Exploration of Supervised Machine Learning Techniques for Runtime Selection of CPU vs. GPU Execution in Java Programs

Gloria Y.K. Kim[1], Akihiro Hayashi[2], and Vivek Sarkar[3]

[1] Rice University
gloria.kim@ricealumni.net
ORCiD: 0000-0003-1623-1818
[2] Rice University
ahayashi@rice.edu
ORCiD: 0000-0001-6861-6272
[3] Georgia Institute of Technology
vsarkar@gatech.edu

Abstract. While multi-core CPUs and many-core GPUs are both viable platforms for parallel computing, programming models for them can impose large burdens upon programmers due to their complex and low-level APIs. Since managed languages like Java are designed to be run on multiple platforms, parallel language constructs and APIs such as Java 8 Parallel Stream APIs can enable high-level parallel programming with the promise of performance portability for mainstream ("non-ninja") programmers. To achieve this goal, it is important for the selection of the hardware device to be automated rather than be specified by the programmer, as is done in current programming models. Due to a variety of factors affecting performance, predicting a preferable device for faster performance of individual kernels remains a difficult problem. While a prior approach uses machine learning to address this challenge, there is no comparable study on good supervised machine learning algorithms and good program features to track. In this paper, we explore 1) program features to be extracted by a compiler and 2) various machine learning techniques that improve accuracy in prediction, thereby improving performance. The results show that an appropriate selection of program features and machine learning algorithm can further improve accuracy. In particular, support vector machines (SVMs), logistic regression, and J48 decision tree are found to be reliable techniques for building accurate prediction models from just two, three, or four program features, achieving accuracies of 99.66%, 98.63%, and 98.28% respectively from 5-fold-cross-validation.

Keywords: Java, Runtime, GPU, Performance heuristics, Supervised machine-learning

1 Introduction

Multi-core CPUs and many-core GPUs are both widely used parallel computing platforms but have different advantages and disadvantages that make it difficult to say that one is comprehensively better than the other. In a multi-core CPU, since threads exclusively occupy each core, each core can handle the execution of completely *different* tasks at once. On the other hand, in a many-core GPU, hundreds or thousands of cores can run hundreds or thousands of threads simultaneously. Since each workgroup of threads in each GPU core is executed together, all threads belonging to the same workgroup must be synchronized for coordinated data processing. Because of the differences between the two platforms, parallelism can vary drastically based on the selection of multi-core CPU or many-core GPU for a particular program.

Many prior approaches explore a good mix of productivity advantages and performance benefits from CPUs and GPUs. Many of them are capable of generating both CPU and GPU code from high-level languages. For example, from OpenMP 4.0 onwards [24], GPU platforms are supported by extending OpenMP's high-level parallel abstractions with accelerator programming. Lime [2] and Habanero-Java [8] accept user-specified parallel language constructs and directives. In addition, IBM's Java compiler [14] supports Java 8 parallel stream APIs, which enables programmers to express parallelism in a high level and machine-independent manner in a widely-used industry standard programming language.

Unfortunately, programmers still have to make the important decision of which hardware device to run their programs on. This method not only relies on programmers to understand low-level issues to make a thoughtful selection, but the nature of non-data-driven prediction also does not guarantee that the full capability of the underlying hardware is utilized. In recent work, the possibility of using supervised machine learning to automate the selection of the more optimal hardware device as a capability in IBM's Java 8 just-in-time (JIT) compiler was explored with success [10]. In [10], a set of program features such as the number of arithmetic/memory instructions is extracted at JIT compilation time, and then a binary prediction model that chooses either the CPU or GPU is generated using LIBSVM (a library of support vector machines) after obtaining training data by running different applications with different data sets.

However, to the best of our knowledge, there is no comparable study on 1) good machine learning algorithms and 2) good program features to track. The focus of this paper is to improve the prediction heuristic in accuracy and time overhead by exploring a variety of supervised machine learning techniques and different sets of program features. Finding the ideal set of features and algorithm can allow us to achieve better accuracy, which can also improve overall performance, and using fewer features to achieve high accuracy can reduce overheads of feature extraction and runtime prediction.

We collected 291 samples from 11 Java applications, each containing ten program features to serve as training data. However, instead of using this information as one training data to build a single binary prediction model, we generated

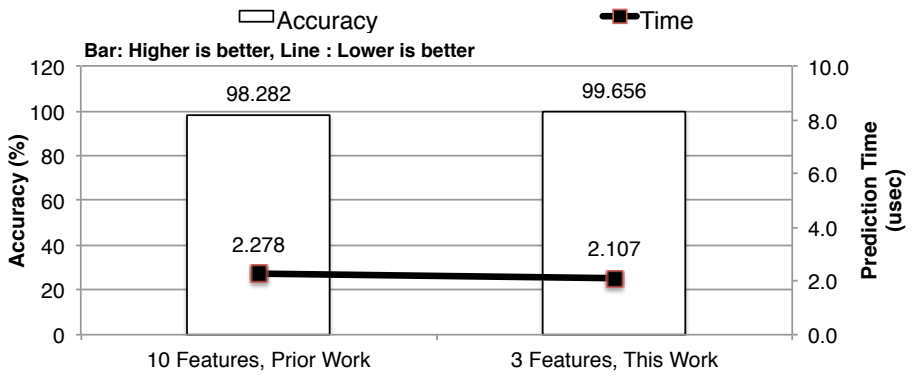

Fig. 1. The impact of changing a set of program features that is fed into LIBSVM on an IBM POWER8 and NVIDIA Tesla platform.

separate training data sets for every possible subset of the original ten features to explore good program features, resulting in a 10^3 order of training data sets. Distinct binary prediction models were trained on these data for every unique supervised machine learning technique: decision stump, J48 decision tree, k nearest neighbors, LIBSVM, logistic regression, multi-layer perceptron, and naive bayes.

This paper makes the following contributions:

- Exploration of supervised machine learning based binary prediction models with various program features for runtime selection of CPU vs. GPU execution.
- Quantitative evaluation of performance heuristics with 5-fold-cross-validation.
- Detailed discussion on implementing prediction models into runtime systems.

The rest of the paper is organized as follows. Section 2 summarizes the background on runtime CPU/GPU selection. Section 3 describes our compilation framework for Java 8 programs. Section 4 discusses how we explore different supervised machine learning algorithms and various program features. Section 5 presents an experimental evaluation. Section 6 discusses related work and Section 7 concludes.

2 Motivation

While optimal hardware selection for heterogeneous platforms is a challenging problem because a wide variety of factors affect performance, it is the programmer's responsibility to select a preferable device even in well-established programming models. We believe that automating the process of runtime CPU/GPU selection can greatly improve productivity and performance portability.

One approach is to build a cost/prediction model to predict a device that could run faster. In this regards, analytical cost models [12, 20] can be very accurate,

Listing 1.1. An example of a parallel stream.

```
1 IntStream.range(0, 100).parallel().forEach(i -> a[i] = i);
```

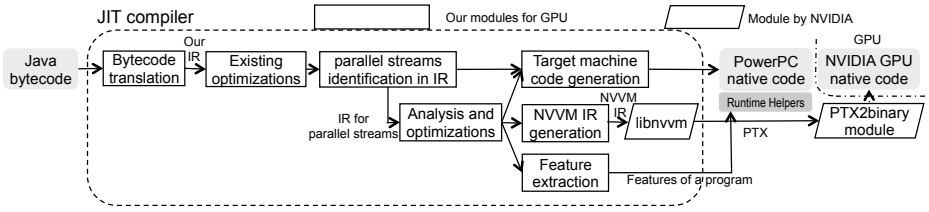

Fig. 2. JIT compiler overview.

assuming deep understandings of target programs and underlying architectures, but are often device-specific. Another direction is to build cost/prediction models empirically [7, 10, 16–19, 29]. Since empirical cost/prediction models are often based on historical performance data, they are platform neutral and can be built without device-specific knowledge.

Prior empirical approaches demonstrated that the use of machine learning algorithms (e.g., linear regression and support vector machines) is a promising way to empirically build prediction models. However, to the best of our knowledge, there is no comparable study on 1) good machine learning algorithms and 2) good program features to use for runtime CPU/GPU selection.

Figure 1 shows the impact of changing a set of program features that is fed into support vector machines (SVMs) on an IBM POWER8 and NVIDIA Tesla platform. Detailed information on the platform can be found in Section 5. The results show that a set of program features can improve the accuracy by 1.374% and the overhead of making a prediction by 8.116% compared to our prior work. Hence, an appropriate selection of program features and a machine learning algorithm can further improve the accuracy and the overheads of prediction models. This motivates us to explore various different machine learning algorithms with different sets of features.

3 Compiling Java to GPUs

This section explains an overview of a parallel loop in Java and compilation of the parallel loop that is used in our framework.

3.1 Java Parallel Stream API

From Java 8 onwards, *Stream* APIs are available for manipulating a sequence of elements. Elements can be passed to a lambda expression to support functional-style operations such as `filter`, `map`, and `reduce`. This sequence can also be used

to express loop parallelisms at a high level. If a programmer explicitly specifies `parallel()` to a stream, a Java runtime can process each element with the lambda expression in this sequence of the stream in parallel.

Listing 1.1 shows an example of a program using a parallel stream. In this case, a sequence of integer elements `i = 0,1,2,...,99` is firstly generated. Then, the sequence is passed with `parallel()` to a lambda parameter `i` in the lambda expression in `forEach()`. The lambda body `a[i] = i` in the lambda expression can be executed with each parameter value in parallel. While this lambda expression can be executed in parallel by multiple threads on CPUs (e.g., by using Java fork/join framework), the specification of the *Stream* API does not explicitly specify any hardware device or runtime framework for parallel execution. This allows the JIT compiler to generate a GPU version of the parallel loop and execute it on GPUs at runtime.

In general, the performance of this parallel execution can be accelerated only when a Java runtime appropriately select one of the available hardware devices.

3.2 JIT Compilation for GPUs

Our framework is built on top of the production version of the IBM Java 8 runtime environment [13] that consists of the J9 Virtual machine and Testarossa JIT compiler [5]. Fig. 2 shows an overview of our JIT compiler.

First, the Java runtime environment identifies a method to be compiled based on runtime profiling information. The JIT compiler transforms Java bytecode of the compilation target method to an intermediate representation (IR), and then applies state-of-the-art optimizations to the IR. The JIT compiler reuses existing optimization modules such as dead code elimination, copy propagation, and partial redundancy elimination.

The JIT compiler looks for a call to the `java.util.Stream.IntStream.forEach()` method with `parallel()` in the IR. If it finds the method call, the IR for a lambda expression in `forEach()` with a pair of lower and upper bounds is extracted. After this extraction, the JIT compiler transforms this parallel `forEach` into a regular loop in the IR. Then, the JIT compiler analyzes the IR and applies optimizations to the parallel loop.

The optimized IR is divided into two parts. One is translated into an NVVM IR [21], which is fed into a GPU code generation library (libNVVM) for GPU execution. Features are extracted from the corresponding IR from this part. The other part is translated into a PowerPC binary, which includes calls to make a decision on selecting a faster device from available devices and to CUDA Drive APIs. The latter includes memory allocation on GPUs, data transfer between the host and the GPU, and a call to GPU binary translator with PTX instructions [22]. When the former call decides to use the GPU, the PowerPC binary calls a CUDA Driver API to compile PTX instructions to an NVIDIA GPU's binary, then the GPU binary is executed.

Currently, the JIT compiler can generate GPU code from the following two styles of an innermost parallel stream code to express data parallelism:

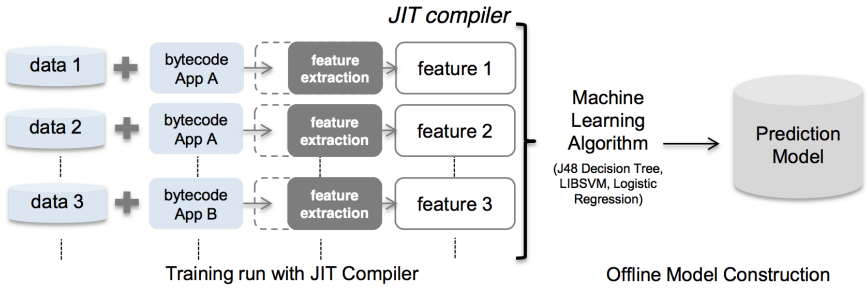

Fig. 3. Supervised machine learning based binary prediction model construction.

Listing 1.2. Supported parallel streams.

```
1 IntStream.range(low, up).parallel().forEach(i -> <lambda>)
2 IntStream.rangeClosed(low, up).parallel().forEach(i -> <lambda>)
```

The function `rangeClosed(low, up)` generates a sequence within the range of $low \leq i \leq up$, where i is an induction variable, up is upper inclusion limit and low is lower inclusion limit. $\langle lambda \rangle$ represents a valid Java lambda body with a lambda parameter i whose input is a sequence of integer values. In the current implementation, only the following constructs are allowed in $\langle lambda \rangle$:

- **types**: all of the Java primitive types
- **variable**: local, parameters, one-dimensional array whose references are a loop invariant, and a field in an instance
- **expression**: all of the Java expressions for Java primitive types
- **statements**: all of the Java statements except all of the following: `try-catch-finally` and `throw`, `synchronized`, a `interface` method call, and other aggregate operations of the stream such as `reduce()`
- **exceptions**: `ArrayIndexOutOfBoundsException`, `NullPointerException`, and `ArithmeticException` (only division by zero)

4 Exploring Supervised Machine Learning Algorithms

This section discusses the supervised machine learning algorithms and various program features explored. We provide descriptions of the algorithms and features, then give an overview of the work flow for constructing models on various subsets of features using each algorithm.

4.1 Supervised Machine Learning

Supervised machine learning is a technique of inferring a function using labeled training data. Typically, *regression* algorithms are used for inferring a real number

Kind	Feature	Description
Size	*range*	Size of parallel loop
Instruction	*arithmetic*	ALU instructions such as addition and multiplication
	branch	Unconditional/conditional branch instructions
	math	Math methods in `java.lang.Math`
	memory	Load and store instructions to and from memory
	other1	Other types of instructions
Array Access	*coalesced*	Aligned array access with zero offset
	offset	Misaligned array access with non-zero offset
	stride	Misaligned array access with stride
	other2	Other types of array accesses

Table 1. Description of Program Features

value (e.g., predicting housing prices) and *classification* algorithms are used for inferring a boolean value (e.g., zero or one). In this paper, we explore classification algorithms to infer a preferable device for a given parallel loop - i.e. CPUs or GPUs.

Figure 3 summarizes our approach to build a binary prediction model. In constructing supervised machine learning based performance heuristics, program features served as the predictors on which the binary prediction models were trained on. Program features were dynamically extracted in the JIT compiler.

4.2 Generating Subsets of Features

Table 1 summarizes program features extracted by the JIT compiler. These features are essentially the dynamic numbers of our IR instructions, which have a strong relationship with execution time. More details on the feature extraction part can be found in our prior work [10]. Every possible subset of features from the union of the sets *{range}*, *{arithmetic, branch, math, memory, other1}* and *{coalesced, offset, stride, other2}* - where the first set with just *range* refers to the number of iterations of a parallel loop, the second set of features refers to the number of such instructions per iteration, and the third set of features refers to the number of such array accesses per parallel loop - was used to create a separate prediction model. Subsets of features (1024 combinations) - as opposed to only the full set of features - were explored to determine which program features more critically contributed to an accurate prediction model.

4.3 Constructing Prediction Models

Algorithms Used Supervised machine learning with each of decision stump, J48 decision tree, k nearest neighbors, LIBSVM, logistic regression, multi-layer perceptron, and naive bayes, was performed to obtain multiple binary prediction models:

- **Decision Stump:** Divide data points into two groups (CPU vs GPU) based on one feature to create the most distinct groups as possible. Results in a

single-level decision tree where the root node represents the feature and the leaf nodes represent either CPU or GPU.
- **J48 Decision Tree:** For each feature, divide data points into two groups (CPU vs GPU) to create the most distinct groups as possible. Results in a multilevel *pruned* decision tree where non-leaf nodes represent features and the leaf nodes represent either CPU or GPU.
- **K-nearest Neighbors:** Map each data point in relation to one another using a distance function. Classify a new data point by assigning it to the class (CPU vs GPU) that is most common amongst its k nearest neighbors.
- **LIBSVM/SVM (Support Vector Machines):** Plot data as points in an n-dimensional space where n is the number of features, then find an optimal hyperplane that splits the data for classification (CPU vs GPU). Prediction on new data is based on which side of the hyperplane it lands when plotted.
- **Logistic Regression:** Predict the probability of an outcome (CPU vs GPU) by fitting data to a logit function. The log odds of binary prediction are modeled as a linear combination of features, which serve as predictor variables.
- **Multi-layer Perceptron:** Build a network of sigmoid nodes and classify instances (CPU vs GPU) using backpropagation.
- **Naive Bayes:** Compute the posterior probability of a class (CPU vs GPU) given features using Bayes' theorem.

These algorithms were deemed appropriate for our context of binary selection (GPU vs CPU) using supervised machine learning (training data includes both predictors as well as the outcomes), and were explored to determine the best algorithm for our model. All prediction models except for those from LIBSVM were built using the Weka software [28] offline. Afterwards, the models were used to make predictions for unseen programs.

Detailed Steps The following steps explain the basic workflow for each model construction:

Step 1: Formatting training data

Training data was formatted for proper processing. In the case of Weka, an ARFF file was formatted as an array of attribute values (features and outcome) for each data sample like such:

⟨value1⟩, ⟨value2⟩, ..., ⟨outcome⟩

where every value is an integer representing the number of times a feature appears in the sample program, and ⟨outcome⟩ is a string ('GPU' or 'CPU') representing which hardware device was the better choice for this particular program.

Step 2: Training

Supervised machine learning was performed with the training data to generate a binary prediction model.

Step 3: Cross Validation

5-fold-cross-validation was used to evaluate the accuracy of the prediction model. In n-fold-cross-validation, data is divided into n sections, then n-1 sections

are used as data to train the model while the remaining section is used as new data to test the accuracy of the model.

Step 4: Additional Testing

The prediction model was used on other testing data to evaluate the accuracy of the model on data unrelated to what was used to build the model.

4.4 Integrating Prediction Models

To integrate a prediction model into the JVM runtime, we first prepare an equivalent C function that takes features and returns a boolean value (CPU or GPU). For example, we put a sequence of if-then-else statements for doing J48 decision tree predictions and put some library calls (e.g., LIBSVM). The JIT compiler generates both CPU and GPU versions of a parallel loop and the runtime selects an appropriate one on the output of the prediction function.

5 Experimental Results

5.1 Experimental Protocol

Purpose: The goal of this experiment is to study how program features and supervised machine learning algorithms affect the accuracy of runtime CPU/GPU selection. For that purpose, we constructed binary prediction models on various subsets of features using the following algorithms: decision stump, J48 decision tree, k nearest neighbors, LIBSVM, logistic regression, multi-layer perceptron, and naive bayes.

Datasets: We used a training dataset from our previous work [10] and an additional dataset obtained on IBM POWER8 and NVIDIA Tesla platform with Ubuntu 14.10 operating system. The platform has two 10-core IBM POWER8 CPUs at 3.69GHz with 256GB of RAM. Each core is capable of running eight SMT threads, resulting in 160-threads per platform. One NVIDIA Tesla K40m GPU at 876MHz with 12GB of global memory is connected over PCI-Express Gen 3. Error-correcting code (ECC) feature was turned off at the time to evaluate this work. The option was either 160 workers on IBM POWER8 or NVIDIA Tesla K40m GPU. The training dataset from [10] was obtained by running the eleven benchmarks shown in Table 2 with different data sets. The additional dataset consisting of 41 samples was obtained by running Bitonic Sort, KMeans, and an IBM's confidential application. In the following, the training dataset from [10] is referred to as *the original dataset* and the additional dataset is referred to as *the unknown testing dataset*. Each sample has the class label showing a faster configuration (160 workers threads on CPU vs. GPU).

5.2 Overall Summary

The binary prediction models were evaluated based on three measures of accuracy: 1) accuracy from 5-fold-cross-validation with the original dataset, 2) accuracy on

Benchmark	Summary	Maximum Data Size	Data Type
Blackscholes	Financial application which calculates the price of European put and call options	4,194,304 virtual options	double
Crypt	Cryptographic application from the Java Grande Benchmarks [15]	Size C with N= 50,000,000	byte
SpMM	Sparse matrix multiplication from the Java Grande Benchmarks [15]	Size C with N = 500,000	double
MRIQ	Three-dimensional medical benchmark from Parboil [25], ported to Java	large size(64×64×64)	float
Gemm	Matrix multiplication: $C = \alpha.A.B + \beta.C$ from PolyBench [26], ported to Java	2,048×2,048	int
Gesummv	Scalar, Vector and Matrix Multiplication from PolyBench [26], ported to Java	2,048×2,048	int
Doitgen	Multiresolution analysis kernel from PolyBench [26], ported to Java	256×256×256	int
Jacobi-1D	1-D Jacobi stencil computation from Polybench [26], ported to Java	N = 4,194,304 T = 1	int
MM	A standard dense matrix multiplication: $C = A.B$	2,048×2,048	double
MT	A standard dense matrix transpose: $B = A^{\mathrm{T}}$	2,048×2,048	double
VA	A standard 1-D vector addition $C = A + B$	4,194,304	double

Table 2. A list of benchmarks used to create the dataset from [10].

prediction of the original dataset, and 3) accuracy on prediction of the unknown testing dataset.

The rest of this section is organized as follows: we first present and discuss accuracies on the full set of features in Section 5.3. Then, in Section 5.4, we present and discuss accuracies on subsets of features. Through deeper analysis and comparison of models, we see how the features included and the algorithm used affect the accuracy of runtime CPU/GPU selection.

5.3 Accuracies on the Full Set of Features

Models trained on the full set of features yielded very different accuracies across different algorithms. The Naive Bayes-trained model performed the worst with a 42.268% accuracy from 5-fold-cross-validation, while LIBSVM and J48 Tree-

	LIBSVM	J48 Tree	Logistic Regression	Multilayer Perceptron	k Nearest Neighbors	Decision Stump	Naive Bayes
Accuracy from 5-fold-CV	98.282%	98.282%	97.595%	88.660%	88.316%	88.316%	42.268%
Accuracy on original training data	99.656%	98.969%	98.969%	96.220%	90.034%	88.316%	42.268%
Accuracy on other testing data	98.282%	92.683%	80.488%	92.683%	92.683%	82.927%	2.439%

Table 3. Accuracies achieved by binary prediction models generated using each ML algorithm from full set of features

LIBSVM	Logistic Regression	J48 Tree	Multilayer Perceptron	k Nearest Neighbors	Naive Bayes	Decision Stump
99.656%	98.625%	98.282%	96.907%	95.876%	91.753%	88.316%

Table 4. Highest 5-fold-cross-validation accuracies achieved by binary prediction models generated using each ML algorithm

trained models performed the best with 98.282%, followed by the Logistic Regression-trained model with 97.595% accuracy. The other models achieved accuracies in the 88.3-88.7% range. A summary of the accuracy results are shown in Table 3 and visually represented in Figure 4.

5.4 Exploring ML Algorithms by Feature Subsetting

Models were also trained on different subsets of features to determine the features that most significantly contribute to high accuracy - the ideal case being the smallest subset of features resulting in the highest accuracy possible. We compared the important features for each algorithm from (1) the subsets achieving highest accuracy and (2) the full set of features. By subsetting, we were able to build models that achieved higher accuracy than those built from the full set of features.

Subsets achieving Highest Accuracy A summary of the accuracy results are shown in Table 4. Here, each of the accuracies refers to the prediction model(s) - based on some subset of features - that achieved the highest accuracy among all other models built using the same algorithm. The result suggests that the top three algorithms are LIBSVM, logistic regression, and J48 tree, all of which achieved highest accuracies of $\geqslant$98.282% from 5-fold-cross-validation.

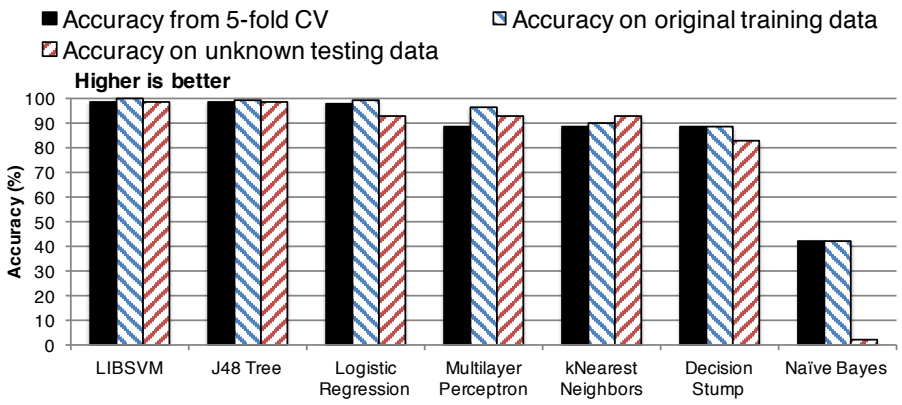

Fig. 4. Accuracies of models with full set of features

99 models built using LIBSVM, 30 models built using logistic regression, and 256 models built using J48 tree achieved highest accuracies of 99.656%, 98.969%, and 98.969% respectively. For each of these top three algorithms, we further analyzed the models based on the smallest subset of features that achieved highest accuracy. For LIBSVM, the smallest subset size was three, for logistic regression four, and for J48 tree two. The results are shown in Table 5 and visually represented in Figure 5.

Comparison of important features and accuracies of models generated using these algorithms led to several key findings. First, the features identified as important were inconsistent across algorithms and slightly differed between subset and full set models for the same algorithm. Second, including more features in the prediction model did not correlate to a higher accuracy.

	LIBSVM	Logistic Regression	J48 Tree
# of features included in model	3	4	2
Accuracy from 5-fold-cross-validation	99.656%	98.625%	98.282%
Accuracy on original training data	99.656%	98.969%	98.282%
Accuracy on other testing data	99.656%	82.927%	92.683%

Table 5. Analysis of binary prediction models with smallest subset of features that achieved highest 5-fold-cross-validation accuracies

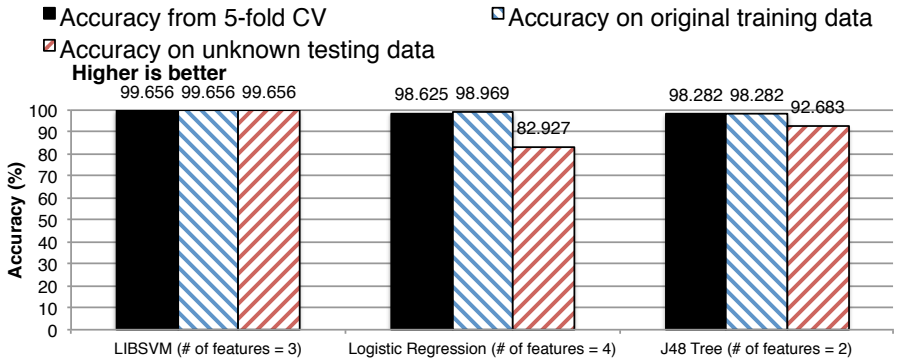

Fig. 5. Accuracies of models with smallest subset size that achieved highest 5-fold-cross-validation accuracies

Comparison of Important Features The models that achieved highest accuracy using LIBSVM, logistic regression, and J48 tree were not built from the same subset of features. To determine which features were important for each algorithm, we analyzed the features of models that achieved the highest accuracy from 5-fold-cross-validation. Although models of various combinations of features achieved the same level of accuracy, for each algorithm, there was at least one feature that was necessarily present in all models. For LIBSVM, this feature is *parallel loop range*; for logistic regression, *coalesced array accesses* and *other array accesses*; for J48 decision tree, *parallel loop range* and *other array accesses*. Table 6, Table 7, and Table 8 detail the number of these models respective to each algorithm that incorporated each feature.

To identify the important features from models trained on the full set of features, the odds ratio of features in logistic regression and the branches representing features in the J48 tree were analyzed. In logistic regression, *offset* was the most important feature; in the J48 tree, *math*, *range*, and *other2* were the most important features. Our analyses indicate that important features identified by subsetting did not exactly match features suggested by the model (e.g., weight vectors) trained on the full set of features. Analyzing the model from LIBSVM was difficult as the model trained on the full set of features produced a (non-)linear hyperplane in a 10-dimensional space.

Comparison of Accuracy As mentioned in Section 5.4, the smallest subset size of the highest accuracy achieved was three for LIBSVM, four for logistic regression, and two for J48 tree. On the other hand, models built from the full set of features at best achieved equivalent accuracy as those built from these subsets. These accuracies are visually represented in Figure 6. A comparison of accuracies between the subset-trained and full-set-trained models suggests that including more features in the prediction model not only fails to improve

Feature	Number of Models with Feature	Percentage of Models with Feature
range	99	100.0%
stride	96	97.0%
arithmetic	65	65.7%
other2	56	56.6%
memory	56	56.6%
offset	55	55.6%
branch	54	54.5%
math	46	46.5%
other1	43	43.4%
coalesced	0	0.0%

Table 6. Feature analysis on highest accuracy (5-fold-cross-validation) subset models built using LIBSVM.

Feature	Number of Models with Feature	Percentage of Models with Feature
other2	30	100.0%
coalesced	30	100.0%
offset	25	83.3%
arithmetic	20	66.7%
stride	18	60.0%
range	16	53.3%
memory	16	53.3%
branch	9	30.0%
math	9	30.0%
other1	6	20.0%

Table 7. Feature analysis on highest accuracy (5-fold-cross-validation) subset models built using Logistic Regression.

accuracy, but in fact decreases the accuracy of the model. This is because as the complexity of a model increases, the risk of overfitting and curse of dimensionality potentially increases. Specifically for LIBSVM, with fewer features (assuming the ideal/appropriate combination of them), the smaller the dimension of space and constraints imposed by each additional dimension as the algorithm searches for an optimal non-linear hyperplane that has the largest separation between two classes.

Analysis of Runtime Prediction Overheads The relationship between subset size and runtime prediction overheads is shown in Table 9. The results show that a smaller subset size can reduce runtime prediction overheads and improve accuracy in general. For J48 decision tree, there was no significant difference in runtime overheads by subsetting because the algorithm does not fully consider all the given features. Also, it is worth noting that each kernel takes at least several

Feature	Number of Models with Feature	Percentage of Models with Feature
other2	256	100.0%
range	256	100.0%
arithmetic	128	50.0%
memory	128	50.0%
branch	128	50.0%
math	128	50.0%
coalesced	128	50.0%
other1	128	50.0%
stride	128	50.0%
offset	128	50.0%

Table 8. Feature analysis on highest accuracy (5-fold-cross-validation) subset models built using J48 Decision Tree.

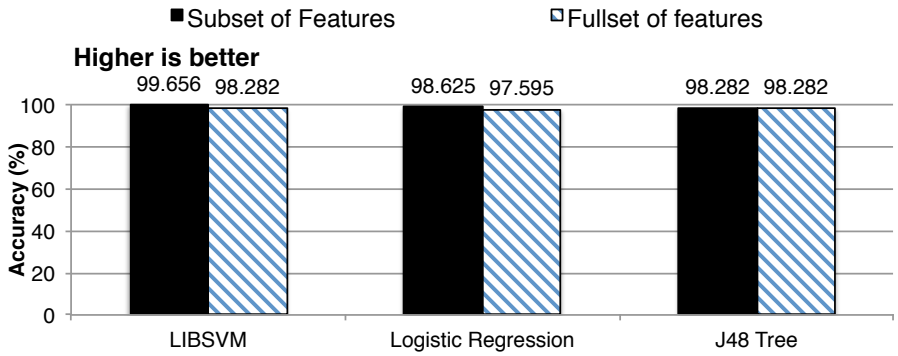

Fig. 6. The impact of feature subsetting

milliseconds, which is several orders of magnitude larger than the prediction overhead shown in Table 9.

5.5 Lessons Learned

Results show that an appropriate selection of program features and machine learning algorithm can improve accuracy and reduce runtime overheads. Based on our analysis, our suggestions for utilizing machine learning techniques for runtime CPU/GPU selection are as follows:

1. LIBSVM, Logistic Regression, and J48 Decision Tree are machine learning techniques that produce models with best accuracies.
2. *Range, coalesced, and other2* are particularly important features. In particular, since *range* is a good metric to measure the amount of work, which in

general significantly correlates to execution time, it is an important feature to incorporate in the prediction models. *range* was present in all models built with LIBSVM or J48 Decision Tree that achieved highest accuracy; *coalesced* was present in all models built with Logistic Regression that achieved highest accuracy; *other2* was present in all models built with Logistic Regression or J48 Decision Tree that achieved highest accuracy. Additionally, *arithmetic* was present in most of the models built with LIBSVM, J48 Decision Tree, or Logistic Regression that achieved highest accuracy.

3. While LIBSVM shows excellent accuracy in prediction, runtime prediction overheads are relatively large compared to other algorithms.
4. J48 Decision Tree shows comparable accuracy to LIBSVM. Also, compared to other approaches, the output of the J48 Decision Tree is more human-readable and fine-tunable because it is a sequence of if-then-else statements.

		LIBSVM	Logistic Regression	J48 Tree
Fullset of Features	nFeatures	10	10	10
	Accuracy from 5-fold-CV	98.282%	97.595%	98.282%
	Prediction Overheads	2.278 us	0.158 us	0.020 us
Subset of Features	nFeatures	3	4	2
	Accuracy from 5-fold-CV	99.656%	98.625%	98.282%
	Prediction Overheads (usec)	2.107 us	0.106 us	0.020 us

Table 9. Overheads of Runtime Prediction

6 Related Work

6.1 GPU Code Generation from High-level Languages

GPU code generation is supported by several JVM-compatible language compilation systems.

Many previous studies support *explicit parallel programming* by programmers on GPU. JCUDA [30] provides a special interface that allows programmers to write Java code that calls user-written CUDA kernels. The JCUDA compiler automatically generates the JNI glue code between the JVM and CUDA runtime by using this interface. Some other tools like JaBEE [31], RootBeer [27], and Aparapi [1] perform runtime generation of CUDA or OpenCL code from a code region within a method declared inside a specific class/interface (e.g. run() method of `Kernel` class/interface).

Other previous work provide higher-level programming models for ease of parallel programming. Hadoop-CL and Hadoop-CL2 [6, 7] are built on top of Aparapi and integrates OpenCL into the Hadoop system. Lime [2] is a Java-compatible language that supports map/reduce operations on CPU/GPU through OpenCL. Firepile [23] translates JVM bytecode from Scala programs to OpenCL

kernels at runtime. HJ-OpenCL [8, 9] generates OpenCL from Habanero-Java language, which provides high-level language constructs such as parallel loop (`forall`), barrier synchronization (`next`), and high-level multi-dimensional array (`ArrayView`). Some other work (e.g. [3, 4]) has proposed the use of high-level array programming models for heterogeneous computing that can also be built on top of the Java 8 parallel stream API.

These approaches leave the burden of selecting the preferred hardware device on the programmer. While they also provide impressive support for making the development of Java programs for GPU execution more productive, these programming models lack the portability and standardization of the Java 8 parallel stream APIs.

6.2 Offline Model Construction

OSCAR [11] is an automatic parallelizing compiler that takes user-provided cost information for heterogeneous scheduling. Some approaches automate this process by constructing performance prediction models offline (e.g., when the JIT compiler is installed on the machine) and making decisions at runtime. For example, Qilin [19] empirically builds a cost model offline for the hybrid execution between CPUs and GPUs. In the context of managed languages like in [18], the runtime predicts absolute performance numbers for CPUs and GPUs with linear models constructed by running micro-benchmarks with different datasets beforehand. Similarly, in [10], the JIT compiler extracts a set of features of a parallel loop (e.g., the number of arithmetic instructions, memory instructions, etc.) at JIT compilation time and the runtime selects the faster device based on a binary prediction model trained on applications with different datasets using support vector machines.

Some of the prior approaches utilize hardware counter information [17, 29] for offline performance model construction, but such information is not usually available.

7 Conclusions

Due to a variety of factors affecting performance, selecting the optimal platform for parallel computing (multi-core CPU vs many-core GPU) for faster performance of individual kernels is a difficult problem. To automate this process and remove the burden from programmers to make the decision between CPU and GPU, we built prediction heuristics from different combinations of program features using a variety of supervised machine learning techniques. Our models achieved accuracies of 99.656% with LIBSVM from three features, 98.625% with logistic regression from four features, and 98.282% with J48 tree from two features. These prediction models can be incorporated into runtime systems to accurately predict, on behalf of the programmers, the more optimal platform to run their parallel programs on, thereby improving performance.

In subsequent work, we plan to increase the training and test data set, then use our prediction-model-based automated selection of CPU vs GPU to compare improvements in runtime performance. In the future, we plan to apply our technique to AOT-compiled programs like OpenMP and OpenACC programs. One challenging problem is how to accurately collect performance metrics since feature extraction is done statically. Another direction of this work is to build prediction models for the systems that have recent GPUs such as Tesla K80, P100, and V100.

A Appendix

- **Backpropagation:** In a Multilayer Perceptron (an artificial neural net), the repeated process of adjusting the weight of each neuron node in order to minimize error.
- **Logit function:** In Logistic Regression, the cumulative distribution function of the logistic distribution.
- **Overfitting:** In machine learning, an undesired occurrence when noise in the training data is learned by the model as concepts, negatively impacting the model's ability to make predictions on new data.
- **Sigmoid node:** In a Multilayer Perceptron (an artificial neural net), a node that is activated based on the Sigmoid function, a special kind of logistic function.

References

1. APARAPI: API for Data Parallel Java (2011), [online] http://code.google.com/p/aparapi/ (Accessed 20 June 2017)
2. Dubach, C., Cheng, P., Rabbah, R., Bacon, D.F., Fink, S.J.: 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. pp. 1–12. PLDI '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2254064.2254066
3. Fumero, J.J., Remmelg, T., Steuwer, M., Dubach, C.: Runtime code generation and data management for heterogeneous computing in java. In: Proceedings of the Principles and Practices of Programming on The Java Platform. pp. 16–26. PPPJ '15, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2807426.2807428
4. Fumero, J.J., Steuwer, M., Dubach, C.: A Composable Array Function Interface for Heterogeneous Computing in Java. In: Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming. pp. 44:44–44:49. ARRAY '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2627373.2627381
5. Grcevski, N., Kielstra, A., Stoodley, K., Stoodley, M., Sundaresan, V.: Javatm just-in-time compiler and virtual machine improvements for server and middleware applications. In: Proceedings of the 3rd Conference on Virtual Machine Research And Technology Symposium - Volume 3. pp. 12–12. VM'04, USENIX Association, Berkeley, CA, USA (2004), http://dl.acm.org/citation.cfm?id=1267242.1267254

6. Grossman, M., Breternitz, M., Sarkar, V.: HadoopCL: MapReduce on Distributed Heterogeneous Platforms Through Seamless Integration of Hadoop and OpenCL. In: Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum. pp. 1918–1927. IPDPSW '13, IEEE Computer Society, Washington, DC, USA (2013), http://dx.doi.org/10.1109/IPDPSW.2013.246
7. Grossman, M., Breternitz, M., Sarkar, V.: Hadoopcl2: Motivating the design of a distributed, heterogeneous programming system with machine-learning applications. IEEE Transactions on Parallel and Distributed Systems 27(3), 762–775 (March 2016)
8. Hayashi, A., Grossman, M., Zhao, J., Shirako, J., Sarkar, V.: Accelerating Habanero-Java Programs with OpenCL Generation. In: Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools. pp. 124–134. PPPJ '13 (2013)
9. Hayashi, A., Grossman, M., Zhao, J., Shirako, J., Sarkar, V.: Speculative execution of parallel programs with precise exception semantics on gpus. In: CaÈŹcaval, C., Montesinos, P. (eds.) Languages and Compilers for Parallel Computing, pp. 342–356. LCPC '13, Springer International Publishing (2014), http://dx.doi.org/10.1007/978-3-319-09967-5_20
10. Hayashi, A., Ishizaki, K., Koblents, G., Sarkar, V.: Machine-learning-based performance heuristics for runtime cpu/gpu selection. In: Proceedings of the Principles and Practices of Programming on The Java Platform. pp. 27–36. PPPJ '15, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2807426.2807429
11. Hayashi, A., Wada, Y., Watanabe, T., Sekiguchi, T., Mase, M., Shirako, J., Kimura, K., Kasahara, H.: Parallelizing Compiler Framework and API for Power Reduction and Software Productivity of Real-Time Heterogeneous Multicores, pp. 184–198. Springer Berlin Heidelberg, Berlin, Heidelberg (2011), http://dx.doi.org/10.1007/978-3-642-19595-2_13
12. Hong, S., Kim, H.: 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. pp. 152–163. ISCA '09, ACM, New York, NY, USA (2009), http://doi.acm.org/10.1145/1555754.1555775
13. IBM Corporation: IBM SDK, Java Technology Edition, Version 8. [online] https://developer.ibm.com/javasdk/downloads/ (Accessed 20 June 2017)(2015)
14. Ishizaki, K., Hayashi, A., Koblents, G., Sarkar, V.: Compiling and optimizing java 8 programs for gpu execution. In: 2015 International Conference on Parallel Architecture and Compilation (PACT). pp. 419–431 (Oct 2015)
15. JGF: The Java Grande Forum benchmark suite. https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/java-grande-benchmark-suite
16. Kaleem, R., Barik, R., Shpeisman, T., Lewis, B.T., Hu, C., Pingali, K.: Adaptive heterogeneous scheduling for integrated gpus. In: Proceedings of the 23rd International Conference on Parallel Architectures and Compilation. pp. 151–162. PACT '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2628071.2628088
17. Karami, A., Mirsoleimani, S.A., Khunjush, F.: A statistical performance prediction model for opencl kernels on nvidia gpus. In: The 17th CSI International Symposium on Computer Architecture Digital Systems (CADS 2013). pp. 15–22 (Oct 2013)
18. Leung, A., Lhoták, O., Lashari, G.: Automatic parallelization for graphics processing units. In: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java. pp. 91–100. PPPJ '09 (2009)

19. Luk, C.K., Hong, S., Kim, H.: Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture. pp. 45–55. MICRO 42, ACM, New York, NY, USA (2009), http://doi.acm.org/10.1145/1669112.1669121
20. Luo, C., Suda, R.: A performance and energy consumption analytical model for gpu. In: 2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing. pp. 658–665 (Dec 2011)
21. NVIDIA: NVVM IR specification 1.3. [online] http://docs.nvidia.com/cuda/pdf/NVVM_IR_Specification.pdf (Accessed 20 June 2017) (2017)
22. NVIDIA: PARALLEL THREAD EXECUTION ISA v5.0 (2017), [online] http://docs.nvidia.com/cuda/pdf/ptx_isa_5.0.pdf (Accessed 20 June 2017)
23. Nystrom, N., White, D., Das, K.: Firepile: Run-time compilation for gpus in scala. SIGPLAN Not. 47(3), 107–116 (Oct 2011), http://doi.acm.org/10.1145/2189751.2047883
24. OpenMP: OpenMP Application Program Interface, version 4.5 (2015), [online] http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf (Accessed 20 June 2017)
25. Parboil: Parboil benchmarks. http://impact.crhc.illinois.edu/parboil/parboil.aspx
26. PolyBench: The polyhedral benchmark suite. http://www.cse.ohio-state.edu/~pouchet/software/polybench
27. Pratt-Szeliga, P., Fawcett, J., Welch, R.: Rootbeer: Seamlessly Using GPUs from Java. In: 14th IEEE International Conference on High Performance Computing and Communication & 9th IEEE International Conference on Embedded Software and Systems, HPCC-ICESS 2012, Liverpool, United Kingdom, June 25-27, 2012. pp. 375–380. HPCC-ICESS '12 (June 2012)
28. at the University of Waikato, M.L.G.: Weka3: data mining software in java. [online] http://www.cs.waikato.ac.nz/ml/weka/ (Accessed 20 June 2017) (2017)
29. Wu, G., Greathouse, J.L., Lyashevsky, A., Jayasena, N., Chiou, D.: Gpgpu performance and power estimation using machine learning. In: 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA). pp. 564–576 (Feb 2015)
30. Yan, Y., Grossman, M., Sarkar, V.: Jcuda: A programmer-friendly interface for accelerating java programs with cuda. In: Proceedings of the 15th International Euro-Par Conference on Parallel Processing. pp. 887–899. Euro-Par '09, Springer-Verlag, Berlin, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-03869-3_82
31. Zaremba, W., Lin, Y., Grover, V.: JaBEE: Framework for Object-oriented Java Bytecode Compilation and Execution on Graphics Processor Units. In: Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units. pp. 74–83. GPGPU-5, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2159430.2159439

All links were last followed on December 9, 2017.