A Comparison of GPU Execution Time Prediction using Machine Learning and Analytical Modeling

Marcos Amarís*, Raphael Y. de Camargo†, Mohamed Dyab‡, Alfredo Goldman*, Denis Trystram‡
* Institute of Mathematics and Statistics
University of São Paulo
São Paulo, Brazil
{amaris, gold}@ime.usp.br
† Center for Mathematics, Computation and Cognition
Universidade Federal do ABC
Santo André, Brazil
raphael.camargo@ufabc.edu.br
‡ Grenoble Institute of Technology
Grenoble, France
{mohamed.dyab, denis.trystram}@imag.fr

***Abstract*—Today, most high-performance computing (HPC) platforms have heterogeneous hardware resources (CPUs, GPUs, storage, etc.) A Graphics Processing Unit (GPU) is a parallel computing coprocessor specialized in accelerating vector operations. The prediction of application execution times over these devices is a great challenge and is essential for efficient job scheduling. There are different approaches to do this, such as analytical modeling and machine learning techniques. Analytic predictive models are useful, but require manual inclusion of interactions between architecture and software, and may not capture the complex interactions in GPU architectures. Machine learning techniques can learn to capture these interactions without manual intervention, but may require large training sets.**

In this paper, we compare three different machine learning approaches: linear regression, support vector machines and random forests with a BSP-based analytical model, to predict the execution time of GPU applications. As input to the machine learning algorithms, we use profiling information from 9 applications executed over 9 different GPUs. We show that machine learning approaches provide reasonable predictions for different cases. Although the predictions were inferior to the analytical model, they required no detailed knowledge of application code, hardware characteristics or explicit modeling. Consequently, whenever a database with profile information is available or can be generated, machine learning techniques can be useful for deploying automated on-line performance prediction for scheduling applications on heterogeneous architectures containing GPUs.

***Keywords*-Performance Prediction, Machine Learning, BSP model, GPU Architectures, CUDA.**

I. Introduction

Today, most computing platforms for HPC have heterogeneous hardware resources (CPUs, GPUs, storage, etc.). The most powerful supercomputers today have millions of those resources. In order to use all the computational power available, applications must be composed of multiple tasks that must use all available resources as efficiently as possible.

978-1-5090-3216-7/16/ 31.00

The Job Management System (JMS) is the middleware responsible for distributing computing power to applications. The JMS requires that users provide an upper bound of the execution times of their jobs (wall time). Usually, if the execution goes beyond this upper bound, the job is killed. This leads to very bad estimations, with an obvious bias that tends to overestimate their durations [1].

Graphics Processing Units (GPUs) are specialized processing units that were initially conceived with the purpose of accelerating vector operations, such as graphics rendering. GPUs are general purpose parallel processing units with accessible programming interfaces, including standard languages such as C, Java and Python. In particular, the Compute Unified Device Architecture (CUDA) is a parallel computing platform that facilitates the development on any GPU manufactured by NVIDIA [2]. CUDA was introduced by NVIDIA in 2006 for their GPU hardware line.

Information from profiling and traces of heterogeneous applications can be used to improve current JMSs, which require a better knowledge about the applications [3]. Predicting execution times in heterogeneous applications is a great challenge, because hardware characteristics can impact their performance in different ways. Some parallel programs can be efficiently executed on some architectures, but not on others.

Parallel computing models have been an active research topic since the development of modern computers [4]. Preliminary works on the characterization of the performance of GPU applications on heterogeneous platforms showed that simple analytical models can be used to predict performance of such applications [5], [6]

In this paper, we implemented a fair comparison between different machine learning approaches and a simple BSP-based model to predict the execution time of GPU applications [5]. The experiments were made using 9 different applications that perform vector operations. We used 9 different NVIDIA

GPUs in the experiments, 6 from Kepler and 3 from Maxwell architecture.

Our main contribution was showing that machine learning techniques provided acceptable predictions for all the applications over all the GPUs. Although the analytical model provided better predictions, it requires knowledge on the application and hardware structure. Consequently, machine learning techniques can be useful for deploying automated on-line performance prediction for scheduling applications on heterogeneous architectures with GPUs, whenever a large data set with information about similar applications is available.

The rest of this paper is organized as follows: In Section II, we present important concepts to understand this work. In Section III, we review the literature about the area. In Section IV, we describe our experiments and methodology. In Section V, we present the results from the experiments. Finally, in Section VI, we present the conclusions of our work and future work.

II. Background

A. NVIDIA GPU Microarchitecture and CUDA

NVIDIA GPU architectures have multiple asynchronous parallel Streaming Multiprocessors (SMs) which contain Scalar Processors (SPs), Special Function Units (SFUs) and load/store units. These GPU architectures vary on a large number of features, such as number of cores, registers, SFUs, load/store units, on-chip and cache memory sizes, processor clock frequency, memory bandwidth, unified memory spaces and dynamic kernel launches. Those differences are summarized in the Compute Capability (C.C.) of an GPU.

The main advantage of GPUs is that they contains thousands of simple cores, which can be used concurrently by many threads. NVIDIA GPUs have hierarchical memory configuration with a global memory, which is shared among all threads. Concurrent accesses by threads from the same warp (groups of 32 threads) to contiguous addresses can be coalesced in a single transaction. But it has a latency of about 400 or 600 cycles per access [7]. To improve memory access efficiency, they provide a small on-chip shared memory, which has a low-latency and can be accessed by all threads in a single SM. Fermi and Kepler also provide a low-latency on-chip L1 cache, with a small access latency. A L2 off-chip cache is also present, with a latency higher than L1 cache, but lower than the global memory.

The CUDA programming model and platform enables the use of NVIDIA GPUs for scientific and general purpose computations. A single *master* thread runs in the CPU, launching and managing computations on the GPU. Data for the computations has to be transferred from the main memory to the GPU's memory.

B. Bulk Synchronous Parallel Model

The main goal of parallel computing models is to provide a standard way of describing and evaluating the performance 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.

One of the most well-established models for parallel computing is the Bulk Synchronous Parallel (BSP), first introduced by Valiant in 1990 [8]. The computations in 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 were delivered and all processors are ready to start the next superstep.

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).

C. BSP-based Model for GPU Applications

In [5] the authors created a simple BSP-based model to predict performance in GPU applications. This model abstracts all the heterogeneity of GPU architectures and many optimizations that GPU application can perform in a parameter λ. We have used this model to do the comparison with the machine learning approaches. The equation 3 shows the predicted running time of a kernel T_k using this model.

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

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

$$Comm_{SM} = (ld_0 + st_0) \cdot g_{SM} \tag{4}$$

$$Comm_{GM} = (ld_1 + st_1 - L1 - L2) \cdot g_{GM} + L1 \cdot g_{L1} + L2 \cdot g_{L2} \tag{5}$$

where 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 is reserved for register spills in local memory. For this reason $L1$ is always 0 for all the experiments. Global loads are cached in $L2$ only.

The parameter λ captures the effects of thread divergence, global memory access optimizations, and shared memory bank conflicts. t is used to adjust the predicted application execution time with the measured one and is defined as the ratio between these values. It needs to be measures only once, for a single input size and a single board. The same lambda should work for all input sizes and boards of the same architecture. For a better description of this analytical model, more info can be found in [5].

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

D. Machine Learning

Machine learning refers to a set of techniques for understanding data. The theoretical subject of "learning" is related to prediction. Machine learning techniques involve building a statistical model for predicting, or estimating an output based on one or more inputs. Regression models are used when the output is a continuous value. In this paper, we used three different machine learning methods: Linear Regression, Support Vector Machines and Random Forest. There exists other machine learning techniques with sophisticated learning process. However, in this work, we wanted to use simple models to prove that they achieve reasonable predictions.

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

$$Y \approx \beta_0 + \beta_1 X_1 + + \beta_2 X_2 + \ldots + + \beta_p X_p + \epsilon \qquad (6)$$

where X_p represents the pth predictor and β_p quantifies the association between that variable and the response.

*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 the data only through dot-products. The dot product can be replaced by a kernel function which computes a dot product in some possibly high dimensional feature space Z. It maps the input vector x into the feature space Z though some nonlinear mapping.

*3) **Random Forest (RF)**:* Random Forests 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 R_m, $1 \leq m \leq M$. The prediction function of a tree is then defined as $f(x) = \sum_{m=1}^{M} c_m I(x, R_m)$, where M is the number of leaves in the tree, R_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 $x \in R_m$, 0 otherwise. The values of c_m are determined in the training process. Random forest consists of an ensemble of decision trees and uses the mode of the decisions of individual trees.

III. Related Work

Juurlink et al. were one of the firsts authors to compare performance predictions of parallel computing models [4], comparing BSP, E-BSP and BPRAM over different parallel platform. Some authors have also focused their work in performance prediction of parallel applications using machine learning [9]. All this work is about parallel applications executed over CPUs and not GPU applications.

In recent years, studies on GPU performance using different statistical and machine learning approaches have appeared. Baldini et al. showed that machine learning can predict GPU speedup from OpenMP applications [10]. They used K-nearest neighbor and SVM as classifier to know the performance of these applications over different GPUs. Wu et al. described a GPU performance and power estimation model [11], 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. Karami et al. proposed a statistical performance prediction model for OpenCL kernels on NVIDIA GPUs [12] using a regression model for prediction and principle component analysis for extracting features of higher weights, thus reducing model complexity while preserving accuracy. Zhang et al. presented a statistical approach on the performance and power consumption of an ATI GPU [13], using Random Forest due to its useful interpretation tools. Hayashi et al. constructed a prediction model that estimates the execution time of parallel applications [14] based on a binary prediction model with Support Vector Machines for runtime CPU/GPU selection. Kerr et al. developed Eiger [15], which is a framework for automated statistical approaches for modeling program behaviors on diverse GPU architectures. They used various approaches, among them principal component analysis, clustering techniques, and regression analysis. Madougou et al. presented a comparison between different GPGPU performance modeling tools [16], they compare between analytical model, statistical approaches, quantitative methods and compiler-based methods. Meswani et al. predicted the performance of HPC applications on hardware accelerators such as FPGA and GPU from applications running on CPU [17]. This was done 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. Ipek et al. trained multilayer neural networks to predict different performance aspects of parallel applications using input data from executing applications multiple times on the target platform [18].

In this work, we compare three different machine learning techniques to predict kernel execution times over NVIDIA

GPUs. We also perform a comparison with a BSP-based analytical model to verify when each approach is advantageous. Although some works have compared analytical models, statistical approaches and quantitative methods, to the best of our knowledge this is the first work that compares analytical model to machine learning techniques to predict running times of GPU applications. Moreover, it offers a comparison between different machine learning techniques.

IV. Methodology

In this section we discuss the algorithms and GPU testbed, the analytical model and the methodology used in the learning process. During our evaluation, all applications were executed using the CUDA profiling tool *nvprof*. Each experiment is presented as the average of ten executions, with a confidence interval of 95%.

A. Algorithm Testbed

The benchmark contains 4 different strategies for *matrix multiplication* [2], 2 algorithms for *matrix addition*, 1 dot product algorithm, 1 vector addition algorithm and 1 *maximum sub-array problem* algorithm [19].

*1) **Matrix Multiplication**:* We used four different memory access optimizations: global memory with non-coalesced accesses (MMGU); global memory with coalesced accesses (MMGC); shared memory with non-coalesced accesses to global memory (MMSU); and shared memory with coalesced accesses to global memory (MMSC). The run-time complexity for a sequential matrix multiplication algorithm using two matrices of size $N \times N$ is $O(N^3)$. In a CUDA application with N^2 threads, the run-time complexity is $O(N)$

*2) **Matrix Addition**:* We used two different memory access optimizations: global memory with non-coalesced accesses (MAU); and global memory with coalesced accesses (MAC); The run-time complexity for a sequential matrix addition algorithm using two matrices of size $N \times N$ is $O(N^2)$. In a CUDA application with N^2 threads, the run-time complexity is $O(1)$.

*3) **Vector Addition Algorithm (vAdd)**:* For two vectors A and B, the Vector Addition $C = A + B$ is obtained by adding the corresponding components. In a GPU algorithm, each thread performs an addition of a position of the vectors A and B and stores the result in the vector C.

*4) **Dot Product Algorithm (dotP)**:* For two vectors A and B, the dot product $C = A \cdot B$ is obtained by adding the multiplication of corresponding components of the input, the result of this operation is a scalar. In a GPU algorithm, each thread performs a multiplication of a position of the vectors A and B and stores the result shared variable. Then a reduction per blocks is performed and a vector of size equal to the number of block in the grid is transferred to the CPU memory for later processing.

*5) **Maximum Sub-Array Problem (MSA)**:* Let X be a sequence of N integer numbers $(x_1, ..., x_N)$. The Maximum Sub-Array Problem (SSM) consists of finding the contiguous sub-array within X which has the largest sum of elements. The implementation used in this paper creates a kernel with 4096 threads, divided in 32 blocks with 128 threads [19]. The N elements are divided in intervals of N/t, and each block receives a portion of the array. The blocks use the shared memory for storing segments, which are read from the global memory using coalesced accesses. Each interval is reduced to a set of 5 integer variables, which are stored in vector of size $5 \times t$ in global memory. This vector is then transferred to the CPU memory for later processing.

B. GPU Testbed

We performed our comparisons over several different NVIDIA microarchitectures. We used 9 GPUs, described in Table I. GPUs with Compute Capability 3.X belong to Kepler architecture. GPUs with Compute Capability 5.X belong to Maxwell architecture.

TABLE I
Hardware specifications of the GPUs in the testbed

Model	C.C.	Memory	Bus	Bandwidth	L2	Cores/SM	Clock
GTX-680	3.0	2 GB	256-bit	192.2 GB/s	0.5 M	1536/8	1058 Mhz
Tesla-K40	3.5	12 GB	384-bit	276.5 GB/s	1.5 MB	2880/15	745 Mhz
Tesla-K20	3.5	4 GB	320-bit	200 GB/s	1 MB	2496/13	706 MHz
Titan Black	3.5	6 GB	384-bit	336 GB/s	1.5 MB	2880/15	980 Mhz
Titan	3.5	6 GB	384-bit	288.4 GB/s	1.5 MB	2688/14	876 Mhz
Quadro K5200	3.5	8 GB	256-bit	192.2 Gb/s	1 MB	2304/12	771 Mhz
Titan X	5.2	12 GB	384-bit	336.5 GB/s	3 MB	3072/24	1076 Mhz
GTX-980	5.2	4 GB	256-bit	224.3 GB/s	2 MB	2048/16	1216 Mhz
GTX-970	5.2	4 GB	256-bit	224.3 GB/s	1.75 MB	1664/13	1279 Mhz

C. Data sets

For the analytical model, each application was executed with input sizes of power of two. For problems of one dimension, 10 samples were taken, from 2^{18} until 2^{27}. For problems of two dimensions, 6 samples were taken, all of them were squares matrices, with number of lines from 2^8 until 2^{13}.

For the machine learning analysis, we first collected the performance profiles (metrics and events) for each kernel and GPU. To be fair with the analytical model, we then choose similar communication and computation parameters to use as data input for the machine learning algorithms. We performed the evaluation using cross-validation, that is, for each target GPU, we performed the training using the other 8 GPUs, testing the model in the target GPU.

To collect data for the machine learning algorithms, we executed the two-dimensional applications using three different size for the CUDA thread blocks, 8^2, 16^2 and 32^2, and input sizes from 2^8 to 2^{13}. We took 32 samples per block size, resulting in 96 samples per GPU and a total of 864 samples. For the uni-dimensional problems we used input sizes from 2^{18} to 2^{27} and took 69 samples for each configuration, resulting in 207 samples per GPU and a total of 1863 samples. For sub-array maximum problem, 96 samples with the original configuration were taken, for a total of 864 samples.

We also evaluate a scenario were we collected more examples of a single application. We executed the matrix

multiplication with shared memory and coalesced accesses (MMSC) using 8 configurations: 16, 64, 144, 256, 400, 576, 784, and 1024 threads per blocks. This resulted in a total of approximately 256 samples for GPU, and more than 2000 samples.

For each sample, the metrics, events and trace information were collected in different phases, therefore avoiding the overhead over the measured execution time of the application. The features which we used to feed the Linear Regression, Support Vector Machines and Random Forest algorithms are described in the Table II.

TABLE II
FEATURES USED AS INPUT IN THE MACHINE LEARNING TECHNIQUES

Feature	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.

To generate the flags totalLoadGM, totalStoreGM, TotalLoadSM and TotalStoreSM, the number of requests was divided by the number of transactions per request for each operation.

We first transformed the data to a log_2 scale and, after performing the learning and predictions, we returned to the original scale using a 2^{pred} transformation [20], reducing the non-linearity effects. Figure 1 shows the difference between the trained model without (left-hand side graph) and with (right-hand side graph) logarithmic scale. The linear regression resulted in poor fitting in the tails, resulting in poor predictions. This problem was solved with the log transformation.

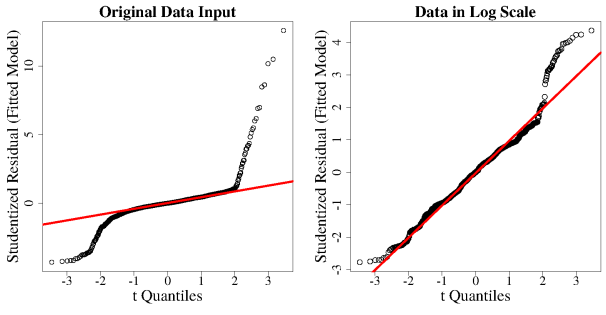

Fig. 1. Quantile-Quantile Analysis of the generated models

In this work, we applied these methods over profiling information about metrics and events of the executions of GPU applications over NVIDIA GPUs. To measure the progress of the learning algorithm we have used the normalized mean square error. With this error we have analysed the reliability of our approaches.

We used R to automate the statistical analyses, in conjunction with the e1071 and randomForest packages to use the svm and randomForest functions respectively.

V. RESULTS

The source code for all the experiments and results are available[1] under Creative Commons Public License for the sake of reproducibility. The comparison between analytical models and machine learning approaches are done taking the accuracy of the predictions, defined as the ratio between the predicted and true values of execution times, i.e, $\frac{y_{pred}}{y_{true}}$.

The rest of this section is organized as follows: In subsection V-A, the Analytical model results are presented. In subsection V-B, results for Machine Learning approach are presented. In subsection V-C, a comparison between the results of both approaches is presented.

A. Analytical Model

The number of computation ($Comp$) and communication (g_{SM}, g_{GM}, g_{L1} and g_{L2}) steps were extracted from the application source codes. These parameters are the same for all the simulations, and are presented in Table III. We did not include the values of the cache L2 for these experiments because they did not impact the execution times.

TABLE III
VALUES OF THE MODEL PARAMETERS OVER 9 DIFFERENT APPLICATIONS

Par.	Matrix Multiplication				Matrix Addition		vAdd	dotP	MSA
	MMGU	MMGC	MMSU	MMSC	MAU	MAC			
comp	$N\cdot$ FMA				$1\cdot 24$			$1\cdot 96$	$(N/t)\cdot 100$
ld_1	$2\cdot N$				2			2	N/t
st_1	1				2			1	N
ld_0	0		$2\cdot N$		0			0	N/t
st_0	0		1		0			$1+log(t)$	5

Different micro-benchmarks were used to measure the number of cycles per computation operation in GPUs [21], with FMAs, additions and multiplications taking approximately 1, 24 and 96 cycles of clock. For all simulations, we considered 5 cycles for latency in the communication for shared memory and 500 cycles for global memory [2]. Finally, when the models were complete, we executed a single instance of each application on each GPU to determine the λ values, described in the Table IV.

[1]Hosted at GitHub: https://github.com/marcosamaris/svm-gpuperf [Accessed on 19 June 2016]

TABLE IV
VALUES OF THE PARAMETER λ FOR EACH APPLICATION IN EACH GPU

	MMGU	MMGC	MMSU	MMSC	MAU	MAC	dotP	vAdd	MSA
GTX-680	4.25	19.00	18.00	68.00	0.85	11.00	14.00	11.00	0.68
Tesla-K40	4.30	20.00	19.00	65.00	2.50	9.50	9.00	10.00	0.48
Tesla-K20	4.50	21.00	18.00	52.00	2.50	9.00	9.00	10.00	0.50
TitanBlack	3.75	17.00	16.00	52.00	1.85	8.00	7.00	8.50	0.35
Titan	4.25	21.00	17.00	50.00	2.50	10.00	9.50	12.00	0.48
Quadro	5.00	22.00	22.00	68.00	1.25	10.00	12.00	11.00	0.50
TitanX	9.00	38.00	38.00	118.00	2.75	10.50	7.50	10.50	1.05
GTX-980	9.00	40.00	40.00	110.00	3.25	9.75	10.00	10.00	1.65
GTX-970	5.50	26.00	24.00	75.00	1.85	5.90	7.00	6.00	1.05

B. Machine Learning Approaches

Figure 2 shows the box plots of the accuracy of the machine learning techniques using many samples. The box plots show the median for each GPU and the upper and lower first quartiles, with whiskers representing the 95% confidence interval. Outliers are marked as individual points.

In this experiment, approximately 260 samples of the application MMSC were collected in each one of the 9 GPUs. For the training set 8 GPUs were used, and the remaining GPU was used for the test set. This was made for each GPU in the three techniques of machine learning. We can see that Linear Regression, Support Vector Machines and Random Forest have a reasonable accuracy for all the GPUs, with a mean between 0.75 and 1.5, for most cases, with some outliers.

The linear kernel in the support vector machine achieved the best performance and accuracy in the prediction. For this reason, Figures 2, 3 show similar results for Linear Regression and for Support Vector Machines. Other kernel like Polynomial, Gaussian (RBF) and Sigmoid were tested, they resulted in worse predictions.

For the random forest, we have changed two default parameters, the number of trees and the number of variables as candidates at each split. For the first parameter, the default value was 500 and for the second parameter, the default value was $p/3$, where p is the number of predictors, 13 in this case according to Table II. We set the number of trees to 50, and the number of predictors to split to 5. These values achieved better prediction, and they were determined manually after many simulations.

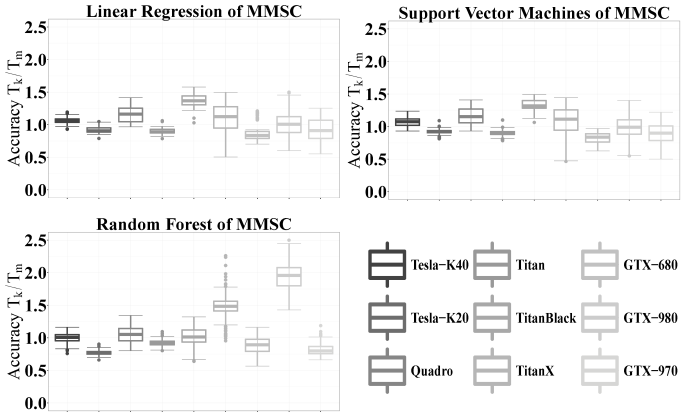

Fig. 2. Accuracy of Machine Learning Algorithms of matMul-SM-Coalesced with many samples

Table V shows the comparison between the different regression models used, in terms of mean accuracy and mean squared error. In this table, we can see that the accuracy of the predictions are between 0.75 and 1.2 for almost all the cases, only the predictions of the GTX-980 with the Random Forest showed irregular predictions, we think that was because the application MMSC showed the best performance in this GPU and the selected parameters to split the decision tree lied at the moment of the predictions.

TABLE V
STATISTICS OF THE MACHINE LEARNING WITH MORE OF 1000 SAMPLES FOR TRAINING PROCESS

GPUs	Accuracy Mean			NMSE		
	LR	SVM	RF	LR	SVM	RF
GTX-680	0.85 ± 0.09	0.82 ± 0.07	0.78 ± 0.08	0.033	0.037	**0.026**
Tesla-K40	1.21 ± 0.05	1.20 ± 0.06	0.97 ± 0.06	0.006	0.008	**0.005**
Tesla-K20	0.85 ± 0.03	0.84 ± 0.03	0.77 ± 0.02	**0.008**	**0.008**	0.051
Titan-Black	1.18 ± 0.07	1.16 ± 0.06	1.12 ± 0.12	**0.145**	0.115	0.019
Titan	0.96 ± 0.04	0.96 ± 0.04	0.98 ± 0.06	0.012	0.012	**0.008**
Quadro	1.00 ± 0.10	1.01 ± 0.10	0.98 ± 0.10	0.041	0.043	**0.017**
TitanX	1.34 ± 0.28	1.30 ± 0.27	1.45 ± 0.17	0.064	**0.059**	0.254
GTX-980	1.05 ± 0.17	1.04 ± 0.17	2.08 ± 0.50	0.029	**0.027**	0.855
GTX-970	0.73 ± 0.13	0.71 ± 0.13	0.75 ± 0.08	**0.035**	0.039	0.039

C. Machine Learning VS Analytical Model

Figure 3 shows a comparison between the accuracy of the Analytical Model (AM), Linear Regression (LR), Random Forest (RF) and SVM Regression (SVM) to predict execution times of each application on each target GPU. Each box plot represents accuracy per GPU, with each column representing a different technique and each line a different application.

We used matrix and vector algorithms with regular behavior, but the usage of thread blocks of different sizes and input sizes resulted in varying levels of occupancy in the GPUs, which made the problem challenging

We could reasonably predict the running time of 9 kernel functions over 9 different GPUs using the analytical model and machine learning techniques. For the Analytical model, the accuracy for all applications and GPUs were approximately between 0.8 and 1.2, showing a good prediction capability. For the machine learning models, the accuracy for all the applications (except MAU) and GPUs for Linear Regression and Random Forest were between 0.5 and 1.5.

When using machine learning, we considered different thread blocks configurations, which resulted in nonlinear changes in the occupancy of the GPU multiprocessors, as this affects the number of active blocks and threads, and in the effective memory bandwidth. This resulted in large variations in the execution times for each application. Also, to predict the results on each GPU, we used training data from the other 8 GPUs, which caused additional errors. These factor caused some prediction errors, but for the vast majority of cases, the predictions were reasonable.

Table VI shows the comparison between both analytical model and machine learning approaches in terms of normalized mean squared error (MSE). This table shows that

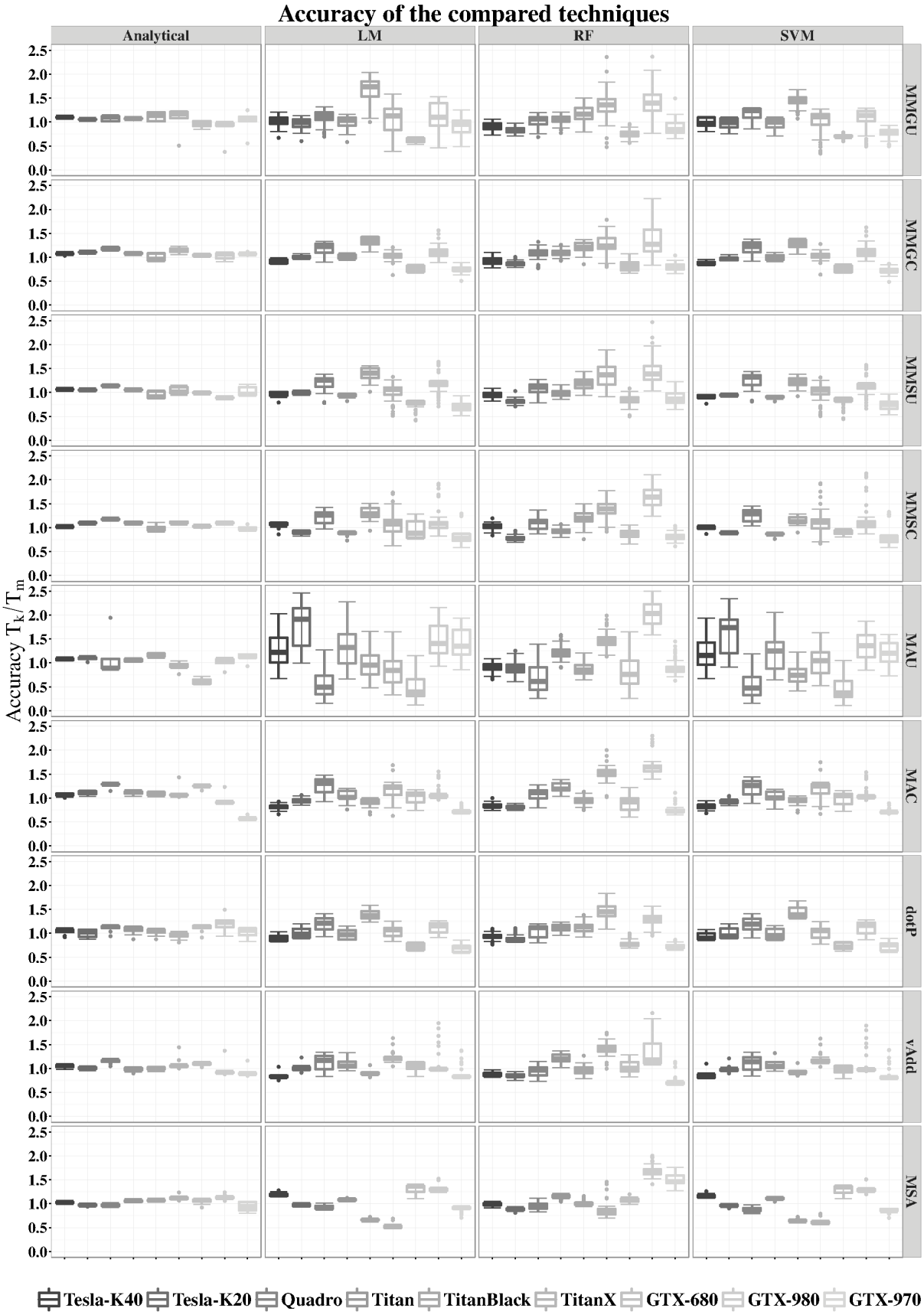

Fig. 3. Accuracy of compared techniques to predict execution times of applications on each GPU.

although the analytical model obtained the best predictions for almost all the cases, machines learning techniques also provided good predictions. Our next step is to use feature extraction to improve these predictions.

TABLE VI
NORMALIZED MSE OF THE DIFFERENT TECHNIQUES USED

Apps	NMSE			
	AM	LR	SVM	RF
MMGU	**0.0291**	0.105	0.061	0.096
MMGC	**0.0110**	0.036	0.036	0.079
MMSU	**0.007**	0.055	0.040	0.071
MMSC	**0.008**	0.046	0.044	0.097
MAC	**0.047**	0.293	0.212	0.262
MAU	0.044	0.037	**0.035**	0.114
dotP	**0.015**	0.052	0.054	0.061
VecA	**0.010**	0.021	0.018	0.062
MSA	**0.007**	0.066	0.059	0.087

VI. CONCLUSIONS AND FUTURE WORKS

We performed a fair comparison between analytical model and machine learning techniques to predict the execution times of applications running on GPUs using similar parameters to both approaches. The machine learning techniques were Linear Regression, Support Vector Machine and Random Forest.

The Analytical model provides relatively better prediction accuracy than machine learning approaches, but it requires calculations to be performed for each application. Furthermore, the value of λ has to be calculated for each application executing on each GPU.

Machine learning could predict execution time with less accuracy than the analytical model, but this approach provides more flexibility because performing specific calculations is not needed as in the analytical model. A machine learning approach is more generalizable for different applications and GPU architectures than an analytical approach.

As future work, we will consider other irregular benchmarks (Rodinia, Sparse and dense matrix linear algebra operation kernels and graph algorithms). We will also consider the scenario of multiple kernels and GPUs where global synchronization among kernels and one extra memory level, the CPU RAM, needs to be considered.

Also for the learning process in the machine learning: we will perform feature selection from a large set of features (All profiling and metrics data) to choose the most relevant ones and try them on all the regression models we tried before.

ACKNOWLEDGMENT

This project was grant-aided by São Paulo Research Foundation (FAPESP) (processes #2012/23300-7 and #2013/26644-1) by CAPES and by CNPq. Thanks to NVIDIA Corporation who donate us a some GPUs of the testbed.

REFERENCES

[1] K. Gaj, T. A. El-Ghazawi, N. A. Alexandridis, F. Vroman, N. Nguyen, J. R. Radzikowski, P. Samipagdi, and S. A. Suboh, "Performance evaluation of selected job management systems," in *Proceedings of the 16th Int'l Parallel and Distributed Processing Symposium*, ser. IPDPS '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 260–.

[2] NVIDIA, *CUDA C: Programming Guide, Version 7.*, March 2015.

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

[4] B. H. H. Juurlink and H. A. G. Wijshoff, "A quantitative comparison of parallel computation models," *ACM Transactions on Computer Systems*, vol. 16, no. 3, pp. 271–318, Aug. 1998.

[5] M. Amaris, D. Cordeiro, A. Goldman, and R. Y. Camargo, "A simple bsp-based model to predict execution time in gpu applications," in *High Performance Computing (HiPC), 2015 IEEE 22nd Int'l Conference on*, December 2015, pp. 285–294.

[6] K. Kothapalli, R. Mukherjee, M. Rehman, S. Patidar, P. J. Narayanan, and K. Srinathan, "A performance prediction model for the CUDA GPGPU platform," in *High Performance Computing (HiPC), 2009 Int'l Conference on*, 2009, pp. 463–472.

[7] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying gpu microarchitecture through microbenchmarking," in *Performance Analysis of Systems Software (ISPASS), 2010 IEEE Int'l Symposium on*, March 2010, pp. 235–246.

[8] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990.

[9] K. Singh, E. İpek, S. A. McKee, B. R. de Supinski, M. Schulz, and R. Caruana, "Predicting parallel application performance via machine learning approaches: Research articles," *Concurr. Comput. : Pract. Exper.*, vol. 19, no. 17, pp. 2219–2235, Dec. 2007.

[10] I. Baldini, S. J. Fink, and E. Altman, "Predicting gpu performance from cpu runs using machine learning," in *Computer Architecture and High Performance Computing (SBAC-PAD), 2014 IEEE 26th Int'l Symposium on*, Oct 2014, pp. 254–261.

[11] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou, "Gpgpu performance and power estimation using machine learning," in *2015 IEEE 21st Int'l Symposium on High Performance Computer Architecture (HPCA)*, Feb 2015, pp. 564–576.

[12] A. Karami, S. A. Mirsoleimani, and F. Khunjush, "A statistical performance prediction model for opencl kernels on nvidia gpus," in *The 17th CSI Int'l Symposium on Computer Architecture Digital Systems (CADS 2013)*, Oct 2013, pp. 15–22.

[13] Y. Zhang, Y. Hu, B. Li, and L. Peng, "Performance and power analysis of ati gpu: A statistical approach," in *Networking, Architecture and Storage (NAS), 2011 6th IEEE Int'l Conference on*, July 2011, pp. 149–158.

[14] A. Hayashi, K. Ishizaki, G. Koblents, and V. Sarkar, "Machine-learning-based performance heuristics for runtime cpu/gpu selection," in *Proc. of the Principles and Practices of Programming on The Java Platform*, ser. PPPJ '15. New York, NY, USA: ACM, 2015, pp. 27–36.

[15] A. Kerr, E. Anger, G. Hendry, and S. Yalamanchili, "Eiger: A framework for the automated synthesis of statistical performance models," in *High Performance Computing (HiPC), 2012 19th Int'l Conference on*, Dec 2012, pp. 1–6.

[16] S. Madougou, A. Varbanescu, C. de Laat, and R. van Nieuwpoort, "The landscape of {GPGPU} performance modeling tools," *Parallel Computing*, vol. 56, p. 18–33, 2016.

[17] M. R. Meswani, L. Carrington, D. Unat, A. Snavely, S. Baden, and 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 Int'l*, May 2012, pp. 1828–1837.

[18] E. Ipek, B. R. de Supinski, M. Schulz, and S. A. McKee, "An approach to performance prediction for parallel applications," in *Proceedings of the 11th Int'l Euro-Par Conference on Parallel Processing*, ser. Euro-Par'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 196–205.

[19] C. Silva, S. Song, and 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.

[20] B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. de Supinski, and M. Schulz, "A regression-based approach to scalability prediction," in *Proceedings of the 22Nd Annual Int'l Conference on Supercomputing*, ser. ICS '08. New York, NY, USA: ACM, 2008, pp. 368–377.

[21] X. Mei, K. Zhao, C. Liu, and X. Chu, "Benchmarking the memory hierarchy of modern gpus," in *Network and Parallel Computing*, ser. Lecture Notes in Computer Science, C.-H. Hsu, X. Shi, and V. Salapura, Eds. Springer Berlin Heidelberg, 2014, vol. 8707, pp. 144–156.