Received: 17 March 2017 | Revised: 12 October 2017 | Accepted: 21 November 2017
DOI: 10.1002/cpe.4409

WILEY

SPECIAL ISSUE PAPER

Maximizing the GPU resource usage by reordering concurrent kernels submission

Rommel A.Q. Cruz[1] | Cristiana Bentes[2] | Bernardo Breder[1] | Eduardo Vasconcellos[1] | Esteban Clua[1] | Pablo M.C. de Carvalho[1] | Lúcia M.A. Drummond[1]

[1]Institute of Computing, Federal Fluminense University, Niterói-RJ 24210-240, Brazil
[2]Department of System Engineering, State University of Rio de Janeiro, Maracanã-RJ 20550-900, Brazil

Correspondence
Rommel Cruz, Institute of Computing, Federal Fluminense University, Niterói-RJ 24210-240, Brazil.
Email: rquintanillac@ic.uff.br

Funding information
CNPq; CAPES; FAPERJ

Summary

The increasing amount of resources available on current GPUs sparked new interest in the problem of sharing its resources by different kernels. While new generations of GPUs support concurrent kernel execution, their scheduling decisions are taken by the hardware at runtime. The hardware decisions, however, heavily depend on the order at which the kernels are submitted to execution. In this work, we propose a novel optimization approach to reorder the kernels invocation focusing on maximizing the resources utilization, improving the average turnaround time. We model the kernel assignments to the hardware resources as a series of knapsack problems and use a dynamic programming approach to solve them. We evaluate our method using kernels with different sizes and resource requirements. Our results show significant gains in the average turnaround time and system throughput compared to the kernels submission implemented in modern GPUs.

KEYWORDS

graphics processing unit, kernel scheduling, multiprogramming

1 | INTRODUCTION

Graphics Processing Units (GPUs) are increasingly popular these days due to their higher performance and lower energy consumption compared with traditional multicore CPUs. GPUs support a massive amount of parallelism at a relatively low cost. The practice of offloading computation to run on GPUs provides significant acceleration for a wide range of arithmetic-intensive data-parallel applications.

In the case of NVIDIA GPUs and the CUDA programming model, *kernels* are offloaded to the GPU. The kernel code is executed by multiple parallel threads running on different GPU cores. The programmer organizes the threads into thread *blocks*, and the GPU architecture divides the cores into sets of Streaming Multiprocessors (SMs). Thread blocks are created at runtime, and the GPU scheduler assigns each block to a specific SM for execution. A thread block cannot migrate between SMs after it is scheduled.

In face of the increasing amount of resources available on current GPUs and the advances in their microarchitecture, an emerging challenge arises: how to share effectively the GPU resources by different kernels? There was expressive growth in core count, shared memory, registers, and threads per block limit, but many applications are not ready to take advantage of all these resources. According to the work of Pai et al,[1] the Parboil2 benchmark suite uses only from 20% to 70% of the Fermi GPU resources. Adriaens et al[2] performed similar studies for 12 real-world applications and show that most of them exhibit unbalanced GPU resource utilization.

In order to better utilize the GPU hardware resources, new generation NVIDIA GPUs support concurrent kernel execution. Initially, introduced in the Fermi architecture, concurrent kernel execution was somewhat restrictive. Programmers specify kernel independence – by placing them on different CUDA streams – and the hardware multiplexes the kernels into a work queue. Up to 16 kernels from different streams can execute simultaneously.[3] While this feature was beneficial for increasing the GPU utilization, there were limitations: (i) only kernels of the same

 © 2018 John Wiley & Sons, Ltd.

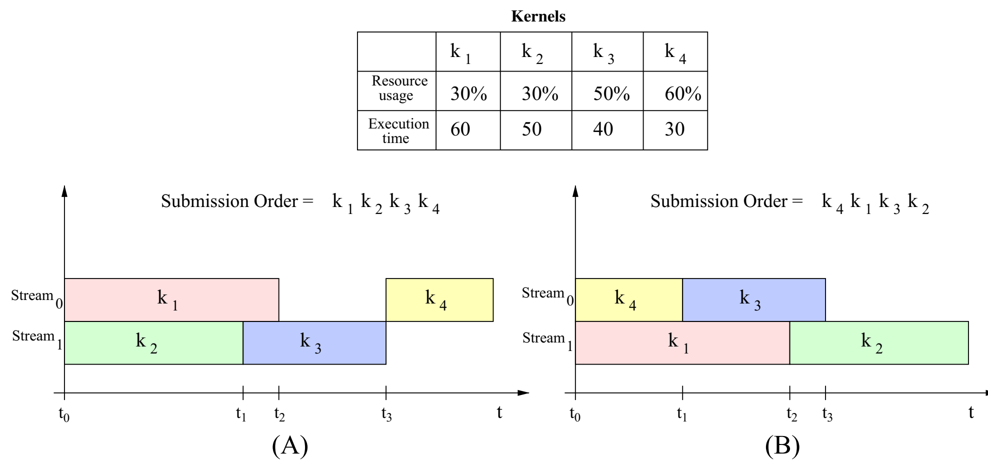

FIGURE 1 A set of four kernels is submitted to the GPU in two orders

application can execute concurrently, and (ii) the issue order to the work queue matters for concurrency, it can cause *false-serialization* (if kernel k_j that follows kernel k_i on the queue are on the same stream, they have to be complete before additional kernels in separate streams can be executed). Later, the Kepler architecture tackled these limitations by providing the Hyper-Q technology, where up to 32 hardware queues enable multiple applications running on different CPU cores to simultaneously utilize the GPU resources.

The Hyper-Q technology allows threads blocks from different streams to share the same resources in the GPU. For example, consider a hypothetical GPU architecture with one SM with 1024 registers, 8 K of shared memory, and a maximum of 1024 threads. Also, consider a kernel k_i with 2 thread blocks, each block containing 256 threads and using 2 K of the shared memory, and each thread using 1 register. In this scenario, the SM will have unused resources: half of the maximum threads, half of the shared memory, and half of the registers. These unused resources can be allocated to other thread blocks from different kernels (placed on different streams).

The NVIDIA scheduling policy is, however, proprietary, and no information has been made available. There is some speculation that the scheduling policy follows a *left-over* strategy.[1] The scheduler dispatches the blocks from the first kernel in the first queue. If there are still available resources, the blocks from the kernels of the next queues are also dispatched in a round-robin fashion. Two kernels from the same queue, however, are dispatched in sequential order. The problem with this scheduling policy is that the order at which the kernels were inserted on the queues has significant impact on the GPU occupancy. Suppose the example of Figure 1, where four kernels were submitted to execution. Each kernel requests a different amount of resources and has different execution times. In Figure 1A, we show their execution considering the submission order $\{k_1, k_2, k_3, k_4\}$. According to the CUDA scheduler, kernel k_1 is submitted first, and kernel k_2 is also submitted due to the left-over policy. When k_2 finishes, k_3 is submitted, but k_4 has to wait since its requested resource is not available. We can observe that, from t_0 to t_1, there are 40% of the resources still available and, from t_2 to t_3, there are 50% available. If we consider a different order to submit the kernels and submit in the order $\{k_4, k_1, k_3, k_2\}$, the CUDA scheduler will execute the kernels, as shown in Figure 1B. We can observe that, until t_3, there are at most 20% of the resources available, which improves significantly the resource usage.

In this work, we propose an optimization strategy that determines the best order to submit the kernels to the GPU. The idea is to simulate the assignment of kernels to the hardware queues in order to maximize the resources utilization, improving the overall throughput. Our approach models the problem of selecting which kernels are better to submit to take the most advantage of the available resources as a series of knapsack problems. We model the set of the available resources as the knapsack capacity and the set of kernels as the items to be put into the knapsack. The kernels have weights and values according to the amount of resource usage and the estimated execution time. The solution to the knapsack problem establishes the subset of kernels whose total weight is smaller than the available resources and the total value is as large as possible. We use a dynamic programming algorithm to solve the knapsack problem, and the algorithm outputs a reordering of the kernels submission that maximizes the resource utilization.

We present results of a series of experiments using real-world and synthetic applications with a different number of kernels and resource requirements. Our reordering approach was able to reduce the average turnaround time and increase the system throughput for real-world and synthetic applications compared to random kernel submission. We also show that the overhead of computing a number of knapsack problems is negligible compared to the obtained gains.

The remainder of this paper is organized as follows. Section 2 gives an overview about GPU Multiprogramming concepts and possibilities. Section 3 presents previous works on GPU concurrent kernel execution. Section 4 gives a brief introduction to the problem we are facing. Section 5 describes the proposed reordering approach. Section 6 presents the experimental results. Finally, Section 7 draws some conclusions and presents future research directions.

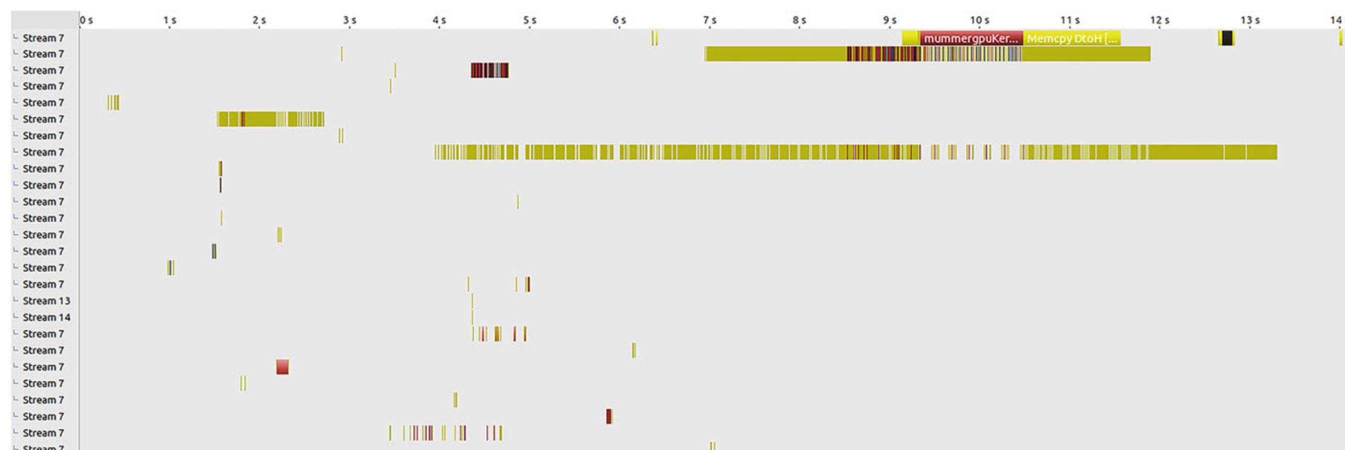

FIGURE 2 Execution profile of all the Rodinia benchmark applications when they are launched at the same time to run on the GPU

2 | GPU MULTIPROGRAMMING

In the earlier years, GPUs could only execute one kernel at a time. Thread-level parallelism was extensively used to exploit the GPU hardware and to hide latencies such as global memory accesses, control flow divergences, or synchronization. Despite the success of thread-level parallelism in providing substantial speedups for a number of applications, the GPU architecture has evolved to support concurrent kernel execution. Sharing the GPU among multiple kernels can improve utilization and unleash the power of the GPU for a dynamic and highly virtualized environment. In addition, the recent hardware trend that unifies the CPU and GPU memory will drastically reduce the data transfer cost and increase the competition for the GPU resources.

Concurrent kernel execution is, however, a relatively new feature in NVIDIA GPUs. Along a decade of using the GPU for general purpose computing, CUDA programmers are used to focus on exploiting thread-level parallelism, rather than kernel-level parallelism. Accordingly, most GPU applications are composed of a number of kernels that execute in order. In the available scientific CUDA-based benchmark suites, we observe a small number of applications that exploit concurrent kernel execution. Rodinia[4] has only one application and SHOC[5] has two applications, whereas Parboil[6] does not include any application with concurrent execution.

NVIDIA has introduced the Multi Process Service (MPS) to overcome this problem. MPS allows kernels from different processes (of different applications) to execute concurrently on the GPU. The MPS service aggregates kernels from multiple processes and issues them to the GPU as if they are coming from a single process. In this way, the Hyper-Q hardware can be better utilized.

We performed an experiment with MPS, where we submitted all the Rodinia benchmark applications at the same time to run on the GPU. Figure 2 shows the result of this execution using the CUDA Profiler tool[7] on a K40 GPU. Each horizontal row in the figure represents the timeline of one application execution. Each timeline shows the start and end of the activities that correspond to the kernel execution (red and black activities) and the CPU-GPU memory transfers (yellow activities). A vertical column in the figure represents the activities that occur at the same time in the GPU. Hence, vertical columns with more than one red or black activity represent concurrent kernel execution on the GPU. Examining the vertical columns, there is scarce competition for the GPU resources in this execution. Kernels are invoked in different timeline moments; they have disparate execution times and memory transfer overheads.

We believe that, in the future, GPUs will behave as multiprogrammed devices such as CPUs are in the present. In this scenario, a great number of kernels will be sharing the GPU resources and our reordering algorithm would bring even more benefits to the overall throughput of the system.

3 | RELATED WORK

Before the GPU had the hardware support to execute different kernels concurrently, there were some tentatives in overcoming this limitation in a software by *merging* the source of different kernels into one single kernel to execute them concurrently. Guevara et al[8] proposed a compile-time system that merges two kernels into a single kernel, which they call *thread interleaving*. Peters et al[9] proposed a persistent kernel (with a fixed number of thread blocks) that can execute different functions concurrently. In this way, it was possible to handle requests from different applications. Wang et al[10] addressed a limitation of the Fermi architecture that all the concurrent kernels must come from the same application. Their approach, called *context funneling*, is similar to kernel merging but uses kernels from different contexts. However, kernels with different features, such as computational intensive or memory bound, are treated equally at scheduling. Gregg et al[11] proposed an OpenCL concurrent kernel scheduler that uses *thread interleaving*. They combine two OpenCL dispatch operations into a single dispatch that launches the scheduler. The scheduler implements a queue, and the thread blocks from the two kernels are put on the queue according to the scheduling algorithms: (i) round-robin with work-stealing

and (ii) partitioning by kernel (ensures that the percentage of the resources assigned to each kernel remains fixed). Wang et al[12] proposed *kernel fusion*. Their focus, however, was on reducing energy consumption and improving power efficiency. Merging kernels at the source code or at the runtime circumvents by software some hardware limitations in launching concurrent kernels, but they do not consider the different resource requirements by the kernels.

On a different direction, some authors proposed mechanisms to modify the granularity of the kernels, by *slicing* them, in order to improve GPU utilization. Zhong et al[13] proposed to slice the kernel to create more opportunities for time sharing. They used a greedy algorithm to schedule the slices. Ravi et al[14] proposed the *molding* technique that allows the changing of the dimensions of grid and thread blocks, but the kernel code has to accept this variation. Pai et al[1] improved this technique, proposing the *elastic kernel*, that elasticizes any kind of kernel. They also proposed different elastic-kernels aware of concurrency management policies. For real-time applications, the works of Tanasic et al[15] and Park et al[16] proposed the implementation of preemption in the concurrent kernel execution.

There are also efforts in dividing the GPU resources among the concurrent kernels, called *spatial multitasking*. These efforts, however, focus on proposing hardware enhancements to improve GPU utilization. Liang et al[17] used the CUDA Profiler to determine the amount of global memory requested for each kernel and computes the kernel behavior in terms of latency and memory bandwidth by executing each kernel with a dummy counterpart. Their approach focuses not only on partitioning the SMs among the kernels but also on deciding which kernels execute concurrently. They use a heuristic approach based on dynamic programming to divide the SMs and select the kernels for concurrent execution. Their spatial division, however, is not currently supported by the hardware. Their framework to emulate the GPU execution can modify the register usage, which can impact the results. Adriaens et al[2] show that GPU applications have unbalanced resource utilization and also show through a simulation that statically partitioning the GPU resources among applications offers performance benefits. They propose four partition schemes to divide the GPU multiprocessors (SMs) among the applications. Their static division, however, does not take into account other resource usage other than the SMs.

Closer to our work are the proposals that focus on achieving maximum utilization based on the order in which GPU kernels are invoked on the host side, called *kernel reordering*. Wende et al[18] proposed kernel reordering for the Fermi hardware, where the concurrent kernel execution uses only one scheduling queue. They create a scheduler thread that arranges the scheduling queue in a round-robin fashion, without considering the threads resource utilization. They were successful in increasing the concurrency by interleaving different streams of execution in the same queue, but modern GPU hardware have the Hyper-Q mechanism that implements these queues in hardware. Our work is closest to the work of Li et al[19] that proposed a reordering scheme for GPU architectures with Hyper-Q technology available. Their approach computes a *symbiotic score* that tries to co-execute kernels with complementary resource usage and predicted power consumption using the concept of *execution rounds* – the round of simultaneous thread block execution on the SMs. They try to fulfill a round with symbiotic kernels using a greedy algorithm for the problem of multidimensional bin-packing. Although they also propose a reordering scheme, their solution is not directly comparable to ours. They focus on power prediction and they schedule for rounds of execution using a greedy approach. Our scheme, on the other hand, bases the scheduling decisions on an estimated kernel execution time and schedules the kernel as a whole. Nevertheless, we compare our dynamic programming scheme with the greedy algorithm to give an idea of their different scheduling decisions.

4 | PROBLEM DEFINITION

For a given GPU device, D, let $Ord = \{k_0, k_1, \dots, k_{N-1}\}$ be the list of N independent kernels in the order at which they are submitted for execution on D (k_i is submitted before k_{i+1}). Since our reordering strategy is dynamically executed, our solution can be used in GPUs that are constantly receiving new kernel execution demands. In this sense, Ord can be dynamically updated. Let NSM denote the number of SMs of D and consider the resources $R_j | j \in \{0, 1, 2\}$, where:

- R_0 is the shared memory size per SM;
- R_1 is the number of registers per SM;
- R_2 is the maximum supported number of threads per SM.

The total capacity of the each resource of D is defined as $W_j = R_j \times NSM \mid j \in \{0, 1, 2\}$. At a certain time during the execution, there are m kernels executing concurrently on D, using a percentage of W_j. The available resources at this time are called $W_j^{av} \mid j \in \{0, 1, 2\}$. Every time a kernel finishes and releases resources, new kernels are selected for execution according to the amount of available resources.

In order to find a new submission order, Ord^{new}, so that the resource utilization is maximized, we model the problem of selecting a number of new kernels to submit and take advantage of the available resources as a multidimensional knapsack problem. In the model, at the beginning of the execution and at each time a kernel finishes, D represents the knapsack with the capacity equal to the available resources, and the non-submitted kernels represent the items to be packed into the knapsack.

For each remaining non-submitted kernel k_i, the following information are collected:

- t_i^{est}: the estimated execution time;
- sh_i: the amount of shared memory required per block;
- nr_i: the number of registers required per block;

- nt_i: the maximum number of threads launched per block;
- nb_i: the number of blocks.

Regarding the estimated execution time, there are a number of approaches that predict it based on different techniques, such as simulation, machine learning, and analytic models.[20] Another solution consists in using time measures of a previous execution of the application.[21] The study and the implementation of such performance models are beyond the scope of this work.

The amount of shared memory, registers, and threads can be dynamically obtained by NVIDIA tools.[7] We assign for each kernel k_i a weight $w_{ij}(i \in \{0, \dots, N-1\}, j \in \{0, 1, 2\})$ that represents the amount of required resource j: $w_{i0} = sh_i \times nb_i, w_{i1} = nr_i \times nb_i$, and $w_{i2} = nt_i \times nb_i$. Note that, if $\exists j | w_{ij} > R_j$, k_i may still run since the GPU can schedule a number of blocks from k_i that fit into the GPU and, after they finish, they can schedule the remaining blocks. In this case, to consider k_i a candidate item, we have to adjust $w_{ij} = R_j$.

We also assign a value v_i for each kernel k_i that accounts for the average of the percentage of the resources required per unit of time. Kernels with smaller execution times will have higher values of v_i.

$$v_i = \frac{\frac{sh_i}{R_0} + \frac{nr_i}{R_1} + \frac{nt_i}{R_2}}{3 \times t_i^{est}} \tag{1}$$

Each knapsack problem consists of choosing the amount of kernels that the corresponding profit sum is maximized without having the weights sums exceeding W.[22] The knapsack 0-1 problem is formulated as the following maximization:

$$\text{maximize} \sum_{i=0}^{N-1} v_i x_i \tag{2}$$

$$\text{subject to} \sum_{i=0}^{N-1} w_{ij} x_i \leq W_j | j \in \{0, 1, 2\}, \tag{3}$$

where $x_i \in \{0, 1\}$, $x_i = 1$ if item i should be included in the knapsack, and $x_i = 0$ otherwise. In this maximization problem, the maximization of the knapsack profit represents the maximization of the GPU resource usage.

5 | REORDERING APPROACH

Basically, our kernel reordering approach simulates the concurrent execution of the kernels. We define a set of S streams. In our current implementation, S is set to the number of hardware scheduling queues. Up to S kernels can be executed concurrently on the GPU. The algorithm considers that the kernels are scheduled by the hardware in a Round-Robin fashion while there are available resources (as described in Section 1).

Algorithm 1 describes our approach. Overall, it iteratively simulates the termination of the earliest kernel on the streams and finds the next kernels to insert into the streams by solving a knapsack problem with the non-submitted kernels and the available resources released by the earliest kernel.

Algorithm 1 Main kernel reordering algorithm

```
1:  function KernelReorder(KernelList, W^av)
2:      while (KernelList not empty) do
3:          NextSubmitList ←
4:            GetKernelsToSubmit(KernelList, W^av)
5:          for Each k_i in NextSubmitList do
6:              // Simulate k_i execution
7:              stream_l ← FindEarlyStream(W^av, Stream)
8:              Insert k_i into stream_j
9:              Retain k_i resources by removing from W^av
10:             Remove k_i from KernelList
11:         end for
12:         Earliest ← SelectEarliestKernel(Stream)
13:         // Simulate Earliest termination
14:         Release Earliest resources by inserting into W^av
15:     end while
16:     Ord^new ← RoundRobingCross(Stream)
17:     return O^new
18: end function
```

Initially, $KernelList$ receives all the kernels on $Ord = \{k_0, k_1, \dots, k_{N-1}\}$, and $W_j^{av} = W_j, \forall j \in \{0, 1, 2\}$. The order Ord represents the order at which the kernels were invoked by the application.

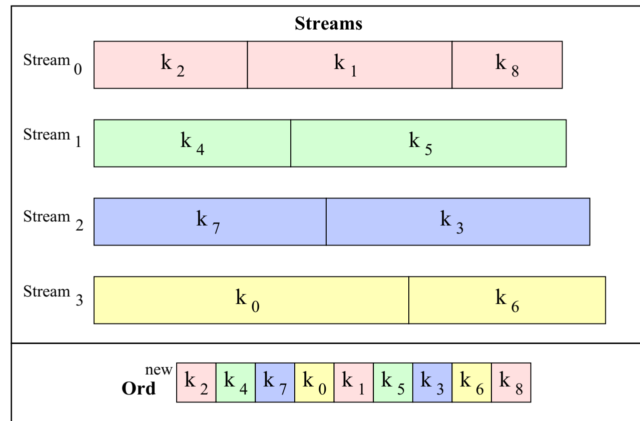

FIGURE 3 Example on building Ord^{new} from 4 streams in a Round-Robin fashion

While there are kernels in *KernelList* to be submitted, the algorithm finds a list of kernels to submit and takes advantage of the available resources. The *GetKernelsToSubmit* function solves a 0-1 knapsack problem using a dynamic programming approach and outputs a list of kernels (*NextSubmitList*) in a descending order of values, whose execution will maximize the use of the available resources. For each kernel k_i in *NextSubmitList*, the algorithm inserts the kernel into the stream whose finishing time (completion time of the last element) is the smallest. After that, the resources required by k_i are removed from W^{av}, and k_i is considered submitted by removing it from *KernelList*. The next step is to advance the execution and search for the next kernel termination by the function *SelectEarliestKernel*. This function searches on the streams for the kernel with the smallest t^{est}, which means the kernel with the earliest finishing time. The completion of the earliest kernel is simulated by inserting its required resources back into W^{av}.

At the end, the algorithm builds Ord^{new} by inspecting the streams in a round-robin fashion (Line 14) as exemplified in Figure 3.

5.1 | Obtaining a kernel submission set by solving a knapsack problem

We modeled our problem as a multidimensional knapsack problem. However, because the multidimensional knapsack is computationally harder than the classical knapsack problem, we adapt it so that it is solved as the classical problem by using a dynamic programming approach. Remark that the dynamic programming algorithm for the knapsack problem has a time complexity of $O(NW)$, where N is the number of items and W is the capacity of the knapsack, and it also requires $O(NW)$ space.

At first, we need to consider that kernel weights and GPU capacity are represented by an integer.

In order to decide which items shall be put into the knapsack using the classic 0-1 dynamic programming, where each item in the knapsack has a weight and an associated value, we need to use a recurrence equation to construct a matrix. In this matrix, the rows represent the number of each item to be included into the knapsack and the columns represent all possible weights from zero to the maximum capacity of the knapsack. To adapt this algorithm to a multi-dimensional problem, each item (kernel) now will have a number of weights, which are registers, shared memory, and a number of threads associated with it. In addition, each column of this matrix will be represented by the linearization of the weights in the three dimensions, transforming the multi-dimensional problem into one-dimensional and leaving the matrix in the same way as that used in the classic dynamic programming. So, to access the elements of the matrix instead of representing the capacity of the GPU by three elements, W_0, W_1, and W_2, we calculated a unique value W to represent it. Analogously, the resources required by each kernel k_i, represented also by three values, were reduced to a unique one, w_i. Then, $W = W_0 \times W_1 \times W_2$, and for each kernel k_i, $w_i = (w_{i0} \times W_1 \times W_2) + (w_{i1} \times W_2) + w_{i2}$. However, note that, when a kernel is chosen, instead of decreasing a unique kernel weight from the GPU capacity, each of the three associated weights (resources) is decremented separately. The linearization is used only to access the elements of the matrix.

Algorithm 2 presents the main procedure of the dynamic programming method used to solve our problem. It requires the following variables and constants:

- W^{av} - constant, representing the available capacity of the GPU
- N - constant, number of kernels
- *tempW* - variable initialized with W
- *SelectedKernels* - variable, representing a set of kernels
- w_i - constant, representing the weight of $kernel_i$
- v_i - constant, representing the value of $kernel_i$
- M - variable, matrix of size $(N + 1) \times (W + 1)$ built by the dynamic programming method
- *NextSubmitList* - list of ordered kernels

Algorithm 2 Dynamic programming- Main procedure

```
1:  function GetKernelsToSubmit(KernelList, W^av)
2:      SelectedKernels ← ∅
3:      tempW ← W^av
4:      M ← KnapsackMatrix(KernelList, W^av)
5:      for i ← N to 1 do
6:          if tempW > 0 then
7:              if M[i][tempW] ≠ M[i − 1][tempW] then
8:                  tempW ← tempW −w_i (Update also all the three weights separately)
9:                  SelectedKernels ← SelectedKernels ∪ k_i
10:             end if
11:         end if
12:     end for
13:     NextSubmitList ← OrderSet(SelectedKernels)
14:     return NextSubmitList
15: end function
```

Basically, algorithm 2 works as follows. In lines 2 and 3, the variables *SelectedKernels* and *tempW* are initialized with an empty set of kernels and the total capacity of the GPU, respectively. In line 4, the *KnapsackMatrix* function is called with parameters *KernelList* and W^{av}, containing all kernels to be ordered and the available capacity of the GPU, and returns the matrix *M*. For each kernel k_i, in line 5, if there are remaining resources, the possibility of including k_i is evaluated by consulting the matrix *M*. Line 7 tests if k_i was selected by the dynamic programming method. If the value of the current line *i* is different from the previous one in column *tempW* (meaning that the dynamic programming method selected k_i), it is subtracted by w_i and k_i is included in the set *SelectedKernels* (lines 8 and 9). After that, *NextSubmitList* receives the elements of *SelectedKernels* ordered in descending order of value by *OrderSet* (line 10).

Algorithm 2 calls the function *KnapsackMatrix*, detailed in Algorithm 3, which returns the matrix *M*. A row of *M* represents a kernel, whereas a column represents a linearized quantity of resources. This matrix is initialized in line 3 of Algorithm 3. It can be updated either in line 7 or line 9. In line 7, a cell of *M* is copied to another one, meaning that the object (kernel k_i) is not selected to be included in the knapsack (GPU) because it exceeds the remaining capacity of the knapsack; otherwise, it can be included in the knapsack whether it results in some profit (line 9).

Algorithm 3 Dynamic programming- Building of matrix M

```
1:  function KnapsackMatrix(KernelList, W^av)
2:      for j = 0 to W^av do
3:          M[0, j] ← 0
4:      end for
5:      for i = 1 to N do
6:          for j = 1 to W^av do
7:              if (w_i > j) then
8:                  M[i, j] ← M[i − 1, j]
9:              else
10:                 M[i, j] ← max(M[i − 1, j], v_i + M[i − 1, j − w_i])
11:             end if
12:         end for
13:     end for
14:     return M
15: end function
```

6 | RESULTS

6.1 | Computational environment

We used two different GPU architectures in our experiments: NVIDIA TITAN X (Maxwell architecture) and Tesla K40 (Kepler architecture). Their characteristics are presented in Table 1. Each GPU was on a separate server machine running GNU/Linux Ubuntu 14.04 (64-bit) with CUDA 7.5 connected via PCIe. Also, the TITAN X is connected to an Intel i7-5930 K @ 3.50 GHz with six cores and 32 GB of RAM, and the K40 is connected to an Intel i7-950 @ 3.07 GHz with four cores and 8 GB of RAM.

TABLE 1 GPU configurations

	TITAN X	K40
Number of cores	3072	2880
Core Clock	1000 MHz	745 MHz
RAM	24 GB	12 GB
Memory Bandwidth	336.5 GB/s	288 GB/s
Capability	5.2	3.5
Number of SMs	24	15
Shared Memory per SM	96 KB	48 KB
Number of Registers per SM	64 K	64 K
Max number of threads per SM	2048	2048
Architecture	Maxwell	Kepler
Peak Performance	6.6 TFLOPS	4.2 TFLOPS

We implemented a wrapper as an intermediate layer between CUDA applications and the CUDA driver that intercepts the kernels invocation. Our wrapper captures the number of blocks and threads from the invocation command and creates an execution queue. The kernels in this execution queue are submitted to run on the GPU in the order given by our approach.

6.2 | Applications

We evaluate our reordering algorithm using two different types of benchmarks. The first type is a set of real-world applications selected from the benchmark suite Rodinia.[4] The second type is a set of synthetic benchmarks whose size and resource usage can be procedurally varied for experimentation. We did not generate tests with all possible permutations of kernel orders due to the intractable number of possibilities.

From the Rodinia benchmark suite, we select Speckle Reducing Anisotropic Diffusion version 2 (SRAD), Breadth-First Search (BFS), LU Decomposition (LUD), Hotspot 3D (HS3) and 2D (HS2), k-Nearest Neighbors (kNN), Path Finder (PF), and Heart Wall (HW). These applications represent general-purpose GPU applications with diverse characteristics, including thread structures, computation, and memory access patterns. The applications details are shown in Table 2.

In order to increase the competition for the GPU resources (avoiding the effects shown in Figure 2), we executed the kernels from these applications as if they were invoked at the same time.

To evaluate future scenarios where the GPU behaves as a multiprogrammed device, we propose a set of synthetic kernels where we can increase the resource competition and empirically simulate different resource consumption.

The general structure of each synthetic kernel k_i in the set consists of a CUDA code that performs a set of arithmetic operations on an array of integer numbers allocated on the shared memory. The number of blocks of k_i and its resource requirements, sh_i and t_i^{est}, are created randomly. The value of sh_i determines the size of the array of integers, it varies from 1024 to 49152 bytes. The value of nt_i determines the number of threads per block; we set it to 32. The value of t_i^{est} determines the number of arithmetic operations performed whose execution time varies from 1 ms to 5 s. In our experiments, we do not stress nr_i (we keep $nr_i = 0$).

We create a number of experiments where the number of competing kernels, N, varies in $\{32, 64, 128, 256\}$, and for each value of N, we generate a set of kernels. For each kernel in the set, the maximum number of blocks, NB, varies within $\{32, 64, 128, 256, 512\}$ and the minimum number of blocks is always set to 4.

TABLE 2 Rodinia applications characteristics

App	Registers	# Blocks	# Threads	Computation (μs)		Shared Memory (b)
				K40	TITAN X	
LUD	32	1	16	25	26	1024
HS3	36	1024	56	152	110	0
SRAD	21	16384	256	739	560	5120
PF	13	463	256	73	55	2048
BFS	19	1954	512	34	20	0
HS2	38	1849	256	204	156	3072
kNN	8	3840	256	68	41	0
HW	38	51	256	12759	12055	11872

6.3 | Metrics

Maximizing the GPU resources occupation does not always lead to better application performance. However, as we focus on multiprogrammed GPUs, our metrics are system oriented, such as average turnaround time and system throughput.

The system oriented metrics used in our experiments are the Average Normalized Turnaround Time (ANTT) and the System Throughput (STP) proposed by Eyerman and Eeckhout[23] and used in the works of Pai et al[1] and Park et al[16] to evaluate multitasking on the GPU. ANTT is a lower-is-better metric that measures the average turnaround time normalized by the kernel execution time (when the kernel is executed without concurrency). STP is a higher-is-better metric that measures the number of tasks completed per unit time. Given that the turnaround time of a kernel k_i is represented by TT_i and the turnaround time of k_i when it is executed alone on the GPU without concurrency is TT_i^{SP}, ANTT and STP are computed according to the following:

$$ANTT = \frac{1}{N}\sum_{i=0}^{N-1}\frac{TT_i}{TT_i^{SP}} \tag{4}$$

$$STP = \sum_{i=0}^{N-1}\frac{TT_i^{SP}}{TT_i}. \tag{5}$$

6.4 | Real applications results

In the experiments with real-world applications, we execute the kernels listed in Table 2 using three different submission orders: *Dynamic*, *Random*, and *Worst*. The Dynamic submission is the one obtained after the execution of our dynamic programming reordering algorithm. The Random submission represents the average result of 10 random submission orders. The Worst submission represents the worst case scenario of the kernels submission we had: the submission order gives preference to kernels with longer execution time.

Figure 4 compares the ANTT results for the three different submission orders. As we can observe, our reordering algorithm is able to reduce the ANTT in 67% when compared to the Random submission. Compared with the worst case scenario, the reduction is significant around 94%. In the worst case scenario, smaller kernels have to wait for longer time, this increases the average waiting time. Our reordering approach, on the other hand, reduces the average waiting time because smaller kernels have higher values of v_i (refer to Eq. (1)) and are submitted before the large ones.

Figure 5 shows the STP results for the three different submission orders. Our reordering algorithm can achieve an increment up to 1.4 times in STP over the random submission. It can further provide an improvement of 1.6 times over the worst schedule.

6.5 | Synthetic applications results

In the experiments with synthetic applications, we compare our approach with the Random submission scheme and also with a *Greedy* submission order. The Greedy submission order uses a greedy algorithm to solve the knapsack problem, where the score is based on the ratio v_i/w_i. The algorithm starts by ordering the list of kernels in the descending order their scores. For all kernel candidates on the list, the algorithm selects the kernels with the highest score that has sufficient resources to execute.

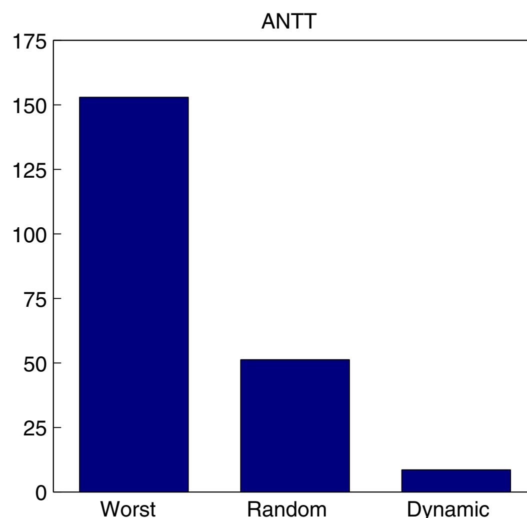

FIGURE 4 ANTT results for Rodinia kernels on TITAN X

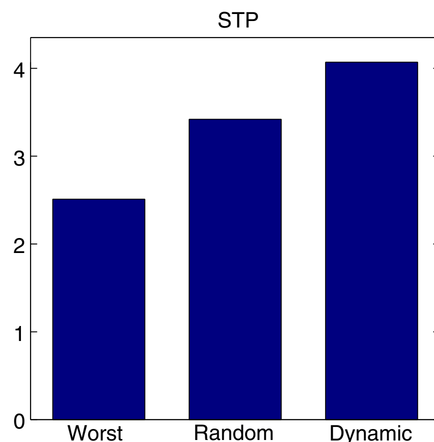

FIGURE 5 STP results for Rodinia kernels on TITAN X

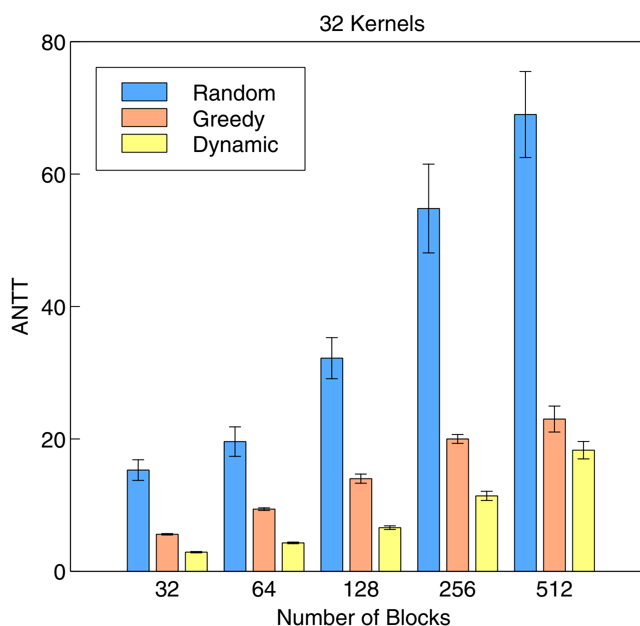

FIGURE 6 ANTT for 32 kernels on TITAN X

In the experiments with synthetic kernels, we have the ability to vary in a controlled way the kernel resource requirements and we also have much more kernels to evaluate. We generate 50 different sets of random kernels for each experiment. The results for Random submission represent the average execution results of the 50 sets of kernels without any reorder scheme. The results for Dynamic and Greedy submissions represent the average execution results of the same 50 sets of kernels using the two reorder algorithms. This way, we guarantee the evaluation of our reordering approach under different substantial resource requirement scenarios.

Figures 6 to 9 show the ANTT results for the different set sizes' executions and the different values of *NB*. Error bars show standard errors. The standard errors are quite small for Dynamic and Greedy and decrease with the increase in the number of kernels. We can observe in these figures that reordering the kernel submission with dynamic programming or the greedy algorithm has significant impact on the average turnaround time of the kernel set. The reductions in ANTT vary from 45% to 75%. We can observe that the kernel set with $N = 256$ produced more pronounced gains. This occurs because there are more items to be put in the knapsack and the algorithm has more possibilities to fulfill it. When comparing our dynamic programming scheme with the greedy approach, we obtain consistent gains in ANTT.

Table 3 shows the STP results for the same variations of *N* and *NB*. We observe in this table that the throughput is considerably increased using a reordering scheme. Our dynamic programming approach increases STP from 1.6 to 3.2 times when compared with the Random submission and from 1.2 to 2.3 times when compared with the greedy reordering scheme. This means that our approach is successfully increasing GPU utilization.

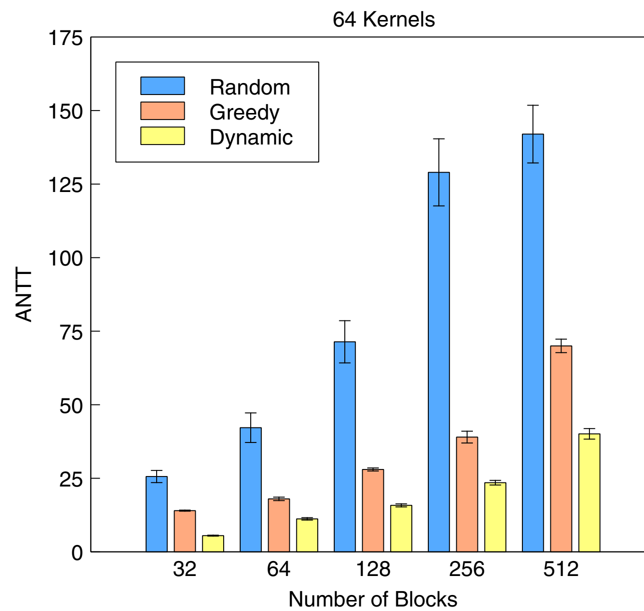

FIGURE 7 ANTT for 64 kernels on TITAN X

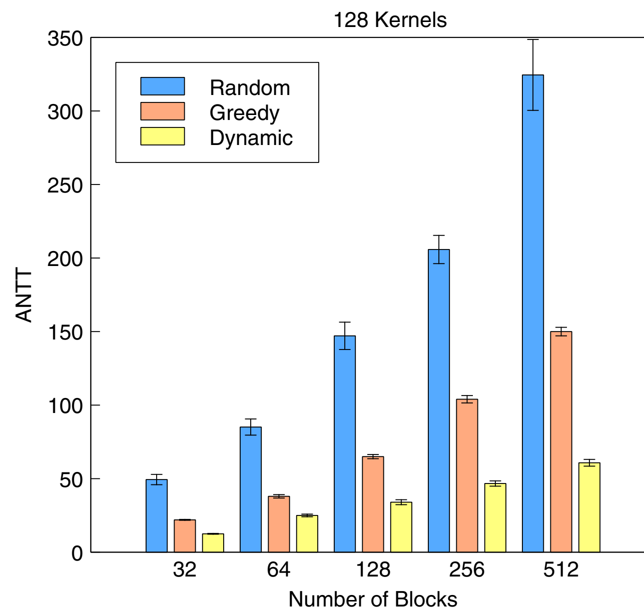

FIGURE 8 ANTT for 128 kernels on TITAN X

We illustrate this utilization in Figure 10. This figure shows the CUDA Profiler results for a part of the execution of $N = 128$ kernels with $NB = 32$. Figure 10A shows the Random execution and Figure 10B shows the Dynamic execution. We can observe that the Random execution allows at most 10 kernels to run concurrently while Dynamic execution allows at most 24 concurrent kernels and a consequent increase in the resource usage.

6.6 | Overhead

The kernel reordering is executed before the kernels are submitted to the GPU. Nevertheless, since the reordering computation involves solving a combinatorial optimization problem, we assess the overhead of the reordering algorithm.

The overhead is computed by comparing the execution time of the reordering algorithm with the absolute gain in the average turnaround time of the kernels (without normalization). We calculate the overhead as the ratio between the execution time of the proposed reordering algorithm with the gain.

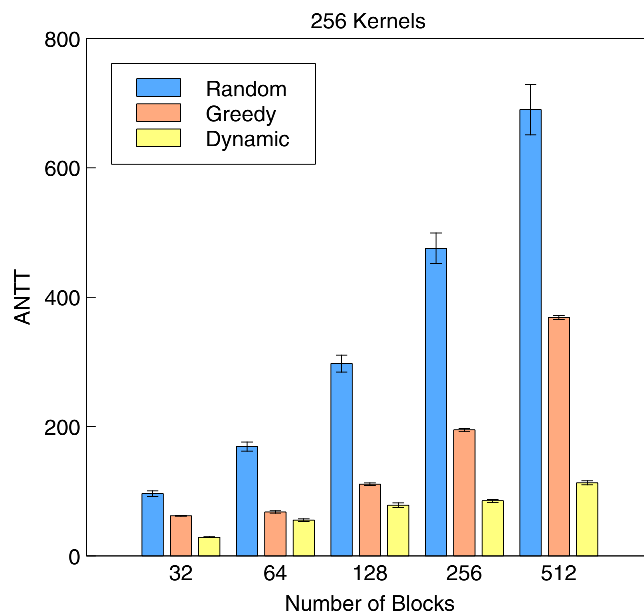

FIGURE 9 ANTT for 256 kernels on TITAN X

TABLE 3 STP results for different values of *N* and *NB*, and Random and Dynamic submissions

	STP							
	32 Kernels		64 Kernels		128 Kernels		256 Kernels	
NB	Random	Dynamic	Random	Dynamic	Random	Dynamic	Random	Dynamic
32	10.5	18.6	13.8	28.5	16.7	41.0	19.7	55.7
64	8.2	15.6	9.5	21.8	11.5	30.8	12.8	41.5
128	6.1	12.1	7.0	16.7	8.3	23.2	9.3	29.9
256	5.3	9.4	5.5	13.4	6.5	17.0	7.5	22.3
512	4.3	7.0	5.5	10.0	6.1	13.6	6.6	16.3

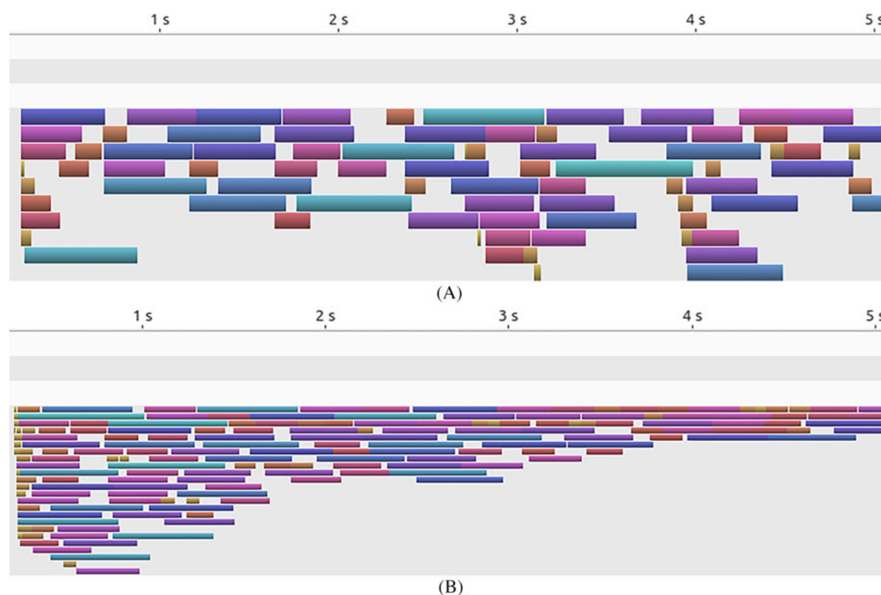

FIGURE 10 Execution profile of a set with 128 kernels. A, Execution using the Random submission scheme. B, Execution using our reordering approach

Table 4 shows, for the execution of 256 kernels, the average turnaround times (in milliseconds) using Random and Dynamic reordering, the absolute gains obtained by Dynamic, the processing time of kernel reordering, and the overhead of this computation compared with the gain obtained.

TABLE 4 Average turnaround time (without normalization) of Random and Dynamic, with the absolute gains of Dynamic, the execution time of kernel reordering, and its overhead for 256 kernels execution

NB	Random	Dynamic	Gain	Reordering Time	Overhead
32	8014.3	4750.7	3263.6	8260.3	2.5
64	15091.3	8559.3	6531.9	8772.4	1.3
128	28746.6	16077.3	12669.3	10338.2	0.8
256	56529.7	33319.8	23209.8	9237.5	0.39
512	108929.2	68426.8	40502.3	10409.1	0.25

We can observe in this table that the overhead decreases with the increase in the number of blocks. This occurs because the small amount of computation leads to smaller gains in the average turnaround time. For the other experiments with the different number of kernels, we observed a similar behavior. The largest overheads were found for small numbers of kernels and blocks. For the larger number of kernels, the execution time of the dynamic programming algorithm increases since the time complexity is proportional to the number of kernels. However, in practice, the overhead of solving the knapsack problems becomes small when compared with the reduction in the average turnaround time obtained with the resulting reordering.

6.7 | Portability

Our reordering scheme is a software solution that can be applied to any NVIDIA GPU architecture that uses the Hyper-Q technology. However, it is important to notice that our reordering scheme requires different estimated execution times for each architecture.

We report on Tables 5 to 9 the ANTT and STP results for the K40 GPU. As shown, our solution still achieves reductions in ANTT and improvements in STP when compared with the Random submissions.

For the kernels from the Rodinia benchmark suite, Table 5 shows that we achieved up to 77% improvement in ANTT and STP was improved 1.1 times. For the synthetic kernels, we obtained substantial reductions. From 31% to 73% of reduction in ANTT, STP was improved from 1.1 to 3.2 times. These results confirm that our reordering scheme is able to provide substantial gains in the Kepler architecture as well.

TABLE 5 ANTT and STP results for the Rodinia kernels on K40

	ANTT		STP	
	Random	Dynamic	Random	Dynamic
Rodinia	86.4	19.39	3.65	4.36

TABLE 6 ANTT and STP results for the synthetic kernel sets with 32 kernels on K40

	ANTT		STP	
NB	Random	Dynamic	Random	Dynamic
32	17	8.5	6.2	8.7
64	18	10	5.0	7.9
128	28	17	4.5	6.9
256	50	34	4.1	5.8
512	53	35	4.1	4.5

TABLE 7 ANTT and STP results for the synthetic kernel sets with 64 kernels on K40

	ANTT		STP	
NB	Random	Dynamic	Random	Dynamic
32	28	12	7.1	15
64	49	16	6.3	12
128	68	22	4.7	9.4
256	98	25	4.9	7.6
512	116	35	5.0	7.5

TABLE 8 ANTT and STP results for the synthetic kernel sets with 128 kernels on K40

	ANTT		STP	
NB	Random	Dynamic	Random	Dynamic
32	56	23	8.4	21
64	91	32	6.9	18
128	133	40	5.6	13
256	166	59	5.9	11
512	220	76	5.6	10

TABLE 9 ANTT and STP results for the synthetic kernel sets with 256 kernels on K40

	ANTT		STP	
NB	Random	Dynamic	Random	Dynamic
32	115	50	9.2	29
64	183	64	7.4	22
128	261	85	6.9	17
256	355	108	6.1	13
512	419	143	6.3	12

Comparing Kepler and Maxwell architectures, we observe some important differences. On Kepler, the shared memory and L1 data cache share 64 KB of memory space. On Maxwell, the shared memory is independent and has 96 KB. In addition, Maxwell has doubled the maximum number of active blocks per SM from 16 to 32. These differences should result in occupancy and concurrency opportunities improvements on Maxwell. Our reordering scheme takes advantage of these improved occupancy and concurrency opportunities of Maxwell and provided more pronounced gains when compared with the Kepler results.

7 | CONCLUSIONS

As the GPU hardware continues to evolve, offering an increasing amount of resources, it is likely that, in the future, the GPU will move toward a multiprogrammed device. In this scenario, sharing the GPU resources is essential to improve the computational throughput. In this work, we propose a reordering strategy that focuses on the order at which the kernels are submitted to execute on the GPU. Our idea is to use an optimization algorithm that solves a series of knapsack problems to find the kernels that maximize the GPU utilization. Every time there are available resources, the algorithm models the amount of resources as the knapsack capacity and attempts to fulfill the knapsack with kernels that take the most advantage of these available resources, favoring kernels with smaller execution time.

We present a series of experiments using two different GPU architectures. The experiments use real-world and synthetic applications with a different number of kernels and resource requirements. Our results show that the reordering provides gains in the average turnaround time of the kernels and in the system throughput compared to the Random kernels submission. The overhead of the reordering algorithm is negligible, compared with the gains in the average turnaround time. We also show that our approach provides similar gains in the Kepler and the Maxwell architecture.

In a future work, we intend to consider the NVIDIA execution model of waves of simultaneous blocks execution in our scheduling approach, instead of considering kernels as monolithic blocks that need to fit their resources into the SMs. We also intend to extend our wrapper to work like the NVIDIA Multi-Process Service (MPS), which handles the tasks from multiple applications, to take advantage of our reordering approach.

ACKNOWLEDGMENTS

This work was partially funded by CNPq, CAPES, and FAPERJ.

ORCID

Rommel A.Q. Cruz http://orcid.org/0000-0002-2270-5162

REFERENCES

1. Pai S, Thazhuthaveetil MJ, Govindarajan R. Improving GPGPU concurrency with elastic kernels. *ACM SIGPLAN Not*. 2013;48(4):407-418.
2. Adriaens JT, Compton K, Kim NS, Schulte MJ. The case for GPGPU spatial multitasking. Paper presented at: IEEE 18th International Symposium on High Performance Computer Architecture (HPCA), IEEE; 2012; New Orleans, LA.

3. Fermi compute architecture white paper. White Paper, NVIDIA Corporation; 2010.
4. Che S, Boyer M, Meng J, Tarjan D, Sheaffer JW, Lee S-H, Skadron K. Rodinia: a benchmark suite for heterogeneous computing. Paper presented at: IEEE International Symposium on Workload Characterization (IISWC), IEEE; 2009; Austin, TX.
5. Danalis A, Marin G, McCurdy C, et al. The scalable heterogeneous computing (shoc) benchmark suite. Paper presented at: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, ACM; 2010; New York, NY.
6. Stratton JA, Rodrigues C, Sung I-J, et al. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. Technical Report. Center for Reliable and High-Performance Computing. Vol. 127. 2012.
7. CUDA Profiler. http://docs.nvidia.com/cuda/profiler-users-guide/index.html
8. Guevara M, Gregg C, Hazelwood K, Skadron K. Enabling task parallelism in the CUDA scheduler. Workshop on Programming Models for Emerging Architectures. Vol. 9. Citeseer; 2009.
9. Peters H, Koper M, Luttenberger N. Efficiently using a CUDA-enabled GPU as shared resource. Paper presented at: IEEE 10th International Conference on Computer and Information Technology (CIT), IEEE; 2010; Bradford, UK.
10. Wang L, Huang M, El-Ghazawi T. Exploiting concurrent kernel execution on graphic processing units. Paper presented at: International Conference on High Performance Computing and Simulation (HPCS), IEEE; 2011; Istanbul, Turkey.
11. Gregg C, Dorn J, Hazelwood K, Skadron K. Fine-grained resource sharing for concurrent GPGPU kernels. Paper presented at: 4th USENIX Workshop on Hot Topics in Parallelism; 2012; Berkeley, CA.
12. Wang G, Lin Y, Yi W. Kernel fusion: an effective method for better power efficiency on multithreaded GPU. Paper presented at: IEEE/ACM Int'l Conference on Cyber, Physical and Social Computing (CPSCom) & Int'l Conference on Green Computing and Communications (GreenCom), IEEE; 2010; Hangzhou, China.
13. Zhong J, He B. High-throughput GPU kernel executions with dynamic slicing and scheduling. *IEEE Trans Parallel Distrib Syst*. 2014;25(6):1522-1532.
14. Ravi VT, Becchi M, Agrawal G, Chakradhar S. Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework. Paper presented at: Proceedings of the 20th International Symposium on High Performance Distributed Computing, ACM; 2011; New York, NY.
15. Tanasic I, Gelado I, Cabezas J, Ramirez A, Navarro N, Valero M. Enabling preemptive multiprogramming on GPUs. *ACM SIGARCH Comput Archi News*. 2014;42(3):193-204.
16. Park JJK, Park Y, Mahlke S. Chimera: collaborative preemption for multitasking on a shared GPU. Paper presented at: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ACM; 2015; New York, NY.
17. Liang Y, Huynh HP, Rupnow K, Goh RSM, Chen D. Efficient GPU spatial-temporal multitasking. *IEEE Transactions Parallel Distrib Syst*. 2015;26(3):748-760.
18. Wende F, Cordes F, Steinke T. On improving the performance of multi-threaded CUDA applications with concurrent kernel execution by kernel reordering. Paper presented at: Symposium on Application Accelerators in High Performance Computing (SAAHPC), IEEE; 2012; Chicago, IL.
19. Li T, Narayana VK, El-Ghazawi T. A power-aware symbiotic scheduling algorithm for concurrent GPU kernels. Paper presented at: The 21st IEEE International Conference on Parallel and Distributed Systems (ICPADS); 2015; Melbourne, Australia.
20. Lopez-Novoa U, Mendiburu A, Miguel-Alonso J. A survey of performance modeling and simulation techniques for accelerator-based computing. *IEEE Transactions Parallel Distrib Syst*. 2015;26(1):272-281.
21. Choi JH, Son DO, Kang SG, Kim JM, Lee H-H, Kim CH. An efficient scheduling scheme using estimated execution time for heterogeneous computing systems. *J Supercomput*. 2013;65(2):886-902. https://doi.org/10.1007/s11227-013-0870-6
22. Martello S, Toth P. *Knapsack Problems: Algorithms and Computer Implementations*. New York, NY: John Wiley & Sons; 1990.
23. Eyerman S, Eeckhout L. System-level performance metrics for multiprogram workloads. *IEEE Micro*. 2008;28(3):42-53.
24. Jiao Q, Lu M, Huynh HP, Mitra T. Improving GPGPU energy-efficiency through concurrent kernel execution and DVFS. Paper presented at: Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, IEEE; 2015; San Francisco, CA.
25. Awatramani M, Zambreno J, Rover D. Increasing GPU throughput using kernel interleaved thread block scheduling. Paper presented at: 31st International Conference on Computer Design (ICCD), IEEE; 2013; Asheville, NC.

How to cite this article: Cruz RAQ, Bentes C, Breder B, et al. Maximizing the GPU resource usage by reordering concurrent kernels submission. *Concurrency Computat Pract Exper*. 2019;31:e4409. https://doi.org/10.1002/cpe.4409