Algorithms for Preemptive Co-scheduling of Kernels on GPUs

Lionel Eyraud-Dubois, Cristiana Bentes

Algorithms for Preemptive Co-scheduling of Kernels on GPUs

Lionel Eyraud-Dubois
Inria
University of Bordeaux
Bordeaux, France, 33000
lionel.eyraud-dubois@inria.fr

Cristiana Bentes
Systems Engineering Department
State University of Rio de Janeiro
Rio de Janeiro, Brazil
cris@eng.uerj.br

***Abstract*—Modern GPUs allow concurrent kernel execution and preemption to improve hardware utilization and responsiveness. Currently, the decision on the simultaneous execution of kernels is performed by the hardware, which can lead to unreasonable use of resources. In this work, we tackle the problem of co-scheduling for GPUs in high competition scenarios. We propose a novel graph-based preemptive co-scheduling algorithm, with the focus on reducing the number of preemptions. We show that the optimal preemptive makespan can be computed by solving a Linear Program in polynomial time. Based on this solution we propose graph theoretical model and an algorithm to build preemptive schedules which minimizes the number of preemptions. We show, however, that finding the minimal amount of preemptions among all preemptive solutions of optimal makespan is a NP-hard problem. We performed experiments on real-world GPU applications and our approach can achieve optimal makespan by preempting 6 to 9% of the tasks.**

I. Introduction

Graphics Processing Units (GPUs) are throughput-oriented co-processors that are witnessing a rapid increase in the amount of computing resources. To avoid keeping these growing resources underutilized and improve performance, concurrent kernel execution (CKE) has been proposed and showed improved GPU throughput and resource utilization [1], [2], [3].

Recently, GPU architectures also provide hardware preemption. The preemption can occur at coarse and fine-grain granularity: 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. In NVIDIA GPUs, the switching of kernels in thread boundary can complete in less than 100 μs [4]. 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, which takes significant memory bandwidth.

In this scenario, scheduling decisions are key to increase resource utilization and improve responsiveness. However, 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 [5]. This allocation policy has shown to lead to an unreasonable use of resources [2], and to be influenced by the order in which the kernels are launched [6], [7]. Therefore, 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.

In this work, we study the problem of co-scheduling kernels to the GPU for future scenarios that present high competition and the GPU will behave as multi-programmed devices, such as CPUs are in the present. Although the problem of co-scheduling applications on CPU multicore nodes has been well studied in the past [8], [9], [10], they are not directly applicable to the GPU environment due to the GPU hardware allocation policies. We propose a novel graph-based preemptive co-scheduling solution, that defines the kernels submission order focusing on reducing the number of preemptions. The idea is to exploit the kernels interference profile provided by previous works analysis of how the kernels resource usage impacts the interference in their co-execution [11], [12]. More formally, we deal with the following problem: given a co-scheduling speedup matrix S, where $S_{i,j}$ is 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 should co-execute. From G, we propose the algorithm CATERPILLARSPLIT to generate kernel submissions with the minimum number of preemptions. However, we show that the problem of minimizing the number of preemptions directly from the speedup matrix S is NP-hard.

We used 60 kernels from real-world applications obtained from the main benchmark suites for evaluating GPUs, Rodinia, Parboil and SHOC, and observed that our algorithms produce very efficient schedules. We obtain the optimal makespan by preempting between 5% and 10% of the kernels depending on the setting.

The rest of the paper is organised as follows. In Section II, we review related works. In Section III, we present the notations and the scheduling model used throughout the paper. Section IV shows how to obtain preemptive schedules with minimal makespan, and how to minimize the number of preemptions. Finally, Section V gives an experimental validation of our algorithms on realistic instances.

II. RELATED WORK

Enabling efficient multiprogramming on GPUs is receiving a lot of attention from the research community in recent years. Software techniques, such as *reordering* were proposed to find a good submission order to improve resource utilization [7], [13], [14]. Other software techniques propose modifying the granularity of the kernels to create more concurrency opportunities [2], [15], [16]. Hardware techniques were also proposed to divide the GPU resources among the concurrent kernels, called *spatial multitasking* [1], [17].

The non-preemptive co-scheduling of kernels of different applications on the GPU have also been studied. The work of Margiolas and O'Boyle [18] 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 different kernels. Chen *et al.* [19] propose a task duration predictor and a task reordering mechanism based on the predictions to guarantee QoS. Ukidave *et al.* [12] present an interference-aware mechanism for co-scheduling on GPUs based on machine learning to predict whether kernels can share a GPU efficiently. Wen *et al.* [20] propose a graph-based algorithm to schedule kernels in pairs. The recent work of Shekofteh *et al.* [6] proposes the co-scheduling of pair of kernels with different execution behaviors. They use kernel slicing to improve the choice of kernels pairs for co-scheduling.

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.* [21] 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.* [22] 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.* [23] 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.* [24] proposed a compiler-runtime software solution for priority-based preemptive scheduling on GPUs, that transforms the kernel for voluntary preemptions. Jin *et al.* [25] 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.

Compared to the other approaches, we propose a preemptive algorithm 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.

III. 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 problem:

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

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 $P(\sigma) = \sum_{i \in \mathcal{T}} P(\sigma, i)$.

a) 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.

IV. PREEMPTIVE SCHEDULES

A. 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 this 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*.

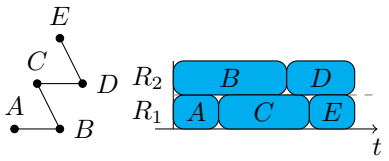
Theorem 1. *For any optimal solution x^*, $G(x^*)$ is a pseudo-forest.*

B. Minimizing the number of preemptions

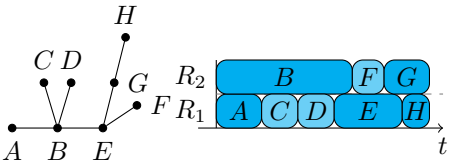
In this Section, we describe how to minimize the number of preemptions of a preemptive solution. We start by characterizing the graphs $G(x^*)$ from which it is possible to build a non-preemptive schedule. As shown on Figure 1a, if the graph is a single path we can build a schedule without any preemption, just by following the path. Case 1b 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 1b the path is $A-B-E-G-H$, with C, D and F (shown in lighter color) connected to it. Case 1c 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 1d shows that any cycle incurs at least one preemption.

This brings us to the following Lemma:

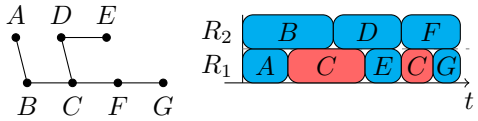
Lemma 1. *The graph G of a preemptive solution can be ordered in a non preemptive way if and only if it is a vertex-disjoint collection of caterpillars.*



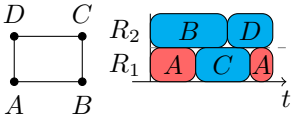
(a) Simple path



(b) Caterpillar graph



(c) More complex tree



(d) Cycle

Figure 1: Possible cases to turn the graph G into a schedule.

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.

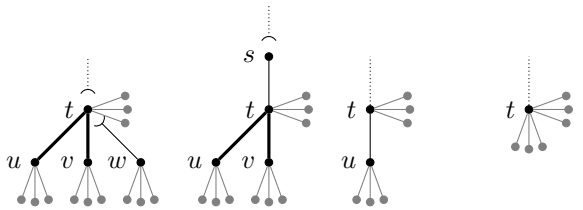
We call the path P the *internal path*, and all nodes and edges of this path are called *internal*.

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 2 (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.



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

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

1) Solving the problem on trees: On trees, the problem CATERPILLARSPLIT can be formulated this way:

Problem 3. 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, illustrated on Figure 2. We assume that the tree T is rooted at some arbitrary root r, allowing to 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.

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 caterpillars, 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, proving 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. □

Algorithm 1: CaterpillarSplit(T, r)

Input: A tree T, rooted at r
Output: A partition of T in caterpillars

$S \leftarrow \emptyset$;
while *T has vertices of height 2* **do**
  $t \leftarrow$ 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**
    $C \leftarrow$ caterpillar (u, t, v) with leaves attached, and the edge from t to its parent;
    $S \leftarrow S \cup \{C\}$;
    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;
    $C \leftarrow$ the caterpillar (u, t, v) with leaves attached;
    $S \leftarrow S \cup \{C\}$;
    Remove edges of C from T;
return $S \cup \{T\}$

These lemmas suggest an algorithm to solve CATERPILLARSPLIT on trees, which is detailed in Algorithm 1. 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.

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),

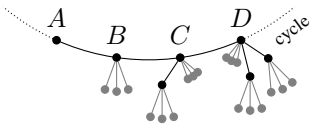


Figure 3: Four possible shapes for a node on the cycle.

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.

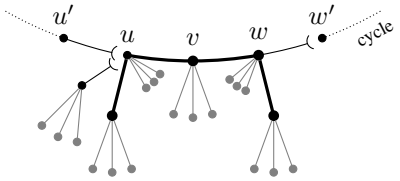
For each $u \in C$, the subgraph of nodes reachable from u without going through C is a tree T_u. By rooting T_u at u, all the Lemmas proven above are still valid, with one exception: Lemma 3 cannot be applied if the vertex t of height 2 is u, since u has no father in T_u. This implies that iteratively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

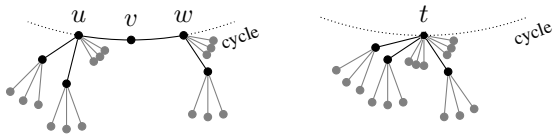
If two legs are separated on the cycle by at least two nodes, or by one node of shape B, we can reduce the problem with the following lemmas:

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. This proof is very similar to the proofs of Lemma 2; in the interest of brevity we only present the main points. In the first part, we prove that there exists an optimal solution in which edges (u, v) and (v, w) are part of the same caterpillar: from any optimal solution in which this is not the case, a case analysis on whether edges (u, v) and (v, w) are internal edges in their respective caterpillar shows that we can always obtain another optimal solution with the required property. Then, we show that such a solution can always be transformed into an optimal solution which satisfies the conditions of this Lemma. □



(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 4: Illustrations for Lemmas 5 and 7.

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, where in the first part we state 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, 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 12, 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 15, 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.

Algorithm 2: CaterpillarSplit(T, C)

Input: A pseudo-tree T, containing one cycle C
Output: A partition of T in caterpillars

1 **foreach** $u \in C$ **do**
2 Solve CATERPILLARSPLIT on T_u, stop when u has at most two children of height 1;
3 **if** *no node of C has a leg* **then**
4 Split any node of C to obtain a path;
5 **return** the caterpillar made of this path and all their leaves attached;
6 **else if** *exactly one node u of C has one or two legs* **then**
7 Split node u to obtain a path;
8 **return** the caterpillar made of this path and all their leaves attached;
9 **else if** *there are at least two legs on cycle C* **then**
10 **if** *there exists two legs separated by a forcing node v* **then**
11 $C_v \leftarrow$ caterpillar forced by v (Lemma 5 or 6);
12 $S \leftarrow$ CATERPILLARSPLIT $(T \setminus C_v)$ (Alg. 1);
13 **return** $S \cup \{C_v\}$;
14 **else**
15 Repeatedly merge two consecutive nodes on the cycle (Lemma 7) and apply Lemma 2;
16 **return** the resulting caterpillars

C. Minimization of both makespan and preemptions

The algorithms presented above work in two steps: 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.

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.

Theorem 2. MINPREEMPTIONS *is NP-complete.*

Proof. The problem clearly belongs 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 its 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 problem:

- 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	$\cdot$

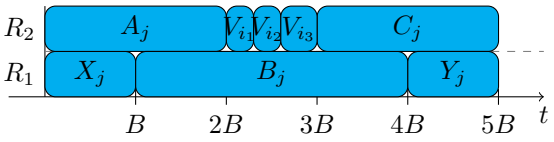


Figure 5: Valid schedule for group j with $P_j = \{i_1, i_2, i_3\}$ if the 3-PARTITION instance is positive.

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 5, where each group j is associated with one part P_j).

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. □

V. EXPERIMENTAL EVALUATION

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

A. 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 a framework that support 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.

Consider a given measurement for kernels K and K', whose standalone execution times are respectively T^1_K and $T^1_{K'}$. 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 $S_{K,K'} = \frac{T^1_K}{T^{K'}_K}$ since the total work is T^1_K, 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:

$$S_{K,K'} = \frac{T^1_K - (T^{K'}_K - O_{K,K'})}{O_{K,K'}} = 1 - \frac{T^{K'}_K - T^1_K}{O_{K,K'}}$$

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 6, 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 6.

B. 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 because the kernels are very different from each other. This case comes

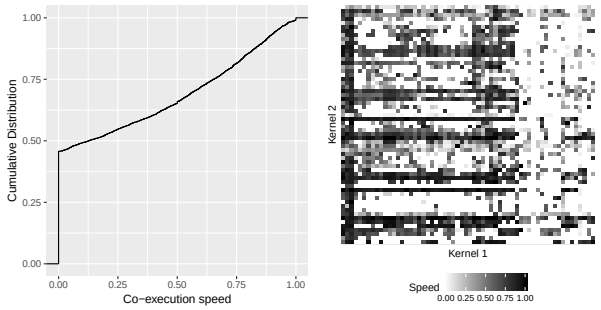


Figure 6: Speeds obtained from measurement data. Cumulative distribution on the left, speed matrix on the right.

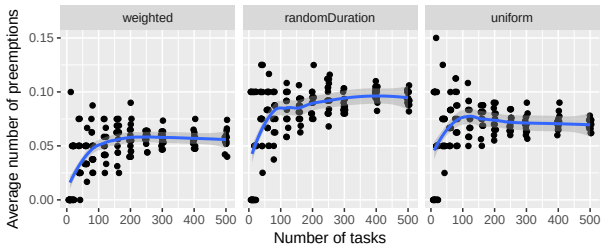


Figure 7: Analysis of the number of preemptions achieved by the CATERPILLARSPLIT algorithm. Each dot is an experiment, the line represents a smoothed average with standard deviation shown in grey.

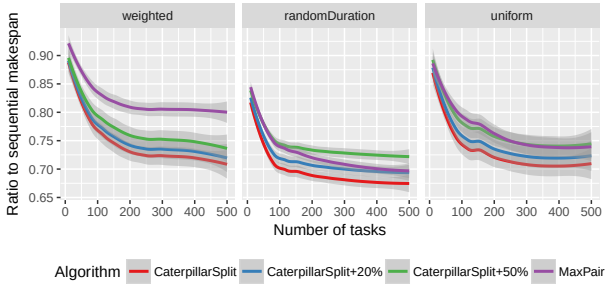


Figure 8: Performance of CATERPILLARSPLIT (with and without preemption overhead) compared to MAXPAIR.

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. We first compute the solution to the preemptive Linear Program (PLP), and then, for each instance, we evaluate the results of the preemptive solution obtained with the CATERPILLARSPLIT algorithm (Algorithm 2). For comparison, we also compute the sequential solution (where each task is executed alone) and the solution obtained from the graph-based algorithm MAXPAIR [20] which schedules tasks non-preemptively by grouping them by best-assorted pairs.

All algorithms are implemented in Python 3.6.5. Linear Programs are solved with CPLEX 12.7.

C. Experimental results

Figure 7 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`.

Figure 8 shows the performance in terms of schedule length, compared to a sequential solution. In addition to the results of CATERPILLARSPLIT and MAXPAIR, this plot includes schedule durations for CATERPILLARSPLIT with the very conservative assumption that each preemption incurs an overhead of either 20 or 50% of the average task duration, during which the GPU can not process any task. We first see that using concurrent execution allows to reduce execution time by 20 to 30%. Using our preemptive solutions allows to reduce this time further, by 10-12% for the `weighted` case, by 5-7% for the `uniform` case, and by 3-5% for the `random duration` case. Furthemore, our solution is more efficient in all cases even with 20% overhead, and in both the `weighted` and `uniform` cases with the very conservative 50% overhead.

VI. CONCLUSIONS

In this paper, we address the problem of co-scheduling on a GPU. We propose a theoretical model and pro-

vide a preemptive scheduling algorithm. 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.

We present an experimental evaluation of our algorithm on realistic instances, based on benchmarks of real applications. 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, which allows to obtain very good performance compared to the literature, even with a conservative overhead.

This work opens many challenging perspectives. One next step would be to design an algorithm to produce non-preemptive schedules which could be more efficient than MAXPAIR, or more generally to take a preemption cost into account. 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 concurrent execution of three kernels.

REFERENCES

[1] 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.

[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] 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.

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

[5] 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.

[6] S. . Shekofteh, H. Noori, M. Naghibzadeh, H. Fröning, H. S. Yazdi, ccuda: Effective co-scheduling of concurrent kernels on gpus, IEEE Transactions on Parallel and Distributed Systems 31 (4) (2020) 766–778.

[7] 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.

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

[9] 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.

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

[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] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.