Future Generation Computer Systems 113 (2020) 528–540

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Using machine learning techniques to analyze the performance of concurrent kernel execution on GPUs

Pablo Carvalho [a,*], Esteban Clua [a], Aline Paes [a], Cristiana Bentes [b], Bruno Lopes [a], Lúcia Maria de A. Drummond [a]

[a] Instituto de Computação - Universidade Federal Fluminense, Niterói, Brazil
[b] Eng. de Sistemas e Computacao - Universidade do Estado do Rio de Janeiro, Rio de Janeiro, Brazil

ARTICLE INFO

Article history:
Received 13 May 2019
Received in revised form 8 July 2020
Accepted 14 July 2020
Available online 15 July 2020

Keywords:
GPU
Concurrency
Scheduling
Machine learning

ABSTRACT

Heterogeneous systems employing CPUs and GPUs are becoming increasingly popular in large-scale data centers and cloud environments. In these platforms, sharing a GPU across different applications is an important feature to improve hardware utilization and system throughput. However, under scenarios where GPUs are competitively shared, some challenges arise. The decision on the simultaneous execution of different kernels is made by the hardware and depends on the kernels resource requirements. Besides that, it is very difficult to understand all the hardware variables involved in the simultaneous execution decisions in order to describe a formal allocation method. In this work, we use machine learning techniques to understand how the resource requirements of the kernels from the most important GPU benchmarks impact their concurrent execution. We focus on making the machine learning algorithms capture the hidden patterns that make a kernel interfere in the execution of another one when they are submitted to run at the same time. The techniques analyzed were *k*-NN, Logistic Regression, Multilayer Perceptron and XGBoost (which obtained the best results) over the GPU benchmark suites, Rodinia, Parboil and SHOC. Our results showed that, from the features selected in the analysis, the number of blocks per grid, number of threads per block, and number of registers are the resource consumption features that most affect the performance of the concurrent execution.

1. Introduction

Graphics Processing Units (GPUs) have proven to be a powerful and efficient platform to accelerate a substantial class of compute-intensive applications. For this reason, many large scale data centers are based on heterogeneous architecture comprising multicore CPUs and GPUs to meet the requirements of high performance and data throughput. GPUs are also being used in computational clouds. Exploring GPUs in clouds, through GPU virtualization, allows physical devices to be logically decoupled from a computational node and shared by any application, resulting in monetary costs reduction, energy savings and more flexibility.

In this scenario, efficiently sharing a GPU across different applications is an indispensable feature. Recent GPUs introduced the concept of concurrent kernel execution, that enables different kernels to run simultaneously on the same GPU, sharing the GPU hardware resources. Concurrent kernel execution facilitates GPU virtualization and can improve hardware utilization and system throughput. The blocks of the concurrent kernels are dispatched to run on the Streaming Multiprocessors (SMs) and the warp scheduler arranges the order at which each warp will execute, with near zero context-switch overhead.

However, one key difficulty for concurrent kernel execution is that, in the GPU, the low-level sharing decisions are proprietary and strictly closed by GPU vendors. Consequently, GPU virtualization software has no control over the actual resource sharing. In a previous work, Carvalho et al. [1] showed the impact that kernel resource requirements have in concurrent execution for the kernels of the most important GPU benchmarks. The results showed that resource-hungry kernels on one resource may prevent concurrent execution. This study, however, did not explain how the resource requirements of the kernels have an effect on the performance of the concurrent execution.

In Fig. 1, we illustrate how difficult it is to understand and predict the execution interference of the kernels and what types of kernels could execute concurrently. Although the widely accepted idea is that kernels with complementary resource requirements would benefit from concurrent execution, sharing the GPU resources dynamically between the kernels is complex and hardware dependent. Fig. 1 shows the pairwise execution of three kernels with low, medium and high resource usage.

* Corresponding author.
E-mail addresses: pablocarvalho@id.uff.br (P. Carvalho), esteban@ic.uff.br (E. Clua), alinepaes@ic.uff.br (A. Paes), cris@eng.uerj.br (C. Bentes), bruno@ic.uff.br (B. Lopes), lucia@ic.uff.br (L.M. Drummond).

https://doi.org/10.1016/j.future.2020.07.038
0167-739X/

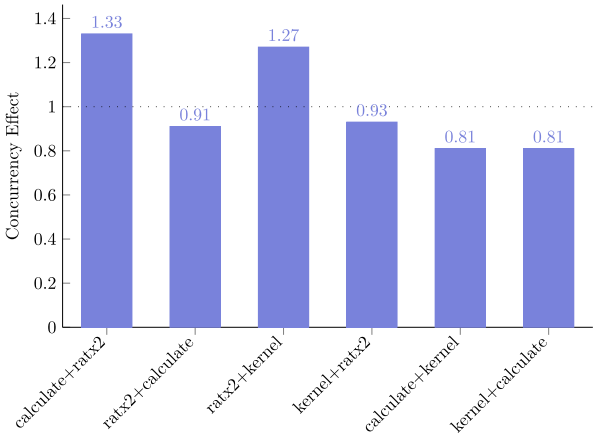


Fig. 1. Ratio of the execution time of kernels *calculate_temp* (from Hotsopt application in the Rodinia benchmarks suite), *kernel* (from Myocite application in Rodinia) and *ratx2_kernel* (from S3D application in the SHOC benchmark suite) running without concurrency over their co-execution with concurrency.

The kernels are from Rodinia and SHOC benchmark suites: *calculate_temp* (from Hotspot) that has high resource usage, *kernel* (from Myocyte) that has medium resource usage, and *ratx2_kernel* (from S3D) that has low resource usage. The *x*-axis shows all the combinations of the pairs of kernels considering the two possible submission orders. The *y*-axis describes the results of the ratio between their combined execution time without concurrency and with concurrency. This figure shows that it is not trivial to understand the relationship between the kernels when they execute concurrently. We can observe that, although *calculate_temp* is a resource-hungry kernel, it can execute concurrently with *ratx2_kernel* and they run 33% faster when executed concurrently. However, when *calculate_temp* executes concurrently with *kernel*, that presents low use of GPU resources, they perform around 20% worse than their execution without concurrency. The *ratx2_kernel* performs 33% better when executed concurrently with *calculate_temp* and 27% better when co-executed with *kernel*, but it depends on their submission order.

In this work, we performed an extensive analysis of the concurrent execution of the kernels from Rodinia, Parboil, and SHOC benchmarks to assess how their performance is affected by the concurrent execution. We use four machine learning techniques [2] to induce models capable of inferring if there is interference in concurrent execution of kernels, and also to classify the slowdown resulting from such interference. Furthermore, we rely on feature selection [3] and feature importance [4] techniques to understand how the resource usage of the kernels impacts the possible interference and the slowdown. We focus on the co-execution of two kernels to better understand their interference avoiding the explosive increase in the number of experiments.

The work has the following contributions: (1) An extensive experimental analysis of the concurrent execution of pairs of kernels from the most important GPU benchmark suites; (2) A machine learning study on the concurrent execution results with four different techniques to unveil how the kernels interfere each other in the concurrent execution; (3) A comparison of the machine techniques, *k*-Nearest Neighbors, Logistic Regression, Multilayer Perceptron and XGBoost in their ability to infer if there is interference in the concurrent execution, given the resource requirements. (4) A feature importance analysis to reveal the features that matters the most for the performance interference on the concurrent execution.

We performed a number of experiments on two different GPU architectures and found that a recently proposed ensemble technique, namely the XGBoost [5], achieves the best quantitative results in learning to distinguish whether or not a pair of kernels would execute concurrently and to distinguish whether the concurrent execution would cause slowdown. Furthermore, by analyzing the variables chosen by the feature selection method and the ones elicited as the most important to induce the ensemble model, we conclude that the number of blocks per grid is the most relevant feature to define if the kernels will execute concurrently and to influence the performance interference. The second most important feature, however, depends on the GPU architecture. For the GPU with more resources, the number of registers is key for the kernels interference. While, for the GPU with less computing resources but the same amount of registers, the number of threads per block is more relevant in the kernels interference.

The remainder of this paper is organized as follows. Section 2 introduces the GPU architecture, the concurrent execution environment and the framework we developed to ensure simultaneous resource requirements. Section 3 shows the machine learning pipeline used to classify the kernels interference in the concurrent execution. Section 4 presents our experimental results. Section 5 presents the previous work on concurrent execution and performance modeling on the GPU. Finally, Section 6 presents our conclusions and directions for future work.

2. GPU execution model

In this section, we use NVIDIA and CUDA terminology to describe the GPU architecture and execution model. The descriptions, though, also can be applied to other GPU vendors.

2.1. GPU architecture

The GPU architecture is designed for fine-grain massive parallel processing. It comprises a number of Streaming Multiprocessors (SMs). Each SM contains many compute cores and resources such as registers, shared memory and L1 cache. The CUDA programming model requires the programmer to define functions called *kernels* that will be offloaded to execute on the GPU. The kernel is executed with a number of threads that are grouped into *thread blocks*. Each block is dispatched by the hardware to be executed in one SM. Once a block is assigned to an SM, its threads are divided into *warps* by the scheduler. Each warp of threads executes the same instruction simultaneously on different data values.

2.2. Concurrent kernel execution

Current generations of GPUs support concurrent execution of kernels. To achieve this, NVIDIA introduced the concept of *streams*. CUDA streams allow the programmer to express execution independence. Kernels that belong to different streams do not depend on each other and can execute concurrently. The Hyper-Q technology proposed by NVIDIA provides 32 hardware work queues, where each stream can be assigned to a different queue. For more than 32 streams, false serialization can occur. NVIDIA also introduced the Multi-Process Service (MPS) [6] that enables kernels from different applications to share the GPU resources. The MPS server filters the work from different processes and submits to concurrent execution.

The software, however, has no control over how the thread blocks are dispatched to execute into SMs. The NVIDIA scheduling policy is proprietary and may change from architecture to architecture. Most likely, the hardware uses a *leftover* policy that assigns as many resources as possible for one kernel and then assigns the remaining resources to another kernel, if there are

sufficient leftover resources [7]. According to this policy, kernels submitted at the same time in different streams may still execute sequentially or may cause interference in each other execution. Inter-kernel interference can have a negative impact on the application execution time.

2.3. Submission framework

In this work, we explore kernel concurrent execution in future scenarios with high competition for GPUs as in heterogeneous large scale data centers. To study the potential performance interference in these scenarios, we implemented a framework to guarantee that the kernels are submitted together to prompt execution and the competition for GPU resources actually occurs. To submit the kernels together, we intercept the CUDA API calls that launch kernels and include synchronization primitives to postpone the calls until all the data has been transferred from the CPU memory to the GPU memory. Our framework is a dynamic library that is linked to each application but uses MPS to submit the kernels to the GPU. The concurrent execution, however, will depend on the hardware scheduler.

3. Machine learning pipeline

Machine Learning (ML) techniques focus on making computers learn how to act, without being explicitly programmed, relying on data and information about the world [2,8]. The data collected from observing the world is called *examples*. Usually, each example is described by a set of *features*, which encompass the descriptors of the examples.

Those techniques are divided into three main groups: supervised, unsupervised and reinforcement learning. Here, we address supervised techniques, which are suitable for handling tasks where the output feature is known. A supervised learning task can be presented as a classification task, in which the output feature lies on a set of categorical or discrete values, or as a regression task, when the output feature is a continuous value. In this work, we tackle a supervised, classification task, as our data is previously labeled and the output feature represents discrete classes of slowdown between a pair of kernel executions.

In order to induce the predictive model responsible for classifying how the kernels behave together, we follow the standard ML pipeline exhibited in Fig. 2. The pipeline starts by collecting the historical data generated from the task one wants to solve. In our case, the information regarding the hardware settings and the kernels becomes the features and each concurrent run data (number of registers, threads per block, blocks per grid, and shared memory) between a pair of kernels becomes an example.

Pre-processing and feature evaluation. Next, the data is transformed in order to make them more amenable to the ML classifier technique, using methods such as normalization and standardization. The dataset can also be transformed by a step of feature selection or dimensionality reduction. Feature selection methods aim at automatically discarding the features that are irrelevant, less important, or even harmful to the task at hand [9]. The kernel interference and slowdown problems that we tackle in this paper do not suffer from the curse of dimensionality, because of that, we did not employ any method for reducing the dimension of the dataset, such as PCA. Still, we rely on a feature selection method to get insights on which features are more important to the task, in order to have some explanation about the classifiers induced from the data.

We selected the Recursive Feature Elimination (RFE) technique [10]. RFE is a greedy optimization procedure that constructs a classifier, chooses either the best or the worst feature, and then repeats the process with the remaining features. This procedure repeats until all the features in the dataset are experimented. After that, they are ranked following the order they have been selected. The final subset of features is selected from this order according to a hyperparameter informed by the user that states how many features one should have in the final dataset.

Machine learning methods. The next step is to train the task represented by the dataset, using a machine learning algorithm. In this work, we compare four methods that represent different categories, namely: (a) an instance-based classification (*k*-NN), (b) a simple and a complex representative of function-based learning (Logistic Regression (LR) and Multi-layer Perceptron (MLP), respectively). While LR represents one function (the logistic function), MLP can be a composition of several functions, according to the number of its layers. Finally, but most importantly, we selected (c) an ensemble technique (XGBoost) that implements a combination of gradient boosted decision trees aiming at speeding up the learning process. XGBoost uses a more regularized model formalization to control overfitting. This is particularly important for the tasks we tackle here, as we have only a few variables that could lead us to overfitting the training data [5]. Our interest in experimenting with XGBoost is motivated by numerous tasks that this approach has solved since its development in 2016, ranging from disease classification to protein entry site prediction [11,12]. But we also would like to observe how simpler methods that induce individual classifiers (Logistic Regression and MLP) and even when there is no induced classifier at all (KNN) behave on the concurrency and interference tasks. Here, we briefly review each one of them.

***k*-NN:** (*k*-Nearest Neighbors [13]) is an instance based method, meaning that the classification is based on computing the distance (usually euclidean) between a pair of observations. Thus, for an instance x its classification will be the mode of its k nearest samples.

Logistic Regression [14]: uses the logistic sigmoid function to return a probability distribution which can then be mapped to a set of classes. A sigmoid function has the support $\mathbb{R} \rightarrow [0, 1]$ and is defined as $S(x) = \frac{1}{1+e^{-x}}$. The answer may be binary (e.g. yes/no), multinomial (e.g. mamal, reptile, amphibe) or ordinal (e.g. worst, average, best).

Multilayer Perceptron [15]: consists of a feed-forward neural network with at least one intermediate layer of neurons. The goal of including the intermediate layer is to describe the features by composing nonlinear functions. Thus, we can see the intermediate layer as computing a function $h = g(\mathbf{W}^T\mathbf{x} + \mathbf{c})$, where $\mathbf{x}$ represents the input of the layer, $\mathbf{c}$ are the biases and $\mathbf{W}$ are the weights of the layer. The training phase has as goal to find these weights by using an algorithm called backpropagation.

XGBoost [5]: (eXtreme Gradient Boosting) is an ensemble technique based on decision trees. Decision trees have the capability of unveiling an interaction structure which may lead to a comprehension on the core characteristics that influence the prediction. XGBoost greedily builds a set of trees and decides the final prediction to an instance by summing out the predictions from each tree. It does not allow full optimization in each round of the training, but, instead, it uses a regularized model to avoid overfitting.

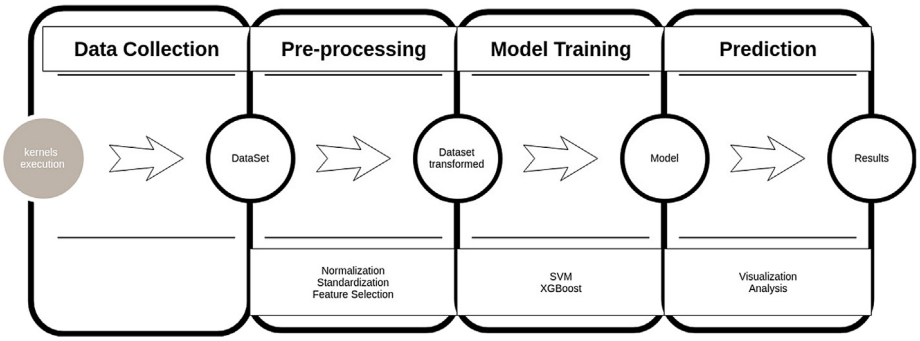


Fig. 2. Machine learning pipeline.

Evaluating the models. The evaluation of the resulting model is guided by matching the class predicted by the model to the class observed in the test samples. We use the *confusion matrix* to compute the predictive measures over the induced models. Such a matrix presents the amount of instances on the real classification over the predicted values. To simplify, in a binary case, the confusion matrix exhibits in its main diagonal the amount of examples that the model is correctly classifying: the true positives (TP), which are the positive instances indeed classified as positive, and the true negatives (TN), the negative instances classified by the model as negative. The other diagonal presents the wrong cases: the false positives (FP), which are the negative instances wrongly classified as positive, and the false negatives (FN), the amount of positive instances classified as negative.

From the confusion matrix, we compute the metrics below, in order to measure the predictive power of the model:

Accuracy: the number of correctly classified instances divided by the total amount of instances: $\frac{TP+TN}{TP+TN+FP+FN}$

Recall is defined as the number of positive instances correctly classified as positive (TP) divided by the amount of positive instances ($TP + FN$).

Precision is the number of positive instances correctly classified as positive (TP) divided by the amount of instances classified by the model as positive, both the right and the wrong cases ($TP + FP$).

Kappa index: an agreement measure used in nominal scales of how much the predictions are far from the expected classification, defined as $\kappa = \frac{\Pr(o)-\Pr(e)}{1-\Pr(e)}$ where $\Pr(o)$ denotes the observed agreement and $\Pr(e)$ is the expected agreement; it may be computed using the confusion matrix; we consider $kappa \leq 0$ as no agreement, $0.01 \leq kappa \leq 0.20$ as none to slight, $0.21 \leq kappa \leq 0.40$ as fair agreement, $0.41 \leq kappa \leq 0.60$ as moderate, $0.61 \leq kappa \leq 0.80$ as substantial, and $0.81 \leq kappa \leq 1.00$ as almost perfect agreement [16].

To correctly evaluate the generalization capability of the model, we rely on a *k-fold cross-validation procedure*. To that, the set of instances is divided into k sets (called as folds) of (approximately) the same size mutually excluded. The learning algorithm runs k times where at each iteration one of the k folds is used to test the model and the other ones are put together into a training set, which are used to induce the model. Then, the metric chosen to evaluate the model is computed as the average over each test fold. In order to maintain the distribution of examples per class within the folds, we can resort to the stratified cross-validation procedure, to have the proportional number of examples in each fold according to the original dataset.

We assess the statistical significance of the model using parametric and non-parametric statistical tests.

Table 1
GPUs specifications.

	Tesla P100	RTX 2080
Number of cores	3584	2944
RAM	16 GB	8G
Memory bandwidth	732 GB/s	448 GB/s
Capability	6.0	7.5
Number of SMs	56	46
Shared Memory per SM	64K	48K
Number of registers per block	64K	64K
Max number of threads per SM	2048	1024
Max thread blocks per SM	32	16
Max registers per thread	255	255
Maximum thread block size	1024	1024
Architecture	Pascal	Turing

4. Experimental tests

This section describes our experimental setup, the machine learning results, and the analysis of the concurrent execution based on the machine learning output.

4.1. Experimental setup

Platform. The experiments were performed on two different GPUs: a GPU Tesla P100-SXM2 based on the Pascal architecture, and a GPU RTX 2080 based on the Turing architecture. Table 1 shows the devices specifications. The kernels were compiled with NVCC CUDA version 8.0. The executions used NVIDIA driver version 384.145. Profiling information was obtained using the NVIDIA command-line profiler *nvprof*. The timing results were obtained as the average of 5 executions for each experiment. The execution results for each GPU were used as training and test datasets for that GPU.

Benchmarks. We used kernels from the three main GPU benchmark suites, Rodinia [17], Parboil [18] and SHOC [19]. We focus here on the pairwise execution of kernels. We selected 60 kernels from the benchmarks and executed all possible 3600 permutations of these kernels.[1] To execute more than 2 kernels concurrently, we would have to consider all possible permutations of the kernels which would lead to an explosive number of experiments and make our study unfeasible. Therefore, we decided to limit our experiments to pairwise co-execution as in [20,21].

The kernels used were selected according to the kernel categorization scheme proposed in [1]. This previous work divides

[1] The 3600 permutations consider repetitions since we also tested the concurrent execution of two instances of the same kernel.

Table 2
Applications selected from each benchmark suite.

Parboil	Rodinia	SHOC
MRI-GRIDDING	BP	QTC
HISTO	LUD	S3D
LBM	SRADv2	GEMM
MRI-Q	PFF	SCAN
SAD	DWT2D	REDUCTION
SGEMM	BPTREE	SPMV
SPMV	GAUSSIAN	STENCIL2D
STENCIL	HOTSPOT	FFT
TPACF	HUFFMAN	SORT
	HYBRIDSORT	
	MYOCYTE	
	PFN	

the kernels from Rodinia, Parboil and SHOC into four categories. Kernels in the same category have similar behavior in terms of resource requirements (computation type, integer or float, usage of memory hierarchy, efficiency and hardware occupancy). Based on this categorization study, we selected 15 representative kernels from each category from the applications described in Table 2 in order to guarantee a balanced presence of kernels with different resource requirements in our study.

Although the kernels are submitted together to execute, they have to be submitted to the GPU driver one at a time. In this matter, we tested for K_i and K_j co-execution the two possible submission orders (K_i, K_j) and (K_j, K_i). The order that they are submitted has effected their concurrent execution due to the leftover scheduling policy of the GPU. Given the huge amount of kernels combinations, the applications were executed with the standard input data provided with each benchmark.

Metrics: We measure the concurrent effect (CE) on the execution of the pairs of kernels (K_i, K_j) as the ratio between the sum of their standalone sequential execution times (when they are executed one after another without concurrency) and the time they take to execute concurrently. The concurrent effect is presented in Eq. (1), where T_{K_i} is the execution time of the first kernel, T_{K_j} is the execution time of the second kernel, and $T_{(K_i,K_j)}$ is the execution time of the concurrent execution of (K_i, K_j). When CE < 1, the performance of (K_i, K_j) concurrent execution is worse than the performance of executing them non-concurrently. This means that the performance interference on their concurrent execution degrade their performance to the point that it does not worth to submit them to execute at the same time. When CE $>$ 1, the concurrent execution provides performance gains.

$$CE = \frac{T_{K_i} + T_{K_j}}{T_{(K_i,K_j)}} \tag{1}$$

Table 3 shows for the two GPUs the number of pairs of kernels that were executed concurrently with CE < 1 (slowdown), the number of pairs of kernels that executed concurrently with CE > 1 (performance gain) and the number of pairs of kernels in which the probability of concurrent execution is less than 80%. On P100, all possible 3600 permutations of pairs of kernels were tested, but on RTX-2080 some pairs of kernels were not able to execute due to some hardware configuration problems, and 3364 pairs of kernels were tested. The pairs of kernels (K_i, K_j) with low probability of executing concurrently can be distinguished in two situations: (1) K_i is very small when compared to K_j, and the time it takes for the GPU to launch K_j, K_i has already finished; and (2) Both K_i and K_j are resource hungry kernels and the GPU could not launch them together. The first case is more common in the 3600 pairs of kernels, that is why, on the P100 GPU, 2173 pairs of kernels presented low probability of executing concurrently, while, on the RTX-2080 GPU, only 818 kernels presented low probability of execution. On the smaller and slower GPU, the small kernels have a higher probability of presenting some concurrency with the other kernel. Nevertheless, we could observe that for both GPUs more than a third of the pair of kernels have performance advantages of using concurrency. This corroborates the importance of creating models that identifies these cases.

Table 3
Concurrent effect (CE) results for all pairs of kernels permutations on P100 and RTX-2080.

Execution	# of pairs	
	P100	RTX-2080
Concurrency with CE < 1	792	1147
Concurrency with CE > 1	635	1399
Low probability of concurrency	2173	818

4.2. Machine learning results

Considering that there are pairs of kernels taking advantage of the concurrent execution, while others do not, we would like to obtain an automatic procedure to decide when pairs of kernels should run together. For that, we rely on machine learning techniques. Here, we present the results[2] obtained from following the pipeline presented in the last section.

4.2.1. Dataset configuration

The curated dataset has four resource variables for each kernel, namely *blocks per grid, threads per block, number of registers and shared memory*. We selected these features because they are exposed to the developer before the kernel execution, and they comprise the SM static resource usage that has impact on the number of thread blocks that can be executed concurrently. Each one of these variables becomes a feature in the dataset that is going to feed the ML tasks. Since our analysis focuses on a *pair of kernels*, we have a total of *eight* features $F_{i,k}$ to describe a ML example, where i is the index of the feature and k is the index of the kernel. In this way, the curated dataset has eight features, namely Features $= \{x_{11}, x_{21}, x_{31}, x_{41}, x_{12}, x_{22}, x_{32}, x_{42}\}$. Concerning the output, we focus on two classification tasks. The first task is to determine if the pair of kernels can execute concurrently. In a positive case the output is *yes*, otherwise the output is *no*. The second one aims at interference, *i.e.*, deciding if there is either a positive or a negative effect when running two kernels concurrently, i.e., we want to determine the concurrency effect (CE). The output is also binary: it is either *yes*, indicating a positive effect, or *no*, indicating a negative effect. In the curated dataset that is going to feed the machine learning methods, both of those outputs are represented as a binary feature y.

In order to induce each classifier and observe its generalization capabilities, we performed a stratified 10-fold cross validation procedure. With that, the dataset is divided into 10 disjunctive sets and we run the classifier 10 times, where at each time one of the folds assumes the role of test set and the rest of them compose the training set, iteratively. The predictive results are computed as an average of the 10 but considering only the results of the test set of each run.

[2] All training and test datasets, together with the source-codes are available at Github on: https://github.com/pablocarvalho/ml-gpu-kernel.

Table 4
Successful number of pair executions for both GPUs.

Successful tries of 5	P100	RTX-2080
	# of kernels	# of kernels
0	1277	339
1	370	159
2	253	125
3	273	195
4	426	560
5	1001	1986

Table 5
The hyperparameters used to train the ML models.

KNN	neighbors = 5, weights = uniform
MLP	hidden_layer_sizes = (5, 2), activation = relu, solver = 'lbfgs' learning_rate = constant, epochs = 200, early_stopping = True
LR	penalty = l2
XGB	max_depth = 3, trees = 100, learning_rate = 0.1, gamma = 0

4.2.2. Learning results for the concurrency problem

We target first on inducing the models to decide whether a pair of kernels would execute concurrently or not, called here *Task 1*. We selected a number of pairs of kernels as the ones with the highest probability of running concurrently making the value of the feature *y* to them be *yes*. The other ones were considered with low or no probability of concurrency, making their class to be defined as *no*. Table 4 shows the distribution of the pair of kernels on both P100 and RTX-2080 according to the successful attempts of running them concurrently. The examples selected as belonging to the class *yes* are the ones that have four or five successful tries, while the examples with the class *no* are the rest of them.

Then, we induced the classifier for the first task (responsible for distinguishing whether or not a pair of kernels can execute concurrently), using the eight features of the curated dataset. As stated in Section 3, we experimented with four machine learning techniques: Multilayer Perceptron (MLP), *k*-NN, Logistic Regression (LR), and XGBoost. Table 5 exhibits the default values of the main hyper parameters used to train the models. Before deciding to proceed with the training with such default values, we experimented a number of other values using the Grid Search technique [22]. Those preliminary results presented no significant difference compared to the ones obtained with the default values.

Tables 6 and 7 present the accuracy, precision, recall and kappa results for Task 1 on P100 and RTX-2080, respectively, considering the curated dataset. The bold values are the best ones for each metric and also the significantly better than the rest of them. We applied statistical tests (T-Test and Wilcoxon Test [23]) to support the significance of these results. The first conclusion we can state from them is that XGBoost achieves the best results on almost all metrics. This is expected, as ensemble methods are known to reach better results than when learning the models individually and XGBoost is the current *defacto* choice of ensemble methods for classification tasks. One can also see that kappa index is consistently better for XGBoost than to the rest of the classifiers in both cases.

The only exception is the value of recall when the kernels run on RTX-2080. In this particular case, the logistic regression performs surprisingly well, with almost no positive test examples incorrectly classified as negative. In fact, with P100, some classifiers have a very disappointing performance: logistic regression performs extremely badly for the positive examples and only achieves an accuracy of 0.6014 because it correctly classifies the negative examples. Similarly, MLP incorrectly classifies several positive examples as negative, yielding a very low value of recall.

Table 6
Predictive results for the concurrency problem on P100 with all the features. The values in bold indicate the statistically significant best results.

	Accuracy	Precision	Recall	Kappa
MLP	0.7295	0.7299	0.1366	0.1162
K-NN	0.7295	0.6836	0.5887	0.4202
LR	0.6014	0.2533	0.0035	−0.0029
XGB	**0.8164**	**0.8113**	**0.7001**	**0.6068**

Table 7
Predictive results for the concurrency problem on RTX-2080 using all the features.

	Accuracy	Precision	Recall	Kappa
MLP	0.7642	0.7832	0.9532	0.1630
K-NN	0.7663	**0.8260**	0.8759	0.3215
LR	0.7574	0.7599	**0.9933**	0.0243
XGB	**0.8008**	**0.8238**	0.9375	**0.3657**

In this way, MLP would say that two kernels cannot run concurrently when they actually can, yielding a very cautious classifier. KNN is consistent on the precision and recall metrics, but it still worse than XGBoost. On the other hand, when the kernels run on RTX-2080, all the classifiers achieve a quite good performance. KNN, for example, has a precision as high as XGBoost and, as said before, the recall of Logistic regression is even higher than the one achieved by XGBoost. As explained in Section 4.1, RTX-2080 presented more pairs of kernels with high probability of executing concurrently, therefore, the classifiers solve the concurrency problem in RTX-2080 easier than on P100.

Figs. 3 and 4 exhibit the Precision-Recall curves for the concurrency results on P100 and RTX-2080, respectively. This type of curve aims at showing the trade-off between these two metrics considering different classification thresholds. The higher the area under the curve, the better is the result since this is the case where both metrics have higher values. From both curves, it is important to notice that the RTX-2080 achieves more consistent results that are less dependent on the threshold when compared to P100. In P100, the threshold that makes the model to achieve higher recall values makes the precision values to be lower. In other words, lowering the threshold to achieve more results also introduced more false positives, making the precision decrease. It is possible to observe that by looking at the end of the curves, when the values of recall are the best possible according to a certain threshold. One can also see that the curves for all the classifiers running with RTX-2080 data are closer to each other, ascertaining our previous observation that the different methods had less trouble to find good classifiers here.

4.2.3. Learning results for the interference problem (concurrency effect)

Given that 1427 and 2546 kernel pairs have a high probability (at least 80%) of concurrent execution, on P100 and RTX-2080, respectively, the next experiment consisted in inducing a classifier for the interference problem, called *Task 2*, computed from the concurrency effect (CE). To that, for each pair (K_i, K_j), if CE $<$ 1, the interference is such that the concurrent execution causes a slowdown in the execution of the kernel when compared to their standalone executions. This case is called here as a *negative* effect. On the other hand, if CE $>$ 1, this means that there is no interference, which we can call a *positive* effect. Inducing a classifier for CE is a different problem from deciding whether or not a kernel pair would run concurrently, as the variables that could impact the CE may be others.

The predictive results are disposed in Tables 8 and 9, for P100 and RTX-2080, respectively. Once again, XGBoost reached the best values for accuracy, precision, recall and kappa. In comparison

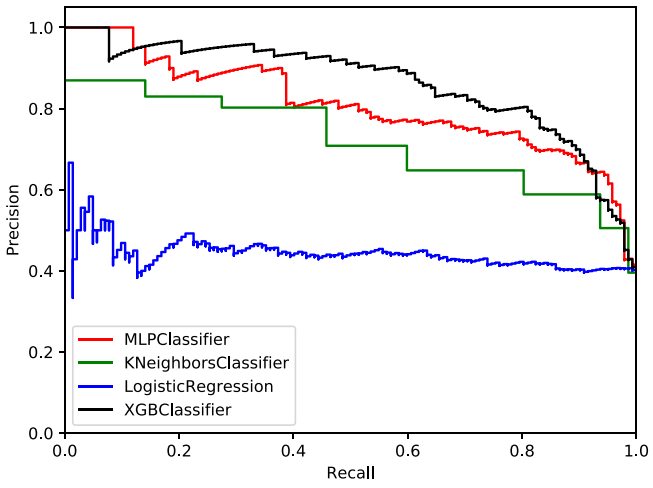


Fig. 3. Comparing the four classifiers for the concurrency problem (P100).

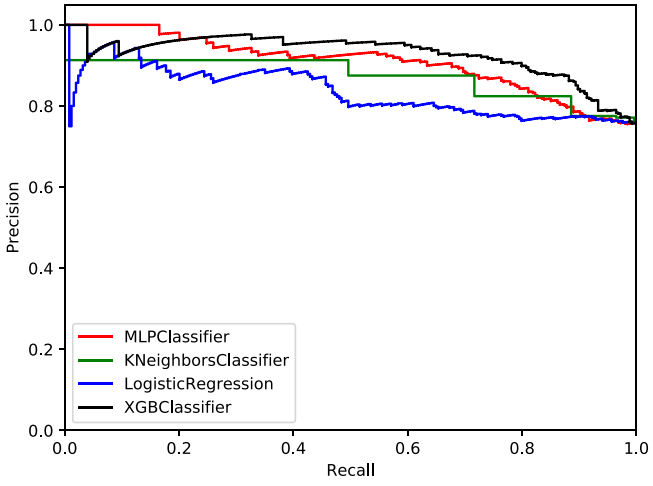


Fig. 4. Comparing the four classifiers for the concurrency problem (RTX-2080).

to the prior experiment, the CE classifier achieves lower metrics results, since the concurrency effect is a more difficult task to find a pattern. Similarly to the behavior of Logistic Regression classifier in Task 1 and RTX-2080, here its recall value is also as high as XGBoost, with a slight small difference between them. Still, its kappa index is quite low, making it not a good classifier to this problem, even though it is good at not mistake the positive examples as negative ones (it has a low value of false negative examples.)

Figs. 5 and 6 show the Precision-Recall curves for the interference task, considering the P100 and RTX-2080 architectures, respectively. Here we notice a more stable behavior of the classifiers when comparing to the previous concurrency tasks. The decaying of the precision curve is more smoothly with the lowering of the threshold to achieve more results that, consequently, improves the recall.

4.3. Concurrency and interference analysis

Most of our ML models are not interpretable, in the sense that their predictions are neither self-explainable nor their final models are easily understood by a human-being.

This is particularly the case of the classifier that has reached the best results, namely XGBoost. The successful achievements of this method come with a disadvantage: as it is an ensemble method, XGBoost lacks a direct interpretation of their results. The collection of one hundred trees induced by the method cannot be merged in a reasonable way.

Evaluating the correlation between all of the features, Fig. 7 presents heat maps of correlation for all of the features in the four studied scenarios. The correlation is very low in all cases

Table 8
Predictive results of the Interference Problem (Task 2) on P100 using all the features.

	Accuracy	Precision	Recall	Kappa
MLP	0.5781	0.5370	0.3858	0.1213
K-NN	0.6279	0.5833	0.5703	0.2448
LR	0.5726	0.5324	0.3149	0.0976
XGB	**0.6889**	**0.6723**	**0.5876**	**0.3623**

Table 9
Predictive results of the Interference Problem (Task 2) on RTX-2080 using all the features.

	Accuracy	Precision	Recall	Kappa
MLP	0.5746	0.6187	0.6339	0.1355
K-NN	0.6426	0.6645	0.7055	0.2732
LR	0.5491	0.5645	0.7813	0.0495
XGB	**0.7133**	**0.7165**	**0.7927**	**0.4140**

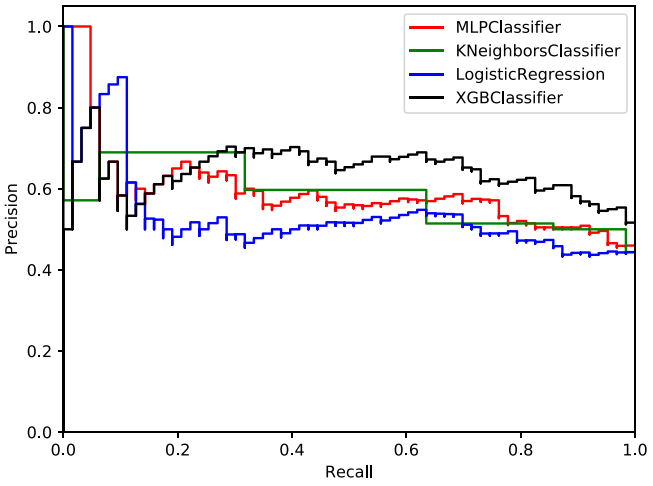


Fig. 5. Comparing the four classifiers for Interference (P100).

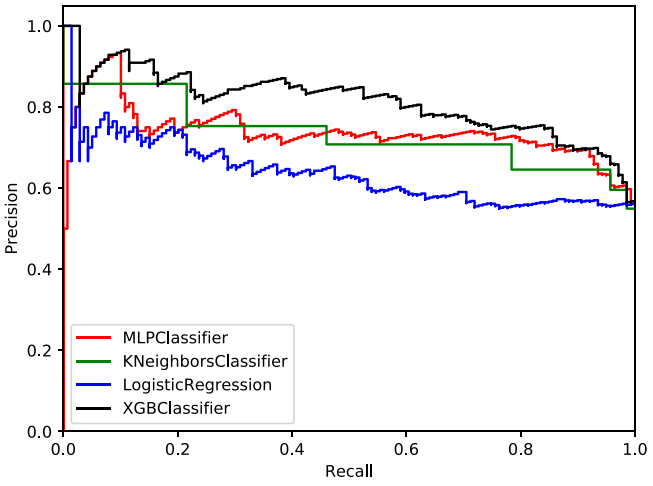


Fig. 6. Comparing the four classifiers for Interference (RTX-2080).

Table 10
Predictive results of the four classifiers when using the RFE selected variables *blocks per grid*, and *threads per block* for concurrency on P100.

	Accuracy	Precision	Recall	Kappa
MLP	0.6067	0.2077	0.1617	0.05513
K-NN	0.7094	0.6804	0.5017	0.3754
LR	0.6008	0.1083	0.0021	0.0004
XGB	**0.8089**	**0.8264**	**0.6559**	**0.5918**

Table 11
Predictive results of the four classifiers when using the RFE selected variables *blocks per grid*, and *threads per block* for concurrency on RTX-2080.

	Accuracy	Precision	Recall	Kappa
MLP	**0.7919**	0.8195	0.9301	0.3399
K-NN	**0.7904**	**0.8393**	0.8947	**0.3870**
LR	0.7568	0.7595	**0.9933**	0.02051
XGB	**0.7922**	0.8188	0.9317	0.3387

which discharges the usage of techniques that requires correlated variables such as Principal Component Analysis (PCA) [24].

As another alternative to alleviate this problem, we run two analysis to verify which features are mostly impacting the results: one based on a feature selection strategy, namely the Recursive Feature Elimination (RFE), and the other based on computing the feature importance. RFE requires a classifier to select which features are less influencing the results of that particular classifier. As XGBoost achieved the best overall results as pointed out in the previous sections, we also employ it as the inner classifier of RFE. The feature importance is measured according to the number of times each feature was used to split the data, weighted by the squared improvement to the model as a result of each split, averaged over all the trees [25].

Regarding first RFE, the concurrency problem (Task 1) and P100, it elicits the *blocks per grid*, and *threads per block* for each kernel, making a total of four features.

By analyzing the behavior of these resources, we can verify that the *number of blocks per grid* of both kernels is an indisputable factor in the decision of the concurrent execution and also on the co-execution interference. According to the leftover policy, a kernel with a great number of blocks, may not leave any free space on the SMs for the computation of other kernel blocks, resulting in a waiting time that would make the kernels execute without concurrency or increase the co-execution interference.

However, we can also observe from the results presented in Table 10 and Table 11 that we were not able to improve the predictive results of almost any metric when running with only the features selected by RFE. Thus, although they are pointed out as the most relevant ones by the method, the others are also useful to the final prediction. The only exception is the precision value of XGBoost, although they are not statistically different, according to the statistical tests (see).

The features that RFE automatically selected as the most relevant differ from the ones presented by Ravi *et. al.* [26], which also aimed at understanding the most relevant features to the concurrency problem but without relying on Machine Learning. There, they heuristically selected the *number of blocks per grid*, the *number of threads per block*, and the *amount of shared memory* used for every kernel involved in a concurrency situation. RFE selected only the *number of threads per block* in common with that previous work. Thus, to verify whether or not the variables selected with RFE match the performance of the experiments we conducted here, we also run the same machine learning techniques with the variables selected in [26]. The results related to the concurrency problem are presented in Tables 12 and 13 to P100 and RTX-2080, respectively. We can observe that the results

Table 12
Predictive results for concurrency of the four classifiers when using Ravi et. al. selected variables on P100.

	Accuracy	Precision	Recall	Kappa
MLP	0.7333	0.6951	0.5858	0.4274
K-NN	0.7817	0.7375	0.6979	0.5394
LR	0.6036	0.0000	0.0000	0.0000
XGB	**0.8053**	**0.8054**	0.6720	**0.5809**

Table 13
Predictive results for concurrency of the four classifiers when using Ravi et. al. selected variables on RTX-2080.

	Accuracy	Precision	Recall	Kappa
MLP	0.7767	0.8193	0.9045	0.3187
K-NN	0.7832	**0.8335**	0.8920	**0.3635**
LR	0.7568	0.7593	**0.9937**	0.0193
XGB	**0.7946**	0.8215	0.9313	0.3485

Table 14
Predictive results of the four classifiers when using the RFE selected variables for Interference on P100.

	Accuracy	Precision	Recall	Kappa
MLP	0.5823	0.5282	**0.5840**	0.1629
K-NN	0.6238	0.5812	0.5592	0.2356
LR	0.5816	0.5537	0.3006	0.1123
XGB	**0.6882**	**0.6714**	**0.5861**	**0.3607**

Table 15
Predictive results of the four classifiers when using the RFE selected variables for Interference on RTX-2080.

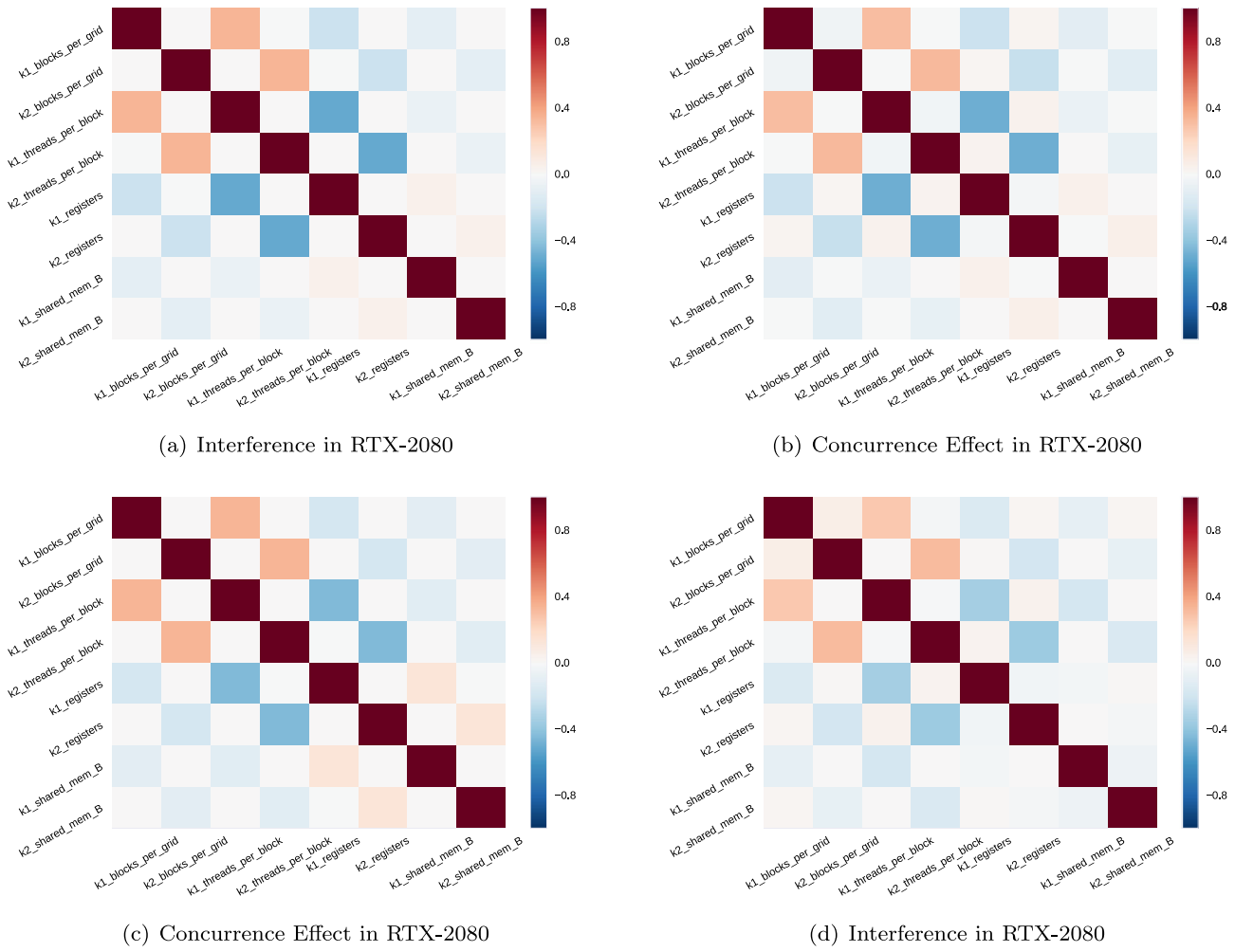
	Accuracy	Precision	Recall	Kappa
MLP	0.6819	0.6994	0.7384	0.3536
K-NN	0.6559	0.6744	0.7220	0.2998
LR	0.5491	0.5637	**0.7906**	0.0475
XGB	**0.7149**	**0.7163**	**0.7970**	**0.4169**

match the values achieved with the variables selected by RFE while they are still lower than when using all the variables.

Now, focusing on the interference problem (see Table 14 and Table 15), RFE selected the following variables for the interference problem and P100: *blocks per grid*, *threads per block* and *number of registers*, all of them for both kernels, making a total of six features. Here, only the shared memory is marked as not relevant. Once again, building the classifiers with only the selected variables have not improved the results, except for a significant improvement of the MLP recall. When improving the recall, the number of false negatives is decreased. For this specific classifier, it may be the case that the amount of shared memory makes the model to return more results, increasing the number of false negatives with this resource.

The number of threads per block and the number of registers required per thread appeared as important interference features. When the kernels execute concurrently, some blocks can share the resources of the same SM, including the register file. The number of threads per block also can have an impact in register usage, since the register file is divided among all the threads that are executing in the SM. On the other hand, when a kernel does not have enough threads per block to occupy the SM resources, other threads can use these resources, improving the GPU throughput and reducing the kernels interference .

We have also experimented the variables elicited in the work of Ravi *et. al.* [26] with the interference task, as presented in Tables 16 and 17 regarding P100 and RTX-2080, respectively. Once again the results are very similar to the ones we achieved with RFE. This similarity also provides more evidence that although there are variables that seem more important than the others, by using all of them we can achieve better predictive results.



(a) Interference in RTX-2080

(b) Concurrence Effect in RTX-2080

(c) Concurrence Effect in RTX-2080

(d) Interference in RTX-2080

Fig. 7. Features correlation heat maps.

Table 16
Predictive results for interference of the four classifiers when using Ravi et. al. selected variables on P100.

	Accuracy	Precision	Recall	Kappa
MLP	0.6645	0.6216	0.6350	0.3224
K-NN	**0.7063**	**0.6695**	**0.6740**	**0.4059**
LR	0.5515	0.4897	0.1431	0.0237
XGB	0.6664	0.6379	0.5780	0.3181

Table 17
Predictive results of the four classifiers when using the RFE selected variables *blocks per grid*, and *threads per block* for concurrency on RTX-2080.

	Accuracy	Precision	Recall	Kappa
MLP	0.6890	**0.7025**	0.7555	0.3662
K-NN	0.6811	0.6985	0.7398	0.3516
LR	0.5534	0.5606	**0.8635**	0.0413
XGB	**0.7046**	**0.7069**	0.7920	**0.3953**

Importance of the selected features to XGBoost. Figs. 8, 9, 10, and 11 show the bar-chart representing the feature importance for both tasks, concurrency (Task 1) and interference (Task 2) on both GPUs. The *x*-axis shows the importance values, while the *y*-axis shows the features.

Regarding the concurrency, shown in Figs. 8 and 9, the number of blocks per grid of both kernels and the number of register of the second kernel are the most relevant features on both GPUs. For kernels (K_1, K_2) to run concurrently from the beginning, K_1 blocks must not occupy all the SMs entirely, so the number of blocks of K_1 must be smaller than the maximum number of blocks

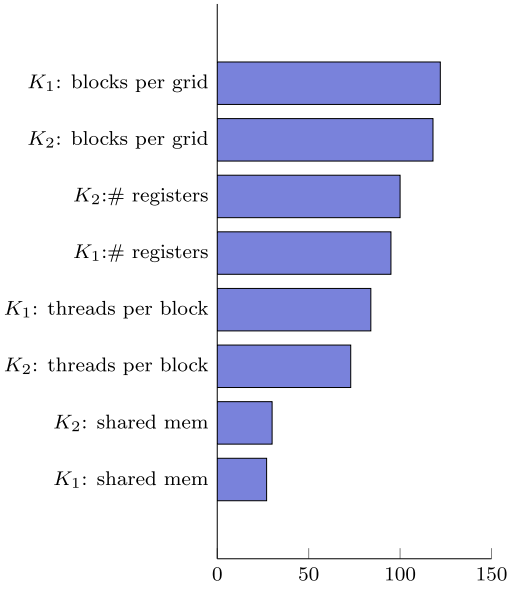


Fig. 8. Concurrency on P100.

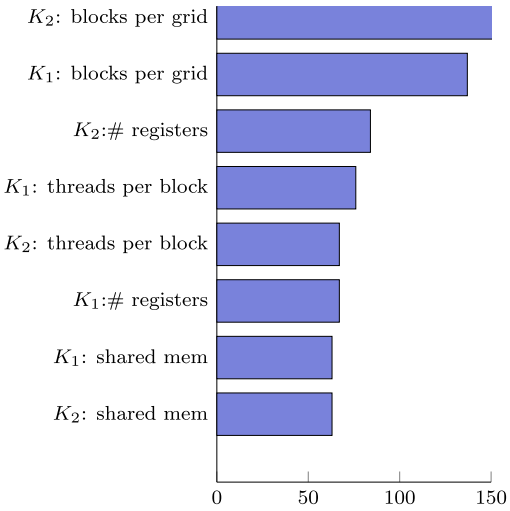


Fig. 9. Concurrency on RTX-2080.

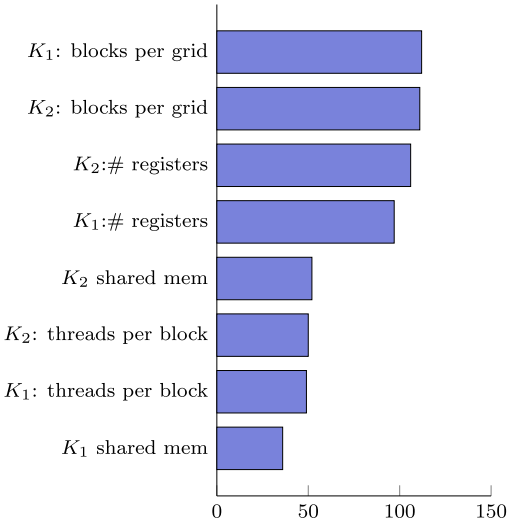


Fig. 10. Interference on P100.

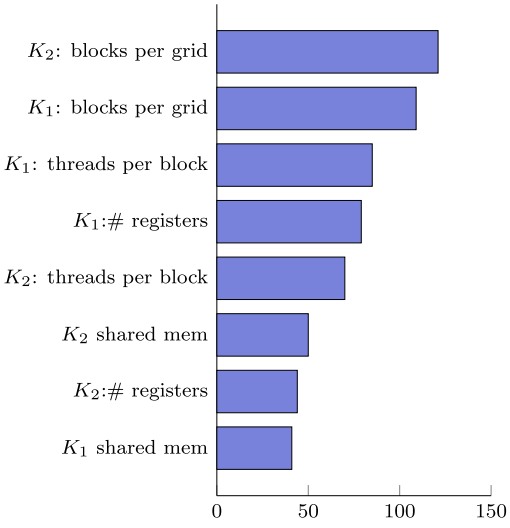


Fig. 11. Interference on RTX-2080.

that the hardware can allow being active in the SMs, which means that K_1 is leaving space for K_2 execution. The number of registers of K_2 is important to determine if there is space in the register file for the K_2 variables. For P100, blocks per grid are around 20% more important than the number of register. For RTX-2080, this difference is more pronounced, blocks per grid have around the double of importance than the number of registers. This occurs because RTX-2080 has a smaller number of SMs, which means that the kernels blocks will have less space to be allocated, increasing the importance of this feature.

Regarding the interference, as exhibited in Figs. 10 and 11, the results are somewhat different for the two GPUs. On P100, the method elicits the blocks per grid and the number of registers as the most important features, with almost the same importance. On RTX-2080, the method also elicits the blocks per grid as the most important feature, around 30% more important than the other features, but the importance of the number of threads per block is increased when compared to the P100 results. The importance of the blocks per grid on the interference results reinforces the leftover policy of NVIDIA. When the kernels execute concurrently, the first kernel allocates the GPU resources and the second kernel can run with the leftover resources. On RTX-2080, nonetheless, the importance of the number of threads per block increases since RTX-2080 has the same number of registers as P100, but its maximum number of threads per SM is half of the P100 maximum. This means that some warps of the second kernel have to have their execution postponed when the maximum number of threads is reached in the SM.

5. Related work

Since the introduction of the concurrent kernel execution feature in NVIDIA's Fermi GPU architecture, several GPU multiprogramming approaches have been studied. Kernel reordering techniques were proposed to improve GPU throughput by taking advantage of concurrent kernel execution focusing on the order in which GPU kernels are invoked on the host side [27–29]. Modifying kernel granularity was another mechanism proposed to improve the GPU utilization. The kernels can be sliced in smaller pieces [30,31] or molded to different dimensions of the grid and thread blocks [26,32] in order to create more concurrency opportunities. There are also efforts in hardware enhancements for dividing the GPU resources among the concurrent kernels, called *spatial multitasking* [33,34]. Other studies address the problem of sharing the GPU with virtualization techniques [35,36].

The co-scheduling of kernels of different applications on the GPU have also been studied in recent years. The work of Margiolas and O'Boyle [37] presented accelOS, a modified OpenCL JIT compiler that analyzes the kernel behavior at the compilation time and transparently modifies the kernel code in terms of the thread blocks size in order to improve fairness in the assignment of the GPU resources among multiple kernels. Belviranli et al. [38] work focused on the data transfer overhead in the scheduling decisions. The work of Wen et al. [20] proposed a graph-based algorithm to schedule kernels in pairs. Their approach models the co-execution of kernels as a graph, in which nodes represent kernels while an edge indicates that the co-execution of the two nodes can experience performance gain, and the edge weight can be labeled with the relative speedup of co-execution. This approach, however, is not able to predict the co-execution effect based on the kernels resource requirements as our approach does. The recent work of Shekofteh et al. [39] claims that using streams without prior knowledge about kernels can result in no performance improvement. They proposed to schedule pair of kernels based on the ratio of the use of compute-related resources and memory-related resources. Their scheduler tries to schedule

together kernels with high values of this ratio with kernels with low values of this ratio. In addition they propose a kernel slicing mechanism to improve the concurrency opportunities. Compared to our approach, we propose the use of metrics that can be derived before the execution and we show the impact of these metrics in the co-execution.

In terms of performance models for GPUs, Baghsorkhi et al. [40] presented a compiler-based approach to performance modeling on GPU to estimate the application execution time. Sim et al. [41] proposed the GPUPerf framework that focuses on determining the bottlenecks of the code and on estimating potential performance benefits from removing these bottlenecks. These models, however, are designed for a single kernel execution.

Closer to our work are the proposals that address the problem of performance interference on co-execution of kernels. Hu et al. [7] proposed a slowdown estimation model for GPUs, whose focus is on memory contention of concurrent kernels. Jog et al. [42] proposed a memory scheduling mechanism that extends the hardware memory scheduler to a more fair policy relying on the bandwidth and L2 behavior of kernels. These two works, however, consider an ideal case where the GPU resources are statically assigned to each kernel, and not the real GPU scheduler.

The work by Yu et al. [43] presented a performance modeling approach used to predict the best block size and estimate the co-execution performance. Their performance modeling, however, is based on classifying the GPU kernels into compute-intensive and memory-intensive by using the t-SNE (t-Distributed Stochastic Neighbor Embedding) technique on the values of performance counters.

In a previous work [44], the necessary conditions for simultaneous execution, aiming to propose an algorithm that described when actual concurrency could occur and a model for slowdown estimation, were studied. Although the proposed strategies were tested and validated with some synthetic and real-world applications successfully, later, when they were evaluated in a larger and more complete test set, they failed to identify concurrency and estimate slowdown correctly.

Machine learning techniques have become increasingly important in task scheduling on heterogeneous systems [45,46] and in performance modeling for GPUs [47,48], however, those works do not consider the concurrent execution of kernels in the GPU. To the best of our knowledge, the only work that uses Machine Learning techniques in concurrent kernel execution is the work by Wen et al. [21]. In this work, they proposed two predictive models based on decision trees which classify newly arriving kernels. The first model determines device affinity in a CPU–GPU environment, separating CPU and GPU kernels. The second model determines whether or not to merge two GPU kernels in order to take advantage of concurrency on the GPU. Their approach, however, used the kernel fusion method that fuses concurrent kernels into a single new one, then dispatches it to the GPU. This fusion, however, can only be performed in kernels of the same application and does not consider that the order at which the kernels are fused makes a huge difference in their performance interference [29].

In addition, our machine learning approach considers not only decision trees but also other techniques (detailed in Section 3) as neural networks, regression and instance-based methods. Although decision trees are easy to interpret, different from some of the methods we used here, the usage of other techniques may lead to better predictive results.

To achieve an interpretable model, we used a decision tree-based technique named XGBoost [5], which has presented excellent predictive results in a number of tasks. XGBoost induces a set of decision trees considering different partitions of the dataset in order to avoid overfitting, a more robust solution than the considered in the work of Wen et al. [21]).

Although it is a ensemble technique, which may compromise the interpretability, XGBoost implementations allows that any instance and a merge of the set of trees are used for classification explanation.

Furthermore, we evaluate the machine learning models considering not only accuracy but also some other important statistics (e.g. precision, recall and kappa — this last one provides a measure on how far from the expected classification the results are).

6. Conclusions

Modern GPU architectures support concurrent sharing of the GPU resources among multiple kernels, which can unleash the power of the GPU for dynamic and highly virtualized environments such as large scale heterogeneous clusters and cloud environments. This work presented an extensive analysis of the concurrent execution of the kernels from Rodinia, Parboil, and SHOC benchmarks on two different GPU architectures to assess how their performance is affected by the concurrent execution. We used machine learning techniques to model and predict the execution interference of the kernels and which types of kernel can execute concurrently.

Our focus was to identify tricky relations among the resource requirements of the kernels and their concurrent execution. We used four machine learning techniques to capture the hidden patterns that make a kernel interfere in the execution of another one. Our results showed that XGBoost, a state-of-the-art ensemble method, achieved the best quantitative results with statistical significance validated. The feature importance method showed the resource requirements that are the most relevant in the concurrent execution performance. By analyzing the variables chosen as the most important to induce the XGBoost model, we conclude that the number of blocks per grid is the most relevant feature to define if the kernels will execute concurrently and to influence the performance interference. The second most important feature depends on the GPU architecture. For the GPU with more resources, P100, the number of registers is key for the kernels interference, while for the GPU with less SM resources, but the same amount of registers, RTX-2080, the number of threads per block is more relevant in the kernels interference. The GPU architectures evolve quickly and show different scheduling behaviors. Although, it is not possible to convey a generic model that would fit in any GPU architecture, the use of machine learning strategies elucidated the variables that matters the most for each architecture.

The results obtained in this work can be further used in the design of a scheduling strategy for GPUs, where the resource requirements of the kernels could help the scheduler in making wise decisions for concurrent execution. For future work, we intend to investigate memory contention and include prediction models based on matrix factorization, to better understand the relation between the kernels and the GPU architecture. We also intend to study the effects of distinct GPU and SM architectures and global memory management in the concurrent execution performance. The execution of more than two concurrent kernels is also in our plans. This would show the effect of concurrency when multiple kernels are submitted to execute. We also intend to investigate kernel concurrency in multi-GPUs environments. However, synchronization issues are not trivial and a more sophisticated framework should be developed. Tensor cores are a new architectural element, present in the latest GPUs. They allow dedicated deep learning task to be executed directly on the hardware level. We intend to explore these capabilities, in

order to create another level of concurrency, whenever some of the kernels are demanding this kind of processing. An analysis of why some instances are wrongly classified also may elicit new conclusions.

CRediT authorship contribution statement

Pablo Carvalho: Conceptualization, Methodology, Software, Investigation, Data curation, Visualization. **Esteban Clua:** Conceptualization, Review, Writing. **Aline Paes:** Methodology, Validation, Investigation, Writing. **Cristiana Bentes:** Conceptualization, Methodology, Investigation, Writing, Review, Supervision. **Bruno Lopes:** Validation, Investigation, Writing. **Lúcia Maria de A. Drummond:** Conceptualization, Resources, Review, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

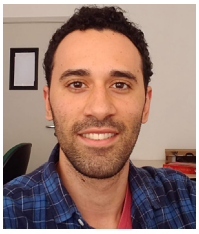
Acknowledgment

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001.

References

[1] P. Carvalho, L.M. Drummond, C. Bentes, E. Clua, E. Cataldo, L.A. Marzulo, Kernel concurrency opportunities based on GPU benchmarks characterization, Cluster Comput. 23 (1) (2020) 177–188.
[2] R.S. Michalski, J.G. Carbonell, T.M. Mitchell, Machine Learning: An Artificial Intelligence Approach, Springer Science & Business Media, 2013.
[3] I. Guyon, A. Elisseeff, An introduction to variable and feature selection, J. Mach. Learn. Res. 3 (Mar) (2003) 1157–1182.
[4] A. Zien, N. Krämer, S. Sonnenburg, G. Rätsch, The feature importance ranking measure, in: Joint European Conference on Machine Learning and Knowledge Discovery in Databases, Springer, 2009, pp. 694–709.
[5] T. Chen, C. Guestrin, Xgboost: A scalable tree boosting system, in: Proceedings of the 22nd Acm Sigkdd International Conference on Knowledge Discovery and Data Mining, ACM, 2016, pp. 785–794.
[6] NVIDIA Multi-process service https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.
[7] Q. Hu, J. Shu, J. Fan, Y. Lu, Run-time performance estimation and fairness-oriented scheduling policy for concurrent GPGPU Applications, in: 45th International Conference on Parallel Processing, ICPP, 2016, 2016, pp. 57–66.
[8] T.M. Mitchell, et al., Machine Learning, McGraw Hill, Burr Ridge, IL, 1997.
[9] G.H. John, R. Kohavi, K. Pfleger, Irrelevant features and the subset selection problem, in: Machine Learning Proceedings 1994, Elsevier, 1994, pp. 121–129.
[10] N. Louw, S. Steel, Variable selection in kernel Fisher discriminant analysis by means of recursive feature elimination, Comput. Statist. Data Anal. 51 (3) (2006) 2043–2055.
[11] A.A. Ogunleye, W. Qing-Guo, XGBoost model for chronic kidney disease diagnosis, IEEE/ACM Trans. Comput. Biol. Bioinform. (2019).
[12] J. Wang, M. Gribskov, IRESpy: an XGBoost model for prediction of internal ribosome entry sites, BMC Bioinform. 20 (1) (2019) 409.
[13] T. Cover, P. Hart, Nearest neighbor pattern classification, IEEE Trans. Inf. Theory 13 (1) (1967) 21–27.
[14] S. Menard, Applied Logistic Regression Analysis, Vol. 106, Sage, 2002.
[15] S.S. Haykin, S.S. Haykin, S.S. Haykin, S.S. Haykin, Neural Networks and Learning Machines, Vol. 3, Pearson Upper Saddle River, 2009.
[16] J. Cohen, A coefficient of agreement for nominal scales, Educ. Psychol. Meas. 20 (1) (1960) 37–46.
[17] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, S.-H. Lee, K. Skadron, Rodinia: A benchmark suite for heterogeneous computing, in: Proceedings of the IEEE International Symposium on Workload Characterization, IISWC, 2009, pp. 44:54.
[18] J.A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G.D. Liu, W. mei W. Hwu, Parboil: A revised benchmark suite for scientific and commercial throughput computing, 2012.
[19] A. Danalis, G. Marin, C. McCurdy, J.S. Meredith, P.C. Roth, K. Spafford, V. Tipparaju, J.S. Vetter, The scalable heterogeneous computing (SHOC) benchmark suite, in: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, 2010, pp. 63:74.
[20] Y. Wen, M.F. O'Boyle, C. Fensch, MaxPair: Enhance OpenCL concurrent kernel execution by weighted maximum matching, in: Proceedings of the 11th Workshop on General Purpose GPUs, ACM, 2018, pp. 40–49.
[21] Y. Wen, M.F. O'Boyle, Merge or separate?: Multi-job scheduling for OpenCL kernels on CPU/GPU platforms, in: Proceedings of the General Purpose GPUs, ACM, 2017, pp. 22–31.
[22] J.S. Bergstra, R. Bardenet, Y. Bengio, B. Kégl, Algorithms for hyper-parameter optimization, in: Advances in Neural Information Processing Systems, 2011, pp. 2546–2554.
[23] D.F. Bauer, Constructing confidence sets using rank statistics, J. Amer. Statist. Assoc. 67 (1972) 687–690.
[24] I. Jolliffe, Principal Component Analysis, second ed., Springer, 2002.
[25] J. Friedman, T. Hastie, R. Tibshirani, The Elements of Statistical Learning, Vol. 1, No. 10, Springer Series in Statistics New York, NY, USA, 2001.
[26] V.T. Ravi, M. Becchi, G. Agrawal, S. Chakradhar, Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework, in: Proceedings of the 20th International Symposium on High Performance Distributed Computing, ACM, 2011, pp. 217–228.
[27] F. Wende, F. Cordes, T. Steinke, On improving the performance of multi-threaded CUDA applications with concurrent kernel execution by kernel reordering, in: Symposium on Application Accelerators in High Performance Computing, SAAHPC, 2012, 2012, pp. 74–83.
[28] T. Li, V.K. Narayana, T. El-Ghazawi, A power-aware symbiotic scheduling algorithm for concurrent GPU kernels, in: IEEE 21st International Conference on Parallel and Distributed Systems, ICPADS, 2015, 2015, pp. 562–569.
[29] R.A. Cruz, C. Bentes, B. Breder, E. Vasconcellos, E. Clua, P.M. de Carvalho, L.M. Drummond, Maximizing the GPU resource usage by reordering concurrent kernels submission, Concurr. Comput.: Pract. Exper. 1 (1) (2018) 1–12, (on line).
[30] J. Zhong, B. He, Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling, IEEE Trans. Parallel Distrib. Syst. 25 (6) (2014) 1522–1532.
[31] A. Tarakji, A. Gladis, T. Anwar, R. Leupers, Enhanced GPU resource utilization through fairness-aware task scheduling, in: Trustcom/BigDataSE/ISPA, 2015 IEEE, Vol. 3, IEEE, 2015, pp. 45–52.
[32] S. Pai, M.J. Thazhuthaveetil, R. Govindarajan, Improving GPGPU concurrency with elastic kernels, in: ACM SIGPLAN Notices, Vol. 48, No. 4, 2013, pp. 407–418.
[33] Y. Liang, H.P. Huynh, K. Rupnow, R.S.M. Goh, D. Chen, Efficient GPU spatial-temporal multitasking, IEEE Trans. Parallel Distrib. Syst. 26 (3) (2015) 748–760.
[34] J.T. Adriaens, K. Compton, N.S. Kim, M.J. Schulte, The case for GPGPU spatial multitasking, in: 2012 IEEE 18th International Symposium on High Performance Computer Architecture, HPCA, 2012, pp. 1–12.
[35] T. Li, V.K. Narayana, E. El-Araby, T. El-Ghazawi, GPU resource sharing and virtualization on high performance computing systems, in: International Conference on Parallel Processing, ICPP, 2011, 2011, pp. 733–742.
[36] Y. Suzuki, S. Kato, H. Yamada, K. Kono, GPUvm: Why not virtualizing GPUs at the hypervisor? in: USENIX Annual Technical Conference, 2014, pp. 109–120.
[37] C. Margiolas, M.F. O'Boyle, Portable and transparent software managed scheduling on accelerators for fair resource sharing, in: Proceedings of the 2016 International Symposium on Code Generation and Optimization, ACM, 2016, pp. 82–93.
[38] M.E. Belviranli, F. Khorasani, L.N. Bhuyan, R. Gupta, CuMAS: Data transfer aware multi-application scheduling for shared GPUs, in: Proceedings of the 2016 International Conference on Supercomputing, ACM, 2016, p. 31.
[39] S.-K. Shekofteh, H. Noori, M. Naghibzadeh, H. Fröning, H.S. Yazdi, Ccuda: Effective co-scheduling of concurrent kernels on gpus, IEEE Trans. Parallel Distrib. Syst. 31 (4) (2019) 766–778.
[40] S.S. Baghsorkhi, M. Delahaye, S.J. Patel, W.D. Gropp, W.-m.W. Hwu, An adaptive performance modeling tool for GPU architectures, in: ACM Sigplan Notices, Vol. 45, ACM, 2010, pp. 105–114.
[41] J. Sim, A. Dasgupta, H. Kim, R. Vuduc, A performance analysis framework for identifying potential benefits in GPGPU applications, in: ACM SIGPLAN Notices, Vol. 47, No. 8, ACM, 2012, pp. 11–22.
[42] A. Jog, O. Kayiran, T. Kesten, A. Pattnaik, E. Bolotin, N. Chatterjee, S.W. Keckler, M.T. Kandemir, C.R. Das, Anatomy of GPU memory system for multi-application execution, in: Proceedings of the 2015 International Symposium on Memory Systems, 2015, pp. 223–234.

[43] L. Yu, X. Gong, Y. Sun, Q. Fang, N. Rubin, D. Kaeli, Moka: Model-based concurrent kernel analysis, in: 2017 IEEE International Symposium on Workload Characterization, IISWC, IEEE, 2017, pp. 197–206.
[44] R. Cruz, L. Drummond, E. Clua, C. Bentes, Analyzing and estimating the performance of concurrent kernels execution on GPUs, in: Anais do XVIII Simpósio em Sistemas Computacionais de Alto Desempenho, SBC, Porto Alegre, RS, Brasil, 2017.
[45] D. Nemirovsky, T. Arkose, N. Markovic, M. Nemirovsky, O. Unsal, A. Cristal, A machine learning approach for performance prediction and scheduling on heterogeneous CPUs, in: Computer Architecture and High Performance Computing (SBAC-PAD), 2017 29th International Symposium on, IEEE, 2017, pp. 121–128.
[46] Y. Wen, Z. Wang, M.F. O'boyle, Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms, in: High Performance Computing (HiPC), 2014 21st International Conference on, IEEE, 2014, pp. 1–10.
[47] T.T. Dao, J. Kim, S. Seo, B. Egger, J. Lee, A performance model for gpus with caches, IEEE Trans. Parallel Distrib. Syst. 26 (7) (2015) 1800–1813.
[48] G. Wu, J.L. Greathouse, A. Lyashevsky, N. Jayasena, D. Chiou, GPGPU performance and power estimation using machine learning, in: High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on, IEEE, 2015, pp. 564–576.



Pablo Carvalho received his B.Sc. degree in Computer Science from Federal Fluminense University in Brazil. Currently, he is a D.Sc. student at the same university. His research interests include high performance and GPU computing.

Esteban Clua is professor at Universidade Federal Fluminense and coordinator of UFF Medialab. He has Ph.D. degree in Computer Science at PUC-Rio and his main research and development areas are Digital Games, Virtual Reality, GPUs and Visualization. He is one of the founders of SBGames (Brazilian Symposium of Games and Digital Entertainment) and was the president of Game Committee of the Brazilian Computer Society from 2010 through 2014. Today he is the commission member for Brazil at the Technical Committee of Entertainment at the International Federation of Information Processing (IFIP). In 2015 he was nominated as CUDA Fellow.

Aline Paes is Assistant professor at the Institute of Computing of the Universidade Federal Fluminense (UFF). She holds a Ph.D. and M.Sc. in Systems Engineering and Computer Science, with an emphasis on Artificial Intelligence, by the Faculty of Computer Science, COPPE, Universidade Federal do Rio de Janeiro (UFRJ). During the Ph.D. she was a visiting scholar for a year at Imperial College London. She received a post-doctoral fellowship from CNPq at the Faculty of Computer Science, COPPE, under the supervision of Professor Valmir Carneiro Barbosa. Aline Paes works in the area of Computer Science, with an emphasis on Artificial Intelligence.

Cristiana Bentes received her B.Sc. degree in Mathematics from State University of Rio de Janeiro, Brazil, and her M.Sc. and D.Sc. degrees in Systems Engineering and Computer Science at Federal University of Rio de Janeiro. She is currently a Full Professor in the Department of Systems Engineering and Computer Science at State University of Rio de Janeiro. Her currently research interests include Highperformance Computing, Parallel Programming, Novel Parallel Architectures and Cloud Computing.

Bruno Lopes is a Professor at Universidade Federal Fluminense (IC/UFF) and researcher at FR∀M∃ Lab. He has been a visiting scholar at Deduc⊢eam/INRIA from which he keeps active partnership together with Université Lyon 3 and TecMF/PUC-Rio. He has been working mainly with logics for concurrent systems but he also has been working into the development of extensible theorem provers, normalization for natural deduction systems, ontologies, formalization of multi-agent systems and proof theory for logic systems. He is also on the committee of the Brazilian Logic Society (SBL).

Lucia M.A. Drummond obtained her D.Sc. in Systems Engineering and Computer Science from the Federal University of Rio de Janeiro, Brazil, in 1994, where she took part of the group which developed the first Brazilian parallel computer. She is in the Department of Computer Science of The Fluminense Federal University (UFF) since 1989, where she is now Full Professor. Her research interests are parallel and distributed computing, including theory and applications.