A Scheduling Theory Framework for GPU Tasks Efficient Execution

A.J. Lázaro-Muñoz, J.M. González-Linares, B. López-Albelda y N. Guil[1]

Abstract— **Concurrent execution of tasks in GPUs can reduce the computation time of a workload by overlapping data transfer and execution commands. However it is difficult to implement an efficient run-time scheduler that minimizes the workload makespan as many execution orderings should be evaluated. In this paper, we employ scheduling theory to build a model that takes into account the device capabilities, workload characteristics, constraints and objective functions. In our model, GPU tasks scheduling is reformulated as a flow shop scheduling problem, which allow us to apply and compare well known methods already developed in the operations research field. In addition we develop a new heuristic, specifically focused on executing GPU commands, that achieves better scheduling results than previous techniques. Finally, a comprehensive evaluation, showing the suitability and robustness of this new approach, is conducted in three different NVIDIA architectures (Kepler, Maxwell and Pascal).**

Keywords— **Scheduling Theory, Flow Shop, CUDA Streams.**

I. Introduction

In a typical application executed in current heterogeneous parallel architectures, some parts are executed in the CPU or host, while others parts are delegated (offloaded) to the GPU. Besides, GPUs are broadly used in multitask environments, e.g. data centers, where applications running on CPUs offload specific functions to GPUs in order to take advantage of the device performance. This way, it is probable to have several independent tasks ready to run concurrently in a GPU. In this context, several works have been published that try to improve the way tasks are scheduled on GPUs [1], [2], [3]. The most recent ones propose hardware [4], [5], [6], [7] or software [8], [9], [10] solutions that support preemption and offer responsiveness, fairness or quality of service capabilities. The proposed solutions are focused on kernel execution and they do not take into account the transfer of data required to compute those kernels. In many cases, the time taken by these transfers is not negligible and can affect the performance of the applied scheduling policy. In addition, software approaches typically require to modify the original kernels.

The impact of transfers on the total execution time is given by the fact that the computation on the device must wait for the transfer of input data from CPU memory to GPU memory to be finished. This delay entails an overhead that can be alleviated by overlapping data transfers with computation. Some Application Programming Interfaces (API) such as CUDA [11] and OpenCL [12] provide features that allow to overlap communication and computation, e.g., CUDA streams or OpenCL commands queues. These features rely on the use of asynchronous communications and hardware managed command queues, and a large performance gain can be obtained when properly configured.

The importance of properly configuring the concurrent execution of several tasks can be seen in the next example, where one single host process launches tasks onto an Nvidia K20c GPU. Figure 1 shows two time-lines corresponding to the execution of four independent tasks in a different order. The tasks have been selected from CUDA SDK [13] and they are scheduled employing a different stream per task. More precisely, the tasks are Matrix Multiplication (Stream 0), Black Scholes (Stream 1), Separable Convolution (Stream 2) and Vector Addition (Stream 3). Time-lines include the time spent in data transfer commands from host to device (HtD) and from device to host (DtH), and the kernel computation commands in the GPU (Kernel) for every stream. As it can be seen from Figure 1, there is no overlapping between kernels execution from different tasks as these tasks are able to exhaust at least one of the available GPU resources (registers, shared memory, etc.). Nevertheless, the total execution time for the bottom execution order is shortened by 28% thanks to a higher overlap between the transfer times of some tasks and the kernel execution times of other tasks. The current Nvidia hardware scheduler solely utilizes the kernel resource requirements to select the order that kernels blocks are launched in the GPU, thus either of these orderings is possible. These results show the importance of choosing the best execution order for a set of GPU tasks so that overlapping between data transfers and kernel computation is optimized.

On the other hand, scheduling theory is a field of applied mathematics that deals with the problem of optimal ordering of a set of jobs. It is used in many areas like management, production, transportation or computer systems. In scheduling theory, a flow shop is an ordered set of processors $P = < P_1, P_2, \ldots, P_m >$ such that the first operation of each job is performed on processor P_1, the second on processor P_2, and so on, until the job completes execution on P_m [14]. Many works have been published on the problem of scheduling a collection of jobs on a flow shop, to optimize measures like the maximum finishing time or the utilization of the machines [15].

In this paper we expose how to apply methods in-

[1]University of Málaga, Andalucía Tech, Dept. of Computer Architecture, Spain, e-mail: {alazaro, jgl, blopeza, nguil}@uma.es.

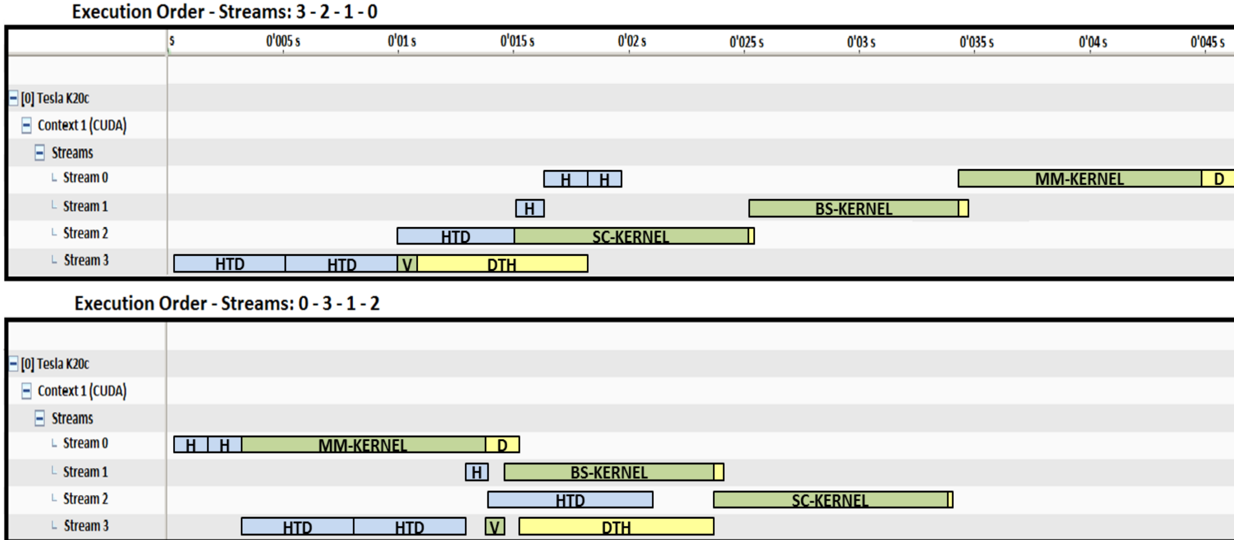

Fig. 1. Two profiling views for the concurrent execution of four tasks belonging to CUDA SDK on a Nvidia K20c card. Matrix Multiplication (MM-Kernel), Black Scholes (BS-Kernel), Separable Convolution (SC-Kernel) and Vector Addition (VA-Kernel) computation task are launched by streams 0, 1, 2, and 3 respectively. The host to device transfers, kernels computation and device to host transfers are represented by blue, green and yellow boxes respectively. Top and bottom views correspond to two different orderings in streams execution.

troduced in scheduling theory to solve GPU tasks execution scheduling. More precisely, we show that the concurrent execution of GPU tasks using CUDA streams can be modelled as a flow shop problem. As an example we also present a run-time scheduler implementation that significantly reduces the makespan of a workload and, consequently, increase the use of the GPU. It takes into account data transfers and GPU capability to overlap transfers and computation. Moreover, in contrast with other software scheduling approaches, original kernels of the tasks do not need to be modified.

The rest of the paper is organized as follows. First, scheduling theory is reviewed in Section II, with a focus on GPU tasks scheduling using the flow shop problem. Next, in Section III, several heuristics obtained from the operations research field are discussed, and a new algorithm that merges the previous theory with a GPU tasks execution model is presented. Then, several experiments that show the suitability and robustness of this new approach are conducted in Section IV. Finally, conclusions are drawn in Section VI.

II. Scheduling theory applied to GPU tasks execution

The problem of launching N tasks in a GPU can be studied using scheduling theory. Scheduling is a decision-making process where n jobs are allocated to m machines. Following the notation introduced by Graham et al. [16], the problem can be identified by three fields, α (that describes the machine environment), β (the processing characteristics and constraints), and γ (the objective function). Below it is shown the most relevant values that can be used for these fields in a GPU tasks launching context.

Launching a task (job in scheduling terminology) in a NVIDIA GPU typically involves executing three commands: a Host to Device transfer (HtD), a kernel command (K), and a Device to Host transfer (DtH). Transfer commands are executed in DMA engines and kernel commands in the GPU itself (through the Grid Management Unit). Although current NVIDIA architectures allow Concurrent Kernel Execution if kernels do not exhaust any of the hardware resources (memory, registers, etc), most kernels in real applications are designed to fully utilize these resources, thus we restrict our analysis to kernels that do not execute concurrently. On the other side, modern GPUs have two DMA engines that allow simultaneous transfer commands in opposite directions, thus we can consider 3 machines in our environment (two DMA engines plus the GPU). This corresponds to a flow shop problem, that is, a problem where each job must be processed on each machine following a given order. Then, field α is represented by $F3$ to describe a 3-machine flow shop problem. Figure 2 shows an example with two jobs in a 3-machine flow shop problem.

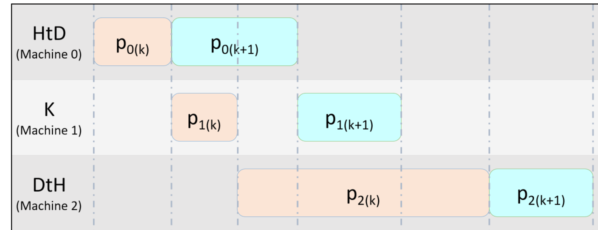

Fig. 2. GPU tasks launching as a 3-machine flow shop problem. HtD commands are executed by machine 0, K commands by machine 1 and DtH commands by machine 2. Commands from kth job precede commands from $(k+1)$th job in all machines (permutation flow shop). $p_{i(j)}$ corresponds to the processing time of job j in machine i.

A generalization of this problem is the flexible flow

shop (also known as hybrid flow shop), where instead of m machines in series there are c stages in series. Each stage includes a number of identical machines in parallel and can be used to describe a cluster of GPUs. Thus, there would be 3 stages (one stage for each command type), with as many machines as GPUs in the cluster, and the problem would be represented as $FF3$.

Regarding the processing characteristics and constraints, several values can be taken into account. For example, a common type of flow shop problems considers that every machine operates under the assumption of First Come First Served policy. This policy does not hold under devices with HyperQ unless events are used to enforce an ordering. On the other hand, it is known that for $F3$ systems there always exist optimal schedules that do not require sequence changes between machines [17] and, in that case, some simplifications can be made that make easier to find an optimal schedule. Thus, in this work events are used to enforce the same ordering in all the machines. This type of flow shop problem is called permutation flow shop and it is noted using the word $prmu$ in the β field. In the example shown in Figure 2 it can be seen than commands from kth job precede commands from $(k+1)$th job in all machines.

In a real world scenario, one or more processes (or threads) execute part of their computation in the CPU and other part in the GPU. Thus, scheduled tasks can arrive to the system at some particular time called release date (r_j). These tasks cannot be launched before its release date and this is noted adding this value, r_j, to the β field. Moreover, each process could execute many different tasks in the GPU, with some precedence constraints among them to ensure correct execution. This constraints are typically encoded using a directed acyclic graph and can be expressed using the word $prec$ in the β field. Finally, tasks could belong to different users thus they must be launched in different GPU contexts. In scheduling theory this is equivalent to job families, that is, jobs in the same family can be processed one after another without any delay, but switching from one family to another requires a setup time, s, before jobs are launched. This setup time is the time that takes to create a GPU context, and in the scheduling theory notation is expressed adding the word $fmls$ to the β field.

It is also possible to include information about the job characteristics in this β field, like its processing time or its priority. For example, we will refer to p_{ij} as the processing time of job j in machine i (as in Figure 2, where $p_{0(k)}$ is the processing time of kth job in machine 0 and so on). Although it is possible to obtain an estimation of the processing time of each command in advance, the real transfer times depend on whether there is another transfer in the opposite direction or not [18]. Therefore, these sources of uncertainties may lead to sub-optimal schedules when solving the objective function. A priority factor can also be assigned to each job using a weight value, w_j, that can be taken into account in the objective function. Furthermore, due dates d_j that reflect the completion date of job j can be also included in the β field.

Finally, the objective function that defines the optimal schedule must be considered. The completion time of job j can be denoted by C_j. The most common function in flow shop problems is to minimize the makespan C_{max}, defined as $\max(C_1, ..., C_n)$, and it is equivalent to minimize the completion time of the last job to leave the system. In this type of problems it is almost equivalent to maximize the usage of the machines that is our main objective. Another useful objective function is the total weighted completion time ($\sum w_j C_j$), also referred as the weighted flow time, that takes into account the priority factors given by the weight values.

Table I resumes these values with their meaning and a short description. All these parameters can be combined to express different real world situations and to select an appropriate solution from the existing literature [19], [20]. The only requisite is to obtain the right notation that captures the problem. For example, a simple scenario with a single GPU and several independent tasks, ready to be executed using the same context, can be modelled as a $F3|prmu|Cmax$ problem. On the other side, a more complex scenario like a multi-GPU system where several users launch independent tasks at different release dates can be studied as a $FF3|r_j, fmls|Cmax$. As a practical example, in next section we will study the $F3|prmu|Cmax$ problem that arises when several threads launch independent kernels that can be computed using the same GPU context.

III. Flow shop scheduling

The problem presented in this section is typical in many situations where one or more threads launch several tasks that can be computed using the same GPU context. For example, in a video surveillance application, there could be several threads analyzing different video streams and part of the computation could be offloaded to a GPU to accelerate the whole process. In a similar way to CUDA MultiProcess-Service (MPS, [21]), a proxy thread could be used to collect all tasks, and launch them using the same GPU context to improve concurrency among tasks. Figure 3 shows this situation. Two threads, A and B, must execute independent CUDA tasks. A proxy thread takes these tasks and schedule them using the same GPU context. Each task is assigned to a different stream, thus some of the commands of one task can be overlapped with the commands of the other task. As it was shown in the introduction, the total execution time can be reduced by choosing the best execution order.

Each task consists of several commands (i.e., HtD, K and DtH commands) that are executed by the DMA engines and the GPU. The total execution time (the makespan C_{max}) can be reduced by selecting the

TABLE I
SCHEDULING THEORY NOTATION

Field	Value	Meaning	Description
α	Fm	Flow Shop with m machines	Each job is processed following a given order
	FFc	Flexible (or Hybrid) Flow Shop with c stages	As Fm but using c stages of identical machines
β	$prmu$	Permutation	Machines operate under FIFO policy
	r_j	Release dates	Tasks arrive at some particular release dates
	w_j	Priority factors	Tasks are assigned a weight or priority factor
	d_j	Due dates	Tasks are assigned due dates
	$prec$	Precedence constraints	Tasks must follow a directed acyclic graph
	$fmls$	Job families	Tasks belong to different users
γ	C_{max}	Makespan	Minimize completion time of last task
	$\sum w_j C_j$	Total Weighted Completion Time	Priority factors are included

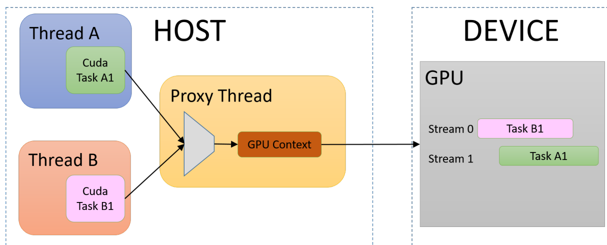

Fig. 3. A host proxy thread collect GPU tasks from two threads and launch them using the same GPU context.

right schedule. Thus, this problem can be modelled as a $F3|prmu|Cmax$, where these commands (*jobs*) are scheduled in the three *machines* to minimize the makespan.

In a more general case, the application run by each thread may need to execute several tasks in order. Typically, these tasks are launched asynchronously, and precedence relationships are enforced using events or by launching them in the same stream. Consequently, at any given time, some of these tasks could be available to the host proxy but only the first one, for each thread, ready to be executed. Therefore, the problem can still be modelled as $F3|prmu|Cmax$ by considering only the first ready task of each thread. Nevertheless, if the task precedence graph of each thread is known in advance, the problem may be modelled as $F3|prmu, prec|Cmax$ and a more precise solution could be found.

It can be proved that $F3||Cmax$ is strongly NP-hard. In fact, for a N tasks flow shop problem, there are $N!$ different permutations, thus many heuristics have been presented to solve efficiently this problem [15]. These heuristics can be classified as *constructive* or *improvement* depending on if they compute the schedule from scratch or by improving a previous solution. In this paper we are interested on real time scheduling, thus we will consider only lightweight algorithms. Nevertheless, for comparison purposes, we have also included a time consuming algorithm to obtain a near optimum schedule. Next subsections present these heuristics.

A. Modified Johnson's rule

The $F2||Cmax$ problem was one of the first problems to be analyzed [22]. With Johnson's rule an optimal schedule that minimizes the makespan can be easily computed. First, two sets are built: Set I contains all jobs with $p_{1j} < p_{2j}$, and Set II the remaining jobs. Jobs in Set I go first, in increasing order of p_{1j}, and the jobs in Set II go in decreasing order of p_{2j}. This schedule is optimal for $F2|prmu|Cmax$ and can be extended to $F3$ problems by considering Set I includes jobs with $p_{1j} + p_{2j} < p_{2j} + p_{3j}$, and Set II the remaining jobs. Therefore, tasks with short HtD and K commands are scheduled first. The solution found using this heuristic is not optimal, but it can be computed very fast. Henceforth we will refer to this heuristic as MJR (Modified Johnson's Rule).

B. Slope index

Other heuristics that take into account an arbitrary number of machines, like for example the Slope heuristic [23], have been proposed in the literature to find quasi-optimal schedules. This heuristic gives priority to the tasks having the strongest tendency to progress from short times to long times in the sequence of processes. In our GPU problem this means that tasks with short HtD times and long DtH times are prioritized over tasks with long HtD times and short DtH times. This priority is established by computing a slope index A_j for job j as $A_j = -\sum_{i=1}^{m}(m - (2i - 1))p_{ij}$, where m is the number of machines, and the jobs are sequenced in decreasing order of the slope index. We will refer to this heuristic as SI (Slope Index).

C. Mixed integer programming

Another form of solving this problem is using Mixed Integer Programming (MIP) [24]. To do so a number of variables must be defined (see Figure 4 for a graphical representation):

- Decision variable x_{jk}, equal to 1 if job j is the kth job in the sequence and 0 otherwise.
- Auxiliary variable I_{ik}, the idle time on machine i between the processing of the kth and $(k+1)$th jobs in the sequence.
- Auxiliary variable W_{ik}, the waiting time of the kth job in the sequence between machines i and

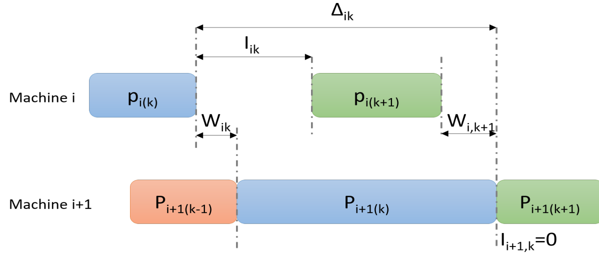

Fig. 4. Graphical representation of variables used in the definition of the MIP problem. Blue boxes correspond to the duration of kth job in machines i and $i+1$, green boxes correspond to $(k+1)$th job and orange box corresponds to $(k-1)$th job. Note in this particular example $I_{i+1,k} = 0$ because $(k+1)$th job in machine $i+1$ starts right after kth job.

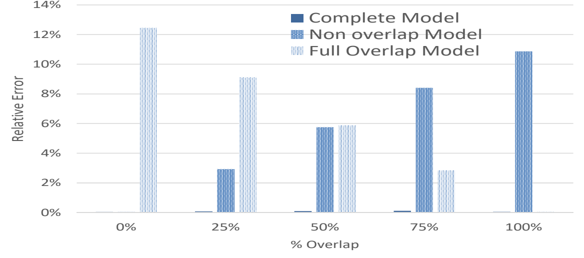

Fig. 5. Prediction time relative error for bidirectional transfers with varying degrees of overlapping in a Titan X GPU card. Three prediction models are considered: non-overlapped transfers, full overlapped transfers, and a complete model that adapts to the overlapping degree.

$i+1$.

- The difference between the time the job in the $(k+1)$th position starts on machine $i+1$ and the time the job in the kth position completes its processing on machine i is denoted by $\Delta_{ik} = I_{ik} + \sum_{j=1}^{n} x_{j,k+1} p_{ij} + W_{i,k+1} = W_{ik} + \sum_{j=1}^{n} x_{jk} p_{i+1,j} + I_{i+1,k}$

Minimizing the makespan is equivalent to minimizing the total idle time on the last machine, that is, $\sum_{i=1}^{m-1} p_{i(1)} + \sum_{j=1}^{n-1} I_{mj}$, where $p_{i(1)}$ is the processing time on machine i of the job in the first position of the sequence. The first term in the equation is the time the first job reaches the last machine (i.e, it completes its DtH command), and the second term is the sum of idle times between the jobs in the last machine.

Using the identity $p_{i(k)} = \sum_{j=1}^{n} x_{jk} p_{ij}$, the MIP problem can be formulated as

$$
\begin{array}{lll}
\texttt{minimise} & \sum_{i=1}^{m-1} \sum_{j=1}^{n} x_{j1} p_{ij} \quad + & \sum_{j=1}^{n-1} I_{mj} \\
\texttt{subject to} & \sum_{j=1}^{n} x_{jk} = 1 & k = 1, \ldots, n \\
& \sum_{k=1}^{n} x_{jk} = 1 & j = 1, \ldots, n, \\
& I_{ik} + \sum_{j=1}^{n} x_{j,k+1} p_{ij} \ldots & k = 1, \ldots, n-1 \\
& \quad + W_{i,k+1} - W_{ik} \ldots & i = 1, \ldots, m-1, \\
& \quad - \sum_{j=1}^{n} x_{jk} p_{i+1,j} \ldots & \\
& \quad - I_{i+1,k} = 0 & \\
& W_{i1} = 0 & i = 1, \ldots, m-1, \\
& I_{1k} = 0 & k = 1, \ldots, n-1.
\end{array} \quad (1)
$$

Both idle and waiting times are non-negative continuous variables, while the decision variable is integer (in fact it is binary). This minimization problem is NP-hard and some heuristic must be used to search the solution space. This can be very time consuming, making it impractical in a real time application.

D. Single Queue

Previous approaches assume all processing times are fixed but in fact they depend on the commands that are currently being processed. More precisely, if both a HtD and a DtH command are executing at the same time, their processing times will change [18]. This is due to the fact that, although the PCIe bus is bidirectional and modern GPUs have two DMA engines, the memory system is shared between both copy operations and the bandwidth of the HtD and DtH commands is reduced. Therefore, these previous heuristics will likely fail to obtain an optimum schedule. Figure 5 shows the prediction time relative errors incurred when two bidirectional transfers, with varying degrees of overlapping, take place in a Titan X NVIDIA card. Three prediction models are considered; a non overlap model that do not take into account the effect of another transfer in the opposite direction, a full overlap model that is not able to compute the overlapping intervals (and thus it always considers there is a full overlap between both transfers), and a complete model that adapt the prediction to the overlapping intervals. The first two models fail to predict well some cases, with errors above 10%, while the relative error of the complete model is always below 1%.

In [18], a GPU tasks execution model and a heuristic based on that model were presented. That model uses an estimation of the HtD, K and DtH commands of each task to simulate the execution of a permutation of the tasks. HtD and DtH times are updated online to reflect the effect of overlapping transfers, achieving an average simulation error below 1.5%. Nevertheless, to obtain a correct estimation, Hyper-Q must be restrained. Hyper-Q is a feature present in NVIDIA GPUs since Kepler architecture. It provides hardware managed queues that can schedule several commands, but it can also reorder the execution of these commands. Any change in the original order performed by Hyper-Q could introduce a significant error in the simulation, thus Hyper-Q is disabled by forcing a single hardware managed queue.

The heuristic presented in that work uses the execution model to perform an online simulation of the execution to choose the tasks that better overlap in each instant. The reasoning behind the heuristic implemented on that work is similar to Johnson's Rule and the Slope Index, but using the tasks execution model to select, in an iterative manner, the task that better adapt to the current tasks commands. We will refer to this heuristic as SQ (Single Queue).

E. NEH heuristic with a GPU tasks execution model

A new algorithm, based on an existing heuristic (NEH, [25]) and the execution model of [18], is proposed in this work. NEH is regarded as one of the best scheduling heuristics for the flow shop problem. It is a constructive heuristic that iterates to compute a solution:

1. For each task compute its total processing time and sort them in non-increasing order.
2. Pick the first two tasks and find the best sequence by computing the makespan, using the execution model, for the two possible sequences.
3. For each of the following tasks, $i = 3, \ldots, n$, find the best schedule by placing it in all the possible i positions in the sequence of tasks that are already scheduled and computing the makespan using the execution model.

Most heuristics try to schedule first the *best*, i.e. shortest, tasks, but this heuristic starts probing the *worst* (longest) tasks and accommodates the remaining tasks to minimize the makespan.

The only drawback is the makespan must be computed $[n(n+1)/2] - 1$ times, of which n are complete sequences and the rest are partial schedules, but it can consistently obtain better results than other heuristics [26]. We will refer to this heuristic as NEH-GPU (NEH heuristic with GPU tasks execution model).

IV. Experiments

In this section the five heuristics presented in previous section will be evaluated in a demanding multithreaded scenario (e.g. heterogeneous computing server) where several threads running applications (workers) offload one or several tasks onto a device. All workers send task information regarding CUDA API calls to a buffer that is constantly polled by a host proxy thread. This thread is in charge of reordering the set of tasks found in the buffer using one of the five heuristics, and submitting the corresponding commands to the device. We consider three scenarios with 4, 8 and 16 workers to reflect different workloads.

TABLE II

Tasks used in the experiments. They are classified as Dominant Kernel (DK) or Dominant Transfer (DT) depending on the duration of the HtD, K and DtH commands. *CONV* task can be DK (CONV1) or DT (CONV2) depending on the input parameters.

Kernel	Source	Description	Dominance
MM	CUDA SDK	Matrix Multiplication	DK
BS	CUDA SDK	Black Scholes	DK
PF	Rodinia	Path Finder	DK
PAF	Rodinia	Particle Filter	DK
CONV	CUDA SDK	Separable Convolution	DK/DT
VA	CUDA SDK	Vector Addition	DT
TM	CUDA SDK	Matrix Transposition	DT
FWT	CUDA SDK	Fast Walsh Transform	DT

All the experiments have been conducted using a set of real kernels obtained from the CUDA and Rodinia SDK like in [18]. Tasks with different transfer and kernel processing times have been selected (see Table II). These tasks can be classified as dominant transfer (DT) or dominant kernel (DK) depending on whether the transfer time dominates over the kernel time, or the other way round. These tasks are grouped in benchmarks with a different composition of dominant transfer or dominant kernel tasks (see Table III). Thus, benchmark BK0 has no dominant kernel tasks, BK25 has a 25% of dominant kernel tasks, and so on.

To run the experiments, the host proxy thread creates a single CUDA context and associates each task to a different stream that can be launched using the ordering proposed by each heuristic. In Kepler and later architectures the Hyper-Q feature must be restrained by using some synchronization method to guarantee that the order calculated by the proxy will not be modified during task execution. This is attained by synchronizing commands within a stream on some specific event (that can be recorded on a different stream) by using *cudaStreamWaitEvent*, and reducing the number of hardware managed queues to only one (by setting the environment variable CUDA_DEVICE_MAX_CONNECTIONS to 1).

All experiments have been executed on three GPU architectures, a K20c (Kepler), a GTX980 (Maxwell) and a Titan X (Pascal). Every heuristic has been tried for each benchmark, in every architecture, fifteen times to record a mean makespan (total execution time). For comparison purposes, the best makespan obtained for each benchmark in each architecture has been recorded to establish a minimum reference value. Then, for every experiment, the makespan obtained by each heuristic is compared with this reference value to assess the proximity to this minimum value. This proximity is computed by dividing both values to obtain a percentage, thus a result close to 100% means the heuristic obtained a good solution.

The first experiment consists of four threads, each launching one task. The five benchmarks (BK0, BK25, and so on) have been tested in the three GPU architectures, and the proximity to the best value is shown in Table IV. The two simplest heuristics, MJR and SI, do not obtain very good results, especially when there is a balanced mix of DK and DT tasks, but they have a negligible overhead (less than $1\mu s$). The third heuristic, MIP, uses integer programming and obtains much better results but its cost makes it impractical in a real time scheduler. In fact, it needs thousands of iterations to converge for the most demanding workloads thus, unlike the other heuristics, it is executed offline to record the execution order and, later, this order is used by the proxy to measure the makespan. It obtains good results for the K20c but not as good in the GTX980 and Titan X because, although it explores many permutations until it reaches a solution, it does not consider the effect of overlapping transfers. The fourth heuristic, SQ, was developed specifically for GPU tasks

TABLE III

Tasks composition for each benchmark in experiments employing 4, 8 and 16 tasks.

# Tasks	4	8	16
BK0	VA: 1, TM: 1, FWT: 2	VA: 2, TM: 2, FWT: 2, CONV2: 2	FWT: 1, TM: 9, VA: 5, CONV2: 1
BK25	MM: 1, VA: 1, TM: 1, FWT:1	MM: 2, BS: 1, VA: 2, TM: 2, FWT: 1	MM: 2, BS: 1, CONV1: 1, TM: 6, VA: 5
BK50	MM: 1, BS: 1 VA: 1, TM: 1	MM: 1, BS: 1, PF: 1, CONV1: 1, TM: 2, VA: 1, FWT: 1	MM: 6, BS: 1, CONV1: 1, FWT: 1, TM: 5, VA: 2
BK75	MM: 1, BS: 1 PF: 1, VA: 1	MM: 1, BS: 1, PF: 1, PAF: 1, CONV1: 1, VA: 1, TM: 1, FWT: 1	MM: 10, BS: 1, CONV1: 1, TM: 2, FWT: 1, VA: 1
BK100	MM: 2, BS: 1, CONV1: 1	MM: 2, BS: 3, CONV1: 3	MM: 14, BS: 1, CONV1: 1

concurrent execution and obtains better results than the simpler heuristics at an acceptable cost (less than $1ms$), but there are several cases where it obtains results around 80%-90% of the best solution. The last heuristic, NEH-GPU, is a combination of the best *constructive* heuristic with a GPU tasks execution model; this new heuristic obtains the best results, above 97.35% for every benchmark and GPU, with an overhead similar to the SQ heuristic.

TABLE IV

Proximity to the best value obtained by each heuristic with four threads launching tasks.

	BK	MJR	SI	MIP	SQ	NEH-GPU
K20c	BK0	99.73%	96.09%	**100.00%**	93.85%	99.53%
	BK25	99.90%	95.69%	98.30%	**100.00%**	97.35%
	BK50	76.08%	85.73%	99.75%	99.61%	**100.00%**
	BK75	92.58%	90.95%	**100.00%**	81.86%	98.20%
	BK100	95.66%	95.71%	**100.00%**	99.96%	99.84%
GTX980	BK0	97.41%	96.56%	97.45%	92.59%	**100.00%**
	BK25	99.52%	99.40%	99.77%	95.02%	**100.00%**
	BK50	84.19%	98.33%	90.86%	98.35%	**100.00%**
	BK75	88.63%	99.31%	99.35%	99.95%	**100.00%**
	BK100	95.96%	94.04%	96.04%	95.90%	**100.00%**
TitanX	BK0	**100.00%**	91.32%	94.91%	88.04%	99.80%
	BK25	**100.00%**	99.87%	99.78%	86.77%	99.85%
	BK50	94.07%	98.14%	94.39%	98.36%	**100.00%**
	BK75	72.61%	87.74%	**100.00%**	87.74%	99.47%
	BK100	98.47%	93.78%	97.28%	97.21%	**100.00%**

The second experiment tries to assess the scalability of each heuristic while the number of host threads, simultaneously launching tasks, is incremented from four to eight and to sixteen. That is, the five benchmarks (BK0, BK25 and so on) have been executed in each architecture using 4, 8 and 16 threads. For each benchmark and number of threads the minimum makespan has been recorded and the proximity values to the minimum of all benchmarks have been averaged to obtain mean proximity values for each architecture and number of threads. Figure 6 shows these average results. It can be seen than NEH-GPU obtains the best results, always above 98% of the best solution. Only MIP obtains comparable results, and the rest of heuristics are always below, with many results in the range of 90% for MJR and SI. It should also be noticed than NEH-GPU results are more stable and neither the architecture nor the benchmark type nor the number of tasks worsen the results.

The third experiment is aimed to reproduce the more general case where each thread may execute an entire application. An application typically alternates GPU tasks with CPU computation, and there is a precedence relationship among the GPU tasks.

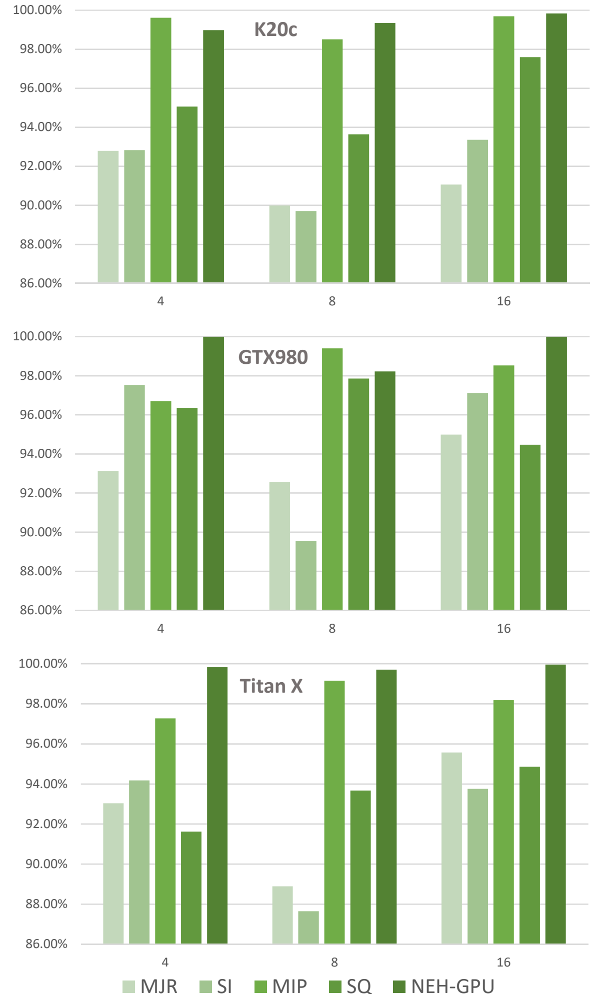

Fig. 6. Proximity to the best value obtained with 4, 8 and 16 host threads launching tasks.

That is, any task cannot be launched before the previous task from that thread has finished. Then, at any given time, it is possible to launch only the first ready task of each thread and the host proxy thread may schedule only this batch of ready tasks. Once this batch has been scheduled, a new batch with the next ready tasks could be scheduled. The ordering of this new batch could be different of the previous one, thus new streams must be used to enforce the desired ordering. Nevertheless, any task of this new batch must fulfill its precedence relationship, that is, its HtD command must wait for the finalization of the DtH command of the corresponding task in

the previous batch. This can be achieved by using CUDA events associated to each command.

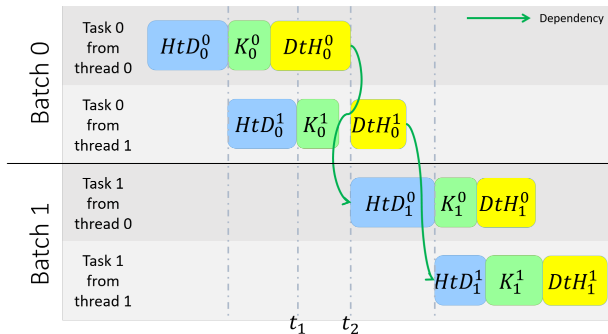

Fig. 7. Dependency between tasks launched by the same thread. HtD_1^0 could start at t_1, right after the end of HtD_0^1, but the dependency between task 1 and task 0 delays the beginning to t_2, after the end of DtH_0^0

Figure 7 shows an example where two threads launch two tasks. Commands for each task are annotated using a subindex for the task index and a superindex for the thread index. At the beginning only task 0 of each thread can be executed, therefore the proxy can only schedule a batch with these two tasks (batch 0). Task 1 of both threads (batch 1) can be scheduled later and, if there were no dependencies with tasks of batch 0, they could start at time t_1, right after the last Htd command of batch 0, but the dependency between tasks of the same thread delays HtD_1^0 to time t_2, right after the end of DtH_0^0. In this example, HtD_1^1 starts after HtD_1^0 because its dependency with respect to DtH_0^1 was already fulfilled.

Heuristics MJR, SI and MIP do not take into account the precedence constraints, thus they schedule each batch separately. Nevertheless, the proxy host does take them into account and batches are executed with the necessary events to comply with the precedence constraints. SQ and $NEH-GPU$ use a GPU tasks execution model that can consider these constraints, thus they schedule each batch taking into account previous batches.

As in the previous experiment, the five benchmarks have been executed using 4, 8 and 16 threads in the three architectures, but instead of considering only one batch the experiments have been conducted considering 1, 2, 3, and 4 consecutive batches to assess the impact of dependencies between tasks. Figure 8 shows the results obtained. Proximity values for the three architectures have been averaged to obtain three bar plots, one for each number of tasks. As the number of batches grows, most heuristics obtain better results. Surprisingly, MJR and SI obtain better results with respect to SQ when the number of batches increases, notwithstanding they do not take into account the precedence constraints. Nevertheless, $NEH-GPU$ obtains once again the best results, for each number of batches and tasks.

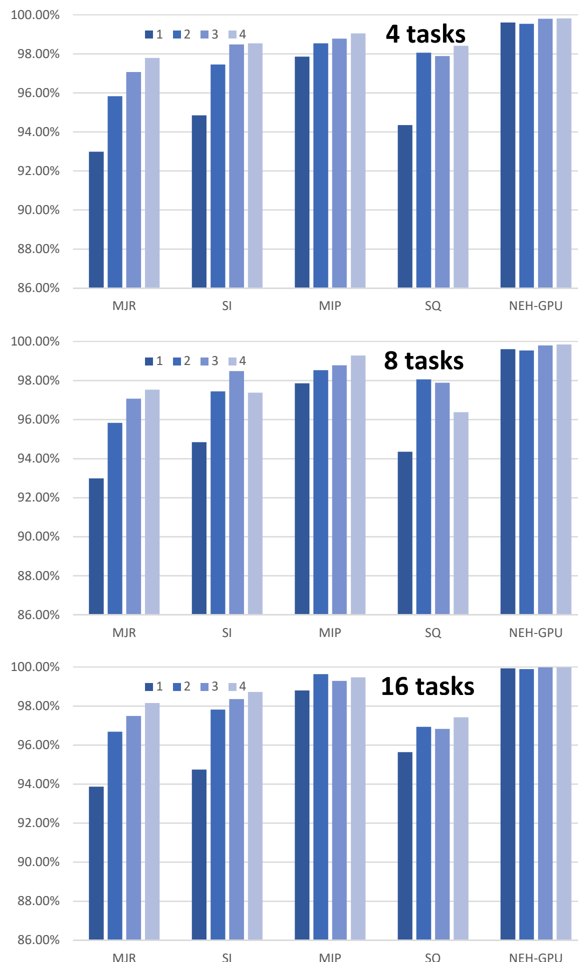

Fig. 8. Proximity to the best value.

V. Related work

Modern GPUs are broadly used in multitask environments, e.g. data centers, where applications running on CPUs offload specific functions to GPUs in order to take advantage of the device performance. This way, it is probable to have several independent tasks ready to run concurrently in a GPU. In this context, several works have been published that try to improve the way tasks are scheduled on GPUs. Thus, [1] presents a scheduler that protects concurrent GPU workloads for interference and supports two priority scheduling policies that takes into account the response time and throughput of scheduled kernels. Another contemporary work [27] proposes two real-time analysis methods with the goal of providing predictability while maximizing processing capabilities. In these early works GPU task cannot be preempted until completion, which can produce high GPU response time. To solve this problem, another work, [3], splits application tasks in smaller jobs that can be assigned to specific SMs according to a timing model. Similarly, [8] proposes Kernelet, a runtime system with dynamic kernel slicing and scheduling techniques to improve GPU utilization. In [9] a support for real-time scheduling is presented. It automatically slices a long-running kernel execution into multiple subkernel launches and splits data trans-

action into multiple chunks at user-level, then inserts preemption points between subkernel launches and memory copy operations at driver-level. Other recent works propose hardware [5] or software [10] solutions supporting round-robin or priority-based techniques to implement both soft-realtime and fair scheduling policies. Most previous works are focused on kernel execution and they do not take into account the transfer of data required to compute those kernels. In many cases, the time taken by these transfer is not negligible and can affect the result of the applied scheduling policy. In addition, unlike in our proposal, current approaches typically require to modify the original kernels. Nevertheless, our scheduling theory framework can be used as well in these approaches to reformulate them using the notation presented in Section II.

VI. Conclusions

This paper has discussed how to apply scheduling theory concepts to the problem of task scheduling on GPUs. It has been shown that the concurrent execution of GPU tasks using CUDA streams can be modelled as a flow shop problem. Under this model, execution of a GPU task involves a Host to Device command, a Kernel command, and a Device to Host command, that are processed by the DMA engines and the Grid Management Unit. The most important advantage of this approach is that an objective function can be defined and an appropriate solution from the existing literature can be selected.

As a practical example, it has been studied the $F3|prmu|Cmax$ problem that arises when several threads launch independent kernels that can be computed using the same GPU context. Several solutions found in both the scheduling and the GPU literature have been presented. Besides, a new heuristic called NEH-GPU, that combines an existing heuristic with a GPU tasks execution model, has been developed. This heuristic can also be included as runtime support because it does not modify the original kernels and it has a low overhead.

Several experiments have been conducted to show the suitability and robustness of this new approach. Three different GPU architectures, Kepler, Maxwell and Pascal, have been evaluated. Several real kernels from the CUDA and Rodinia SDK have been selected to form five benchmarks that reflect different composition of dominant transfer and dominant kernel tasks. Furthermore, the number of tasks and the number of consecutive batches have been varied to assess the scalability of the heuristics. In all of them NEH-GPU has obtained results very close to the best makespan obtained by any heuristic.

VII. Acknowledgements

This work has been funded by project TIN2016-80920R (Spanish Ministry of Science and Technology) and University of Malaga (Campus de Excelencia Internacional Andalucía Tech). We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Titan X Pascal GPU used for this research.

Referencias

[1] Shinpei Kato, Karthik Lakshmanan, Ragunathan Rajkumar, and Yutaka Ishikawa, "Timegraph: Gpu scheduling for real-time multi-tasking environments," in *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, Berkeley, CA, USA, 2011, USENIXATC'11, pp. 2–2, USENIX Association.

[2] C. Basaran and K. D. Kang, "Supporting preemptive task executions and memory copies in gpgpus," in *2012 24th Euromicro Conference on Real-Time Systems*, July 2012, pp. 287–296.

[3] Haeseung Lee and Mohammad Abdullah Al Faruque, "Gpu-evr: Run-time event based real-time scheduling framework on gpgpu platform," in *Proceedings of the Conference on Design, Automation & Test in Europe*, 3001 Leuven, Belgium, Belgium, 2014, DATE '14, pp. 220:1–220:6, European Design and Automation Association.

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

[5] Jason Jong Kyu Park, Yongjun Park, and Scott 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*, New York, NY, USA, 2015, ASPLOS '15, pp. 593–606, ACM.

[6] Qiumin Xu, Hyeran Jeon, Keunsoo Kim, Won Woo Ro, and Murali Annavaram, "Warped-slicer: Efficient intra-sm slicing through dynamic resource partitioning for gpu multiprogramming," in *Proceedings of the 43rd International Symposium on Computer Architecture*, Piscataway, NJ, USA, 2016, ISCA '16, pp. 230–242, IEEE Press.

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

[8] J. Zhong and B. He, "Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1522–1532, June 2014.

[9] H. Zhou, G. Tong, and C. Liu, "Gpes: a preemptive execution system for gpgpu computing," in *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2015, pp. 87–97.

[10] Guoyang Chen, Yue Zhao, Xipeng Shen, and Huiyang Zhou, "Effisha: A software framework for enabling efficient preemptive scheduling of gpu," in *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, New York, NY, USA, 2017, PPoPP '17, pp. 3–16, ACM.

[11] NVIDIA, "CUDA Programming Guide," September 2015.

[12] Khronos Group, "OpenCL 2.0 API Specification," October 2014.

[13] NVIDIA, "CUDA Samples," September 2015.

[14] M. R. Garey, D. S. Johnson, and Ravi Sethi, "The Complexity of Flowshop and Jobshop Scheduling," *Mathematics of Operations Research*, vol. 1, no. 2, pp. 117–129, 1976.

[15] Rubén Ruiz and Concepción Maroto, "A comprehensive review and evaluation of permutation flowshop heuristics," *European Journal of Operational Research*, vol. 165, no. 2, pp. 479–494, sep 2005.

[16] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G.Rinnooy Kan, "Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey," *Annals of Discrete Mathematics*, vol. 5, no. C, pp. 287–326, 1979.

[17] Michael Pinedo, *Scheduling. Theory, algorithms, and systems. With CD-ROM. 3rd ed*, 01 2008.

[18] A.J. J. Lázaro-Muñoz, J.M. González-Linares, J. Gómez-Luna, and N. Guil, "A tasks reordering model to reduce

transfers overhead on GPUs," *Journal of Parallel and Distributed Computing*, vol. 109, pp. 258–271, nov 2017.

[19] Jm Framinan, Jnd Gupta, and R Leisten, "A review and classification of heuristics for permutation flow-shop scheduling with makespan objective," *Journal of the Operational Research Society*, vol. 55, pp. 1243–1255, 2004.

[20] Ali Allahverdi, C.T. Ng, T.C.E. Cheng, and Mikhail Y Kovalyov, "A survey of scheduling problems with setup times or costs," *European Journal of Operational Research*, vol. 187, no. 3, pp. 985–1032, jun 2008.

[21] NVIDIA, "Cuda multi-process service," March 2015.

[22] S.M. Johnson, "Optimal Two- and Three-Stage Production Schedules With Set-up Time Included," *Naval Research Logistics Quarterly*, vol. 1, pp. 61–68, 1954.

[23] D. S. Palmer, "Sequencing Jobs Through a Multi-Stage Process in the Minimum Total TimeA Quick Method of Obtaining a Near Optimum," *Journal of the Operational Research Society*, vol. 16, no. 1, pp. 101–107, mar 1965.

[24] Harvey M. Wagner, "An integer linear-programming model for machine scheduling," *Naval Research Logistics Quarterly*, vol. 6, no. 2, pp. 131–140, jun 1959.

[25] Muhammad Nawaz, E. Emory Enscore, and Inyong Ham, "A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem," *Omega*, vol. 11, no. 1, pp. 91–95, 1983.

[26] E Taillard, "Some efficient heuristic methods for the flow shop sequencing problem," *European Journal of Operational Research*, vol. 47, pp. 65–74, 1990.

[27] Glenn A. Elliott and James H. Anderson, "Globally scheduled real-time multiprocessor systems with gpus," *Real-Time Syst.*, vol. 48, no. 1, pp. 34–74, Jan. 2012.