UNIVERSIDADE FEDERAL FLUMINENSE

PABLO MOREIRA CAVALCANTE DE CARVALHO

Concurrency and Interference Analysis of Kernels on GPUs

NITERÓI

2020

UNIVERSIDADE FEDERAL FLUMINENSE

PABLO MOREIRA CAVALCANTE DE CARVALHO

Concurrency and Interference Analysis of Kernels on GPUs

Dissertation presented to the Computing Graduate Program of the Universidade Federal Fluminense in partial fulfillment of the requirements for the degree of Master of Science. Topic Area:Computing Systems.

Advisor:
Lúcia Maria de Assumpção Drummond

Co-advisor:
Cristiana Bentes

NITERÓI

2020

Ficha catalográfica automática - SDC/BEE
Gerada com informações fornecidas pelo autor

D278c De carvalho, Pablo Moreira Cavalcante
Concurrency and Interference Analysis of Kernels on GPUs / Pablo Moreira Cavalcante De carvalho ; Lúcia Maria De Assumpção Drummond, orientadora ; Cristiana Barbosa Bentes, coorientadora. Niterói, 2020.
53 f. : il.

Dissertação (mestrado)-Universidade Federal Fluminense, Niterói, 2020.

DOI: http://dx.doi.org/10.22409/PGC.2020.m.14359475764

1. GPU. 2. Análise de Concorrência. 3. Aprendizado de Máquina. 4. Produção intelectual. I. De Assumpção Drummond, Lúcia Maria, orientadora. II. Barbosa Bentes, Cristiana, coorientadora. III. Universidade Federal Fluminense. Instituto de Computação. IV. Título.

CDD -

Bibliotecário responsável: Sandra Lopes Coelho - CRB7/3389

Pablo Moreira Cavalcante de Carvalho

Concurrency and Interference Analysis Between Kernels on GPUs

Dissertation presented to the Computing Graduate Program of the Universidade Federal Fluminense in partial fulfillment of the requirements for the degree of Master of Science. Topic Area: Computing Systems.

Approved on March 2020.

APPROVED BY

Prof. Lúcia Maria de Assumpção Drummond - Advisor, UFF

Prof. Cristiana Barbosa Bentes, Co Advisor, UERJ

Prof.Alba Cristina Magalhaes Alves de Melo, UnB

Prof. Aline Marins Paes Carvalho, UFF

Prof. Olivier Beaumont, INRIA - Bordeaux

Niterói

2020

Acknowledgements

I am grateful to Lucia Drummond and Cristiana Bentes for the guidance in this work, the encouragement through all those years since graduation and the example of commitment.

I also would like to thank Esteban Clua, Aline Paes and Bruno Lopes for the valuable contribution to this work.

I want to thank my family, especially my mother, Fátima, for the support and incentive.

And Finally, thanks CAPES, for the scholarship that sponsored me for the most part of those two years as a M.Sc student.

Resumo

Os sistemas heterogêneos que empregam CPUs e GPUs estão se tornando cada vez mais populares em data centers e ambientes de nuvem em larga escala. Nessas plataformas, o compartilhamento de uma GPU entre diferentes aplicativos é um recurso importante para melhorar a utilização do *hardware* e *throughput*. No entanto, em cenários em que as GPUs são compartilhadas competitivamente, alguns desafios surgem. A decisão sobre a execução simultânea de diferentes kernels é tomada pelo *hardware* e depende dos requisitos de recursos dos *kernels*. Além disso, é muito difícil entender todas as variáveis de *hardware* envolvidas nas decisões de execução simultânea, a fim de descrever um método formal de alocação. Neste trabalho, usamos técnicas de aprendizado de máquina para entender como os requisitos de recursos dos kernels mais importantes *benchmarks* de GPU afetam sua execução simultânea. Nosso foco é fazer com que os algoritmos de aprendizado de máquina capturem os padrões ocultos que fazem um kernel interferir na execução de outro quando são enviados para execução ao mesmo tempo. As técnicas analisadas foram k -NN, regressão linear, *Multilayer Perceptron* e XGBoost (que obtiveram os melhores resultados) sobre os *benchmarks suites* de GPU, Rodinia, Parboil e SHOC. Nossos resultados mostraram que, dentre os recursos selecionados na análise, o número de blocos por grid, o número de *threads* por bloco e o número de registradores são os recursos de consumo de recursos que mais afetam o desempenho da execução simultânea.

Palavras-chave: GPU, Análise de Concorrência, Aprendizado de Máquina.

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, Linear 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.

Palavras-chave: GPU, Concurrrency Analysis, Machine Learning.

List of Figures

List of Tables

List of Abbreviations

CUDA	:	Compute Unified Device Architecture;
CPU	:	Central Processor Unit;
GPU	:	Graphic Processor Unit;
SM	:	Streaming Multiprocessor;

Contents

Chapter 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. More recently 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 cost reduction, energy savings and more flexibility.

In this scenario, sharing efficiently 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 Stream Multiprocessor 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 previous work, Carvalho *et al.* [7] 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 might 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 Figure 1, we illustrate how difficult it is to understand and predict the execution interference of the kernels and what type 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. Figure 1 shows the pairwise execution of three kernels with low, medium and high resource usage. 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.

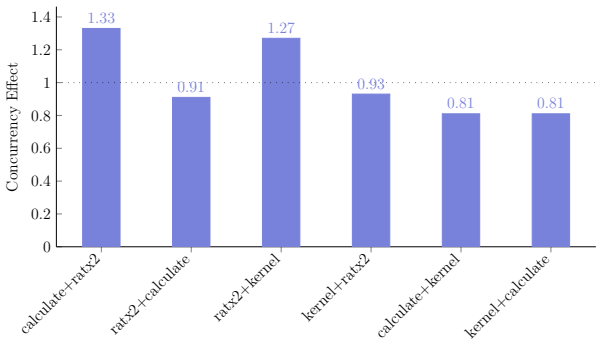


Figure 1.1: Ratio of the execution time of kernels *calculate_temp* (from Hotsopt application in the Rodinia benchmarks suite), *kernel* (from Myocite application in Rodinia) and *ratx2_kernel* (from S3D application in the SHOC benchmark suite) running without concurrency over their co-execution with concurrency

In this work, we performed an extensive analysis of the concurrent execution of the kernels from Rodinia [8], Parboil[42], and SHOC[15] benchmarks to assess how their

performance is affected by the concurrent execution. We use four machine learning techniques [30] 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 [18] and feature importance [54] 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 exponential 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 with each other in the concurrent execution; (3) A comparison of the machine learning techniques, k-Nearest Neighbours, 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 matter 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 [9], 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 characterization and concurrency study presented in Chapter 4 was published as: Carvalho, P.; Drummond, L. M.; Bentes, C.; Clua, E.; Cataldo, E.;Marzulo, L. A.Kernel concurrency opportunities based on gpu benchmarks characterization. *Cluster Computing* (on line) 1, 1 (2019), 1–12.

This dissertation is organized as follows. Chapter 2 gives an overview on GPU architecture and introduces to Machine Learning concepts applied latter on this work. Chapter

3, exposes related works. Chapter 4 introduces our first concurrency experiment and the kernel characterization study that served as basis, this chapter also presents our concurrent execution framework that made possible the concurrency experiments. Chapter 5 presents our machine learning analysis on concurrent kernel execution, explains the motivation and details the workflow. Chapter 6 exhibits our results and in Chapter 7 we draw our conclusions.

Chapter 2

Background

This chapter presents background regarding the research presented in this dissertation. We start with a brief review on the NVIDIA GPU architecture and CUDA terminology. Following, we describe some introductory concepts on Machine Learning techniques.

2.1 Graphic Processor Units

The GPU architecture is designed for fine-grain massive parallel processing. It comprises several Streaming Multiprocessors (SMs). Each SM contains several computing cores and resources such as registers, shared memory and L1 cache. The SM composition tends to change as architectures evolve, Figure 2.1 demonstrates the SM structure of the Pascal architecture (P100). As can be seen, it is not only composed of standard cores that perform integer and floating-point operations but also a specific unit for double-precision arithmetic. A Turing architecture SM can be observed in Figure 2.2, this is a more recent architecture than Pascal and some interesting features can be observed: Cores are separated by integer and floating-point units this time. It also includes tensor flow units and an RT core used for ray tracing calculation.

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.

Figure 2.1: P100 Stream Multiprocessor[1]

[1] https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

2.1.1 Software Architecture

As presented by Figure 2.3, when creating a CUDA application, the source code (in form of a .cu file) needs to be compiled by nvcc. Nvcc is not exactly a compiler, but it works with the compiler used in the environment that it is installed and it only works with a restrictive set of compilers. For CUDA runtime applications, nvcc attachs GPU code into strings on the output binary. When using the fatbin option, the binary will carry the microcode for the target GPU. The PTX option creates a "Parallel Thread eXecution" file containing an intermediate version of the GPU native microcode. The task of translating it to GPU microcode is handled by the driver[50].

Normally CUDA applications are developed by using components of the CUDA Runtime library as it gives a high level control of functionalities like memory transfer, stream creation and kernel related callbacks. The CUDA runtime works as an abstraction layer to the CUDA driver, and provides some complexity reduction for the device control while de CUDA Driver API gives more control over contexts and module loading.

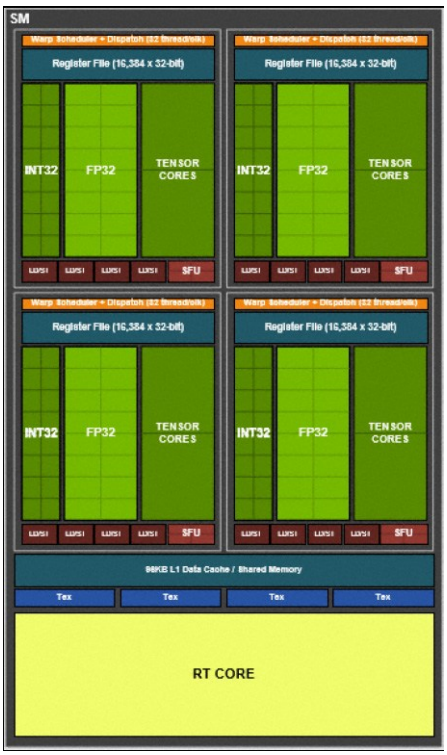


Figure 2.2: Turing RTX Stream Multiprocessor[2]

[2] https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

2.1.2 CUDA MPS

A CUDA application is composed by a hierarchy of parallel thread blocks that starts with the creation of a CUDA context. The context encloses all the resources required for the parallel execution. Since the Hyper-Q technology (introduced in the Kepler architecture, 2012), kernels can be assigned to different streams which indicates that they are independent and can be executed at the same time, whenever there are available hardware resources for that. The hardware scheduler assigns each stream to a different work queue and kernels launched from work queues belonging to the same CUDA context can execute concurrently. Figure 2.4 describes the model of MPS execution and how the MPS server process achieve the concurrent execution from kernels of different processes.

To allow kernels from different context to execute concurrently, NVIDIA created a software solution called Multi-Process Service (MPS) [34]. MPS is a client-server implementation that was designed mainly to allow different processes to share the GPU resources. The MPS client runtime is built into the CUDA driver library. The MPS server is started by the MPS control daemon, that listens to MPS clients from different CUDA contexts in order to bypass the hardware scheduling limitations. There is no need

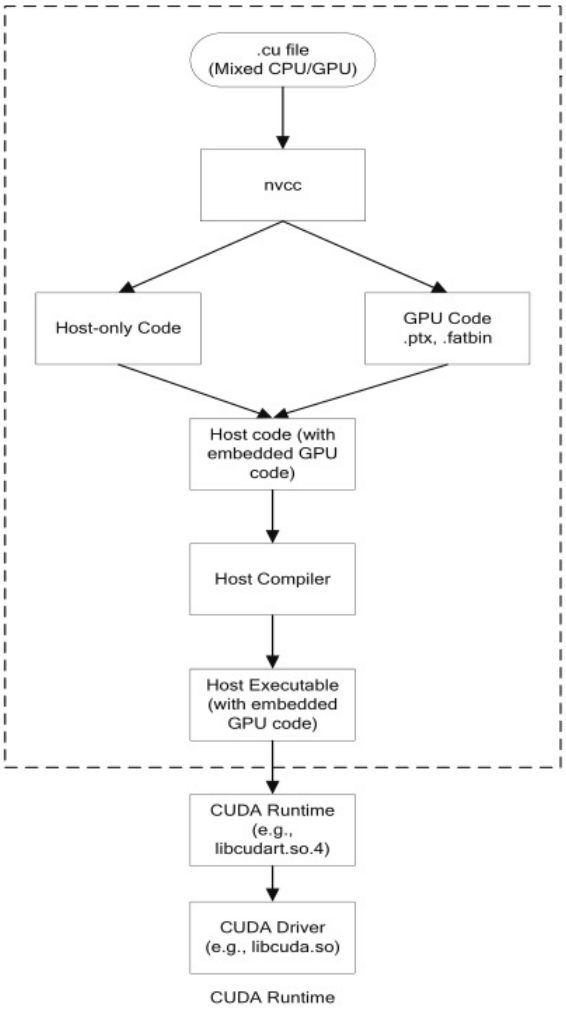


Figure 2.3: CUDA software architecture [50]

for any application modification.

2.2 Machine Learning

Machine Learning (ML) techniques focus on making computers learn how to act, without being explicitly programmed, relying on data and information about the world [30, 31]. 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

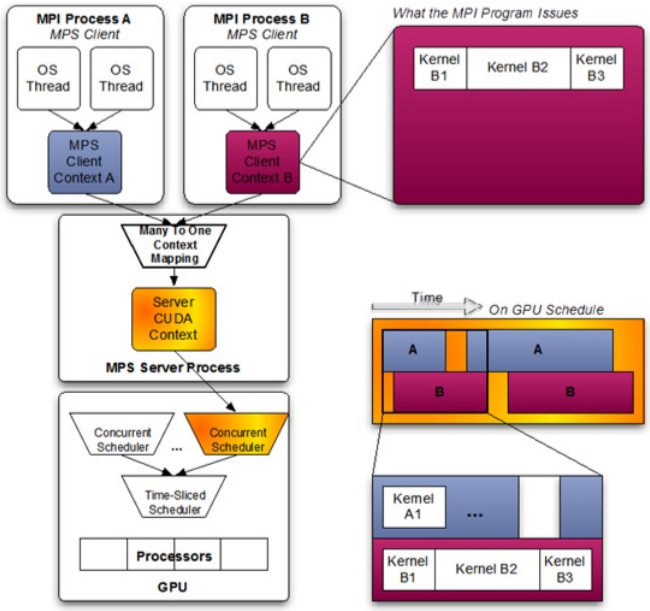


Figure 2.4: MPS architectural model[3]

[3] https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf

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 labelled 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 Figure 2.5. 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.

2.2.1 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 [23]. 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

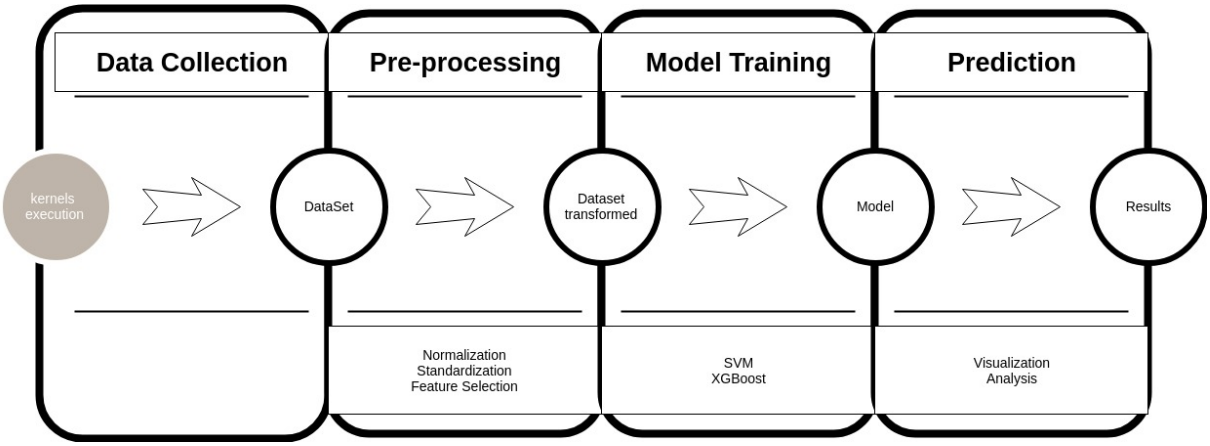


Figure 2.5: Machine Learning pipeline

Principal Component Analysis. 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 [27]. 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.

2.2.2 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 [9]. 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 [36, 45]. 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 Neighbours [11]) 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 [29]: 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 $\mathbb{R} \to [0, 1]$ 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 [19]: 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 [9]: (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 greedly 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.

2.2.3 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 [10].

To 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 as t-test.

Chapter 3

Related Works

In this chapter, we present previous research in GPUs in the areas of: concurrent kernel execution, performance interference and machine learning.

3.1 Concurrent Kernel Execution

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. Wende *et. al* [49] presented a kernel reordering system based on producer-consumer problem. The work describes the amount of memory and the number of Streaming Multiprocessors available as the main restrictions for concurrency. Li *et. al* [25] proposed a scheduling technique focused on reducing the power consumption based on optimizing the Streaming Multiprocessor static resources as shared memory and registers. Cruz *et. al*[13] presented a reordering technique based on the knapsack problem, associating kernels as knapsack items and using resource requirements to calculate their weight, considering shared memory, registers, number of threads and number of blocks.

Modifying kernel granularity was another mechanism proposed to improve GPU utilization. The kernels can be sliced in smaller pieces [44] or molded to different dimensions of the grid and thread blocks in order to create more concurrency opportunities. Zhong *et. al* [53] defined kernel slice as a subset of the thread blocks of a launched kernel, the reason for not considering a smaller unit as the slice, a warp for example, is the data dependency in the same block. The work proposed a system to improve concurrent kernel execution throughput by performing dynamic slicing and scheduling.

The molding technique consists of modifying kernel characteristics as the number of threads, blocks, or grids, for example. For real application usage, the technique drawback is that kernels must be written in a way that accepts modifying the resource usage in execution time and not in compilation time. Although this is considered a good programming practice, it is very common to find applications that set the number of threads, blocks or grids as constants on the source code and uses those values to perform some internal tasks inherent to the algorithm that the kernel is proposed to implement. Ravi *et. al* [38] presented a multi-context system that enables kernels from different applications to execute concurrently in the same GPU years before Nvidia MPS[33] was released. They proposed an algorithm based on affinity score calculation to solve the problem of determining if two kernels could run concurrently and efficiently only using pre-execution information like number of threads, number of blocks and the shared memory used. Ravi *et. al* applied molding to reduce kernel throughput when running concurrently. Their work presented two molding techniques: Forced Space Sharing, which consists of reducing the number of thread blocks to use a reduced number of SMs, and Time Sharing with Reduced Threads, which reduces the number of threads within the blocks. Pai *et. al* [37] presented a similar concept to molding, the elastic kernel. An elastic kernel, by definition, is a kernel that runs with different blocks dimension or number of blocks different to defined by the programmer on development.

There are also efforts in hardware enhancements for dividing the GPU resources among the concurrent kernels, Adriaens *et. al*[1] demonstrated the benefits of sharing GPU resources through spacial multitasking, allowing GPU resources to be partitioned among different applications being executed on the same GPU. The work presented six heuristics focused on partitioning the number of SM through different applications and analyzing its performance. Liang *et. al* [26] discussed the concept of spacial-temporal multitasking. They described a *phase* as a set of kernels executing concurrently and define temporal multitasking as the number of execution phases where kernels in the same phase execute concurrently while spatial multitasking determines the resource allocation for kernels in the same phase. The work presented a heuristic based on thread-block interleaving to implement the spacial-temporal multitasking.

Other studies address the problem of sharing the GPU with virtualization techniques. Li *et. al* [24] presented a virtualization system divided into two layers, an API layer that holds GPU usage requests by processes and communicates with a base layer that encapsulates the access to the CUDA API by the implementation of a GPU Virtualization Manager and also controls the GPU memory through a virtual shared memory. Suzuki *et.*

al [43] presented a hypervisor-based GPU virtualization system, the GPUvm. The system works with a guest device driver to run on all client virtual machines. The hypervisor exposes to every client driver, a virtual device that communicates with GPUvm. Thus the physical GPU is never directly accessed by the clients.

The co-scheduling of kernels of different applications on the GPU has also been studied in recent years. The work of Margiolas and O'Boyle [28] 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.* [4] work focused on the data transfer overhead in the scheduling decisions. The recent work of Wen *et al.* [47] 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. All these approaches, however, are not able to predict the co-execution effect based on the kernels' resource requirements as our approach does.

Reference	**Year**	**Technique**	**Implementation**
Wende *et al.* [49]	2012	Reordering	Software
Li *et al.* [25]	2015	Reordering	Software
Cruz *et al.* [13]	2018	Reordering	Software
Zhong *et al.* [53]	2014	Kernel Slicing	Software
Tarakji *et al.* [44]	2015	Kernel Slicing	Software
Pai *et al.* [37]	2013	Elastic Kernels	Software
Ravi *et al.* [38]	2011	Kernel fusion	Software
Liang *et al.* [26]	2015	Spatial Multitasking	Hardware
Adrien *et al.* [1]	2012	Spatial Multitasking	Hardware
Li *et al.* [24]	2011	Resource Virtualization	Software
Suzuki *et al.* [43]	2014	Resource Virtualization	Operating System

Table 3.1: Summary of concurrent kernel execution studies

3.2 Performance Interference

In terms of performance models for GPUs, Baghsorkhi *et al.* [2] presented a compiler-based approach to performance modeling on GPU to estimate the application execution time. Sim *et al.* [40] 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.* [20] proposed a slowdown estimation model for GPUs, whose focus is on memory contention of concurrent kernels. Jog *et al.* [22] 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.* [52] 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 [12], 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.

3.3 Machine Learning

Machine learning techniques have become increasingly crucial in task scheduling on heterogeneous systems [32, 48] and in performance modeling for GPUs [16, 51], 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.* [46]. In this work, they proposed two predictive models based on decision trees that 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 [13].

In addition, our machine learning approach considers not only decision trees but also other techniques 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.

We used a decision tree-based technique named XGBoost [9], 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.* [46]).

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 of how far from the expected classification the results are).

Chapter 4

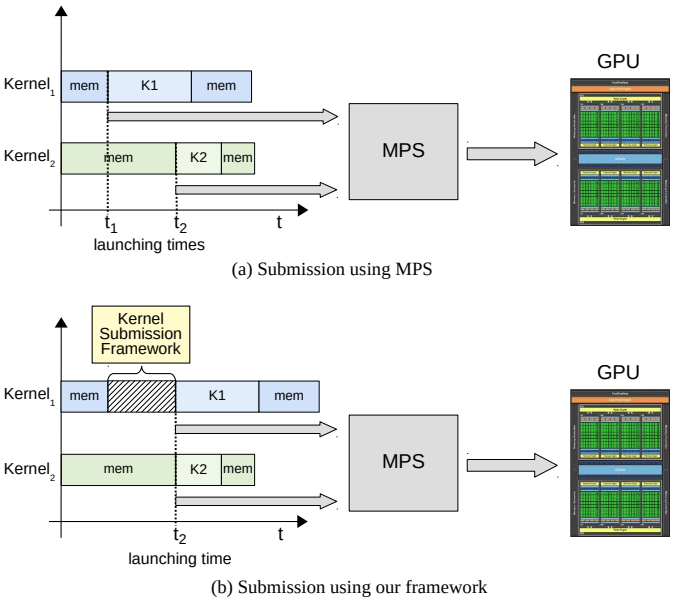
Concurrent Kernel Execution

This chapter presents a study in kernel concurrent execution on the GPU [7]. First it explains the kernel submission framework that was implemented on top of MPS to allow experiments in co-scheduling scenarios. After that, the chapter presents the study in characterizing the kernels of the main GPU benchmark suites in terms of resource usage and a preliminary experiment in running kernel concurrently according to the characterization made.

4.1 Kernel Submission Framework

Although NVIDIA MPS allows the concurrent execution of kernels from different contexts, the problem of using MPS to assess the effects of concurrency in the kernels execution time is that launching two applications at the same time does not guarantee that their kernels will be launched simultaneously and compete for the GPU resources. The MPS client intercepts kernel launching during the application execution, but the kernels can be invoked in different timelines, due to the difference in the memory transfer overheads of the applications. Suppose, for example, two CUDA contexts that are composed of a memory transfer operation (mem), followed by a kernel execution (K_i) and another memory transfer operation (mem), as described in Figure 4.1. If the two contexts were submitted at the same time with MPS, kernels K_1 and K_2 will not be launched simultaneously, and may not experience concurrency. Our framework, then, improves the probability of the kernels executing concurrently by launching then at same time, it enforces some synchronization but there is no guarantee that both will start at the same moment on the GPU.

In the future, in scenarios where GPUs can be virtualized and shared by multiple



(a) Submission using MPS

(b) Submission using our framework

Figure 4.1: kernel concurrency framework

CPUs, it is likely that they will behave as multiprogrammed devices with a lot of resource competition. Therefore, our submission framework is able to reproduce and analyze these broad competition scenarios. Our framework is built upon MPS, but takes charge of kernel launching times. Nevertheless, the actual concurrent execution will depend on the hardware scheduler and the resources required by each kernel.

The framework is implemented as a dynamic library that has to be linked to each application. The idea is to intercept the CUDA API call to launch a kernel. This call is implemented in the framework with synchronization primitives in a way that the kernel launching calls are postponed until all the kernels, that are to be submitted simultaneously, are ready to execute. After that, the kernels are submitted to MPS. Resource contention is one of the main causes of the difference in their starting time in the GPU.

4.2 Concurrent Execution

In this first study on kernel concurrent execution, the idea was to submit a number of pairs of kernels to run concurrently on the GPU and to analyse their behaviour in terms of how the simultaneous execution affects the execution time of each kernel. The great problem in this analysis was to choose which pair of kernels to submit together. If we consider the amount of kernels provided by the main GPU benchmarks, Rodinia, Parboil and SHOC, we would have 173 kernels in which the combination of pair kernels would

be intractable. Therefore, in a previous work [7], we performed a study to characterize the kernels in terms of resource usage and we proposed classifying the kernels in different groups, taking into account their characteristics in terms of resource usage, and then executed concurrently pairs of kernels from different groups in order to perform an early assessment on how kernel resource requirements have in concurrent kernel execution.

The experiments described in this chapter were performed on a GPU GTX 980 with the Maxwell architecture. The GPU has 2048 CUDA cores running at 1126 MHz in 16 SMs, with 4GB of global memory and 2MB of L2 cache. Each SM has 96KB of shared memory and 256KB of registers. To compile and run the benchmarks we used CUDA version 7.5. The statistical analysis was performed using the R language. All applications were executed with the standard input data sets.

4.2.1 Kernels Characterization

The main focus of the kernels characterization was to understand how kernels behave and their similarities in terms of resource usage. For this characterization, kernels from Rodinia[8], Parboil[42] and SHOC[15] benchmark suites were executed and profiled with Nvprof[35]. From the profiled data, we selected occupancy, efficiency, percentage of main application time, number of global memory transactions, number of shared memory transactions, number of registers and number of float, double and integer instructions performed.

The number of metrics resulted in a multidimensional space hard to extract any conclusions. We then used Principal Component Analysis (PCA) for dimensionality reduction. The PCA is a multivariate analysis technique, which consists of performing a linear transformation with a dataset so that this same set is represented by its most important components [21].

After transforming the multidimensional space into a two-dimensional space, we grouped similar kernels in categories. By using the output of PCA, we applied the K-Means algorithm to accomplish that. The K-means algorithm consists of grouping non labeled data. This algorithm takes k as input, which means the number of groups to describe the data clustering. By empirically testing different values for k, we observed four distinct groups of kernels in the three benchmark suites.

Figure 4.2 shows the biplot chart for the results of the components (PC1 vs PC2) followed by the K-means clustering ($K = 4$). The name of the kernels in this chart

is formed with a coding scheme where each kernel name starts with a letter indicating which benchmark suite it belongs (R, P or S). Observing the direction of the vectors in this chart, we can observe that the large number of kernels in SHOC influenced the position of some vectors such as the percentage of execution time and the number of double-precision floating-point instructions.

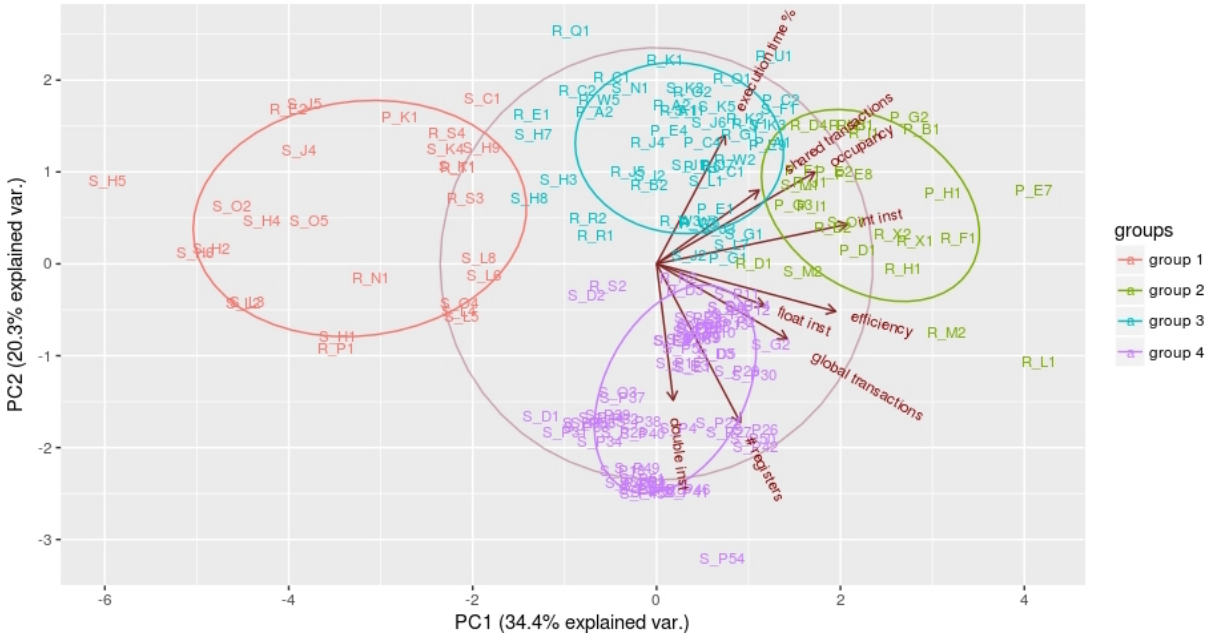


Figure 4.2: PCA and Kmeans output

Group	Parboil	Rodinia	SHOC
1	1	6	20
2	11	12	3
3	11	23	17
4	0	3	66

Table 4.1: Number of kernels for each group and benchmark suite.

Group 1 - Little Resource Usage This group contains 27 kernels, and is a result of the combination of the kernels from benchmark suites that present low use of all the resources analyzed (integer and floating-point operations, SM efficiency, GPU occupancy and memory operations). Table 4.1 shows the number of kernels of each benchmark suite that comprises this group. We can observe in this table that most of the kernels in this group belong to SHOC. Most of them are level one SHOC parallel algorithms.

Group 2 - High Resource Utilization Group 2 consists of 26 kernels that have high efficiency, due to a great number of integer and single-precision floating-point operations.

These kernels also have an average occupancy of about 70%. This is the group that presents the highest resource utilization.

Group 3 - Medium Resource Utilization This group is composed of 51 kernels, and has some similar characteristics to group 2, high number of integer operations and average occupancy around 60%. Kernels in this group presents a less significant percentage of execution time of the application they belong.

Group 4 - Low Occupancy and High Efficiency This group is composed of 69 kernels, characterized by low occupancy, high efficiency and low percentage of execution time. This group contains mostly S3D kernels from the SHOC suite. From the 69 kernels, 44 are from S3D. This application is a computational chemistry application that solves Navier-Stokes equations for a regular 3D domain [41]. The computation is floating-point intensive, and it was parallelized by assigning each 3D grid point to one thread. The low occupancy of each of its kernels impels their concurrent implementation. Another feature of this group is the smaller number of operations with integers and the highest average use of registers than the other groups.

4.2.2 Experimental Results

From the characterization analysis, we observed that Rodinia and Parboil presented more diversity in their kernels. SHOC, on the other hand, provides less diversity but it is the only suite that exploits kernel concurrency massively. The four distinct groups of kernels observed, (1) Little Resource Usage, (2) High resource utilization, (3) Medium resource utilization and (4) Low occupancy and high efficiency, provide different opportunities for concurrency. Kernels from group 1 have short execution time and low resource usage, they would allow concurrent execution, but barely share the GPU resources with other kernels since they are too fast to provide significant sharing. Kernels from group 2 and 3 present high resource utilization, which indicates that they would only allow concurrent execution when the other kernel do not compete for the same resources used, such as shared memory for example. Kernels from group 4 have medium resource utilization and relatively low occupancy. These kernels are more likely to leave unused resources and provide space for concurrent execution.

Therefore, our first assessment on the effects of the concurrent execution was to co-execute kernels from the different groups. We then proposed a second experiment where

we executed concurrently a sample of pairs of kernels from different groups. Our focus is to verify whether kernels with low or medium resource usage can take advantage of the amount of available resources in the GPU and execute simultaneously.

We selected three kernels from each group using as criterion their proximity to the centroid of the group. We believe that the kernels near the centroid are the most representative kernels of the group. The kernels selected from each group are displayed in Table 4.2. Concurrency efficiency is measured as how the simultaneous execution affects the execution time of each kernel. The concurrency efficiency, E, of kernel K_i is given by the ratio between the execution time when K_i is executed alone and the execution time when K_i is executed concurrently with K_j (as described in (4.1)). The E metric thus is a higher-is-better metric. So, when both execution times have the same value, E is equal to one, meaning that concurrency does not affect kernels execution.

$$E(K_i) = Time_{alone}/Time_{concurrent} \tag{4.1}$$

Table 4.3 shows concurrent execution effects on kernels from the 4 groups, called G1, G2, G3 and G4. We evaluate kernel pairs execution from each combination of different groups. The selected kernels are disposed in the rows and columns of the table. Each table position (K_i,K_j) displays the value of $E(K_i)$, which means the concurrency effect on kernel execution described on the row. We did not evaluate the combination of kernels from the same group, which are marked as 'X' in the table. All pairs of kernels were submitted to run at the same time in different streams according to our framework for concurrent execution. However, for some pairs, the GPU scheduler did not assign them to execute concurrently, possibly because there were no resources for the concurrent execution. We distinguish these executions in Table 4.3 as 'no'. Some $E(K_i)$ values are slightly greater than one. This is due to the difference in the execution times presented in the GPU environment.

In (K_i,K_j) concurrent execution, we observed that if K_i demands much more resources than K_j, K_i is much less penalized in its execution than K_j. Suppose that K_i takes 100 time units to run, and K_j takes 1 time unit. When they are executed together, concurrency may severely affect their execution times, and K_j can take three times more to execute. So, $E(K_j) = 0.33$, but for K_i this interference take effect in only a small part of its execution, around 3%, and the value of $E(K_i)$ would be around 0.97. This explains the $E(K_i)$ values obtained for the executions of kernels from G1 with kernels from G2, except for the R_S3 kernel from G1. R_S3 is a small kernel from G1 that performs great part

Group	Code	Benchmark	Application	Kernel Name
1	S_O5	SHOC	QtClustering	update_clustered_pnts_mask
	R_N1	Rodinia	LUD	lud_diagonal
	R_S3	Rodinia	particlefilter-float	normalize_weights_kernel
2	S_O1	SHOC	QtClustering	QTC_device
	P_E8	Parboil	mri-gridding	splitSort
	R_D2	Rodinia	dwt2d_1024	dwt_cuda::fdwt97Kernel
3	P_C4	Parboil	histo	histo_intermediates_kernel
	R_J4	Rodinia	hybridsort	bucketsort
	S_J1	SHOC	Scan	reduce
4	S_P4	SHOC	S3D	ratt_kernel
	S_P15	SHOC	S3D	ratt10_kernel
	S_E4	SHOC	GEMM	gemm_kernel2x2_tile_multiple_core

Table 4.2: Selected kernels for concurrency evaluation.

		G1			G2			G3			G4		
		S_O5	R_N1	R_S3	S_O1	P_E8	R_D2	P_C4	R_J4	S_J1	S_P4	S_P15	S_E4
G1	S_O5	X	X	X	0.598	0.497	no	no	0.611	no	no	0.927	no
	R_N1	X	X	X	no	0.603	0.795	no	0.764	no	0.719	no	no
	R_S3	X	X	X	0.940	0.933	0.888	0.689	0.713	1.006	no	0.969	0.985
G2	S_O1	0.887	no	1.145	X	X	X	0.929	1.061	no	1.023	0.869	no
	P_E8	1.012	0.895	0.704	X	X	X	1.013	0.989	no	0.940	0.952	no
	R_D2	no	1.119	1.109	X	X	X	no	0.763	no	1.077	no	1.004
G3	P_C4	no	no	1.022	0.203	0.886	no	X	X	X	no	no	0.840
	R_J4	1.003	0.893	0.989	0.250	0.635	0.894	X	X	X	0.664	0.987	no
	S_J1	no	no	0.487	no	no	no	X	X	X	no	0.344	no
G4	S_P4	no	1.051	no	0.127	0.214	0.284	no	0.837	no	X	X	X
	S_P15	0.915	no	0.910	0.109	0.432	no	no	0.967	0.903	X	X	X
	S_E4	no	no	0.694	no	no	0.285	0.299	no	no	X	X	X

Table 4.3: Efficiency provided by concurrent execution (E) on the combination of pairs of kernels from different groups. The value of E in the table position (K_i,K_j) corresponds to the efficiency on kernel K_i.

of the computation on only one thread of the block, what explains why its execution is almost not affected by concurrency.

The execution of G1 and G3 kernels, and G1 and G4 kernels resulted mostly in 'no concurrency'. This is due to the short execution time of the kernels in G1, which allows fewer opportunities for simultaneous execution. When two kernels are submitted to run at the same time, and the smallest kernel is scheduled first, the overhead of the second kernel submission may overshadow concurrency. For pairs where concurrent execution was achieved, the major drop in performance was obtained on the concurrent execution of G1 and G3, since G3 kernels are more resource demanding than G4 kernels. G4 kernels do a considerable number of arithmetic operations, especially double precision floating

point, but their low occupancy, even with the high efficiency, left space in the SMs so that the small G1 kernels are executed. A notable exception is G3 S_J1 kernel execution. It performs poorly with concurrency. This occurs because it is a reduction kernel which relies heavily on synchronization. This means that S_J1 warps are stalled great part of the time. When its warps are stalled, the warp scheduler can dispatch the other kernel warps, and this can increase S_J1 stall time.

When G2 kernels were executed with G4 kernels, concurrency has severely affected the execution times of the G4 kernels, while the kernels from G2 maintained almost the same efficiency. G4 kernels present mostly a much smaller execution time than G2 kernels, and the concurrency interference is less pronounced in G2 kernels as explained before.

Kernel pairs composed by G2 and G3 members presented interesting results. G2 and G3 comprise the most resource demanding kernels from our analysis. These groups are located near in the biplot graph of the PCA analysis shown in Figure 4.2, and G2 kernels present the greatest values for occupancy. In concurrent execution, we observed that G2 kernels were less affected by concurrency, and G3 kernels, P_C4, R_J4, and S_J1, provided distinct results. P_C4 and R_J4 presented mainly low efficiency when executed in parallel with G2 kernels, as expected. However, P_C4 execution with S_O1, and R_J4 execution with R_D2 provided particular results. In these combinations, G3 kernels executions were less affected by concurrency. This occurred because both G2 kernels, S_O1 and R_D2, include synchronization. The synchronization stall of the G2 kernel warps can create an opportunity for the G3 warps to execute. S_J1 kernel was unable to run together with any of G2 kernels. In fact, S_J1 barely can execute concurrently with other kernels, this occurs because S_J1 heavily uses the shared memory, leaving no space for other kernels to share this resource. S_J1 was able to run concurrently with R_S3 and S_P15 because these two kernels use a very small portion of the shared memory.

In the execution of G4 and G3 pairs of kernels, G4 kernels were also more affected by concurrent execution than G3, for the same reason presented for G4 and G2 execution. However, G3 kernels are smaller and use fewer resources than G2 kernels, and so the effects of concurrency on G4 kernels were also less pronounced than when they were executed together with G2 kernels. Again, the notable exception is the executions of the G3 kernel S_J1, as explained above.

4.2.3 Important Remarks

The concurrent execution showed that kernels with too many requirements on one resource may prevent concurrent execution and synchronization stall time can be used to other kernel execution. Cross-kernel interference can drastically affect performance of applications executed in GPUs concurrently. The problem is caused by concurrent access of co-located kernels to shared resources.

The results give some indications and interesting evidences on the execution interference and on what type of kernels could execute concurrently. However, these early results are far from conclusive in terms of how the resource requirements would impact the concurrent execution on GPUs. In this way, this proposes a more in depth study, using machine learning to understand the tricky relations among the kernels resource requirements and how it affects their concurrent execution.

Nevertheless, the grouping scheme previously proposed was valuable to our concurrency study. This previous study allowed us to select a controlled number of kernels that truly present different resource requirements. Without the grouping scheme, there would be an immense number of possible combinations of kernels. An uninformed pairing of the 173 kernels would make our study impractical.

Chapter 5

Using Machine Learning to Analyze the Concurrent Kernel Execution

This chapter describes the methodology to use machine learning techniques to understand how the resource requirements of the kernels impact their concurrent execution. It presents the methodology applied in form of a workflow, the techniques applied, and the hardware and software tools used to achieve the results. The chapter also describes all developed scripts that composes the described workflow.

5.1 Motivation

The previous study presented in Chapter 4 exposed that the relation between kernel pairs may not be simple to explain, specially when using a reduced set of kernels, 12 in total. It was specially hard to find the same behavior for members of one specific group co-executing with another group. The previous kernel co-executing experiment was performed by choosing the kernels by hand to feed our kernel submission framework, working with a larger set of kernels would be impossible in this way.

Therefore, to provide a more in depth analysis, this work proposes the use of 60 kernels in the concurrent experiments. The co-execution of these 60 kernels created 3600 pairs to be analyzed. The main questions this research wants to answer using machine learning are: (1) Can a pair of kernels run concurrently? (2) How their resource requirements influence in their co-execution interference? It is important to notice that the kernels resource requirements information used are known in advance, before the kernel starts its execution.

5.2 Applications

The kernels used are part of the three main GPU benchmark suites, Rodinia [8], Parboil [42] and SHOC [15]. We focus here on the execution of pairs of kernels due to the huge number of possible combinations of kernels from these benchmarks, and due to the fact that the performance gains of concurrent execution decrease with the increase in the number of co-executing kernels.

The kernels used in the experiments were selected according to the kernel categorization scheme described in Chapter 4, but differently from that experiment, this one does not try to correlate the groups' characteristics with concurrency. This experiment only relies on that characterization to create a diverse set of kernels for the co-execution. We selected 15 kernels from each category belonging to the applications described in Table 5.1, to create diversity but also ensuring that our set contains kernels with some characteristics in common. The selection criteria consisted of selecting the n, where $n = 15$, kernels closest to the k-means centroids in order to find the most representative kernels from each group.

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	

Table 5.1: Applications selected from each benchmark suite

5.3 Hardware specification

Table 5.3 describes the GPUs used in our study. The intention of using GPUs with different computational capacity are:

- The tesla P100 is used to observe how concurrency is handled in a GPU environment for high-performance computing tasks.

- The RTX 2080 is the less powerful one, but it was the most modern NVIDIA architecture at the time, and also creates a more favorable scenario to observe resource contention. Another point is that on P100, it can happen that one kernel can execute so fast that the second submission would not provide competition scenarios.

	Tesla P100	**RTX 2080**
Number of cores	3584	2944
RAM	16GB	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

Table 5.2: GPUs specifications

5.4 Workflow

The workflow of the experiment presented in this work can be divided into five steps, as described in Figure 5.1. Stages 1 and 3 are focused on data acquisition by kernel execution, standalone and concurrently, respectively. Stages 2 and 4 are the data preparation stages. Stage 5 is where data receives its last treatment and is used as an input to the classifiers for training and prediction, the output of this stage are precision, recall, accuracy and Kappa score. The two databases generated and populated during this workflow follows the structure presented in Figure 5.2 and are open for consultation on the web [1]. The following subsections (5.5 to 5.9) describes in detail each stage of the workflow.

5.5 Standalone application execution

As described in Figure 5.1, the first step consists of running all applications from the benchmark suites while using nvprof to get execution information like execution times, kernels metadata and resources requested by kernels through CUDA runtime. A script

[1] https://github.com/pablocarvalho/ml-gpu-kernel/tree/master/data

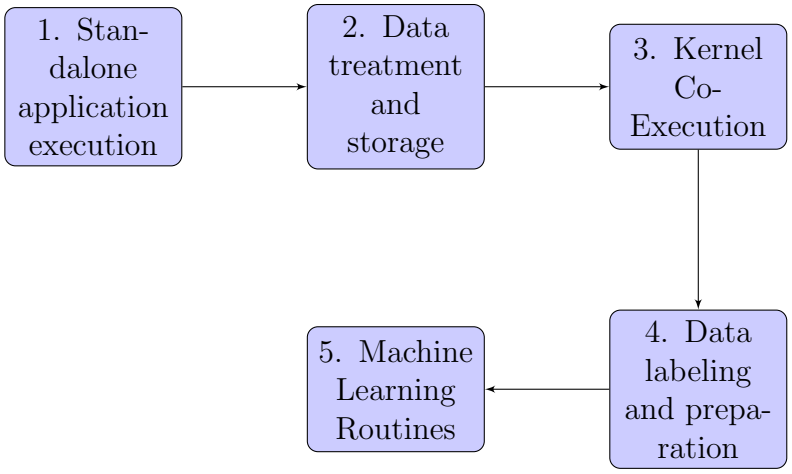


Figure 5.1: Workflow representation

(profile.py [14]) executes these steps while storing information on the kernels table present on the database. The script executes each application thirty times for statistical reliability purposes.

5.6 Data treatment and storage

This stage concerns in treating data created in step 1 after kernel standalone execution information was added to the Kernel table shown in Figure 5.2 . This stage consists of two steps:

- First, we need to establish a correlation with kernels described by Carvalho *et. al* work [7], the kernel names provided by stage 1 are all mangled names, but the original function prototypes are needed. Name mangling is performed by C++ compilers to preserve compatibility with C linkers due to C++ features not present in C. It consists of encoding a function name, scope, type and template argument into a text identifier[39]. As an example: "void dwt_cuda::fdwt53Kernel¡128, 8¿(int const*, int*, int, int, int)" , is converted in "_ZN8dwt_cuda12fdwt53KernelILi128ELi8EEE vPKiPiiii" by name mangling. What this step does is converting it back, a task handled by the script translate.py [14].

- After having all kernel prototypes, the prepare.py[14] script relates each kernel to the ones used by Carvalho *e*t al. [6] work in order to obtain kernel classification and some additional identification information.

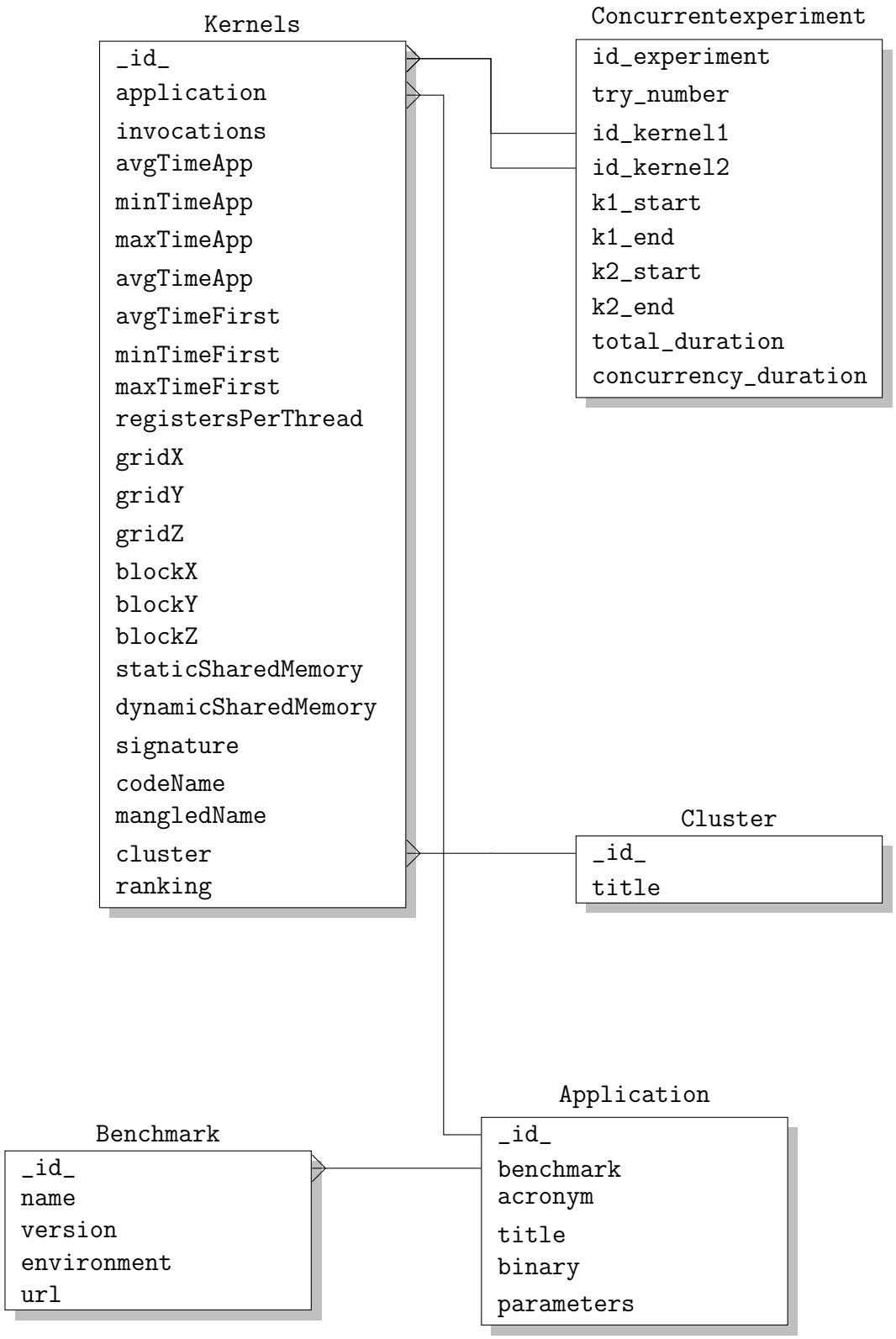


Figure 5.2: Crow's ER diagram for the experiment databases

5.7 Kernel co-execution

This experiment intention is to launch two kernels and observe its execution time in a concurrency situation. To achieve this objective, this stage of the workflow relies on a script that launches 3 processes, 2 for the application kernels and one for nvprof. First, nvprof is launched in a mode to catch information of all GPU applications that will run while its execution. Then, the kernels are launched at the same time using our kernel submission framework explained in Chapter 4. When both kernels finishes and nvprof creates its logs, our script query the logs and saves the data on our database.

5.8 Data labeling and preparation

This stage produces the input dataset for stage 5. It consists of performing a query in the database constructed and populated in the previous stages. The features used were selected following the premise that the information needed for classification should be known before the kernel was sent to the GPU, and should not rely on post execution information.

The only step that relies on post execution information is the labeling used for training and testing purposes. For the new kernels (unknown to the database), the only information needed to the classification process are the resource requests.

For comparison reasons we also prepared another input dataset using the features selected by Ravi [38], which considers the number of blocks, threads and used shared memory.

5.8.1 Data labeling for machine learning concurrency experiment

As reported in the last section, a kernel pair is executed five times in stage 3. In this step, every pair executed concurrently with a success rate of at least 80% is labeled "concurrent", the reminiscent are labeled "non-concurrent". We consider the success rate being the rate of executions that executed concurrently over all attempts. After that a SQL query [2] performs this labeling while it also queries the input variables and outputs a .csv file composed by the columns: k1_name, k2_name, k1_blocks_per_grid,

[2] https://github.com/pablocarvalho/ml-gpu-kernel/blob/master/data/querry.sql

k2_blocks_per_grid, k1_threads_per_block, k2_threads_per_block, k1_registers, k2_registers, k1_shared_mem_B, k2_shared_mem_B, classification.

5.8.2 Data labeling for machine learning interference experiment

A SQL script also performs this step, but this time we consider only the set of kernel pairs that were labeled as "concurrent" in the previous stage. The script calculates 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. (5.1), 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 doesn't worth to submit them to execute at the same time. When CE $\geqslant 1$, the concurrent execution provides performance gains when compared to sequential execution. A reason for having values bigger than one is that even for doing the same task with the same input, a kernel execution time may vary. So, in certain cases, when a kernel is not affected by the other, its execution time may be very close to the average time of its standalone execution, but not the same.

$$CE = \frac{T_{K_i} + T_{K_j}}{T_{(K_i,K_j)}} \tag{5.1}$$

5.9 Machine Learning Pipeline

The machine learning pipeline relies on scikit-learn[3] functions and XGboost[4] python implementation. The script responsible for handling this stage[5] executes all the steps on the Machine Learning pipeline explained on Chapter 2 for Multilayer Perceptron, KNN, Logistic Regression and XGBoost. Those algorithms are executed each time for a different set of variables: considering all variables, the ones used by Ravi *et al.* [38], and finally the ones selected by Recursive Feature Elimination. While executing those algorithms,

[3]https://scikit-learn.org/
[4]https://github.com/dmlc/xgboost
[5]https://github.com/pablocarvalho/ml-gpu-kernel/blob/master/stratified_classification_methods.py

grid search was also executed in order to optimize their parameter inputs. The execution of this stage generates the results to be analyzed in the next chapter.

Chapter 6

Experimental Results

This chapter describes the machine learning results, and the analysis of the concurrent execution based on the machine learning output. All the datasets and source codes are available at Github [1].

6.1 Dataset configuration

The curated dataset, as explained in session 5.8, has four resource variables for each kernel, namely *blocks per grid, threads per block, number of registers and shared memory*. Each one of those variable 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

[1] https://github.com/pablocarvalho/ml-gpu-kernel

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.

6.2 Machine 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 6.1 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.

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

Table 6.1: Successful number of pair executions for both GPUs

Then, we induced the classifier for the first task (responsible for distinguishing whether or not a pair of kernels can execute concurrently), using four features from each kernel on the curated dataset, totaling eight characteristics. As stated in Section 2.2, we experimented with four machine learning techniques: Multilayer Perceptron (MLP), k-NN, Logistic Regression (LR), and XGBoost. Table 6.2 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 [5]. Those preliminary results presented no significant difference compared to the ones obtained with the default values.

Tables 6.3 and 6.4 present the accuracy, precision, recall and kappa results for Task

KNN	neighboors = 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

Table 6.2: The hyperparameters used to train the ML models

1 on P100 and RTX-2080, respectively, considering the curated dataset. The bold values are the best ones for each metric and also the significanttly better than the rest of them. We applied statistical tests (T-Test and Wilcoxon Test [3]) 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. 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. The 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.

Figures 6.1 and 6.2 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.

	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 6.3: Predictive results for the concurrency problem on P100. The values in bold indicate the statistically significant best results.

	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**

Table 6.4: Predictive results for the concurrency problem on RTX-2080.

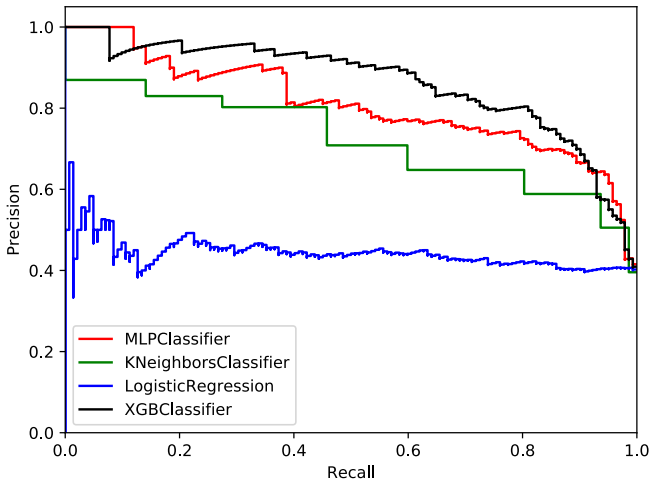


Figure 6.1: Comparing the four classifiers for the concurrency problem (P100).

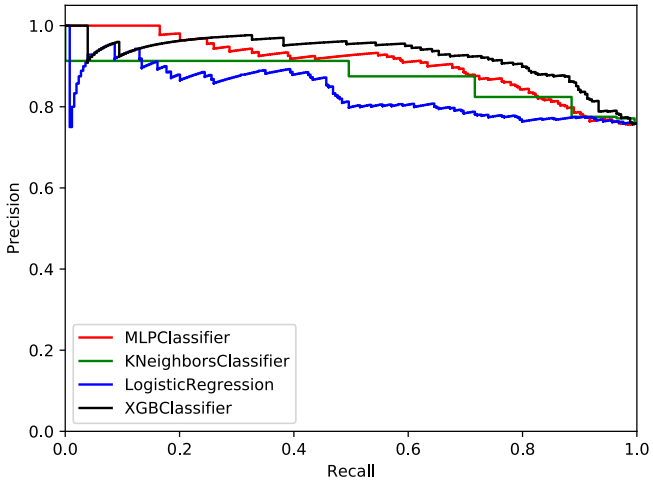


Figure 6.2: Comparing the four classifiers for the concurrency problem (RTX-2080).

6.3 Machine Learning Results for the Interference Problem (Concurrency Effect)

Given that 1,427 and 2,546 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 6.5 and 6.6, for P100 and RTX-2080, respectively. Once again, XGBoost reached the best values for accuracy, precision, recall and kappa. In comparison to the prior experiment, the CE classifier achieves on average lower metrics results, since the concurrency effect is a more difficult task to find a pattern. Similarly to the behaviour 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.)

Figures 6.3 and 6.4 show the Precision-Recall curves for the interference task, considering the P100 and RTX-2080 architectures, respectively. Here we notice a more stable behaviour 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.

	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 6.5: Predictive results of the Interference Problem (Task 2) on P100.

	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**

Table 6.6: Predictive results of the Interference Problem (Task 2) on RTX-2080.

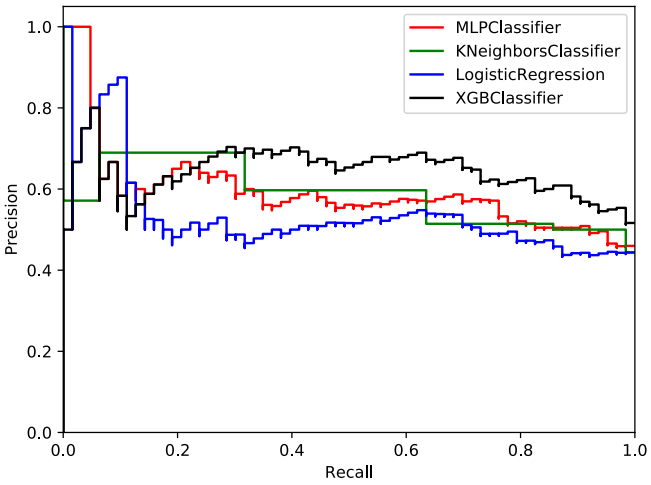


Figure 6.3: Comparing the four classifiers for Interference (P100).

6.4 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.

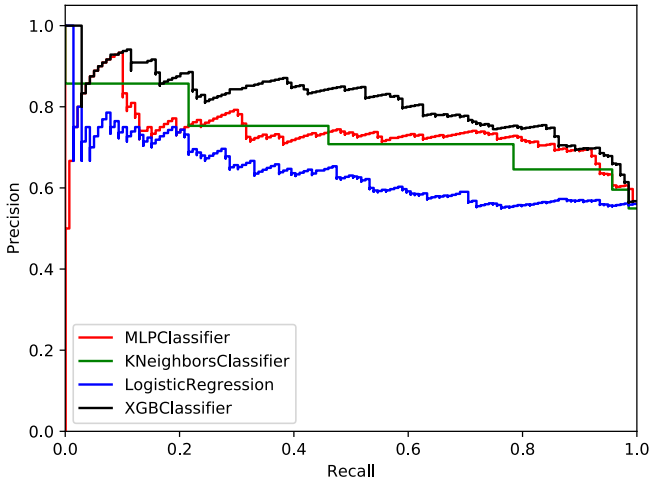


Figure 6.4: Comparing the four classifiers for Interference (RTX-2080).

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.

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 [17].

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 analysing the behaviour 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 6.7 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.

	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 6.7: 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	**0.7919**	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

Table 6.8: Predictive results of the four classifiers when using the RFE selected variables *blocks per grid*, and *threads per block* for concurrency on RTX-2080.

The features that RFE automatically selected as the most relevant differ from the ones presented by Ravi *et. al.* [38] , 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 [38]. The results related to the concurrency problem are presented in Tables 6.9 and 6.10 to P100 and RTX-2080, respectively. We can observe that the results 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, 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

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

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

shared memory is marked as not relevant. Once again, building the classifiers with only the selected variables have not improved the results, as one can see in Table 6.11, 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.

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

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.

	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**

Table 6.12: Predictive results of the four classifiers when using the RFE selected variables for Interference on RTX-2080.

We have also experimented the variables elicited in the work of Ravi *et. al.* [38] with the interference task, as presented in Tables 6.13 and 6.14 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.

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

	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**

Table 6.14: Predictive results of the four classifiers when using the RFE selected variables *blocks per grid*, and *threads per block* for concurrency on RTX-2080.

Importance of the selected features to XGBoost: Figures 6.5, 6.6, 6.7, and 6.8 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 Figures 6.5 and 6.6, 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,

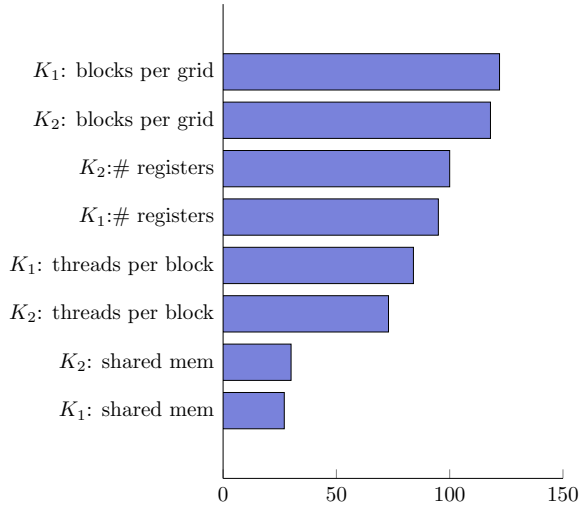


Figure 6.5: Concurrency on P100

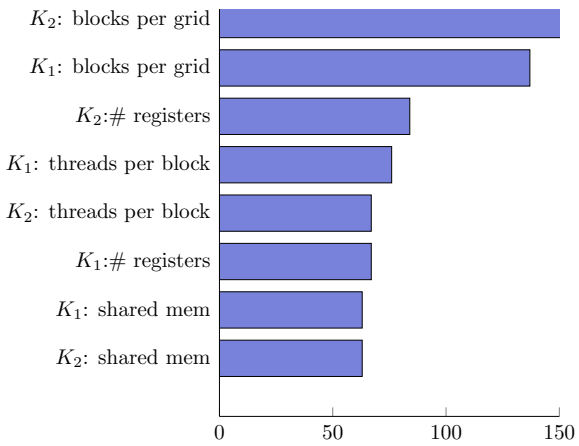


Figure 6.6: Concurrency on RTX-2080

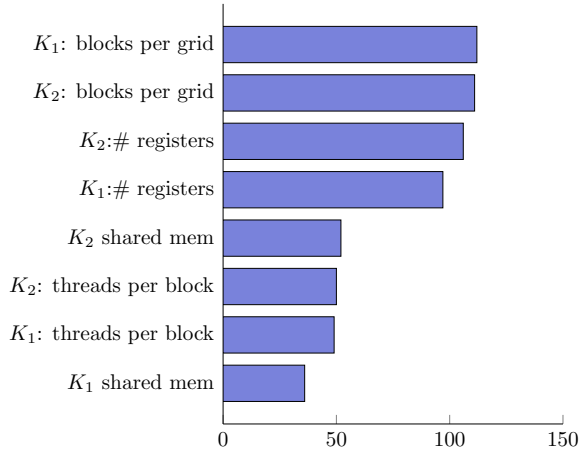


Figure 6.7: Interference on P100

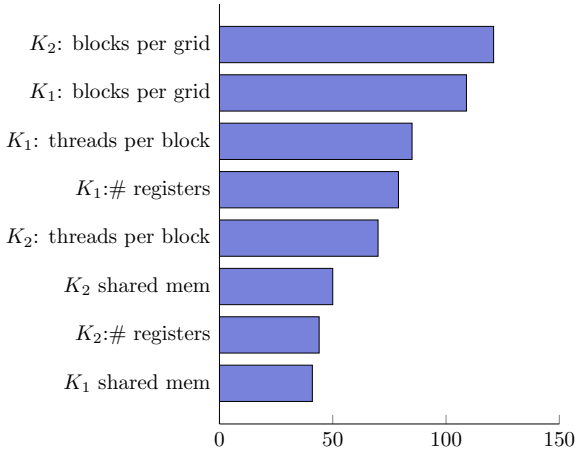


Figure 6.8: Interference on RTX-2080

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 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 Figures 6.7 and 6.8, 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.

Chapter 7

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 understand 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 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 elict new conclusions.

References

[1] Adriaens, J. T.; Compton, K.; Kim, N. S.; Schulte, M. J. The case for GPGPU spatial multitasking. In *2012 IEEE 18th International Symposium on High Performance Computer Architecture (HPCA),* (2012), pp. 1–12.

[2] Baghsorkhi, S. S.; Delahaye, M.; Patel, S. J.; Gropp, W. D.; Hwu, W.-m. W. An adaptive performance modeling tool for gpu architectures. In *ACM Sigplan Notices* (2010), vol. 45, ACM, pp. 105–114.

[3] Bauer, D. F. Constructing confidence sets using rank statistics. *Journal of the American Statistical Association 67* (1972), 687–690.

[4] Belviranli, M. E.; Khorasani, F.; Bhuyan, L. N.; Gupta, R. Cumas: Data transfer aware multi-application scheduling for shared gpus. In *Proceedings of the 2016 International Conference on Supercomputing* (2016), ACM, p. 31.

[5] Bergstra, J. S.; Bardenet, R.; Bengio, Y.; Kégl, B. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems* (2011), pp. 2546–2554.

[6] Carvalho, P.; Drummond, L.; Bentes, C.; Clua, E.; Cataldo, E.; Marzulo, L. Analysis and characterization of gpu benchmarks for kernel concurrency efficiency. *Mocskos E., Nesmachnow S. (eds) High Performance Computing. CARLA 2017. Communications in Computer and Information Science 796* (2017).

[7] Carvalho, P.; Drummond, L. M.; Bentes, C.; Clua, E.; Cataldo, E.; Marzulo, L. A. Kernel concurrency opportunities based on gpu benchmarks characterization. *Cluster Computing (on line) 1*, 1 (2019), 1–12.

[8] Che, S.; Boyer, M.; Meng, J.; Tarjan, D.; Sheaffer, J. W.; Lee, S.-H.; ; Skadron, K. Rodinia: A benchmark suite for heterogeneous computing. *In Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)* (2009), 44:54.

[9] Chen, T.; Guestrin, C. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining* (2016), ACM, pp. 785–794.

[10] Cohen, J. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement 20*, 1 (1960), 37–46.

[11] Cover, T.; Hart, P. Nearest neighbor pattern classification. *IEEE transactions on information theory 13*, 1 (1967), 21–27.

[12] Cruz, R.; Drummond, L.; Clua, E.; Bentes, C. Analyzing and estimating the performance of concurrent kernels execution on gpus. In *Anais do XVIII Simpósio em Sistemas Computacionais de Alto Desempenho* (Porto Alegre, RS, Brasil, 2017), SBC.

[13] Cruz, R. A.; Bentes, C.; Breder, B.; Vasconcellos, E.; Clua, E.; de Carvalho, P. M.; Drummond, L. M. Maximizing the gpu resource usage by reordering concurrent kernels submission. *Concurrency and Computation: Practice and Experience (on line) 1*, 1 (2018), 1–12.

[14] cudabenchs, 2019. `https://github.com/pablocarvalho/cudabenchs`;.

[15] Danalis, A.; Marin, G.; McCurdy, C.; Meredith, J. S.; Roth, P. C.; Spafford, K.; Tipparaju, V.; Vetter, J. S. The scalable heterogeneous computing (SHOC) benchmark suite. *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units* (2010), 63:74.

[16] Dao, T. T.; Kim, J.; Seo, S.; Egger, B.; Lee, J. A performance model for gpus with caches. *IEEE Transactions on Parallel and Distributed Systems 26*, 7 (2015), 1800–1813.

[17] Friedman, J.; Hastie, T.; Tibshirani, R. *The elements of statistical learning*, vol. 1. Springer series in statistics New York, NY, USA:, 2001.

[18] Guyon, I.; Elisseeff, A. An introduction to variable and feature selection. *Journal of machine learning research 3*, Mar (2003), 1157–1182.

[19] Haykin, S. S.; Haykin, S. S.; Haykin, S. S.; Haykin, S. S. *Neural networks and learning machines*, vol. 3. Pearson Upper Saddle River, 2009.

[20] Hu, Q.; Shu, J.; Fan, J.; Lu, Y. 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.

[21] Jackson, E. *A User's Guide to Principal Components, 1/E*. Willey-Interscience, 1991.

[22] Jog, A.; Kayiran, O.; Kesten, T.; Pattnaik, A.; Bolotin, E.; Chatterjee, N.; Keckler, S. W.; Kandemir, M. T.; Das, C. R. Anatomy of GPU memory system for multi-application execution. In *Proceedings of the 2015 International Symposium on Memory Systems* (2015), pp. 223–234.

[23] John, G. H.; Kohavi, R.; Pfleger, K. Irrelevant features and the subset selection problem. In *Machine Learning Proceedings 1994*. Elsevier, 1994, pp. 121–129.

[24] Li, T.; Narayana, V. K.; El-Araby, E.; El-Ghazawi, T. GPU resource sharing and virtualization on high performance computing systems. In *International Conference on Parallel Processing (ICPP), 2011* (2011), pp. 733–742.

[25] Li, T.; Narayana, V. K.; El-Ghazawi, T. 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.

[26] Liang, Y.; Huynh, H. P.; Rupnow, K.; Goh, R. S. M.; Chen, D. Efficient GPU spatial-temporal multitasking. *IEEE Transactions on Parallel and Distributed Systems 26*, 3 (2015), 748–760.

[27] Louw, N.; Steel, S. Variable selection in kernel fisher discriminant analysis by means of recursive feature elimination. *Computational Statistics & Data Analysis 51*, 3 (2006), 2043–2055.

[28] Margiolas, C.; O'Boyle, M. F. Portable and transparent software managed scheduling on accelerators for fair resource sharing. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization* (2016), ACM, pp. 82–93.

[29] Menard, S. *Applied logistic regression analysis*, vol. 106. Sage, 2002.

[30] Michalski, R. S.; Carbonell, J. G.; Mitchell, T. M. *Machine learning: An artificial intelligence approach*. Springer Science & Business Media, 2013.

[31] Mitchell, T. M., et al. *Machine learning*. Burr Ridge, IL: McGraw Hill, 1997.

[32] Nemirovsky, D.; Arkose, T.; Markovic, N.; Nemirovsky, M.; Unsal, O.; Cristal, A. 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* (2017), IEEE, pp. 121–128.

[33] NVIDIA. NVIDIA Multi-process service `https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf`.

[34] NVIDIA. Cuda multi process service overview, 2017.

[35] NVIDIA Corp. Profiler user's guide. `http://http://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof-overview`, 2017. An optional note.

[36] Ogunleye, A. A.; Qing-Guo, W. Xgboost model for chronic kidney disease diagnosis. *IEEE/ACM transactions on computational biology and bioinformatics* (2019).

[37] Pai, S.; Thazhuthaveetil, M. J.; Govindarajan, R. Improving GPGPU concurrency with elastic kernels. In *ACM SIGPLAN Notices* (2013), vol. 48, pp. 407–418.

[38] Ravi, V. T.; Becchi, M.; Agrawal, G.; Chakradhar, S. Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework. In *Proceedings of the 20th international symposium on High performance distributed computing* (2011), ACM, pp. 217–228.

[39] Samuel, A., 2020. `https://github.com/gcc-mirror/gcc/blob/master/gcc/cp/mangle.c`;.

[40] Sim, J.; Dasgupta, A.; Kim, H.; Vuduc, R. A performance analysis framework for identifying potential benefits in gpgpu applications. In *ACM SIGPLAN Notices* (2012), vol. 47, ACM, pp. 11–22.

[41] Spafford, K.; Meredith, J. S.; Vetter, J. S.; Chen, J.; Grout, R. W.; Sankaran, R. Accelerating S3D: A GPGPU case study. In *Euro-Par Workshops* (2009), Springer, pp. 122–131.

[42] Stratton, J. A.; Rodrigues, C.; Sung, I.-J.; Obeid, N.; Chang, L.-W.; Anssari, N.; Liu, G. D.; mei W. Hwu, W. Parboil: A revised benchmark suite for scientific and commercial throughput computing, 2012.

[43] Suzuki, Y.; Kato, S.; Yamada, H.; Kono, K. Gpuvm: Why not virtualizing GPUs at the hypervisor? In *USENIX Annual Technical Conference* (2014), pp. 109–120.

[44] Tarakji, A.; Gladis, A.; Anwar, T.; Leupers, R. Enhanced gpu resource utilization through fairness-aware task scheduling. In *Trustcom/BigDataSE/ISPA, 2015 IEEE* (2015), vol. 3, IEEE, pp. 45–52.

[45] Wang, J.; Gribskov, M. Irespy: an xgboost model for prediction of internal ribosome entry sites. *BMC bioinformatics 20*, 1 (2019), 409.

[46] Wen, Y.; O'Boyle, M. F. Merge or separate?: Multi-job scheduling for opencl kernels on cpu/gpu platforms. In *Proceedings of the General Purpose GPUs* (2017), ACM, pp. 22–31.

[47] Wen, Y.; O'Boyle, M. F.; Fensch, C. Maxpair: Enhance opencl concurrent kernel execution by weighted maximum matching. In *Proceedings of the 11th Workshop on General Purpose GPUs* (2018), ACM, pp. 40–49.

[48] Wen, Y.; Wang, Z.; O'boyle, M. F. Smart multi-task scheduling for opencl programs on cpu/gpu heterogeneous platforms. In *High Performance Computing (HiPC), 2014 21st International Conference on* (2014), IEEE, pp. 1–10.

[49] Wende, F.; Cordes, F.; Steinke, T. 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.

[50] Wilt, N. *The CUDA Handbook, A Comprehensive Guide To GPU Programming, 1/E*. Pearson, 2013.

[51] Wu, G.; Greathouse, J. L.; Lyashevsky, A.; Jayasena, N.; Chiou, D. Gpgpu performance and power estimation using machine learning. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on* (2015), IEEE, pp. 564–576.

[52] Yu, L.; Gong, X.; Sun, Y.; Fang, Q.; Rubin, N.; Kaeli, D. Moka: Model-based concurrent kernel analysis. In *2017 IEEE International Symposium on Workload Characterization (IISWC)* (2017), IEEE, pp. 197–206.

[53] Zhong, J.; He, B. Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling. *IEEE Transactions on Parallel and Distributed Systems 25*, 6 (2014), 1522–1532.

[54] Zien, A.; Krämer, N.; Sonnenburg, S.; Rätsch, G. The feature importance ranking measure. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases* (2009), Springer, pp. 694–709.