RC25487 (WAT1408-021) August 15, 2014
Computer Science

IBM Research Report

Should I Use a GPU? Predicting GPU Performance from CPU Runs

Ioana Baldini, Stephen J. Fink, Erik Altman
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598
USA

Should I Use a GPU? Predicting GPU Performance from CPU Runs

Ioana Baldini Stephen J. Fink Erik Altman

IBM Research

{ioana, sjfink, ealtman}@us.ibm.com

1. Introduction

Over the past decade, graphics processing unit (GPU) platforms have evolved into general purpose programmable accelerators. General purpose GPU (GPGPU) programming languages such as OpenCL and CUDA bring GPU acceleration within reach for a large programming audience.

Despite GPGPU languages, effective GPU programming remains difficult, for all the reasons parallel programming remains difficult. In addition to general parallel programming challenges such as managing locality, communication and synchronization, GPUs present challenges with device-specific optimization and portability. To realize significant performance gains, the GPU programmer must carefully tune for low-level device-specific issues such as branch divergence, thread group resources, and a particular non-uniform memory hierarchy [27]. Not all vendors support the same programming model or API, and the same code can perform much differently on different devices [6].

When considering a code for GPU acceleration, how can the programmer predict whether the port is worth the effort? GPU speedup, compared to an optimized parallel multi-core implementation, varies considerably depending on the application and its inputs [22]. Many anecdotes report successful GPU acceleration stories, but many efforts fail to achieve the desired speedup. While general guidelines describe styles of algorithms that match the GPU architecture, no method can reliably predict speedup for a unique, individual application.

This paper addresses the problem of identifying which parallel loops could benefit from GPU acceleration *without* incurring the effort of porting to the new device. Given a program that runs on a CPU, we present a method to predict the speedup achievable on a GPU, based on machine learning from previous porting exercises. In addition, we demonstrate predictive models that can choose the best device (i.e., obtaining the best performance) for an application given a specified input.

Since GPUs offer a large number of parallel processing elements, one might simply predict that highly parallel applications benefit from GPU acceleration. Unfortunately, as shown later in this paper, parallel many-core performance does not correlate highly with GPU performance.

For example, consider Figure 1, which shows the parallel speedup on a 12-way multi-core system for three applications implemented in OpenMP, varying the number of executing threads. For comparison, the figure also shows the speedup obtained by two high-end GPUs for the same applications using OpenCL.[1] The graph normalizes all data relative to speed of sequential execution (i.e., the OpenMP version running with only one thread). The OpenMP speedup varies across applications, but all three scale fairly well when increasing the number of threads.

[1] Section 3 provides a detailed description of our methodology.

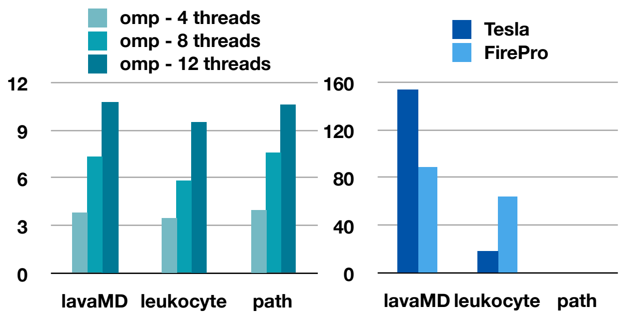

Figure 1. Speedup, compared to one OpenMP thread, for OpenMP applications and corresponding GPU speedup.

Although these applications scale similarly with OpenMP on the multi-core platform, the same applications show significantly different trends on GPUs. Two applications achieve considerable speedup on a GPU; however, speedups vary dramatically depending on the application and the target device. For the third application, the GPU performs 50% and 25% worse, respectively, than sequential execution; hence, the bars are not distinguishable in the graph.

Results presented later in the paper validate that multi-core speedup does not accurately predict GPU speedup. Instead, we must resort to more sophisticated methods, to have any hope of accurate predictions.

Previous approaches for predicting performance on accelerators depend on analytical models [27, 29] and/or idiom recognition from source code [2, 25, 26]. Such approaches rely on domain-specific expertise and detailed architectural knowledge, in order to build accurate analytical models. In addition, every hardware accelerator needs its own model; models do not readily generalize to a class of accelerators. Identifying accelerator-friendly idioms requires user input or incomplete pattern-matching techniques applied to existing code [25].

In general, previous approaches attempt to provide fairly accurate performance prediction, and thus restrict their attention to relatively simple code structures amenable to precise analysis using detailed models. We take a qualitatively different approach: *fully automatic* construction of predictive models that can be applied to any parallel loops, without static analysis of code structure or detailed models of the target architecture. Instead, our approach provides a rough estimate of the expected speedup, learning from a corpus of past exercises in porting other applications to the hardware in question.

This paper presents techniques to apply machine learning to predict GPU performance based on CPU profile data. In particular, we build predictive models using features extracted using dynamic instrumentation running on a general purpose multi-core processor. The results show that a small set of features, derived from dynamic instruction profiles, suffices to train accurate models. These models predict GPU speedup, cast as a binary classification question, with accuracies varying between 77% and 96%, depending on the prediction scenario.

Our main contributions include:

- We show how machine learning classifiers can be used to accurately predict GPU speedup from OpenMP applications. The models do not require static code analysis, nor analytical models for the target device. Model tuning for a new device can be performed automatically, given data gathered from exercises porting to the targeted device.
- We show that the prediction requires small feature vectors based on dynamic instruction profiles collected on a general purpose multi-core processor. Our experiments identify features based on computation instructions, loads, and branches as most significant.
- We demonstrate that similar models can predict the best device for a system with more than one GPU. Such models can guide scheduling in heterogeneous environments.

The next section presents our approach to build predictive models, followed by sections that describe the experimental methodology and results.

2. Predictive Models for GPU Performance

This section describes our approach to build predictive models for GPU performance based on profile data from runs on general purpose multi-core processors.

Figure 2 shows the high-level flow of our approach. As typical in most predictive techniques, our approach entails a *training phase* to calibrate a model, and then a *prediction phase* which applies the model to predict performance for a new application.

The training phase requires a set of existing applications and inputs, along with their observed GPU speedups on the target device. Training requires a version of each code suitable for execution on a traditional CPU, which our tool can instrument to gather dynamic profile information. The training tool runs each application and input on a general purpose CPU system, gathering profile data representing a set of application features.

Next, the tool feeds the application feature profile data and observed GPU speedups into a machine learning toolkit, training a model which predicts GPU speedup based on these features. The tool refines the models via an iterative tuning process, which aims to select a model with a small set of input features, while still achieving an acceptable level of accuracy. The tuning process explores models with more or fewer features, and also considers different machine learning algorithms to build the model.

The training and tuning process produces a predictive model that can be used to forecast GPU performance outcomes for new applications. All aspects of the training and tuning process run fully automatically, without user intervention.

Having produced a predictive model, we can forecast GPU speedup given a CPU-only implementation of a new application, along with representative inputs. The tool runs the new application to collect profile data, extracts model features from the profile data, and predicts the GPU performance via the computed model. Each step in the prediction process runs fully automatically, without user intervention.

The rest of this section discusses the application features and predictive models considered in this study.

Category	Feature	Mnemonic
Computation	Arithmetic and logic instructions	ALU
	SIMD-based instructions	SIMD
Memory	Memory loads	LD
	Memory stores	ST
	Memory fences	FENCE
Control flow	Conditional and unconditional branches	BR
OpenMP	Speedup of 12 threads over sequential execution	OMP
OpenCL	Data transferred to the GPU (in KB)	READ-IN
	Data transferred from the GPU (in KB)	WRITE-OUT
Aggregate	Total number of instructions	TOTAL
	Ratio of computation over memory	ALU-MEM
	Ratio of computation over GPU communication	ALU-COMM

Table 1. The application features used in the predictive models

2.1 Application features

Our methodology employs supervised machine learning to build predictive models, based on a set of *features* which encapsulate relevant characteristics of an application.

A key design question concerns whether to collect features via static analysis, dynamic analysis or both. Some previous works have used static code features to guide compiler optimization tuning (*e.g.*, [5, 9, 10]). We believe that identifying critical code paths and execution counts is crucial for predicting performance on accelerators, and such information is difficult or impossible to gather with static analysis alone. Thus, in this work, we rely on features gathered with *dynamic* analysis.

In particular, we instrument the application to gather dynamic instruction counts, and group instructions into categories of computation as listed in Table 1. We restrict our attention to the categories presented in Table 1, since we expect the performance of GPU code to be dominated by the computation, memory and control flow present in each application. To avoid scale issues, we normalize all instruction-based features to the total number of instructions. We also include the total number of instructions among the features used in our models.

In addition to instruction counts, we consider OpenMP and OpenCL specific features. For OpenMP runs, we include 12-thread speedup over sequential execution (i.e., only one thread) in the feature set[2]. Including the OpenMP speedup as a feature rarely improved the accuracy of the model studied, which provides further evidence that there is a weak correlation between speedup on a multi-core and GPU performance.

Communication costs between the host and device often dominate performance when accelerating a kernel. When training from a sequential or OpenMP program, it can be difficult to automatically infer the communication patterns that would emerge after restructuring and optimizing the program for a GPU offload model. So, when training based on these programs, we cannot rely on any feature to directly capture host-device communication. Instead, we must hope that the trained model adequately accounts for host-device communication costs, learning indirectly from other features derived from the instruction mix.

However, we also consider training based on OpenCL programs running on a general-purpose CPU. In this scenario, the structure of the OpenCL program directly reflects the anticipated host-device communication pattern. In this scenario, we include communication volumes as features for training, as indicated by the *OpenCL* features in Table 1.

We can also build predictive models based on *aggregate* features designed to capture relevant performance characteristics. For

[2] We use 12 threads as the machine used in our experiments has 12 cores. Increasing the number of threads beyond 12 did not increase the performance for the benchmarks under study.

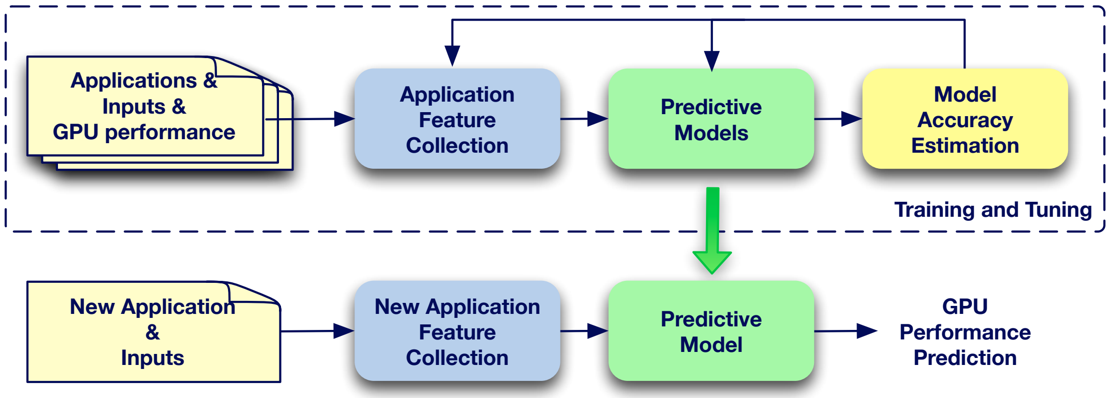
Applications & Inputs & GPU performance
Application Feature Collection
Predictive Models
Model Accuracy Estimation
Training and Tuning
New Application & Inputs
New Application Feature Collection
Predictive Model
GPU Performance Prediction

Figure 2. The high-level diagram for building and using predictive models.

example, we considered the ratio of memory communication versus computation and the ratio of memory loads compared to compute instructions. In a few cases, these aggregate features slightly improve the accuracy of the models.

Our feature set does not directly capture branch divergence issues, which can greatly impact performance on today's GPUs. We investigated building an estimate of branch divergence based on dynamic instrumentation. Since the instrumented code runs on a general purpose processor, the branch instructions in separate threads do not execute in lock-step as they would on a GPU. Instead, to estimate branch divergence effects, we analyzed the outcomes of same branch executed closely in time by separate threads. In our preliminary experiments, these estimates did not significantly improve the accuracy of the predictive models, but considerably increased the feature collection time, since every branch needs to be instrumented. For this reason, we do not include the branch divergence estimate in the feature set for experiments in this paper.

In our experiments, we collect all features via binary instrumentation of user code, using Pin [23]. Since we target GPU performance issues, we restrict the instrumentation to parallel loops in the case of OpenMP code and kernels for OpenCL code.

Binary instrumentation incurs an overhead over original application code, in some rare cases increasing the application runtime by an order of magnitude. We envision using the approach presented in this paper in off-line settings, and, for this reason, we find the instrumentation overhead tolerable. All the training for one device, including choosing among different mathematical models, can be done in less than a day. Once we have trained the model for a certain GPU, predicting the GPU performance for a new, unseen application incurs only the cost of a Pin run for the new application to collect the feature vector use in the prediction.

To use this type of predictive models in runtime contexts (e.g., scheduling in a heterogeneous system), hardware performance counters could be used to collect similar profiles with extremely low overhead, possibly sacrificing accuracy. Detailed investigation of on-line prediction tradeoffs is left for future work.

2.2 Classifiers for GPU performance

When considering porting to a GPU, the key question asks whether the potential GPU speedup justifies the porting effort. To answer this question, it would be useful to know whether a code would likely exhibit a speedup beyond a threshold T if ported to a GPU device.

Such a question can be formulated as a binary classification problem, where the class identifier denotes whether or not the application running with a particular input exhibits a GPU speedup greater than a threshold T.

Binary classification is commonly addressed via *supervised learning*, a type of machine learning. Supervised learning operates on *labeled* data, where each x in the data set has an associated label y. Based on a training set of labeled data, a supervised learning algorithm infers a function to predict the labels y for unseen examples x. Binary classification refers to problems in which the data has only two distinct labels.

In this work, we build binary classifiers using the dynamic application features previously mentioned. In the next section, we briefly described the two supervised learning algorithms used to build the classifiers: nearest neighbor with generalized exemplars (NNGE), and support vector machines.

2.3 Nearest neighbor with generalized exemplars

K-nearest neighbor [17] is one of the simplest classification algorithms. The algorithm decides the class of an unseen exemplar by computing distances between the new exemplar and the labeled data in the training set and selecting the dominant class among the k closest neighbors. When k=*1*, the algorithm chooses the class of the nearest neighbor.

Nearest neighbor with generalized exemplars (NNGE) [24] improves upon the classical nearest neighbor learning algorithm by applying generalization for the elements in the data set. In a nutshell, NNGE generalizes the closest two exemplars belonging to the same class into the hyper-rectangle determined by the two exemplars. NNGE computes distances between an unseen exemplar and the hyper-rectangles in the dataset, as opposed to individual data points. The algorithm tries to generalize new examples to the closest neighbor of the same class, unless it finds a conflict with other examples or hyper-rectangles in the model. If an existing hyper-rectangle conflicts with a new example, the hyper-rectangle is split into multiple hyper-rectangles.

The algorithm uses a weight formula when computing distances. The formula rewards features that contribute to the accurate predictions. As a consequence, NNGE produces a list of features, ordered by their weights. In our experiments, we used this ordered list to experiment with feature subsets.

NNGE can be tuned using different number of tries for generalization. For the data in our experiments, the accuracy of NNGE did not vary significantly with the number of generalization tries, and hence, NNGE was easy to tune.

2.4 Classification with support vector machines

Support vector machines (SVM) is a popular technique used in machine learning for classification and regression analysis. Used as a binary classifier for a set of data points, SVM tries to find a hyperplane that separates the data in such a way that the distances from the nearest points to the hyperplane is maximized; such a hyperplane is usually referred to as the maximum-margin hyperplane.

SVM are commonly used in conjunction with kernel methods [12]. Kernel methods map the initial data set to a higher-dimensional space using kernel functions with special properties that make the computation in the higher-dimensional space tractable. The maximum-margin hyperplane identified by SVM in the higher-dimensional space corresponds to a non-linear surface in the initial space. Thus, kernel SVM [4] find non-linear separation surfaces for data points that are not linearly separable.

Several kernel functions can be used with SVM. In this work, we used the radial basis function (or Gaussian kernel). The resulting model depends on two main parameters: a factor C used in regularization and a parameter *gamma* used by the kernel function. For the data we experimented with, the accuracy of the models were sensitive to *gamma*, while large values for C provided similar accuracy; thus, we tuned only for the *gamma* parameter.

3. Experimental Methodology

In this section we describe the infrastructure and the methodology used in the experiments.

3.1 Hardware infrastructure

All experiments were run on a server machine with a dual-processor Intel® Xeon® CPU X5690. Each processor has six cores running at 3.47GHz, for a total of 12 cores. The server is equipped with two high-end graphics cards: an ATI FirePro™ v9800 and a Nvidia Tesla™ C2050.

3.2 Benchmarks and data set

The experiments include 18 benchmarks from the Parboil 2.0 suite [28] and Rodinia 2.2 suite [3]. Several benchmarks were not included in the study due to compilation issues in our infrastructure or inconsistent results across the OpenCL runs on different devices. For each benchmark, we use 1-3 different inputs (provided in the benchmark suite or generated with the tools included with the benchmark). In total, we examine 48 runs, which represents the data set used to derive the predictive models presented in this paper.

All benchmarks come with OpenMP and OpenCL implementations; the OpenCL implementations appear to have been tuned to achieve good GPU performance. All results measure GPU speedups using the OpenCL implementations. Unless indicated otherwise, the CPU performance and feature collection were performed using the OpenMP program on the Intel multi-core platform. The exceptions are the results in Sections 4.4 and 4.5; those sections rely on CPU performance and feature collection using the OpenCL implementation running on the Intel multi-core platform.

Unless mentioned otherwise in the text, speedups are computed against baseline runs that take full advantage of the 12 cores available in our server (as opposed to sequential execution on a single thread). The 12-core performance provides a more useful reference point when evaluating GPU speedup compared to the multi-core processor. Speedups are computed with respect to end-to-end application wall clock time, including the time to transfer the data to/from the GPU.

3.3 Cross-validation

Cross-validation is a standard technique to estimate how well a statistical model built using a limited data set will generalize to an independent data set. Cross-validation is commonly used to evaluate the accuracy of predictive models, which indicates how well models are expected to perform in practice.

Cross-validation involves several computation rounds. One round of cross-validation splits the data available in two disjoint sets: a training set and a testing/validation set. The training set is used to build the model, while the accuracy of the model is evaluated on the testing set. To reduce variability, multiple rounds of cross-validation are performed using different partitions of the data set.

In this study, we use *leave-one-out* cross-validation. As the name suggests, leave-one-out cross-validation involves using a single element from the original data set as the testing data, and the remaining elements as the training data. Cross-validation is repeated for each element in the data set and the results are aggregated across all rounds.

The results which follow report the accuracy of the predictive model computed using a variant of leave-one-out cross-validation. Since our data set contains several runs for the same benchmark, instead of leaving one run out of the data set, we leave out all runs for a particular benchmark (i.e., *leave-one-benchmark-out*, labeled in the graphs as *one-out*). This corresponds to having no prior information on one particular benchmark. To understand the robustness of the predictive model, we also conducted cross-validation leaving two benchmarks out. For this set of experiments, we randomized the set of benchmarks and performed cross validation leaving all runs from two benchmarks out. To eliminate variations due to pairing benchmarks, we run this type of experiment ten times and aggregate the accuracies over all trials.

3.4 Feature subsets

NNGE uses weights to reward features that contribute to good predictions. When tuning feature sets for predictive models, we considered features for inclusion in the order of their weights provided by NNGE. This method results in a linear search over the features, refining the model by adding one feature at a time. We did not conduct an exhaustive search of all possible combinations of feature subsets, which would be infeasible in reasonable time due to the exponential blow-up in the search space. In addition, we also considered a few feature vectors based on domain specific knowledge, e.g., include memory operations, branch counts and computations as main features expected to determine GPU performance.

In the results, we present the feature subset that yield the highest accuracy. When multiple feature subsets yield the same accuracy, the subset with fewer features is shown.

3.5 Machine learning software

We used the implementations of NNGE and SVM available in Weka 3 [11] for all predictive models studied in this paper. Weka is an open source collection of machine learning algorithms written in Java, which includes tools for data pre-processing, classification, regression, clustering, association rules, and visualization. In our experiments, we did not modify any of the algorithms used; instead, we used the off-the-shelf Weka implementation. We augmented the software only with statistics pertinent to our data set, to help in training and debugging.

4. Experimental Results

The results presented in this section address the following questions:

1. What is the correlation between OpenMP and GPU performance?
2. What is the accuracy of the models for predicting magnitude of GPU speedup from OpenMP runs? Does the accuracy of the

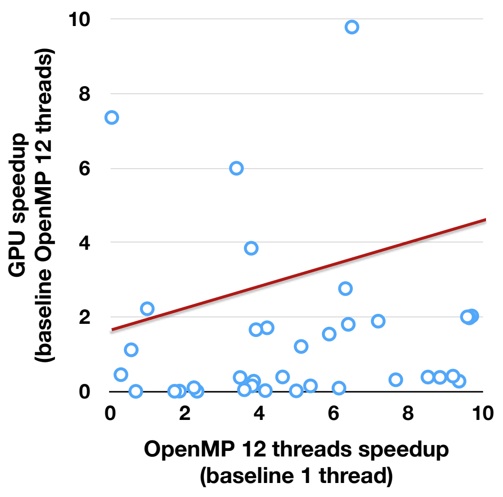

Figure 3. Correlation between OpenMP speedup and GPU speedup on the FirePro card. The red line depicts the best linear fit.

models vary with the GPU device used? How sensitive are the models when trained with less data?

3. Which application features are crucial in the models studied?
4. Can similar predictive models be built from OpenCL runs on multi-cores? If so, might such models be used for scheduling in heterogeneous environments?

Throughout the entire section, GPU speedup is computed using as baseline the performance of OpenMP running with *12 threads*. We are interested to understand how much better the application will run on a GPU compared to utilizing *all* the cores on our system.

4.1 Correlation between OpenMP and GPU speedup

Does the speedup realized by GPU parallelism correlate highly with the speedup observed by a parallel implementation on a conventional multi-core?

To answer this question, for each benchmark/input combination, we compare the speedup of the GPU implementation to the speedup realized by an OpenMP implementation running with 12 threads on the multi-core host.

Figure 3 shows a scatter plot with OpenMP speedup on the *x axis* and GPU speedup on the *y axis* for the FirePro card. The OpenMP speedups are normalized to single-threaded performance; the GPU speedups are normalized to OpenMP performance on the host using all available cores (12 threads).

To make the graph legible, we present only the data points with values lower than 10; but include all data points in the data analysis that follows.

The figure shows the best-fit least squares linear regression line for the speedup data. Considering OpenMP and GPU speedup as two different variables, the correlation coefficient between these variables is 0.146. This is considered a low correlation, and indicates that a predictive model based solely on OpenMP speedup would explain little of the variance in observed values for GPU speedup.

This observation is consistent with the trends presented in Figure 1 in the introduction and motivates investigating different models and including more application features.

The results for the Tesla look similar to the results in Figure 3; the corresponding chart is omitted.

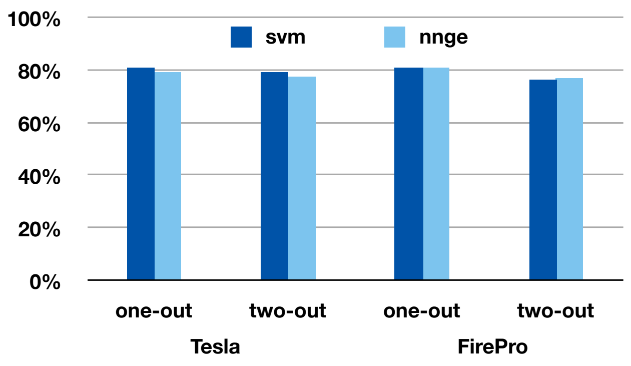

Figure 4. Classifier accuracy for predicting whether GPUs provide any speedup at all. Speedup threshold $T = 1$. GPU speedups are computed relative to OpenMP 12-thread runs on a 12-core CPU.

Overall, the data shows that 48% of the runs achieve a GPU speedup less than one; that is, the 12-thread multi-core implementation runs faster than the GPU implementation. This result may be surprising, since the benchmark suites selected (Parboil and Rodinia) specifically target GPU-friendly algorithms. Overall, the benchmark suite happens to include a good variety of runs for which GPU acceleration is effective, and also many runs for which GPU acceleration is ineffective.

Based on this observation, we believe the data set provides a suitable corpus for further experiments regarding predictive models for GPU acceleration. The following sections report these experiments.

4.2 Learning whether GPU acceleration is beneficial

When considering porting an application to a GPU, a first question to ask is whether the application would benefit from GPU acceleration at all.

To address this question, we build a binary classifier as explained in Section 2.2. This classifier employs a speedup threshold of *one*, which corresponds to the following two classes of runs: those that improve upon OpenMP performance (with 12 threads), and those that do not.

We build two different models, one using support vector machines (in the graphs, for short, *SVM*) and one using nearest neighbor with generalized exemplars (in the graphs, for short, *NNGE*). For each GPU, a different model is trained and tuned.

Figure 4 reports the accuracies achieved by these classifiers using two different methods of cross-validation: *leave-one-benchmark-out*, and *leave-two-benchmarks-out*, as described earlier in Section 3.3.

For each model considered, we vary the feature subsets when tuning the model and present the results for the best subset. Several subsets can exhibit the same accuracies. Table 2 reports the feature vectors selected by tuning for each model. For this set of experiments, *NNGE* uses one or two features more than the *SVM* model.

Overall, the models achieve classifier accuracy approaching 80%. In most experiments, *NNGE* achieves slightly lower accuracy than *SVM*, but it requires considerably less tuning.

For both GPUs used in this study, the models exhibit similar accuracy. The set of features selected is identical for both cards when the model is trained using *SVM*, but differs slightly when building models with *NNGE*.

The models offer high accuracy even when less data is used in training, as indicated by the *leave-two-out* bars in the graph. In gen-

	SVM	NNGE
Tesla	ALU, LD, BR, TOTAL	ALU, LD, ST, ALU-MEM, BR, TOTAL
FirePro	ALU, LD, BR, TOTAL	ALU, LD, BR, TOTAL, OMP

Table 2. The application features used in the classifiers with the speedup threshold *T = 1*. GPU speedups are computed relative to OpenMP 12-thread runs on a 12-core CPU.

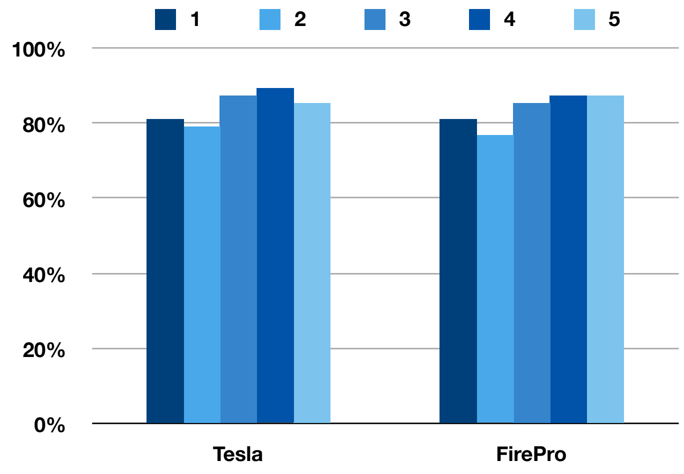

Figure 5. Classifier accuracy for learning magnitude of GPU speedup, when varying the speedup threshold *T* from 1 to 5. GPU speedups are computed relative to OpenMP 12-thread runs on a 12-core CPU.

eral, for all experiments that we conducted, *leave-two-out* achieves accuracy within 1-2% of the *leave-one-out* validation technique.

For the remainder of the paper we focus on results for models trained with *SVM* only, and present results using only *leave-one-out* cross-validation.

4.3 Learning the speedup factor of GPU execution

The previous section evaluated predictive models for whether or not GPU acceleration is effective at all; *i.e.*, classifiers with threshold $T = 1$. Next, we evaluate the accuracy of predictive models for higher speedup thresholds, since in some scenarios only significant acceleration would justify porting costs.

In this section, we present the accuracy of the models trained with *SVM* when the speedup threshold T for the classifiers varies between 1-5. Our data set contains only a few examples that exhibit GPU speedups greater than 5; thus due to sparse data we do not consider thresholds greater than 5.

Figure 5 shows the accuracies of the classifiers for the two GPU cards. The accuracies vary between 77-90% with similar trends for the two cards.

Table 3 lists the features selected via model tuning for these experiments. The card *FirePro* needs more features to achieve higher accuracies. Reducing the number of features usually lead to only a 2-5% drop in accuracy. Across the board, computation and memory reads appear crucial for performance prediction. The number of features considered in the model directly impacts the time it takes to train the model and the time to make a prediction. Usually, the fewer the features, the less time is needed for feature collection, training and prediction.

OpenMP speedup was not selected as a feature to improve the accuracies of the classifiers, except for one case trained for the card *FirePro*, when the speedup threshold is 5. We conclude that

Threshold	Tesla	FirePro
T = 1	ALU, LD, BR, TOTAL	ALU, LD, BR, TOTAL
T = 2	ALU, LD	ALU, LD, BR, TOTAL
T = 3	ALU, LD, BR	ALU, LD, BR, TOTAL
T = 4	ALU, LD	ALU, LD, ST, ALU-MEM, BR, TOTAL
T = 5	ALU, LD	ALU, LD, BR, TOTAL, OMP

Table 3. The application features used in the classifiers when the speedup threshold varies between 1 and 5. GPU speedups are computed relative to OpenMP 12-thread runs on a 12-core CPU.

the instruction count-based features deliver more predictive power in this domain than the OpenMP speedup. This reinforces the hypothesis that multi-core parallel speedup is not a good predictor for GPU parallel speedup.

Aggregate ratios between the features did not show a significant improvement in accuracies. In one case for *FirePro* and speedup threshold 4, including the ratio of computation versus memory operations improved the accuracy by 2%.

The accuracies for *NNGE* show similar trends, with 2-10% lower accuracies. OpenMP speedup appears more often in the best set of features when the model is trained with *NNGE*.

4.4 Predicting GPU performance from OpenCL runs

The previous experiments all gathered features based on OpenMP programs running on a multi-core, to predict performance of OpenCL program running on a GPU. We believe this represents a common usage scenario: when a shared-memory parallel program is available before porting to OpenCL.

OpenMP supports a programming model with parallel loops in an implicitly shared address space, while the OpenCL programming model supports explicit communication between a host and device. In order to maximize performance, the OpenCL programmer must carefully manage explicit copies to avoid unnecessary data transfers and to overlap communication and computation. As a result, the structure of a tuned OpenCL program may differ considerably from the original sequential or OpenMP code. This difference adds extra challenges for our performance prediction methodology, since key features including host-device communication behavior do not appear in the OpenMP program.

However, OpenCL programs can run on multi-core and conventional processors as well as on GPUs. In this section, we ask the question: can predictors deliver better accuracy when training on OpenCL multi-core runs?

If the answer is affirmative, this option might be attractive for some usage cases, such as when a user already has an OpenCL implementation and is considering whether to purchase a particular device for acceleration.

Figure 6 reports accuracy results for classifiers similar to those from the previous section, but using OpenCL runs on multi-cores to train the prediction models. For the remainder of this section, the GPU speedup is computed relative to the run time of the parallel OpenCL program on the multi-core as baseline. All application features are collected from OpenCL runs on the general purpose multi-core.

As shown in the figure, the accuracy of these classifiers varies between 80-96%, higher than previously reported when training using OpenMP runs. This results confirms the hypothesis that the OpenCL multi-core code generates better predictive data, since the code structure on the CPU more closely matches the GPU code.

In OpenCL, it is straightforward to measure the communication volume between host and device, by instrumenting the OpenCL API calls for explicit data transfer. Our experiments include features that measure data transfer to/from the card (*READ-IN/READ-*

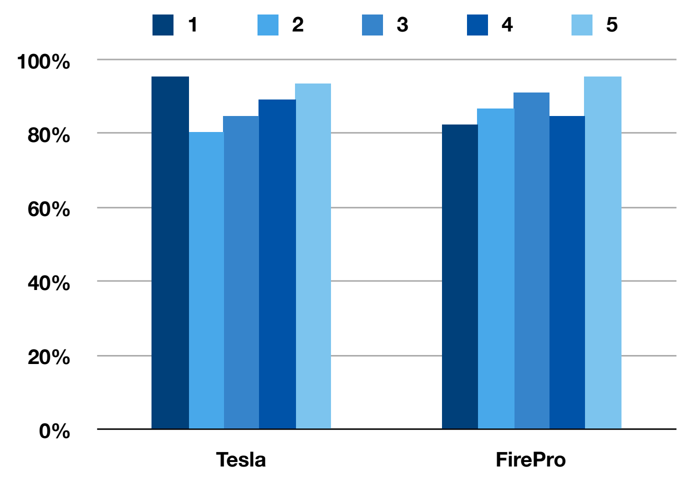

Figure 6. Classifier accuracy for learning magnitude of GPU speedup from OpenCL runs on a CPU, when varying the speedup threshold T from 1 to 5. GPU speedups are computed relative to OpenCL runs on a 12-core CPU.

Threshold	**Tesla**
T = 1	ALU, BR, ALU-MEM, ALU-COMM
T = 2	ALU, LD
T = 3	ALU, LD, BR
T = 4	ALU, LD, BR, TOTAL, READ-IN, WRITE-OUT
T = 5	ALU, LD, BR, TOTAL
Threshold	**FirePro**
T = 1	ALU, BR, ALU-MEM, ALU-COMM
T = 2	ALU, LD, BR, TOTAL
T = 3	ALU, LD, BR, TOTAL, READ-IN, WRITE-OUT
T = 4	ALU, LD, BR, TOTAL
T = 5	ALU, LD, ST, BR, TOTAL

Table 4. The application features used in the classifiers when the speedup threshold varies between 1 and 5. GPU speedups are computed relative to OpenCL runs on a 12-core CPU.

OUT) and aggregates using this information, such as the amount of computation versus communication (*ALU-COMM*).

Table 4 shows the features employed by the models selected by tuning. Several classifiers benefit from including information about the data transferred to and from the GPU card. We conclude that having direct data representing explicit communication costs improves predictor accuracy.

It remains a topic for future work on how to infer communication costs from OpenMP or sequential loops. Possible solutions might entail user annotations and/or static code analysis.

4.5 Predicting best device from OpenCL runs

Computing systems will become more heterogeneous as hardware accelerators diversify and pervade the computational landscape. For such systems, the best device for a particular computation may be non-obvious.

This section evaluates the potential applicability for our GPU performance prediction for scheduling in heterogeneous systems. In particular, we investigate whether predictive models for OpenCL performance could predict the device that provides the best performance, in a system with a CPU multi-core and two different GPUs.

Specifically, we study the accuracy of a three-class classifier based on NNGE. The three classes correspond to *CPU*[3], *Tesla* and *FirePro*. The class for each run in our data set corresponds to the device that obtains the highest performance for that particular run. The classifier learns to predict which device performs the best for a particular run.

As the previous experiments reported, computation and memory operations consistently were chosen in the most accurate models for GPU performance. For this reason, we considered only these two features when building the three-class predictor.

To compute the accuracy of the classifier, we used leave-one-run-out cross validation. For the benchmarks with multiple inputs, this means that the model has available data about the benchmark on a different input set. This assumption is practical for runtime scheduling systems, where same application can be observed multiple times, but not necessarily with the same input. The accuracy of the classifier is 85%.

To better understand how useful such a classifier would be in practice, we present detailed results for the prediction outcome. Figure 7 shows for each benchmark and its runs the performance of the three devices, in the following order *CPU*, *Tesla*, *FirePro*, normalized to the performance of the best device (i.e, the device with the highest performance). For each benchmark run (i.e., the set of three bars with performance for each device), we superimpose the outcome of the predictor. The bar is colored in dark red whenever the predictor mispredicts, and in dark blue when the predictor picks the best device. Whenever the dark super-imposed bar reaches the value of one in the graph, it means that the predictor picked the best device, and its prediction is considered correct.

The classifier mispredicts seven times out of the 46 runs, for an accuracy of approximately 85%. However, for *three* out of the *seven* mispredicted runs, the performance of the predicted device is within 2-5% of the best device, which translates into an effective accuracy of 91%. These results are promising and indicate that scheduling in heterogeneous environments could highly benefit from predictive models.

In a runtime system, collecting application features needs to incur low overhead. We envision using either low-cost dynamic instrumentation through sampling or hardware performance counters to extract application profiles.

5. Discussion

Summary of Results The results in our study show predictive models that achieve accuracies of at least 80%, across a variety of configurations. As this is one of the first study to predict GPU performance, we have no objective basis on which to argue whether or not an 80% accuracy result is "good enough" for use in a production-level tool.

As the first study of this kind, we believe that 80% accuracy represents an encouraging result, suggesting that this methodology remains a viable path for future research. We hope that follow-on work will establish even more effective algorithms and features which increase accuracy, perhaps eventually leading to a production-ready prediction tool.

Threats to Validity The most significant threat to the validity of these results may concern the choice of benchmark suite. Our data arises from the Parboil and Rodinia benchmark suites; both of these benchmark suites specifically target applications suitable for GPU acceleration.

A completely unbiased study might start with a clean sheet of potential applications without any preconceived notions of how they suit GPUs. Unfortunately, that hypothetical study would have

[3] CPU denotes the 12-core general purpose processor used in this study

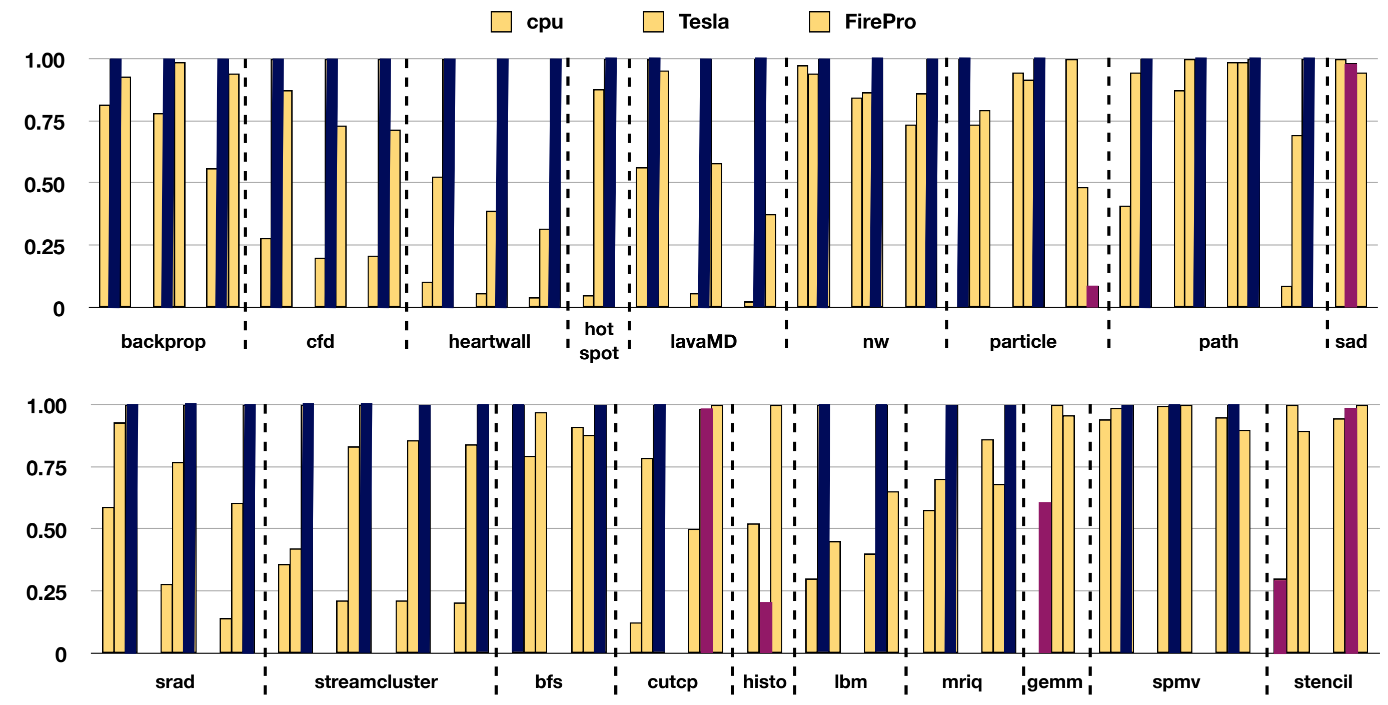

Figure 7. Performance of three devices (CPU, Tesla and FirePro) for OpenCL runs, normalized to the performance of the best device (light yellow bars). For each set of runs (i.e., set of 3 bars), the best device as predicted by an NNGE-based classifier is indicated by a dark bar. Correct predictions are indicated in dark blue, mispredictions in dark red.

to invest an enormous development effort to produce optimized GPU implementations.

We believe the results in this study provide valuable data despite the apparent bias in the data set, for two important reasons.

First, we believe that in the foreseeable future, programmers will *not* consider arbitrary code out-of-the-blue for GPU acceleration, but will rather consider only parallel loops that appear to match the data-parallel GPU architecture. So, we claim that the benchmarks considered represent a reasonable sample of the population of candidates for GPU acceleration.

Second, we note that even for these apparently GPU-friendly benchmark suites, roughly half the runs *failed* to improve performance over the multi-core CPU implementation. Thus, roughly half our data set falls into each of the two classes (speedup vs. slowdown) presented for the first experiment. This even division is advantageous when evaluating a binary classification predictor.

This study used OpenMP implementations as the parallel CPU baseline, since the benchmark suites provided OpenMP versions. Our methodology is not limited to OpenMP applications, and it could be used with any CPU implementation, including sequential code or thread-based parallel versions. Since none of the features collected were OpenMP-specific, we hypothesize that any CPU version of these codes would generate similar dynamic instruction mixes, and thus lead to similar overall results. However, this hypothesis remains to be tested in future work.

Limits of Machine Learning The results show that machine learning can build accurate predictors based on dynamic instruction counts. However, machine learning approaches produce opaque models built on potentially non-linear underlying relationships. The resultant models do not clearly elucidate the mechanisms that underly relationships between features and outcomes.

Standard statistical approaches like linear regression produce simpler models that elucidate relationships, but machine learning more easily accommodates non-linear relationships. Regression approaches can accommodate non-linear relationships when the non-linear terms are made explicit; however, it is difficult to choose appropriate non-linear terms without deep insights into cause-and-effect.

Section 4 reported the features that contributed to model accuracy; most models relied on ALU instructions, load counts, and branch counts. It remains for future work to find analytic models that clearly explain how these features translate into GPU performance metrics. Results from machine learning give hints that these features are important, but machine learning does not immediately produce the desired models that explain cause-and-effect or reasons why the features are relevant.

Note that our experimental data embodies past porting exercises for the target architecture. These exercises capture the effects of human porting decisions; the GPU speedup data depends highly on how a human partitioned, refactored, and optimized the code. It may be extremely difficult to find simple linear models which account for these factors.

It remains for future work to find generally applicable, practical predictive models built directly from simple relationships between code and the architecture. We review some previous efforts in this space in the next section.

6. Related Work

Machine learning techniques have been successfully applied to predict application performance on multi-cores from single-core runs [21], to explore design spaces in computer architecture simulations [14, 20], to optimize phase ordering in compilers [19] or to optimize the execution of different processor components [15]. Jia *et al.*, [16] used regression techniques to predict the results of simulations of GPU architectures.

A few works address the core problem we consider: can one estimate the performance on a GPU *before* investing the effort to (fully) port the code to a new programming model.

Meng *et al.*, [25] present GROPHECY, a tool to predict the GPU performance for loops. In GROPHECY, the user creates a "skeleton" of the code in question: a sketch of the controlling loop logic for the kernel to accelerate. The GROPHECY tool simulates various loop transformations on the skeleton, and predicts the performance for each transformed version using a detailed analytic model [13]. In contrast to GROPHECY, our approach is fully automatic and requires no work from the user, and our approach can handle more general control structures in the outer loops. However, GROPHECY uses a more detailed, precise performance model, which most likely will predict performance more accurately when applicable.

Meswani *et al.*, [26] present a system and methodology to predict performance on accelerators, targeting certain patterns of high-performance computing (HPC) kernels. Their methodology employs a simple pattern-matching static analysis to identify loops which match common *idioms* such as 'gather/scatter' and 'stream'. For each idiom, they calibrate detailed performance models on an accelerator for various input parameters. Then, for each idiom-matching loop in the application, the tool collects dynamic information to trace and simulate the code, to derive parameters for the performance model. Additionally, they describe methods to predict the communication costs for data transfers between the host and accelerator. Compared to our work, this approach is much more detailed and likely accurate; however, it is limited to loops which match a small set of limited idioms. We suspect that few, if any, of the benchmarks considered in this paper fit these idioms.

The Kremlin [8] tool provides a methodology to discover opportunities for parallelization in sequential code, primarily by deriving upper bounds on potential parallelism from a dynamic critical path measurement. This methodology include "parallelism discovery" and "planning", where the latter phase considers architectural details of the target device. Garcia *et al.*, describe a planner for OpenMP implementations on multi-cores, which considers features such as nested parallelism, reductions, and doacross dependencies. One could characterize our work as a "planning personality" for a GPU, built automatically via machine learning. Preliminary results applying the Kremlin methodology for GPUs show promising results identifying which kernels to parallelize [7].

Kim *et al.*, [18] present *Parallel Prophet*, a tool to estimate parallel performance based on instrumented, annotated serial programs. This tool focuses heavily on a memory performance model, predicted parallel speedup in the presence of memory contention due to traffic from parallel threads. This work considers predictions based on both analytic models, and also by executing generated code that can account for system issues such as thread scheduling policies.

Other work focuses on deriving extremely detailed analytic performance models for GPUs. Generally, these models require precise information about the code to run on the device, and so do not immediately apply when predicting performance before porting to the target architecture.

Baghsorkhi *et al.*, [1] present a performance prediction tool based on a rich analytic model, a detailed hardware model, and a symbolic execution engine. This model includes parameters to account for hardware structures such as memory bank conflicts and memory coalescing. This model was designed to support compiler auto-tuning when choosing parameters for loop transformations.

Hong *et al.*, [13] present a detailed GPU performance model, which focuses on the GPU memory access pattern and computational density per memory request. The model is fairly detailed, using 21 parameters, and estimates the memory performance based on the number of outstanding parallel memory requests, coalescing for memory requests, and the number of computational cycles per memory request. This technique requires a fairly detailed model of the code to run on the device. The GROPHECY tool uses this model to predict performance for loop skeletons [25].

Recently, Grewe et al. [9, 10] use machine learning predictive models to help split code between the cores of a CPU and the GPU in a heterogeneous system. Similarly to our technique, the authors choose program features; however, unlike our approach, these studies use program features based on the static analysis of the code, while we use only dynamic features extracted by instrumenting and running the applications on a general purpose processor.

In general, the related works described attempt to provide fairly accurate performance prediction, and thus restrict their attention to relatively simple code structures or employ detailed analytical models. In contrast, we take a qualitatively different approach: our technique is fully automatic and can apply to any parallel loops, and does not require any static analysis of the code structure or any detailed performance model of the target hardware. Instead, our approach provides a rough estimate of the expected payoff based on learning from a corpus of past exercises in porting to the hardware in question. Therefore, in contrast to previous approaches, our methodology is more generally applicable and has a lower barrier to entry, in exchange for less accurate performance predictions.

7. Conclusions

The results of this study demonstrate that machine learning classifiers can be used to predict the magnitude of GPU speedup from runs on a conventional multi-core. The models are derived automatically, learning from past porting experiences to the targeted device. The predictions rely on a small set of features extracted from dynamic instruction profiles collected on a general purpose multi-core. Similar models can be used to predict the best device in a heterogeneous system to guide scheduling of applications to devices.

To our knowledge, this paper presents the first study applying machine learning to automatically predict GPU performance from multi-core runs. We believe the results indicate that this is a promising approach, and would justify further study in this area. We hope that follow-on work will increase accuracy, and shed more light on the fundamental mechanisms that allow accurate prediction.

References

[1] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu, "An adaptive performance modeling tool for GPU architectures," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '10. New York, NY, USA: ACM, 2010, pp. 105–114. [Online]. Available: http://doi.acm.org/10.1145/1693453.1693470

[2] L. Carrington, M. M. Tikir, C. Olschanowsky, M. Laurenzano, J. Peraza, A. Snavely, and S. Poole, "An idiom-finding tool for increasing productivity of accelerators," in *Proceedings of the international conference on Supercomputing*, ser. ICS '11. New York, NY, USA: ACM, 2011, pp. 202–212. [Online]. Available: http://doi.acm.org/10.1145/1995896.1995928

[3] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC '09.

Washington, DC, USA: IEEE Computer Society, 2009, pp. 44–54. [Online]. Available: http://dx.doi.org/10.1109/IISWC.2009.5306797

[4] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, Sep. 1995. [Online]. Available: http://dx.doi.org/10.1023/A:1022627411411

[5] C. Dubach, J. Cavazos, B. Franke, G. Fursin, M. F. O'Boyle, and O. Temam, "Fast compiler optimisation evaluation using code-feature based performance prediction," in *Proceedings of the 4th international conference on Computing frontiers*, ser. CF '07. New York, NY, USA: ACM, 2007, pp. 131–142. [Online]. Available: http://doi.acm.org/10.1145/1242531.1242553

[6] C. Dubach, P. Cheng, R. Rabbah, D. F. Bacon, and S. J. Fink, "Compiling a high-level language for GPUs: (via language support for architectures and compilers)," in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254066

[7] S. Garcia, D. Jeon, C. Louie, and M. Taylor, "The kremlin oracle for sequential code parallelization," *Micro, IEEE*, vol. 32, no. 4, pp. 42–53, 2012.

[8] S. Garcia, D. Jeon, C. M. Louie, and M. B. Taylor, "Kremlin: rethinking and rebooting gprof for the multicore age," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 458–469. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993553

[9] D. Grewe and M. F. O'Boyle, "A static task partitioning approach for heterogeneous systems using opencl," in *CC '11: Proceedings of the 20th International Conference on Compiler Construction*. Springer, 2011.

[10] D. Grewe, Z. Wang, and M. F. O'Boyle, "Portable mapping of data parallel programs to opencl for heterogeneous systems," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, ser. CGO '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/CGO.2013.6494993

[11] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *SIGKDD Explor. Newsl.*, vol. 11, no. 1, pp. 10–18, Nov. 2009. [Online]. Available: http://doi.acm.org/10.1145/1656274.1656278

[12] T. Hofmann, B. Schölkopf, and A. J. Smola, "Kernel methods in machine learning," *The annals of statistics*, pp. 1171–1220, 2008.

[13] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," in *Proceedings of the 36th annual international symposium on Computer architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 152–163. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555775

[14] E. İpek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz, "Efficiently exploring architectural design spaces via predictive modeling," in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS XII. New York, NY, USA: ACM, 2006, pp. 195–206. [Online]. Available: http://doi.acm.org/10.1145/1168857.1168882

[15] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ser. ISCA '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 39–50. [Online]. Available: http://dx.doi.org/10.1109/ISCA.2008.21

[16] W. Jia, K. A. Shaw, and M. Martonosi, "Stargazer: Automated regression-based gpu design space exploration," *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, vol. 0, pp. 2–13, 2012.

[17] D. Kibler and D. W. Aha, "Learning representative exemplars of concepts: an initial case study," in *Proceedings of the Fourth International Workshop on Machine Learning*. San Mateo, CA: Morgan Kaufmann, 1987, pp. 24–30.

[18] M. Kim, P. Kumar, H. Kim, and B. Brett, "Predicting potential speedup of serial code via lightweight profiling and emulations with memory performance model," in *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, 2012, pp. 1318–1329.

[19] S. Kulkarni and J. Cavazos, "Mitigating the compiler optimization phase-ordering problem using machine learning," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, ser. OOPSLA '12. New York, NY, USA: ACM, 2012, pp. 147–162. [Online]. Available: http://doi.acm.org/10.1145/2384616.2384628

[20] B. C. Lee and D. M. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS XII. New York, NY, USA: ACM, 2006, pp. 185–194. [Online]. Available: http://doi.acm.org/10.1145/1168857.1168881

[21] B. C. Lee, J. Collins, H. Wang, and D. Brooks, "Cpr: Composable performance regression for scalable multiprocessor models," in *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 41. Washington, DC, USA: IEEE Computer Society, 2008, pp. 270–281. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2008.4771797

[22] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU," in *Proceedings of the 37th annual international symposium on Computer architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 451–460. [Online]. Available: http://doi.acm.org/10.1145/1815961.1816021

[23] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 190–200. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065034

[24] B. Martin, "Instance-based learning: nearest neighbour with generalisation," Ph.D. dissertation, University of Waikato, 1995.

[25] J. Meng, V. A. Morozov, K. Kumaran, V. Vishwanath, and T. D. Uram, "Grophecy: GPU performance projection from cpu code skeletons," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 14:1–14:11. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063402

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

[27] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc, "A performance analysis framework for identifying potential benefits in GPGPU applications," in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12. New York, NY, USA: ACM, 2012, pp. 11–22. [Online]. Available: http://doi.acm.org/10.1145/2145816.2145819

[28] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W. mei W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," University of Illinois, at Urbana-Champaign, Tech. Rep. IMPACT-12-01, March 2012. [Online]. Available: http://impact.crhc.illinois.edu/Shared/Docs/impact-12-01.parboil.pdf

[29] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for GPU architectures," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*. IEEE, 2011, pp. 382–393.