# TOWARDS A FRAMEWORK FOR AUTOMATING THE WORKFLOW FOR BUILDING MACHINE LEARNING BASED PERFORMANCE TUNING MODELS

by

Biplab Kumar Saha

A thesis submitted to the Graduate College of
Texas State University in partial fulfillment
of the requirements for the degree of
Master of Science
with a Major in Computer Science
August 2016

Committee Members:

Apan Qasem, Chair

Michael Ekstrand

Vangelis Metsis

## FAIR USE AND AUTHOR'S PERMISSION STATEMENT

### Fair Use



### Duplication Permission

## DEDICATION

I have been exceptionally supported by my Father. It was not possible to pursue higher studies in the USA without his exceptional support. This thesis is my greatest achievement to date, and I dedicate this thesis to every individual who supported me to reach at this stage.

## ACKNOWLEDGEMENTS

This thesis would not have been possible without my advisor Dr. Apan Qasem. He was very passionate in the every step of the thesis. His step by step guidance and suggestion supported me a lot to finish the thesis. I would like to especially thank him for his relentless support and guidance.

I would also like to thank Dr. Michael Ekstrand and Dr. Vangelis Metsis to support me throughout the thesis whenever I required for any suggestion.

## TABLE OF CONTENTS

**Page**

## LIST OF TABLES

**Table** **Page**

## LIST OF FIGURES

**Figure** **Page**

## ABSTRACT

Recent interest in machine learning-based methods have produced many sophisticated models for performance modeling and optimization. These models tend to be sensitive to architectural parameters and are most effective when trained on the target platform. Training of these models, however, is a fairly involved process and requires knowledge of statistics and machine learning that the end-users of such models may not possess. This paper presents a framework for automatically generating machine learning-based performance models. Leveraging existing open-source software, we provide a tool-chain that provides automated mechanisms for sample generation, dynamic feature extraction, feature selection, data labeling, validation and model selection. We describe the design of the framework and demonstrate its effectiveness by developing a learning heuristic for register allocation of GPU kernels. The results show the newly created models are accurate and can predict register caps that lead to substantial improvements in execution time without incurring a penalty in power consumption.

# I. INTRODUCTION

Machine learning has emerged as an effective strategy for performance modeling and tuning. In this approach, a supervised learning algorithm is trained to learn the complex relationship between a program and its execution environment. This learning is then used to guide the application of an optimization or the allocation of a resource to improve a target objective, such as execution time or power consumption. Many sophisticated predictive models have been developed, spanning the domains of HPC (Stock et al., 2012), data centers (Liao et al., 2009), desktop (Cavazos et al., 2007) and mobile computing (Ge and Qiu, 2011). Two recent trends suggest that the area of machine learning based performance modeling and tuning (MLMT)[1] will grow in importance in the near future.

1. The availability of large code bases in open software repositories such as GitHub.

2. The increased number of exposed hardware performance counters on newer processor architectures.

Both imply greater access to pertinent data, creating new opportunities for learning application behavior on future architectures.

Inspite of its success and promise, two key limitations can be identified in current work in MLMT

1. *Lack of portability:* The state-of-the-practice maintains that learning algorithms be trained on the developer platform and the pre-built models be embedded within a software tool, such as a compiler (Fursin et al., 2011) or an autotuner (Ding et al., 2015), before being shipped to the end-user. This practice is adopted for two reasons. First, model training is a time

[1] Although the area is abbreviated MLMT it includes ML applied to resource allocation, compiler optimizations and other

consuming process and performing the task at the factory relieves the user of this burden. Second, training requires knowledge of machine learning and statistics which the practitioners (e.g., programmers,performance engineers) may lack, making it difficult for them to carry out this task in an error-free manner. This practice imposes an inherent limitation on the models' predictive capabilities, Since program behavior is intimately tied to characteristics of the target architecture, model accuracy is highly sensitive to variations in parameters of the underlying platform. Even a small change in the processor configuration, such as the number of available P states, can render a model ineffective. This issue is further magnified with the growing scale and heterogeneity of HPC architectures. Thus, it is imperative that the learning occur in the target environment.

2. *The black-box treatment:* Developed models have mostly been *predictive* rather than *descriptive*. In the few instances, where a descriptive model has been used, its descriptive properties have not been exploited. Models by-and-large have been treated as black-boxes. A consequence of this approach is that we have gained little insight about application behavior from the many excellent heuristics that have been developed. Another indirect impact is that this has prevented ML-based techniques from being adopted more widely as practitioners are often resistant to using a tool that they do not understand well.

This paper describes the design and implementation of a modular, extensible software framework that addresses the above issues. The framework consists of a language interface for MLMT specification, a performance database and software tool-chain for automating the major steps in an ML workflow. In addition, plugins allow integration of open-source ML algorithm libraries. Given the description of a desired learning outcome, the system can automatically generates a supervised classifier for a new target platform. All major steps in an ML workflow are automated, including feature extraction, feature selection,

training data generation, labeling, validation, hyper-parameter tuning and model selection. We design the system around a set of abstractions that effectively hide the complexities of the ML workflow and have a natural correspondence to entities in performance modeling and tuning. These abstractions are developed based on the key observation that although the ML workflow is extensive, many commonalities can be found when developing performance-related models. For instance, a requirement in MLMT is that feature values be comparable across two different program instances. One way to achieve this is to scale each feature value with respect to the execution time of the program in favor of a standard normalization. Similar commonalities can be found in data clean-up, feature selection and other tasks.

In our system, training data is generated on the target platform. This produces a model customized for a specific architecture and execution environment which implicitly addresses the portability concern. To address the issue of opaque models, we incorporate in the framework analyses to expose the internals of the learning algorithm. The techniques are specifically customized for learning that involves performance-related ideas. Among the technique implemented are clustering, PCA, varimax rotation and decision tree analysis. We also construct context-specific *meta* features to make model outcome intuitive to practitioners. The tool-chain has been used to develop heuristics for compiler optimization, hardware prefetching, thread mapping and migration, and DVFS. In this paper, we demonstrate its utility by deriving a heuristic for allocation of registers for GPU code. Based on the runtime behavior of a kernel, the model recommends the number of registers that should be allocated to it. We analyze the learned heuristic to understand the reasoning behind the recommendations.

To summarize, the main contributions of this paper are as follows:

- A tool-chain for automating ML workflow and generating platform-specific performance heuristics
- Analysis techniques to make learned heuristics more insightful to

programmers

- The first ML heuristic for determining the number of registers to be allocated to a GPU kernel

## II. BACKGROUND

In this section, we provide background in Machine-Learning based performance Modeling and Tuning. (MLMT) techniques[1]

A study of the application of machine-learning techniques in performance modeling and tuning during the recent years in the field of HPC shows a pattern of facing challenges and how ML practitioners have tackled these challenges. The initial application of MLMT emerged as a response to prohibitively long tuning times for search-based autotuning. As such, some of the earliest work in this area were aimed at reducing the parameter space and finding early stopping criteria (Vuduc et al., 2004). Soon after, different ML techniques were devised to adjudicate whether a given optimization should be applied or not. (Cavazos et al., 2007) led the charge in this venture beginning with their work on identifying optimal compiler optimization sequences using multiple logistic regression models. The idea of predicting whether an optimization is beneficial or not is a reduction of the larger problem of finding an optimal set of parameters, and this idea worked well for a multitude of scenarios. However, as the number of optimization available remains large, the number of classifiers required to predict an optimization sequence also remains large. For example, GCC 4.8.2 has 193 optimizations and choosing an optimal sequence essentially means creating an array of 193 classifiers. Furthermore, the widely changing architectures in HPC landscape posed the challenge of adaptability. Fursin *et al.* turned to crowdsourcing to address this challenge by gathering collective optimization knowledge across multiple architectures.

Similar to many ML problems, success of ML techniques hinges on accurate input characterization. Researchers have attempted to characterize programs using program control flow graph, static program features, and hardware

[1] although the area is abbreviated MLMT it includes ML applied to resource allocation,compiler optimization's and others

performance counter values. Hardware performance counters have the added benefit of being dynamic and able to capture architecture-specific system response. However, there are large in number and it is difficult to pick effective ones. Many have resorted to hand-picking them, while some have employed statistical methods to select most varying counters Rahman et al. (2015).

In spite of challenges faced by researchers in applying ML to HPC, the evolution of ML in HPC has been astounding, primarily because of the success obtained from it. Kashnikov *et al.* used four different ML algorithms to select compiler optimization's flags for HPC kernels and compared this strategy with mainstream compilers. Experimental results show that on 38% of cases, the ML models provide better results than applying `-O3` compiler option on GCC (Kashnikov et al., 2012).

The target objective in these ML-based applications have been diverse, too. For example, in order to predict optimal loop unroll factors, Monsiftro *et al.* have used decision trees and Stephenson and Amara singhe have used support vector machines a nearest neighbor classifier. Their models are successful in 80% and 65% cases. Liao *et al.* develop a supervised learning model to choose among sixteen different hardware prefetch configurations in the context of a data center (Liao et al., 2009).

As can be seen, the use of MLMT in HPC has garnered much success and also resulted in a broad spectrum of applications. While this emerging landscape is exciting and full of potential, it is also difficult to navigate for non domain experts. There is a vacuum for a generalist tool chain or approach to HPC problems and this has motivated us to explore this avenue of research.

## III. WORKFLOW COMPARISON

Fig. III.1 outlines a typical ML workflow that may be used in scientific or social studies. The unique aspects of the MLMT workflow and degree of manual input required and currently practiced in different steps are also indicated in the figure. We elaborate on these distinctions next.

*Training data collection* In most domains, data collection does not play a major role in the process of building a model. The data is either already available in some form (e.g., social network data) or handled in a separate and different phase (e.g., genome sequencing). In MLMT, training data needs to be explicitly generated for every new model that is to be created. Training data for an optimization $X$, is unlikely to be useful another optimization $Y$. Developing a database of performance data is problematic, as it will need to be updated for each new architectural model. Training data generation is not only the most time consuming step in the MLMT workflow but also requires significant manual involvement. On the other hand, because data needs to be collected explicitly in many cases there is insufficient data or the quality of the data is poor.

*Clean-up and Processing* Standard scaling and normalization, based solely on the values present in the training data are ineffective for MLMT. Context-aware scaling and normalization algorithms need to be developed. Ideally, scaling should be done not based on the magnitude or range of a an attribute but on

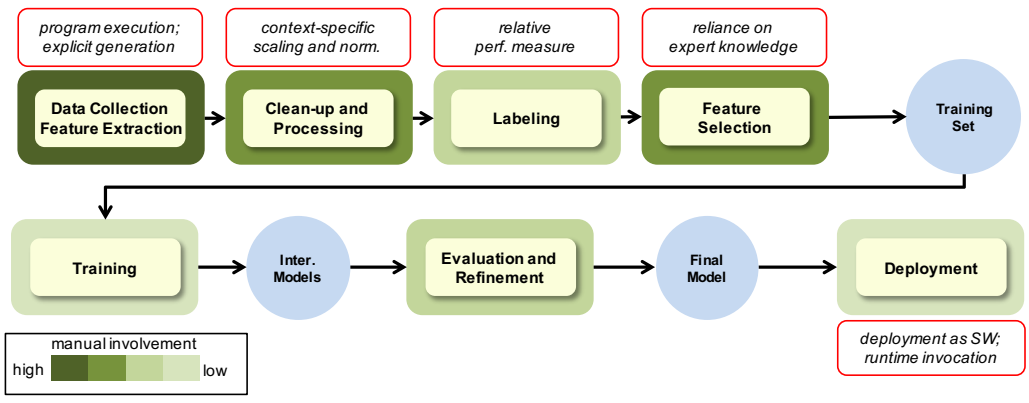


Figure III.1: Generic ML and MLMT workflow comparison

how it affects the performance. For instance, an LLC miss should carry higher weight than an L1 miss. Normalization should generally be done with respect to the execution time to obtain attribute values that can be compared across different programs.

*Labeling* An advantage in MLMT over other ML workflow is that domain expertise is not required to label instances. All that is required to measure relative performance of the un-optimized and optimized versions of workloads. The only exception are cases where ML is being used to classify bottlenecks. In those situations an expert will need to label the instances based on knowledge of the workload being executed.

*Feature selection* Standard practice in most domains is to perform feature selection with the help of domain experts either manually or semi-automatically. This practice is problematic for MLMT because in most cases what is needed is a committee of experts, including architects, systems engineers, compiler writers, programmers and algorithm developers. There is evidence that focusing on attributes from a particular layer can lead to omission of critical features (Leather et al., 2014).

*Deployment* MLMT models are typically deployed as software, either standalone or embedded inside a performance-enhancement tool. Thus, the models operate in a dynamic environment and must make decision at runtime. This implies that model invocation must have very little overhead and the relevant features must be extractable from the target environment.

## IV. MLMT ABSTRACTIONS

To build an automated system, we need to establish a set of abstractions that (i) capture essential elements of a generic ML workflow; (ii) effectively hide complexities in MLMT that would otherwise prevent automation (e.g., divergence in objective metrics) and (iii) are relatable to practitioners (e.g., representation of programs). In this section, we describe these abstractions that serve as the foundation of the proposed software tool-chain. We discuss the rationale behind their construction and outline the terminology and notation used for these abstractions in the remainder of the paper.

**Decision** This is the final desired outcome of the learning model. A *decision* $d$ is a recommended action about a code transformation, a transformation parameter or a resource allocation. Multiple decisions can be combined to create a composite decision and is denoted, $D = \{d_0, ..., d_n\}$. For instance, predicting a compiler optimization sequence involves composing a series of atomic decisions involving the application of an individual optimization.

**Feature** A *feature* $f$ is a source-level, assembly-level or runtime attribute of a code *variant*. A runtime attribute is one that can be measured or estimated via hardware performance counters. All features are numeric.
$fv = \{f_0, ..., f_n\}$ denotes a feature vector.

**Variant** A *variant* $v$ is a multi-program workload, a single application, an accelerator kernel or an extracted code fragment (e.g., a loop-nest). $v$ can be in either binary or source form and is represented solely in terms of a feature vector. $d(v) \rightarrow v_d$ denotes an application of $d$ to $v$. An application of $d$ means executing $v$ when $d$ is taken. Applying a sequence of $k$ decisions produces a new variant, denoted as $D(v) \rightarrow v_D$, where $D = \{d_0, ..., d_k\}$.

**Environment** The execution platform in which $v$ is executed is referred to as the environment $E$. The environment consists of architectural, compilation and system parameters. These values are not included in $fv$ but implicitly incorporated into the model by generating training data and creating models for each $E$ separately.

**Target** a target $T$, is an objective metric such as throughput or energy and must be readily measurable in the execution environment. Targets can incorporate multiple objectives in which case a *pareto-normal* model is considered.

Based on the above abstractions, the goal of an MLMT model is to learn how code variants, described by feature vectors, behave in an execution environment with respect to a specific target and use this learning to take a decision that maximizes (or minimizes) the target for a new and unseen variant. To achieve this goal, the model needs to learn what happens to $v$ with respect to $T$ when $d$ is applied. To provide this knowledge, we construct training data with instances of the form $I = \{f_0, f_1, ..., f_n, L\}$, where $\{f_o, f_1, ..., f_n\}$ are feature values collected for some $v$ in $E$ when $D$ is *not* applied and $L$ is a *label* that captures the effects on $t$ when $D$ is applied to $v$ in the same execution environment. In the simplest case, $L$ can be derived by taking the ratio of $t$ with respect to the two executions of $v$. A ratio of $> 1$ implies a positive effect while $< 1$ implies a negative effect. Thus, the model construction and invocation can be summarized as follows

$$TRAIN(\{I\}) \rightarrow M_T^E; \quad M_T^E(v = fv) \rightarrow \{D\} \tag{IV.1}$$

Given the above formulation, we observe that it is possible to develop a model automatically as long necessary information is provided with respect to generating the training instances. In Section V, we explain what information is necessary and how it is processed by our framework.

# V. DESIGN AND IMPLEMENTATION

Fig. V.1 gives an overview of our framework. Starts with a specification of an MLMT model. Based on this specification a set of scripts are generated customized for the execution environment for which the model is to be developed. These scripts drive the tasks of feature extraction, feature selection, training data generation, model training, evaluation, selection. The newly created is then presented to the user to be invoked on unseen programs. All of these steps can be done one go or they can be done separately (i.e., only generate training data). We highlight the key modules in the framework next.

## Model Specification

We developed a simple language interface to allow users to fully describe an abstract MLMT model, as defined in Section IV. An MLMT specification is comprised of two sections: (i) a set of model parameters and (ii) a sequence of *action blocks*. Parameter values control different aspects of model construction. Each action block contains a pair of action statements. Each statement describes a sequence of actions that need to be taken to apply a *decision* to a *variant*. Actions can be *build* or *execute* commands. A build command denotes that a decision is taken at some stage prior to execution (e.g., source-to-source transformation) while an execute command denotes that the decision is taken at

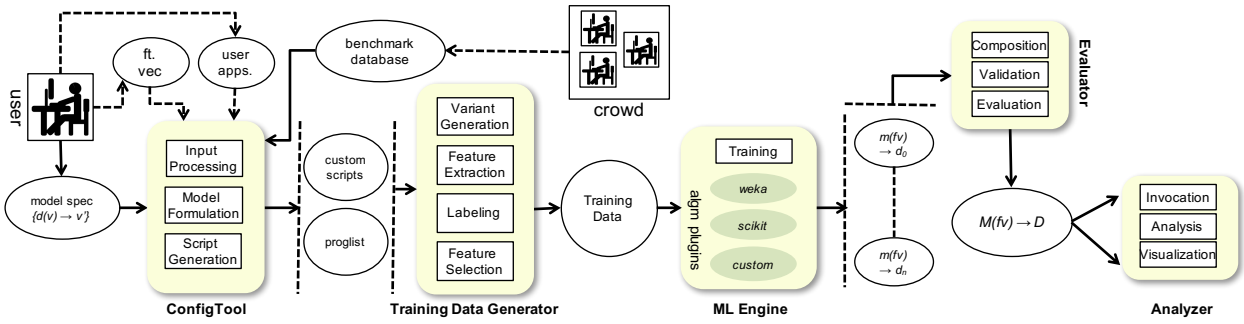


Figure V.1: MLMT framework overview

runtime (e.g., resource allocation).

A *build* can be a series of compilation and link command, a make command or a script. An *execute* command is the command used to execute a *variant* or a script. Each action block must contain a *trivial* action that shows the absence of a decision. One action block must be specified for each elementary decision in the final outcome.

```
MLMTSPEC          :   <info> <instructions>
<info>            :   user training program location
                  |   training time limit
                  |   other
<instructions>    |   <instruction>
                  |   <instruction> <instructions>
<instruction>     |   <meta> ;; <action> ;; <action>
<meta>            :
                  |   build
                  |   exec
<action>          :   compilation command
                  |   make command
                  |   shell script
```

```
user training programs
path to training programs
time to train
;; make -f make.no.decision v; make -f make.decision v
```

The configurer parses the specification file and generates a set of scripts for training data generation. For each action block, the tool determines the difference between the trivial action and applied actions in each action block and uses this information to generate build and execute scripts including feature selection, training and validation.

```
For each action block do
   for each program in database
      generate makefile(trivial, applied action)
      build <- generate build command
      exec <- generate exec command
```

The configurer parses the specification file and parameterizes the execution of each task marked with a green rectangle in Fig. III.1. For instance, if the user specifies

1. `feature extraction [ model parameters: user-supplied, generic, combination ]`
2. `feature selection [ aggressiveness ] model`
3. `training data generation [ generate proglist - build ;; execute training data generation]`

Build scripts are created for each program in the benchmark database. Makefiles for user-supplied training programs are also adjusted.

### Training Data Generation

Instructions for generating training data is supplied to the system via a file called *proglist*. Proglist follows a simple syntax where each line takes the following form

```
<meta_data> ;; <build> ;; <execute>
```

`meta data` and `<build>` are optional. `meta data` contains information about the specific model being trained. `<build>` are the build instructions for a workload $w$, which can be a makefile, a sequence of compilation directives or a shell script that encapsulates all build instructions. The <execute> is a set of commands for invoking $w$, which can be a shell script or a direct invocation command. <build> or <execute> may contain instructions for generating alternate variants $w'$. For instance, a compiler optimization flag may be embedded in the makefile or a resource allocation scheme may be Incorporated in <execute>. Our system executes each proglist command, collects feature values and relevant *targets*. The data is split into different sets based on the <meta data> that is supplied.
A requirement is that the training programs follow the `train/benchmarks/prog/src` structure as in the Parboil benchmark suite. execute scripts are directly inserted into the training data script (next section). To create more extensive data sets the framework includes an autotuner. The tuning capabilities are not used. Rather the autotuner is only used to generate

different code variants. The tuning framework is able to expose control knobs from source code and compilation flags, giving it the ability to create many different variants from the same program.

The framework comes with a pre-defined proglist for some common and important decisions, including (i) GCC optimization sequence (ii) nvcc optimization sequence (iii) GPU register allocation (iv) GPU thread configuration (v) CPU thread affinity (vi) DVFS (vii) tiling and loop interchange and (viii) function inlining. These pre-defined proglists operate on standard benchmarks (e.g., Parboil, Rodinia, SPEC, PARSEC, HPCC), synthetic benchmarks and other applications. All dependencies are resolved at install time and the only user involvement in generating the training data is selecting the decision around which a model needs to be constructed.

The framework also provides a tool for generating proglist files. In this case, the user needs to specify the <buid> and <execute> command that describes how to get from $w$ to $w'$. If $D$ is composite then a seperate <buid> and <execute> command needs to be supplied for each $d_i$. From this information, the tool will infer all of the necessary proglist commands for the target platform.

### Feature Extraction and Processing

The framework can extract any dynamic feature supported by the target platform. We leverage the `perf` module which is standard on Linux kernels $\geq$ 3.0. At install time, the framework determines the number of measurable events that can be used as features. When a new model is to be built all measurable events are probed and these serve as the initial feature set. Measuring one event per program run can be time consuming given that there are hundreds of events and potentially millions of program runs. To address this issue, we include in the framework, a module that takes advantage of multiplexing to automatically determine subsets of performance events that can be measured during a single program run without causing conflicts in hardware counters.

Centering and scaling also play a crucial role in MLMT. The difference in the range of features values can be many orders of magnitude. For instance, the number of executed FP instructions per unit time can be in the billions, while number of page faults can be ins single digits. Both can be equally important for performance and must be included in the feature vector.

### Data Labeling

Labeling can be a tedious and time consuming process. The framework implements an algorithm that performs this task automatically. The roofline model (Williams et al., 2009) is used to establish upper and lower bounds for performance on the target architecture of a given code variant. The relative performance of each entry in the training data is then determined and ranked. A histogram is created based on the ranking and adjusted for the distribution of values. The buckets in the adjusted histogram form the classes for the target model and each entry in the training data is labeled accordingly.

### Feature Selection

Selecting the right features is an extremely important step in MLMT. In most ML-based tuning work, features are typically selected by hand by performance experts (Liao et al., 2009; Stephenson and Amarasinghe, 2005). Although effective in some situations, this ad-hoc approach can be limiting because not all attributes that influence the outcome vector may be known to experts. The framework employs the following series of automated feature selection techniques: (i) eliminating low variance (ii) leave-one-out and (iii) univariate. In each case, how aggressively the pruning is done can be controlled via a parameter.

### Model Evaluation and Selection

Generally, it is not known *a priori* which model is most suitable for a particular instance. The choice of a model often depends on the characteristics of the

training data. In the given framework, the generated training data is analyzed and a set of learning algorithms is selected based on the properties of the data. The selected models are passed through a battery of cross-validation tests. Confidence levels (based on t-test) is computed for the cross-validation results. Only the highest performing ones are presented to the user for testing.

# VI. A MODEL FOR REGISTER ALLOCATION OF CUDA KERNELS

The shared register space is sufficiently large on current GPUs such that the compiler can often make an allocation without incurring too many spills. However, the problem lies in the fact that the number of registers allocated is directly linked with the thread block size. Allocating sufficient registers to avoid spills might enforce a smaller thread block size, which can lead to reduced occupancy and loss of performance. On the other hand, selecting a larger block size might enforce an implicit constraint on the number of registers to be allocated per thread. Moreover, since the launch configuration is determined at runtime it is difficult for the compiler to make a good decision. As an example, consider Fig. VI.1 that shows how the the register allocation scheme adopted by the `nvcc` compiler can force many of the Parboil benchmark to operate at less the 100% occupancy.

We used the framework to develop a learning model that when given a new kernel (source or binary) will predict the number of registers that should be allocated to maximize performance. Thus, in terms of the MLMT abstractions explained in Section V, the model can be expressed as follows

ch:design and implementation

$$M_T^E(\{F\}) \rightarrow \{D\}$$

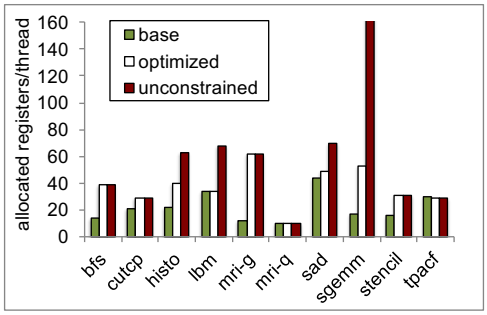


Figure VI.1: Register allocation by nvcc for a subset of Parboil benchmarks

where $E$ represents two Nvidia GPUs based on the Fermi and Kepler architecture, respectively. $T$ is kernel execution time. $D$ is the number of registers that should be allocated which is a composite *decision* with an outcome between $\{16..MAXREG\}$, where $MAXREG$ is environment dependent. To reduce training time and make the model less cumbersome we use expert knowledge to eliminate some outcomes and define
$D = \{d_{def}, d_{16}, d_{24}, d_{32}, d_{40}, d_{48}, d_{64}, d_{512}\}$, where $d_n$ indicates a register allocation of $n$ per kernel and $d_{def}$ is the default allocation.
$F$ initially includes all dynamic *metrics* and *events* measurable in each target GPU. These are obtained via `nvprof` using the `-query-metrics` and `-query-events` flags respectively. $F$ goes through the selection process described in Section IV.
Training set: The base set includes 26 kernels taken from the Parboil, CUDA SDK Samples, SLAM and Rodinia benchmark suites. Each kernel was fed into the code variant generator to generate different variants. The parameters that were changed in the code variant generator include (i) `nvcc` optimizations flags, e.g., O1, O3 etc. (ii) number of registers allocated and (iii) thread block size. These variants created a total of 1230 training instances *per model*.

# VII. EXPERIMENTAL EVALUATION

In this section, we evaluate the register allocation model developed with MLTUNE. We also demonstrate the utility of different aspects of the tool. To evaluate the MLTUNE tool, we have generated 9 training data sets.

## Machine Configuration

The Table VII.1 lists the configuration off each machine where we have conducted the experiments. We name the machines as Knuth and Shadowfax.

Table VII.1: Machine Configuration

| | Knuth | Shadowfax |
|---|---|---|
| Kernel Version | Linux 3.13.0-29-generic | Linux 3.16.0-36-generic |
| Bit Supported | 64 | 64 |
| No of Processors/Cores | 12 | 16 |
| GPU Generation | Kepler | Fermi |
| Memory | 4 GB | 8 GB |

## Feature Selection

I have performed three different feature selection techniques: Univariate feature selection, Accumulated feature selection and incremental feature selection. In this section, I will discuss feature selection techniques and how feature selection impacts the prediction accuracy. We predict the performance of any unseen program in terms of execution time, power and energy. For example, if any unseen program predicts top in terms of time, then we expect that this unseen program will show speedup if we execute the program.

### Univariate Feature Selection

Univariate feature selection technique uses the univariate statistical tests to find out the best selected features. I used a term, feature percentile, in the figures, it

means that percentage of top features selected by the univariate feature selection technique. For example, 10% feature percentile on 44 features is, top 4 features have been selected to generate the prediction accuracy by applying cross validation tests.


Figure VII.1: Feature percentile vs cross validation prediction accuracy of Logistic Regression classifier

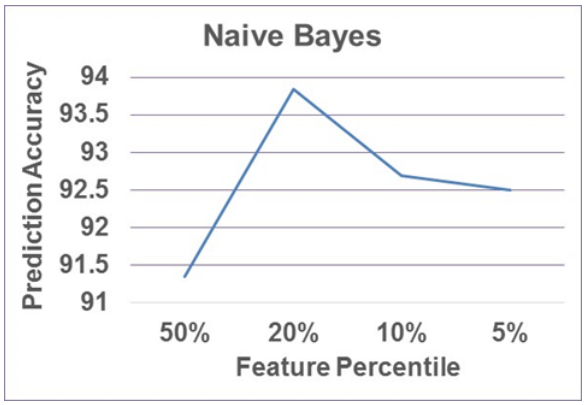


Figure VII.2: Feature percentile vs cross validation prediction accuracy of Naive Bayes classifier

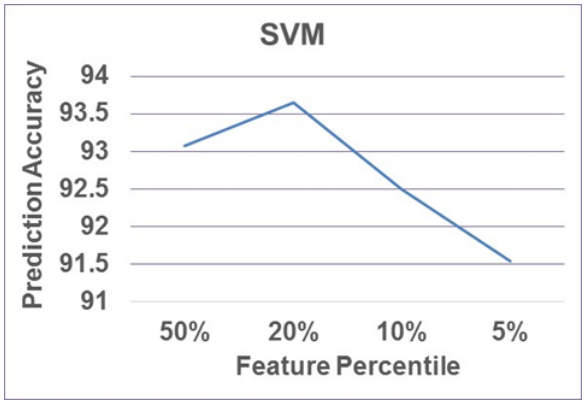


Figure VII.3: Feature percentile vs cross validation prediction accuracy of SVM classifier


Figure VII.4: Feature percentile vs cross validation prediction accuracy of Decision Tree classifier

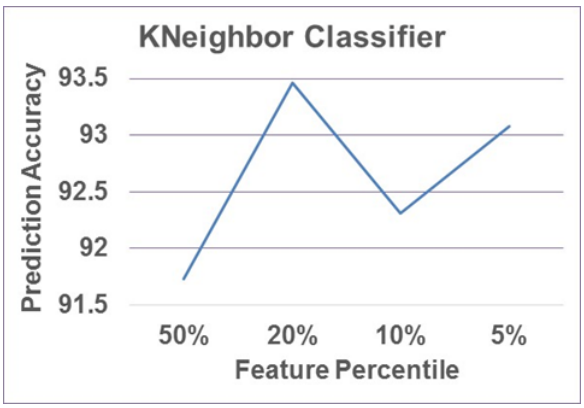


Figure VII.5: Feature percentile vs cross validation prediction accuracy of KNeighbors classifier

If we look at Figure VII.1, we see that selecting less number of features can give highest prediction accuracy. But on the other hand, Figure VII.2 shows decreasing trend in prediction accuracy with less number of features. If we observe the figures VII.1, VII.2 and VII.3, we see that if we choose 20% feature percentile, then it always shows highest prediction accuracy. Figure VII.4 shows that selecting fewer features steadily increases the prediction accuracy, so, we get highest prediction accuracy with 5% best features. But the figure VII.5 shows a sharp increase and decrease with decreasing feature percentile. Overall, all of the ML classifiers shows good prediction accuracy of above 90%.

Accumulated Feature Selection

Accumulated feature selection techniques select the features where prediction accuracy increases by 0%, 2% and 5% from base prediction accuracy. Here, feature threshold means the set of features which are selected by comparing with the base prediction accuracy, so if prediction accuracy increases by greater than 0%, 1% and 2% from previous accuracy, then we keep that feature in the feature threshold in a hashtable data structure. Then we apply each set of features to check the prediction accuracy.

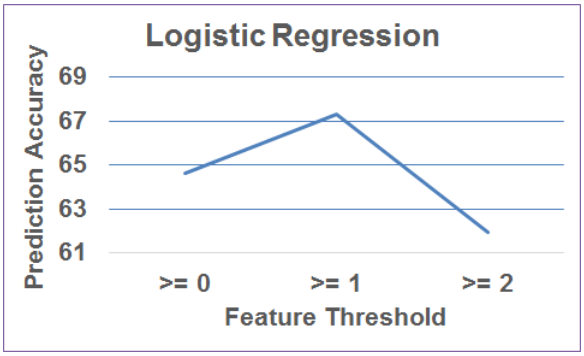


Figure VII.6: Features threshold vs cross validation prediction accuracy of Logistic Regression classifier

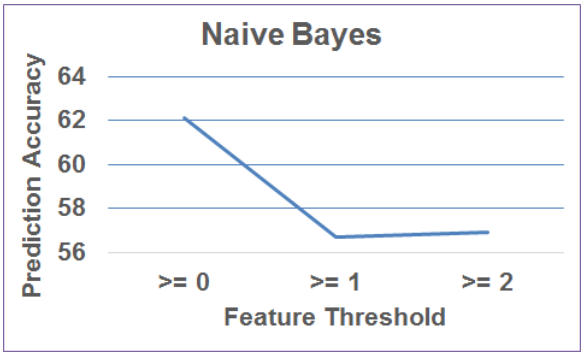


Figure VII.7: Features threshold vs cross validation prediction accuracy of Naive Bayes classifier

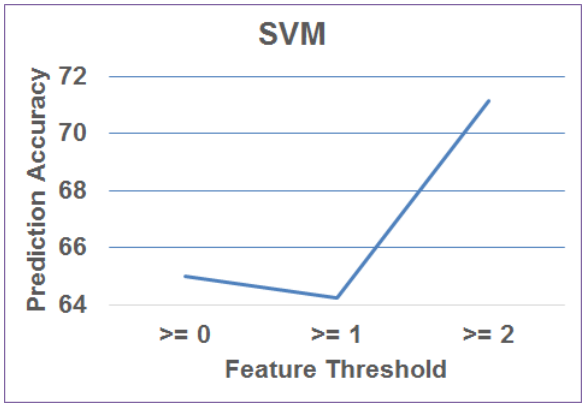


Figure VII.8: Features threshold vs cross validation prediction accuracy of SVM classifier

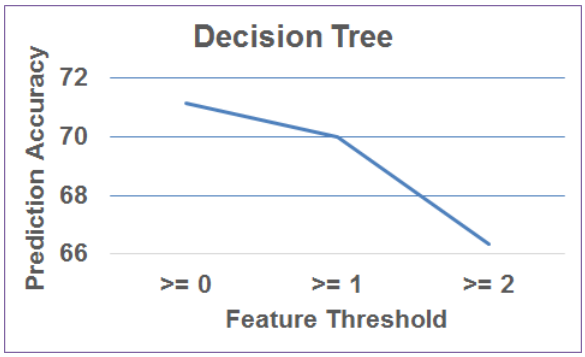


Figure VII.9: Features threshold vs cross validation prediction accuracy of Decision Tree classifier

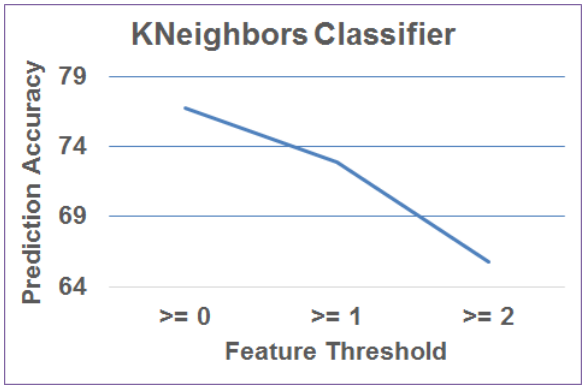


Figure VII.10: Feature threshold vs cross validation prediction accuracy of KNeighbors classifier

Here we see, In most of the cases, figure VII.6, VII.7, VII.9 and VII.10 shows higher feature threshold have lower prediction accuracy, whereas only SVM classifier, figure VII.8, shows higher prediction accuracy with higher feature threshold. Here, KNeighborsClassifier shows higher prediction accuracy among all the Machine Learning classifiers.

Incremental Feature Selection

Incremental or Leave-one-out feature selection techniques drops one feature at a time and then gets the prediction accuracy, and compares with the base prediction accuracy to measure the feature strength. This technique only selects the feature where there is an increase in prediction accuracy by 0%, 1%, 2%, and 5% from base prediction accuracy. So, Here, feature thresholds are 0%, 1%, 2%, and 5%. Each feature threshold lists a set of features.


Figure VII.11: Features threshold vs cross validation prediction accuracy of Logistic Regression classifier

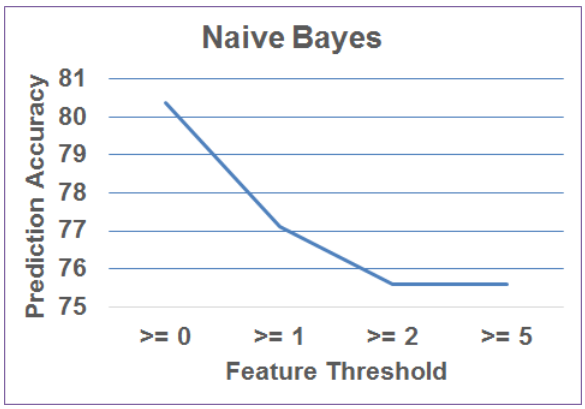


Figure VII.12: Features threshold vs cross validation prediction accuracy of Naive Bayes classifier

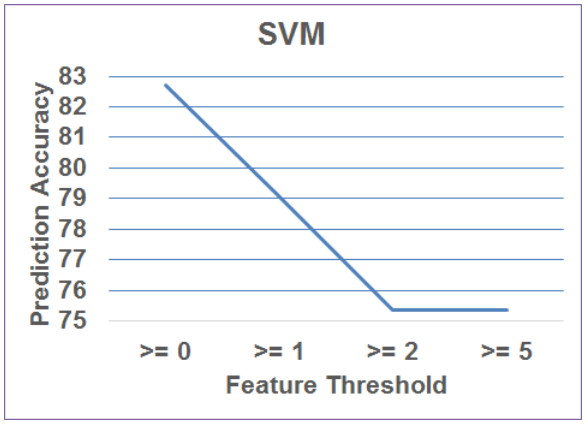


Figure VII.13: Features threshold vs cross validation prediction accuracy of SVM classifier

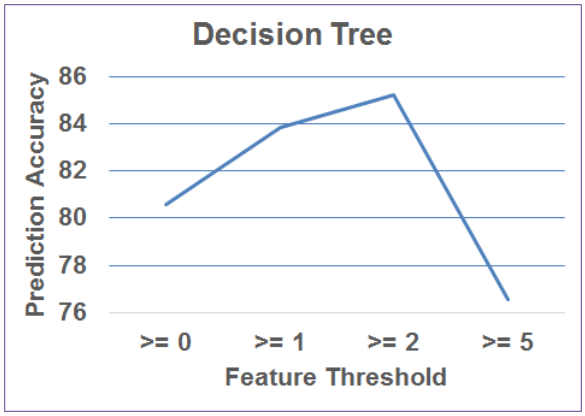


Figure VII.14: Features threshold vs cross validation prediction accuracy of Decision Tree classifier

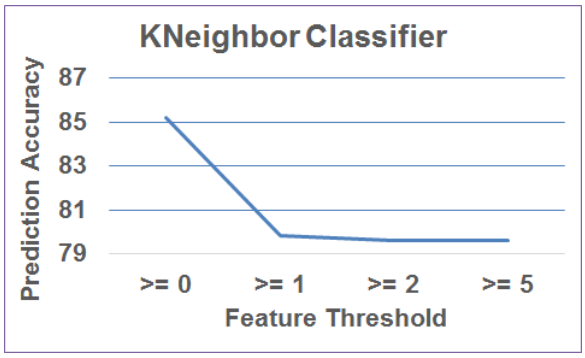


Figure VII.15: Features threshold vs cross validation prediction accuracy of KNeighbors classifier

In terms of incremental feature selection techniques, we see, all of the figures VII.11, VII.12, VII.13, VII.14, and VII.15 show lower prediction accuracy with higher feature threshold. In all of the cases, feature threshold with >=0% shows highest prediction accuracy except in Decision tree classifier.

Merging Feature Selection Techniques

This section compares the different feature selection techniques discussed above. Here we will see, how prediction accuracy varies for different feature selection techniques and with different Machine Learning algorithms.

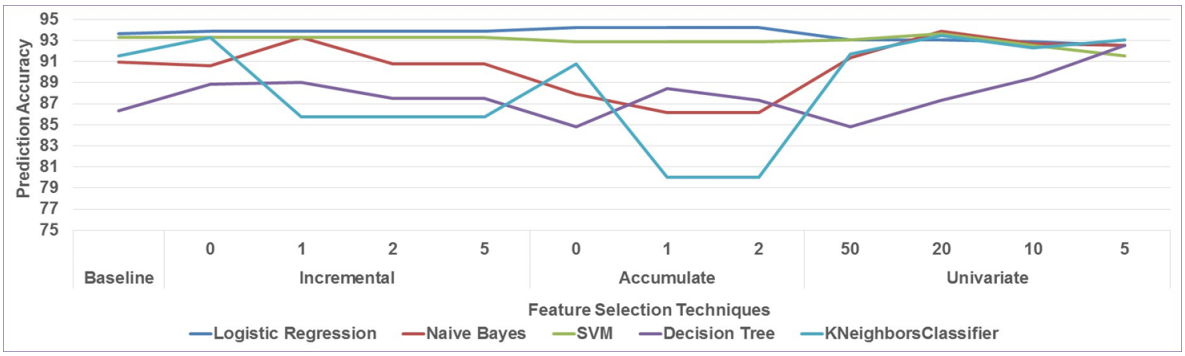


Figure VII.16: Different feature selection techniques vs cross validation prediction accuracy

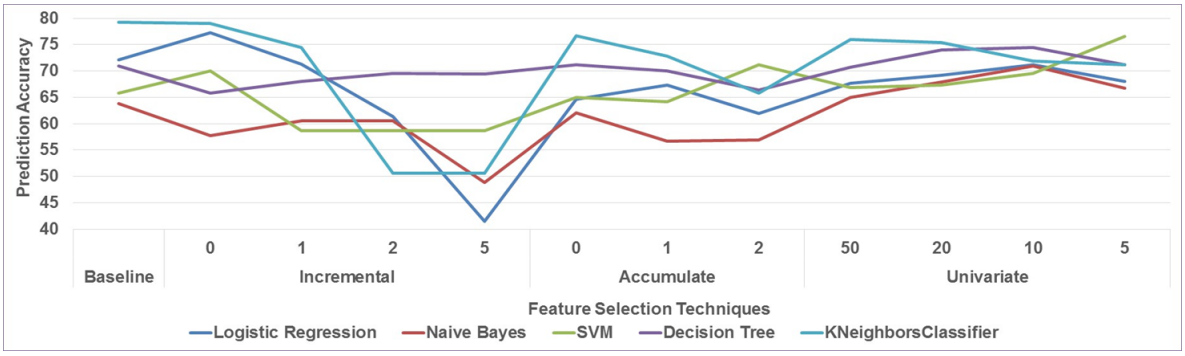


Figure VII.17: Different feature selection techniques vs cross validation prediction accuracy

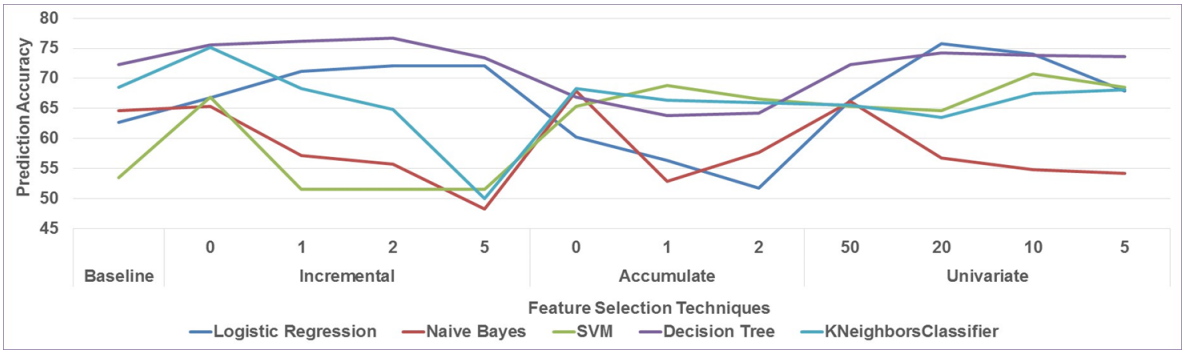


Figure VII.18: Different feature selection techniques vs cross validation prediction accuracy

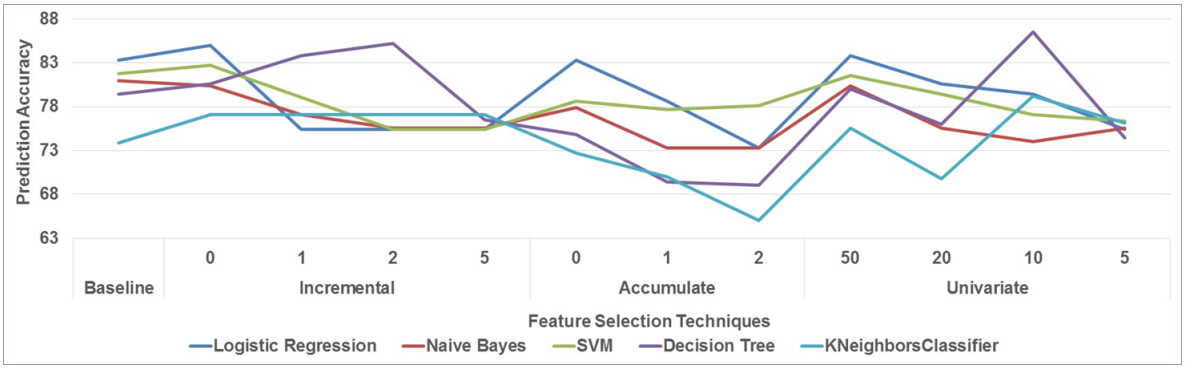


Figure VII.19: Different feature selection techniques vs cross validation prediction accuracy

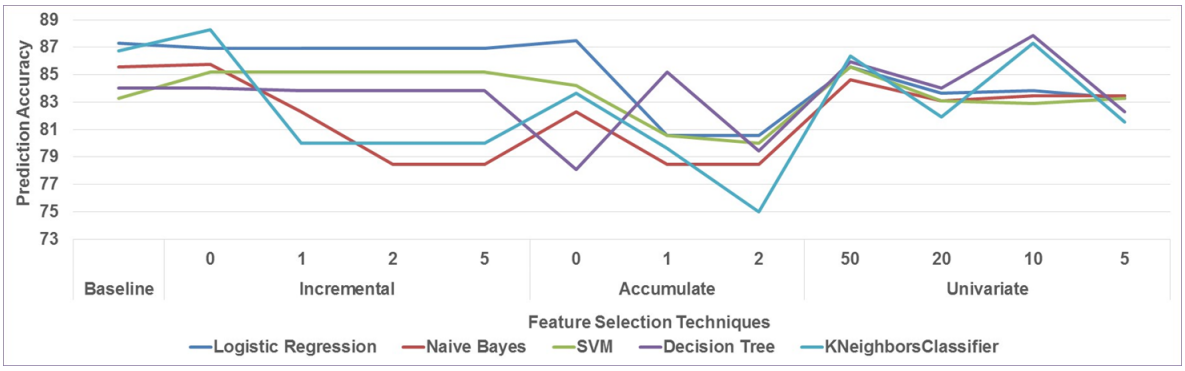


Figure VII.20: Different feature selection techniques vs cross validation prediction accuracy

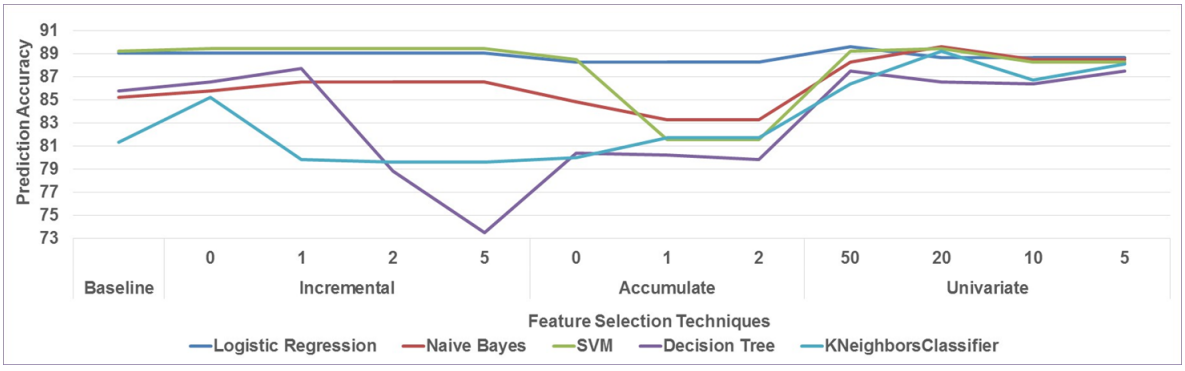


Figure VII.21: Different feature selection techniques vs cross validation prediction accuracy

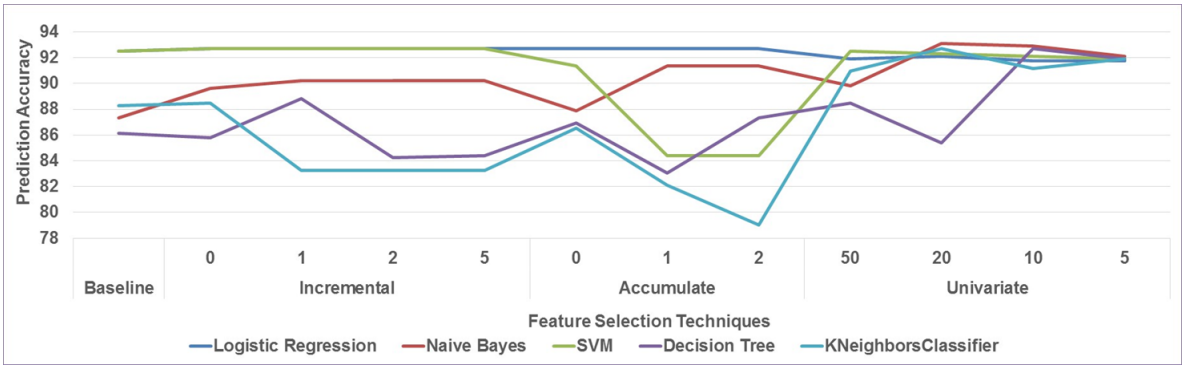


Figure VII.22: Different feature selection techniques vs cross validation prediction accuracy

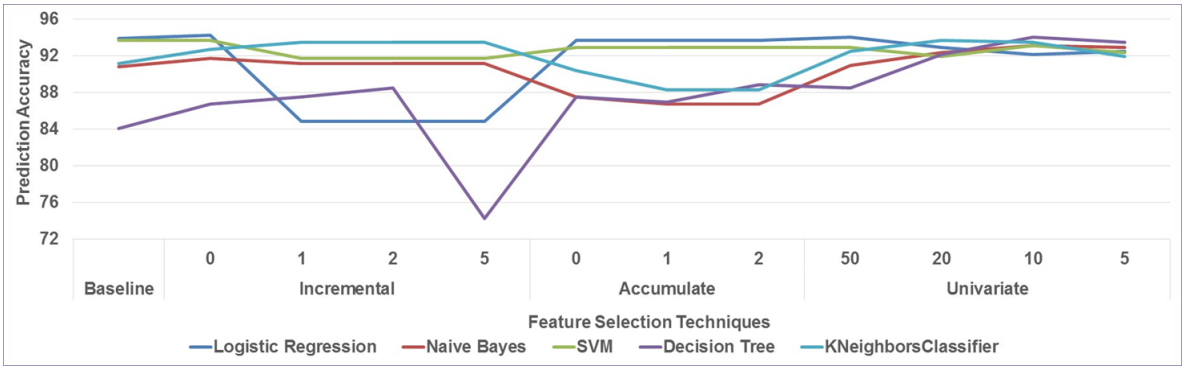


Figure VII.23: Different feature selection techniques vs cross validation prediction accuracy

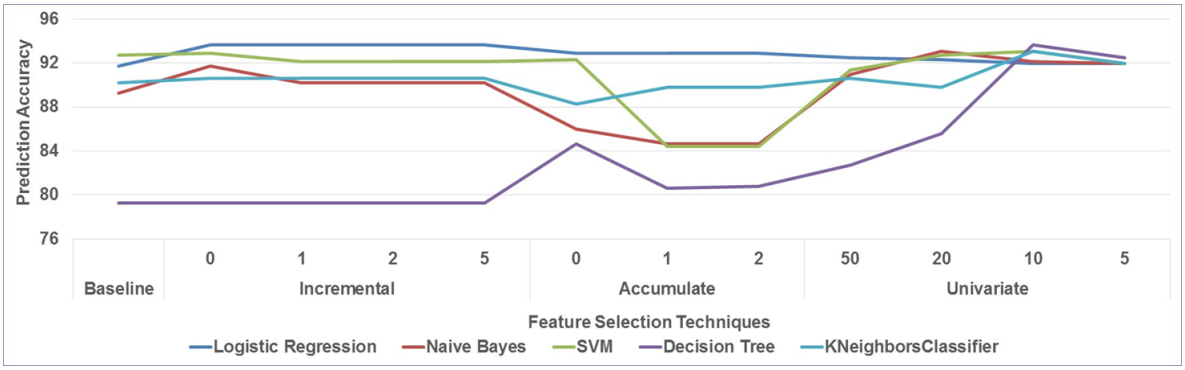


Figure VII.24: Different feature selection techniques vs cross validation prediction accuracy

If we observe all of the figures, we see that prediction accuracy varies with number of features which shows we need feature engineering which can give us a better prediction accuracy. For Example, if we observe figure VII.16, we see that decreasing number of features collected by univariate feature selection shows good prediction accuracy with all of the Machine Learning algorithms. On the

other hand, incremental feature selection techniques shows decreasing trend in prediction accuracy as feature threshold increases. If we also observe the figure VII.17, we see not all of the Machine Learning shows increased prediction accuracy with lower univariate feature selection threshold. So, it can not be easily determined which feature selection techniques would be appropriate to get higher prediction accuracy.

## Model Selection

I performed 10-fold cross validation to get the prediction accuracy. MLTUNE selects the model with the highest prediction accuracy to predict result for any unseen program.

Table VII.2: Prediction accuracy of Different ML models

| ML Models | Scikit-Learn | Weka |
|---|---|---|
| Logistic Regression | 93.65 | 92.50 |
| Naive Bayes | 90.96 | 71.53 |
| KNeighborClassifiers | 91.53 | 94.61 |
| SVM | 93.26 | 85.00 |

If we observe the table VII.2, we see KNeighbors Classifier has the best prediction accuracy over other classifier, whereas, Naive Bayes shows very poor performance.

To understand the quality of the model, precision and recall have chosen as classification metrics.

Table VII.3: Classification metrics

| ML Models | scikit-learn | | weka | |
|---|---|---|---|---|
| | Precision | Recall | Precision | Recall |
| Logistic Regression | 92.00 | 94.00 | 93.00 | 92.00 |
| Naive Bayes | 90.00 | 91.00 | 93.00 | 72.00 |
| KNeighborClassifiers | 92.00 | 92.00 | 94.50 | 94.60 |
| SVM | 91.00 | 93.00 | 72.30 | 85.00 |

If we observe the table VII.3, we see KNeighbors Classifier has the better prediction quality in terms of classification metrics over other classifier, whereas, Naive Bayes shows very poor performance in terms of recall metrics.

## Model Internals

This section discuss the feature importance and model internals described by decision tree figure.

### Decision Tree

Figure VII.25 shows that performance counter local_replay_overhead, text_cache_throughput, and local_store_throughput are the deciding features for the next branches to explore and to decide the prediction.

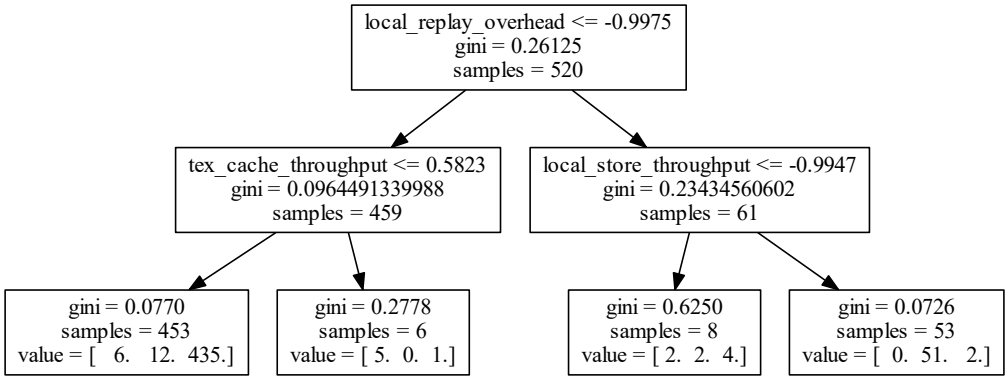


Figure VII.25: Model Internals

Principal Component Analysis with Varimax Rotation

Principal Component Analysis(PCA) converts multi-dimensional data sets into a set of orthogonal components that explain a maximum amount of variance in the dataset.

Datasets converted by PCA can have dense data which is sometimes hard to interpret. Varimax rotation transforms the data into explainable format which makes it easy to analyze the important factors of the dataset.

I have done PCA with varimax rotation on Kepler dataset. I have considered the components which have standard deviation greater than 1.0. I have found out there are 10 factors which correspond the performance counters most. For example, Figure VII.26 shows that local_load_transactions, local_store_transactions,dram_read_transactions, dram_write_transactions,l2_read_transacriotions and l2_write_transactions highly loads together for a factor. I name this factor as Memory Transactions.

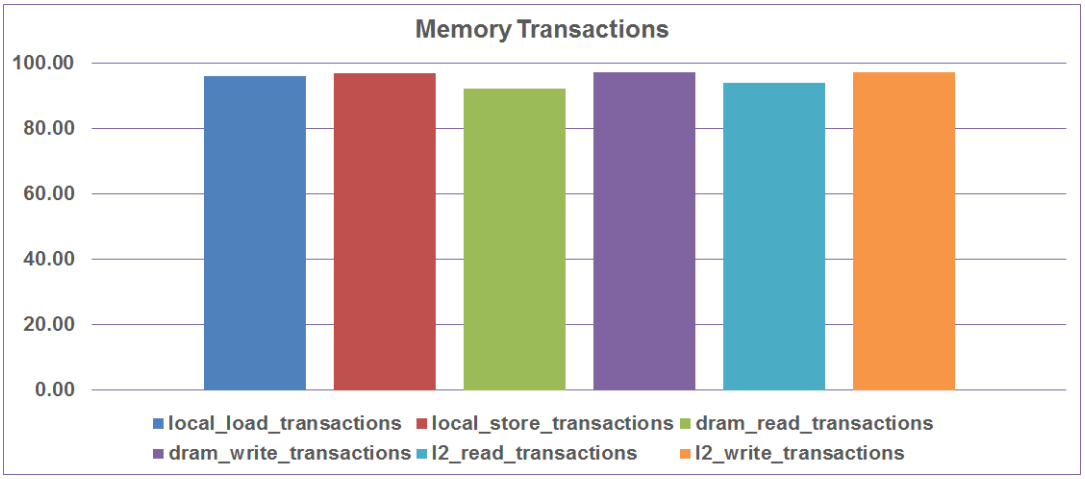


Figure VII.26: Varimax rotation: Memory Transaction

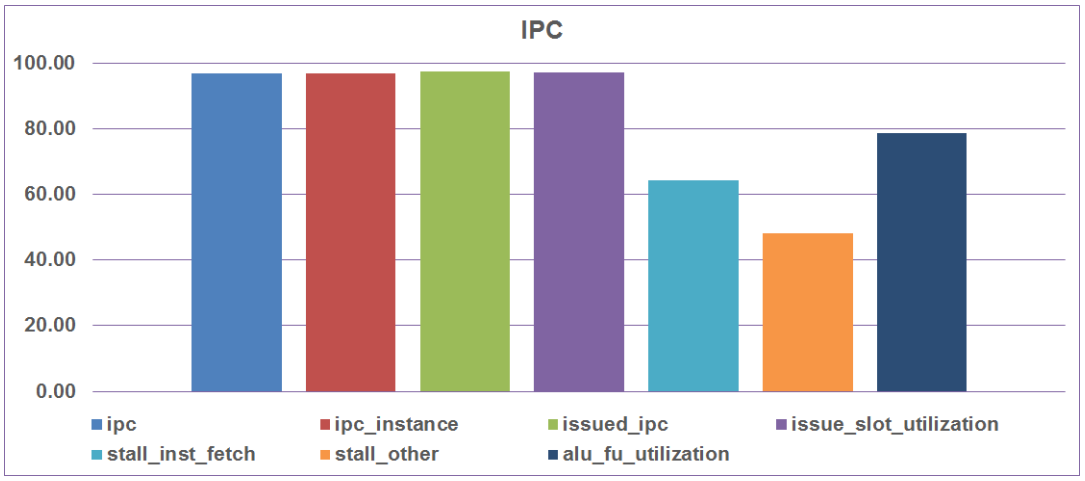


Figure VII.27: Varimax rotation: Instructions

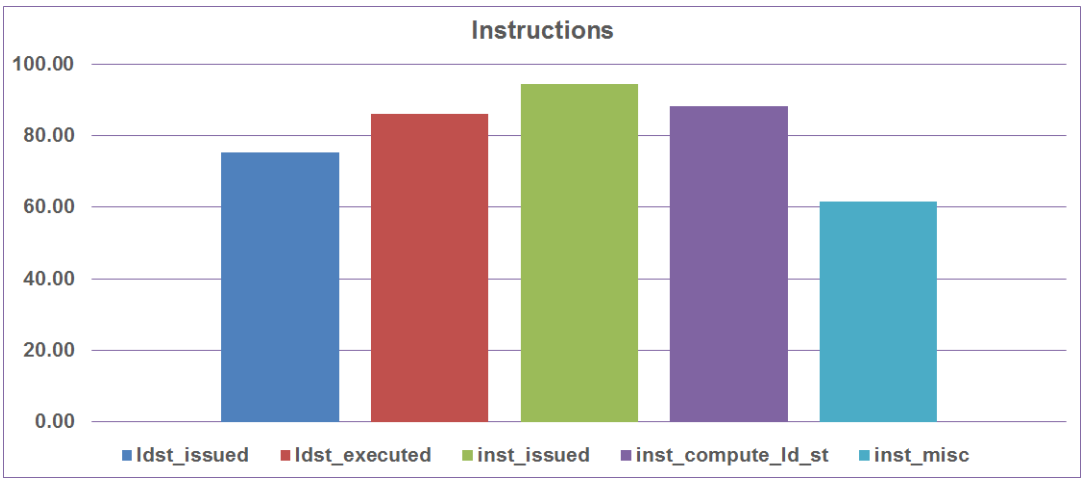


Figure VII.28: Varimax rotation: Register Usage

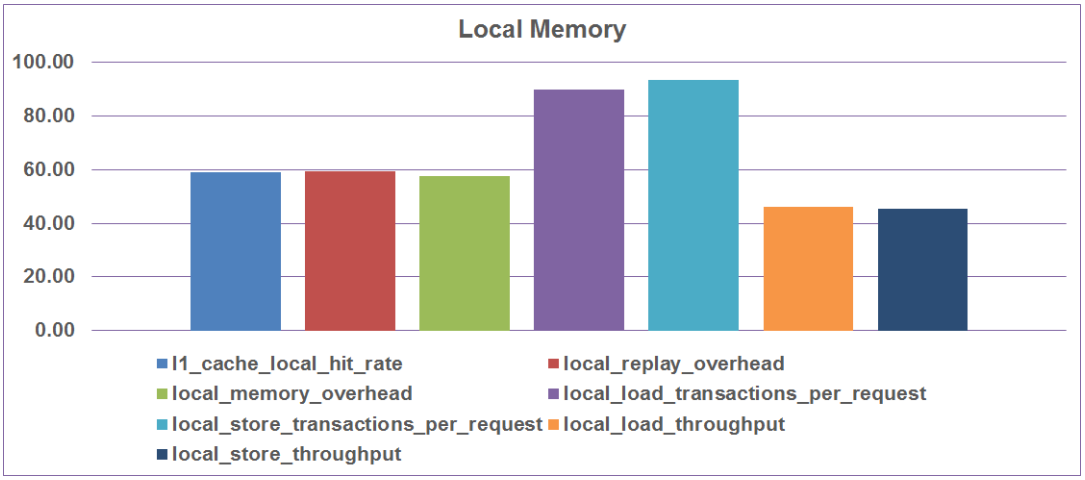


Figure VII.29: Varimax rotation: IPC

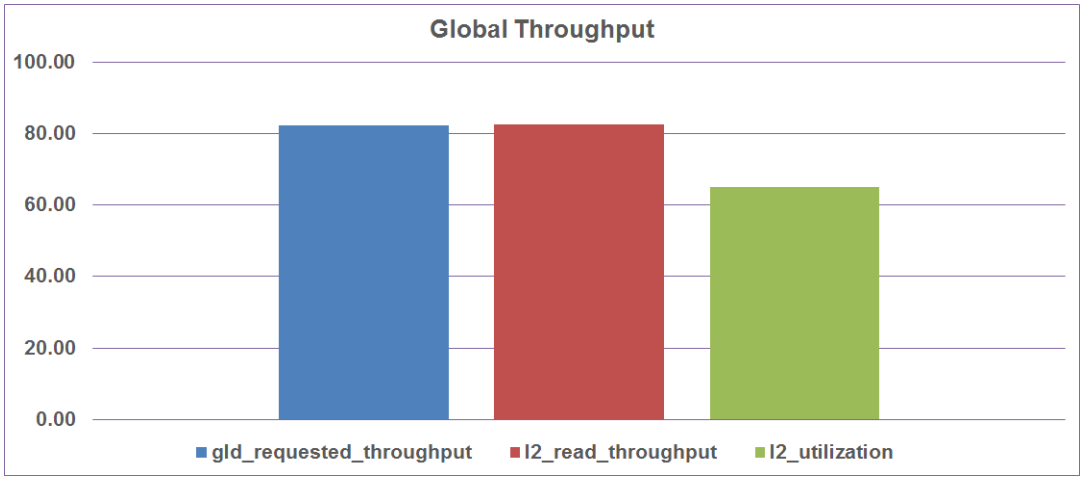


Figure VII.30: Varimax rotation: Local Memory

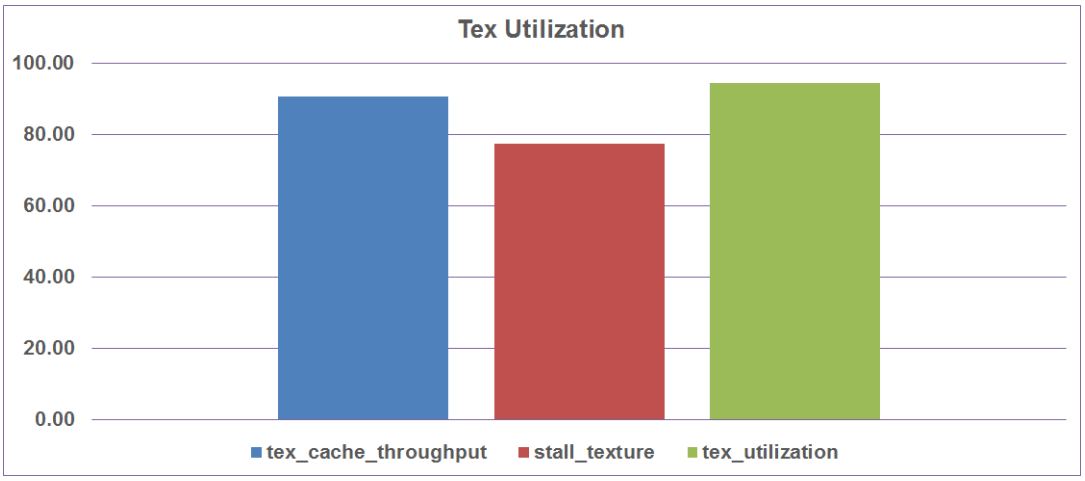


Figure VII.31: Varimax rotation: Read Throughput

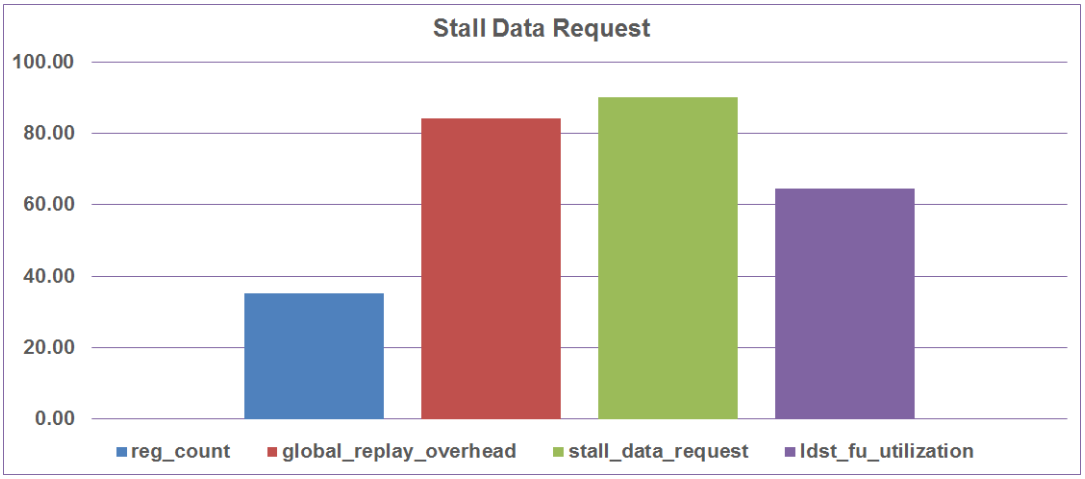


Figure VII.32: Varimax rotation: Global Throughput

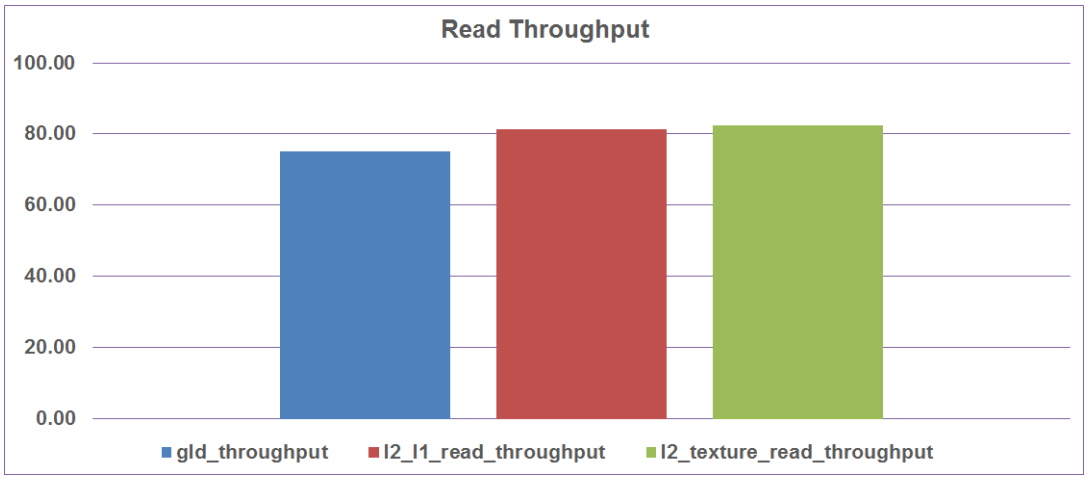


Figure VII.33: Varimax rotation: Cache Throughput

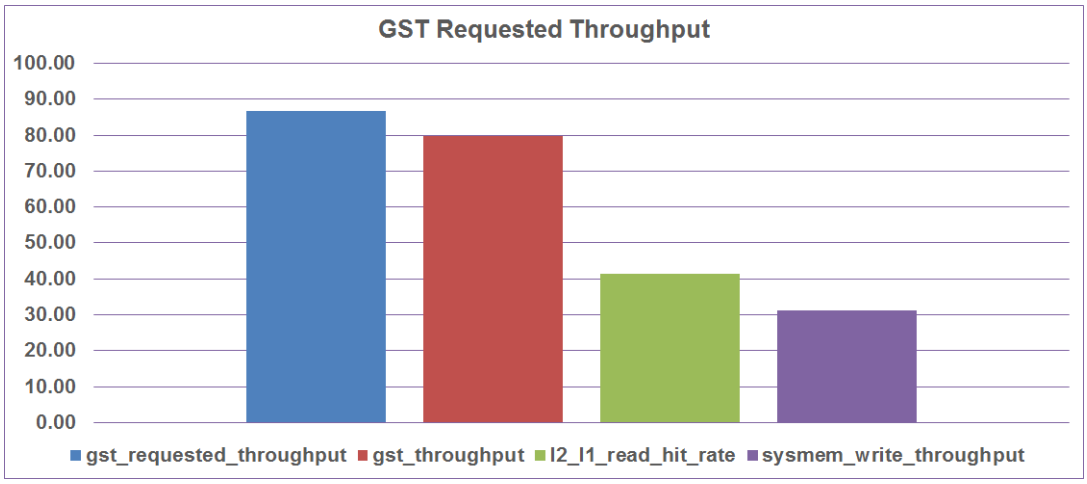


Figure VII.34: Varimax rotation: Store Throughput

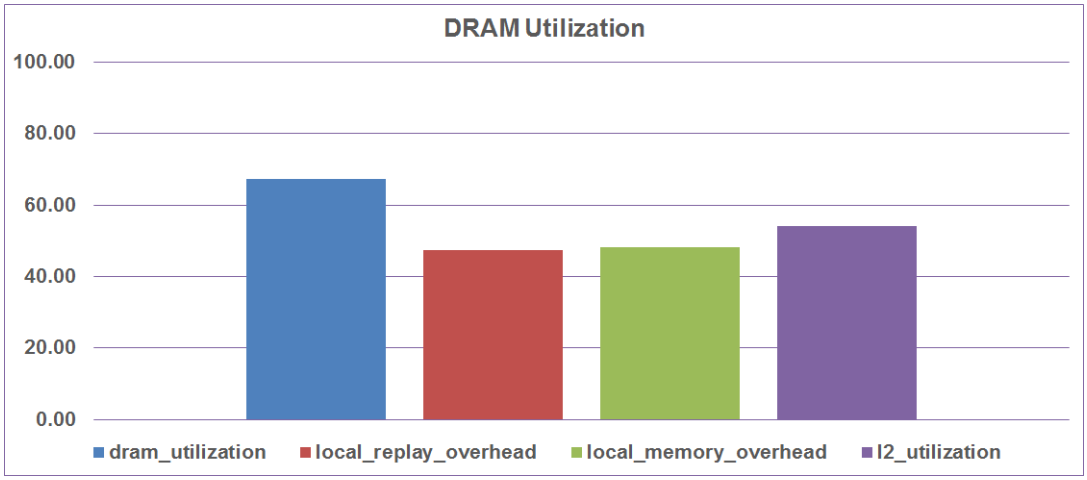


Figure VII.35: Varimax rotation: DRAM Utilization

## Portability

### Performance Counters

MLTUNE tool can automatically extract the performance counters of the respective system. The performance counters can vary system to system. To check the portability of the MLTUNE tool, Performance counters have been extracted from Fermi and Kepler.

Table VII.4 shows the performance counters in Kepler and Fermi. There is a one performance counter global_replay_overhead which exist in Kepler, but not in Fermi.

Table VII.4: Performance counters

| **Features in Kepler** | **Features in Fermi** |
|---|---|
| l1_cache_local_hit_rate | l1_cache_local_hit_rate |
| ipc | ipc |
| gld_requested_throughput | gld_requested_throughput |
| gst_requested_throughput | gst_requested_throughput |
| ipc_instance | ipc_instance |
| global_replay_overhead | tex_cache_throughput |

| | |
|---|---|
| tex_cache_throughput | dram_read_throughput |
| dram_read_throughput | gst_throughput |
| gst_throughput | gld_throughput |
| gld_throughput | local_replay_overhead |
| local_replay_overhead | l2_l1_read_hit_rate |
| l2_l1_read_hit_rate | l2_l1_read_throughput |
| l2_l1_read_throughput | l2_texture_read_throughput |
| l2_texture_read_throughput | local_memory_overhead |
| local_memory_overhead | issued_ipc |
| issued_ipc | issue_slot_utilization |
| issue_slot_utilization | local_load_transactions_per_request |
| local_load_transactions_per_request | local_store_transactions_per_request |
| local_store_transactions_per_request | local_load_transactions |
| local_load_transactions | local_store_transactions |
| local_store_transactions | dram_read_transactions |
| dram_read_transactions | dram_write_transactions |
| dram_write_transactions | l2_read_transactions |
| l2_read_transactions | l2_write_transactions |
| l2_write_transactions | local_load_throughput |
| local_load_throughput | local_store_throughput |
| local_store_throughput | l2_read_throughput |
| l2_read_throughput | sysmem_write_throughput |
| sysmem_write_throughput | ldst_issued |
| ldst_issued | ldst_executed |
| ldst_executed | stall_inst_fetch |

| | |
|---|---|
| stall_inst_fetch | stall_data_request |
| stall_data_request | stall_texture |
| stall_texture | stall_other |
| stall_other | l2_utilization |
| l2_utilization | tex_utilization |
| tex_utilization | dram_utilization |
| dram_utilization | ldst_fu_utilization |
| ldst_fu_utilization | alu_fu_utilization |
| alu_fu_utilization | inst_issued |
| inst_issued | inst_compute_ld_st |
| inst_compute_ld_st | inst_misc |
| inst_misc | |

Model Accuracy

To test the trained model and the model accuracy, 99 unseen programs have been invoked to the model.

Figure VII.36 shows the model accuracy over cross platform. Here K-K means unseen programs generated on Kepler and tested in Kepler Model and K-F means unseen programs generated on Kepler and tested on Fermi model. Even though prediction accuracy is high for Logistic Regression, Naive Bayes and Decision Tree, but the accuracy score does not show a match with the prediction accuracy. We are investing that why the model accuracy and prediction accuracy differs.

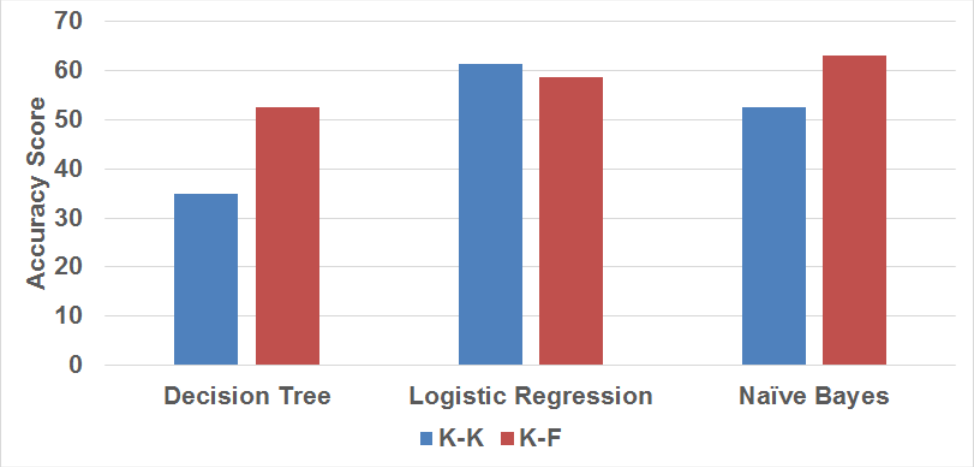


Figure VII.36: Cross Platform Model Accuracy

Performance Gains

This section discuss the speedup over predicted registers. I have trained five models for each dataset so I have total 45 models for Kepler and 40 models for Fermi. I have employed 99 unseen programs to see the prediction and the speedup over those predicted registers. MLTUNE employs forward checking and reverse checking to predict the top performed register for the unseen program. For example, If we perform forward model prediction checking, an unseen program with 16 registers may be predicted to run on 24 registers to get good performance. On the other hand, if we perform reverse model prediction checking, the same program may be predicted to perform good on 512 registers. All of the figures in this section show the experiment results with ten different applications: depthvertex, halfsample, integrate, raycast, reduce, renderdepth, rendertrack, rendervolume, track and vertexnorm with four different register variants: default, 16, 20, and 24 registers. For example, If we look at figure VII.37, we gathered execution time of halfsample application with four different number of registers: default, 16, 20 or 24 registers. And then we run the same program with the predicted registers. Then we plot the graph to show the speedup. So, here, for example, halfsample application with 16 registers perform better with predicted registers. We follow the same behavior with all the

experiments in this section.

Speedup

Figure VII.37 shows that prediction of unseen programs generated in kepler and its speedup on the predicted register generated from the models trained on kepler. It shows a generous speedup for several unseen programs over predicted registers, but there are some programs where performance downgrades over predicted registers. For example, If we invoke rendervolume application with 16 registers, then we see a speedup.

Figure VII.39 shows that unseen programs generated on kepler performs worst on fermi model when applied forward model checking. For example, halfsample application with 20 and 24 registers shows degraded performance. On the other hand Figure VII.40 shows speedup for several unseen programs when predicting using reverse model checking. For example, reduce application with 16 registers shows double speedup.

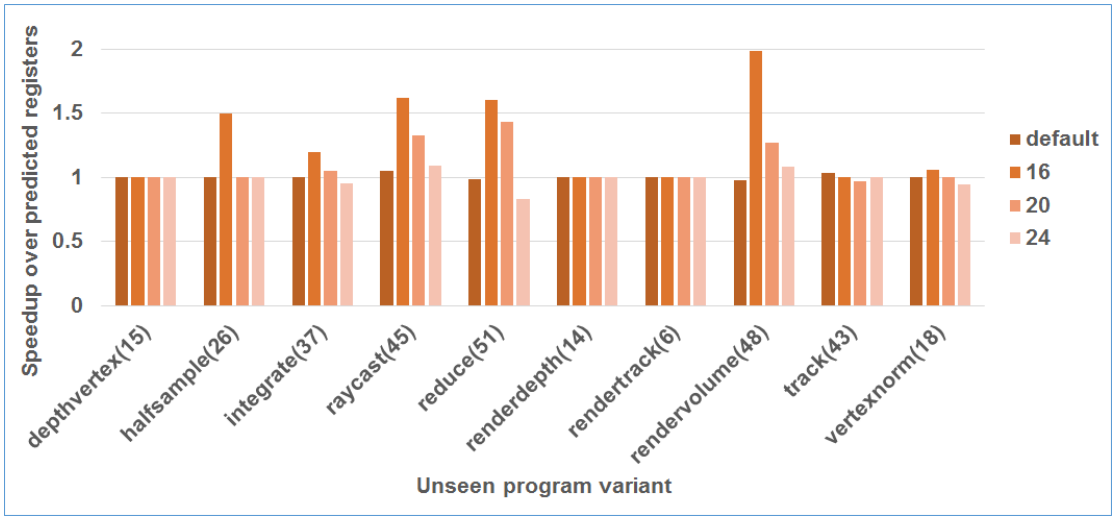


Figure VII.37: Speedup over predicted registers: Kepler programs on Kepler models using forward model checking

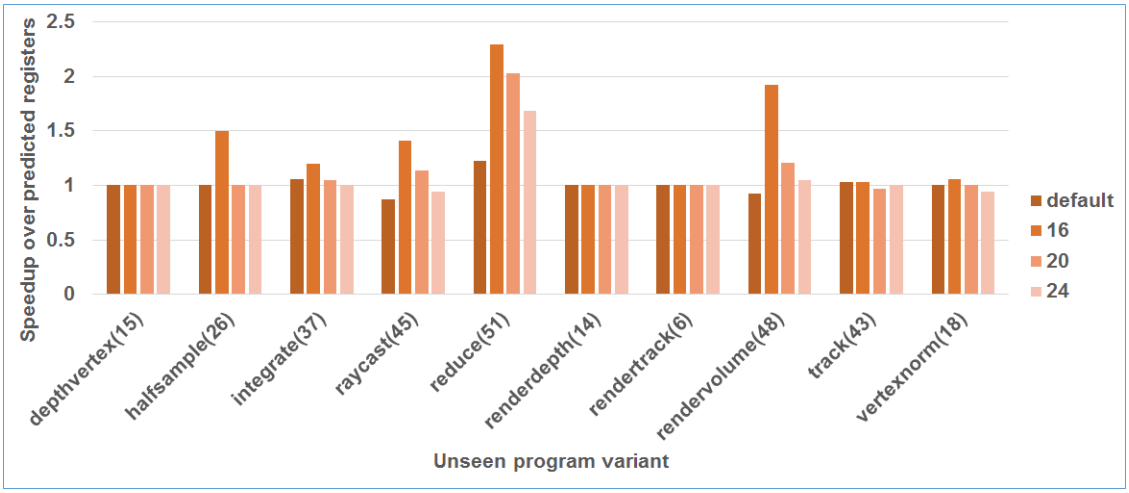


Figure VII.38: Speedup over predicted registers: Kepler programs on Kepler models using reverse model checking

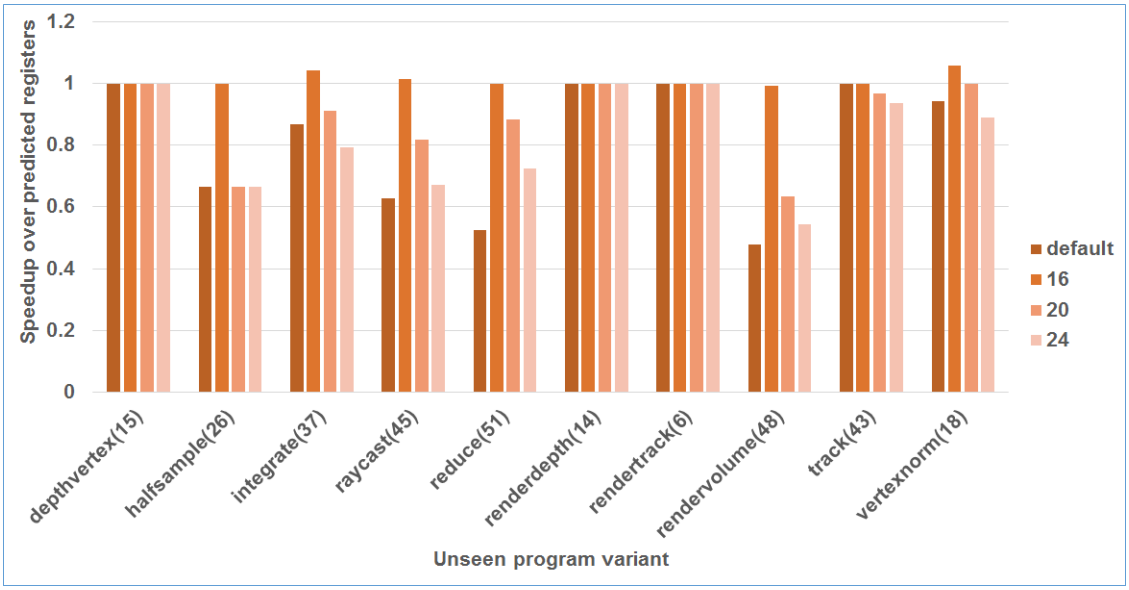


Figure VII.39: Speedup over predicted registers: Kepler programs on Fermi models using forward model checking

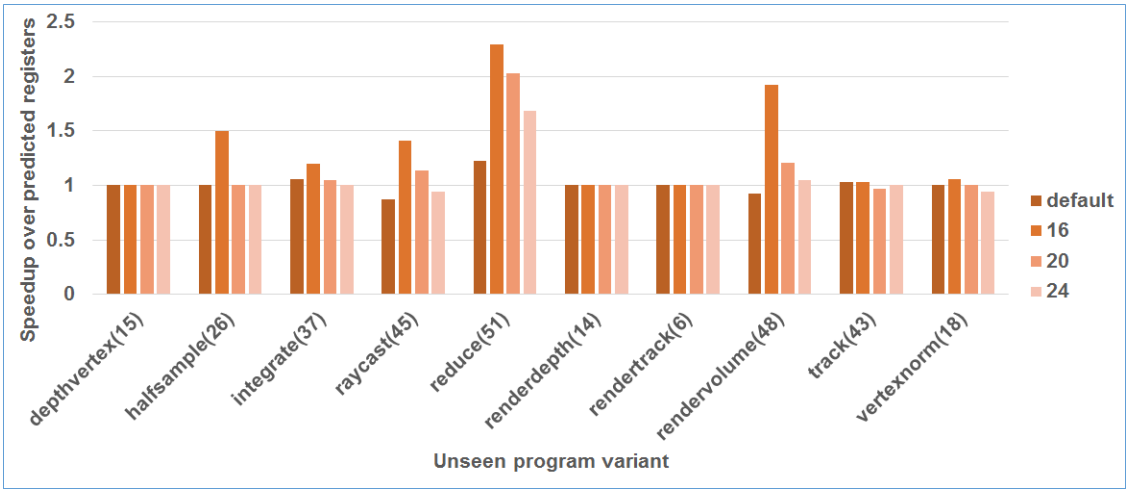


Figure VII.40: Speedup over predicted registers: Kepler programs on Fermi models using reverse model checking

## Power Gain

I have analyzed power gain over predicted registers. As we see from the figure, For example Figure VII.41, Unseen programs does not show much improvement over predicted registers. For example, if we run depthvertex application with the predicted registers instead of 16, 20 or 24 registers, it shows no power gain. The same scenario is for all the figures related to power gain.

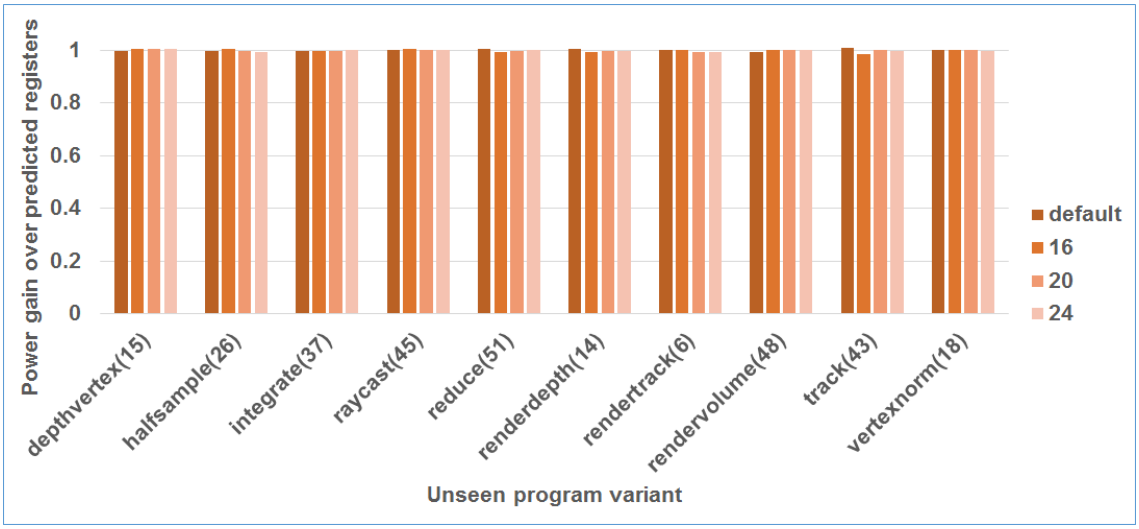


Figure VII.41: Power gain over predicted registers: Kepler programs on Kepler models using forward model checking

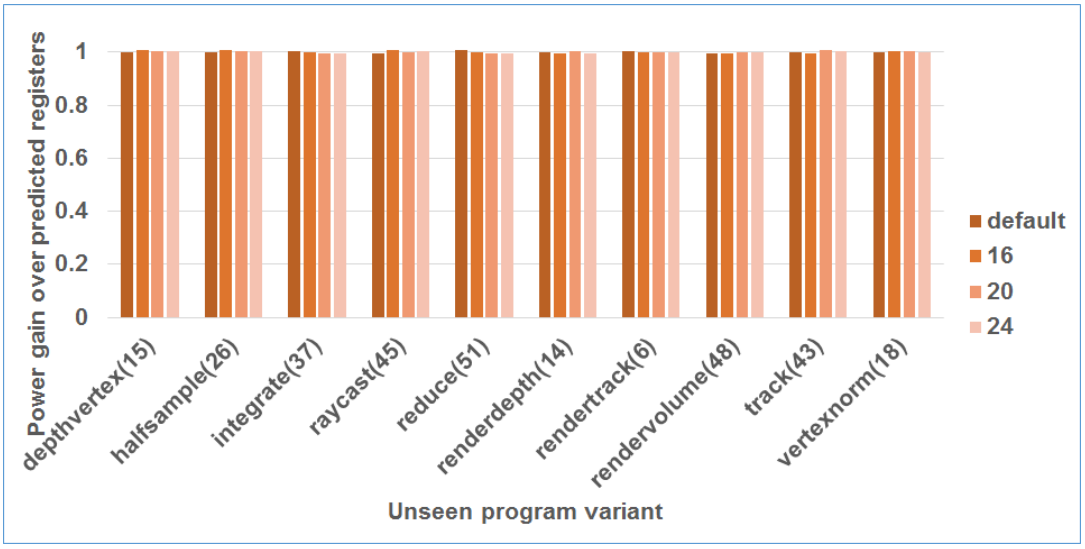


Figure VII.42: Power gain over predicted registers: Kepler programs on Kepler models using reverse model checking

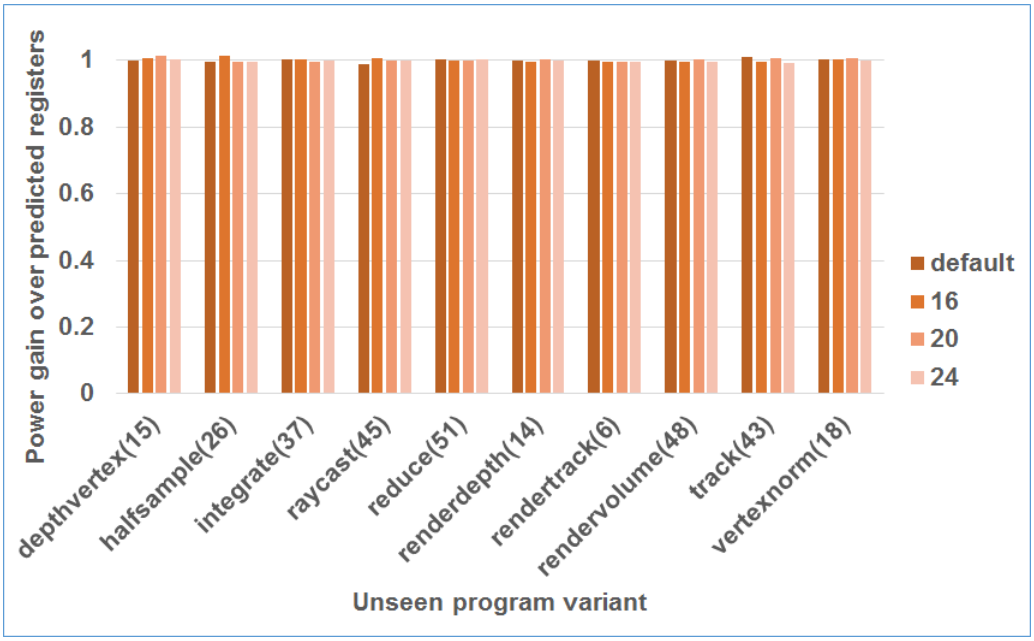


Figure VII.43: Power gain over predicted registers: Kepler programs on Fermi models using forward model checking

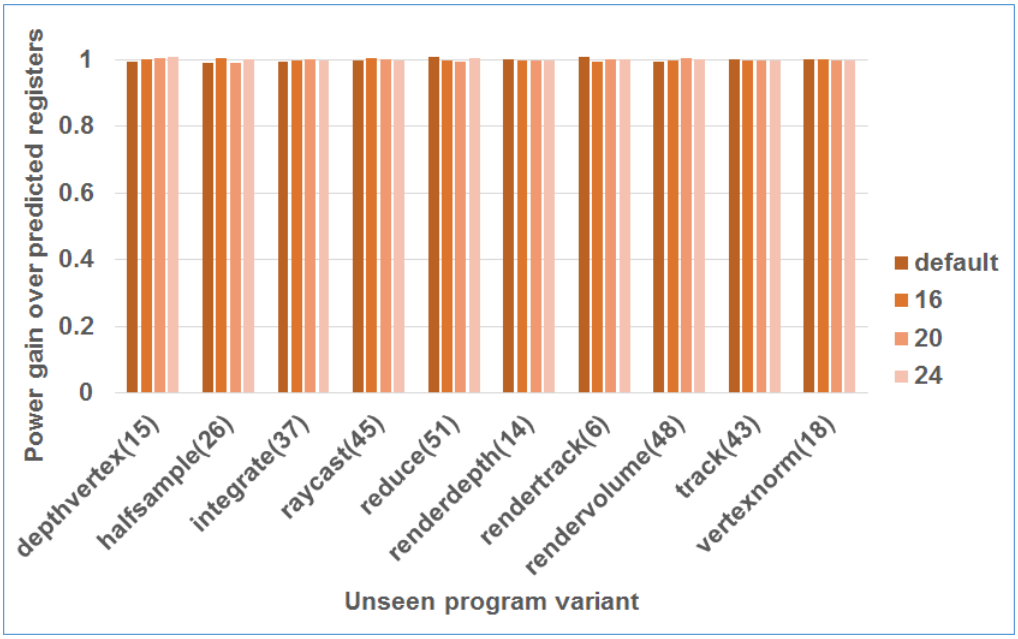


Figure VII.44: Power gain over predicted registers: Kepler programs on Fermi models using reverse model checking

Speedup in Different ML Models

In this section, I discuss how speedup differs over predicted registers in different models using forward/in order model checking and reverse model checking.

### In order Model Invocation

In this process of in order model invocation, I invoked unseen programs to the models sequentially from 16 registers to 512 registers. Here, as soon as the workflow get good prediction, it suggest to use that register.

Figure VII.45 shows that halfsample programs have degraded performance in Decision tree model. But we see stable performance of halfsample programs in other models. If we observe all the figures VII.45, VII.46, VII.47, VII.48 and VII.49, we see that every model show equal or better performance in predicted registers except decision tree.

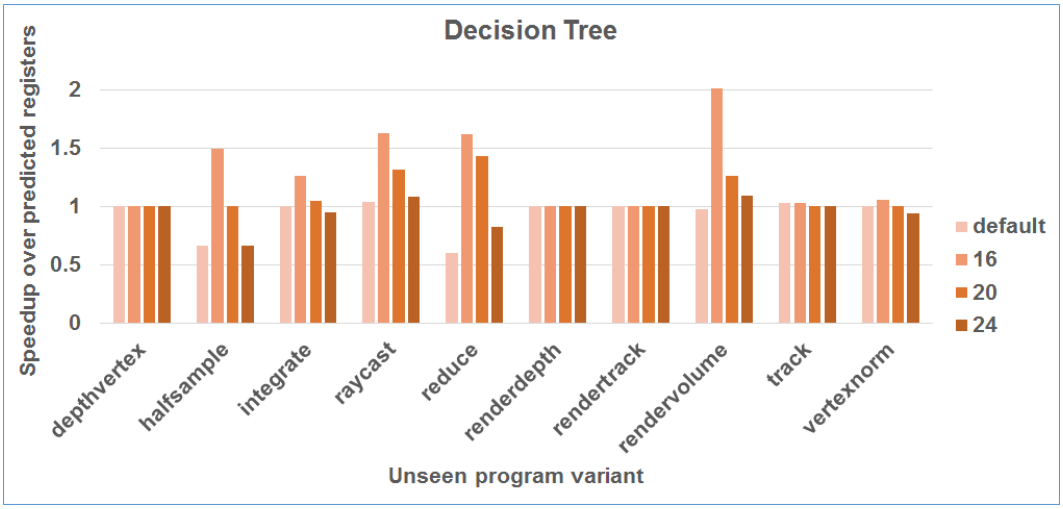


Figure VII.45: Speedup over predicted registers: Kepler programs on Kepler models

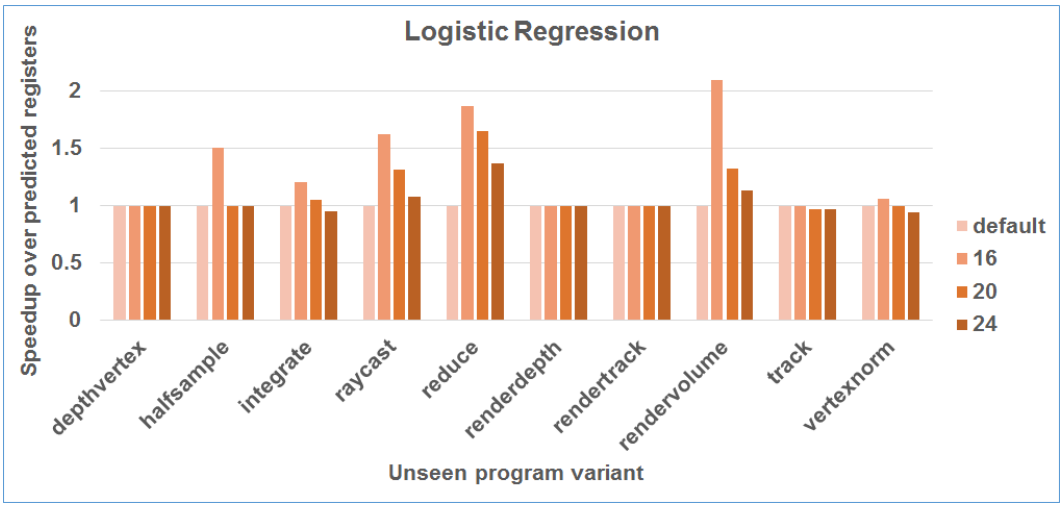


Figure VII.46: Speedup over predicted registers: Kepler programs on Kepler models

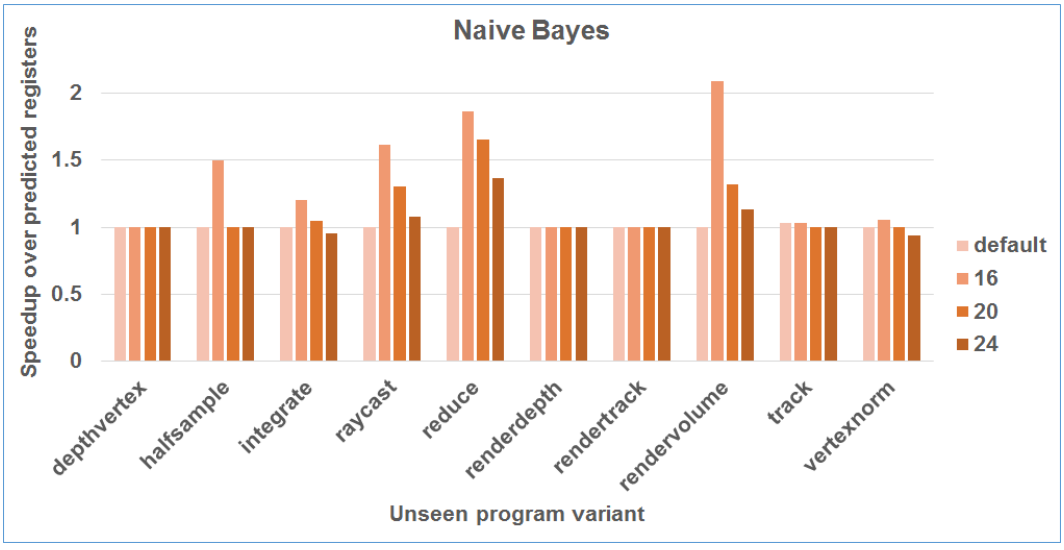


Figure VII.47: Speedup over predicted registers: Kepler programs on Kepler models

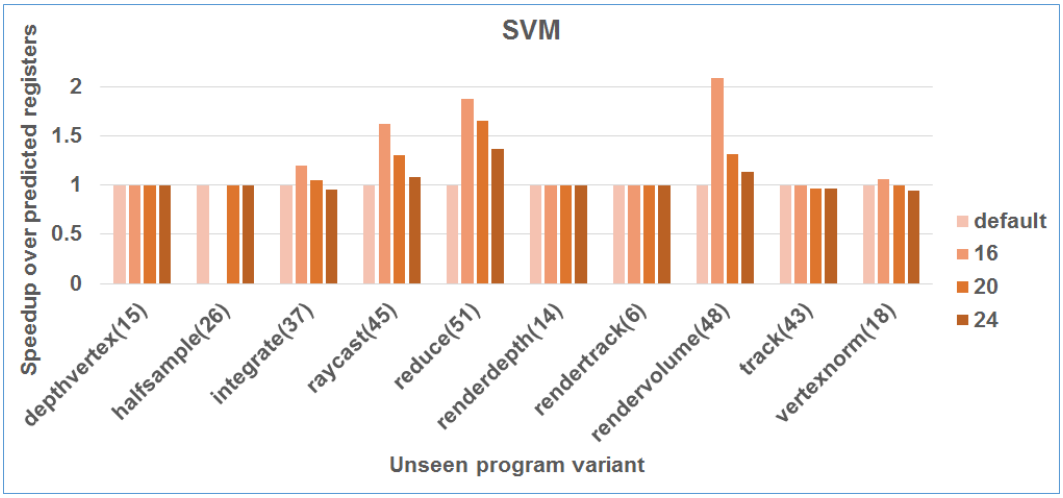


Figure VII.48: Speedup over predicted registers: Kepler programs on Kepler models

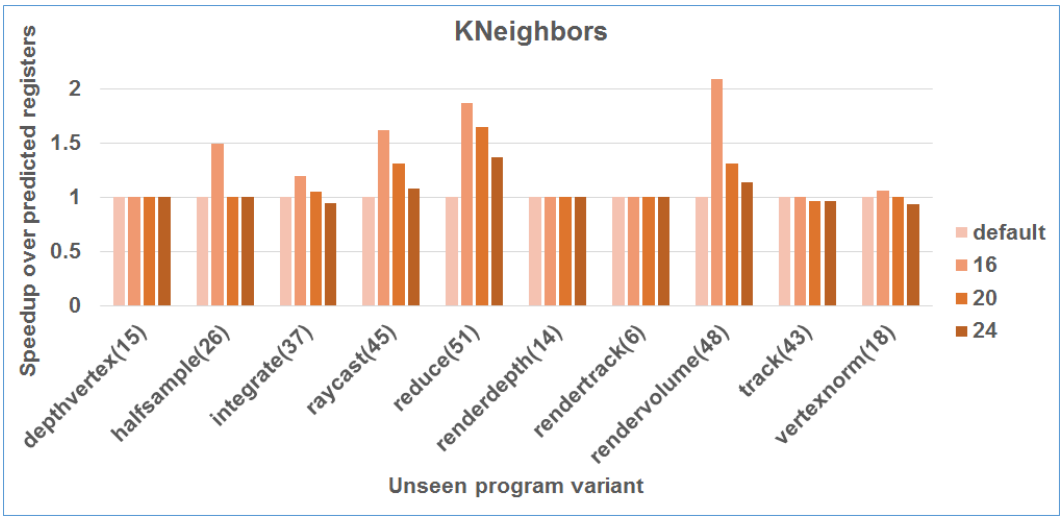


Figure VII.49: Speedup over predicted registers: Kepler programs on Kepler models

### Reverse order Model Invocation

In this process of reverse order model invocation, I invoked unseen programs to the models sequentially from 512 registers to 16 registers. Here, as soon as the workflow get good prediction, it suggest to use that register.

We see that figures VII.50, VII.51, VII.52, VII.53, VII.54 show better or equal performance improvement if we run the program with the predicted registers.

For example, In all of the models, reduce application shows double speedup if we run a program with the predicted register instead of 16 registers.

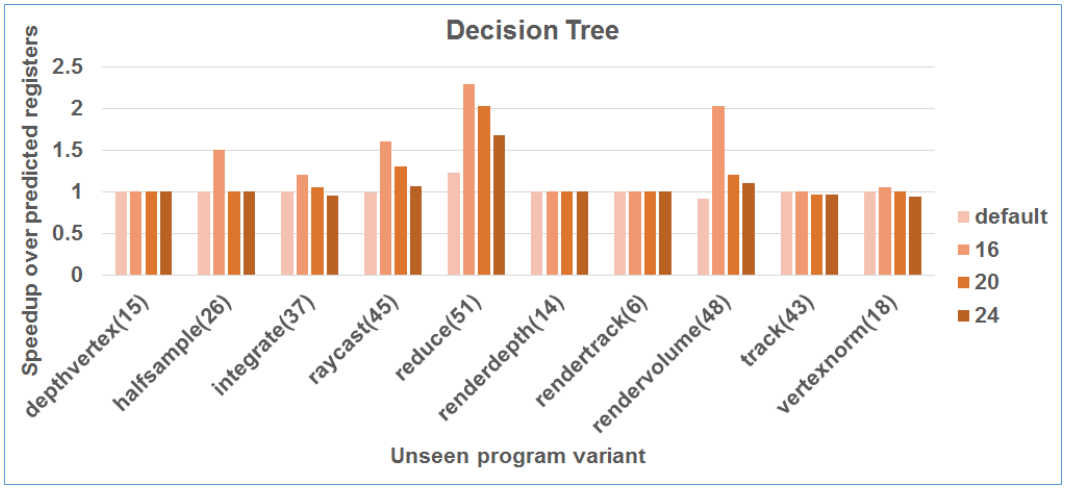


Figure VII.50: Speedup over predicted registers: Kepler programs on Kepler models

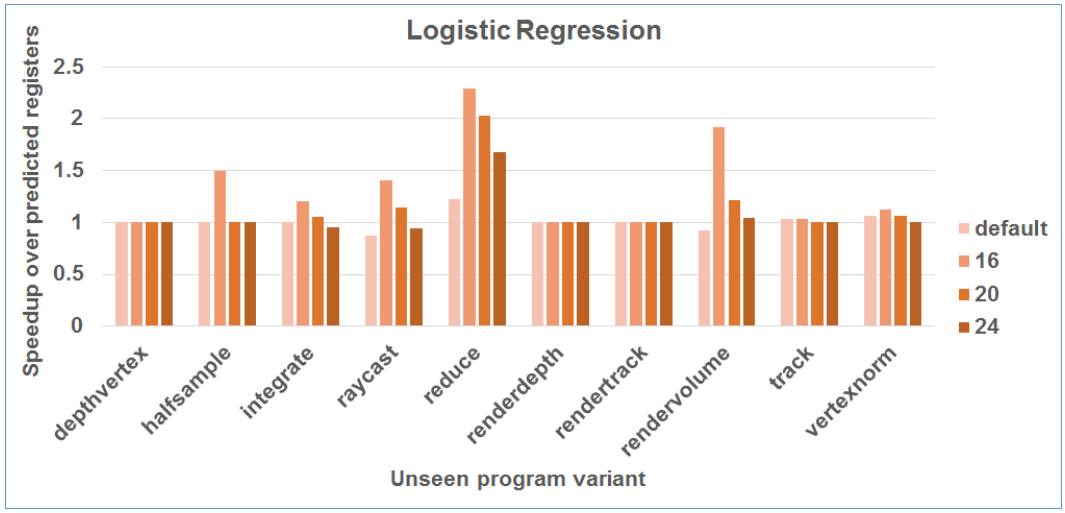


Figure VII.51: Speedup over predicted registers: Kepler programs on Kepler models

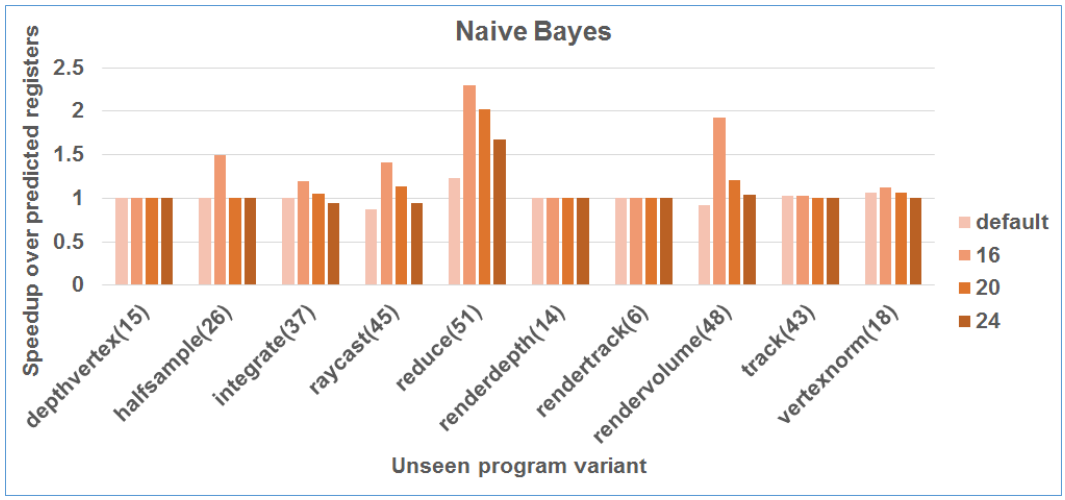


Figure VII.52: Speedup over predicted registers: Kepler programs on Kepler models

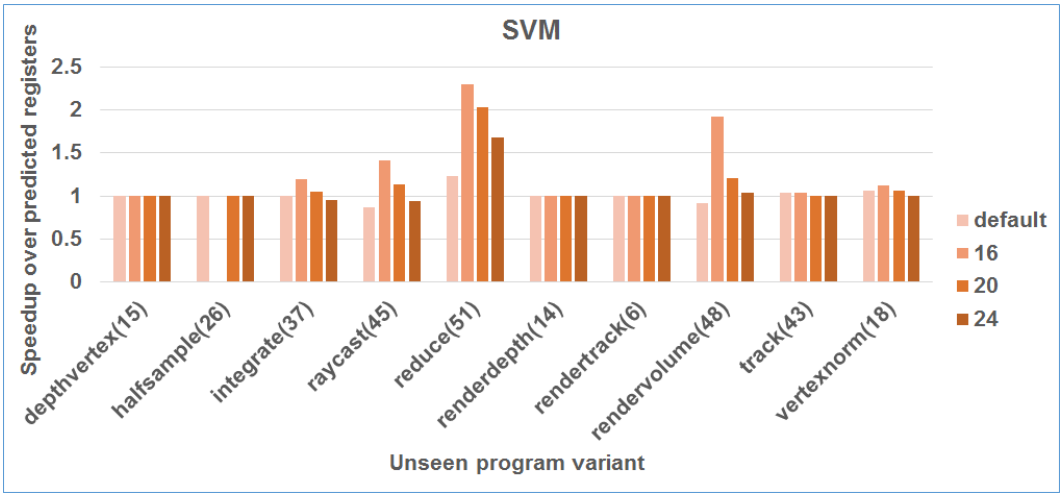


Figure VII.53: Speedup over predicted registers: Kepler programs on Kepler models

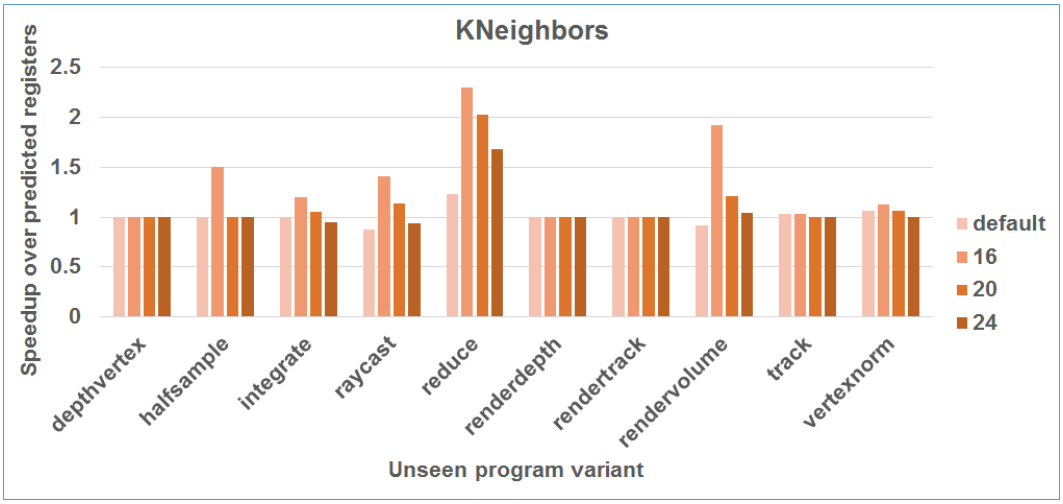


Figure VII.54: Speedup over predicted registers: Kepler programs on Kepler models

Power Gain in Different ML Models

### In order Model Invocation

If we observe all the figures, we see that usage of number of registers have no impact in power. It shows equal or no improvement. For example, figure VII.55 shows that all of the application have mostly same power gain.

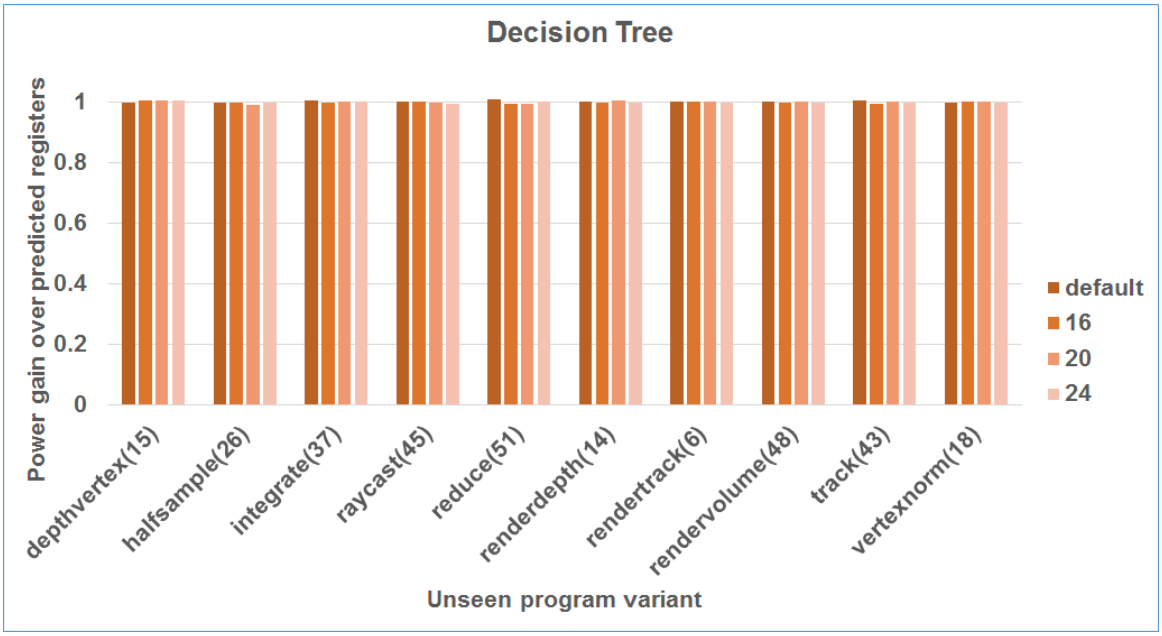


Figure VII.55: Power gain over predicted registers: Kepler programs on Kepler models

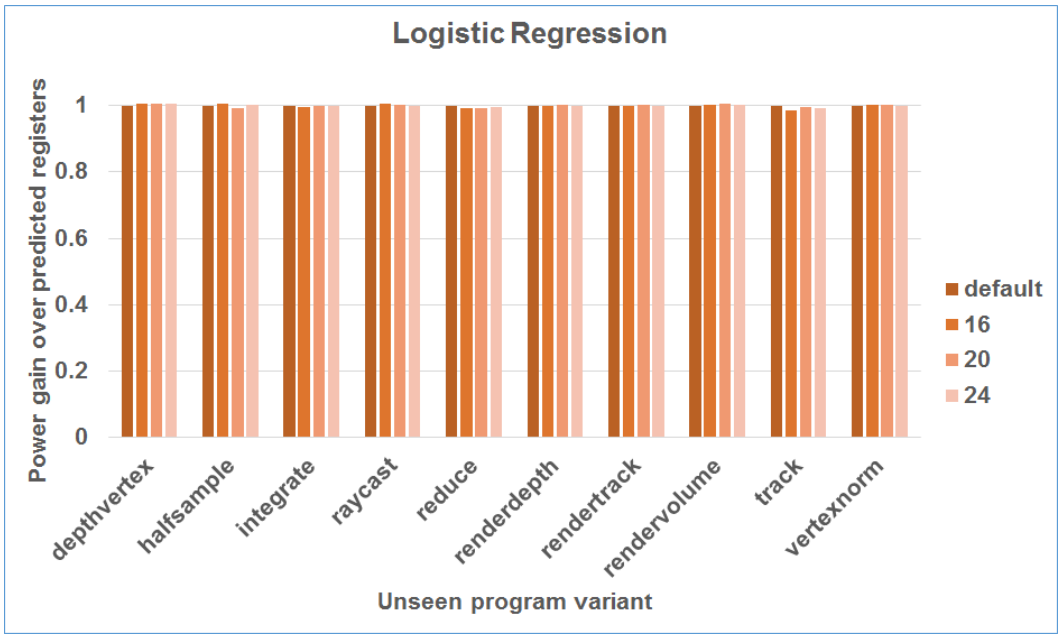


Figure VII.56: Power gain over predicted registers: Kepler programs on Kepler models

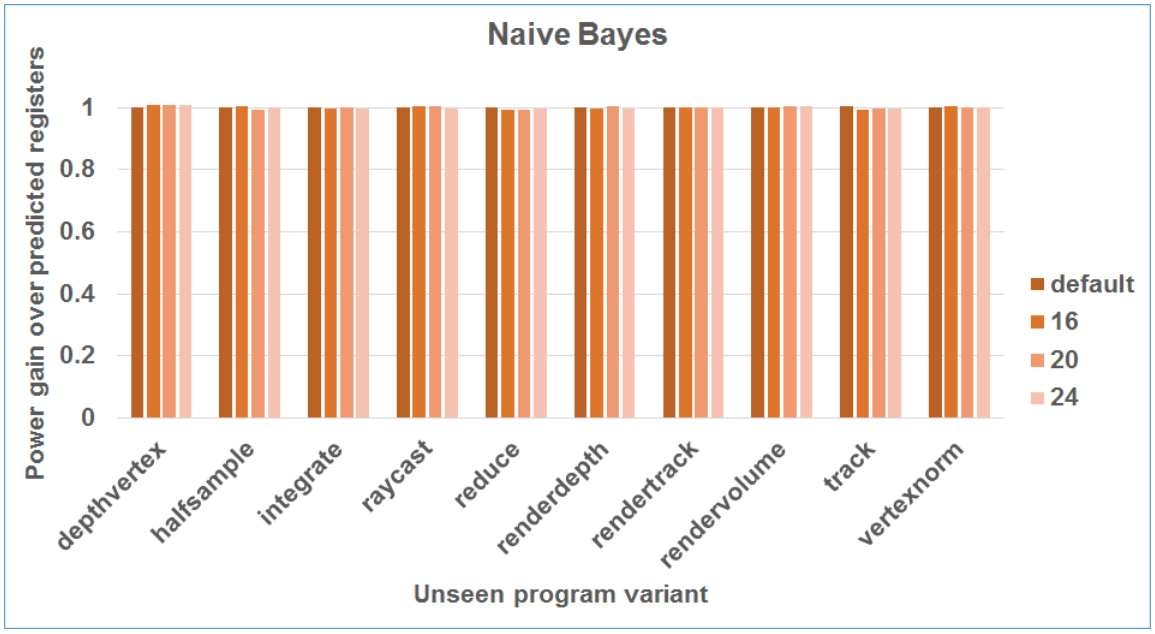


Figure VII.57: Power gain over predicted registers: Kepler programs on Kepler models

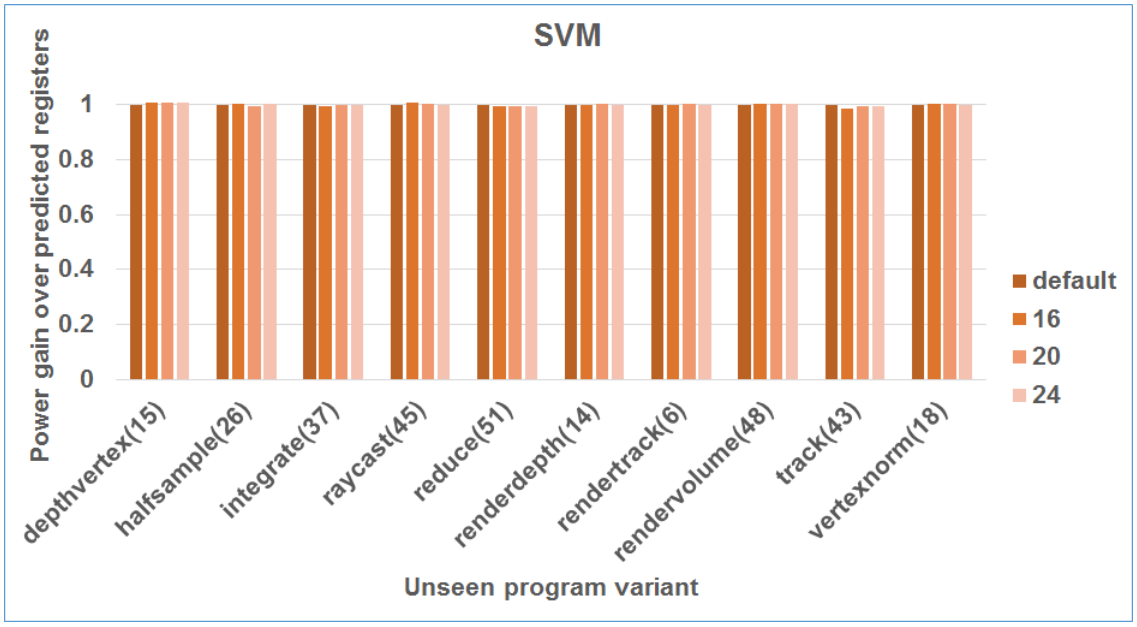


Figure VII.58: Power gain over predicted registers: Kepler programs on Kepler models

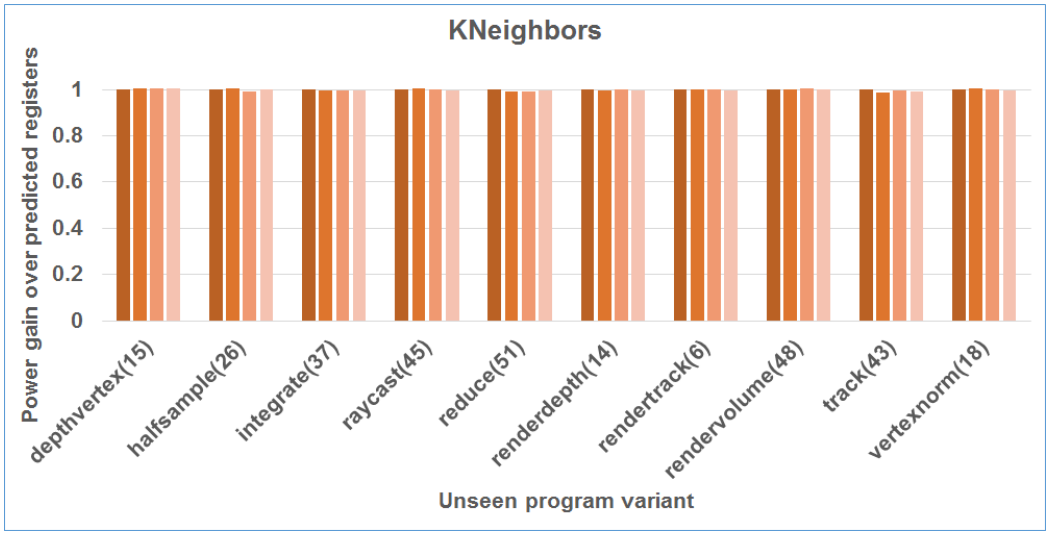


Figure VII.59: Power gain over predicted registers: Kepler programs on Kepler models

### Reverse order Model Invocation

We also see the same pattern in reverse order model invocation as we see in forward order model invocation. If we observe the figures VII.60, VII.61, VII.62, VII.63 and VII.64, we see that there is no improvement in power if we run the program with the predicted registers. For example, figure VII.60 shows that, track application with the predicted registers take more power.

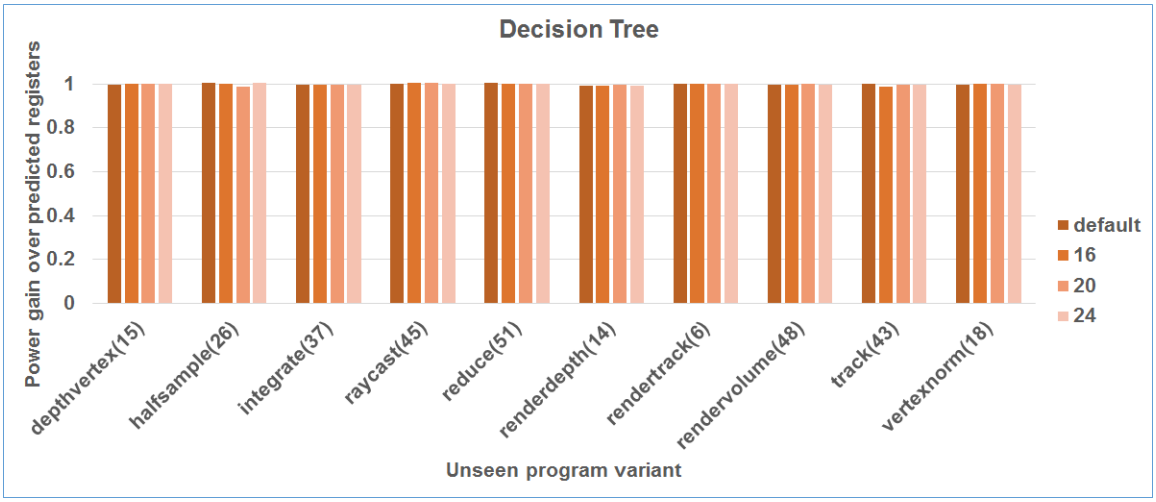


Figure VII.60: Power gain over predicted registers: Kepler programs on Kepler models

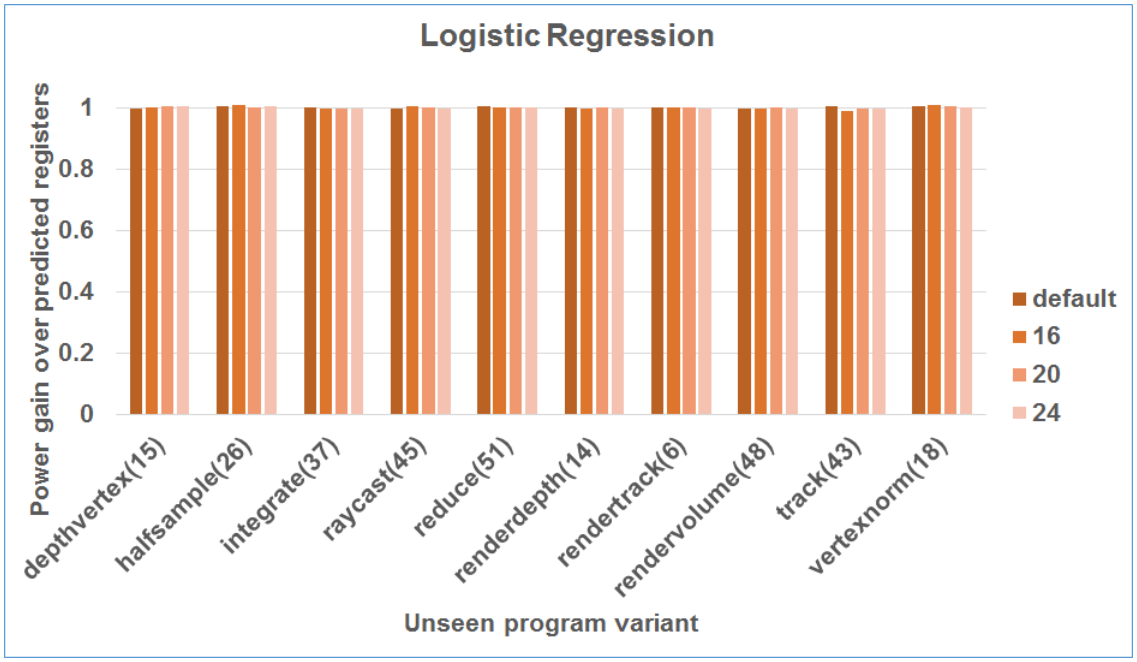


Figure VII.61: Power gain over predicted registers: Kepler programs on Kepler models

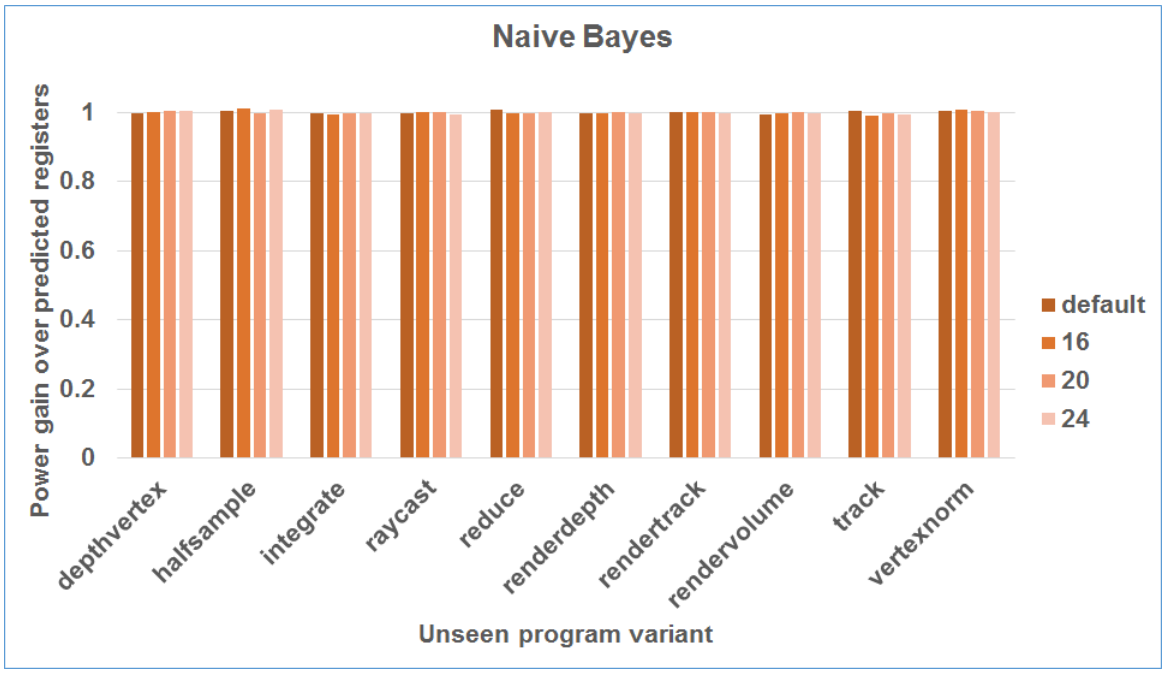


Figure VII.62: Power gain over predicted registers: Kepler programs on Kepler models

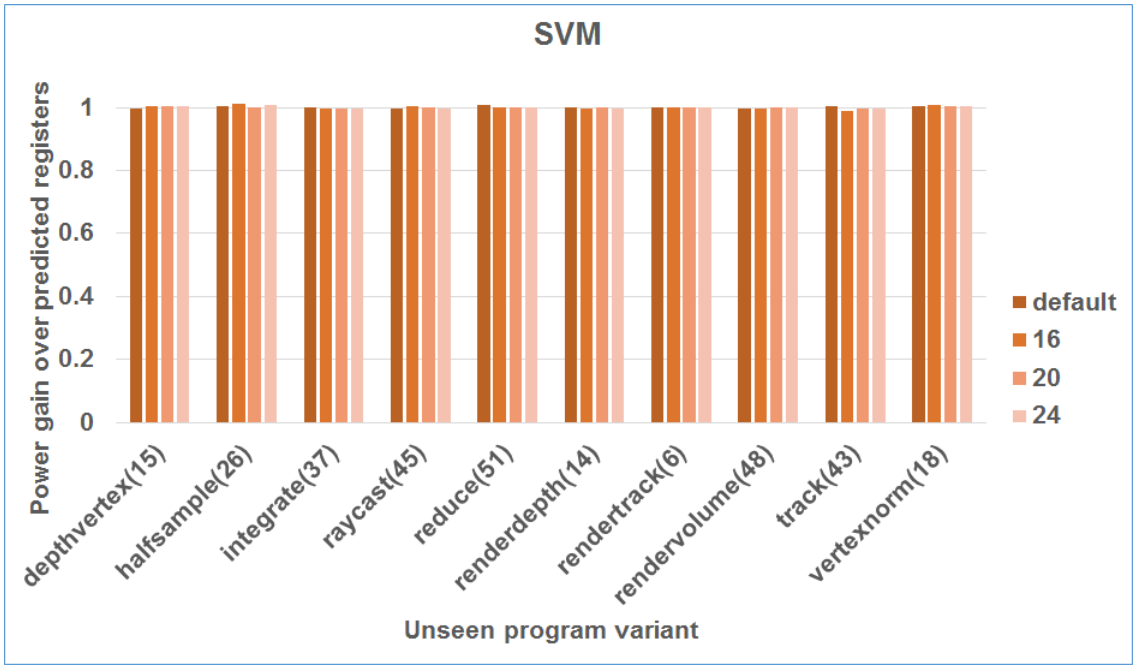


Figure VII.63: Power gain over predicted registers: Kepler programs on Kepler models

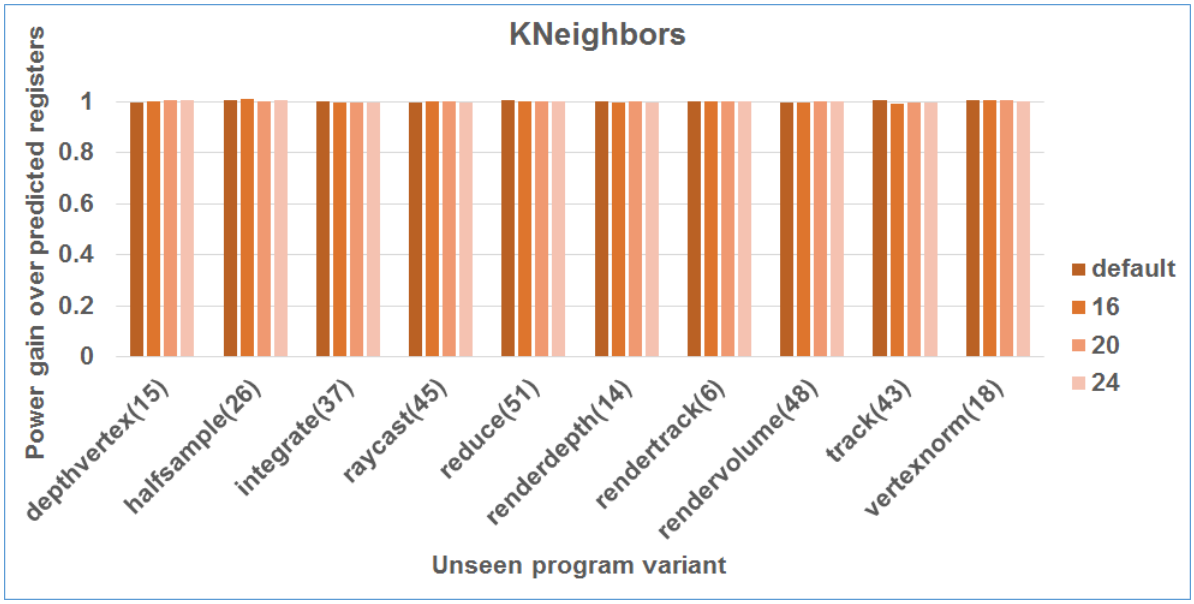


Figure VII.64: Power gain over predicted registers: Kepler programs on Kepler models

# VIII. CONCLUSION

This section summarizes the contribution and the work done in this thesis, and indicates the direction this work can take in the future.

## Contribution

This thesis presents a tool, MLTUNE, which can automatically build a model by extracting the performance counters on the target platform. I have used parboil benchmark to generate training data which is needed to extract performance event values to train Machine Learning Models. I have analyzed different feature selection techniques and how prediction accuracy differs over feature selection. I have looked into the insight of the performance counters using Principal component analysis with varimax rotation. I have analyzed GPU thread configuration with different ML models. I also analyzed cross platform model accuracy in kepler and fermi.

The thesis shows that the number of features have a substantial impact in prediction accuracy of the model. Different feature selection techniques show different prediction accuracy, so its not possible to build a model with good prediction accuracy for a fixed set of features. We see that program variant shows performance improvement in terms of execution time with the predicted registers, which makes it easy for the programmer or performance engineer to optimize the application or program performance before deploying into live system. MLTUNE framework also show the internal mechanism of feature importance through decision tree. The thesis also see that varying number of registers have less impact in power gain. Overall, the thesis show the importance of an automated machine learning based performance tool which can exhaustively employs and verifies the internals of machine learning utilities as an abstraction to the user and easy use of such tools in application.

## Future Work

There are several directions I would like to extend this work. First, I would like to do automatic parameter tuning to generate complex models, for example tuning learning parameter for logistic regression. Second, I would like to check cross platform model accuracy, for example, how the speedup/power gain varies if we test the program on the models trained on different platform. Third, I would like to check the applicability of MLTUNE in general purposes such as to predict stock price. I believe MLTUNE can be used as a generalized tool for many other purposes as this framework support manual and automatic model generation.

## REFERENCES


Agakov, F., Bonilla, E., Cavazos, J., Franke, B., Fursin, G., O'Boyle, M., Thomson, J., Toussaint, M., and Williams, C. (2006). Using machine learning to focus iterative optimization. In *International Symposium on Code Generation and Optimization, 2006. (CGO 2006).*, New York, NY.

Cavazos, J., Fursin, G., Agakov, F., Bonilla, E., O'Boyle, M. F. P., and Temam, O. (2007). Rapidly Selecting Good Compiler Optimizations using Performance Counters. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '07)*, pages 185–197, Washington, DC, USA. IEEE Computer Society.

Cavazos, J. and Moss, J. E. B. (2004). Inducing heuristics to decide whether to schedule. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*, pages 183–194.

Cochran, R., Hankendi, C., Coskun, A. K., and Reda, S. (2011). Pack & cap: Adaptive DVFS and thread packing under power caps. In *MICRO*, pages 175–185.

Curtis-Maury, M., Shah, A., Blagojevic, F., Nikolopoulos, D. S., de Supinski, B. R., and Schulz, M. (2008). Prediction models for multi-dimensional power-performance optimization on many cores. In *Proc. of the 17th international conference on Parallel architectures and compilation techniques*.

Ding, Y., Ansel, J., Veeramachaneni, K., Shen, X., O'Reilly, U.-M., and Amarasinghe, S. (2015). Autotuning algorithmic choice for input sensitivity. In *PLDI*, pages 379–390.

Emani, M. K. and O'Boyle, M. (2015). Celebrating diversity: A mixture of experts approach for runtime mapping in dynamic environments. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 499–508, New York, NY, USA. ACM.

Fursin, G., Kashnikov, Y., Memon, A. W., Chamski, Z., Temam, O., Namolaru, M., Yom-Tov, E., Mendelson, B., Zaks, A., Courtois, E., Bodin, F., Barnard, P., Ashton, E., Bonilla, E., Thomson, J., Williams, C., and O'Boyle, M. (2011). Milepost GCC: Machine Learning Enabled Self-Tuning Compiler. *International Journal of Parallel Programming*, 39.

Ge, Y. and Qiu, Q. (2011). Dynamic thermal management for multimedia applications using machine learning. In *Proceedings of the 48th Design Automation Conference*, DAC '11, pages 95–100, New York, NY, USA. ACM.

Jain, N., Bhatele, A., Robson, M. P., Gamblin, T., and Kale, L. V. (2013). Predicting application performance using supervised learning on communication features. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 95:1–95:12, New York, NY, USA. ACM.

Jayasena, S., Amarasinghe, S., Abeyweera, A., Amarasinghe, G., De Silva, H., Rathnayake, S., Meng, X., and Liu, Y. (2013). Detection of false sharing using machine learning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 30:1–30:9, New York, NY, USA. ACM.

Kashnikov, Y., Beyler, J. C., and Jalby, W. (2012). Compiler optimizations: Machine learning versus O3. In *Languages and Compilers for Parallel Computing, 25th International Workshop, LCPC 2012, Tokyo, Japan, September 11-13, 2012, Revised Selected Papers*, pages 32–45.

Kulkarni, S. and Cavazos, J. (2012). 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*, OOPSLA '12, pages 147–162.

Leather, H., Bonilla, E. V., and O'Boyle, M. F. P. (2014). Automatic feature generation for machine learning-based optimising compilation. *TACO*, 11(1):14.

Liao, S.-w., Hung, T.-H., Nguyen, D., Chou, C., Tu, C., and Zhou, H. (2009). Machine Learning-Based Prefetch Optimization for Data Center Applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 56:1–56:10.

Moore, R. W. and Childers, B. R. (2013). Automatic generation of program affinity policies using machine learning. In *Proceedings of the 22Nd International Conference on Compiler Construction*, CC'13, pages 184–203, Berlin, Heidelberg. Springer-Verlag.

Pusukuri, K. K., Vengerov, D., Fedorova, A., and Kalogeraki, V. (2011). Fact: A framework for adaptive contention-aware thread migrations. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, pages 35:1–35:10, New York, NY, USA. ACM.

Rahman, S., Burtscher, M., Zong, Z., and Qasem, A. (2015). Maximizing hardware prefetch effectiveness with machine learning. In *17th IEEE International Conference on High Performance Computing and Communications (HPCC15)*.

Seo, S., Lee, J., Jo, G., and Lee, J. (2013). Automatic opencl work-group size selection for multicore cpus. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*.

Shen, H., Lu, J., and Qiu, Q. (2012). Learning based dvfs for simultaneous temperature, performance and energy management. In *Quality Electronic Design (ISQED), 2012 13th International Symposium on*, pages 747–754.

Stephenson, M. and Amarasinghe, S. (2005). Predicting Unroll Factors Using Supervised Classification. In *CGO*, San Jose, CA, USA.

Stock, K., Pouchet, L.-N., and Sadayappan, P. (2012). Using Machine Learning to Improve Automatic Vectorization. *ACM Trans. Archit. Code Optim.*, 8(4):50:1–50:23.

Tournavitis, G., Wang, Z., Franke, B., and O'Boyle, M. F. (2009). Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*.

Vuduc, R., Demmel, J., and Bilmes, J. (2004). Statistical Models for Empirical Search-Based Performance Tuning. *International Journal of High Performance Computing Applications*, 18(1):65–94.

Williams, S., Waterman, A., and Patterson, D. (2009). Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52.