no constraint on the absence of cycles, so in general the obtained solution is a collection of cycles with additional degree-one vertices.

Nevertheless, in a similar fashion as for the Algorithm B in [37], we can build feasible schedules by solving this problem and cutting the edge of smallest cost in each cycle of G'. We call the resulting algorithm DirectAlg. Note that the optimal solution of the linear program is also a lower bound on the makespan of a non preemptive solution.

6. Experimental evaluation

In this section, we present an experimental validation on realistic data, in order to assess the practical performance of the algorithms discussed above.

6.1. Benchmarks

The data was obtained by benchmarking a number of GPU kernels from different well-known GPU benchmark suites (Rodinia, Parboil and SHOC) that reproduce real-life applications [11]. The experiments were performed on a GPU Tesla P100-SXM2 based on the Pascal architecture. The kernels were compiled with NVCC CUDA version 8.0 and the executions used NVIDIA driver version 384.145, with a framework that allows the submission of two given kernels at the same time. Measuring the resulting execution time of both kernels allows to obtain realistic values for the slowdown experienced by kernels when running together. In some cases, submitting the kernels at the same time does not result in simultaneous execution, and we assume that this comes from an impossibility for both kernels to run at the same time.

Consider a given measurement for kernels K and K', whose standalone execution times are respectively T_K^1 and $T_{K'}^1$. We measure the *co-execution* time $O_{K,K'}$: the time interval during which both kernels are running together. We also measure the execution times of kernel K when running with K': $T_K^{K'}$, and respectively $T_{K'}^K$ is the execution time of K' when running with K.

With similar ideas to those used in [11] to define the so-called *concurrency index*, we can use these measurements to compute the speed $S_{K,K'}$ of kernel K when run with kernel K'. If the co-execution interval is the whole duration of $T_K^{K'}$, then $S_{K,K'} = \frac{T_K^1}{T_K^{K'}}$ since the total work is T_K^1, and it was performed in time $T_K^{K'}$. For shorter co-execution intervals, we will assume that outside of the co-execution interval, both kernels behaved as if they were in isolation. We can thus consider that all of the slowdown incurred by each kernel is due to this interval. Then the duration $T_K^{K'} - O_{K,K'}$ is common between both executions, and can be removed. Since the remaining duration was executed in time $O_{K,K'}$, the speed $S_{K,K'}$ is thus:

$$S_{K,K'} = \frac{T_K^1 - (T_K^{K'} - O_{K,K'})}{O_{K,K'}} = 1 - \frac{T_K^{K'} - T_K^1}{O_{K,K'}}$$

We have thus computed these values for each measurement for all kernels analyzed in [11], and assigned speed 0 for all measurements that resulted in too short of an overlap. Each kernel pair was measured 5 times, and the maximum speed recorded for each pair was used as a measure of how fast these kernels may run together. The CDF of the resulting speed distribution is shown on the left of Figure 12, which shows that about half of kernel pairs can not run together at all, and the speeds of the remaining pairs is roughly uniformly distributed. This does not tell the whole story though, since we expect correlations between the speed of kernels relative to each other. The complete speed matrix is provide on the right of Figure 12.

6.2. Experimental setting

From the speed measurements, we construct random instances to our scheduling problem in the following way. We fix a number n of tasks, and each task computes one of the 60 kernels, picked at random. To assign task durations, we can use the measured standalone time of the kernels (modeling the fact that this task computes the kernel once). This is the `actual` case, and it results in large variety in task execution times

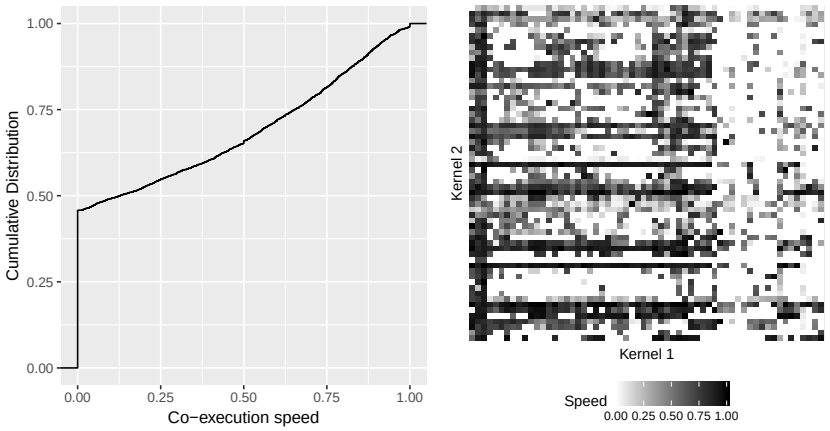


Figure 12: Speed values obtained from measurement data. Cumulative distribution on the left, speed matrix on the right.

because the kernels are very different from each other. This case comes in two flavors: either `uniform` where the kernels are picked with equal probability, or `weighted` where each kernel is picked with a probability inversely proportional to its duration. In the `weighted` case, if two kernels have duration d and d', the expected number of tasks of each type will be $c \cdot n/d$ and $c \cdot n/d'$ for some constant c. Thus, the expected total workload corresponding to each type of task is equal to $c \cdot n$, independently of the kernel duration. We have also analyzed the `random duration` case, in which the duration of a task is a uniform value between 0 and 10 (modeling the fact that each task computes one given kernel several times, and that tasks have relatively similar durations). In this last case, two tasks may represent the same kernel but have different durations.

For each case and each number of tasks, we have generated 15 different instances. For each instance, we evaluate the results of the preemptive and non-preemptive approaches. We first compute the solution to the preemptive Linear Program (PLP), and then we tested the two algorithms proposed here: (1) a preemptive solution with the smallest amount of preemptions by using the CaterpillarSplit algorithm (Algorithm 2), and (2) a non preemptive solution by using NonPreemptiveAlg (Algorithm 4). In addition, we also compute the result of directly solving the non-preemptive problem using a MILP approach with DirectAlg. The non preemptive results are compared to the graph-based algorithm MaxPair proposed in [25] to schedule pairs of kernels on the GPU.

All algorithms are implemented in Python 3.6.5. Linear Programs are solved with CPLEX 12.7, and the matching algorithm in MaxPair uses the NetworkX library[1] version 2.4.

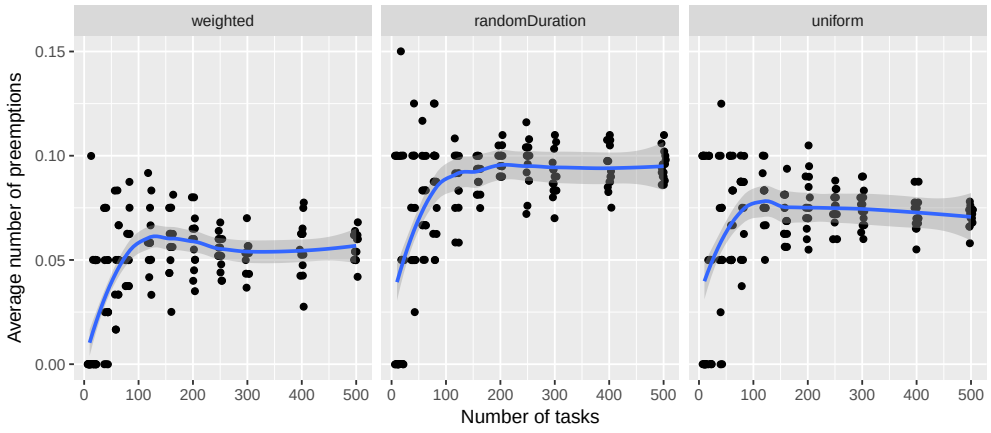


Figure 13: Analysis of the number of preemptions achieved by the CaterpillarSplit algorithm. Each dot is an experiment, the line represents a smoothed average with standard deviation shown in gray.

6.3. Experimental results

Figure 13 shows the number of preemptions achieved by the CaterpillarSplit algorithm, on the solution returned by the Linear Program. The plot actually shows the *average* number of preemptions per task, *i.e.* the total number of preemptions divided by the number of tasks. This average reaches a constant value relatively quickly, showing that our solution requires to preempt at most 12% of tasks (9% in average) for the worst setting `random duration`.

Experimental results in the non preemptive setting are provided on Figure 14. This plot shows the ratio of the makespan obtained by all three non preemptive algorithms, DirectAlg, NonPreemptiveAlg, and MaxPair, to the optimal preemptive makespan. Each experiment is represented by a dot, and a smoothed average is represented with a solid line. The gray area around the line represents the standard deviation. We can see that the solutions obtained with our NonPreemptiveAlg are very close to the optimal value, almost always within 5%, and within 2% on average. As expected, DirectAlg obtains even better schedules (within 1% for the `weighted` and `random duration` cases, and within 2% for `uniform`), but its long execution time makes it limited to about 120 tasks. On the other hand, the MaxPair approach obtains significantly worse results: the makespan is 5% away from the preemptive makespan in the `random duration` and `uniform` cases, and even 12% away for the largest instances of the `weighted` case. This last result is to be expected: because of how they are generated, these instances contain some long duration tasks, and many tasks with short duration. They will thus look like the worst case example of Figure 10, in which it is not possible to find a suitable task to process together with the long task, which incurs significant idle time.

[1] https://networkx.github.io/

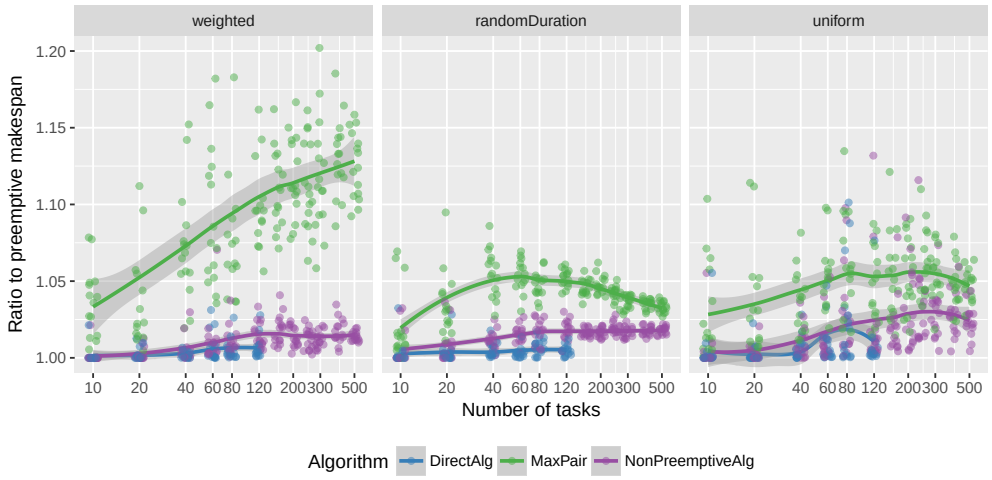


Figure 14: Experimental results for the three non-preemptive algorithms. Each dot is an experiment, the line represents a smoothed average with standard deviation shown in gray.

Execution time of all algorithms (preemptive and non preemptive) is shown on Figure 15, where logscale is used to better emphasize all the value ranges. We observe that the running time of DirectAlg increases much faster than the others, and becomes prohibitive for instances with more than 100 tasks. For the other algorithms, the plot shows that for small instances, the overhead of interfacing Python with the linear solver is significant. However these small instances are very quick to solve anyway (less than 100ms), so it is not clear that solving them very quickly is important. For larger size instances however (more than 80-100 tasks), the LP-based algorithms compute their solution faster. We can also observe that the complexity of MaxPair is higher: the plots are almost linear for large number of tasks, with a larger slope for MaxPair than for the others (remind that linear plots on a log-log scale denote a $y = x^c$ dependency, and c can be measured as the slope of the plot).

7. Conclusions

In this paper, we address the problem of scheduling different kernels on a single GPU. We propose a theoretical model based on real benchmarks, and provide optimal algorithms to build preemptive and non-preemptive schedules. We show that the optimal preemptive makespan can be computed in polynomial time, and that we can schedule any solution of optimal makespan with a minimal number of preemptions. However, computing the minimal amount of preemptions among all preemptive solutions of optimal makespan is an NP-hard problem. Similarly, computing the optimal non-preemptive schedule is NP-hard, but we present an algorithm that transforms a preemptive solution of optimal makespan into a non-preemptive solution with the smallest possible overhead.

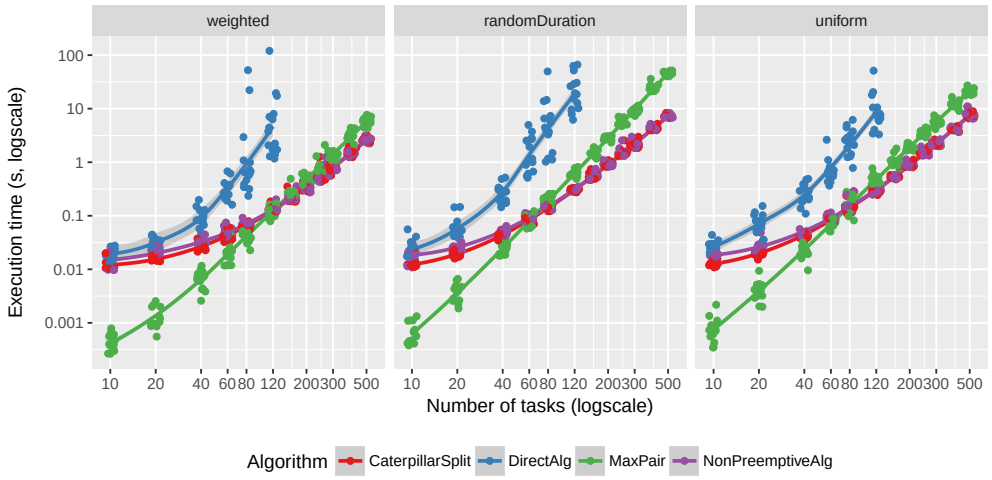


Figure 15: Execution time for all algorithms (logscale). Each dot is an experiment, the line represents a smoothed average.

We present an experimental evaluation of our algorithms on realistic instances, based on benchmarks of real applications. In the preemptive case, we show that our approach is able to achieve the optimal makespan by preempting 6 to 9% of all tasks depending on the experimental condition. In the non-preemptive case, We show that our algorithm produces schedules whose makespan is within 2.5% of the optimal preemptive schedules, whereas previous approaches exceed the preemptive makespan by 5 to 12%.

This work opens many challenging perspectives. One next step would be to consider the case of several GPUs: since our solutions are based on a set of paths, it seems possible in practice to assign paths to GPUs in a balanced way, if necessary by breaking additional edges to provide a better load-balancing. On the theoretical side, it would be valuable to provide worst-case estimates on the average number of preemptions (we conjecture that it can not be more than $\frac{1}{3}$), and to consider the problem with three kernels execution.

References

[1] J. Zhong, B. He, Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling, IEEE Transactions on Parallel and Distributed Systems 25 (6) (2014) 1522–1532.

[2] S. Pai, M. J. Thazhuthaveetil, R. Govindarajan, Improving GPGPU concurrency with elastic kernels, in: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2013, ACM, 2013, pp. 407–418.

[3] R. A. Cruz, C. Bentes, B. Breder, E. Vasconcellos, E. Clua, P. M. de Carvalho, L. M. Drummond, Maximizing the gpu resource usage by reordering concurrent kernels submission, Concurrency and Computation: Practice and Experience.

[4] P. Carvalho, R. Cruz, L. M. Drummond, C. Bentes, E. Clua, E. Cataldo, L. A. Marzulo, Kernel concurrency opportunities based on gpu benchmarks characterization, Cluster Computing (2019) 1–12.

[5] J. T. Adriaens, K. Compton, N. S. Kim, M. J. Schulte, The case for GPGPU spatial multitasking, in: 2012 IEEE 18th International Symposium on High Performance Computer Architecture (HPCA),, 2012, pp. 1–12.

[6] X. Zhao, Z. Wang, L. Eeckhout, Classification-driven search for effective sm partitioning in multitasking gpus, in: Proceedings of the 2018 International Conference on Supercomputing, ACM, 2018, pp. 65–75.

[7] M. Shantharam, Y. Youn, P. Raghavan, Speedup-aware co-schedules for efficient workload management, Parallel Processing Letters 23 (02) (2013) 1340001.

[8] G. Aupy, M. Shantharam, A. Benoit, Y. Robert, P. Raghavan, Co-scheduling algorithms for high-throughput workload execution, Journal of Scheduling 19 (6) (2016) 627–640.

[9] H. Sun, R. Elghazi, A. Gainaru, G. Aupy, P. Raghavan, Scheduling parallel tasks under multiple resources: List scheduling vs. pack scheduling, in: 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2018, pp. 194–203.

[10] Q. Hu, J. Shu, J. Fan, Y. Lu, Run-time performance estimation and fairness-oriented scheduling policy for concurrent GPGPU applications, in: 45th International Conference on Parallel Processing (ICPP), 2016, 2016, pp. 57–66.

[11] P. Carvalho, L. M. Drummond, C. Bentes, E. Clua, E. Cataldo, L. A. Marzulo, Analysis and characterization of gpu benchmarks for kernel concurrency efficiency, in: Latin American High Performance Computing Conference, Springer, 2017, pp. 71–86.

[12] Y. Ukidave, X. Li, D. Kaeli, Mystic: Predictive scheduling for gpu based cloud servers using machine learning, in: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2016, pp. 353–362.

[13] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, , K. Skadron, Rodinia: A benchmark suite for heterogeneous computing, In Proceedings of the IEEE International Symposium on Workload Characterization (IISWC) (2009) 44:54.

[14] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, W. mei W. Hwu, Parboil: A revised benchmark suite for scientific and commercial throughput computing (2012).

[15] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, J. S. Vetter, The scalable heterogeneous computing (SHOC) benchmark suite, Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (2010) 63:74.

[16] Nvidia gp100 pascal whitepaper, http://www.nvidia.com.

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

[18] J. J. K. Park, Y. Park, S. Mahlke, Chimera: Collaborative preemption for multitasking on a shared gpu, in: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2015, ACM, 2015, pp. 593–606.

[19] Z. Lin, L. Nyland, H. Zhou, Enabling efficient preemption for simt architectures with lightweight context switching, in: International Conference for High Performance Computing, Networking, Storage and Analysis, SC16, IEEE, 2016, pp. 898–908.

[20] C. Li, J. Yang, A. Zigerelli, Y. Guo, Pep: proactive checkpointing for efficient preemption on gpus, in: 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC), IEEE, 2018, pp. 1–6.

[21] C. Margiolas, M. F. O'Boyle, Portable and transparent software managed scheduling on accelerators for fair resource sharing, in: Proceedings of the 2016 International Symposium on Code Generation and Optimization, ACM, 2016, pp. 82–93.

[22] S. Kato, K. Lakshmanan, R. Rajkumar, Y. Ishikawa, Timegraph: Gpu scheduling for real-time multitasking environments, in: Proc. USENIX ATC, 2011, pp. 17–30.

[23] Q. Chen, H. Yang, J. Mars, L. Tang, Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers, ACM SIGPLAN Notices 51 (4) (2016) 681–696.

[24] Q. Chen, H. Yang, M. Guo, R. S. Kannan, J. Mars, L. Tang, Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers, ACM SIGARCH Computer Architecture News 45 (1) (2017) 17–32.

[25] Y. Wen, M. F. O'Boyle, C. Fensch, Maxpair: Enhance opencl concurrent kernel execution by weighted maximum matching, in: Proceedings of the 11th Workshop on General Purpose GPUs, ACM, 2018, pp. 40–49.

[26] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, M. Guo, Quality of service support for fine-grained sharing on gpus, in: Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA), 2017, ACM, 2017, pp. 269–281.

[27] Q. Xu, H. Jeon, K. Kim, W. W. Ro, M. Annavaram, Warped-slicer: efficient intra-sm slicing through dynamic resource partitioning for gpu multiprogramming, in: Proceedings of the 43rd Annual International Symposium on Computer Architecture (ISCA), 2016, IEEE, 2016, pp. 230–242.

[28] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, M. Guo, Simultaneous multikernel gpu: Multi-tasking throughput processors via fine-grained sharing, in: IEEE International Symposium on High Performance Computer Architecture (HPCA), 2016, IEEE, 2016, pp. 358–369.

[29] G. Chen, Y. Zhao, X. Shen, H. Zhou, Effisha: A software framework for enabling effficient preemptive scheduling of gpu, ACM SIGPLAN Notices 52 (8) (2017) 3–16.

[30] S. Jin, Z. Wang, Q. Chen, M. Guo, Preemption-aware kernel scheduling for gpus, in: 2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC), IEEE, 2017, pp. 525–532.

[31] B. Wu, X. Liu, X. Zhou, C. Jiang, Flep: Enabling flexible and efficient preemption on GPUs, in: Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, 2017, pp. 483–496.

[32] F. Wende, F. Cordes, T. Steinke, On improving the performance of multi-threaded CUDA applications with concurrent kernel execution by kernel reordering, in: Symposium on Application Accelerators in High Performance Computing (SAAHPC), 2012, 2012, pp. 74–83.

[33] T. Li, V. K. Narayana, T. El-Ghazawi, A power-aware symbiotic scheduling algorithm for concurrent GPU kernels, in: IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS), 2015, 2015, pp. 562–569.

[34] V. T. Ravi, M. Becchi, G. Agrawal, S. Chakradhar, Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework, in: Proceedings of the 20th international symposium on High performance distributed computing, ACM, 2011, pp. 217–228.

[35] Y. Liang, H. P. Huynh, K. Rupnow, R. S. M. Goh, D. Chen, Efficient GPU spatial-temporal multitasking, IEEE Transactions on Parallel and Distributed Systems 26 (3) (2015) 748–760.

[36] M. E. Belviranli, F. Khorasani, L. N. Bhuyan, R. Gupta, Cumas: Data transfer aware multi-application scheduling for shared gpus, in: Proceedings of the 2016 International Conference on Supercomputing, ACM, 2016, p. 31.

[37] S. Moran, I. Newman, Y. Wolfstahl, Approximation algorithms for covering a graph by vertex-disjoint paths of maximum total weight, Networks 20 (1) (1990) 55–64, available at http://www.cs.technion.ac.il/~moran/r/PS/MNW90.pdf.

[38] J. Edmonds, E. L. Johnson, Matching: A well-solved class of integer linear programs, in: Combinatorial Optimization—Eureka, You Shrink!, Springer, 2003, pp. 27–30, original Publication: 1969.

[39] Y. Shiloach, Another look at the degree constrained subgraph problem, Information Processing Letters 12 (2) (1981) 89 – 92. doi:https://doi.org/10.1016/0020-0190(81)90009-0.
URL http://www.sciencedirect.com/science/article/pii/0020019081900090

	Type of Scheduling	**Scheduling Decisions**	**Scheduling Algorithms**	**Goals**
AccelOs [21]	Non-preemptive	Compilation time	Modifies the size of the thread block	Improve fairness
TimeGraph [22]	Non-preemptive	Pre-defined priorities	Priority queue	Make high-priority tasks responsive
Baymax [23]	Non-preemptive	Task Duration Prediction	Reorder queue of submissions	Guarantee QoS
Prophet [24]	Non-preemptive	Performance interference model for pair of kernels	Different scheduling queues by kernel type with a ready time table	Guarantee QoS of latency-sensitive applications
Mystic [12]	Non-preemptive	Interference prediction based on machine learning	Select on-the-fly the least interference score	Guarantee QoS and improve system throughput
MaxPair [25]	Non-preemptive	Relative speedups (experimentally)	Graph-based weighted max matching	Maximize system throughput
Fine-Grain QoS [26]	Preemptive	IPC goal	Allocates core compute cycles following quotas	Guarantee QoS
Real Time [27]	Preemptive	Worst-case execution time (WCET) analysis	Earliest deadline first(EDF)	Enhance schedulability of real-time tasks
SMK [28]	Preemptive	Kernel and SM resource usage	Resource partitioned, with a heuristic to the Knapsack problem	Fair sharing among concurrent kernels
EffiSha [29]	Preemptive	Preemption-enabling code transformation	Priority queue	Enable preemption
Preemption-aware [30]	Preemptive	Cache miss behavior	Greedy algorithm	Improve throughput
FLEP [31]	Preemptive	Performance model	Highest priority first and Fairness first	Fairness and Improve normalized average turnaround time
Our approach	Preemptive and Non-preemptive	Relative speedups (experimentally)	Linear Programming and two graph algorithms	Minimize makespan and the number of preemptions

Table 1: Comparison of our work with previous approach, considering the type of scheduling strategy, the main input in scheduling decisions, the scheduling algorithm used and the main goals of the algorithm.