Performance Prediction for Multi-Application Concurrency on GPUs

Diksha Moolchandani*, Sudhanshu Gupta§, Anshul Kumar* and Smruti R. Sarangi*†

*Department of Computer Science and Engineering
Indian Institute of Technology Delhi
New Delhi, India
*{diksha.moolchandani, anshul, srsarangi}@cse.iitd.ac.in,

§Department of Computer Science
University of Rochester
Rochester, NY, United States
§sgupta45@cs.rochester.edu

***Abstract*—With the advent of edge computing and 5G, multiple mobile applications are being offloaded to cloud servers to meet their computational demands. Computer vision workloads dominate this space. Since the vision workloads are composed of linear algebra kernels, they perform significantly well on SIMT/SIMD architectures such as GPUs. While an application can maximize its performance on a GPU when it is the sole consumer of the GPUs resources, it fails to maintain that performance in a multi-application scenario. The primary cause of this problem is the lack of efficient virtualization techniques for GPUs and contention among the applications for the shared resources. Sadly, most of the prior work in this area is devoted to predicting single application performance. To the best of our knowledge, we propose the first machine learning-based predictor to predict the performance of an ensemble of applications on a GPU. Our predictor achieves an error of 9% across a suite of representative vision workloads for predicting the execution time. Competing algorithms that primarily work for single application scenarios have significantly inferior prediction accuracy and their error rates are more than 140%.**

I. Introduction

The rapid growth of 5G technologies [1], [2] and the increasing bandwidth and latency requirements of mobile applications [3] have led to a rapid expansion of the edge computing market. Predictions by Gartner indicate that 30% of the total mobile workloads in 2021 will be driven by strict real-time requirements [3]. Additionally, the growing number of IoT devices are expected to generate 507.5 zettabytes of data by the end of 2020 [4]. The dire need to process this data at the local collection points also promotes the growth of the edge computing market [5], which is expected to grow at a rate of 32.8% per year [6].

It is thus predicted that the majority of applications running on mobile devices will move to the edge [7], and this space will be dominated by computer vision and image processing workloads [8]–[10]. Conventional wisdom would suggest that these applications should be run on a GPU server because such applications are mainly composed of linear algebra kernels [11] that are known to work well on SIMT architectures [12]–[14]. However, in this case GPU resources will be shared among competing vision applications, and thus concurrency shall play a very key role. There is a need to distribute resources among threads temporally and spatially.

† Smruti R. Sarangi is currently the Usha Hasteer Chair Associate Professor in the department of Computer Science and Engineering, and holds a joint appointment with the department of Electrical Engineering.

Let us have a look at the effect on performance numbers of an application using both the types of multiplexing: temporal and spatial. Ausavarungnirun et al. [15] show that the performance with temporal multiplexing degrades as the number of concurrent applications increases. Another approach is spatial multiplexing (NVIDIA MPS [16]) that provides address space isolation. We performed multi-application experiments on the latest NVIDIA Turing [17] GPU and found the performance getting worse with an increasing number of concurrent applications (explained later in Section IV-C and Figure 2). Surprisingly, the performance degradation on a CPU with multi-application concurrency was much less worse (see Figure 1). To summarize, we did not observe good results with multi-application concurrency. This may be attributed to the destructive interference among the applications that does not allow each application to utilize the allocated resources fully. Thus, there is a need to predict these performance variations beforehand to maximize the performance benefit of offloading applications.

A lot of work has been done in predicting the GPU performance based on the performance of the code on a CPU, however none of them solve the performance prediction problem for multi-application concurrency. Existing predictors use different techniques: ❶ detailed analytical models [18]–[20] for performance prediction, design space exploration [21], ❷ idiom recognition [22], and ❸ ML-based performance prediction [23] for design space exploration [24], and prediction based on performance binning [25]. These are discussed in detail in Section III. The standard approach is to profile a program on a CPU, collect statistics, and predict the performance on a GPU for a single application. However, the problem of predicting the performance of a bag of tasks is *open*.

This forms the motivation of our work. To the best of our knowledge, we are the first to propose a predictor for predicting the performance of multiple applications concurrently running on a GPU. We propose a decision tree based predictor that uses standard features such as the execution time on the CPU and the instruction mix, and some novel features such as the GPU execution time for a single instance and the fairness. The fairness metric quantifies the maximum relative slowdown an application experiences when running in a multi-application setting on a multicore processor. The crux of our technique is to measure runtime parameters of applications on a multicore server, collect basic single-instance execution statistics on a GPU, and use them to predict the performance of a bag of tasks on a GPU. Moreover, our simple decision tree based model outperforms many other machine learning algorithms such as support vector regression, and allows us to explain the features of learned model. This gives us additional insights to multi-application concurrency in a GPU.

Our primary contributions are:

❶ We show that it is possible to predict the performance of a bag of tasks on a GPU using simple metrics that are primarily collected on a multicore machine. Furthermore, we show that the two most important metrics in terms of their predictive value are the execution time of a single instance on a GPU, and the fairness metric (defined in Section V).

❷ We show that a simple decision tree is more accurate than regression based approaches, and because of its explainability, it gives us unique insights into the performance of an ensemble of applications running on a GPU.

❸ We demonstrate a very low error rate (9%) as compared to state of the art work that do not take multi-application concurrency into account. They have error rates that exceed 140%.

The organization of the paper is as follows. We provide the relevant background in Section II, discuss related work in Section III, and motivate the paper in Section IV. The implementation of the scheme is described in Section V. We evaluate the proposed approach in Section VI, and finally conclude in Section VII.

II. Background

A. Multi-application Concurrency

Any GPU server will have to deal with multiple concurrent applications. To effectively support this feature in an edge/cloud computing setting some support for concurrency is required. Sadly, unlike multicore servers, GPUs were originally not designed for this purpose. Enabling application level concurrency came as an afterthought, and that too very recently. The two approaches are *time multiplexing* and *space multiplexing*. Many initial works employed time multiplexing by interleaving the applications at some pre-determined scheduling points [26]. However, this kind of resource sharing was not effective since it caused destructive interference in the GPU memory system (studied in detail by Jog et al. [27]) and also lead to severe performance degradation when the number of concurrent applications was scaled [15].

An alternative approach uses spatial multiplexing. Initially, NVIDIA's *CUDAstream* feature allowed applications from different streams to execute concurrently but at the cost of reduced memory protection as they all ran in the same address space [28], [29]. Later CUDA MPS (multi-process service) allowed spatial multiplexing where different applications are assigned to different partitions of the same address space [30] and isolation is guaranteed as long as there are no illegal memory accesses that try to access words outside the allocated memory regions.

There are several issues with concurrent multi-application execution on GPUs with any form of multiplexing: ❶ The TLBs that provide the address translation service are shared among the applications, and their limited size leads to frequent flushing of the context of other applications [15]. ❷ This leads to TLB misses and hence increased latency. ❸ The concurrent applications cause interference in the GPU memory system due to interference in the L2 caches and beyond [15], [27]. ❹ Since the error reporting resources are shared among the concurrent applications, an exception raised by any one application causes all the applications to terminate. ❺ Scheduling many threads belonging to different applications adds to the overheads.

The latest NVIDIA GPU, Turing [17], extends this basic MPS support and provides full address space isolation. Even though such modifications increase the security, the basic performance challenges highlighted in the previous paragraph still remain. In fact in the latest GPU there are provisions to limit the scalability (threads per application) to place limits on the amount of destructive interference and time spent in scheduling. This motivates our work because predicting the performance loss due to such factors is important.

B. ML-based Prediction Models

For such problems, we typically use supervised machine learning techniques. Since we need to predict a value and not a class label, we opt for an approach using regression. A regression based approach needs an error metric (also known as the loss function). Typically the mean-square error (MSE) is used. Equation 1 is the equation for the MSE loss function for N predictions. Here Y_i is the real value and $\hat{Y_i}$ is the predicted value. It can come from any kind of a regression algorithm (eg: linear regression, decision tree, or support vector regression).

$$Error = \frac{\sum_{i=1}^{N}(Y_i - \hat{Y_i})^2}{N} \tag{1}$$

1) Linear Regression: Linear regression is the simplest regression technique that models a linear relationship between the input and the output. Given input X, we compute a W and b such that $Y = WX + b$. In essence, it uses an independent variable X to predict the dependent variable Y.

2) Support Vector Regression: The primary aim in Support Vector Regression (SVR) is to find a hyperplane that can fit the maximum number of points. In addition, we define boundary lines at ϵ deviation from the hyperplane such that maximum number of points lie within the boundary and the error is restricted. These points are called the *support vectors*. The prediction for any new point is done by finding its similarity (dot product) with these points. The technique is highly dependent on finding a good hyperplane. To get a distinctive hyperplane, the points are sometimes transformed to a higher dimensional space. The dot product of the transfomed points is called a *kernel* function, which represents the similarity metric in the transformed space.

3) Decision Tree based Regression: Decision trees are normally used for classification. However, there are many upcoming applications that also use them for performing regression primarily because it is possible to interpret the models very well. In our work, explainabilty of the model is an important requirement, and thus we use this technique.

Let us start with the root node (while building the tree). Each node has a condition, a corresponding prediction, and is associated with a set of data points (N input features, and 1 output). All the data points are associated with the root node, and the prediction for the root node is a value that minimizes the mean square error for all the input points. Then we add two children to the root node (left and right). We split all the points associated with the root node into two sets (left and right resp.) such that the sum of the mean square error (MSE) is minimized. For the left and right nodes we compute their predicted values in the same way as we did for the root node. We are however not allowed to arbitrarily split the points. We can only split them based on the value of a given input feature. For example, the left child may be associated with all the points whose third feature is less than 4.6, and the rest of the points are associated with the right child (*condition*). Once this is done, we proceed recursively till we reach a certain pre-specified depth (hyper-parameter of the algorithm), or till the sum of the MSEs stops decreasing.

To traverse this tree, we keep checking the conditions, and based on their results we go to either the left child or the right child. We keep doing this till we reach a leaf node. The output of the regression algorithm is the *predicted value* of the leaf node. A manual analysis of the set of conditions can give us insights into the structure of the feature space; this can be used to derive insights.

III. Related Work

TABLE I
Comparison with related work

Year of	Model	Features	Prediction	Task
2009 [18]	Analytical	GPU parameters	Execution time	Perf. Pred.
2011 [19]	Analytical	GPU parameters	Execution time	Perf. Pred.
2011 [22]	Pattern Matching	Idioms from source code	Execution time	Perf. Pred.
2011 [20]	Analytical	CPU code skeleton	Performance(Gflops/s)	Perf. Pred.
2012 [31]	Stepwise Regression	GPU parameters	Execution time	Design space expl.
2012 [21]	Analytical	GPU parameters	Speedup	Perf. pred.
2012 [32]	Roofline	CPU parameters	Execution time	Perf. pred.
2014 [25]	ML Classification	CPU parameters	Speedup range	Classification
2015 [24]	Neural network	GPU parameters	Feature similarity	Design space expl.
2015 [23]	Stepwise regression	CPU parameters	Speedup	Perf. pred.

A. *Why not existing models?*

1) Analytical Models: Table I shows a comparative analysis of different proposals on the basis of the type of model they use, the input features, and the task they accomplish. Almost all the proposals either predict the performance on a GPU on the basis of some features (CPU performance counters, and architectural parameters of a GPU) or predict a binary value suggesting if there will be a speedup or not on the GPU. None of the works explicitly consider the case of multi-application execution on a GPU. Notably, Hong et al. [18] propose a detailed analytical model that uses the number of parallel memory requests and the memory bandwidth of an application on a GPU to predict the execution time. Similarly, Zhang et al. [19] characterize the execution phases of a program on the basis of the most time consuming component: instruction issue pipeline, shared memory access, or global memory access. Such a component acts as a performance bottleneck of that phase. Depending upon the nature of the bottleneck of an execution phase, they predict the performance. This does not take multi-application contention into account.

2) Idiom-based Models: Meswani et al. [22] proposed a performance predictor for HPC applications on generic accelerators. Since these applications have thousands of lines of code, porting their code to accelerators is difficult. Hence, they identified commonly occurring patterns of computation and memory access in the codes. These patterns are called *idioms*. Subsequently, they built performance models to predict the performance of these idioms on a diverse set of accelerators. The performance model recognizes the presence of idioms in the test data and predicts their performance on a target accelerator. Idioms are good predictors for single applications; however, when multiple applications are executing together the interaction of different idioms from different applications needs to be taken into account.

3) ML-based Models: Recent proposals use machine learning models. Ardalani et al. [23] propose to predict the performance of a code on a GPU by analyzing the performance counter data collected by running it on a CPU and the properties of the code. They develop machine learning models to predict the execution time on a GPU using this data. Similarly, Baldini et al. [25] show that fairly accurate performance classification can be done by using a minimal set of architecture-independent features. They tackle the problem by predicting if an application will achieve a speedup or slowdown on a GPU. In contrast, Jia et al. [31] and Wu et al. [24] propose to exploit the combination of architecture dependent and architecture independent features for prediction. All of these machine learning models perform the prediction on the basis of features specific to a program or a program-architecture pair. They are not suited for a bag of tasks.

Our model is different from the proposed approaches in the sense that we capture the interactions among the applications in a bag-of-tasks along with their individual architectural features. We are able to show with our experiments (explained later in Section VI) that for a more accurate performance prediction of multi-application concurrency on a GPU, the architecture independent features are not as important as the captured interaction among the applications.

IV. Motivation

A. *Overview of the Benchmarks*

Table II describes the benchmarks used in this study. These benchmarks are representative kernels used in popular applications of computer vision. They are inspired by the MEVBench [35] and SD-VBS [36] suites. We use OpenCV [37] to generate the basic kernels for the CPU code for most of these benchmarks. We then use the CUDA equivalent APIs to get the GPU-compatible codes for each of them. For *SVM*, we used the ThunderSVM library [38]

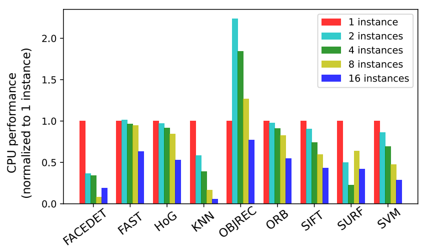


Fig. 1. CPU performance with multi-application concurrency

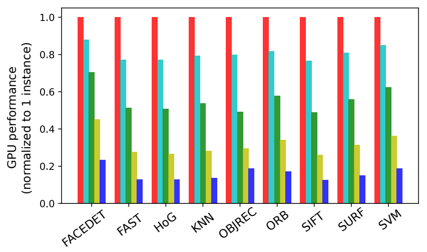


Fig. 2. GPU performance with multi-application concurrency

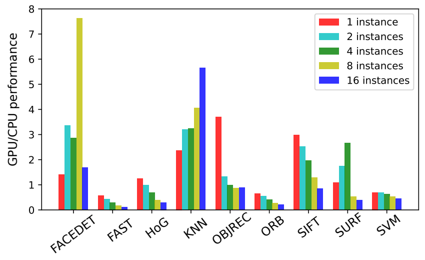


Fig. 3. GPU / CPU performance with multi-application concurrency

Benchmark	Description
Sift	Extracts those features from an image that are invariant to image orientation, illumination and scaling
Surf	Feature extraction [33] with scale invariance.
Fast	Extracts corners from an image by using a learning based technique.
Orb	Uses the FAST feature detector and the BRIEF [34] feature descriptor to extract and categorize features.
HoG	Describes a feature on the basis of the number of gradients in a certain orientation within a window of the image.
SVM	Trains a support vector, which is then used to predict the class of detected features.
KNN	Classifies the features based on the nearest neighbor algorithm.
ObjRec	Object recognition algorithm that uses both feature extraction and classification to identify the object present in a scene.
FaceDet	Face detection algorithm based on the Haar cascade classifier.

TABLE II
Benchmarks (derived from MEVBench [35])

and *KNN* was built on the lines of reference [39]. Both [38] and [39] come with equivalent CPU and GPU versions of the codes.

In some cases it was not possible to create the CUDA codes for some benchmarks without changing the algorithmic complexity significantly. This would have rendered the CPU and GPU implementations significantly different from each other and hence would have caused an algorithmic mismatch as defined by Ardalani et al. [23]. Hence, these workloads were omitted.

B. Experimental Setup

We ran all the experiments on a server whose configuration is shown in Table III. Our 2-socket server has 48 logical cores (with hyperthreading enabled). For the experiments on the GPU, we use the latest NVIDIA Turing GPU with MPS enabled. For each application we choose that configuration (number of threads) that has the least execution time. Our experiments indicated that the OpenCV-based benchmarks performed the best when parallelized with OpenMP [40] as opposed to Pthreads [41] or Intel TBB [42]. For *KNN* and ThunderSVM, the number of parallel threads was specified in the code. For all the GPU runs, we used NVIDIA CUDA MPS [16] (explained in Section II) to enable the concurrent execution of multiple applications. Note that **performance is defined as the reciprocal of the execution time** in our case.

TABLE III
Details of the baseline system

Parameter	Type/Value
CPU	2 x Intel Xeon Gold 5118 (Skylake)
# of cores	24 physical
Frequency	2.3 GHz
Main memory	128 GB
GPU	NVIDIA Tesla T4 (Turing)
CUDA cores	2560
Tensor cores	320

C. Performance Variation of the Workloads with Multi-application Concurrency

Multi-application concurrency can be of two types: homogeneous or heterogeneous. Homogeneous refers to multiple instances of the same application running concurrently. On the contrary, heterogeneous refers to different applications being launched concurrently. In this section, we look at homogeneous multi-application concurrency. We chose a homogeneous bag of applications because characterization and identifying the source of deviation from expected behavior is much easier. In contrast, a heterogeneous bag has different concurrent applications and to assess the relative contributions of different applications is much harder analytically.

Figures 1 and 2 show the performance variation of different benchmarks with the variation in the number of instances of the benchmarks on the multicore CPU and the GPU respectively. Comparing the trends in the two figures, we can make the following observations: ❶ The performance variation with multi-application concurrency on a CPU is significantly different than that on a GPU. For example, for *ObjRec*, the performance on a GPU decreases as the number of concurrent applications increases while it shows a non-deterministic behavior with the variation in the number of parallel instances on a CPU. Similar is the case for other benchmarks like *Surf*, *FaceDet*, and *HoG* too. ❷ From Figure 3, we observe that GPU performance for a single instance is better than the CPU performance for most of the benchmarks (with some exceptions: *Fast*, *Orb*, *SVM*), however, it does not scale well with the number of instances of the application. Despite superior compute capabilities, the performance degrades on a GPU with an increasing number of instances. This is because the applications contend for the shared resources/caches as a consequence of increased destructive interference. On the contrary, there are well developed policies to handle contention [43]–[45] in multicore CPUs and hence the scalability is much better with multi-application concurrency as compared to GPUs. ❸ Figure 2 also shows that the trend for the performance across the benchmarks on a GPU remains roughly the same for multi-instance runs.

Based on these observations, we can gather the following insights:

Insights:
❶ The performance of multiple instances on a GPU cannot be simply correlated with the corresponding performance of the workload on a CPU.
❷ Contention in the shared resources plays an important role when multiple instances of an application are launched in parallel.
❸ The performance of such a multi-instance workload on a GPU can be directly correlated with the performance of a single instance in the case of a GPU.

V. Implementation

Our aim is to predict the performance of a multi-application bag of tasks on a GPU. As already explained in Section II-B, the task that fits our problem is a regression task and the algorithm that goes well with our data is a decision tree (accuracy and explainability). The loss function of the regression task is defined by the MSE as explained in Section III.

A data point is an input-output pair that is used for training our machine learning model. The broad overview of our implementation is as follows: ❶ We first identify the input features to be used. ❷ We then conduct experiments to collect all the training data. Each experiment/program execution corresponds to a single data point. ❸ These data points are then fed to the learning model after being divided into two sets: training and test data. ❹ The model formed after training is used to predict the output for the test data points.

A. Defining the Features

Defining the representative features for the data points that could be learned and used later for accurate predictions is the most important step of building any machine learning model. The chosen features should be characteristic of the benchmarks' behavior and the features should be correlated with the predicted quantity.

Particularly for GPUs there is no accepted methodology for defining features that determine the final performance [23], [25]. We start with the list of features that have been proven to work well for GPU performance prediction and we define a few additional features based on our observed insights.

Based on **Insight ❷**, we define *fairness* as one of the features because we are dealing with a multi-application scenario. It quantifies the slowdown in an environment with resource contention [46]. The equation for the fairness, $fairness_{\mathcal{T}}$, of a bag-of-task, $\mathcal{T}$, is given by the following equation, where i and j are tasks in the bag.

$$fairness_{\mathcal{T}} = min\left(\frac{IPC_i^{shared}}{IPC_i^{alone}} \Big/ \frac{IPC_j^{shared}}{IPC_j^{alone}}\right) \quad (2)$$

For each task we find its slowdown when it is running in a shared environment as compared to when it is running in isolation. The fairness is the minimum slowdown by the maximum slowdown across all pairs of tasks. The **intuition** behind using this metric is that since fairness captures the slowdown ratio in a multi-application scenario, it can directly capture the effect of contention as mentioned in **Insight ❷**. We show in Section VI that fairness plays an important role in reducing the prediction error. It also contributes significantly to the way the decision tree is constructed.

Based on **Insight ❸**, we define *GPU execution time* for running a single benchmark as one of the features. The single-instance performance on a GPU can be very well utilized to predict the application's behavior in a multi-application case. For most programs this is very easy to measure. Table IV shows the list of features that we consider. The instruction mix and CPU execution time have been used in prior work to make different predictions. Our novelty is in combining them, and in introducing two additional features: single-instance GPU execution time and fairness (shaded rows in the table).

TABLE IV
List of features

Num	Feature	Description
1	CPU_time	Execution time of the benchmark on a CPU
2	GPU_time	Execution time of the benchmark on a GPU
3	SSE	% of SSE instructions
4	ALU	% of arithmetic instructions
5	MEM	% of load/store instructions
6	FP	% of floating point instructions
7	Stack	% of stack push/pop instructions
8	String	% of string operations
9	Shift	% of multiply/shift operations
10	Control	% of control/branch instructions
11	Fairness	Fairness of concurrent multi-application execution

1) Handling Variable Sized Feature Vectors: Recall that we consider both homogeneous and heterogeneous bags of applications in our multi-application concurrency experiments. When a homogeneous bag of applications is run concurrently, they share the same set of features. On the contrary, for a heterogeneous bag, the features for each application are different. Hence, the feature vector for a homogeneous bag can be completely represented by one set of features while for the heterogeneous case, it can be as large as the number of concurrent applications. Thus, the length of the feature vector varies across the data points.

When there are multiple applications, we replicate the feature vector. Nevertheless, using a variable sized feature vector makes learning very difficult. We thus limit the number of concurrent applications to two for our experiments. Making the learning process to scale in terms of the number of applications is an *open problem*. This is because we need to generate a lot of additional training data that considers all possible interactions between applications and ensure the uniform length of the feature vector across the data points. The other *open problem* is to consider all kinds of variations in the number of threads. Currently, we take the best performing configuration (in terms of the number of threads) for each application.

B. Creating the Data Points

The predictor model is trained on a set of training data points, which is an input-output pair. The input is the feature vector of the data point and the output is its multi-application performance on a GPU. Thus, to create a data point, we need to have equivalent CPU and GPU implementations of the benchmarks (also explained in Section IV-A). Since the number of benchmarks is limited, we are limited by the number of data points and this could easily lead to over-fitting. Thus, we increased the number of data points by using 5 different inputs for each benchmark. This is a standard approach to increase the number of data points [23], [25]. The standard input to each of our benchmarks is a batch of 20 images, which is increased to 40, 80, 160, and 320 images to generate five different data points out of each benchmark. The number of images in a batch (batch size) is determined empirically. We chose the batch size that is representative of the behavior of the benchmark. Any change in the input of the benchmark changes its execution time and hence the feature vector and can be considered a new data point [23].

As explained in Section IV-C, a heterogeneous multi-application concurrency scenario is the one that allows multiple benchmarks to run concurrently. Another method of increasing the data points is by permuting the combinations of benchmarks to generate different multi-application scenarios. In total, we examine *91 runs* including both homogeneous and heterogeneous workloads with different combinations of batch sizes. In general, such applications of machine learning in computer architecture have small size of the training data. This is because it is limited by the number of unique benchmarks and their input combinations. For example, Baldini et al. [25] use 48 unique data points for training. Similarly, Ardalani et al. [23] generate 122 data points for training by combining benchmarks from six different benchmark suites and applying the standard technique of increasing the number of data points as described in the previous paragraph.

C. Collection of Features

We collect the feature values for the feature vector by instrumenting the benchmark codes using the PIN 3.7 [47] and MICA 1.0 [48] tools to capture the percentages of different kinds of instructions. The IPC data for different benchmarks in the bag (to be used in the calculation of fairness) is calculated using the Linux Perf tool. All the time values (GPU_time and CPU_time) in the feature vector are normalized with respect to the range ($maximum - minimum$) of the CPU_time feature in the training data.

D. Predictor Model

We used the open-source machine learning library Scikit-learn [49] to implement the regression models for our prediction task. As explained in Section II-B, linear regression is used when the features of the data points are completely independent. This is not the case in our features, so we chose the decision tree and SVR regression algorithms. Upon further analysis, it was found that the sparsity of our data points does not allow SVR to learn a unique hyperplane. This also appeared in the prediction error: it was 10X more in SVR as compared to that in the decision tree.

Next, we define the methodology used for collecting the training data, test data and to perform cross validation.

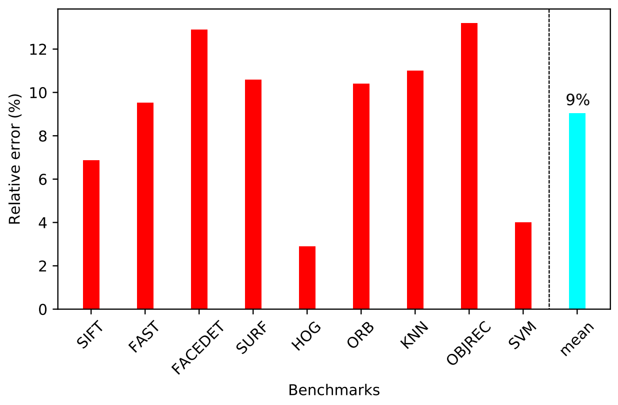


Fig. 4. Relative error for leave-one-out cross validation (LOOCV)

1) Cross-validation: Before we test our trained model on unseen test data, there is a need to validate the model. It is a standard way to test how well a model generalizes. The standard procedure for cross-validation is to split the training dataset into two disjoint sets: training and validation sets. We use the training set for building the model and the validation set for evaluating the accuracy of the model. To get a more accurate idea of the generalizability of the model, we perform multiple rounds of cross-validation with different splits of the dataset each time. We use leave-one-out cross-validation (LOOCV) [25] to test the robustness of our model. The basic idea is to leave one data point from the entire dataset for testing and using other data points for training. In our case, we have multiple data points corresponding to a benchmark. Thus, to perform LOOCV for a particular benchmark (experiments in Figure 4), we leave all the data points corresponding to that benchmark for testing and use the rest for training. This ensures that the validation data is unseen. Figure 4 shows the cross-validation error for each benchmark. We observe a 9% relative error (mean) for predicting the execution time of running multiple applications on the GPU. Here each x-axis label denotes the benchmark that was left out in the LOOCV.

2) Test Data: We partition the entire dataset into two disjoint sets, one for training (80%) and other for testing (20%). The test data is not seen by the model at any time during the training. It is used only during the evaluation to evaluate the accuracy and the relative error in our predictions.

VI. Results and Analysis

There are two different kinds of bags used in our evaluation: homogeneous and heterogeneous. In a homogeneous bag the concurrently running applications are the same: code, input size, and threads. A heterogeneous bag is formed by all possible combinations of different benchmarks. We evaluate the bags for a concurrency of two. We compare the schemes on the basis of the relative error, which is defined similar to [23]:

$$Error = \mid (true_{val} - predicted_{val})/true_{val} \mid \times 100$$

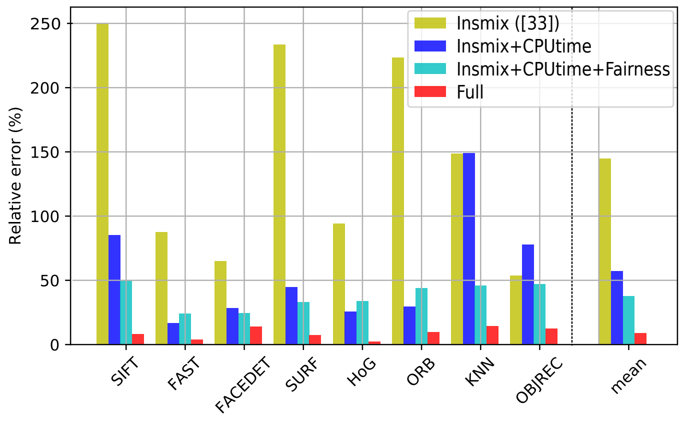


Fig. 5. Comparison with related work (used their features for the multi-application concurrency scenario)

A. Comparison with Related Work

Figure 5 shows the comparison of four different schemes on the basis of their relative errors in the prediction. The schemes primarily differ in the kind of features used in the machine learning model. The first scheme uses *insmix* (instruction mix); these set of features were also used by Baldini et al. [25]. Though they achieved significant accuracy for predicting the range of GPU speedups of a single application, the scheme does not capture the interactions among the applications in a multi-application concurrency scenario. Hence, it has large errors (144.6%) as shown in Figure 5. Another comparison that we make is with a combination of *insmix* and *CPU_time* as features in the feature vector. These features provide a better prediction than just the instruction mix (57.05% error). This is because the *CPU_time* of a benchmark is positively correlated (correlation=0.95) with the multi-application performance on a GPU.

The third comparison is with the *CPU_time*, *fairness* and *insmix* as the features in the feature vector. This scheme performs significantly better than the first two approaches (89.55% and 19.32% better respectively) because both CPU_time and fairness are able to capture the performance degradation in a bag-of-task. Additionally, fairness is also intuitively a good predictor of the performance of multi-application concurrency because it is able to capture the contention at the shared resource in the presence of multiple applications (see Equation 2). Lastly, we compare with the scheme that has all of the above metrics along with the *GPU_time*. Since the *GPU_time* of an application is the most correlated with its multi-application execution time (Insight ❸), the prediction error reduced further (9.05%). We do not eliminate the instruction mix from the feature vector in this experiment because it improves the prediction accuracy when used in combination with *CPU_time* and *fairness* (as observed in the previous experiment). To summarize, we improve the error rate by 135.55% over the state-of-the-art machine learning model [25] for GPU performance prediction (reference work by Baldini et al. [25] which is the left-most bar).

B. Sensitivity of the Prediction Error w.r.t the Features

1) Effect of CPU_time: Figure 6 shows the effect of the CPU_time feature on the prediction error. The x-axis labels represent the features used in the prediction. It can be observed that for any feature combination (x-axis labels), the prediction error decreases with the introduction of CPU_time in the combination. Thus, a general insight is that having the CPU execution time (for running a single instance of the benchmark) in the feature vector leads to better splitting in the decision tree used to predict the performance of multi-application concurrency on a GPU. Additionally, it can be observed that the relative error of prediction increases when fairness is combined with the instruction mix in the absence of CPU_time (89.54% to 98.17%), while the introduction of CPU_time to this combination reduces the error to only 37.73%.

Another observation is that arithmetic+sse+fairness performs worse than mem+fairness implying that the percentage of memory instructions in a benchmark is a better indicator of GPU performance as compared to the compute (arithmetic+sse) instructions. However, this relationship reverses with the introduction of the CPU_time. This is because CPU_time in combination with the compute (arithmetic+sse) instructions and fairness ratio can accurately predict the performance degradation on a GPU.

2) Effect of the GPU_time: Figure 7 shows the effect of the GPU_time feature on the prediction error. It can be observed that for any feature combinations (x-axis labels), the prediction error decreases with the introduction of the GPU_time in the feature combination except the arithmetic+sse+fairness combination of features. This reduction in error is even more pronounced than that achieved by the introduction of CPU_time as shown in Figure 6. This was also evident in the insights in Section IV, where we concluded that CPU_time alone cannot be a good feature for prediction while GPU_time is positively correlated with the multi-instance GPU performance. Furthermore, we observe that the prediction error increases when fairness is combined with the instruction mix and increases further when the instruction mix consists of only the compute instructions. This makes the case for considering a diverse set of instruction types in the instruction mix.

3) Effect of the Instruction Mix: Figure 8 shows the effect of the instruction mix on the relative error of prediction. It can be observed that for some feature combinations the instruction mix is useful while for others it does not have a sizeable positive impact. In general, the instruction mix with CPU_time makes the prediction better while it has no positive impact when it is exclusively combined with the GPU_time.

4) Effect of Fairness: Figure 9 shows the importance of fairness on the quality of prediction. It can be observed that for any feature combination (x-axis labels), the prediction error decreases with the introduction of fairness in the combination. This implies that fairness has a positive effect all the time.

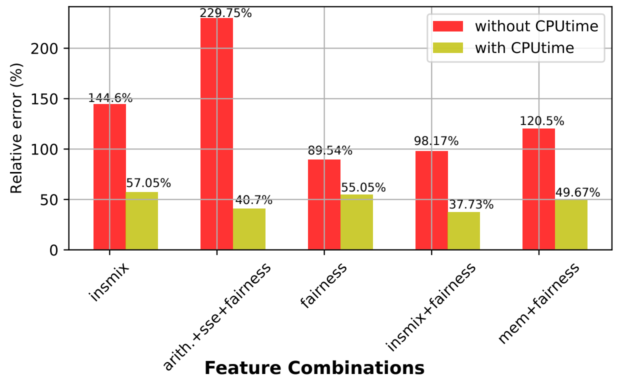


Fig. 6. Effect of CPU time on the prediction error

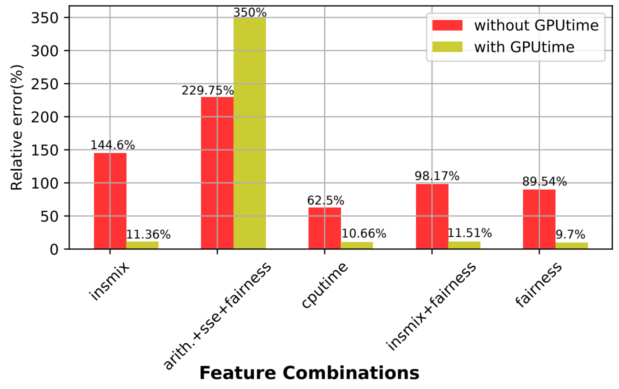


Fig. 7. Effect of GPU time on the prediction error

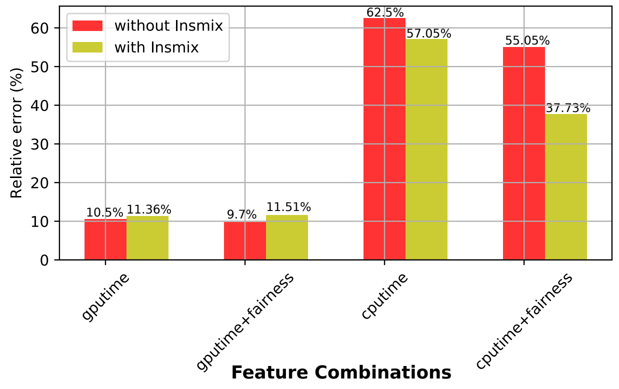


Fig. 8. Effect of the instruction mix on the prediction error

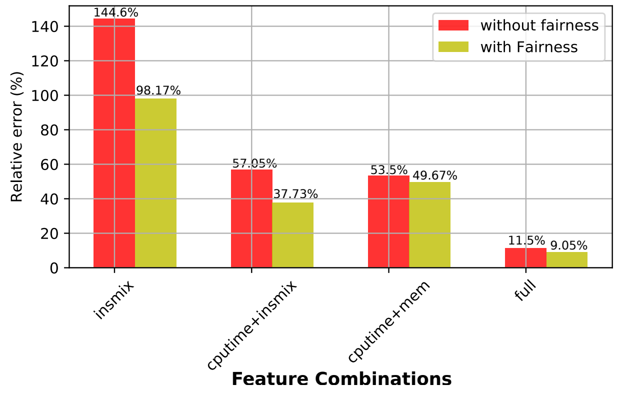


Fig. 9. Effect of Fairness on the prediction error

Additionally, it can be observed that CPU_time in combination with the instruction mix performs nearly the same as compared to CPU_time with memory instructions. However, the introduction of fairness makes CPU_time and instruction mix a better predictor because it captures a more holistic picture of the slowdown in a multi-application scenario. This also makes the case for using a diverse mix of instructions in the instruction mix.

The summary of our findings are:

❶ CPU_time has a positive effect on the prediction error when combined with the instruction mix.
❷ Fairness improves the prediction given by a combination of CPU_time and instruction mix.
❸ Fairness, GPU_time and CPU_time reduce the prediction error. However, the extent of reduction is different depending on the already existing features in the feature vector.
❹ The positive effect of GPU_time on the prediction error is more pronounced as compared to the CPU_time.

These results corroborate our insights in Section IV.

C. Analysis of the Decision Paths

Based on the observations in Section VI-B, we can conclude that the features proposed by us reduce the error. In this section, we perform an even deeper analysis of what features are actually used in the splitting of the decision nodes. This would help us understand if some features are redundant and can be eliminated. We analyzed the decision path for all the test points. A *decision path* is the path that contains all the decision nodes, the features used in making the decisions, the threshold values used for the comparison in decision making, the branches and the leaf nodes. Every test data point follows its own path in the decision tree formed by the training data points. We answer a few questions based on our analysis.

1) Is fairness more important than the instruction mix?: The answer to this question is **yes**. Figure 10 shows the percentage of test points that utilize a particular feature in its decision path. We can observe that GPU_time occurs in 100% of the test data points to decide one or more splits.

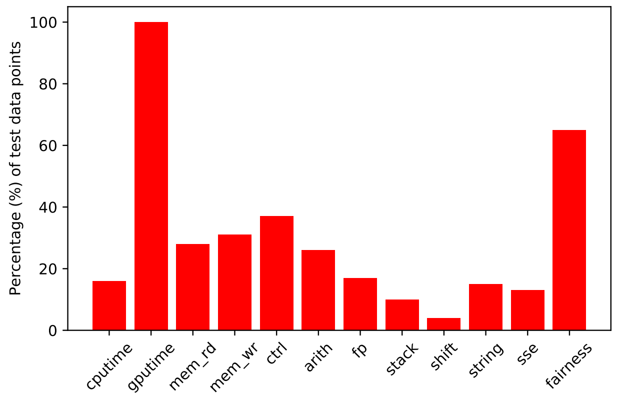


Fig. 10. Percentage of the test points containing a feature in their decision path

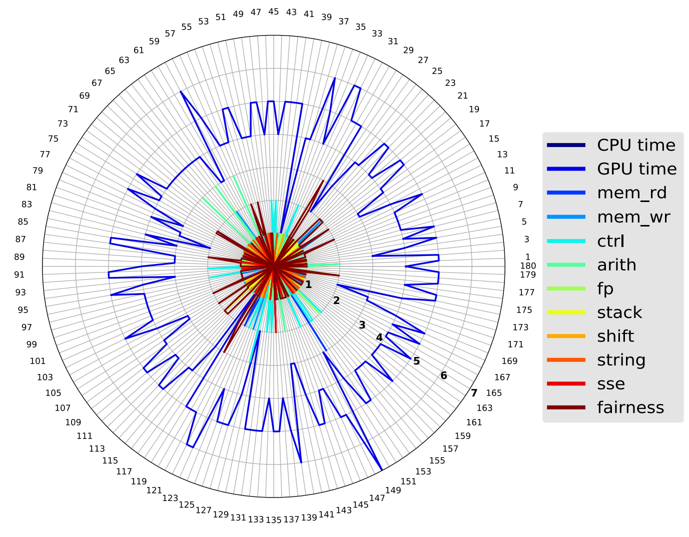
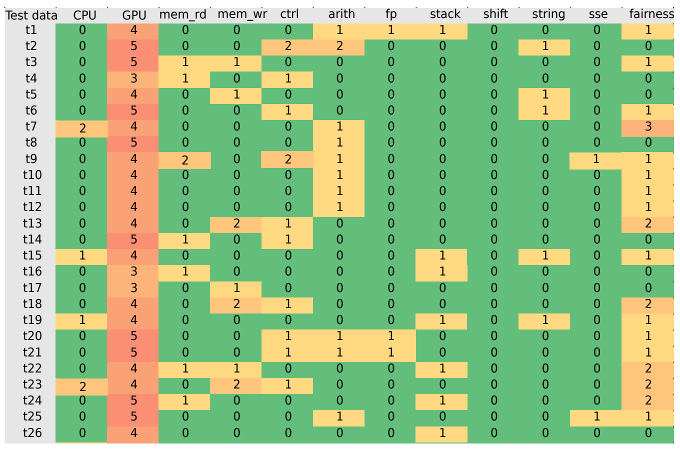


Fig. 11. Radar plot of the frequency of each feature for different test points



Test data	CPU	GPU	mem_rd	mem_wr	ctrl	arith	fp	stack	shift	string	sse	fairness
t1	0	4	0	0	0	1	1	1	0	0	0	1
t2	0	5	0	0	2	2	0	0	0	1	0	0
t3	0	5	1	1	0	0	0	0	0	0	0	1
t4	0	3	1	0	1	0	0	0	0	0	0	0
t5	0	4	0	1	0	0	0	0	0	1	0	0
t6	0	5	0	0	1	0	0	0	0	1	0	1
t7	2	4	0	0	0	1	0	0	0	0	0	3
t8	0	5	0	0	0	1	0	0	0	0	0	0
t9	0	4	2	0	2	1	0	0	0	0	1	1
t10	0	4	0	0	0	1	0	0	0	0	0	1
t11	0	4	0	0	0	1	0	0	0	0	0	1
t12	0	4	0	0	0	1	0	0	0	0	0	1
t13	0	4	0	2	1	0	0	0	0	0	0	2
t14	0	5	1	0	1	0	0	0	0	0	0	0
t15	1	4	0	0	0	0	0	1	0	1	0	1
t16	0	3	1	0	0	0	0	1	0	0	0	0
t17	0	3	0	1	0	0	0	0	0	0	0	0
t18	0	4	0	2	1	0	0	0	0	0	0	2
t19	1	4	0	0	0	0	0	1	0	1	0	1
t20	0	5	0	0	1	1	1	0	0	0	0	1
t21	0	5	0	0	1	1	1	0	0	0	0	1
t22	0	4	1	1	0	0	0	1	0	0	0	2
t23	2	4	0	2	1	0	0	0	0	0	0	2
t24	0	5	1	0	0	0	0	1	0	0	0	2
t25	0	5	0	0	0	1	0	0	0	0	1	1
t26	0	4	0	0	0	0	0	1	0	0	0	0

Fig. 12. Snapshot of the heatmap of different features in the test data (the values indicate the number of times that a feature is used in any decision node on the decision path of the test point)

Similarly, fairness contributes to the decision path of 65% of the test points. Since most of the test points rely on GPU_time, and fairness, other metrics have a lesser importance. This experiment confirms the presence of a feature in the decision path of the test set, but it does not convey how many times the same feature has been used at different decision nodes for predicting the execution time of a test point. Note that a feature may be used multiple times in one decision path at different decision nodes present at different depths of the decision tree. This is answered in the next section.

2) What is the importance of each feature?: Figure 11 shows the number of times a feature is used in the decision path of a test point. Each concentric circle is a distinct number that represents the number of times a feature is used in the decision making of a test data point. The radar plot shows this result for all the test data points used in our experiments (test data points generated by LOOCV for every benchmark). The basic observation is that the decision paths give the maximum importance to the GPU_time; it is used 5-6 times in the decision path of each test point. The second highest importance is given to the fairness metric; it is used 1-3 times in the decision making of 65% of the total test points. The rest of the features are still important and appear at least 1-2 times in the decision path.

In Figure 11 the basic pattern of the decision making process is visible; however, it is not possible to infer the details. Thus, we separately show the data for a set of sample points in Figure 12. We show the number of times each feature is used in the decision path of a test point. We can observe that among the features depicting the instruction mix only control, arithmetic, stack, load, and store instructions make a sizeable contribution to the decision making. Surprisingly, CPU_time does not contribute to more than two decision making nodes in the test points. However, CPU_time made a significant contribution in reducing the prediction error in our sensitivity analysis. Thus, even though it appears in 1-2 decision nodes, these decision nodes play a critical role in deciding the correct splitting of the data points.

VII. Conclusion

In this paper we showed that GPUs are not *necessarily faster* than CPUs when we consider multiple instances of a workload, or in a multi-application scenario. This is because of the contention in shared structures and inadequate support for virtualization. Hence, there is a need to accurately predict the performance of such ensemble workloads on a GPU. Traditionally used metrics such as the execution time on the CPU, and the instruction mix do not yield a good prediction accuracy. We noticed that we need to introduce two additional metrics: execution time of a single instance on a GPU, and the fairness. They increased the accuracy significantly, and the mean error was 9%. Competing algorithms that were originally designed to predict the performance of a single instance had much larger error rates (in excess of 140%). Finally, our choice of a simple decision tree based algorithm as opposed to sophisticated non-linear regression algorithms is justified by the fact that our error rate is much lower, and we were able to explain the relative importance of the features and their contributions by analyzing the learned decision tree. The problem of predicting the performance when the number of threads per workload is variable, and the number of applications is more than 3 or 4 is still open.

VIII. Acknowledgements

The authors would like to thank the anonymous reviewers for their valuable comments. This work has been partially funded by the Semiconductor Research Corporation (SRC) via grant number 2017-CT-2735.

References

[1] Y. Yu, "Mobile edge computing towards 5g: Vision, recent progress, and open challenges," *China Communications*, vol. 13, no. 2, pp. 89–99, 2016.

[2] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young, "Mobile edge computing a key technology towards 5g," *ETSI white paper*, vol. 11, no. 11, pp. 1–16, 2015.

[3] R. W. News. Four edge computing market predictions. https://www.rcrwireless.com/20180829/cloud-computing/four-edge-computing-market-predictions.

[4] TechRepublic. Edge computing. https://www.techrepublic.com/article/edge-computing-the-smart-persons-guide/.

[5] N. Hassan, S. Gillani, E. Ahmed, I. Yaqoob, and M. Imran, "The role of edge computing in internet of things," *IEEE Communications Magazine*, vol. 56, no. 11, pp. 110–115, 2018.

[6] A. M. Research. Global edge computing market. https://www.alliedmarketresearch.com/edge-computing-market.

[7] M. Aazam, S. Zeadally, and K. A. Harras, "Offloading in fog computing for iot: Review, enabling technologies, and research opportunities," *Future Generation Computer Systems*, vol. 87, pp. 278–289, 2018.

[8] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. ACM, 2012, pp. 13–16.

[9] RTInsights. Time to get real: 3 tips for success with edge iot and streaming analytics. https://www.rtinsights.com/time-to-get-real-3-tips-for-success-with-edge-iot-and-streaming-analytics/.

[10] ——. Why edge computing can help iot reach full potential. https://www.rtinsights.com/why-edge-computing-can-help-iot-reach-full-potential/.

[11] J. Gallier and J. Quaintance, "Linear algebra for computer vision, robotics, and machine learning," 2019.

[12] J. Fung and S. Mann, "Openvidia: Parallel gpu computer vision," in *Proceedings of the 13th Annual ACM International Conference on Multimedia*. ACM, 2005, pp. 849–852.

[13] T. Komuro, S. Kagami, and M. Ishikawa, "A dynamically reconfigurable simd processor for a vision chip," *IEEE journal of solid-state circuits*, vol. 39, no. 1, pp. 265–268, 2004.

[14] R. D. Jackson, P. C. Coffield, and J. N. Wilson, "A new simd computer vision architecture with image algebra programming environment," in *1997 IEEE Aerospace Conference*, vol. 1. IEEE, 1997, pp. 169–185.

[15] R. Ausavarungnirun, V. Miller, J. Landgraf, S. Ghose, J. Gandhi, A. Jog, C. J. Rossbach, and O. Mutlu, "Mask: Redesigning the gpu memory hierarchy to support multi-application concurrency," in *ACM SIGPLAN Notices*, vol. 53, no. 2. ACM, 2018, pp. 503–518.

[16] N. Corp. Multi-process service. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.

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

[18] S. Hong and H. Kim, "An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness," in *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3. ACM, 2009, pp. 152–163.

[19] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for gpu architectures," in *2011 IEEE 17th international symposium on high performance computer architecture*. IEEE, 2011, pp. 382–393.

[20] J. Meng, V. A. Morozov, K. Kumaran, V. Vishwanath, and T. D. Uram, "Grophecy: Gpu performance projection from cpu code skeletons," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, p. 14.

[21] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc, "A performance analysis framework for identifying potential benefits in gpgpu applications," in *ACM SIGPLAN Notices*, vol. 47, no. 8. ACM, 2012, pp. 11–22.

[22] M. R. Meswani, L. Carrington, D. Unat, A. Snavely, S. Baden, and S. Poole, "Modeling and predicting application performance on hardware accelerators," in *2011 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2011, pp. 73–73.

[23] N. Ardalani, C. Lestourgeon, K. Sankaralingam, and X. Zhu, "Cross-architecture performance prediction (xapp) using cpu code to predict gpu performance," in *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 2015, pp. 725–737.

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

[25] I. Baldini, S. J. Fink, and E. Altman, "Predicting gpu performance from cpu runs using machine learning," in *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*. IEEE, 2014, pp. 254–261.

[26] Y. Suzuki, "Making gpus first-class citizen computing resources in multi-tenant cloud environments," 2018.

[27] A. Jog, O. Kayiran, T. Kesten, A. Pattnaik, E. Bolotin, N. Chatterjee, S. W. Keckler, M. T. Kandemir, and C. R. Das, "Anatomy of gpu memory system for multi-application execution," in *Proceedings of the 2015 International Symposium on Memory Systems*. ACM, 2015, pp. 223–234.

[28] N. Corp. Nvidia tesla p100. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.

[29] ——. Cuda toolkit documentation. https://docs.nvidia.com/cuda/cuda-runtime-api/stream-sync-behavior.html.

[30] ——. Nvidia tesla v100 gpu architecture. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[31] W. Jia, K. A. Shaw, and M. Martonosi, "Stargazer: Automated regression-based gpu design space exploration," in *2012 IEEE International Symposium on Performance Analysis of Systems & Software*. IEEE, 2012, pp. 2–13.

[32] C. Nugteren and H. Corporaal, "The boat hull model: adapting the roofline model to enable performance prediction for parallel computing," in *ACM Sigplan Notices*, vol. 47, no. 8. ACM, 2012, pp. 291–292.

[33] H. Bay, T. Tuytelaars, and L. Van Gool, "Surf: Speeded up robust features," in *European conference on computer vision*. Springer, 2006, pp. 404–417.

[34] M. Calonder, V. Lepetit, C. Strecha, and P. Fua, "Brief: Binary robust independent elementary features," in *European conference on computer vision*. Springer, 2010, pp. 778–792.

[35] J. Clemons, H. Zhu, S. Savarese, and T. Austin, "Mevbench: A mobile computer vision benchmarking suite," in *Workload Characterization (IISWC), 2011 IEEE International Symposium on*. IEEE, 2011, pp. 91–102.

[36] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor, "Sd-vbs: The san diego vision benchmark suite," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2009, pp. 55–64.

[37] G. Bradski and A. Kaehler, *Learning OpenCV: Computer vision with the OpenCV library*. " O'Reilly Media, Inc.", 2008.

[38] Z. Wen, J. Shi, Q. Li, B. He, and J. Chen, "Thundersvm: A fast svm library on gpus and cpus," *The Journal of Machine Learning Research*, vol. 19, no. 1, pp. 797–801, 2018.

[39] V. Garcia, E. Debreuve, F. Nielsen, and M. Barlaud, "K-nearest neighbor search: Fast gpu-based implementations and application to high-dimensional feature matching," in *2010 IEEE International Conference on Image Processing*. IEEE, 2010, pp. 3757–3760.

[40] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[41] B. Nichols, D. Buttlar, J. Farrell, and J. Farrell, *Pthreads programming: A POSIX standard for better multiprocessing*. " O'Reilly Media, Inc.", 1996.

[42] T. Willhalm and N. Popovici, "Putting intel® threading building blocks to work," in *Proceedings of the 1st international workshop on Multicore software engineering*. ACM, 2008, pp. 3–4.

[43] A. Fedorova, S. Blagodurov, and S. Zhuravlev, "Managing contention for shared resources on multicore processors," *Queue*, vol. 8, no. 1, p. 20, 2010.

[44] G. Fernandez, J. Abella, E. Quiñones, C. Rochange, T. Vardanega, and F. J. Cazorla, "Contention in multicore hardware shared resources: Understanding of the state of the art," in *14th International Workshop on Worst-Case Execution Time Analysis*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2014.

[45] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *ACM Sigplan Notices*, vol. 45, no. 3. ACM, 2010, pp. 129–142.

[46] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt, "Utility-based acceleration of multithreaded applications on asymmetric cmps," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, pp. 154–165, 2013.

[47] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm sigplan notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.

[48] K. Hoste and L. Eeckhout, "Comparing benchmarks using key microarchitecture-independent characteristics," in *2006 IEEE International Symposium on Workload Characterization*. IEEE, 2006, pp. 83–92.

[49] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.