Performance and Power Prediction for Concurrent Execution on GPUs

DIKSHA MOOLCHANDANI, ANSHUL KUMAR, and SMRUTI R. SARANGI, Indian Institute of Technology Delhi, India

The [1] unprecedented growth of edge computing and 5G has led to an increased offloading of mobile applications to cloud servers or edge cloudlets. The most prominent workloads comprise computer vision applications. Conventional wisdom suggests that computer vision workloads perform significantly well on SIMD/SIMT architectures such as GPUs owing to the dominance of linear algebra kernels in their composition. In this work, we debunk this popular belief by performing a lot of experiments with the concurrent execution of these workloads, which is the most popular pattern in which these workloads are executed on cloud servers. We show that the performance of these applications on GPUs does not scale well with an increase in the number of concurrent applications primarily because of contention at the shared resources and lack of efficient virtualization techniques for GPUs. Hence, there is a need to accurately predict the performance and power of such ensemble workloads on a GPU. Sadly, most of the prior work in the area of performance/power prediction is for only a single application. To the best of our knowledge, we propose the first machine learning-based predictor to predict the performance and power of an ensemble of applications on a GPU. In this paper, we show that by using the execution statistics of standalone workloads and the fairness of execution when these workloads are executed with three representative microbenchmarks, we can get a reasonably accurate prediction. This is the first such work in the direction of performance and power prediction for concurrent applications that does not rely on the features extracted from concurrent executions or GPU profiling data. Our predictors achieve an accuracy of 91% and 96% in estimating the performance and power of executing two applications concurrently, respectively. We also demonstrate a method to extend our models to 4-5 concurrently running applications on modern GPUs.

CCS Concepts: • **Computing methodologies → Modeling methodologies**; *Machine learning*; Cross-validation.

Additional Key Words and Phrases: power and performance prediction, machine learning, concurrent execution, GPUs

ACM Reference Format:
Diksha Moolchandani, Anshul Kumar, and Smruti R. Sarangi. 2022. Performance and Power Prediction for Concurrent Execution on GPUs. 1, 1 (March 2022), 25 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In any cloud computing setup, it is expected that users will submit a large number of jobs with varied characteristics, and it is subsequently necessary to efficiently schedule them keeping in mind power,

[1]Extension of Conference Paper: Moolchandani et al. [25]. In the conference paper [25], we had proposed a performance predictor model for predicting the performance of a concurrent execution on GPUs. This paper extends the conference paper by developing a novel power predictor for predicting the power consumption of the concurrent execution on GPUs without relying on the GPU performance counters, unlike state-of-the-art works. Our model uses the performance counters obtained from the execution of standalone applications (in the bag of tasks) on CPUs and the GPU power of standalone applications. There is no GPU profiler needed. We also make a major paradigmatic change, where we propose a new fairness metric (different from the conference version). We do not need to run combinations of real-world workloads in the training or inference phases any more. This paper provides detailed insights into different feature combinations and ML models. We also extend our power and performance models to provide accurate predictions for 4-5 concurrently running applications (our previously published conference paper provided predictions for only two concurrent applications).

Authors' address: Diksha Moolchandani, diksha.moolchandani@cse.iitd.ac.in; Anshul Kumar, anshul@cse.iitd.ac.in; Smruti R. Sarangi, srsarangi@cse.iitd.ac.in, Indian Institute of Technology Delhi, Hauz Khas, New Delhi, India.

2022. XXXX-XXXX/2022/3-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

performance, and real-time execution metrics [17]. Modern clouds based on either data centers or edge computing devices have a large variety of hardware and thus the scheduling problem becomes even more complex. A vital algorithm in this workflow is a prediction technique that answers a simple question: If a set of applications is executed concurrently on a given node (multicore CPU or GPU), what is the expected power and performance? If we can get an accurate a priori estimate, then the scheduling problem becomes much easier and the final results are significantly better [43]. Hence, performance estimation of a bag of tasks is a vital component of any job management technique in cloud computing setups including edge clouds. Moreover, also note that even if the set of applications is finite, we might have a very large number of unique combinations of applications; it is impossible to characterize the runtime figures for each combination. We can at best characterize the behavior of applications individually (standalone execution).

Performance estimation per se is a fairly well established area and its usage in job management and scheduling is well accepted [43]. Moreover, there are very good techniques for predicting power and performance of applications on multicore CPUs. However, the area of power/performance estimation in GPUs where we have spatial multiplexing (NVIDIA's multiprocess service) is quite sparse – we shall call such applications as *concurrent applications*. Given that GPUs are first-class computing devices today, this problem is extremely relevant. Before proceeding further with describing our contributions, let us further motivate the need for GPUs and why they are absolutely necessary today.

We are increasingly seeing a new set of workloads take centerstage in modern computer architecture research. These workloads are in areas such as computer vision, image processing, AI/ML, and augmented/virtual reality [32]. As compared to traditional workloads that are similar to SPEC and Parsec benchmark suites, these workloads are quite different. They are far more compute and memory intensive, and also tremendously benefit from GPUs and other custom accelerators. We should also appreciate the importance of such workloads in the context of ultra-high bandwidth communication enabled by 5G. It is widely expected [1] that many compute kernels in mobile applications will move to edge computing devices. Recent predictions by Gartner [29] suggest that 40% of the total edge computing market will comprise such applications by 2023, and the market itself is estimated to grow at a CAGR of 38.4% per year [30] from 2021 to 2028. As a result, it is important to focus on such futuristic workloads today. Finally, if we consider self-driving applications [36], then also edge computing will play a major role in localization, driver assistance, emergency services, and inter-vehicle coordination. Much of this uses GPUs.

Given this short motivation, our aim in this paper **is to predict the power and performance of concurrent applications running on a GPU**. In any cloud scheduling framework, this important power/performance-estimating component is missing as of today. Let us now look at the shortcomings of prior work and prove why a trivial extension does not suffice.

Most of the related work focuses on predicting the performance and power of standalone applications on GPUs based on the CPU and GPU performance counters, however none of them aim to *predict the performance and power of the execution of concurrent applications* on a GPU. These works employ the following prediction techniques: ❶ analytical modeling of GPU power based on architectural details of the GPU [15, 42], and ❷ ML-based models [7, 37, 39]. We discuss them in detail in Section 3. For performance prediction on the GPU, many of these works emphasize on exclusively or mostly using CPU statistics to avoid the cumbersome process of writing and profiling CUDA codes. For power prediction on the GPU, almost all the related papers utilize GPU-specific performance counters, which are far harder to extract and use than their CPU counterparts. Hence, we do not use any GPU-specific performance counters.

Statement of novelty: To the best of our knowledge, we are the first to propose a predictor for predicting the performance and power of concurrent applications running on a GPU. We completely avoid the cumbersome process of performance counter-based profiling on GPUs; the only GPU parameters that we use are the standalone performance and power of a workload. Finally, note that we only use the profiling results of standalone executions (workloads executed individually) on CPUs. We also propose to use a fairness metric that quantifies the slowdown of execution in a concurrent setup. This metric cannot be collected for standalone applications and collecting them for each unique combination of workloads is prohibitive. For example, consider a use-case of 60 workloads that can be executed with a concurrency of 3, we will need to execute 37,820 unique combinations to collect the fairness values, which is prohibitive. Hence, we introduce 3 microbenchmarks to collect three fairness values for each workload. Thus, for a 60-workload case, the number of executions is $180 = 60 \times 3$ to collect the fairness values for each workload. We reduce an polynomial time problem to a linear time problem. The scheme is explained in detail in Section 5.1.4. This makes our algorithm very easy to use and brings it in line (in terms of performance and usability) with comparable works for predicting CPU power and performance.

1.1 Contributions and Organization of the Paper

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 5).

❷ We show that it is possible to predict the power of a bag-of-tasks on a GPU using the metrics collected on a multicore machine and the power of the individual applications in the bag obtained on the GPU. We remove the reliance on the GPU performance counters unlike the related work in this domain.

❸ We show that there are broadly three types of memory access behaviors in these workloads for any level of concurrency. Hence, instead of calculating fairness for every combination of workloads, we calculate three fairness values for each workload when running concurrently with three representative microbenchmarks. Hence, we reduce an polynomial time problem to a linear time problem without sacrificing accuracy.

❹ We provide detailed insights into different feature and ML model combinations for the power prediction. In addition, we provide insights into the importance of different features for the performance predictor model. We obtain an accuracy of 91% and 96% for performance and power prediction for a concurrency of 2, respectively.

❺ We show that our model generalizes well if the number of concurrent applications are increased.

The organization of the paper is as follows. We provide the relevant background in Section 2, discuss related work in Section 3, and motivate the paper in Section 4. The implementation of the scheme is described in Section 5. We evaluate the proposed approach in Sections 6, 7 and 8, and finally conclude in Section 10.

2 BACKGROUND

2.1 Multi-application Concurrent Execution

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 popular approaches are *time multiplexing* and *space multiplexing*. Many initial works employed time multiplexing by interleaving the applications at some pre-determined scheduling points [34]. However, this kind of resource sharing was not effective since it caused destructive interference in the GPU memory system and also led to severe performance degradation when the number of concurrent applications was scaled [5].

An alternative approach uses spatial multiplexing. Nvidia's CUDA MPS (multi-process service) allows spatial multiplexing where different applications are assigned to different partitions of the same address space and isolation is guaranteed as long as there are no illegal memory accesses. The latest NVIDIA GPUs, Turing [11] and Ampere [10], extend this basic MPS support and provide full address space isolation. Though such modifications increase the security, some basic performance challenges still remain.

There are several issues in concurrently executing applications 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 [5]. 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 [5]. 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.

2.2 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 a regression-based approach. A regression-based approach needs an error metric (also known as the loss function). Typically the mean-squared error (MSE), or the mean absolute percentage error (MAPE) is used. Equation 1 is the equation for the MAPE 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 regression algorithm (eg: linear regression, decision tree, support vector, random forest, multi-layer perceptron, or gradient boosting regression).

$$Error = \frac{\sum_{i=1}^{N}(Y_i - \hat{Y}_i)/Y_i}{N} \times 100 \tag{1}$$

2.2.1 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.

2.2.2 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, boundary lines are defined at a deviation of ϵ from the hyperplane such that the 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 more distinctive hyperplane, the points are sometimes transformed to a higher dimensional space.

2.2.3 Decision Tree Regressor. In order to build a decision tree, we start with a root node. 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 initially 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. In order for the tree to grow, the points associated with the root node are split into two child nodes (left and right) based on the value of a given input feature. The split ensures that the MSE is minimized. The predicted values for the child nodes are calculated in the same way as described for the root node. This is done till a certain pre-specified depth (hyper-parameter of the algorithm) is reached, or till the sum of the MSEs stops decreasing.

To traverse this tree, we keep checking the conditions corresponding to the features, and decide to go to either the left child or the right child. We keep doing this till we reach a leaf node and the *predicted value* of the leaf node is the output of the regression.

2.2.4 Random Forest Regressor. A random forest regressor is based on an ensemble learning technique, which combines the predicted value from multiple decision trees to make a final prediction. The combination of different predictions is equal to the mean of the predictions from all the decision trees. Here, multiple decision trees are trained using the same training dataset. A random forest is usually more accurate than a decision tree (also visible in our analyses in Section 7), however the interpretability of the features is reduced.

2.2.5 Gradient Boosting Regressor. A Gradient boosting regressor is also an ensemble learning technique that combines the prediction from multiple decision trees, however the decision trees are not constructed independently unlike the case of random forest. Initially, one decision tree is trained with the dataset and the ground truth. Subsequently, another decision tree is introduced to *boost* the prediction made by the first decision tree. This is done by training the second tree with the dataset and the prediction error from the first decision tree. This process is repeated till a pre-specified number of learners is reached or we exhaust the maximum number of iterations. The inferencing process is also sequential. The output from the first decision tree goes to the second and so on. All the outputs are added to get the final prediction.

2.2.6 Multi-layer Perceptron. A multi-layer perceptron (MLP) is an artificial neural network that consists of multiple fully-connected layers. It can have a minimum of 3 layers: input layer, hidden layer, and output layer. The input layer takes in the input or the features and computes a weighted sum of all the inputs to generate the neurons of the hidden layer. These are then passed through a non-linear activation function to generate the final output. There can be multiple hidden layers to increase the depth of the MLP.

3 RELATED WORK

Table 1 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. There have primarily been two kinds of approaches to predict the power and performance of an application on GPUs. The first approach relies on analytical models while the second approach uses machine learning-based models. Almost all the proposals either predict the performance on a GPU on the basis of some features (CPU performance counters, and/or architectural parameters of a GPU) or predict a binary value suggesting if there will be a speedup or not on the GPU. For power prediction, almost all the works rely on GPU performance counters, and its architectural parameters. The GPU performance counters are obtained by profiling the execution of the application on the GPUs using various tools [7] such as CUPTI, nvprof, LLVM, GPUOcelot, GPGPUSim, and MacSim.

3.1 Performance Models

Several analytical models have been proposed for performance prediction. Hong et al. [15] proposed 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 of the application on the GPU. Zhang et al. [42] proposed to associate the execution phases of an application with the most time-consuming architectural components corresponding to that phase. They identify the most time-consuming execution phase at runtime and predict the performance of the application on the basis of this phase.

Recent proposals use machine learning models. Ardalani et al. [3] proposed to predict the performance of a GPU program using a regression model. The features are based on the performance counter data collected by running the application on a CPU and the properties of the code. Similarly, Baldini et al. [6] showed that fairly accurate performance classification (speedup or slowdown w.r.t. the CPU) of applications on GPUs can be done by using a minimal set of architecture-independent features. In contrast, Wu et al. [39] proposed to exploit the combination of architecture-dependent and architecture-independent features for prediction. Ahmad et al. [2] proposed a neural network-based performance predictor to choose between multicores and GPUs for a particular workload-input combination. They also predict the level of multithreading and other architectural settings to maximize performance. Though they rely primarily on the workload characteristics to make a prediction, they do not solve the problem for the case of concurrent execution. All these machine learning models perform the prediction on the basis of features specific to a single program or a program-architecture pair. They are not suited for a bag of tasks running concurrently. Xu et al. [41] proposed a machine learning-based interference predictor to predict the interference between concurrent VMs running on Nvidia vGPU, however they rely heavily on GPU profiling data and architectural statistics collected using nvidia-smi, nvprof, and ESXi. In contrast, we prove a very important point in our paper: *from single-application executions, it is possible to predict the execution statistics of multi-application executions without relying on GPU profiling data.*

Table 1. **Comparison with related work**

Year of	Model	Features	Task
2009 [15]	Analytical	GPU perf. counters	Perf. pred.
2009 [24]	SVR	GPU perf. counters	Power pred.
2010 [28]	Linear Regression	GPU perf. counters	Power pred.
2010 [16]	Analytical model	runtime statistics of SMs	Perf. + power pred.
2011 [23]	Analytical model	static GPU parameters	Perf. + energy pred.
2011 [42]	Analytical	GPU perf. counters	Perf. pred.
2013 [33]	ANN	GPU perf. counters	Power pred.
2013 [20]	McPAT-based	GPU architectural components	Power pred.
2014 [21]	McPAT-based	GPU architectural components	Power pred.
2014 [6]	ML Classification	CPU perf. counters	Speedup pred.
2015 [39]	Neural network	GPU perf. counters	Perf. + power pred.
2015 [3]	Stepwise regression	CPU perf. counters	Speedup pred.
2016 [40]	McPAT+GPUWattch-based	GPU architectural components	Power Pred.
2019 [2]	Neural Network	Hardware-independent features	Perf. pred.
2019 [41]	ML-based	GPU perf. counters	Perf. Pred.
2020 [7]	Random Forest	Hardware-independent features	Perf. + power pred.

3.2 Power Models

Luo et al. [23] developed a performance and energy prediction model based on static parameters (Example: # instructions, #SMs, max. compute bandwidth, etc.) obtained from an analysis of CUDA applications using Ocelot. Such models do not capture runtime effects, and thus are not considered to be very accurate. Hence, Hong et al. [16] proposed an empirical power and performance predictor model that relied on runtime statistics of the SMs, main memory, and warps. Leng et al. [20] proposed to estimate the energy per access of different components using a modified McPAT model. A similar approach was taken by Lim et al. [21] to model the GPU power using the modified McPAT model and the GPU configuration, which was later tuned using empirical data. Similarly, Xu et al. [40] used McPAT and GPUWattch for evaluating the power consumption of a concurrent execution. Here, they relied on collecting profiling stats of the applications corresponding to different micro-architectural components of the GPU. Arunkumar et al. [4] proposed an energy model for multi-module GPUs where the energy of execution was calculated using the energy per instruction and energy per transaction for compute and memory instructions, respectively. These values were obtained empirically. In contrast, we prove a very important point in our paper: *from single-application executions, it is possible to predict the execution statistics of multi-application executions without relying on GPU profiling data.*

The other category of models were built using machine learning-based methods. Ma et al. [24] built a SVR-based regression model using the runtime statistics of the workloads. Nagasaka et al. [28] also built a statistical regression model but they take into account the memory access behavior of the workloads on the GPUs along with other performance counters. Song et al. [33] proposed a power model using artificial neural networks and used hardware counters as features. This led to an increase in the accuracy at the cost of increased complexity. All these models use platform-specific features and are hence not easily portable. In this direction, Braun et al. [7] proposed a random forest-based regression model that relies on hardware-independent features and is easily portable to different GPU architectures.

As can be clearly established, none of the aforementioned works developed a model for predicting the power and performance of multiple concurrent applications without using GPU profiling data. We, on the other hand, develop a model that captures the interactions between the concurrent applications in a bag-of-tasks along with their individual statistics. In addition to capturing the interactions, we were able to develop the power model without using any GPU performance counters, instead we rely on the GPU power and CPU performance counters of the individual applications.

3.3 Challenges: Power and Performance Prediction for Concurrent Execution

There are multiple challenges associated with predicting the power and performance of a concurrent execution. As explained in Section 2, we leverage Nvidia MPS to execute multiple applications concurrently. In MPS, the resources are shared by multiple applications and it distributes the resources proportionally to different applications [14]. However, it has been proved that the relationship between the performance of concurrent applications and the amount of allotted resources is non-linear and sometimes non-convex [40]. This is because the interference between applications occurs at a very fine granularity and at multiple levels such as the L1/L2 caches, interconnects, and main memory [19]. Unless there is a well-known relationship that quantifies the effect of contention/interference on the execution of the application kernels, it is challenging to predict the performance of the concurrent execution [41].

In order to model an accurate predictor that predicts the performance degradation in the case of concurrent execution, we need to model the system-level behavior of GPUs under application interference [41]. We basically need the microarchitecture-level details of GPUs, resource requirements of applications, bandwidth usage of the applications, maximum memory bandwidth of the GPUs, wait time of requests, and usage of micro-architectural queues. While performing experiments on real hardware, the entire focus is on characterization and prediction on real machines. The visibility into the architectural features is limited by the performance counters. Furthermore, we are limited by an architectural version of the Heisenberg uncertainty principle – the more code we add to measure runtime statistics, the more noise is introduced. This limits the amount of data that we can collect without adversely changing the runtime statistics of applications. We are limited to collecting the performance counter data (as done by other papers, which solve problems in this domain).

Given that simple analytical approaches or regression-based methods have proved to be inadequate, the current line of thinking is to model such complicated scenarios using an ML-based model that tries to fit the empirically collected data. Though neural networks are known to be very efficient for such problems, they are completely black-box models. In order to derive profound insights from the predictor model, it is a wise idea to take the middle path: use a white-box model such as decision trees or random forests that are interpretable and explainable.

4 MOTIVATION

4.1 Overview of the Benchmarks

Table 2. **Benchmarks (derived from MEVBench)**

Benchmark	Description
Sift	Extracts those features from an image that are invariant to image orientation, illumination and scaling
Surf	Feature extraction with scale invariance.
Fast	Extracts corners from an image.
Orb	Uses the FAST feature detector and the BRIEF 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.
Input: Each benchmark processes 20 VGA images in one execution round	

Table 2 describes the benchmarks used in this study. These benchmarks are the representative kernels that are used in popular applications of computer vision. They are inspired by the MEVBench [8] and SD-VBS [35] benchmark suites. We used OpenCV to generate the CPU codes of the basic kernels of these benchmarks. We then used the CUDA equivalent APIs to get the GPU-compatible codes for each of them. We used the ThunderSVM library [38] to implement the CPU and GPU versions of *SVM*. The implementation for *KNN* was developed along the lines of reference [13]. The corresponding CPU and GPU implementations are algorithmically equivalent [3].

4.2 Experimental Setup

Table 3. **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

Table 3 shows the configuration of the server where all the experiments were performed. Our 2-socket server has 48 logical cores (with hyperthreading enabled). For the experiments on the GPU, we use the latest NVIDIA Turing Tesla T4 GPU with MPS enabled. The CPU implementations are multi-threaded. We choose the number of threads that provide the least execution time. Our experiments indicated that the OpenCV-based benchmarks performed the best when parallelized with OpenMP as opposed to Pthreads or Intel TBB. For all the GPU runs, we used NVIDIA CUDA MPS [9] (explained in Section 2) to enable the concurrent execution of multiple applications. Note that **performance is defined as the reciprocal of the execution time** in our case.

4.3 Performance Variation of the Workloads

In a multi-application concurrent execution, there can be two types of bags: either all the applications are the same (homogeneous bag) or at least one application is different from the other applications (heterogeneous bag). In this section, we look at the homogeneous bag. We chose a homogeneous bag of applications for characterization purposes because identifying the source of deviation from the expected behavior is much easier for a homogeneous bag as compared to a heterogeneous bag, where the behavior of the applications in a bag is different. The obtained insights are then tested on heterogeneous bags-of-tasks in Section 6.

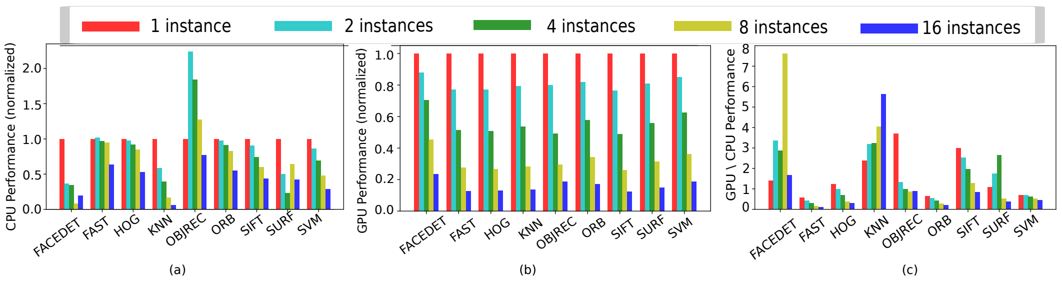


Fig. 1. (a) CPU performance (normalized), (b) GPU performance (normalized), and (c) GPU/CPU performance

Figures 1(a) and 1(b) show the performance variation of different benchmarks with the variation in the number of concurrent 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 variation in performance on the CPU with an increasing number of concurrent applications is significantly different than that on the GPU. For example, for *ObjRec*, the performance on the GPU decreases as the number of concurrent applications increases while it shows a non-deterministic behavior on the CPU. Similar is the case for other benchmarks like *Surf*, *FaceDet*, and *HoG* too. ❷ From Figure 1(c), 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 on the GPU degrades as the number of concurrent instances increases. 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 [12] in multicore CPUs and hence their scalability is much better as compared to GPUs. ❸ Figure 1(b) also shows that the trend for the performance across the benchmarks on the GPU remains roughly the same with the number of concurrent applications.

4.4 Energy Variation of the Workloads

Figures 2(a) and 2(b) show the normalized energy for executing multiple instances (2, 4, 8, and 16 instances) of the benchmarks concurrently on the multicore processor and the GPU (normalized with respect to the single-instance case), respectively. ❶ We observe that the energy consumption on the multicore processor and the GPU is fairly intuitive when we consider multiple, concurrent instances. The energy increases with an increase in the number of concurrent instances. However, the increase in the energy consumption is not necessarily worse for any one platform. For some workloads, the energy consumption on the multicore processor gets amortized with an increasing number of instances while for others the GPU is able to scale well. For example, the concurrent execution of 16 instances of KNN is highly energy-inefficient (19× as compared to the single-instance execution) on the multicore, while this is not true for the energy consumption of the corresponding execution on the GPU (5.5×). Even though there is some correlation between the

CPU and GPU energies as the concurrency increases, one cannot be used solely for the prediction of the other quantity. ❷ It is also observed that the trends for energy consumption on the GPU across the benchmarks remain the same with an increasing number of concurrent instances (see Figure 2(b)).

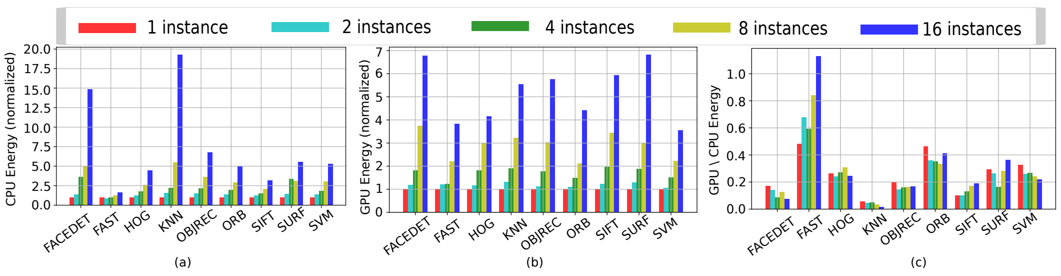


Fig. 2. (a) CPU energy (normalized), (b) GPU energy (normalized), and (c) GPU/CPU energy

Figure 2(c) shows the ratio of energy consumption on the GPU and multicore processor. ❸ We observe that the ratio of energy consumptions on the two platforms is not uniform across the workloads as the number of concurrent instances increases. For example, for SVM, the GPU energy improves over the CPU energy as the number of instances increases. On the other hand, for FAST, the reverse is true. Similar observations hold for other benchmarks. This is due to different degrees of contention in the execution of different applications. Thus, the parameters specific to the application-platform pair can play an important role in estimating the energy consumption. Based on these observations, we can gather the following insights:

Insights:
❶ The performance and energy of executing multiple instances of an application on the GPU cannot be simply correlated with the corresponding performance and energy of the execution on the multicore processor.
❷ Contention in the shared resources plays an important role when multiple applications are launched in parallel.
❸ The performance and energy of such a concurrent execution on a GPU can be directly correlated with the performance and energy of the corresponding single instance execution on the GPU.

5 METHODOLOGY

In order to predict the performance and energy of the concurrent execution on the GPU, we develop two predictors: *a power predictor* and *a performance predictor*. The power and performance predictor models can be used together to get an estimate of the energy consumption of the concurrent execution on the GPU. We model this problem as a regression task and experiment with different machine learning models to find an appropriate fit. Figure 3 shows the overall design of our predictor. Both the performance and power predictors use different sets of feature vectors to provide a prediction.

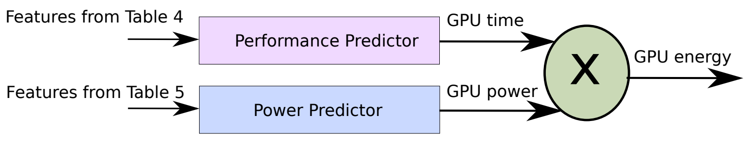


Fig. 3. Black box diagram of the predictor

A data point is an input-output pair that is used to train the machine learning model. A 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 points.

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: the training (80%) and test sets (20%). ❹ The trained model is then used at runtime to predict the power and performance of the test data points. We report the mean absolute percentage error ($MAPE(\%)$) [3] for different model and feature combinations as shown in Equation 1.

5.1 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 should be correlated with the predicted quantity.

Table 4. **Features used in the performance estimation model**

Feature	Description
CPU_time	Execution time of the benchmark on a CPU
GPU_time	Execution time of the benchmark on a GPU
SSE	% of SSE instructions
ALU	% of arithmetic instructions
MEM	% of load/store instructions
FP	% of floating point instructions
Stack	% of stack push/pop instructions
String	% of string operations
Shift	% of multiply/shift operations
Control	% of control/branch instructions
Fairness	Fairness of concurrent application execution

5.1.1 Features used in the Performance Model. There is no accepted methodology for defining features that determine the final performance [3, 6] of a bag-of-tasks on the GPU. We start with a list of features that have been proven to work well for the performance prediction of a single application on a GPU and we define a few additional features based on our insights.

Based on **Insight ❷**, we define *fairness* [26] as one of the features because we are dealing with a multi-application scenario. It quantifies the slowdown in an environment with resource contention. The equation for fairness, $fair_{\mathcal{F}}$, of a bag-of-tasks, $\mathcal{F}$, is given by: $fair_{\mathcal{F}} = min\left(\frac{IPC_k^{shared}}{IPC_k^{alone}} \Big/ \frac{IPC_l^{shared}}{IPC_l^{alone}}\right)$, where k and l are tasks in the bag.

For each task, we find its slowdown as a ratio of its IPC when it is running in a shared environment and its IPC when it is running in isolation. Fairness is the minimum slowdown divided by the maximum slowdown across all the tasks in a bag-of-tasks. The **intuition** is that since fairness captures the relative slowdown of the applications in a multi-application scenario, it can directly capture the effect of contention as mentioned in **Insight ❷**. We show in Section 6 that fairness plays an important role in reducing the prediction error.

Based on **Insight ❸**, we define *GPU execution time* for executing a single instance of an application as one of the features. The single-instance performance on GPU can be very well utilized to predict the application's behavior in a concurrent application case. Table 4 shows the list of features that we consider. The instruction mix and CPU execution time have been used in prior work. Our novelty is in combining them, along with introducing two additional features: single-instance GPU execution time and fairness (shaded rows in the table).

5.1.2 Features used in the Power Model. Table 5 shows the features that we consider in our experiments for the power predictor. These features are chosen intuitively. First, we consider the CPU and GPU power of the individual benchmarks in the bag based on **Insight ❸**. Next, we consider the features related to memory access because the energy to access the memory is much more than the energy consumed in arithmetic operations [27]. We consider IPC in addition to cache miss rates and TLB miss rates because it is affected as a result of cache miss rates. Finally, we consider branch miss rates to quantify the wasted energy as a result of wrong prediction.

The top half of Table 5 shows the features that are derived from the executions of standalone applications in the bag on the CPU. These are also called the **individual features**. These features increase with increasing the number of concurrent applications because they are defined uniquely for each application in the bag. The bottom half of Table 5 shows the features obtained from the concurrent execution of the applications in the bag on the CPU. These features are named with **_sh** in the end (see Table 5); they are also called the **concurrent features**. We consider the concurrent counterpart of all the individual features except the *GPU_power* feature because this is to be predicted. Fairness is the only exception in the nomenclature; it does not have **_sh** in its name and is a concurrent feature.

Table 5. **Features used in the power estimation model**

	Feature	Description
Individual Features	CPU_power	Power of the individual benchmarks on CPU
	GPU_power	Power of the individual benchmarks on GPU
	IPC	Instructions per cycle of individual benchmarks on CPU
	LLC_miss	% LLC misses of individual benchmarks on CPU
	Branch_miss	% branch misses of individual benchmarks on CPU
	L1_miss	% L1 misses of individual benchmarks on CPU
	DTLB_miss	% DTLB misses of individual benchmarks on CPU
Concurrent Features	IPC_sh	Instructions per cycle of the concurrent execution on CPU
	LLC_miss_sh	% LLC misses of the concurrent execution on CPU
	Branch_miss_sh	% branch misses of the concurrent execution on CPU
	L1_miss_sh	% L1 misses of the concurrent execution on CPU
	DTLB_miss_sh	% DTLB misses of the concurrent execution on CPU
	CPU_power_sh	Total power of the concurrent execution on CPU
	Fairness	Fairness of concurrent application execution on CPU

Our aim is to design a power predictor that does not rely on the architectural details of the GPU, and GPU profiling information other than the single-instance GPU power consumption. As we have argued before, GPU profiling information is hard to collect, and CPUs are in general more available. This is because we need multiple runs of an application on a GPU to obtain all the metrics from the limited number of available hardware counters [44]. Hence, we explicitly try to use as few GPU features as possible without substantially sacrificing on the final accuracy.

5.1.3 Handling Variable-sized Feature Vectors. Recall that we consider both homogeneous and heterogeneous bags of applications in our characterization 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.

We handle this scenario by training different predictors for different degrees of concurrency. This is primarily because it is not possible to efficiently train an ML model using variable-sized feature vectors. Hence, we use an ensemble of predictors and the appropriate predictor is chosen at runtime based on the size of the feature vector.

5.1.4 The Fairness Feature. The way the fairness metric is defined is quite problematic – it involves running all combinations of benchmarks on multicore CPUs, which has a prohibitive overhead. For a use-case of 60 workloads that can be executed with a concurrency of 3, we will need to execute 37,820 unique combinations, which is prohibitive. Furthermore, it raises the question that why can't we run a similar, short profiling run on GPUs and not use our prediction model.

Hence, to ameliorate the situation, we propose a new fairness metric; we shall later show that we achieve good accuracy using this metric, and we no more need to collect data for all combinations of workloads on multicore CPUs. With our new metric, we exclusively use statistics that are specific to a given workload and do not require other real-world workloads to generate these statistics.

We first try to find a pattern in the behavior of these workloads and their combinations (with a concurrency 2, 3, and 4). We perform a clustering on a total of 530 combinations (called data points in ML jargon). We deemed 530 to be a large enough number based on experimental results. After clustering, we measure its efficacy using the popularly used silhouette score [31]. For each datapoint, we consider its fairness of execution, the percentage of LLC misses and the percentage of L1-D cache misses as the features. Based on these features, we cluster these data points and find 3 unique tightly bound clusters (silhouette score of 0.75 as shown in Figures 4, and 5). Figure 4 shows the clusters of the workload combinations on the basis of the features described above. Figure 5 shows the silhouette score of the clusters. A silhouette score value more than 0.7 implies that the clusters are tightly bound.

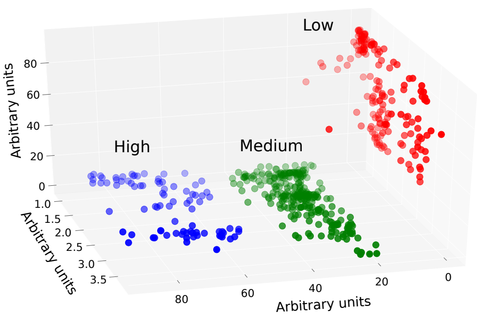


Fig. 4. Clustering of workloads

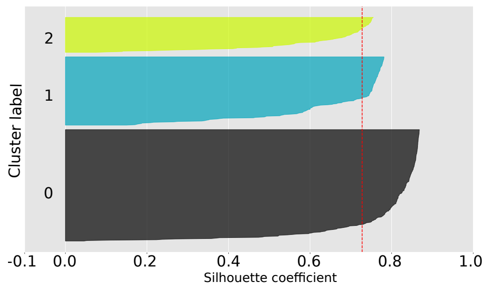


Fig. 5. Silhouette score (a score above 0.7 means that the clusters are tightly bound)

Upon further analysis, we found that these clusters correspond to the memory behavior of the workload combinations. In our workload combinations, all the combinations with KNN as one of the workloads witness a low percentage of LLC misses, while those with SIFT witness a high percentage of LLC misses. For all the other cases, the percentage of LLC misses is modest. This observation is very strong because it is generic and holds regardless of the level of concurrency. In order for this scheme to scale well to any number of workloads and unseen workloads, we designed three microbenchmarks corresponding to high, medium, and low percentages of LLC misses, respectively. We then calculated 3 fairness values for each of our workloads when run along with these three microbenchmarks (with a concurrency of 2).

At the time of training the ML model, we choose a fairness value (out of the 3 fairness values) for each workload on the basis of the total LLC misses of the other workloads that are running concurrently with it. We add the total LLC misses of the rest of the workloads and then categorize the sum into three categories based on two threshold values. The categories are Low, Medium, and High. Then, based on the category, we choose a fairness value (collected in the offline, pre-processing phase). This is done for all the workloads that are running together. Thus, all the features are now obtained from either the execution of the standalone workloads or the concurrent execution of the workloads with three microbenchmarks.

Hence, we present a linear time solution, where all that we require is a workload-specific vector that can be generated by the developer of a workload offline. The workload-specific vector for the performance predictor contains the instruction mix, CPU execution time, GPU execution time of the workload, and 3 fairness values obtained by executing it with the three microbenchmarks. As will be shown in Section 7.3, the final set of features for the power predictor includes the following features: fairness, % LLC misses, CPU power, GPU power of standalone workloads. Hence, the workload-specific vector for the power predictor contains the % LLC misses, CPU power, GPU power of the workload, and 3 fairness values obtained by executing it with the three microbenchmarks.

Microbenchmarks: The task is to create microbenchmarks that are broadly similar to real-world workloads that we consider (similar to vision and ML workloads) and exhibit the memory behavior depicted by the three clusters. The three microbenchmarks are named **Low**, **Medium**, and **High** based on the percentage of LLC misses that they witness. We describe these microbenchmarks below.

(1) **High:** This microbenchmark reads a new image in every iteration and then adds the pixels of the image frame to a 2D array and stores it in another 2D array.
(2) **Medium:** This microbenchmark reads a new image and an element of a 1D array in every iteration. Subsequently, it increments the value read from the 1D array and stores it in a temporary variable (limited compute phase).
(3) **Low:** This microbenchmark keeps reading the same image and an element of a 1D array in every iteration. Subsequently, it increments the value read from the 1D array and stores it in a temporary variable. The primary difference with respect to the **Medium** benchmark is that the same image is read here in every iteration, which is a regular VGA image.

5.2 Creating the Data Points

Table 6. **Configurations of ML models**

Predictors	**Scikit configuration (best)**
Decision Tree (DT)	max_depth=10
Gradient Boosting (GB)	loss=least squares regression
Linear Regression (linear)	fit_intercept=True
Multi-layer Perceptron (MLP)	hidden layer=1, neurons=100, activation=sigmoid
Random Forest (RF)	max_depth=10, n_estimators=100
Support Vector Regression (SVR)	kernel=rbf

The predictor model is trained on a set of training data points, which are input-output pairs. The input is the feature vector of the data point and the output is its performance and power on the GPU for the performance and power models, respectively. Thus, to create a data point, we need to have equivalent CPU and GPU implementations of the benchmarks (also explained in Section 4.1). 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 [3, 6]. Any change in the input of the benchmark changes its execution time and energy consumption and hence the feature vector. It can be considered a new data point [3]. Another method of increasing the data points is by permuting the benchmarks (shown in Table 2) to generate different combinations of concurrent applications, also called a *heterogeneous bag*. In total, we examine *81 datapoints* for the execution of two concurrent applications, *120, 330, and 792 datapoints* for three, four, five concurrent applications, respectively. In general, such applications of machine learning in computer architecture use training data that has a limited size. This is because it is limited by the number of unique benchmarks and their input combinations. For example, Baldini et al. [6] used 48 unique data points for training. Similarly, Ardalani et al. [3] generated 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 by varying the inputs to the benchmarks.

5.3 Collection of Features

We use the PIN 3.7 [22] and MICA 1.0 [18] tools to capture the percentages of different kinds of instructions. The IPC data for different workloads in the bag (to be used in the calculation of fairness) is calculated using the Linux Perf tool. We use the Linux Perf tool to collect the memory-related features on the multicore processor. The power values for the GPU execution are obtained using *nvidia-smi*. All the values are normalized using the min-max scaling technique.

5.4 Predictor Models

We used the open-source machine learning library Scikit-learn to implement the regression models for our prediction tasks.

5.4.1 Performance Predictor. As explained in Section 2.2, 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 with SVR as compared to that with the decision tree ($max\_depth = 7$). We thus evaluate the decision tree-based performance predictor model in Section 6.

5.4.2 Power Predictor. Table 6 shows the configurations of the ML models that we compared. We consider the most popular classic ML regression models: Decision Tree, Random Forest, Support Vector Regression, Multi-layer Perceptron, Linear Regression, and Gradient Boosting Regression. We do not use CNNs for our purpose because CNNs are known to work well for datasets where there is spatial or temporal dependence in the data. CNNs can capture this dependence using a set of filters. In our case, there is no such dependence in the dataset or the extracted features. We train all the 6 ML models using the training data and validate them using the LOOCV [25] validation method, and test them by randomly splitting the entire dataset into 80% training data and 20% test data (standard practice) in Section 6.

6 PERFORMANCE PREDICTOR: RESULTS AND ANALYSIS

In this section, we evaluate the accuracy of the performance predictor on both the homogeneous and heterogeneous bags for the execution of two concurrent applications. We use the features as described in Table 4 along with the new fairness metric as defined in Section 5.1.4. We compare different schemes using the MAPE metric as defined in Equation 1. The setup is described in Section 4.

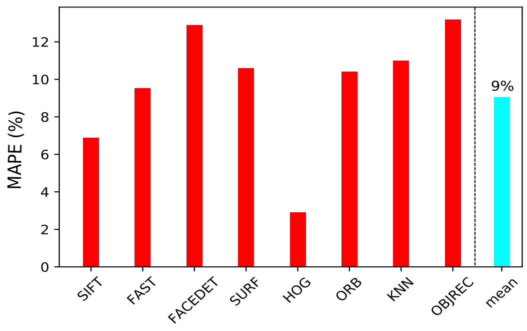


Fig. 6. MAPE (%) for leave-one-out cross validation

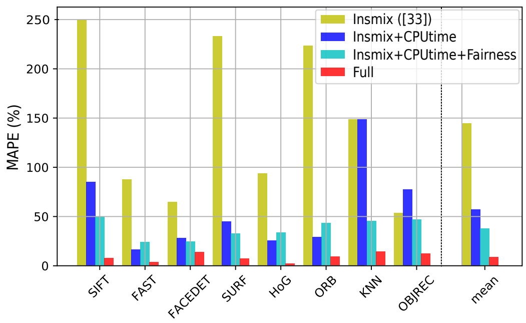


Fig. 7. Comparison of different feature combinations

6.1 Cross-validation

Before we test our trained model on unseen test data, there is a need to validate the model to assess the generalizability of the model. We 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 the leave-one-out cross-validation (LOOCV) [6] technique. The basic idea is to leave one data point from the entire dataset for validation and use the 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 6), we leave out all the data points corresponding to that benchmark for validation and use the rest for training. This ensures that the validation data is unseen. Figure 6 shows the cross-validation error for each benchmark. We observe a 9% mean absolute percentage error for predicting the execution time of running two applications concurrently on the GPU. Here each x-axis label denotes the benchmark that was left out in the LOOCV process.

6.2 Comparison of Different Feature Combinations

Figure 7 shows the comparison of four different feature combinations on the basis of their MAPE in the prediction. The first scheme uses *insmix* (instruction mix) for training the decision tree-based predictor; these set of features were also used by Baldini et al. [6]. Though they achieved a high accuracy for predicting the range of speedups on a GPU for a single application case, the scheme does not capture the interactions among the applications when they run concurrently. Hence, it has large errors (144.6%) as shown in Figure 7. The next combination contains *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 execution time of the concurrent execution on the GPU.

The third combination contains the *CPU_time*, *fairness* and *insmix* as the features in the feature vector. This scheme performs significantly better than the first two schemes (89.55% and 19.32% better, respectively) because both CPU_time and fairness are able to capture the performance degradation and contention in a bag-of-tasks. Lastly, we use all the features as described in Table 4. Since the *GPU_time* of an application is the most correlated with its concurrent 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 state-of-the-art machine learning model [6] for GPU performance prediction (reference work by Baldini et al. [6] which is the left-most bar).

6.3 Sensitivity Analyses of the Prediction Error

Figure 8 shows the effect of CPU_time, GPU_time, fairness, and instruction mix on the prediction error. It shows a heatmap of the prediction error for different feature combinations. The darker shade implies a higher error. It can be observed that for any feature combination (y-axis labels), the prediction error decreases with the introduction of CPU_time and GPU_time (except the ALU+SSE+fairness) in the combination. The reduction in error with the introduction of GPU_time is even more pronounced than that achieved by the introduction of CPU_time. This was also evident in the insights in Section 4, where we concluded that CPU_time alone cannot be a good feature for prediction while GPU_time of an application is positively correlated with its multi-instance performance on the GPU. Thus, a general insight

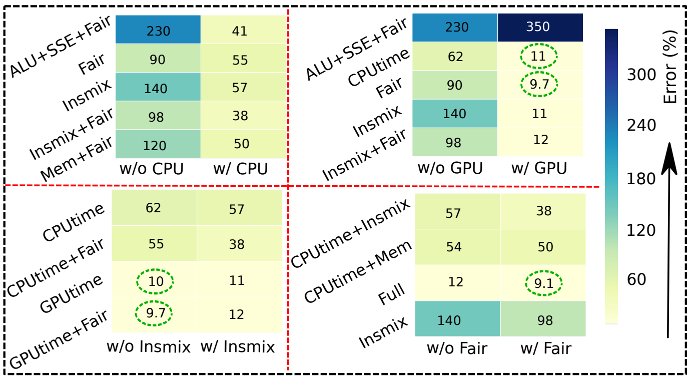


Fig. 8. MAPE (%) for different feature combinations

is that having CPU_time and GPU_time (for running a single instance of the benchmark) in the feature vector leads to a better splitting in the decision tree and hence better prediction.

Additionally, it can be observed that the MAPE of prediction increases when fairness is combined with the instruction mix (90% to 98%). The error increases even 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. The introduction of CPU_time reduces the error to a large extent. Another observation is that ALU+SSE+fairness combination 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 (ALU+SSE) instructions. However, this relationship reverses with the introduction of the CPU_time. This is because CPU_time in combination with the compute (ALU+SSE) instructions and fairness can accurately predict the performance degradation on a GPU.

In general, the addition of instruction mix to the CPU_time makes the prediction accuracy better while it has no positive impact when it is exclusively combined with the GPU_time. The introduction of fairness as a feature to any combination has a positive effect on the prediction error.

6.4 Analyses of the Decision Paths

Based on the observations in Section 6.3, we can conclude that the features proposed by us reduce the error. In this section, we perform an even deeper analysis of which features are actually used in the decision making. 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 comparison during decision making, the branches, the leaf nodes, and the final prediction in the leaf nodes. Every test data point follows its own path in the decision tree. We answer a few questions based on our analyses.

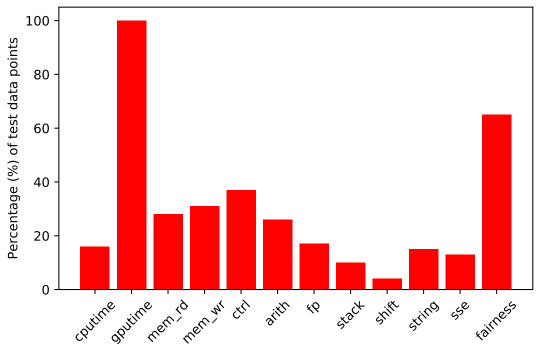


Fig. 9. Percentage of the test data points containing a feature in their decision path

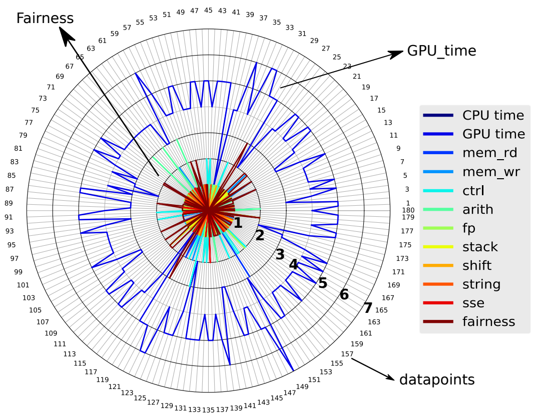


Fig. 10. Radar plot of the frequency of each feature in the decision making for different test points

Is fairness more important than the instruction mix? The answer to this question is **yes**. Figure 9 shows the percentage of test points that utilize a particular feature in their decision paths. We can observe that *GPU_time* and *fairness* occur in 100% and 65% of the test data points, respectively, to decide one or more splits. This experiment confirms the presence of a feature in the decision paths of the test data points, but it does not convey how many times the same feature has been used at different decision nodes for making the prediction. We answer this question next.

What is the importance of each feature? Figure 10 shows the number of times a feature is used in the decision path of a test point. The radius of 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 (in the circumference) used in our experiments (test data points are 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 data 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 data points. The rest of the features are still important and appear at least 1-2 times in the decision paths of the test data points.

The summary of our findings are:

❶ CPU_time has a positive effect on the prediction error when combined with the insmix feature.
❷ Fairness improves the prediction error when used with a combination of CPU_time and insmix.
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 effect of CPU_time.

7 POWER PREDICTOR: RESULTS AND ANALYSIS

7.1 Experimental Setup

Table 7 shows the combinations of the features that we used in our experiments. The list is not exhaustive but covers all the relevant combinations. We use the following intuition behind these combinations:

❶ Analyze the prediction error solely on the basis of individual features (see Table 5) and at most one concurrent feature, *fairness*. This combination has the overhead of collecting only the individual features because the *fairness* feature is available from the performance predictor model. We design comb1 to comb6, and comb16 in this pattern.

❷ Analyze the prediction error solely on the basis of concurrent features. The overheads include running the applications in the bag concurrently on the multicore processor to obtain these features. We design comb7 to comb9, comb12, and comb15 using this intuition.

❸ To analyze the effect of CPU and GPU power of the individual benchmarks in the bag-of-tasks on the prediction accuracy, we design another category of combinations using the individual features except the individual power values of the benchmarks on the two platforms. Comb10, comb11, comb13, and comb14 fall in this category. The overheads are similar to ❶.

❹ Lastly, we also design the combinations using a mix of individual features and concurrent features (more than one). Comb17 to comb20 fall in this category. This category has higher overheads because we need to run both the individual benchmarks and the bag-of-tasks to obtain the features.

7.2 Cross-Validation

We use the same cross-validation methodology as used in the performance predictor model. Figure 11 shows the violin plot for the 6 ML models trained using 20 different feature combinations. Each violin plot shows the error distribution across the 20 combinations for all the 6 ML models. The separate violin plot for each benchmark indicates that the ML models were trained by leaving out the bags-of-tasks containing the corresponding benchmark and validated using the left out bags-of-tasks. The width of the violin plot at a particular error (y-axis) is directly proportional to the number of feature combinations having that error. We make the following observations:

❶ SVR has the least spread of the violin and hence the least variance in the prediction error across different combinations of features for all the benchmarks. Its mean error is at par with RF, DT, and GB for all the benchmarks except *FaceDet* and *Orb*, where it performs poorly. A low variance in the prediction error of SVR with the changing feature combinations suggests that the SVR model

Table 7. **Combination of features and MAPE (%) of the regressors**

Num	Combination of features	DT	GB	Linear	MLP	RF	SVR
Comb1	CPU_power, GPU_power, IPC, LLC_miss, fairness	4.65	3.9	7.52	10	3.87	6.7
Comb2	CPU_power, IPC, LLC_miss, fairness	6.75	7.5	10.95	13	8.1	6.7
Comb3	GPU_power, IPC, LLC_miss, fairness	4.65	4.22	6.2	9.3	3.72	7.2
Comb4	CPU_power, GPU_power, IPC, LLC_miss	4.55	3.78	7.48	10.92	3.8	6.69
Comb5	CPU_power, GPU_power, IPC, fairness	4.49	3.83	4.57	8.43	3.95	6.6
Comb6	CPU_power, GPU_power, IPC	4.41	3.76	4.53	7.58	3.88	6.65
Comb7	CPU_power_sh, IPC_sh, LLC_miss_sh, fairness	13.5	11.05	10.3	10.48	11.34	7.29
Comb8	CPU_power_sh, IPC_sh, LLC_miss_sh, fairness, Branch_miss_sh, L1_miss_sh, DTLB_miss_sh	9.42	8.23	21.17	11.64	8.75	7.09
Comb9	IPC_sh, LLC_miss_sh, fairness, Branch_miss_sh, L1_miss_sh, DTLB_miss_sh	9.8	8.76	12.67	24.83	8.89	7.47
Comb10	IPC, LLC_miss, Branch_miss, L1_miss, DTLB_miss, fairness	8.68	7.27	16	11.55	7.49	7.44
Comb11	IPC, LLC_miss, Branch_miss, L1_miss, DTLB_miss	8.09	7.09	16.44	11.93	7.44	7.45
Comb12	IPC_sh, LLC_miss_sh, Branch_miss_sh, L1_miss_sh, DTLB_miss_sh	9.85	9.22	18.2	21.9	9.2	7.47
Comb13	IPC, LLC_miss, fairness	13.82	9.08	12.67	20.22	10.76	7.56
Comb14	IPC, LLC_miss, Branch_miss, L1_miss, fairness	8.8	7.73	13.39	14.38	7.56	7.47
Comb15	CPU_power_sh, IPC_sh, Branch_miss_sh, L1_miss_sh, DTLB_miss_sh	9.9	8.36	23.4	13.16	8.6	6.94
Comb16	CPU_power, GPU_power, LLC_miss, fairness	4.3	3.78	7.14	9.49	3.59	6.65
Comb17	GPU_power, LLC_miss, fairness, CPU_power_sh	4.65	3.97	5.40	8.40	3.41	7.08
Comb18	GPU_power, IPC, LLC_miss, fairness, CPU_power_sh	4.83	4.10	6.39	10.63	3.68	7.05
Comb19	GPU_power, IPC, LLC_miss, Branch_miss, L1_miss, DTLB_miss, fairness, CPU_power_sh	4.75	3.78	16.66	12.71	3.66	6.97
Comb20	GPU_power, CPU_power_sh, IPC_sh, LLC_miss_sh, Branch_miss_sh, L1_miss_sh, DTLB_miss_sh, fairness	4.17	3.62	16.86	13.95	3.66	7.01
The individual features appear in a pair of 2. For example, for comb1, the number of features is 2*4+1 for a concurrency of two.							

is saturated and is not able to improve with the addition or removal of any feature. However, other models show significant variations in the prediction error for different feature combinations.

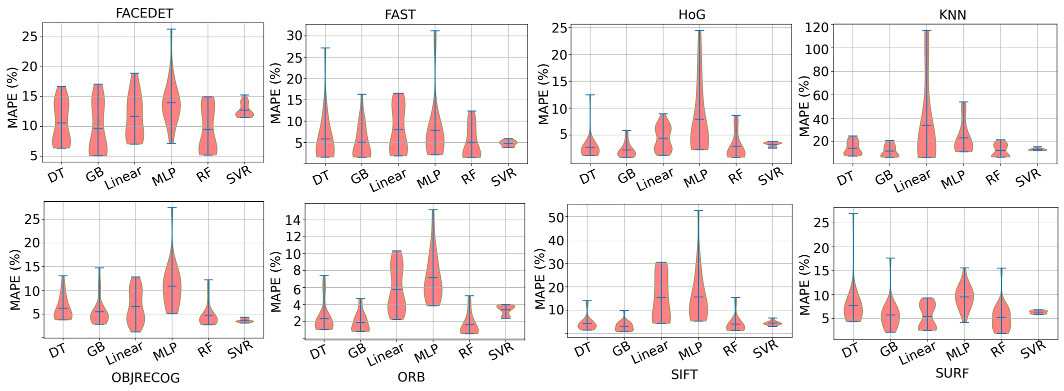


Fig. 11. Violin Plots showing the error distribution of the 20 combinations for 6 ML models for each benchmark

❷ MLP and Linear Regression perform the worst for all the benchmarks except SURF, where linear regression is better than all the other models. This implies that MLP and Linear Regression models are not able to generalize to unseen data.

❸ DT, RF, and GB perform nearly equally for all the benchmarks except SURF, and OBJRECOG, where DT is inferior to RF and GB.

7.3 Comparison of Different Feature Combinations

Figure 12 and Table 7 show the mean absolute percentage error of different ML models for different combinations of features averaged across all the benchmarks (from Figure 11). We observe the following trends. Focusing on the three best models, DT, RF, and GB (abbreviations defined in

Table 6), we observe in Figure 12 that the feature combinations can be divided into three categories based on the percentage of error: high error, medium error, and low error.

Referring to Table 7, Comb1, comb3-6, and comb16-comb20 fall in the low error category because the error is less than 5% (cyan color). Comb2, comb10, comb11, and comb14 fall in the medium error category as they have an error between 5 to 9% (grey color). Comb 7-9, comb12, comb13, and comb15 fall in the high error category because their error is greater than 9% (white color). We observe that the combinations in the high error category are those that use the shared metrics as features (see features 8-13 in Table 4 and Table 7) except comb13. Comb13 falls in the high error category because of the absence of any power feature: CPU_power or GPU_power. From hereon, we perform detailed analyses of the DT, RF, GB, and SVR models. We consider SVR because it is equally accurate as the top-3 models, however its prediction error always lies between 6-8% for any combination of features (see Table 7).

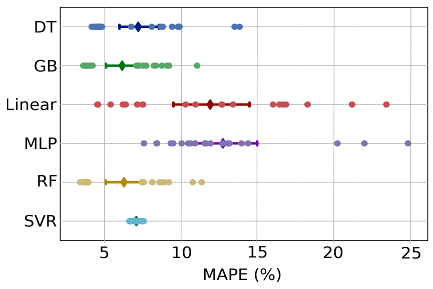
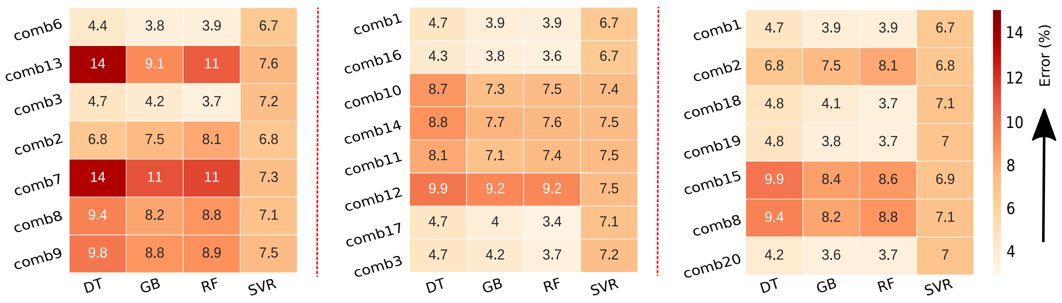


Fig. 12. Strip Plot showing the MAPE for 20 combinations of features using LOOCV



(a) Heatmap1 **(b) Heatmap2** **(c) Heatmap3**

Fig. 13. Heatmaps of different feature combinations and ML models (results show the MAPE (%) error)

Figure 13 shows the heatmap of the power prediction error for different combinations of features and ML models. We show the heatmap for the ML models that had low MAPE in the violin plots shown in Figure 11, that is, RF, DT, GB, and SVR. The feature combinations in each heatmap are grouped such that any two consecutive rows in a heatmap differ by atmost two features or differ in the category of features (concurrent or individual).

Heatmap 1:

❶ Comparing comb13 and comb6 in Figure 13(a), we observe that replacing the CPU and GPU power of the individual benchmarks (comb6) by the LLC miss rate and fairness (comb13) leads to a 2.5-3X increase in the prediction error across all the ML models. This is because IPC, percentage of LLC miss rates and fairness are relative metrics and cannot accurately predict the absolute power of the concurrent execution on their own.

❷ Comparing comb13, and comb2 in Figure 13(a), we observe that upon removing CPU power of individual benchmarks in comb13, the accuracy drops by upto 2X. Similarly, with the removal of GPU power of individual benchmarks (compare comb13 and comb3), the accuracy drops by 3X. Owing to the reason stated in the previous observation, the accuracy drops in the absence of a feature that can quantify power in absolute terms. Moreover, since GPU power of the concurrent execution is more correlated with the GPU power of the individual workloads, it has a greater impact on the prediction accuracy as compared to the CPU power of the individual workloads.

❸ Comparing comb2 and comb7 in Figure 13(a), we observe that with the use of concurrent features in comb7, the error becomes 2X for DT and 1.5X for RF and GB. This is because the standalone features have more information about a workload's characteristics than concurrent features. This is suitably exploited by the ML model to train a more accurate predictor.

❹ Comparing comb7 and comb8 in Figure 13(a), we observe that if we use all the concurrent features to train as opposed to a subset of the concurrent features, we can reduce the error by upto 4.5% for DT, RF, and GB. This is because of the addition of branch misses to the feature vector. A wrong branch can lead to unnecessary power dissipation due to the execution of wrong instructions in the pipeline.

❺ Comparing comb8 and comb9 in Figure 13(a), we observe that by using the CPU power of the concurrent execution as one of the features along with a combination of memory-related concurrent features, the prediction error reduces by a very small amount (up to 0.6% for DT, RF, GB). Owing to a low correlation between the GPU power of the concurrent execution and CPU power of the concurrent execution, the observed reduction in error is low.

Heatmap2:

❶ Comparing comb1 and comb16 in Figure 13(b), we observe that the IPC of the individual benchmarks has a negligible effect on the error. With the removal of IPC, the error improves by up to 0.4%. This is because the useful information regarding the individual IPCs is already captured in the fairness metric, hence individual IPC values have negligible impact.

❷ Comparing comb10 and comb14 in Figure 13(b), we observe that the DTLB miss rate of the individual benchmarks reduces the error by only up to 0.4% for the DT, RF, and GB models. This is because the major contributors to power in the concurrent execution are the shared resources, hence private DTLBs do not make much difference.

❸ Upon comparing a combination of the concurrent features (see Figure 13(b)) (comb12) with the corresponding individual features (comb11), we observe an increase in the percentage prediction error (up to 2%) with the concurrent features as compared to the individual features. This is because the individual features contain more information as compared to the corresponding concurrent features, hence the trained predictor is more accurate.

❹ Adding CPU_power_sh to a subset of individual features reduces the error by up to 0.5% (see comb3 and comb17 in Figure 13(b)). The effect is negligible owing to the already stated reason: low correlation of the CPU power of the concurrent execution with the GPU power of the concurrent execution.

Heatmap 3:

❶ Comparing comb1 and comb2 in Figure 13(c), we observe that the removal of the GPU_power feature has an adverse effect on all the ML models. The error increases by up to 4.5%. This is because the GPU power of standalone execution is the most correlated with the GPU power of the concurrent execution. Hence, its removal leads to an increase in the prediction error.

❷ Comparing comb18 and comb19 in Figure 13(c), we observe that the percentage of branch_miss rates, L1_miss rates, and DTLB_miss rates have a negligible effect on the error for DT, GB, and RF (up to 0.3%). Their effect is being overshadowed by the presence of the GPU power of standalone workloads, which is a major determinant of the GPU power of the concurrent execution.

❸ Comparing comb15 and comb8 in Figure 13(c), we observe that the addition of fairness and LLC_miss_sh can have a positive effect on the prediction error for all the ML models. The error decreases by up to 0.5%. It is worth noting that when CPU_power_sh is used with the individual features, the error is in the range of 4-5%, while using it with the corresponding concurrent features makes the error as large as 9-10% (compare comb19 and comb8).

❹ Adding the individual GPU_power to the combination of concurrent features as in comb8 and comb20, the error reduces by up to 5% (see Figure 13(c)) owing to the reasons explained above.

From the above discussion, we derive the following insights.

Insights:
❶ Addition of GPU_power to any combination of concurrent features reduces the error by at least 2X suggesting that GPU_power has the maximum positive impact in reducing the prediction error.
❷ The percentages of L1_miss, Branch_miss, and DTLB_miss rates of standalone applications have a negligible effect on accuracy.
❸ The top two combinations in terms of accuracy are comb16 and comb20. We finally choose comb16 because it does not require any of the concurrent features. The final features are: GPU_power, fairness, CPU_power, and LLC_miss (see Table 5).
❹ The final models are DT, GB, RF, and SVR.

7.4 Testing the Model: Random Split

For testing with a random subset of 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 phase. It is used only during evaluation to evaluate the error in our predictions. However, a random split does not guarantee that a benchmark is not seen before. This is something that we need to ensure and we have shown the insights from these experiments in Section 7.2.

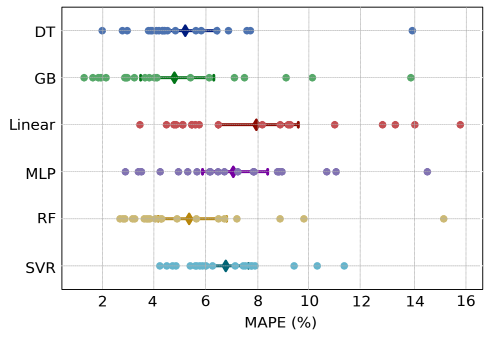


Fig. 14. Strip plot showing the MAPE for 20 combinations of features

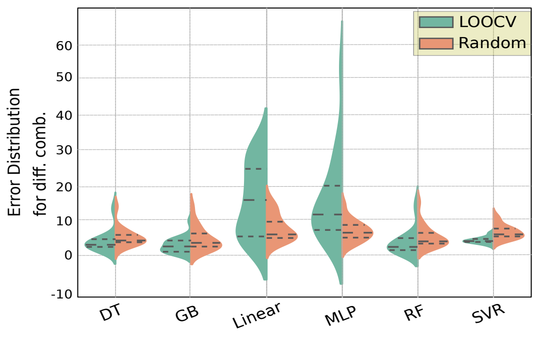


Fig. 15. Split violin plot showing the error distribution with LOOCV and random data split validation

Figure 14 shows the MAPE error for all the feature and model combinations. We observe that with random splits, MLP, and Linear regression models also perform significantly well. Their mean error across different combinations was greater than 10% for the LOOCV validation data (see Figure 12), while it becomes less than 8% for the random split. This is because the data points are split randomly between the training and test sets. Thus, each benchmark appears in the training set, albeit with a different set of concurrent applications. Hence, we can conclude that MLP, and Linear regression models do not generalize well for completely unseen benchmarks.

Figure 15 shows the split violin plot of the error distributions across the 20 feature combinations for the LOOCV validation dataset and the random dataset. The spread of the error across different combinations is much more constrained with the random dataset for MLP and Linear regression models, while it is nearly the same for the other models. It is also evident that DT, GB, and RF are the top-3 models that are both generalizable and sufficiently accurate. This can be concluded from both the LOOCV and random split experiments.

8 GENERALIZATION TO A HIGHER CONCURRENCY

Till now, we had discussed all the results by considering the execution of two concurrent applications. In this section, we show that our features are easily generalizable to the execution of three or more concurrent applications. Recall that the individual features correspond to each application present in the bag-of-tasks. Hence, with an increase in the number of concurrent applications, the number of individual features will increase and their concatenation will form a larger feature vector. For example, the size of the feature vector for three concurrent applications will be more than its size for two concurrent applications. Note that training ML models with variable-sized feature vectors is still an open problem. Thus, we cannot train the same predictor for the concurrent executions with different number of concurrent applications. We, thus, solve this problem by deploying multiple predictors.

We observe that all the three models DT, RF, and GB have a tolerable error between 8-10% for predicting the performance of executing three applications concurrently. We obtained 9% error with DT for executing two concurrent applications (see Section 6). Hence, the error does not get worse with an increase in the number of concurrent applications. For the power predictor too, all the top models DT, RF, GB, and SVR have a tolerable error between 5-7% for executing three applications concurrently, which was between 3-6% for the case of two concurrent applications. Hence, the features generalize well with an increase in the number of concurrent applications for both the power and performance models.

Upon increasing the number of concurrent applications to 4 and 5, the performance prediction errors still remain tolerable, 7.2% and 8.4%, respectively. The power prediction errors for the two cases are 7.7% and 8.5%, respectively. Thus, our features for both the predictor models are easily generalizable. Also note that the GPU execution times for the concurrent execution are $0.5 - 1.3$ s, $0.58 - 1.4$ s, and $0.6 - 1.5$ s for a concurrency of 3, 4, and 5, respectively, while predicting the GPU time for a concurrent execution takes only 1.3 ms. Hence, using our predictor can save a lot of time.

9 GENERALIZATION TO DIFFERENT GPU ARCHITECTURES

The basic idea is that our model should generalize to unseen GPUs. Hence, if the workload-specific model card for an application is collected on Nvidia Tesla T4, the model should be able to predict the performance on another GPU with a reasonably different architecture, such as Nvidia P100, fairly accurately. However, making a model generalize to a different GPU architecture is difficult because the computational power can vary significantly across GPU generations.

Since there are significant differences in the architectural capabilities and power requirements of GPUs belonging to different generations, we need a *machine translation card* in addition to the previously proposed *application model card* to be able to predict accurately for a different GPU. This is a ***one-time, offline*** mechanism and can be obtained using micro-benchmarks. We used the popular CUDA examples from OpenCV as microbenchmarks and measured the performance of these examples on both the GPUs. We then developed a linear regression model to predict the performance on Nvidia P100 given the performance of a benchmark on Nvidia Tesla T4. We call this model a *machine translation model.* Our linear regression curve or the machine translation model predicts the performance with an accuracy of 91-93% for all the four cases with 2, 3, 4, and 5 concurrent applications. When this model is used in conjunction with the performance predictor (developed earlier), the overall performance prediction accuracy drops by 1-2%. We still remain within the 10% tolerable error margin. Hence, our models are easily generalizable across GPU architectures.

10 CONCLUSION

In this paper, we debunked the popular belief that GPUs are better than CPUs for computer vision workloads in general. We performed analyses with concurrent workloads and found that there is no clear winner in terms of the platform. Because of contention at the shared resources and inefficient contention handling policies in GPUs, the performance of a concurrent execution has scalability issues. Given that these workloads are growing in prevalence with the growth in IoT/edge computing, there is a need to predict beforehand if offloading these workloads to a GPU would be beneficial in terms of both performance and power. Hence, there is a need to accurately predict the performance and power of such ensemble workloads on a GPU. In this paper, we show that the following features are sufficient to predict the performance and power of a concurrent execution: the fairness of execution when these workloads are executed with three representative microbenchmarks on CPUs and the performance and power statistics obtained from the execution of standalone workloads on CPUs and GPUs. This is the first such work in the direction of performance and power prediction for concurrent applications that does not rely on features extracted from concurrent executions, GPU-based performance counters, or architectural parameters of GPUs. We also show a method to extend our models for up to 5 concurrent applications. We obtain an accuracy of 91% and 96% for the performance and power predictor models, respectively (while executing two concurrent applications).

REFERENCES

[1] Mohammad Aazam, Sherali Zeadally, and Khaled A Harras. 2018. Offloading in fog computing for IoT: Review, enabling technologies, and research opportunities. *Future Generation Computer Systems* 87 (2018), 278–289.

[2] Masab Ahmad, Halit Dogan, Christopher J Michael, and Omer Khan. 2019. HeteroMap: a runtime performance predictor for efficient processing of graph analytics on heterogeneous multi-accelerators. In *2019 IEEE ISPASS*. IEEE, 268–281.

[3] Newsha Ardalani, Clint Lestourgeon, Karthikeyan Sankaralingam, and Xiaojin Zhu. 2015. Cross-architecture performance prediction (XAPP) using CPU code to predict GPU performance. In *MICRO*. ACM.

[4] Akhil Arunkumar, Evgeny Bolotin, David Nellans, and Carole-Jean Wu. 2019. Understanding the future of energy efficiency in multi-module gpus. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 519–532.

[5] Rachata Ausavarungnirun, Vance Miller, Joshua Landgraf, Saugata Ghose, Jayneel Gandhi, Adwait Jog, Christopher J Rossbach, and Onur Mutlu. 2018. Mask: Redesigning the gpu memory hierarchy to support multi-application concurrency. In *ACM SIGPLAN Notices*, Vol. 53. ACM, 503–518.

[6] Ioana Baldini, Stephen J Fink, and Erik Altman. 2014. Predicting gpu performance from cpu runs using machine learning. In *SBAC-PAD*. IEEE.

[7] Lorenz Braun, Sotirios Nikas, Chen Song, Vincent Heuveline, and Holger Fröning. 2020. A simple model for portable and fast prediction of execution time and power consumption of GPU kernels. *ACM TACO* 18, 1 (2020), 1–25.

[8] Jason Clemons, Haishan Zhu, Silvio Savarese, and Todd Austin. 2011. MEVBench: A mobile computer vision benchmarking suite. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*. IEEE, 91–102.

[9] NVIDIA Corp. [n.d.]. *Multi-Process Service*.

[10] Nvidia Corp. [n.d.]. *NVIDIA Ampere GPU ARCHITECTURE*.

[11] Nvidia Corp. [n.d.]. *NVIDIA TURING GPU ARCHITECTURE*.

[12] Alexandra Fedorova, Sergey Blagodurov, and Sergey Zhuravlev. 2010. Managing Contention for Shared Resources on Multicore Processors. *Queue* 8, 1 (2010), 20.

[13] Vincent Garcia, Eric Debreuve, Frank Nielsen, and Michel Barlaud. 2010. K-nearest neighbor search: Fast GPU-based implementations and application to high-dimensional feature matching. In *ICIP*. IEEE, 3757–3760.

[14] Guin Gilman and Robert J Walls. 2021. Characterizing concurrency mechanisms for NVIDIA GPUs under deep learning workloads. *Performance Evaluation* 151 (2021), 102234.

[15] Sunpyo Hong and Hyesoon Kim. 2009. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *ACM SIGARCH Computer Architecture News*, Vol. 37. ACM, 152–163.

[16] Sunpyo Hong and Hyesoon Kim. 2010. An integrated GPU power and performance model. In *Proceedings of the 37th annual international symposium on Computer architecture*. 280–289.

[17] Seyedmehdi Hosseinimotlagh, Farshad Khunjush, and Rashidaldin Samadzadeh. 2015. SEATS: smart energy-aware task scheduling in real-time cloud computing. *The Journal of Supercomputing* 71, 1 (2015), 45–66.

[18] Kenneth Hoste and Lieven Eeckhout. 2006. Comparing benchmarks using key microarchitecture-independent characteristics. In *2006 IEEE International Symposium on Workload Characterization*. IEEE, 83–92.

[19] Adwait Jog, Onur Kayiran, Tuba Kesten, Ashutosh Pattnaik, Evgeny Bolotin, Niladrish Chatterjee, Stephen W Keckler, Mahmut T Kandemir, and Chita R Das. 2015. Anatomy of gpu memory system for multi-application execution. In *Proceedings of the 2015 International Symposium on Memory Systems*. ACM, 223–234.

[20] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: Enabling energy optimizations in GPGPUs. *ACM SIGARCH Computer Architecture News* 41, 3 (2013), 487–498.

[21] Jieun Lim, Nagesh B Lakshminarayana, Hyesoon Kim, William Song, Sudhakar Yalamanchili, and Wonyong Sung. 2014. Power modeling for GPU architectures using McPAT. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 19, 3 (2014), 1–24.

[22] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, Vol. 40. ACM, 190–200.

[23] Cheng Luo and Reiji Suda. 2011. A performance and energy consumption analytical model for GPU. In *2011 IEEE ninth international conference on dependable, autonomic and secure computing*. IEEE, 658–665.

[24] Xiaohan Ma, Mian Dong, Lin Zhong, and Zhigang Deng. 2009. Statistical power consumption analysis and modeling for GPU-based computing. In *ACM SOSP HotPower*, Vol. 1. Citeseer.

[25] Diksha Moolchandani, Sudhanshu Gupta, Anshul Kumar, and Smruti R Sarangi. 2020. Performance Prediction for Multi-Application Concurrency on GPUs.. In *ISPASS*. 306–315.

[26] Diksha Moolchandani, Anshul Kumar, José F Martínez, and Smruti R Sarangi. 2020. VisSched: An Auction-Based Scheduler for Vision Workloads on Heterogeneous Processors. *IEEE TCAD* 39, 11 (2020), 4252–4265.

[27] Diksha Moolchandani, Anshul Kumar, and Smruti R Sarangi. 2020. Accelerating CNN Inference on ASICs: A survey. *Journal of Systems Architecture* (2020), 101887.

[28] Hitoshi Nagasaka, Naoya Maruyama, Akira Nukada, Toshio Endo, and Satoshi Matsuoka. 2010. Statistical power modeling of GPU kernels using performance counters. In *International conference on green computing*. IEEE, 115–122.

[29] RCR Wireless News. [n.d.]. *Four edge computing market predictions*.

[30] Grand View Research. [n.d.]. *Edge Computing Market Size, Share and Trends Analysis Report By Component*.

[31] Peter J Rousseeuw. 1987. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics* 20 (1987), 53–65.

[32] RTInsights. [n.d.]. *Time to Get Real: 3 Tips for Success with Edge IoT and Streaming Analytics*.

[33] Shuaiwen Song, Chunyi Su, Barry Rountree, and Kirk W Cameron. 2013. A simplified and accurate model of power-performance efficiency on emergent GPU architectures. In *2013 IEEE IPDPS*. IEEE, 673–686.

[34] Yusuke Suzuki. 2018. Making GPUs First-Class Citizen Computing Resources in Multi-Tenant Cloud Environments. (2018).

[35] Sravanthi Kota Venkata, Ikkjin Ahn, Donghwan Jeon, Anshuman Gupta, Christopher Louie, Saturnino Garcia, Serge Belongie, and Michael Bedford Taylor. 2009. SD-VBS: The San Diego vision benchmark suite. In *IISWC*. IEEE.

[36] VXCHNGE. [n.d.]. *5 Edge Computing Use Cases With Huge Potential*.

[37] Qiang Wang and Xiaowen Chu. 2017. GPGPU power estimation with core and memory frequency scaling. *ACM SIGMETRICS Performance Evaluation Review* 45, 2 (2017), 73–78.

[38] Zeyi Wen, Jiashuai Shi, Qinbin Li, Bingsheng He, and Jian Chen. 2018. ThunderSVM: A fast SVM library on GPUs and CPUs. *The Journal of Machine Learning Research* 19, 1 (2018), 797–801.

[39] Gene Wu, Joseph L Greathouse, Alexander Lyashevsky, Nuwan Jayasena, and Derek Chiou. 2015. GPGPU performance and power estimation using machine learning. In *HPCA*. IEEE.

[40] Qiumin Xu, Hyeran Jeon, Keunsoo Kim, Won Woo Ro, and Murali Annavaram. 2016. Warped-slicer: efficient intra-sm slicing through dynamic resource partitioning for gpu multiprogramming. In *2016 ACM/IEEE ISCA*. IEEE, 230–242.

[41] Xin Xu, Na Zhang, Michael Cui, Michael He, and Ridhi Surana. 2019. Characterization and prediction of performance interference on mediated passthrough GPUs for interference-aware scheduler. In *11th Usenix HotCloud 2019*.

[42] Yao Zhang and John D Owens. 2011. A quantitative performance analysis model for GPU architectures. In *2011 IEEE 17th international symposium on high performance computer architecture*. IEEE, 382–393.

[43] Yilei Zhang, Zibin Zheng, and Michael R Lyu. 2012. Real-time performance prediction for cloud components. In *IEEE 15th ISORC*. IEEE.

[44] Keren Zhou, Mark W Krentel, and John Mellor-Crummey. 2020. Tools for top-down performance analysis of GPU-accelerated applications. In *Proceedings of the 34th ACM International Conference on Supercomputing*. 1–12.