This is the peer reviewed version of the following article: López-Albelda B, Lázaro-Muñoz AJ, González-Linares JM, Guil N. Heuristics for concurrent task scheduling onGPUs. Concurrency Computat Pract Exper. 2020;32:e5571. https://doi.org/10.1002/cpe.5571, which has been published in final form at https://doi.org/10.1002/cpe.5571. This article may be used for non-commercial purposes in accordance with Wiley Terms and Conditions for Use of Self-Archived Versions. This article may not be enhanced, enriched or otherwise transformed into a derivative work, without express permission from Wiley or by statutory rights under applicable legislation. Copyright notices must not be removed, obscured or modified. The article must be linked to Wiley's version of record on Wiley Online Library and any embedding, framing or otherwise making available the article or pages thereof by third parties from platforms, services and websites other than Wiley Online Library must be prohibited

Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

ARTICLE TYPE

Heuristics for concurrent task scheduling on GPUs

B. López-Albelda | A.J. Lázaro-Muñoz | J.M. González-Linares | N. Guil*

[1]Computer Architecture Department, University of Malaga, Spain

Correspondence
*Nicolas Guil. Email: nguil@uma.es

Present Address
2.2.48, Complejo Tecnológico (E.T.S.I. Informática), Bulevar Louis Pasteur, 35. 29071 Málaga, Spain

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 runtime 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 heuristics 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 ones. It leverages on a precise GPU command execution model for both computation and data transfers to carry out more advantageous scheduling decisions. A comprehensive evaluation, showing the suitability and robustness of this new approach, is conducted in three different NVIDIA architectures (Kepler, Maxwell and Pascal). Results confirms the proposed heuristic achieves the best results in more than 90% of the experiments. Furthermore, a comparison has been made with MPS (Multi-Process Service,[1]), the NVIDIA API that deals with the execution of concurrent tasks, which shows that our solution obtains speed-ups ranging from 1.15 to 1.20.

KEYWORDS:
Task Scheduling, Flow Shop, GPU, MPS

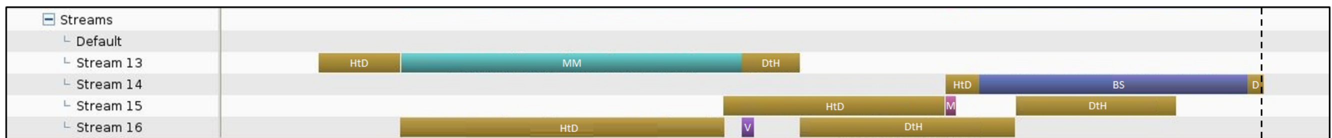
1 | INTRODUCTION

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[2,3]. The most recent ones propose hardware[4,5,6] or software[7,8,9] 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 [10] and OpenCL [11] 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.

Permutation I



Reduction

Permutation II

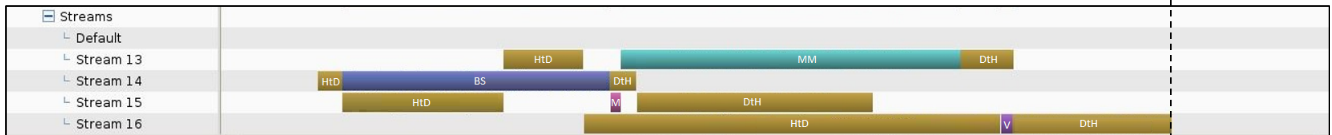


FIGURE 1 Profile views of 4 independent tasks launched on a Nvidia K20c card. Matrix Multiplication (MM), Black Scholes (BS), Matrix Transposition (M) and Vector Addition (V) are launched by streams 13, 14, 15, and 16 respectively. Brown boxes represent transfers between host and device for each task; more precisely, transfers before kernel execution are host to device (*HtD*), and transfers after kernel execution are device to host (*DtH*). Top view corresponds to the ordering selected by an scheduling heuristic, while bottom view shows the ordering selected by another heuristic. The makespan reduction is solely due to the launching tasks different orderings.

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 [12] and they are scheduled employing a different stream per task. More precisely, the tasks are Matrix Multiplication (Stream 13), Black Scholes (Stream 14), Matrix Transposition (Stream 15) and Vector Addition (Stream 16). 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 around 10% 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 [13]. 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 [14].

Previous works that have studied tasks scheduling on GPUs either ignore transfers overlapping [9] or propose ad-hoc heuristics [15]. In this paper we expose how to apply methods introduced 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. Thus, many heuristic already developed for this kind of problem can be tested on GPUs. As a result of our study we also propose a new heuristic, called NEH-GPU, implemented in a run-time scheduler that significantly reduces the makespan of a workload and, consequently, increases 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. Thus, the main contributions of this paper are the following:

- Modeling concurrent execution of tasks on GPUs as a flow shop problem.
- Identifying flow shop heuristics that can be employed for concurrent task scheduling at run-time.
- Developing a precise execution model for data transfer commands using either pageable or pinned memory.
- Proposing a new heuristic that leverages on the execution model and scheduling theory.

We have carried out several experiments which confirm that the proposed heuristics obtains better scheduling results not only when compared to previous flow shop heuristics but also with respect to MPS (Multi-Process Service, [1]), the NVIDIA API currently used to execute concurrent applications on GPUs. We also show that the conclusions achieved in our studies can be used to increase the performance of tasks scheduling by classifying incoming tasks as compute or memory bound and shuffling them in a balanced way.

The rest of the paper is organized as follows. First, scheduling theory is reviewed in Section 2, with a focus on GPU tasks scheduling using the flow shop problem. Next, in Section 3, 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. In Section 4 a new model for data transfer, using either pinned or pageable memory, is proposed. Then, several experiments that show the suitability and robustness of this new approach are conducted in Section 5 . This section also shows the advantages of shuffling the execution of compute and memory bound tasks in order to improve the scheduling efficiency. Finally, conclusions are drawn.

2 | 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, i.e., resources that can complete the work. 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). Each of these fields can accept one or more values and, below, the most relevant ones in a GPU tasks launching context are explained.

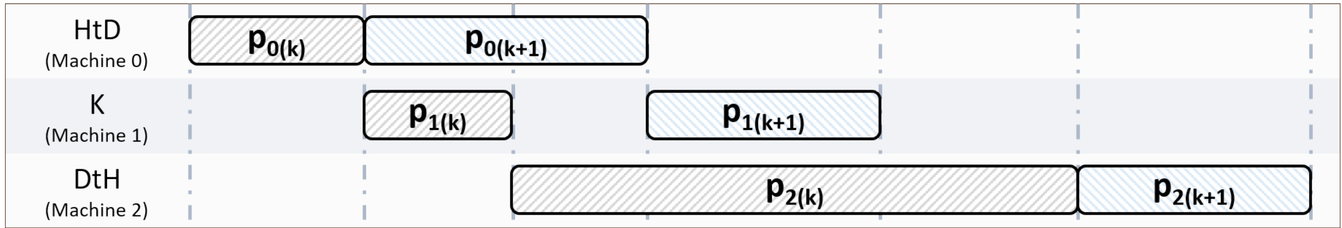


FIGURE 2 GPU tasks launching as a 3-machine flow shop problem. *HtD* commands are executed by machine 0, K commands by machine 1 and *DtH* commands by machine 2. Commands from kth job precede commands from $(k+1)$th job in all machines (permutation flow shop). $p_{i(j)}$ corresponds to the processing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

Launching a task (job in scheduling terminology) in a NVIDIA GPU typically involves executing in order three types of commands: Host to Device transfer (*HtD*), kernel command (*K*), and Device to Host transfer (*DtH*). There can be zero, one or more of each of these commands; in this case we consider them as a single command that encompasses all of them. Transfer commands are executed in DMA engines and kernel commands in the GPU itself (through the Grid Management Unit). When kernels commands are launched in the GPU, hardware resources (memory, registers, etc.) are assigned to each kernel. If kernels do not exhaust any of these resources, Concurrent Kernel Execution can be possible. Nevertheless, 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.

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 set of identical machines in parallel that can execute the same job type. This configuration 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[1] 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 can be 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 [15]. 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 1 resumes these values with their meaning and a short description. All or some of these parameters can be combined to express different real world situations and to select an appropriate solution from the existing literature [18,19]. 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

[1] HyperQ is a feature of modern GPUs since Kepler architecture, where several hardware managed queues can schedule different kernel commands, but can also change the execution order of these commands.

TABLE 1 Scheduling Theory notation summary

Field	Value	Meaning	Description
α	Fm	Flow Shop with m machines	Each job is processed following a given order
	FFc	Flexible 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 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

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.

3 | 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, [1]), a proxy thread could be used to collect all tasks, and launch them using the same GPU context to improve concurrency among tasks.

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 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.

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 [14]. 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 except for a time consuming heuristic that will be executed offline.

In next subsections, we present several heuristics that can be used in our run-time scheduler. For comparison purposes, we have included some classical heuristics that do not take into account the effect of data transfer overlapping, another heuristic specifically developed for efficient GPU tasks execution, and a new heuristic that combines a classical scheduling heuristic with a GPU tasks execution model. We considered another two classical heuristics, the Modified Johnson's rule (MJR) [20] and Mixed Integer Programming (MIP) [21], that were finally discarded because their results were similar to the other classical heuristics and, in the case of MIP, its computational cost rendered it unusable in a real time application.

3.1 | Slope index

Some heuristics that take into account an arbitrary number of machines, like for example the Slope heuristic [22], have been proposed in the literature to find quasi-optimal schedules for the flow shop problem. 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 (3 in our GPU problem), and the jobs are sequenced in decreasing order of the slope index. We will refer to this heuristic as SI (Slope Index).

3.2 | NEH heuristic

Nawaz, Enscore and Ham developed in [23] an algorithm for solving the flow shop problem that is regarded as one of the best scheduling heuristics. This algorithm is typically noted as NEH and 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 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.

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 [24].

3.3 | Single Queue

Previous approach 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 [15]. 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. A transfer model that do not consider overlapping can incur in errors above 10%. Therefore, these previous heuristics will likely fail to obtain an optimum schedule.

In [15], a GPU tasks execution model and a heuristic based on an overlapping transfers 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. 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 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. Tasks with short *HtD* commands are selected first, while next tasks are chosen looking for the best fit between the remaining K commands of the previous selected tasks and the *HtD* command of the new task, and between the remaining *DtH* commands of the previously selected tasks and the K command of the new task. We will refer to this heuristic as SQ (Single Queue).

3.4 | NEH heuristic with a GPU tasks execution model

In the original NEH heuristic, the makespan is computed using a fixed duration for each command, but in a GPU the duration of two overlapping transfer commands can vary significantly with respect to its standalone execution, as it will be shown in Section 4. Therefore, the predicted makespan can have a large error that may lead to an erroneous order selection. In this work we propose to combine the NEH method with a GPU tasks execution model that can predict more accurately the makespan. We will refer to this heuristic as NEH-GPU (NEH heuristic with GPU tasks execution model).

To illustrate the advantages of using the execution model, we discuss again Figure 1 (in Introduction section). This picture compares the results obtained by the original NEH (Permutation I) with NEH-GPU (Permutation II) when scheduling four independent tasks: matrix multiplication (MM), Black-Scholes (BS), matrix transposition (TM) and vector adition (VA). For each task it is given, in the left side of Table 2, the duration of its *HtD*, *K* and *DtH* commands, and its total makespan if it is launched alone. Also, in the right side of the same table, the partial makespans computed by each heuristic are shown. The two longest taks, 3 and 0, are tried in the first iteration. Each algorithm computes the makespan using the data on the left and, although they predict a different makespan for permutation $3-0$, they select the same permutation, $0-3$, as the one with the best partial makespan. In the second iteration they try all possible permutations inserting task 2 in ordering $0-3$ and predict different makespans that lead them to select different permutations, $0-3-2$ for NEH and $2-0-3$ for NEH-GPU. Finally, in the last iteration, each heuristic inserts the remaining task obtaining different schedules. The NEH heuristic predicts a makespan of 27.32 when scheduling $0-3-2-1$, but the real execution using this permutation takes 29.79. On the other side, NEH-GPU predicts a makespan of 26.58 when scheduling $1-0-3-2$ and the real execution takes 26.97. That is, the GPU model predicts very accurately the real execution time, which leads to a better schedule decision.

4 | DATA TRANSFER MODELING

As explained in previous section, some of the heuristics require to know in advance the execution time of data transfer and kernel commands. In [15] two models are proposed for the calculation of the kernel execution time and the data transfer time. We have adopted the model for kernel

TABLE 2 Computation of 4 independent tasks optimal ordering using NEH and NEH-GPU. Left table show the *HtD*, *K* and *DtH* commands duration of 4 tasks, and right table the tested orderings and predicted makespans using each method.

	Task	*HtD*	K	*DtH*	Total
0	MM	2.52	10.61	1.26	14.39
1	BS	0.73	8.50	0.49	9.72
2	TM	5.03	0.31	4.98	10.32
3	VA	10.04	0.37	4.98	15.39

	NEH		NEH-GPU	
	Ordering	Makespan	Ordering	Makespan
1st	3-0	24.43	3-0	25.32
	0-3	**19.37**	0-3	**19.37**
2nd	2-0-3	24.40	2-0-3	**25.57**
	0-2-3	24.35	0-2-3	26.26
	0-3-2	**24.35**	0-3-2	26.26
3rd	1-0-3-2	31.06	1-2-0-3	**26.58**
	0-1-3-2	32.08	2-1-0-3	31.28
	0-3-1-2	27.47	2-0-1-3	33.04
	0-3-2-1	**27.32**	2-0-3-1	29.30

execution time estimation as it is precise enough. However, the model proposed for data transfers lacks some features for our purposes. More precisely, that model only considers pinned memory data transfers while some GPU tasks use pageable memory transfers that cannot be estimated with that model. In this work we propose a more general model for data transfers that considers both types of data transfers, pinned and pageable.

In Figure 3 the difference between data transfers using pinned or pageable memory is illustrated. A *HtD* or *DtH* pinned memory transfer command can be completed by the DMA without the involvement of the CPU, as it is shown in the right part of the figure. However, pageable memory transfer commands involve a temporary allocation of pinned memory, a data copy from pageable memory to pinned memory, and finally a data transfer to device memory, as illustrated on the left part of Figure 3. Thus, as the behaviour of pageable memory data transfers is very different for that of pinned memory transfer, we have extended the model presented in [15] to include pageable memory transfer commands.

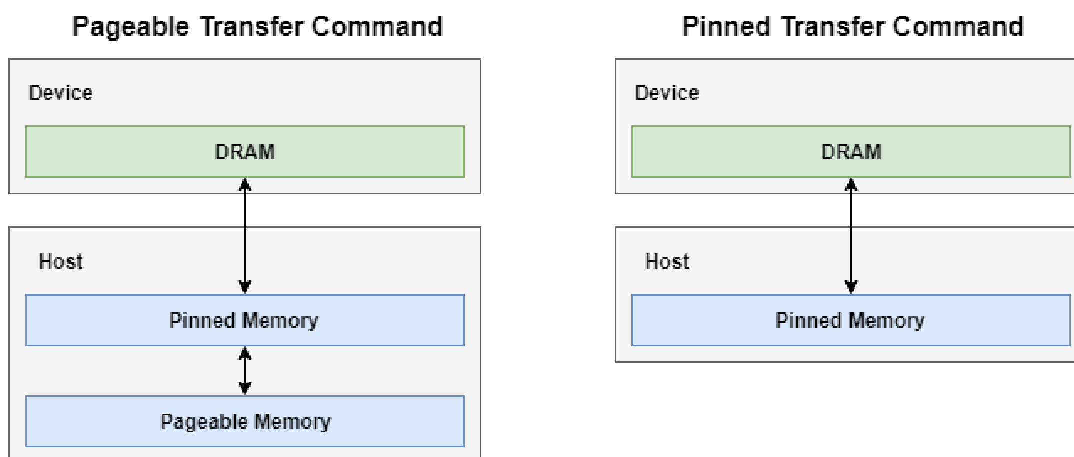


FIGURE 3 Pageable and Pinned Memory Transfer Commands

According to our experiments, the temporary allocation of pinned memory during a pageable memory data transfer typically causes a reduction of the achieved bandwidth with respect to the values obtained by a pinned memory data transfer. Furthermore, concurrent data transfers in opposite directions can affect to the final bandwidth. Therefore, we have conducted several experiments to analyze the behaviour of the different

transfer commands. First of all, we have executed *HtD* and *DtH* transfer commands alone, for both pinned and pageable memory transfers, using different data lengths to calculate the achieved bandwidth. Second, we have launched two transfer commands of the same memory type (pinned or pageable), simultaneously but in opposite directions, in order to study the impact on performance. Third, we have conducted a similar experiment but using a pinned memory transfer in one direction and a pageable memory transfer in the opposite one. Finally, we have also tested the possibility of overlapping two transfer commands in the same direction.

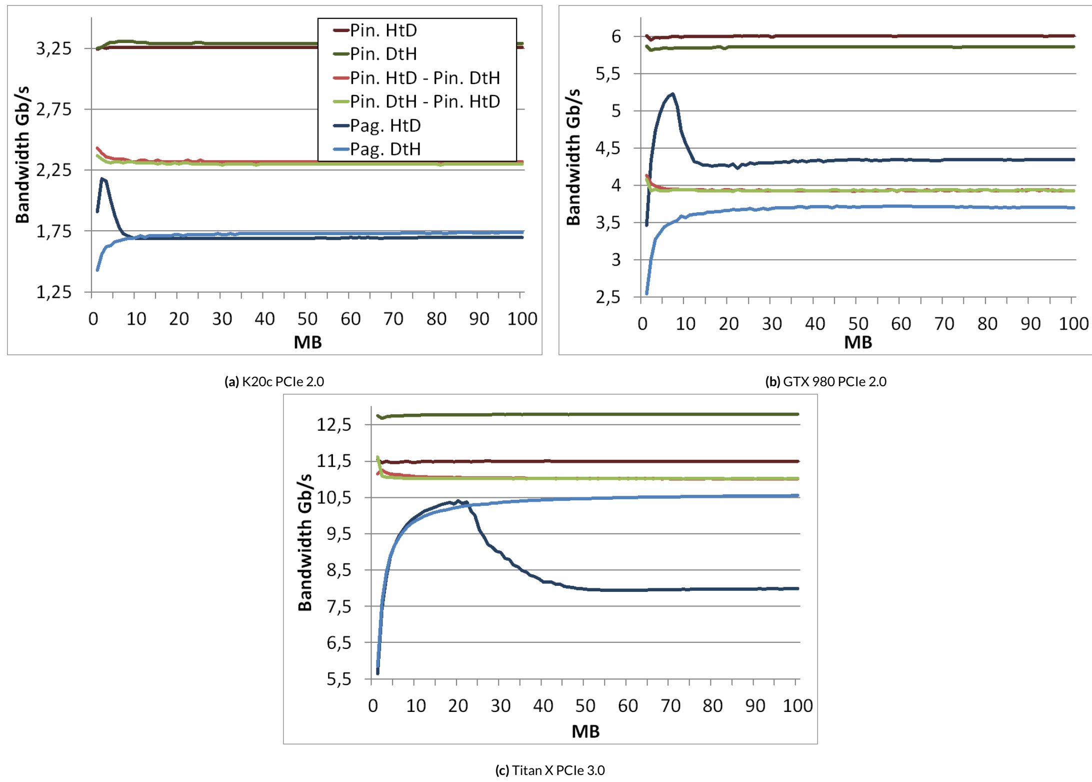
Experiments have been conducted using CUDA 10 on three NVIDIA GPU cards, namely, K20c (with PCIe 2.0), GTX 980 (with PCIe 2.0 too) and Titan X (with PCIe 3.0) with different data sizes for each transfer command, ranging from 1 MB to 512 MB. Every experiment has been repeated fifty times to compute an average bandwidth for each experiment.

In Figure 4 we show the results for the first and second experiment, that is, the bandwidth corresponding to a single transfer in any direction using either pageable or pinned memory. In the figure we can see that the bandwidth of pinned transfer commands, labelled as *Pin. HtD* (dark red) and *Pin. DtH* (dark green), is higher than that achieved by pageable transfer commands, labelled as *Pag. HtD* (dark blue) and *Pag. DtH* (blue). In addition, the bandwidth of pinned memory transfer is very stable for all sizes and directions, while the bandwidth of pageable memory transfers are more variable for small data sizes. We have also carried out experiments where *HtD* and *DtH* pinned memory transfers are concurrently executed. Thus, the red line, labelled as *Pin. HtD - Pin. DtH*, shows the performance of a *HtD* transfer when a *DtH* is running concurrently. Similarly, the green line, labelled as *Pin. DTH - Pin. HtD*, depicts the results for a *DtH* transfer when a *HtD* is also executing. It can be observed that overlapping draws a reduction of performance in the three devices. For example, the *Pin. DtH* transfer has a bandwidth above 12.5 Gb/s in Titan X, but when this transfer command is overlapped with a *HtD* pinned memory transfer in the opposite direction, the bandwidth decreases to 11.0 Gb/s (*Pin. DtH - Pin. HtD* line). We have conducted a similar experiment for pageable memory transfers and confirmed that pageable memory transfers in opposite directions do not overlap.

Figure 5 illustrates the results for the third experiment. Thus, a pair of data transfers in opposite directions are executed concurrently, but one transfer uses pageable memory and the other one employs pinned memory. The *Pag. HtD - Pin. DtH* line shows the performance of a *HtD* pageable memory transfer when a pinned memory transfer is concurrently executed in the opposite direction. The other of possible combinations are also shown in the figure. Once again, pageable memory transfers have a smaller bandwidth than pinned memory transfers, except for the NVIDIA GTX 980 card (top right plot in Figure 5b), where the bandwidth is similar from 10 MB onward. Moreover, a pinned memory transfer overlapped with a pageable memory transfer in the opposite direction has a higher bandwidth than when two pinned memory transfers are overlapped. This shows that a pageable memory transfer affects less than a pinned memory transfer overlapped with a pinned memory transfer. Finally, in the fourth experiment, we have confirmed than two transfers in the same direction never overlaps for neither pageable nor pinned memory.

Previous results have been used to build the following expression for estimating the execution time of a data transfer with a size of k bytes:

$$t_t = L + o + (1-\lambda)kG^1 + (1-\alpha)\lambda kG^2 + \alpha\lambda kG^3 \tag{1}$$



(a) K20c PCIe 2.0

(b) GTX 980 PCIe 2.0

(c) Titan X PCIe 3.0

FIGURE 4 Average bandwidth obtained for K20c, GTX 980 and Titan X GPU cards by pageable and pinned memory transfers. Results show the achieved performance when transfers are either executed alone or overlapped with a similar transfer in an opposite direction

In previous expression L is an upper bound of the latency incurred when sending a message from one endpoint to another endpoint, and o is the length of time that the CPU is involved in registering the DMA request with the controller. Similarly, α is a parameter that is set to 0 when two pinned memory transfers run concurrently in opposite directions and 1 when a pinned memory transfer is executed at the same time than a pageable memory transfer. In addition, G^1 is the inverse of the bandwidth when the transfer takes place alone, G^2 the inverse of the bandwidth when two pinned transfers execute concurrently, and G^3 the inverse of the bandwidth when a pinned memory transfer is executed with a pageable transfer. Finally, λ is the fraction of bytes transferred concurrently. Table 3 shows the average values computed for these parameters in the three NVIDIA GPU cards used in this work. In the case of pageable memory transfer, there is not a G^2 value because two concurrent pageable memory transfers are not possible. In addition, G^1 and G^3 for pageable memory transfers, and G^3 for pinned memory transfers, correspond to the values where the bandwidth is stable. For short memory transfers the bandwidth is fitted using a piecewise function. Values of $L + o$ are shown in the *LoHtD* and *LoDtH* columns. PCIe version and maximum speed for each GPU card are shown in the *PCIe* column.

This data transfer time prediction model can be easily included in our concurrent tasks execution model [15]. It is not necessary to compute beforehand the value of λ and α of each transfer command, only the L, o, G^1, G^2 and G^3 model parameters for each architecture. When a HtD (or DtH) ready command is selected, its execution time is computed using the non overlap model. If another DtH (or HtD) is already executing, then the execution time of both commands is recalculated assuming a full overlap and the minimum of both times is selected. This time reflects the amount of

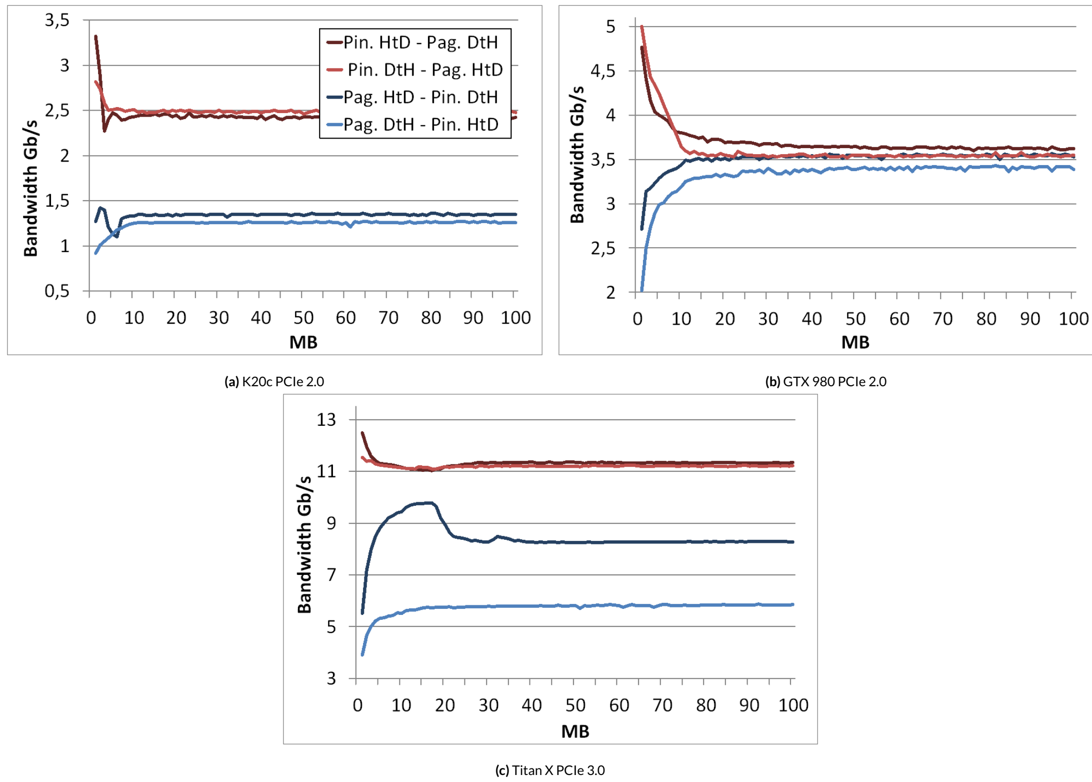


FIGURE 5 Average bandwidth obtained for K20c, GTX 980 and Titan X GPU cards by different combinations of a pageable and a pinned memory transfer in opposite directions.

TABLE 3 Parameters of the data transfer model on K20c, GTX 980 and Titan X for different types of memory transfers.

		Pageable				Pinned					PCIe
		LoHtD (ms)	LoDtH (ms)	G^1 (s/Gb)	G^3 (s/Gb)	LoHtD (ms)	LoDtH (ms)	G^1 (s/Gb)	G^2 (s/Gb)	G^3 (s/Gb)	
K20c	*HtD*	0.0135	0.0193	1.70	1.34	0.0177	0.0155	3.26	2.32	2.41	2.0 x16 5Gbit/s
	DtH			1.74	1.26			3.29	2.30	2.48	
GTX 980	*HtD*	0.0113	0.0141	4.35	3.54	0.0134	0.0121	6.01	3.93	3.63	2.0 x16 5Gbit/s
	DtH			3.68	3.39			5.86	3.94	3.54	
TITAN X	*HtD*	0.0081	0.0098	8.00	8.28	0.0096	0.0089	11.49	11.02	11.33	3.0 x16 8Gbit/s
	DtH			10.57	5.81			12.79	11.02	11.2	

overlap among them and can be used to compute the value of λ for the longest transfer command. Finally, the predicted time for that command can be updated using Eq. 1.

5 | EXPERIMENTS

All the experiments in this work have been conducted using a set of real kernels obtained from the CUDA and Rodinia SDK like in [15]. Tasks with different transfer and kernel processing times have been selected (see Table 4). There are tasks with a short *HtD* command and a long *K* command,

tasks with long *HtD* and *DtH* commands and short *K* commands, and so forth. Although some of these tasks have several commands of the same type (for example, matrix multiplication must transfer two matrices from host to device, or pathfinder launches several kernel commands in order), the scheduler considers there is a single command that encompasses all the commands of the same type.

In order to represent a more varying computational load, different input parameters have been selected to increase the number of tasks to 21. Moreover, to extend the applicability of the results, three GPU architectures, a K20c (Kepler), a GTX 980 (Maxwell) and a Titan X (Pascal), have been considered. Finally, to avoid the effect of outliers, every experiment is executed fifteen times to record a mean makespan.

TABLE 4 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

5.1 | Statistical analysis

First, we will analyze statistically our concurrent tasks execution model to assess the validity of the new heuristic and compare it with the previous heuristics. With this aim, we have obtained every possible combination of 4 tasks, with repetition, from the set of 21 tasks. This represents a total of $(\binom{21}{4}) = 10626$ benchmarks of four tasks each one. For each benchmark there are $4! = 24$ different schedules, thus there are 255024 different experiments that have been executed 15 times to record a mean makespan for each one. We have selected the median and minimum makespan for each benchmark as well as the makespan obtained by each heuristic. Finally, we have computed the speed-up of each heuristic with respect to the median per benchmark, and for comparison purposes the speed-up of the minimum with respect to the median (best speed-up). These values are represented in Figure 6.a using box plots to visualize their statistical properties [25]. On each box, the central mark corresponds to the median, the edges of the box are the 25th and 75th percentiles, the whiskers are drawn with dashed lines and extend to the most extreme datapoints not considered to be outliers (about $\pm 2.7\sigma$ and 99.3% coverage), the outliers are plotted individually with '+' signs, and there are notches at the median marks for comparison intervals. Two medians are significantly different at the 5% significance level if their intervals do not overlap.

The leftmost box of Figure 6.a corresponds to the Slope Index heuristic. This heuristic is very simple but the results are poor. Thus, its median value is very close to 1 (that is, almost no speed-up is obtained) and in many samples the speed-up is below 1. Next box shows the results of the NEH algorithm. Results are much better, with a median speed-up close to 1.05, and most speed-up values are above 1 although the lower whisker

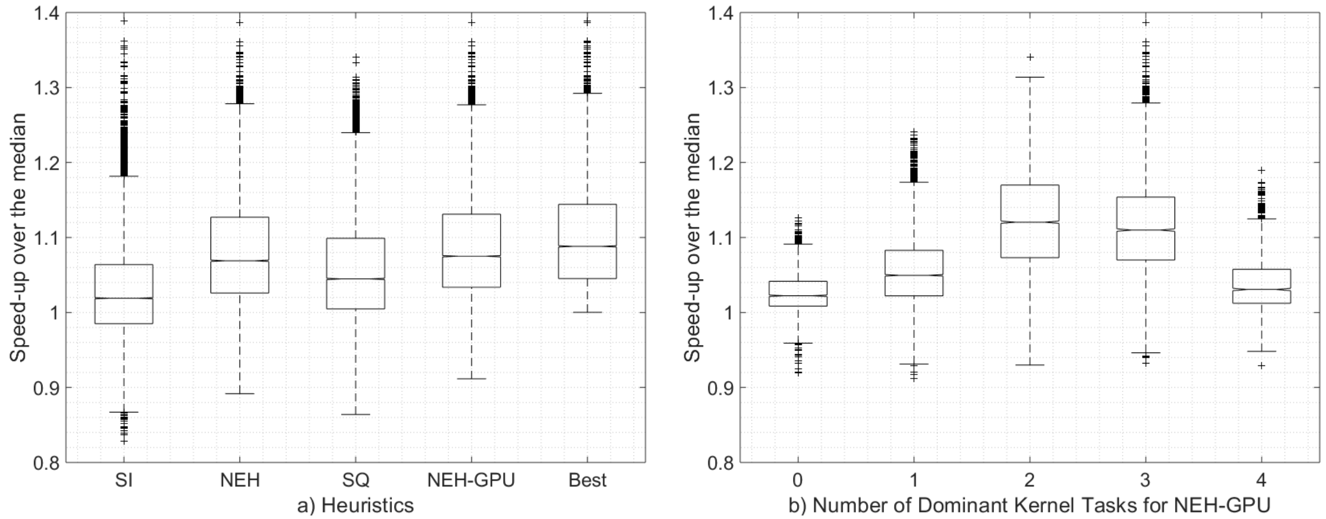


FIGURE 6 Box plots of speed-up over mean makespan values of 10626 benchmarks of 4 tasks in K20c, GTX 980 and Titan X devices. Central marks correspond to median values, boxes encompass the second and third quartiles, whiskers extend to cover 99.3% of the values, outliers are plotted individually with '+' signs, and notches at the median marks can be used for comparison intervals. Leftmost figure (a) shows results for each heuristic (SI, NEH, SQ and NEH-GPU) and for the best makespan (Best). Rightmost figure (b) shows results for the NEH-GPU heuristic and different combinations of DK and DT tasks (left box considers only benchmarks with 4 DT tasks, in next column benchmarks with 1 DK task and 3 DT tasks, and so on).

extends below 0.9. The box at the center corresponds to the Single Queue heuristic. Results are better than SI but not as good as NEH. Most speed-up values are above 1 and the lower whisker extends below 0.9 as in the NEH algorithm. Finally, the two last boxes correspond to the new algorithm presented in this work, NEH-GPU, and the best attainable speed-up. NEH-GPU obtains results very close to the best, with a slightly lower median and the lower whisker extended above 0.9. Compared with the other heuristics, its median speed-up value is higher and it has no outliers below the lower whisker. Notches in all boxes are very narrow but none of them overlap, thus every heuristic is significantly different at the 5% significance level. We have confirmed it using an unequal variances t-test between NEH and NEH-GPU speed-up values to discard they belong to the same distribution.

Both NEH-GPU and Best results show there are many experiments where speed-ups up to 1.40 can be obtained, and it would be interesting to ascertain what type of experiments can obtain more benefits from a good schedule selection. A sensible approach could be to study the overlapping opportunities among several tasks; for example, tasks with long K commands can be easily overlapped with tasks with long *HtD* commands. Therefore, we have classified the tasks as dominant transfer (DT) or dominant kernel (DK) depending on whether the transfer time dominates over the kernel time, or the other way round, and we have analyzed the results obtained by different combinations of DT and DK tasks. Figure 6.b shows the box plots of the speed-up values obtained by the NEH-GPU algorithm separating the results by the number of DK tasks. Leftmost box corresponds to experiments where there are no DK tasks (the four tasks are DT), next box takes into account experiments with only one DK task and three DT tasks, and so on. It can be seen that experiments where all the tasks are of the same type get little benefit from a good schedule, but experiments with a fair mix of DK and DT tasks have more opportunities to overlap and speed-up values can be much higher.

5.2 Applicability and scalability

In this section the four heuristics will be evaluated in a demanding multithreaded scenario (e.g. heterogeneous computing server) where several threads running applications (workers) offload tasks onto a device. All workers send task information to a buffer that is constantly polled by a host proxy thread. This information includes the worker id and CUDA API *HtD*, *DtH* and kernel launch calls. The proxy thread is in charge of reordering the set of tasks found in the buffer using one of the four heuristics, and submitting the corresponding commands to the device.

The scalability of each heuristic is tested by increasing the number of tasks from 4 to 8 and 16. This number of tasks represents a very large number of different benchmarks, $(\binom{21}{8}) \approx 3\,10^6$ for 8 tasks and $(\binom{21}{16}) \approx 10^9$ for 16 tasks. Besides, there are $8! = 40320$ possible schedules with 8 tasks, and $16! \approx 10^{13}$ with 16 tasks. Therefore, a random subset of different benchmarks have been selected for each combination of number of tasks (4, 8 and 16) and GPU architectures (K20c, GTX 980 and Titan X), and only the results obtained by each heuristic have been computed. For comparison purposes, the best makespan (BM) 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 (HM) is compared with this reference value to assess the proximity value. This proximity is computed by dividing both values ($BM/HM \cdot 100$) to obtain a percentage with maximum value of 100% (the higher the better). Results include the overhead of schedule computation incurred by each heuristic. This overhead ranges from $1\,\mu s$ for SI to less than $1ms$ for the rest of heuristics when 16 tasks are scheduled. Thus, taking into account the makespan duration, scheduling overhead is negligible.

Figure 7 shows the average of the results obtained with this experiment per each heuristic. In order to extract conclusions they are grouped attending to the number of tasks, the proportion of dominant kernel tasks, and the different GPU architectures, respectively. A proximity value closer to 100% means that heuristic obtains the best makespan, or a value very close to it, most of the times. It can be seen that the new NEH-GPU heuristic consistently obtains better proximity values than the other heuristics in all the situations and GPU architectures.

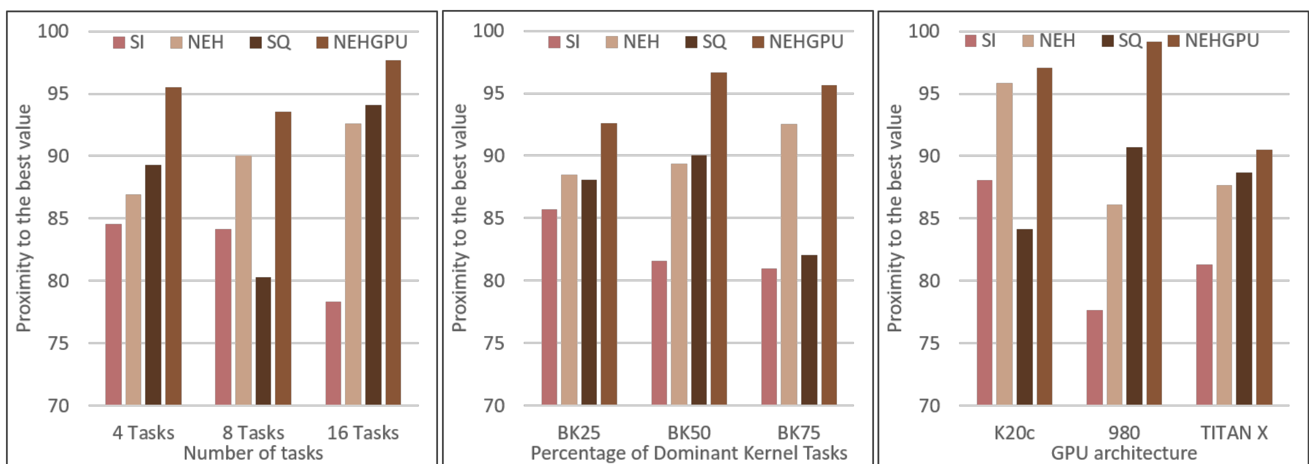


FIGURE 7 Proximity to the best values with different number of tasks (left plot), different proportion of dominant kernel tasks (center plot), and different GPU architectures (right plot)

There is a limitation, imposed by current CUDA implementations, on the number of tasks that can be concurrently executed on a device. This limitation is due to the implicit synchronization caused by any freeing command of device memory. More precisely, when `cudaFree` is called from any task, all previous CUDA commands in that context must finish before any new command is launched. Thus, all tasks executing concurrently must fit in the device memory. GPU cards used in this work have a memory size between 4 GB and 12 GB, while most kernels in Table 4 need between 1 Mb and 128 MB depending on the input parameters. Therefore, a sensible maximum number of tasks executing concurrently could be 64 for most current NVIDIA architectures. Nevertheless, this number of tasks is too high to be scheduled by our heuristic in a reasonable time, thus in the following we will present a fast and efficient method to solve this problem.

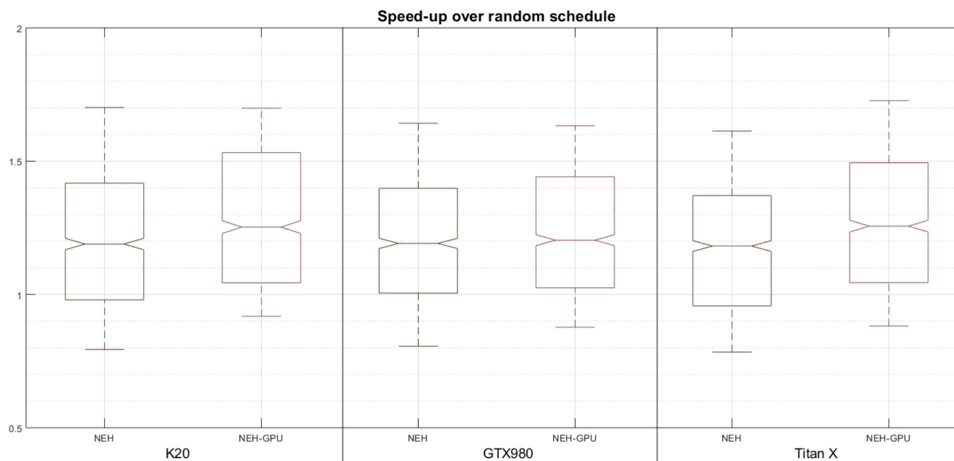
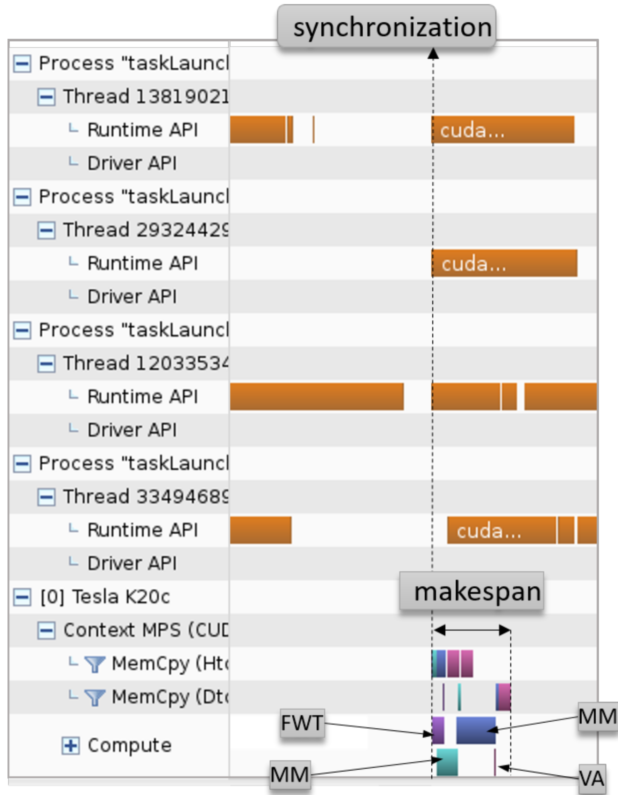


FIGURE 8 Box plots of speed-up of NEH and NEH-GPU heuristics over random schedule when launching groups of 64 tasks in K20c, GTX 980 and Titan X devices.

From previous experiments, it can be seen, on one hand, that best results are obtained when a fair mix of DK and DT tasks are scheduled together and, on the other hand, up to 16 tasks can be scheduled with a low overhead. Then, instead of computing the schedule of the 64 tasks altogether, we will group these tasks in four batches looking for the best balance between DK and DT tasks, and compute the schedule of each batch separately.

In the next experiment we will consider there are hundreds or thousands of tasks waiting to be executed in a cluster of GPUs. To maximize GPU utilization, these tasks are grouped in sets of 64 tasks that are launched onto each GPU. We will simulate this situation by randomly selecting 64 tasks from Table 4 and launching them with no particular order. Then, we will use previous method with the NEH and the NEH-GPU heuristics to obtain another schedule, and their execution times will be compared. We have repeated this 10^5 times in each of our GPU cards to obtain box plots of the speed-up of both NEH and our heuristic with respect to the random schedule. These box plots are shown in Figure 8, where the median values for NEH-GPU are 1.2630 for K20c, 1.2050 for GTX 980 and 1.2468 for Titan X. These experiments use random subsets of tasks, thus the speed-ups are highly variable. Nevertheless, values in the second and third quartile are always above 1 in all GPU cards and better than the speed-ups obtained by NEH.

5.3 | Comparison with MPS



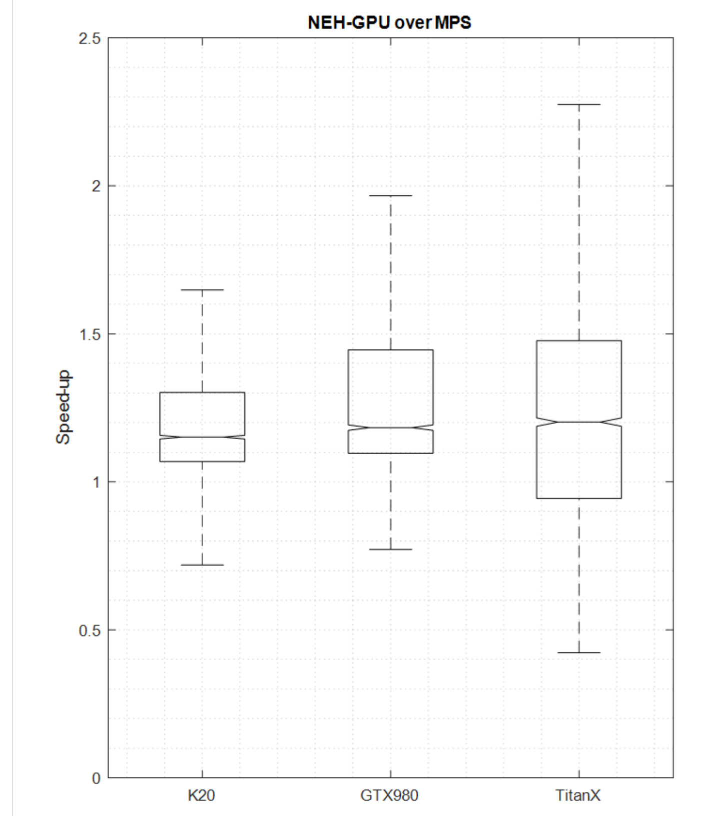


FIGURE 9 Left: profiler output of 4 processes launching different tasks (two MM, one FWT and one VA) running in the same CUDA context managed by the MPS server process. Right: box plots of speed-up of NEH-GPU over MPS. Makespan values of 10626 benchmarks of 4 tasks in K20c, GTX 980 and Titan X devices are shown.

CUDA provides MPS to enable co-operative multi-process CUDA applications to utilize Hyper-Q capabilities. With MPS, CUDA applications funnel their work through a MPS server process that creates a GPU context shared by all their clients. Thus, MPS is able to overlap GPU data transfer and kernel execution commands launched by different applications. As our proposal also takes advantage of this command concurrency to reduce the makespan, we conduct a comparative experiment in this section. Concretely, we compare the performance achieved by MPS when several process launch GPU tasks to that obtained by our scheduler.

Left part in Figure 9 shows a snapshot of the output of the Nvidia Visual Profiler while four processes launch different tasks (two MMs, with different matrices sizes, one FWT and one VA). Dark orange boxes correspond to CUDA API calls, while the other colored boxes correspond to the *HtD*, kernel and *DtH* commands running in the same CUDA context (managed by MPS) in a Tesla K20c. To make a fair comparison with our proxy, all CUDA applications synchronize, using POSIX semaphores, before launching their first host to device transfer. This synchronization tries to avoid scheduling penalties introduced by the OS while running the applications. Finally, the makespan is computed measuring the time since the beginning

of the first *HtD* command to the end of the last *DtH* command. As a side result of using MPS, now kernel commands can partially overlap as it is shown in Figure 9 with one MM task and the FWT task.

We have compared MPS with the results obtained by NEH-GPU using a host proxy thread as in the previous subsection. As in the first experiment, 10626 benchmarks of four tasks have been executed fifteen times in each GPU card to record makespan values. Sometimes the OS introduces a severe penalty that seriously degrade the performance of MPS, thus we have removed all these abnormal values using the median absolute deviation [26]. Right part of Figure 9 shows box plots of the speed-up of our heuristic with respect to MPS. It can be seen that, although there are some results were NEH-GPU performs worse than MPS, most values are above 1. Values below to 1 usually correspond to benchmarks where two or more kernel commands can overlap, while values above 1 are typically due to a better schedule of the four tasks by NEH-GPU. The median speed-up of NEH-GPU over MPS is 1.1509 for K20c, 1.1830 for GTX 980 and 1.2018 for Titan X. These results confirm the benefits of using an efficient reordering heuristic when independent task are concurrently executed on GPU.

6 | CONCLUSIONS

There have been several previous works studying tasks scheduling policies on GPUs. As far as we know, this is the first paper that discusses 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. 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 (SI, NEH and SQ heuristics). 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. The GPU tasks execution model includes a precise data transfer model that predicts the execution time of each data transfer command. This model can consider different GPU architectures (Kepler, Maxwell, and Pascal), memory types (pageable and pinned), and concurrent data transfers.

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 using several real kernels from the CUDA and Rodinia SDK. A statistical analysis has been conducted to assess the significance of the NEH-GPU heuristic and to show its advantage over other heuristics. Furthermore, the number of tasks has been varied to assess the scalability of the heuristics. In all of them NEH-GPU has obtained the closest results to the best makespan obtained by any heuristic. Finally, it has been made a comparison with MPS that shows that our solution obtains speed-ups ranging from 1.15 to 1.20.

We believe this approach can be included in either the MPS server or the runtime support of any GPU to improve concurrency among kernel and transfer commands. Another option is to use a host proxy thread, like in Section 5.2, to take care of every GPU command in a complex application. In future works we plan to extend our model to include scheduling policies for concurrent kernel execution (CKE).

ACKNOWLEDGMENTS

This work has been supported by the Ministry of Education of Spain (TIN2016-80920-R) and the Junta de Andalucía of Spain (TIC-1692). We also thank Nvidia for hardware donations within its GPU Grant Program.

References

1. NVIDIA . CUDA Multi-Process Service. March 2015.
2. Basaran C, Kang KD. Supporting Preemptive Task Executions and Memory Copies in GPGPUs. In: 24th Euromicro Conference on Real-Time Systems. ; 2012: 287–296
3. Lee H, Al Faruque MA. GPU-EvR: Run-time Event Based Real-time Scheduling Framework on GPGPU Platform. In: Proceedings of the Conference on Design, Automation & Test in Europe. European Design and Automation Association; 2014; 3001 Leuven, Belgium, Belgium: 220:1–220:6.
4. Park JJK, Park Y, Mahlke S. 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. ACM; 2015; New York, NY, USA: 593–606
5. Xu Q, Jeon H, Kim K, Ro WW, Annavaram M. Warped-slicer: Efficient intra-SM Slicing Through Dynamic Resource Partitioning for GPU Multiprogramming. In: Proceedings of the 43rd International Symposium on Computer Architecture. IEEE Press; 2016; Piscataway, NJ, USA: 230–242
6. Wang Z, Yang J, Melhem R, Childers B, Zhang Y, Guo M. Simultaneous Multikernel GPU: Multi-tasking throughput processors via fine-grained sharing. In: 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA). ; 2016: 358-369
7. Zhong J, He B. Kernelet: High-Throughput GPU Kernel Executions with Dynamic Slicing and Scheduling. *IEEE Transactions on Parallel and Distributed Systems* 2014; 25(6): 1522-1532. doi: 10.1109/TPDS.2013.257
8. Zhou H, Tong G, Liu C. GPES: a preemptive execution system for GPGPU computing. In: 21st IEEE Real-Time and Embedded Technology and Applications Symposium. ; 2015: 87-97
9. Chen G, Zhao Y, Shen X, Zhou H. EffiSha: A Software Framework for Enabling Effficient Preemptive Scheduling of GPU. In: Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. ACM; 2017; New York, NY, USA: 3–16
10. NVIDIA . CUDA Programming Guide, version 10.1.105. Februrary 2019.
11. Khronos Group . OpenCL 2.0 API Specification. October 2014.

12. NVIDIA . CUDA Samples, version 10.1.105. Februrary 2019.

13. Garey MR, Johnson DS, Sethi R. The Complexity of Flowshop and Jobshop Scheduling. *Mathematics of Operations Research* 1976; 1(2): 117–129.

14. Ruiz R, Maroto C. A comprehensive review and evaluation of permutation flowshop heuristics. *European Journal of Operational Research* 2005; 165(2): 479–494. doi: 10.1016/j.ejor.2004.04.017

15. Lázaro-Muñoz AJ, González-Linares J, Gómez-Luna J, Guil N. A tasks reordering model to reduce transfers overhead on GPUs. *Journal of Parallel and Distributed Computing* 2017; 109: 258–271. doi: 10.1016/j.jpdc.2017.06.015

16. Graham R, Lawler E, Lenstra J, Kan A. Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey. *Annals of Discrete Mathematics* 1979; 5(C): 287–326. doi: 10.1016/S0167-5060(08)70356-X

17. Pinedo M. *Scheduling: Theory, Algorithms, and Systems*. New York: Springer. 3 ed. 2008.

18. Framinan J, Gupta J, Leisten R. A review and classification of heuristics for permutation flow-shop scheduling with makespan objective. *Journal of the Operational Research Society* 2004; 55: 1243–1255. doi: 10.1057/palgrave.jors.2601784

19. Allahverdi A, Ng C, Cheng T, Kovalyov MY. A survey of scheduling problems with setup times or costs. *European Journal of Operational Research* 2008; 187(3): 985–1032. doi: 10.1016/j.ejor.2006.06.060

20. Johnson S. Optimal Two- and Three-Stage Production Schedules With Set-up Time Included. *Naval Research Logistics Quarterly* 1954; 1: 61–68.

21. Wagner HM. An integer linear-programming model for machine scheduling. *Naval Research Logistics Quarterly* 1959; 6(2): 131–140. doi: 10.1002/nav.3800060205

22. Palmer DS. Sequencing Jobs Through a Multi-Stage Process in the Minimum Total Time: A Quick Method of Obtaining a Near Optimum. *Journal of the Operational Research Society* 1965; 16(1): 101–107. doi: 10.1057/jors.1965.8

23. Nawaz M, Enscore EE, Ham I. A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. *Omega* 1983; 11(1): 91–95. doi: 10.1016/0305-0483(83)90088-9

24. Taillard E. Some efficient heuristic methods for the flow shop sequencing problem. *European Journal of Operational Research* 1990; 47: 65–74.

25. Tukey JW. *Exploratory Data Analysis*. Pearson. 1 ed. 1977.

26. Leys C, Ley C, Klein O, Bernard P, Licata L. Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median. *Journal of Experimental Social Psychology* 2013; 49(4): 764–766.