Available online at www.sciencedirect.com

ScienceDirect

ELSEVIER

Journal of Parallel and Distributed Computing 00 (2022) 1–15

JPDC

Evaluating Execution Time Predictions on GPU Kernels Using an Analytical Model and Machine Learning Techniques

Marcos Amaris[a], Raphael Camargo[b], Daniel Cordeiro[c], Alfredo Goldman[d], Denis Trystram[e]

[a] *Federal University of Pará, Faculty of Computing Engineering, Tucuruí — Brazil*
[b] *Federal University of ABC, Center for Mathematics, Computing and Cognition, São Paulo — Brazil*
[c] *University of São Paulo, School of Arts, Science and Humanities, São Paulo — Brazil*
[d] *University of São Paulo, Institute of Mathematics and Statistics, São Paulo — Brazil*
[e] *University of Grenoble-Alpes, Inria, CNRS, LIG, Grenoble — France*

Abstract

Predicting the performance of applications executed on GPUs is a great challenge and is essential for efficient job schedulers. There are different approaches to do this, namely analytical modeling and machine learning (ML) techniques. Machine learning requires large training sets and reliable features, nevertheless it can capture the interactions between architecture and software without manual intervention.

In this paper, we compared a BSP-based analytical model to predict the time of execution of kernels executed over GPUs. The comparison was made using three different ML techniques. The analytical model is based on the number of computations and memory accesses of the GPU, with additional information on cache usage obtained from profiling. The ML techniques Linear Regression, Support Vector Machine, and Random Forest were evaluated over two scenarios: first, data input or features for ML techniques were the same as the analytical model and, second, using a process of feature extraction, which used correlation analysis and hierarchical clustering. Our experiments were conducted with 20 CUDA kernels, 11 of which belonged to 6 real-world applications of the Rodinia benchmark suite, and the other were classical matrix-vector applications commonly used for benchmarking. We collected data over 9 NVIDIA GPUs in different machines.

We show that the analytical model performs better at predicting when applications scale regularly. For the analytical model a single parameter λ is capable of adjusting the predictions, minimizing the complex analysis in the applications. We show also that ML techniques obtained high accuracy when a process of feature extraction is implemented. Sets of 5 and 10 features were tested in two different ways, for unknown GPUs and for unknown Kernels. For ML experiments with a process of feature extractions, we got errors around 1.54% and 2.71%, for unknown GPUs and for unknown Kernels, respectively.

Keywords: Performance Prediction, Machine Learning, GPU Applications, Bulk Synchronous Parallel Model

1. Introduction

Nowadays, most HPC computing platforms have heterogeneous processing devices, such as CPUs, GPUs (Graphics Processing Units), FPGAs (Field-Programmable Gate Array), among others. GPUs are specialized processing units. They were initially conceived to accelerate vector operations, such as graphics rendering.

GPUs are often used in conjunction with multiple Central Processing Units (CPUs) to perform high-performance computing. For instance, the number of platforms in the TOP500 equipped with GPUs has increased significantly during the last years [1]. Predicting of the execution times of GPU applications is a great challenge and is necessary for efficient load-balancing and task scheduling [2]. There are different approaches to do that, such as analytical modeling, statistical approaches, and machine learning techniques.

Analytical models are useful for capturing the interaction between hardware and software, but they usually require manual intervention and a static algorithm analysis. The accuracy of a GPU performance model is subject to low-level elements such as instruction pipeline usage and small cache hierarchies. GPU performance obtains its peak when the instruction pipeline is saturated but becomes unpredictable when the pipeline is underutilized [3, 4] or when the application has an irregular behavior [5]. A performance model that considers all these particularities would be too complex and, consequently, not useful for execution time predictions. Considering the

effects of small cache hierarchies [6, 7] and memory-access divergence [8] is also critical to a GPU performance model. Changes during the execution of a GPU application will bring reconfiguration of the performance model. This reconfiguration will lead to a complex and high level analysis.

Analytical models typically are much more straightforward, capturing high-level interactions between software and hardware. An example is the Bulk Synchronous Parallel (BSP) model [9]. The BSP may serve potentially as a unified programming model for both coarse-grained and fine-grained tasks. In a previous work [10], we proposed a simple BSP-based analytical model to predict execution times of GPU applications. The model uses the number of computations and memory accesses into the GPU, with additional information on cache usage obtained from profiling. To adjust the effect of optimizations and differences in architecture a single parameter is used.

A different approach to predicting execution times is to use supervised machine learning techniques, which could capture the interactions between algorithm execution and GPU characteristics without explicit modeling. Current GPU architectures have hardware performance counters, which provide information on different levels of execution. We used these counters to determine several metrics about the throughput of processing and communication operations, and memory usage on all levels, including global memory, shared memory, and L1 and L2 caches. These metrics are typically used in the development phase of parallel applications to tune their performance. However, Job Management Systems (JMS) do not use data to generate training datasets for ML techniques, which would enable performing execution time predictions [11]. Those models can support JMs to schedule jobs and perform workload tasks in high-performance platforms.

Both models, analytical and machine learning-based, have trade-offs on their use as execution time predictors. On the one hand, analytical models have great potential to accurately predict the execution times on some specific cases, but they are hard to devise and work well only when all possible relevant parameters are captured by the model. On the other hand, machine learning-based prediction models are easier to use and may capture the main characteristics of a set of data, resulting on unstable prediction models, as good as the representativity of the training data.

In this work, we study and compare both approaches. We evaluated execution time predictions of GPU Kernels using a BSP-based analytical model and three different machine learning techniques. We used Multiple Linear Regression, Support Vector Machines and Random Forest algorithms in two scenarios: (i) using the same set of features (inputs) to the ML algorithms as the ones used by the analytical model, and (ii) applying a feature extraction process using correlation analysis and hierarchical clustering from information profiling.

We evaluated our models using 9 matrix-vector kernels and other kernels from Rodinia Benchmark suite [12]. We performed our evaluations using data from 9 GPUs, from three NVIDIA microarchitectures: Kepler, Maxwell and Pascal.

The rest of this paper is organized as follows. Section 2 presents background concepts on GPU architecture and CUDA. Section background also presents our BSP-based analytical model, and the techniques of machine learning used in this work. Section 3 presents the related works of this research, followed by Section 4, where we describe the methodology of this work. In Section 5, we describe the experimental results and, finally, the conclusions are presented in Section 6.

2. Background

2.1. Graphic Processing Unit Architectures

Since the emergence of general-purpose computing on GPUs (GPGPU), NVIDIA has launched several GPU architectures. The codename of these architectures are Tesla, Fermi, Kepler, Maxwell, Pascal, Volta, and untimely Turing. Each architecture has different capabilities summarized as the Compute Capability (C.C.) of the GPU. GPUs have multiple asynchronous and parallel Streaming Multiprocessors (SMs), which contain many Scalar Processors (SPs), specialized Special Function Units (SFUs), and load/store units. NVIDIA GPU architectures vary in a large number of features, such as the number of cores, registers, SFUs, load/store units, on-chip and cache memory sizes, processor clock frequency, memory bandwidth, unified memory spaces, among others.

The hierarchical memory of a GPU contains global and shared portions. The global memory is large, off-chip, has high latency and can be accessed by all threads of a kernel. Meanwhile, the shared memory is small, on-chip, has a low-latency and can be accessed only by threads in a same SM. Each SM has its own shared L1 cache, and new architectures have coherent global L2 caches.

Optimizing thread access to different memory levels is essential to achieving good performance [13]. From Tesla architecture, global memory access can coalesce to one transaction. In fact, Tesla architecture implemented fused multiply-add (FMA) for double precision [14]. After that, Fermi architecture implemented the IEEE 754-2008 floating-point standard [15] for both single and double precision arithmetic, which performs multiplication and addition with a single rounding step. Figure 1 shows the memory access hierarchy in a Kepler architecture.

2.2. Compute Unified Device Architecture (CUDA)

CUDA enables the use of NVIDIA GPUs for scientific and general-purpose computation. The execution flow of a CUDA application can be divided into three key stages. First, the main program executed on the CPU (called host) transfers data to the memory of the GPU; in a second stage, the host starts threads in the GPU (called device),

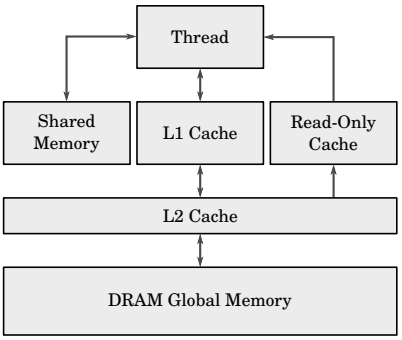

Figure 1. Memory hierarchy of threads in a kernel executed in Kepler architectures, adapted from [16]

launching a function (called kernel). Finally, the device transfers back results to the host.

A CUDA function is represented by a grid (1D, 2D or 3D) of thread blocks. Thread blocks are assigned to SMs, which can concurrently execute groups of threads called warps (a group of 32 threads is called *warp*). Threads from the same warp need to execute the same instructions. If threads from a single warp must execute different instructions (*e.g.* conditionals in the kernel), the execution takes different branches, running serially, which makes the program lose performance due to this *warp divergence*.

The CUDA language extends C language and provides a multi-step compiler. The CUDA compiler is called *NVCC*, it translates CUDA code to Parallel Thread Execution code or *PTX*. *NVCC* uses the host's C++ compiler in several compilation steps and also to generate code to be executed in the host. The final binary generated by *NVCC* contains code for the GPU and the host [13]. When *PTX* code is loaded by an application at run-time, it is compiled to binary code by the host's device driver. This binary code can be executed in the device's processing cores and is architecture-specific. The targeted architecture can be specified using *NVCC* parameters [13].

2.2.1. Performance Counters and Profile Tools

Nowadays, both GPUs and CPUs have performance counters, which are hardware units that measure different events during application execution. In the literature, they are named Performance Monitoring Units (PMU), originally built for CPU-Debugging. The main need for these counters was to analyze execution behavior at the instruction level.

Profiling is a form of dynamic program analysis that uses these performance counters. As a result of this, nowadays, it is possible to count instructions in single or double precision, branch instructions, load and store instructions, load and store throughput, arithmetic instructions, cache events (misses or hits), and many others. There are different tools to profile GPU applications. CUDA platforms provide a NVIDIA profiling tool (*nvprof*). The number of events and metrics collected varies according to the Compute Capability of each GPU model or CUDA version.

2.3. Analytical Parallel Models

The main goal of parallel computing models is to provide a standard way of describing and evaluating the performance and simulations of parallel applications. For a parallel computing model to succeed, it is paramount to consider the characteristics of the underlying architecture of the hardware used.

2.3.1. Bulk Synchronous Parallel Model

One of the most well-established models for parallel computing is the Bulk Synchronous Parallel (BSP), first introduced by Valiant in 1990 [9]. The computations in the BSP model are organized in a sequence of *supersteps*, each one divided into three successive—logically disjointed—phases. On the first phase, all processors use their local data to perform local sequential computations in parallel (i.e., there is no communication among the processors.) The second phase is a communication phase, where all nodes exchange data performing personalized all-to-all communication. The last phase consists of a global synchronization barrier that guarantees that all messages are delivered and all processors are ready to start the next superstep.

As follows, consider a BSP program that runs on S supersteps. Let g be the bandwidth of the network and L the latency for the synchronization—i.e., the minimum duration of a superstep—which reflects not only the latency of the network, but also the overhead of the synchronization step. The cost to execute the i-th superstep is then given by

$$w_i + gh_i + L \tag{1}$$

where w_i is the maximum amount of local computations executed, and h_i is the largest number of packets sent or received by any processor during the superstep. If $W = \sum_{i=1}^{S} w_i$ is the sum of the maximum work executed on all supersteps and $H = \sum_{i=1}^{S} h_i$ the sum of the maximum number of messages exchanged in each superstep, then the total execution time of the application is given by:

$$T = W + gH + LS \tag{2}$$

It is common to present the parameters of the BSP model as a tuple (w, g, h, L). This model has been implemented as API libraries, programming languages [17] and API libraries have been developed to use BSP on GPU architectures [18].

2.3.2. A BSP-based Model for GPU Applications

We proposed a simple BSP-based model to predict performance in GPU applications [10]. This model abstracts all the heterogeneity of GPU architectures and single optimizations that GPU applications can perform in a parameter λ. Equation 3 shows the predicted running time of a kernel T_k. Equations 4 and 5 determine communication values in the global memory and shared memory, respectively.

$$T_k = \frac{t \cdot (Comp + Comm_{GM} + Comm_{SM})}{R \cdot P \cdot \lambda} \quad (3)$$

$$Comm_{GM} = (ld_1 + st_1 - L1 - L2) \cdot g_{GM} + L1 \cdot g_{L1} + L2 \cdot g_{L2} \quad (4)$$

$$Comm_{SM} = (ld_0 + st_0) \cdot g_{SM} \quad (5)$$

where t is the number of threads launched. $Comp$ is the number of processing cycles spent by each thread. $Comm_{GM}$ is the communication cost of global memory accesses of one thread (Equation 4). $Comm_{SM}$ is the communication cost of shared memory accesses of one thread (Equation 5). R is the clock rate, P is the number of cores of a specific GPU, and λ models the effects of application optimizations.

Regarding communications, g_{SM}, g_{GM}, g_{L1}, and g_{L2} are constants representing the latency in communication over shared, global, L1 cache, and L2 cache memory, respectively. ld_0 and st_0 represent the average number of load and stores for one thread in the shared memory, and ld_1 and st_1 global memory. $L1$ and $L2$ are average cache hits in $L1$ and $L2$ cache for one thread. $L1$ caching in Kepler and Maxwell architectures are reserved for register spills in local memory.

The parameter λ captures the effects of thread divergence, global memory access optimizations, and shared memory bank conflicts. λ is used to adjust the predicted application execution time with the measured one and is defined as the ratio of these values. It needs to be measured only once, for a single input size and a single board. The same λ should work for all input sizes and boards of the same architecture [10].

Intra-block synchronization is very fast and did not need to be included. Nevertheless, we maintained the inspiration on the BSP-model because extended versions of the model for multiple GPUs need global synchronizations.

2.4. Machine Learning Techniques

Machine learning refers to a set of techniques for understanding data and performing predictions. When used for prediction, supervised models—which involve building a mathematical equation to estimating an output based on one or more inputs—are normally used. Regression techniques are used when the output is a continuous value. The metric to evaluate errors in performance prediction using regression techniques is commonly the Mean Absolute Percentage Error (MAPE). Below, the equation of MAPE.

$$MAPE = \frac{100}{N} \sum_{n=1}^{N} \left| \frac{M_n - F_n}{M_n} \right| \quad (6)$$

Where M_n are the measured values, and F_n are the forecasted values, and N is the number of all the compared samples.

In this paper, we used three different machine learning methods: Linear Regression, Support Vector Machines and Random Forests.

2.4.1. Linear Regression (LR)

Linear regression is a straightforward technique for predicting a quantitative response of Y based on single or multiple predictor variables X_p. It assumes that there is approximately a linear relationship between each X_p and Y. It gives each predictor a separate slope coefficient in a single model. Mathematically, we can write the multiple linear regression model as

$$Y \approx \beta_0 + \beta_1 X_1 + + \beta_2 X_2 + \ldots + + \beta_p X_p + \epsilon \quad (7)$$

where X_p represents the p-th predictor and β_p quantifies the association between that variable and the response.

2.4.2. Support Vector Machines (SVM)

Support Vector Machines is a widely used technique for classification and regression problems. It belongs to the general category of kernel methods, which are algorithms that depend on data only through dot-products. The dot product can be replaced by a kernel function, which computes a dot product in a possibly high dimensional feature space Z. It maps the input vector x into the feature space Z though nonlinear mapping. The regression is then performed on this higher dimensional space.

2.4.3. Random Forests (RF)

Random Forest algorithms belong to decision tree methods, capable of performing both regression and classification tasks. In general, a decision tree with M leaves divides the feature space into M regions K_m, $1 \leq m \leq M$. The prediction function of a tree is then defined as $f(x) = \sum_{m=1}^{M} c_m I(x, K_m)$, where M is the number of leaves in the tree, K_m is a region in the features space, c_m is a constant corresponding to region m and I is the indicator function, which is 1 if $x \in K_m$, 0 otherwise. The values of c_m are determined in the training process. The random forest consists of an ensemble of decision trees and uses the mode of the decisions of individual trees.

2.4.4. Feature Extraction Techniques

Correlation techniques and hierarchical clustering algorithms can be used in the feature extraction phase to reduce dimensionality. Here, we show a short theoretical background of the techniques used in this work.

There are different correlation functions, among them, Pearson and Spearman. Pearson's correlation evaluates the linear relationship between two continuous variables. It is commonly represented by the letter r when it is used for samples and is given by:

$$r = \frac{cov(X, Y)}{\sigma_X \sigma_Y}, \quad (8)$$

where $\text{cov}(X, Y)$ is the covariance between two vectors, X and Y, σ_X is the standard deviation of X, and σ_Y is the standard deviation of Y.

The Spearman's correlation evaluates the monotonic relationship between two continuous or ordinal variables. It is defined as the Pearson correlation coefficient between the ranked variables [19]. For a sample of size n, the n raw scores X_i, Y_i are converted to ranks $\text{rg}X_i$ $\text{rg}Y_i$. Here, we denote this correlation as:

$$r_s = \frac{cov(rgX, rgY)}{\sigma_{rgX}\sigma_{rgY}}, \tag{9}$$

where $\text{cov}(rgX, rgY)$ is the covariance of the rank variables, and σ_{rgX} and σ_{rgY} are the standard deviations of the rank variables. Spearman correlation considers relations in different scales and spaces of the features due to the use of rank variables.

Hierarchical clustering creates groups from a distance matrix. Different metrics or distance functions exist in the literature, among them, Euclidean, Manhattan, Canberra, Binary, or Minkowski. Besides, correlation functions can also be used. A dendrogram is a tree diagram frequently used to illustrate the arrangement of the clusters produced by a hierarchical clustering algorithm. Figure 2 shows a dendrogram of 10 features with their respective distance matrix using a Spearman correlation function.

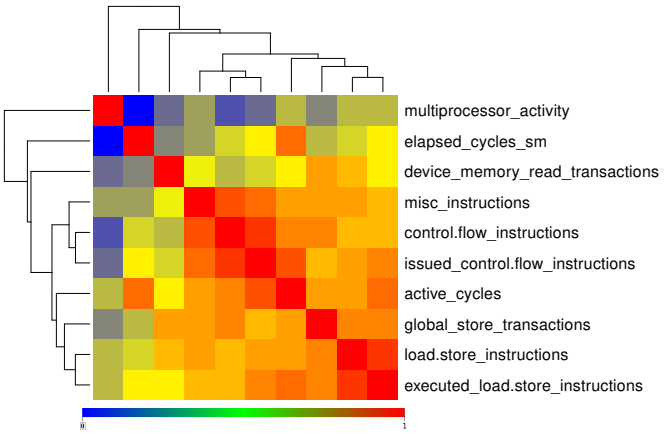

Figure 2. Heatmap and Dendrogram of a correlation matrix with some features of our experiments

3. Related Works

In recent years, studies comparing GPU performance predictions using analytical modeling, statistical analysis, and machine learning approaches have become popular. Zhang et al. [20] presented a statistical approach using Random Forest on the performance and power consumption of OpenCL applications in an ATI GPU.

Meswani et al. [21] predicted the performance of HPC applications on hardware accelerators such as FPGA and GPU from applications running on CPU. Those predictions were made by identifying common compute patterns or idioms, then developing a framework to model the predicted speedup when the application is run on GPU or FPGA using these idioms. An idiom is a pattern of computation and memory access that may occur within an application.

Karami et al. [22] proposed a statistical performance prediction model for OpenCL kernels on NVIDIA GPUs using a regression model for prediction and principal component analysis for extracting features of higher weights, thus reducing model complexity while preserving accuracy.

Baldini et al. [23] showed that ML techniques can predict GPU speedup of OpenCL applications from OpenMP applications. They used K-nearest neighbor and SVM as a classifier to know the performance of these applications over different GPUs. They showed that a small set of easily-obtainable features could predict the magnitude of GPU speedups on two different high-end GPUs, with accuracies varying between 77% and 90%, depending on the prediction mechanism and scenario.

Wu et al. [24] described a GPU performance and power estimation model, using K-means to create sets of scaling behaviors representative of the training kernels and neural networks that map kernels to clusters, with experiments using OpenCL applications over AMD GPUs. They only worked with neural networks, and one neural network is built and trained per cluster set. This increases the complexity of the process depending of the number of clusters organized by K-means. They tested their performance and power models over a AMD hardware with errors within 15% and 10%, respectively.

Madougou et al. [25] compared analytical models, statistical approaches, quantitative methods, and compiler-based methods. They tested three kernels between them Matrix Multiplication over four GPUs from two generations of NVIDIA GPUs (GTX480, Tesla C2050, GTX-Titan, and/or K20). They concluded that modeling GPU Applications is on the critical path to the generalized adoption of heterogeneous platforms.

New approaches have become widely used recently, for instance, in [26], Konstantinidis and Cotronis proposed in 2017 a quadrant-split visual representation, which captures the characteristics of multiple processors in relation to a specific kernel. Authors designed a quantitative roofline model for GPU kernel performance. They performed experiments on stencil computation, matrix multiplication kernel and a total of 28 kernels of the Rodinia benchmark suite. They used six Nvidia GPUs (GTX-480, GTX-660, GTX-960, GTX-1060, Tesla S2050, Tesla K20c) and showed an absolute error in predictions of 27.66% in the average case.

Another interesting approach was taken by Alavani *et. al.* [27], where they built an analytical model for predicting execution times of CUDA kernels using static analysis of PTX code. Their experimental analysis was done over a set of 45 CUDA kernels and a Nvidia GPU Tesla-K20. In general, they presented a mean absolute error of 26.86% they predictions are compared to the measured execution

time.

Some works proposed using machine learning techniques to predict the performance of GPU applications from CPU applications. Ardalani *et. al.* [28] proposed Cross-archi-tec-ture Performance Prediction (XAPP), a machine learning based technique that uses single-threaded CPU implementation to predict GPU performance. They correlated CPU and GPU features from a limited set of applications and used these features in a machine learning ensemble method. They used Rodinia Benchmark and 2 GPUs (Maxwell GTX-750 and Kepler GTX-660Ti). They presented average error of 14% and 17%.

Recent efforts to modeling performance of GPU application were made by Yehia *et. al.* and Lorenz *et. al.* [29, 30]. Lorenz *et. al.* worked with 189 CUDA kernels from benchmarks such as Parboil, Rodinia, Polybench-GPU, and SHOC. They used 5 GPUs (Tesla-K20, Titan-XP, Tesla-P100, Tesla-V100, and GTX-1650) from 4 microarchitectures (Kepler, Pascal, Volta, and Turing). They used portable code features and independent GPU information to predict GPU kernel execution times using Random Forests. To reduce overhead during the data acquisition, the authors profiled the applications using CUDA Flux profiler [31]. Their time predictions yielded a median Mean Average Percentage Error (MAPE) of 8.86–52.0%.

Yehia *et. al.* proposed a scalable performance prediction toolkit for GPUs, named PPT-GPU. PPT-GPU is built in order to create a set of cycle-approximate analytical models to predict compute and memory performance of CUDA kernels. Memory models predicted memory metrics such as memory requests, transactions, hit rates, and divergence; and compute models predicted computational metrics such as cycles, instructions executed, and occupancy. They traced the executions of 27 applications (1034 kernels) over an Nvidia GPU with microarchitecture Volta, used the traces to predict kernel performance on a Turing GPU, and obtained error MAPE around 16% for execution cycles (run time).

Generally, authors defend that reasonable predictions of computing workloads are around 30% of error [32]. Predictions around 5% of error are acknowledged as excellent results. It could be cheaper, computationally, to find high correlation features in profile information to predict performance of GPU functions, as we present in this work. We considered predictions for both unseen GPUs and kernels to evaluate the reliability of the selected features. Besides, this work also compares Machine Learning techniques with a simple analytical model, which is based in the Bulk Synchronous Parallel model.

4. Method and Materials

In this work, we evaluate and compare the performance prediction of GPU applications using a BSP-based analytical model [10] and three different machine learning algorithms: Linear Regression, Support Vector Machine, and Random Forest. We evaluated the machine learning algorithms using two different scenarios defined by the set of features used. In scenario (i), ML algorithms used the same set of features (inputs) defined by the BSP-based analytical model. In contrast, in the scenario (ii), we propose a feature extraction process to select the best features to be used by the machine learning algorithms.

We evaluated the performance prediction methods using a set of 20 CUDA kernels. 9 matrix-vector algorithms commonly used for benchmark purposes and 11 CUDA kernels belonging to 6 applications from the Rodinia benchmark suite [12] executed on 9 different NVIDIA GPUs, including Kepler, Maxwell and Pascal architectures.

In the remainder of this section, we will present the experimental evaluation of our performance predictors using a set of applications whose performances were measured on different GPUs. Section 4.1 presents the chosen applications and the GPU platforms used on our experimental testbed. Section 4.2 presents dataset which was crated running these applications on all GPUs, which was used as ground truth to build and evaluate the accuracy of the predictors. We showed the results with the BSP-based analytical predictor in Section 4.3, while Section 4.4 presents the results obtained using our machine learning predictors.

4.1. Algorithm and GPU Testbeds

Collected data, experimental results, and source codes are publicly available[1] under a Creative Commons Public License for the sake of reproducibility.

4.1.1. Algorithm Testbed

Our algorithm testbed is composed of 20 CUDA kernels, 9 classical Matrix-Vector algorithms and 11 CUDA kernels belonging to 6 applications of the Rodinia benchmark suite. Below, we discuss some details of these algorithms, and introduce a series o code letters for some kernels and applications. Our benchmark contains two different algorithms for matrix addition (**M**), four different algorithms for Matrix Multiplication (**MM**) (these kernels change in the access memory pattern. Letters G and S describe whether the CUDA functions use merely Global memory or Share memory too. Letters C and U denote whether the accesses into memory coalesce or uncoalesce (non-coalesced)), one for vector addition (DotP), one for matrix dot product (VAdd) and one for the maximum subarray problem (MSA) [33]. Table 1 summarizes each application. This table shows the thread hierarchy and the request shared memory per block in each kernel. Columns *dimGrid* and *dimBlock* show the thread hierarchy solution of each kernel. BS is the block size, and GS is the grid size. Column *Shared_Mem* shows the size of shared memory that each kernel needs per block during its execution.

[1] Hosted at GitHub: `https://github.com/marcosamaris/gpuperfpredict` [Accessed on May 2020]

Kernel	*dimGrid*	*dimBlock*	*Shared_Mem* (Bytes)
MMGU	(GS, GS, 1)	(BS, BS, 1)	0
MMGC			0
MMSU			$(BS^2 \times 2 \times 4B)$
MMSC			$(BS^2 \times 2 \times 4B)$
MAU	(GS, GS, 1)	(BS, BS, 1)	0
MAC			
VAdd	(GS, 1, 1)	(BS, 1, 1)	0
dotP	(GS, 1, 1)	(BS, 1, 1)	$(BS \times 4B)$
MSA	(48, 1, 1)	(128, 1, 1)	$(4096 \times 4B)$

Table 1. MM and MA mean Matrix multiplication and Matrix addition. G does not use Share memory. S means that the application uses shared memory. C memory access coalesce, and U memory access do not coalesce. GS and BS are the sizes of the grid and block of threads.

The testbed also contains 11 CUDA kernel functions used by 6 applications from the Rodinia Benchmark Suite. Rodinia is well accepted in the GPGPU community [34, 35], and is used in GPU simulators [36]. In total, we used 20 CUDA kernels to test our prediction methods. In this work, we have chosen the following applications from the Rodinia suite: Back Propagation (BCK), Gaussian (GAU), Heartwall (HLW), Hotspot (HOT), LU Factorization (LUD), and Needleman-Wunsch (NDL). Table 2 characterizes the Rodinia applications according to whether they are iterative or not, their number of parameters, their kernel names, and the number of collected samples per kernel.

Application	Iter	Param.	Kernels	Samples
BCK	No	1 - [57]	layerforward	57
			adjust-weights	
GAU	Yes	1 - [32]	Fan1	34800
			Fan2	
HWL	Yes	1 - [84]	heartWall	5270
HOT	Yes	2 - [5,4]	calculate-temp	396288
LUD	Yes	1 - [32]	diagonal	8448
			perimeter	8416
			internal	8416
NDL	Yes	2 - [16,10]	needle-1	21760
			needle-2	21600

Table 2. Rodinia applications used in the experiments

The number of samples of kernel executions depends of each application configuration. For each application, we have executed the kernels using different values for the parameters (the numbers of different parameter values are inside the brackets). For instance, application HOT receives two parameters, the first evaluated with 5 different values and the second with 4, resulting in 20 different experiments in each GPU. All applications except BCK invoke their kernels multiple times. In this sense, 1 parameter is changed 57 times in the application BCK, so the number of samples – represented in column **Samples** is 57. The application HWL was executed 84 times, but the execution of its only kernel is iterated – represented by the columns **Iter** – for this reason, the number of samples is much larger.

4.1.2. GPU Testbed

We executed each algorithm on all GPUs listed on Table 3. We have 9 different GPUs comprising three NVIDIA microarchitectures: Kepler (Compute Capability 3.X), Maxwell (Compute Capability 5.X), and Pascal (Compute Capability 6.X).

Model	C.C.	Memory	Bandwidth	L2 (MB)	Cores/SM
GTX-680	3.0	2 G	192.2 G/s	0.5	1536/8
Tesla-K40	3.5	12 G	276.5 G/s	1.5	2880/15
Tesla-K20	3.5	4 G	200 G/s	1	2496/13
Titan	3.5	6 G	288.4 G/s	1.5	2688/14
Q K5200	3.5	8 G	192.2 G/s	1	2304/12
Titan X	5.2	12 G	336.5 G/s	3	3072/24
GTX-970	5.2	4 G	224.3 G/s	1.75	1664/13
GTX-980	5.2	4 G	224.3 G/s	2	2048/16
Pascal-P100	6.0	16 G	732 G/s	4	3584/56

Table 3. Hardware specifications of the GPUs in the testbed

4.2. Samples (Execution times and Profile Information)

We collected, separately, the set of events, metrics, and execution times of each application in Section 4.1.1, using different machines with a Linux operating system. Overall GPUs are shown in Table 3.[2]

We collected all data using the CUDA profiling tool **nvprof**, which enabled us to collect data from the command-line, reducing the overhead. We performed this process without making modifications to the application source codes. We iterated all the applications over their parameter space and executed them on the machines. The number of events and metrics varied according to the compute capability of the device. All collected information resulted in 2.5 GB of data, and the data collected over each GPU took about two weeks. During our evaluation, we executed all applications using the CUDA profiling tool *nvprof*. Solely the execution time of each sample is used to test our analytical model. Machine learning algorithms had as input the features which were collected in the profiling process, and Section 4.4 explains this. Each experiment is presented as the average of ten executions, with a confidence interval of 95%.

We cleaned the collected information, dropping features with no variation among the whole dataset, and selecting the features with representation in all the GPUs.

[2] We executed exclusively Rodinia applications over the GPU Tesla P100 due to usage constraints.

Each cleaned sample of the dataset resulted in 85 execution features (profile information) plus 11 GPU architecture features (bandwidth, clock rate, number of SMs, number of cores, L2 size, among others). See Table 10 for a total of 96 features.

4.3. The Analytical model

We evaluated the model using the applications presented in Table 1, written in CUDA, and executing a single kernel. For matrix multiplication, we used 4 kernels with different optimizations; and for matrix addition we used 2 kernel implementations. The Matrix-Vector algorithms Dot Product, Vector addition, and Maximum Subarray Problem applications have a single kernel version.

Thus, our BSP-based model was implemented using 9 classical matrix-vector algorithms and 6 from Rodinia benchmarks suite, shown in Table 2. We did not use our model over the Kernels of the application LUD and NDL, because those kernels executed iteratively and their internal configurations changed in each iteration.

For the CUDA kernels from Rodinia, each execution of the back-propagation application resulted in a single sample (kernel execution) of its 2 kernels. Each execution of the other Rodinia applications generated multiple samples (kernels executions) for our experiments since these kernels are executed inside loops.

The number of computational cycles (*comp*) and the communication costs (ld_0, st_0, ld_1 and st_1) were extracted from the application source codes, information about cache hits in cache L1 and L2 were extracted from profiling. We also confirmed the usage of FMA and SFU instructions. The process of extracting the parameter values of the analytical model is explained below.

4.3.1. Vector-Matrix Applications

We collected 69 samples for one-dimensional problems (i.e. vAdd, dotP, MSA) and 32 samples for two-dimensional problems. The sizes of one-dimensional problems range from 2^{23} to 2^{28}, with a step of 2^{22}. The size of two dimensional problems range from 2^8 to 2^{13}, with a step of 2^8. We used this information to compute the values of equations 3, 5, and 4.

For matrix multiplication, *comp* was determined by the number of multiplications and/or operations computed by a thread. In this case, each thread performs N FMA single-precision operations. IEEE 754-2008 floating-point standard [15] states that those operations need a single rounding step. The value of *comp* is the same for all four optimization modes since they differ only in the memory access patterns. With those values, we could compute — using equations 5 and 4— the values of *comp*, $comm_{GM}$, and $comm_{SM}$. These variables were then multiplied by the number of threads t in the kernel execution and divided by the number of Processors P times the clock rate R of the GPU.

For matrix addition, each thread requests 1 element from the matrix and computes a single addition. The values of the parameters of the model for (MAU) and (MAC) are $comp = 1$, $ld_1 = 2$, $st_1 = 1$, $ld_0 = 0$ and $st_0 = 0$. We determined the parameter values for vector addition in the same way.

The dot product application performs a reduction sum operation after a vector product. The multiplication uses data stored in global memory. The reduction is performed using the shared memory and requires $log(block\_size)$ steps, where $block\_size$ is the number of threads per block.

The kernel of the Maximum Sub-Array Problem (MSA) is implemented using the CGM model. It has several branch divergences in the source code, but its execution time regularly increases with input size. This kernel uses 4096 threads, divided into 32 thread blocks with 128 threads on each. The N elements are divided into intervals of N/t elements, one per block, and each block receives a portion of the array. The blocks use the shared memory for storing segments of their interval, which are read from the global memory using coalesced accesses. The values of the parameters of the model for (MSA) are shown in Table 4.

Par.	**MM**				**MA**		**vAdd**	**dotP**	**MSA**
	MMGU	**MMGC**	**MMSU**	**MMSC**	**MAU**	**MAC**			
comp	$N\cdot$ FMA				$1 \cdot 24$			$1 \cdot 96$	$(N/t) \cdot 100$
$\mathbf{ld_1}$	$2 \cdot N$				2			2	N/t
$\mathbf{st_1}$	1				1			$1/GS$	5
$\mathbf{ld_0}$	0		$2 \cdot N$		0			$log(BS)$	N/t
$\mathbf{st_0}$	0		1		0			$log(BS)$	N/BS

Table 4. Values of the model parameters for the matrix-vector applications

Different published micro-benchmarks were used to establish the number of cycles per computation and communication operations in the GPUs [37], with FMAs, additions, multiplications, and divisions taking 2, 24, 32 and up to 96 cycles of the clock. For all simulations, we also considered 5 cycles for latency in the communication for shared memory and 500 cycles for global memory [13].

The optimizations affect merely the performance of the communication between threads. As explained above, λ is obtained by the ratio of the predicted execution time of the application, when considering $\lambda = 1$, with the actual measured execution time. The parameter λ captures the effects of thread divergence, global memory access optimizations, and shared memory bank conflicts. It needs to be measured simply once, for a single input size and a single board. The same λ should work for all input sizes and boards of the same architecture. Table 5 shows the obtained λ values. Gray cells denote boards with Kepler architecture, and the other denotes boards with Maxwell architecture. We can see that the λ values for different boards of the same architecture are very close.

4.3.2. Rodinia Benchmark Applications

We evaluated our analytical model using four Rodinia algorithms: Back-propagation (BCK), Gaussian Elimination (GAU), Heartwall (HWL), and Hotspot (HOT). Sim-

GPUs	MM				MA		vAdd	dProd	MSA
	MMGU	MMGC	MMSU	MMSC	MAU	MAC			
GTX-680	4.50	19.00	20.00	68.00	1.50	9.25	14.00	11.00	0.68
Tesla-K40	4.30	20.00	19.00	65.00	2.50	9.50	5.50	10.00	0.48
Tesla-K20	4.50	21.00	18.00	52.00	2.50	9.00	6.00	10.00	0.55
Titan	4.25	21.00	17.00	50.00	2.50	10.00	5.50	12.00	0.48
Quadro	4.75	20.00	20.00	64.00	1.50	8.25	11.00	9.50	0.55
TitanX	9.50	36.00	36.00	110.00	3.00	9.50	7.00	9.75	0.95
GTX-970	13.00	44.00	46.00	120.00	3.75	9.50	8.00	10.50	1.95
GTX-980	13.00	44.00	46.00	120.00	3.25	9.50	7.00	9.50	1.50

Table 5. Values of the parameter λ for each matrix-vector CUDA kernel in the GPUs used

ilarly to the matrix-vector applications mentioned before, the variables of computation (*comp*) and communication (ld_0, st_0, ld_1 and ld_1) were extracted from the application source codes. For the (HWL) kernel, it was not possible to extract parameters for determining the computation and communication costs from the source code, due to a large number of code lines. For this reason, these were extracted from profile information. Table 6 shows the parameter values for all Rodinia aplications.

Finally, when the values of the model parameters were completed, we executed a single instance of each application on each GPU to determine the λ values, shown in Table 7. Cell colors soft gray, hard gray, and white represent Kepler, Maxwell, and Pascal architectures. Like classical applications, we can notice that λ values are very close when they belong to the same architecture.

Par.	Back propagation		Gaussian		HWL	HOT
	BCK-1	BCK-2	GAU-1	GAU-2		
comp	404	116	36	96	760	500
$\mathbf{ld_1}$	2	4	1	3	7000	2
$\mathbf{st_1}$	1/GS	1	1	1	2000	1
$\mathbf{ld_0}$	BS	0	0	0	2800	2
$\mathbf{st_0}$	BS	0	0	0	1	1

Table 6. Parameter values of the BSP model over 6 CUDA kernels of Rodinia Benchmark Suite

GPUs	Back propagation		Gaussian		HWL	HOT
	BCK-1	BCK-2	GAU-1	GAU-2		
GTX-680	13.20	7.50	0.13	0.65	1.07	20.00
Tesla-K40	13.50	8.50	0.17	1.25	1.12	35.00
Tesla-K20	13.80	8.50	0.17	1.25	1.25	36.25
Titan	13.20	8.50	0.17	1.25	1.12	35.00
Quadro	12.00	7.00	0.12	0.70	0.83	20.00
TitanX	9.00	5.50	0.20	2.00	1.62	17.50
GTX-970	12.00	6.50	0.35	3.50	2.50	20.00
GTX-980	9.60	5.75	0.23	2.50	1.88	17.50
Tesla-P100	18.00	10.50	0.15	2.00	2.75	62.50

Table 7. Values of the parameter λ for each Kernels of the Rodinia Benchmark in the GPUs used

In all experiments, we modeled only communication over global memory and shared memory. We did not include the values of L2 cache for these experiments because they did not impact the variance of the predictions. Regarding L1 cache, it is reserved for register spills in local memory in Kepler, Maxwell, and Pascal architectures.

4.4. Machine Learning

This section presents the methodological process undertaken to use machine learning (ML) techniques to predict GPU kernel execution times. Three different machine learning algorithms were used: Linear Regression, Support Vector Machine, and Random Forest. We followed two scenarios for selecting which features to use: (i) we utilized merely those features used in the analytical model (AM); and (ii) we performed a feature extraction over all available features obtainable from execution profiling.

We used R Language [38] in conjunction with `e1071` [39] and `randomForest` [40] packages, which contain the `SVM` and `randomForest` functions, respectively. The source code can be found in the repository which was mentioned above. For Random Forest, we set the number of trees to 50, and the number of predictor candidates at each split to 3, as these values resulted in better predictions and faster executions.

4.4.1. Scenario 1: Prediction using features from the analytical model

We first applied ML algorithms using the same communication and computation parameters from the analytical model. The objective was to compare both approaches (ML and AM) trying to ensure fairness in the comparison.

We performed the evaluation using cross-validation, that is, for each target GPU, we performed the training using the other GPUs. This process was done for each application separately. The features that we used to feed the Linear Regression, Support Vector Machines, and Random Forest algorithms are presented in Table 8. To generate the features `totalLoadGM`, `totalStoreGM`, the number of requests was divided by the number of transactions per request.

Feature Name	Description
`num_of_cores`	Number of cores per GPU
`max_clock_rate`	GPU Max Clock rate
`Bandwidth`	Theoretical Bandwidth
`Input_Size`	Size of the problem
`totalLoadGM`	Load transaction in Global Memory
`totalStoreGM`	Store transaction in Global Memory
`TotalLoadSM`	Load transaction in Shared Memory
`TotalStoreSM`	Store transaction in Global Memory
`FLOPS SP`	Floating operation in Single Precision
`BlockSize`	Number of threads per blocks
`GridSize`	Number of blocks in the kernel
`No. threads`	Number of threads in the applications
`Achieved Occupancy`	Ratio of the average active warps per active cycle to the maximum number of warps ed on a multiprocessor.

Table 8. Features used as input in the machine learning techniques

We transformed all data to a log_2 scale before performing the learning process. We returned to the original scale after the prediction process, using a 2^{T_m} transformation [41], where T_m is the measured time. This transformation resulted in a reduction in the non-linearity effects. Figure 3 shows the difference between the trained model without (left-hand side graph) and with (right-hand side graph) logarithmic scale.

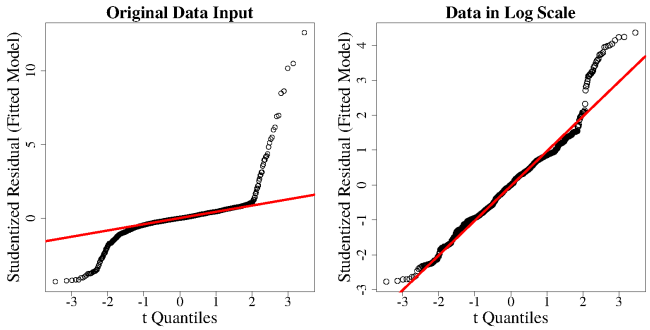

Figure 3. Quantile-Quantile Analysis of the generated models

4.4.2. Scenario 2: Prediction using feature extraction

The second scenario with ML techniques was to perform a feature extraction over the full set of features. Algorithm 1 explains the steps that we have followed to perform our features extraction process.

Algorithm 1: Algorithm of the methodological process done in this work

```
   Result: Prediction of the ML techniques
1  Input: Data sets;
2  ScaledData = scaleLog(Data sets);
3  NormalizedData = Normalize(ScaledData);
4  SCC = corr(NormalizedData, 0.75);
5  forall No. Param in [5, 10] do
6      dendroG = hclust(corr(SCC));
7      cutDendro = cutTree(dendroG, No. Param);
8      features = variance(cutDendro);
9      forall context in [GPUs, Kernels] do
10         forall ML in [LM, SVM, RF] do
11             training_set == features[context];
12             test_set != features[context];
13             Model = ML(training_set);
14             output = predict(test_set);
15         end
16     end
17 end
```

First, we transformed the data into a log_2 scale, followed by normalization (lines 2 and 3 of Algorithm 1). To reduce the number of features to be used with the ML algorithms, we performed correlation and clustering analyses over the data. We performed the analysis using the Spearman Correlation Coefficient (SCC) since it captures relations and variations among features over different scales. The scale of the correlation coefficient is between -1 and 1, where -1 represents a perfect negative correlation and 1 a maximum positive correlation. We evaluated the SCC for all features against the kernel execution times and applied an absolute threshold of 0.75, keeping only the 20 application features that had an SCC above the threshold value (line 4). We tested thresholds of 0.50, 0.75 and 0.90. A threshold of 0.5 resulted in a larger number of features without improvements in model predictions, while a threshold of 0.9 missed several important features for predictions. The kept features after the threshold are shown in Table 9. This value represented an optimum set of features with high correlations with the target feature which is execution time.

elapsed_cycles_sm	multiprocessor_activity
gld_inst_32bit	issued_control.flow_instructions
gst_inst_32bit	executed_load.store_instructions
inst_executed	device_memory_read_transactions
inst_issued1	global_store_transactions
active_cycles	global_load_transactions
issue_slots	control.flow_instructions
misc_instructions	load.store_instructions
l2_read_transactions	executed_control.flow_instructions
l2_write_transactions	issued_load.store_instructions

Table 9. Selected features after a threshold process with a SCC higher than .75 against the kernel execution times

To further reduce the features, we used a hierarchical clustering algorithm. We first created a similarity matrix overall dataset, using the Spearman correlation coefficient. We then performed a hierarchical clustering by first assigning each feature to its cluster. Following the initial assignment, we proceed iteratively, at each stage joining the two most similar clusters, continuing until there is just a single cluster. Figure 4 presents a dendrogram built from 20 features. We can then select several clusters by cutting the tree at a certain height. For instance, in Figure 4, the horizontal blue dashed line cuts the tree at a height that has 10 clusters. Finally, we performed a variance analysis to select one feature from each cluster.

Regarding the GPU features (see Table 10), 11 GPU hardware parameters were extracted they were ordered from higher to lower of the SCC against execution times. To further reduce the GPU features, we experimented machine learning models with different number of GPU parameters.

No.	**Feature Names**
1	**L2_size**
2	**num_sp_per_sm**
3	memory_clock
4	compute_capability
5	theoretical_flops
6	max_clock_rate
7	gpu_clock_rate
8	num_of_sm
9	bus
10	bandwidth
11	global_memory_size

Table 10. GPUs hardware features. We used the 6 first features in experiment of Figure 5.

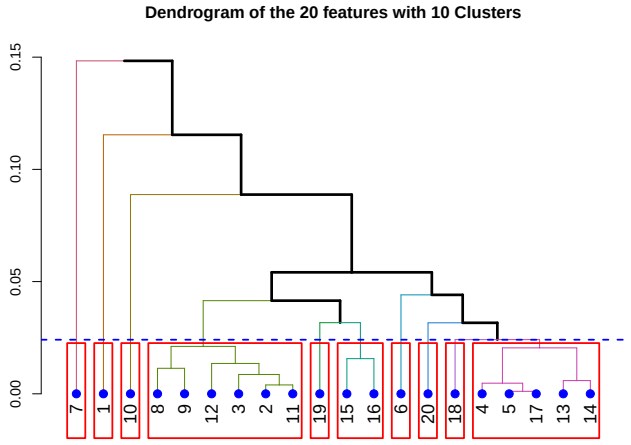

Figure 4. Dendrogram with 10 clusters to select 1 features from each one

We tested Random Forest models with different number of GPU parameters to use, following order on Table 10. We limited the models to maximum 6 GPU parameters, because kernels information is much more relevant for the experiments. Figure 5 shows the average Mean Absolute Percentage Error (MAPE) of the Random Forest models with different numbers of GPU features in two contexts (models with unknown GPU and models with unknown kernel, with 5 and 10 kernel features).

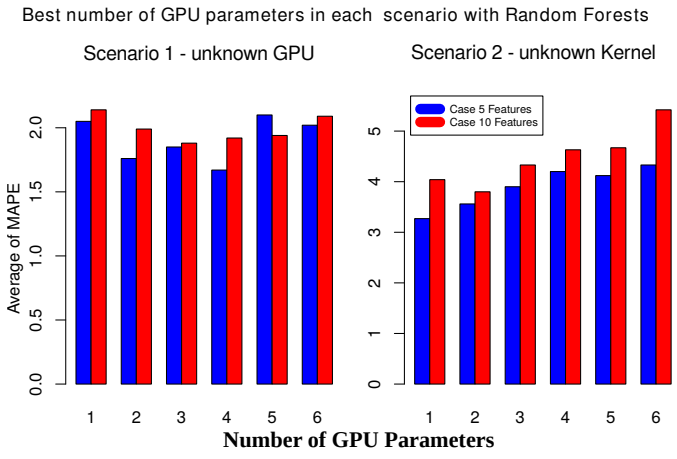

Figure 5. Best number of GPU parameters using the Random Forest algorithm

In general, we can see in both bar plots of Figure 5, left-side, and right-side, that ML models with few GPU parameters provided in general better predictions. Experiments in Figure 5 were done with CUDA kernels of the Rodinia Benchmark suite in Table 2. Consequently, we decided to use 2 GPU parameters in the experiments, they are in bold in Table 10. Thus, using Kernel information and exclusively the number of cores and the size of L2 cache as GPU paremeters was enough to obtain reasonable predictions.

We evaluated the use of machine learning regression models when the GPUs or kernels are previously unknown to the mod. In this way, we evaluate the ML algorithms using sets of 5 and 10 features, and 2 physical GPU features, considering two experimental cases:

1. **(GPUs)**: we evaluated if the ML algorithm could predict the execution time over a previously unseen GPU. We used a leave-one-out cross-validation procedure by selecting the samples from one GPU as a test set and using the samples from the other 7 GPUs as a training set.
2. **Kernels**: we evaluated if the ML algorithm could predict the execution time over a previously unseen CUDA Kernel. We used the same leave-one-out cross-validation procedure mentioned above, but now we are selecting the samples from a single kernel as a test set and using the samples from the other kernels as training sets.

Finally, for most kernels, we generated thousands of samples, because they are invoked several times in a single execution of the application. However, both kernels of Back-Propagation (BCK), *layerforward* and *adjust-weights*, generated only 57 samples (Table 2). To improve data balance and reduce model bias, we selected 100 random samples from the other Rodinia kernels.

5. Analysis of the Results

We evaluated the execution times predicted by the analytical model and the two ML scenarios. The accuracy of the model is measured using the MAPE (Equation 6).

5.1. Scenario 1: Prediction comparisons using features from the analytical model

The measured MAPE for the analytical model was 10.39% ± 6.9 for all the matrix-vector applications (Table 11), providing us with acceptable predictions, except for the Maximum Sub-Array (MSA) application. This kernel uses a lot of constant instructions in its communication which makes its high accuracy difficult to predict without a specific model.

The prediction provided by the AM was more accurate than the ML algorithms, probably due to the **well-behaved** nature of the applications, which could be modeled using the few parameters about computation and communication. Even if the analytical model resulted in a more accurate prediction, the ML algorithms would obtain errors close to our BSP-model.

For the predictions obtained using machine learning, the measured MAPE was 17.7% ± 20.31 when using Linear Regression, 15.2% ± 2.9 when using Support Vector Machine, and 11.2% ± 9.8 when using Random Forest.

Running times of the Back-Propagation kernels (BCK-K1 and BCK-K2), the Heartwall kernel and the Hotspot

Kernels	AM	LR	SVM	RF
MMGU	4.41±4.97	16.47±13.40	20.48±14.88	10.65±10.21
MMGC	7.35±3.49	15.53±11.28	13.93±7.49	10.02±7.44
MMSU	7.95±5.31	5.33±1.93	11.79±4.25	4.90±1.99
MMSC	8.31±6.44	14.83±10.22	17.04±8.21	8.41±2.87
MAU	9.71±3.09	69.51±51.77	14.76±9.62	36.76±22.23
MAC	11.00±5.27	12.65±2.77	17.67±8.52	10.83±3.85
dotP	6.89±5.56	3.30±1.42	15.37±8.31	4.67±1.35
vAdd	9.92±7.71	18.98±16.56	14.68±8.15	8.64±4.56
MSA	28.02±13.05	3.02±1.62	10.91±5.02	5.67±4.11

Table 11. MAPE (in %) of the execution time predictions over unseen GPUs with the matrix-vector applications.

Kernels	AM	LR	SVM	RF
BCK-K1	3.9±1.0	19.9±15.6	31.0±49.9	24.5±15.9
BCK-K2	4.9±3.3	86.4±158.7	140.2±283.9	97.5±188.5
GAU-K1	54.1±5.9	65.2±116.7	14.82±3.1	35.3±53.3
GAU-K2	63.7±5.1	26.6±12.4	18.5±11.7	18.6±7.5
HTW	3.7±2.2	100.8±192.8	41.6±38.0	26.3±25.1
HOT	5.5±2.1	167.2±293.6	57.6±84.0	116.9±183.9
LUD-K1	-	27.6±41.3	10.5±8.4	27.4±41.2
LUD-K2	-	57.7±79.2	42.5±54.7	48.2±70.4
LUD-K3	-	41.2±22.1	25.9±24.0	42.1±37.1
NDL-K1	-	16.1±5.3	15.5±10.1	14.0±3.8
NDL-K2	-	15.0±5.1	10.6±4.9	13.2±3.2

Table 12. MAPE (in %) of the execution time predictions over unseen GPUs with the Rodinia CUDA kernels.

kernel were well predicted by the analytical model (Table 12). Both Back-Propagation kernels contain matrix-vector operations and their behavior is regular. The CUDA kernels of the LU Decomposition and NeedlemanWunsch applications required various adaptations of our BSP-based analytical model. These kernels vary in terms of the configuration of the block numbers during their execution, and, therefore, other values of λ are necessary. For this reason, the actual version of the analytical model was not adjustable for these CUDA kernels. Although block configurations do not change in both CUDA kernels of the Gaussian application, the divergence for the number of threads vary. This impact the performance of the kernels during their execution and makes it hard to reach good accuracy in the predictions.

The prediction with the analytical model presented a MAPE of $22.6\% \pm 28.3$ for the 6 CUDA kernels of Rodinia shown in Table 12. With all the CUDA kernels of the Rodinia Benchmarking suite, LR technique obtained a MAPE of 56.69 ± 46.7, Support Vector Machine of 37.2 ± 37.4, and 42.8 ± 34.2 with Random Forest algorithm. Rodinia applications implement a different parallel programming pattern, named dwarves [34]. The observed MAPEs in the Rodinia benchmarks was high, indicating that a feature extraction process could be applied to improve the results.

Table 12 indicates that SVM and Random Forest algorithms obtained the best predictions for most of the CUDA kernels. The large number of features and few number of samples resulted in an underfitting linear model with a high variance. Rodinia kernels apply a different parallel programming pattern which impact their execution times on GPUs. Some of those patterns determine its parameters iteratively, for example, the number of blocks per grid, thread workers (divergence), communication in the texture memory and constant variables, etc. The experiments which consider a feature extraction process are presented below.

5.2. Scenario 2: Prediction using feature extraction

In this section we investigate how feature extraction can be used to improve the prediction of the machine learning models. First, we show how the 10 most important features—presented on Table 13—where determined using correlation and hierarchical clustering techniques. Then, we evaluated the performance of the machine learning predictors using: (i) only the top 5 features, and (ii) the set of all discovered features.

No.	Name Feature
1	elapsed_cycles_sm
2	gld_request
3	gst_request
4	executed_control.flow_instructions
5	device_memory_read_transactions
6	active_cycles
7	global_store_transactions
8	gst_inst_32bit
9	executed_load.store_instructions
10	misc_instructions

Table 13. Features selected using the complete feature extraction process

We evaluated the MAPE for each machine learning algorithm: linear regression (LR), support vector machine (SVM), and random forest (RF). We used 2 GPU parameters (number of cores and amount of L2 cache) and considered either 5 or 10 application features. We have used two GPU parameters `L2_size` and `num_cores_sm`, see Table 10 and Figure 5.

We first utilized the ML algorithms to predict the execution time over a previously unseen GPU. Table 14 shows the MAPE for each predicted GPU, with light and dark gray cells representing the best and worst predictions, respectively.

As we said above, we decided to investigate the use of machine learning prediction algorithms when the GPUs or kernels are previously unknown to the algorithm. All the ML algorithms had acceptable predictions for different GPUs, and their overall performance was similar in both ways (unseen GPU and unseen Kernel) with 5 and 10 features. For the three ML techniques, the measured MAPE was not superior to 3%. The higher accuracy was obtained by Linear regression with 2.66% when 10 parameters were used.

Prediction over the Tesla-P100 and GTX-680 GPUs were the worst because there were no GPUs with the same Compute Capability in the training set. Nevertheless, the predictions are accurate, with MAPEs between 1.31 and 6.24 percent.

Regarding the number of kernel features, using 5 or 10 did not cause large differences in prediction performance and, consequently, using less features is preferable depending on the number of samples. Analyzing the results presented in Table 13, we can see that most features from 6 to 10 are related to some of the feature from 1 to 5. Table 13 presents information about communication into global memory and information about level instructions.

GPUs	Model Errors Over Unseen GPUs					
	LR		RF		SVM	
	5 feat	10 feat	5 feat	10 feat	5 feat	10 feat
GTX-680	6.24	6.01	2.92	2.94	6.53	7.40
Tesla-K40	1.34	1.07	0.84	0.88	1.76	1.27
Tesla-K20	1.73	1.31	1.36	1.51	1.97	1.32
Titan	1.31	1.39	2.00	1.94	1.43	1.53
Quadro	2.63	2.25	2.91	3.02	2.81	2.20
TitanX	2.51	2.17	2.87	2.89	2.63	2.38
GTX-970	3.11	3.38	2.46	2.33	3.26	3.47
GTX-980	1.84	1.74	1.81	1.64	2.01	1.82
Tesla-P100	3.67	4.68	8.18	8.64	4.25	4.87
Total	2.70	2.66	2.81	2.86	2.91	2.96

Table 14. MAPE of the execution time predictions (in %) for previously unseen GPUs with 5 and 10 different features.

We then evaluated the 3 ML algorithms before mentioned to predict the execution time over a previously unseen CUDA Kernels. Table 15 shows the MAPE of this evaluation. All machine learning techniques reached high accuracy when unseen kernels were used to validate the built models. Linear regression techniques using 5 features as training dataset obtained a MAPE of 2.70%. The SVM models, in both cases, obtained errors around 3.5%. Random forests resulted in predictions with an approximated error of 6%.

The predictions obtained by the machine learning models were accurate when the appropriate features were used. The ML models were able to predict the execution times with MAPE values around 5% for all kernels except MSA, for which the measured MAPE was 24.43%.

For unseen GPU kernels, the predictions were less accurate for a few of them because of the different characteristics and programming patterns of the CUDA Kernels. The SVM and the LR obtained close results because we used a linear kernel for SVMs, since it provided better predictions than nonlinear kernels, such as polynomial, Gaussian (RBF), and sigmoid.

The results achieved in this scenario show that the use of machine learning predictors combined with feature extraction resulted in accurate execution time prediction, even for unseen kernels. As expected, when the GPU was not known in advance, the predictions were less accurate for some kernel because of the different characteristics of the CUDA kernel with SVM, resulting in better predictions on such conditions.

In this section, we showed that ML techniques obtained a high accuracy when a process of feature extraction is implemented. For ML experiments with a feature extraction process, we got errors around of 1.54% and 2.71%, for unknown GPUs and for unknown Kernels, respectively. The best results—in terms of quality of predictions and execution time of the predictor—for the Random Forest model were obtained using 50 trees and three predictor candidates.

Kernels	Model Errors over Unseen Kernels					
	LR		RF		SVM	
	5 Feat	10 Feat	5 Feat	10 Feat	5 Feat	10 Feat
BCK-K1	2.02	2.07	2.29	2.01	2.55	2.47
BCK-K2	2.79	2.38	2.76	2.74	3.60	2.76
GAU-K1	3.98	4.11	3.67	4.17	3.60	3.37
GAU-K2	1.56	1.74	6.87	5.95	2.03	1.98
HTW	3.72	2.50	8.19	5.81	2.98	4.63
HOT	2.72	2.62	2.63	2.45	2.70	2.32
LUD-K1	4.00	3.87	2.53	4.05	3.82	2.93
LUD-K2	1.80	1.81	2.51	2.67	2.17	1.93
LUD-K3	0.97	1.09	1.56	2.16	1.61	1.41
NDL-K1	1.27	1.33	0.50	0.45	1.62	1.01
NDL-K2	1.19	1.19	0.49	0.50	1.55	0.98
MAU	2.13	4.01	12.00	11.83	2.55	4.77
MAC	1.31	1.23	1.92	2.04	1.74	1.88
dotP	2.24	9.24	8.51	5.33	2.32	3.64
vAdd	2.99	4.80	4.09	2.69	3.23	3.67
MSA	18.55	11.07	42.14	39.06	22.20	13.61
Total	2.70	3.43	6.41	5.86	3.76	3.33

Table 15. MAPE of the execution time predictions (in %) for previously unseen CUDA kernels with 5 and 10 different features.

6. Conclusions and Future Works

In this work, we investigated the problem of predicting the execution times of GPU kernels. We evaluated the application of a BSP-based analytical model and three machine learning algorithms—namely Linear Regression, Support Vector Machine, and Random Forest—as predictors for the execution times of CUDA kernels. This comparison was made in under two different scenarios.

In order to devise an accurate prediction promptly, it is crucial to choose the minimum set of parameters (features) needed by the predictors. We studied two different approaches to this choice. In the first scenario, we evaluated the accuracy of the algorithms when using the same set of parameters required by the analytical model. Our simple analytical model predicted execution times reasonably well of CUDA kernels that scale regularly (e.g., matrix multiplications, matrix additions, dot product, among others). The usage of one adaptable parameter λ was sufficient to model the effect of data coalescing, divergence, and the usage of other memories during the execution of these kernels, which scale regularly. In our fair comparison, we validated that our simple analytical model had reasonably better predictions than machine learning techniques. This result is due to the linear growth of the execution times of matrix-vector CUDA kernels and because of the bias of the prediction process for a few and similar data samples.

In the second scenario, we used a feature extraction procedure to select which features to use. We showed that predicting the running time of GPU applications with

high accuracy is possible when using a statistical process of feature extraction. We implemented an automated approach for feature extraction based on a correlation analysis and hierarchical clustering. We evaluated sets of 5 and 10 application features as the input of the machine learning techniques. These techniques were studied with the purpose of verifying if they can be applied in cases where a predictor never has evaluated this kind of GPU or this particular kernel before. When the GPU was unseen, the Linear Regression algorithm was able to predict the execution time of the applications with a MAPE of 1.37% when using 10 application features and 1.65% when using 5 features. On average, we obtained a global error of 1.5% over all tested ML algorithms. When an unseen Kernel is predicted, we got an error of 2.71% with the SVM and a global average error of 3.17%. Regarding the GPU parameters, we showed that using only the number of cores and the amount of L2 cache was sufficient to characterize the GPUs.

We showed that the analytical model can predict the execution time of CUDA kernels that scale regularly more accurately. Once the parameters of the application are known, the prediction process is easier and less time consuming. For irregular applications —where typically devising an analytical model is harder— our machine learning-based predictor was capable of modeling the execution times with few parameters and small errors. These results showed that both approaches are complementary, and could be combined to aid the performance prediction of different applications.

For future work, we will use both approaches (the analytical model and machine learning techniques) over online scheduling heuristics to evaluate the gains that these predictions can generate in scheduling techniques. Recent GPU micro-architectures and benchmarks will be used. We will also evaluate another approach based on automatic text mining techniques to extract features for modeling CUDA kernels. This text mining techniques will use different source code information and PTX files of the applications as input. They were evaluated to improve the feature extraction process used by our machine learning algorithms.

Acknowledgments

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) - Brasil - Finance Code 001, CNPq, and São Paulo Research Foundation - FAPESP (#2012/23300-7, #2015/19399-6).

The authors thank NVIDIA for providing some of the GPUs used on the testbed.

References

[1] J. J. Dongarra, H. W. Meuer, E. Strohmaier, et al., Top500 supercomputer sites, Supercomputer 13 (1997) 89–111.

[2] E. Gaussier, D. Glesser, V. Reis, D. Trystram, Improving backfilling by using machine learning to predict running times, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '15, ACM, New York, NY, USA, 2015, pp. 64:1–64:10. doi:10.1145/2807591.2807646.

[3] Y. Zhang, J. Owens, A quantitative performance analysis model for GPU architectures, in: High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on, 2011, pp. 382–393. doi:10.1109/HPCA.2011.5749745.

[4] S. Liu, C. Eisenbeis, J.-L. Gaudiot, Speculative execution on GPU: An exploratory study, in: Parallel Processing (ICPP), 2010 39th International Conference on, 2010, pp. 453–461. doi:10.1109/ICPP.2010.53.

[5] M. Burtscher, R. Nasre, K. Pingali, A quantitative study of irregular programs on GPUs, in: 2012 IEEE International Symposium on Workload Characterization (IISWC), 2012, pp. 141–151. doi:10.1109/IISWC.2012.6402918.

[6] T. T. Dao, J. Kim, S. Seo, B. Egger, J. Lee, A performance model for GPUs with caches, Parallel and Distributed Systems, IEEE Transactions on 26 (7) (2015) 1800–1813. doi:10.1109/TPDS.2014.2333526.

[7] J. Picchi, W. Zhang, Impact of L2 cache locking on gpu performance, in: SoutheastCon 2015, Fort Lauderdale, Florida, USA, 2015, pp. 1–4. doi:10.1109/SECON.2015.7133036.

[8] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, W.-m. W. Hwu, An adaptive performance modeling tool for GPU architectures, SIGPLAN Not. 45 (5) (2010) 105–114. doi:10.1145/1837853.1693470.
URL http://doi.acm.org/10.1145/1837853.1693470

[9] L. G. Valiant, A bridging model for parallel computation, Communications of the ACM 33 (8) (1990) 103–111. doi:10.1145/79173.79181.

[10] M. Amaris, D. Cordeiro, A. Goldman, R. Y. Camargo, A simple BSP-based model to predict execution time in GPU applications, in: High Performance Computing (HiPC), 2015 IEEE 22nd International Conference on, 2015, pp. 285–294. doi:10.1109/HiPC.2015.34.

[11] D. Carastan-Santos, R. Y. de Camargo, Obtaining dynamic scheduling policies with simulation and machine learning, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '17, ACM, New York, NY, USA, 2017, pp. 32:1–32:13. doi:10.1145/3126908.3126955.
URL http://doi.acm.org/10.1145/3126908.3126955

[12] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, K. Skadron, Rodinia: A benchmark suite for heterogeneous computing, in: Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC), IISWC '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 44–54. doi:10.1109/IISWC.2009.5306797.

[13] NVIDIA Corporation, NVIDIA CUDA C programming guide, version 9.0 (2018).

[14] E. Lindholm, J. Nickolls, S. Oberman, J. Montrym, Nvidia tesla: A unified graphics and computing architecture, IEEE Micro 28 (2) (2008) 39–55. doi:10.1109/MM.2008.31.
URL http://dx.doi.org/10.1109/MM.2008.31

[15] ISO/IEC/IEEE, International standard - floating-point arithmetic, ISO/IEC 60559:2020(E) IEEE Std 754-2019 (2020) 1–86doi:10.1109/IEEESTD.2020.9091348.

[16] N. Corporation, Web site: [Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Kepler TM GK110] Visited on Nov, 2015.

[17] J. M. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, R. H. Bisseling, Bsplib: The bsp programming library, Parallel Computing 24 (14) (1998) 1947 – 1980. doi:10.1016/S0167-8191(98)00093-3.

[18] Q. Hou, K. Zhou, B. Guo, BSGP: bulk synchronous GPU programming, ACM Transaction on Graphics (2008) 12doi:10.1145/1360612.1360618.

[19] J. L. Myers, A. Well, R. F. Lorch, Research design and statis-

tical analysis, Routledge, 2010. doi:10.4324/9780203726631.

[20] Y. Zhang, Y. Hu, B. Li, L. Peng, Performance and power analysis of ATI GPU: A statistical approach, in: Networking, Architecture and Storage (NAS), 2011 6th IEEE International Conference on, 2011, pp. 149–158. doi:10.1109/NAS.2011.51.

[21] M. R. Meswani, L. Carrington, D. Unat, A. Snavely, S. Baden, S. Poole, Modeling and predicting performance of high performance computing applications on hardware accelerators, in: Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International, 2012, pp. 1828–1837. doi:10.1109/IPDPSW.2012.226.

[22] A. Karami, S. A. Mirsoleimani, F. Khunjush, A statistical performance prediction model for OpenCL kernels on NVIDIA GPUs, in: The 17th CSI International Symposium on Computer Architecture Digital Systems (CADS 2013), 2013, pp. 15–22. doi:10.1109/CADS.2013.6714232.

[23] I. Baldini, S. J. Fink, E. Altman, Predicting GPU performance from CPU runs using machine learning, in: Computer Architecture and High Performance Computing (SBAC-PAD), 2014 IEEE 26th International Symposium on, 2014, pp. 254–261. doi:10.1109/SBAC-PAD.2014.30.

[24] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, D. Chiou, GPGPU performance and power estimation using machine learning, in: 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), 2015, pp. 564–576. doi:10.1109/HPCA.2015.7056063.

[25] S. Madougou, A. Varbanescu, C. de Laat, R. van Nieuwpoort, The landscape of GPGPU performance modeling tools, Parallel Computing 56 (2016) 18–33. doi:10.1016/j.parco.2016.04.002.
URL http://www.sciencedirect.com/science/article/pii/S0167819116300114

[26] E. Konstantinidis, Y. Cotronis, A quantitative roofline model for gpu kernel performance estimation using micro-benchmarks and hardware metric profiling, Journal of Parallel and Distributed Computing 107 (2017) 37 – 56. doi:10.1016/j.jpdc.2017.04.002.
URL http://www.sciencedirect.com/science/article/pii/S0743731517301247

[27] G. Alavani, K. Varma, S. Sarkar, Predicting execution time of cuda kernel using static analysis, in: 2018 IEEE Intl Conf on Parallel Distributed Processing with Applications, Ubiquitous Computing Communications, Big Data Cloud Computing, Social Computing Networking, Sustainable Computing Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom), 2018, pp. 948–955. doi:10.1109/BDCloud.2018.00139.

[28] N. Ardalani, U. Thakker, A. Albarghouthi, K. Sankaralingam, A static analysis-based cross-architecture performance prediction using machine learning, CoRR abs/1906.07840. arXiv:1906.07840.
URL http://arxiv.org/abs/1906.07840

[29] L. Braun, S. Nikas, C. Song, V. Heuveline, H. Fröning, A simple model for portable and fast prediction of execution time and power consumption of gpu kernels, ACM Trans. Archit. Code Optim. 18 (1). doi:10.1145/3431731.
URL https://doi.org/10.1145/3431731

[30] Y. Arafa, A.-H. Badawy, A. ElWazir, A. Barai, A. Eker, G. Chennupati, N. Santhi, S. Eidenbenz, Hybrid, scalable, trace-driven performance modeling of gpgpus, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '21, Association for Computing Machinery, New York, NY, USA, 2021, pp. 1–15. doi:10.1145/3458817.3476221.
URL https://doi.org/10.1145/3458817.3476221

[31] L. Braun, H. Fröning, Cuda flux: A lightweight instruction profiler for cuda applications, in: 2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), IEEE, 2019, pp. 73–81.

[32] E. D. Lazowska, J. Zahorjan, G. S. Graham, K. C. Sevcik, Quantitative system performance: computer system analysis using queueing network models, Prentice-Hall, Inc., 1984.

[33] C. Silva, S. Song, R. Camargo, A parallel maximum subarray algorithm on GPUs, in: 5th Workshop on Applications for Multi-Core Architectures (WAMCA 2014). IEEE Int. Symp. on Computer Architecture and High Performance Computing Workshops, Paris, 2014, pp. 12–17. doi:10.1109/SBAC-PADW.2014.15.

[34] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, K. Skadron, A characterization of the rodinia benchmark suite with comparison to contemporary CMP workloads, in: Workload Characterization (IISWC), 2010 IEEE International Symposium on, IEEE, 2010, pp. 1–11. doi:10.1109/IISWC.2010.5650274.

[35] H. Nagasaka, N. Maruyama, A. Nukada, T. Endo, S. Matsuoka, Statistical power modeling of GPU kernels using performance counters, in: Green Computing Conference, 2010 International, IEEE, 2010, pp. 115–122. doi:10.1109/GREENCOMP.2010.5598315.

[36] J. Power, J. Hestness, M. Orr, M. Hill, D. Wood, gem5-gpu: A heterogeneous cpu-gpu simulator, Computer Architecture Letters 13 (1). doi:10.1109/LCA.2014.2299539.
URL http://gem5-gpu.cs.wisc.edu

[37] X. Mei, K. Zhao, C. Liu, X. Chu, Benchmarking the memory hierarchy of modern GPUs, in: C.-H. Hsu, X. Shi, V. Salapura (Eds.), Network and Parallel Computing, Vol. 8707 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2014, pp. 144–156. doi:10.1007/978-3-662-44917-213.

[38] R Core Team, R: A Language and Environment for Statistical Computing, R Foundation for Statistical Computing, Vienna, Austria (2016).
URL https://www.R-project.org/

[39] D. Meyer, E. Dimitriadou, K. Hornik, A. Weingessel, F. Leisch, e1071: Misc Functions of the Department of Statistics, Probability Theory Group (Formerly: E1071), TU Wien, r package version 1.6-7 (2015).
URL https://CRAN.R-project.org/package=e1071

[40] A. Liaw, M. Wiener, Classification and regression by randomforest, R News 2 (3) (2002) 18–22.
URL http://CRAN.R-project.org/doc/Rnews/

[41] B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. de Supinski, M. Schulz, A regression-based approach to scalability prediction, in: Proceedings of the 22Nd Annual International Conference on Supercomputing, ICS '08, ACM, New York, NY, USA, 2008, pp. 368–377. doi:10.1145/1375527.1375580.