A Machine Learning Approach to Automatic Creation of Architecture-sensitive Performance Heuristics

Biplab Kumar Saha
Texas State University
San Marcos, TX
Email: biplabks@txstate.edu

Tiffany A. Connors
Texas State University
San Marcos, TX
Email: tac115@txstate.edu

Saami Rahman
Texas State University
San Marcos, TX
Email: saamirahman@gmail.com

Apan Qasem
Texas State University
San Marcos, TX
Email: apan@txstate.edu

***Abstract*—Recent interest in machine-learning based methods has produced many sophisticated models for performance modeling and optimization. These models tend to be sensitive to parameters of the underlying architecture and hence yield the highest prediction accuracy when trained on the target platform. Training a classifier, 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 new framework for automatically generating machine-learning based performance models. A tool-chain is developed that provides automated mechanisms for sample generation, dynamic feature extraction, feature selection, data labeling, validation and hyper parameter tuning. We describe the design and implementation of this system and demonstrate its efficacy by developing a learning heuristic for register allocation in GPU kernels. Results show that auto-generated models can predict register thresholds that lead to integer factor performance improvements over kernels produced by state-of-the-art optimizing compilers.**

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 [10], [19], data centers [14], desktop [1] and mobile computing [8]. 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. First is the availability of large code bases in open software repositories such as GitHub and the second is the increased number of exposed hardware performance counters on newer processor architectures [9]. Both imply greater access to pertinent data, creating new opportunities for learning application behavior on future architectures.

In spite of its success and promise, a key limitation in MLMT has been its 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 [7], runtime system [6] or autotuner [5], before being shipped to the end-user. This practice is adopted for two reasons. First, model training is a time 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' capabilities. Since program behavior is intimately tied to characteristics of the target architecture, prediction accuracy is highly sensitive to variations in parameters of the 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. This makes it imperative that the learning occur in the target environment.

This paper presents a new approach to building performance classifiers to address this limitation. The proposed framework consists of a language interface for MLMT model specification, a benchmark repository and an extensible software tool-chain for performing major tasks in the ML workflow. In addition, plug-ins allow integration of open-source ML libraries and crowd contribution to the benchmark repository. Given the description of a desired learning outcome, the system can automatically generate supervised classifiers for a new target platform for a large subclass of MLMT problems. All major steps in an ML workflow are automated, including feature extraction, feature selection (FS), labeling, validation, hyper-parameter tuning and the very challenging problem of sample generation and training. 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 common elements can be found when we specifically look at the construction of performance models. For instance, a requirement in MLMT is that feature

[1]although abbreviated MLMT, it includes ML applied to resource allocation, compiler optimization and other performance-enhancement strategies

values be comparable across two different program instances. One way to achieve this is to normalize each feature value with respect to the execution time of the program in favor of a standard normalization technique. Similar commonalities can be found in feature extraction, feature selection and training data evaluation.

In our system, training data is generated on the target platform for each new instance of a model. This results in heuristics that are customized for a specific execution environment which addresses the portability concern. A key challenge in training data generation, automatically or otherwise, is to be able to generate a sufficiently large sample from a relatively small number of programs. We tackle this problem by leveraging the machinery of an autotuning system. From the same base program, the autotuner generates multiple variants that exhibit *distinct* behavior in the target environment with respect to the particular objective that is being modeled.

Another issue that is implicitly addressed by this work is the black-box treatment of ML models. MLMT models have mostly been *predictive* rather than *descriptive*. In the few instances, where a descriptive model has been used [10], [16], its descriptive properties have not been exploited. As a consequence we have gained little insight about application behavior from the many excellent heuristics that have been developed and the large volume of training data that has been generated. To address the issue of opaque models, we incorporate in the framework, analyses to expose the internals of the learning algorithm and the salient properties of the generated training data.

We demonstrate the utility of our system by deriving a heuristic for predicting the number of registers to be allocated to a GPU kernel. Based on execution time feedback, the model makes recommendations that improve the performance of kernels that suffer from high register pressure. We analyze the feature space and the learned heuristic to gain insight into performance problems caused by ineffective use of the register space. The analysis also provides guidance for developing effective strategies for MLMT model construction.

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

- a tool-chain for automating MLMT workflow and generating platform-specific performance heuristics
- a novel autotuning-based approach for generating representative sample instances from a small number of seed applications
- analyses and visualization techniques to gain performance insight from generated MLMT models
- an ML heuristic to determine the number of registers to be allocated to a GPU kernel

II. Related Work

A study of recent applications of machine-learning techniques in performance modeling and tuning in HPC shows a pattern of incoming challenges and how ML practitioners have tackled them. 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 based on using heuristic modeling, pruning and empirical search in order to reduce the parameter space and find early stopping criteria [20]. As neural networks and logistic regression models gained popularity, they were applied to autotuning problems in HPC. Cavazos *et al.* led the charge in this venture beginning with their work on identifying optimal compiler optimization sequences using multiple logistic regression models [1]. Estimating the performance gain or loss of applying a particular optimization as a reduction of the larger problem of finding an optimal set of optimizations worked well for a multitude of scenarios. This technique, however, overlooks the possibility of synergistic and antagonistic behavior between multiple optimizations. Moreover, as the number of optimizations available remains large, the time to generate training data and the number of classifiers required also remains large. For instance, GCC 4.8.2 has 193 optimizations and choosing an optimal sequence essentially means creating an array of 193 classifiers and training data sets for each classifier. Furthermore, the widely changing architectures in HPC landscape has posed the challenge of adaptability. Fursin *et al.* turned to crowd-sourcing to address this challenge by gathering collective optimization knowledge across architectures [7].

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 [4], [15], static program features [7] and hardware performance counters [1], [17]. Hardware performance counters have the added benefit of being dynamic and are able to capture architecture-specific system response. However, there is a large number of performance events and it is difficult to pick effective ones. Many have resorted to hand-picking them [18], while some have employed statistical methods to select events that vary most across different program executions [11], [17].

In spite of challenges faced by HPC researchers in their application of ML, the evolution of ML in HPC has been impressive. Many variants of popular ML techniques have been successfully applied to different branches of HPC - in performance optimization through code changes [3], [18], predicting optimal build configurations [12], runtime configurations [6], [14], identifying performance bottlenecks [10], [11] and recently, also in efficient energy management [2], [8].

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 been the primary motivation behind this work.

III. Conventional ML vs MLMT Workflow

Fig. 2 outlines a typical ML workflow that may be used in scientific or social studies. The unique aspects of the MLMT

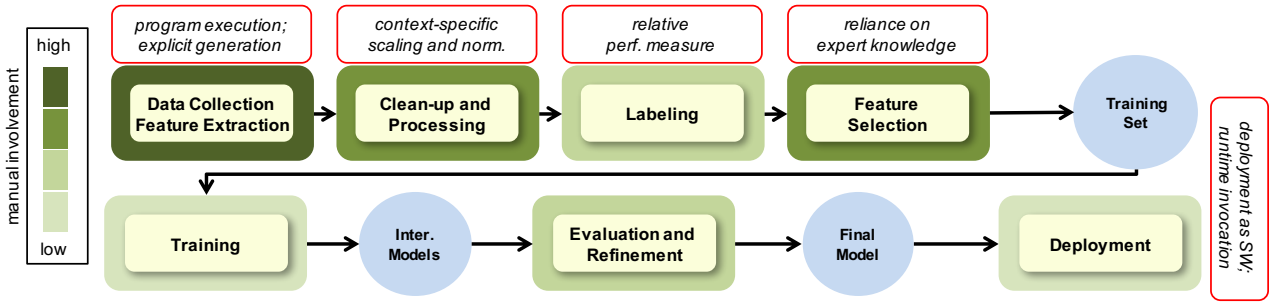


Fig. 1. Conventional ML and MLMT workflow

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 phase (e.g., gene sequencing). In MLMT, training data needs to be explicitly generated for each new model that is to be created. Training data for optimization X, is unlikely to be useful for 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. Furthermore, because data needs to be collected explicitly in many cases, there is often 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 an attribute but on how it affects 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 capture performance snapshot per unit time across different programs.

Labeling - Unlike other domains, unlabeled data is not a major problem in MLMT since domain expertise is not required to label training instances. All that is needed is a means to measure relative performance of the unoptimized and optimized versions of workloads. The exceptions 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 FS 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, system 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 [13].

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 decisions at runtime. This implies that model invocation must have minimal overhead and 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., disparity in objective metrics) and (iii) are relatable to practitioners (e.g., representation of programs). In this section, we describe the 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 in the remainder of the paper.

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

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 directive. Multiple decisions can be combined to create a *composite decision* $D = \{d_0, ..., d_n\}$. For instance, predicting a compiler optimization sequence involves composing a series of atomic decisions involving the application of an individual optimization.

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 *decision* d to v. Application of a sequence of k decisions produces a new variant shown by $D(v) \rightarrow v_D$, where $D = \{d_0, ..., d_n\}$.

Environment: The execution platform in which a *variant* 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 a performance metric such as throughput or energy and must be readily measurable in the execution environment. We use $T(v)$ to denote an objective metric measured while executing *variant* v.

We can use the above abstractions to express the objective of any MLMT model as follows

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 an unseen variant

To reach this objective, the model needs to pose a series of queries of the form "*What happens to* variant v *with respect to* target T *when* decision d *is taken?*" To provide the answer to these queries, training data needs to be compiled with instances of the form $I = \{f_0, f_1, ..., f_n, l\}$, where $\{f_0, f_1, ..., f_n\}$ are feature values collected for some *variant* v during execution in *environment* E in the absence of *decision* d. l is a *label* that captures the effects on *target* T when *decision* d is applied to *variant* v in the same execution environment. When the goal is to minimize an objective, *label* l can be derived from the ratio $T(v)/T(d(v))$. For instance, for execution time this is simply the speedup obtained when applying the decision to the variant. A ratio of > 1 implies a positive effect of d while < 1 implies a negative effect and this forms the basis for classification.

Let S describe an MLMT model, then model construction and invocation can be concisely expressed as follows

$$\begin{aligned} gen\_train\_data(S) &\rightarrow \{I\} \\ train(\{I\}) &\rightarrow M_T^E \\ invoke(M_T^E, fv\prime) &\rightarrow \{D\} \end{aligned} \quad (1)$$

Given this formulation, we observe that automatic model construction is essentially reduced to (i) constructing S and (ii) defining $gen\_train\_data()$. We explain how these tasks are carried out in our framework in Section V.

V. Design and Implementation

Fig. 2 gives an overview of our framework. Based on a concise specification of an abstract model, a set of custom scripts are generated for the target platform on which the model is to be developed. These scripts drive the tasks of feature extraction, feature selection, training data generation, model training, evaluation, and selection. The newly created model is stored as a Python script or a Java class file, depending on the ML engine used, and provides an interface for the user to invoke it on unseen programs. An interactive mode is also supported to perform subtasks selectively. The framework, including the benchmark repository, is available for download at `https://github.com/biplabks/MLTUNE`.

A. MLSPEC

MLSPEC is a simple language interface to allow users to fully describe an abstract MLMT model, as defined in Section IV. A sample specification is presented in Table I.

TABLE I
SAMPLE MLSPEC INPUT FILE

```
# user training programs? : Y
# path to user programs: /path/to/user_programs
# training time: relaxed
# rule 0
# trivial statement
;; nvcc $v.cu -o $v.o;
nvcc main.o -o $v
# outcome statement
;; nvcc $v.cu --maxrregcount=$d0 -o $v.o;
nvcc main.o -o $v
# rule 1
;; nvcc $v.cu -o $v.o;
nvcc main.o -o $v
;; nvcc $v.cu --maxrregcount=$d1 -o $v.o;
nvcc main.o -o $v
```

An MLMT specification is comprised of two sections: a set of parameter values that control different aspects of model construction followed by a sequence of *rule* blocks. A rule block is an example of how a *decision*, *d* is applied to a *variant*, *v*. A rule consists of a pair of *action statements* called the *trivial* and *outcome* statements. The trivial statement describes the sequence of actions needed to build and/or execute *v* in the absence of *d*. The outcome statement lists the actions for *d* to be applied. Actions can be either *build* or *execute* commands. A build command denotes that *d* is taken at some stage prior to execution (e.g., source-to-source transformation) while an execute command denotes that *d* is taken at runtime (e.g., resource allocation). One rule must be specified for each $d_i \in D$.

The MLSPEC parser processes the specification file and generates custom scripts for training data generation. For each rule, the tool calculates the *difference* between the trivial and outcome statements and uses this information to generate build and execute scripts for each program in the benchmark repository.

B. AutoTrainer

Following the creation of custom makefiles and execute scripts, the system generates an *actionlist* file which encapsulates necessary information for collecting training data on the target platform. An *actionlist* is a series of instructions, where each instruction takes the form of an *action statement*, with the variant placeholder replaced with actual program ids from the repository. There is one action statement for each *(v,d)* pair. Feature values are extracted when the *trivial* action is taken. Performance metrics are collected for both *trivial* and *output* statements.

We re-purpose an autotuner to perform the task of variant generation. Autotuners expose control knobs at different layers and provide a mechanism to manipulate these knobs to create multiple variants from the same base program. The resulting variants can exhibit widely varying behavior and thus can represent independent instances within the training data set. We take advantage of this and use the autotuner to generate *performance-distinct* code variants from each program in the repository. A subset of the controls knobs are listed in Table II. Not all parameters are applicable to every program and the

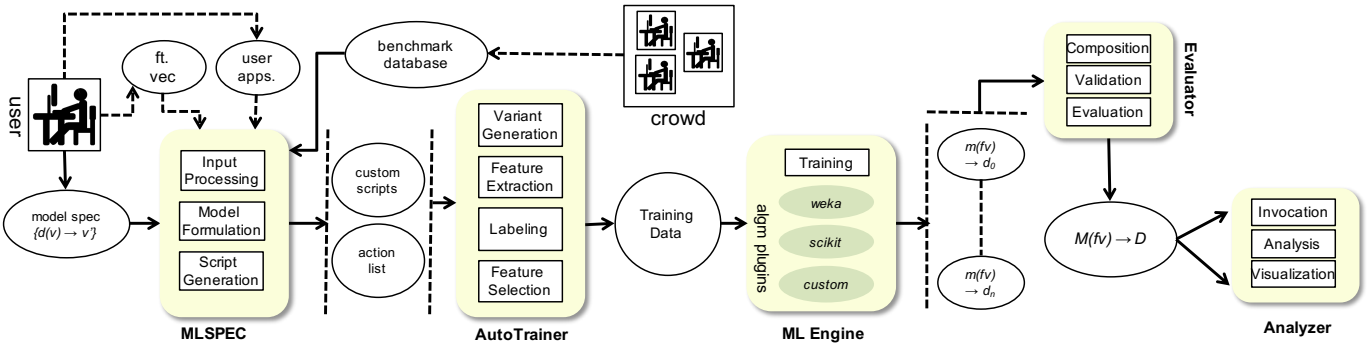


Fig. 2. MLMT framework overview

TABLE II
CONTROL KNOBS FOR VARIANT GENERATION

level	control knob	example value
source	tiling	tile sizes
source	interchange	loop combination
source	unrolling	unroll factors
compile	optimization level	O1, O2, O3, O4
compile	optimization flag	inline, prefetch etc.
runtime	thread affinity	consolidate, disperse
runtime	thread geometry	block and grid size

TABLE III
MLMT FORMULATION OF RECAP

D	composite decision; what is the register cap that most improves T? $D = \{d_i \mid 16 \leq i \leq 512 \;\&\&\; i \bmod 8 = 0\}$
d_i	elementary decision; how much improvement is expected in T when i is set as the cap?
v	CUDA executable kernel
fv	all runtime performance *metrics* and *events* measurable in E
E	Nvidia GPU based on *Fermi*, *Kepler* or *Maxwell* architecture; CUDA RT 7.0; `nvcc` 7.0
T	kernel execution time (T_t), power consumption (T_P); energy ($T_E = T_t \times T_p$)
I	training instance; $\{f_0, f_1, ..., f_n, L\}$, $n_{Fermi} = 382, n_{kepler} = 294, n_{maxwell} = 480$
f_i	i^{th} feature value; measured, scaled and normalized runtime event
L	speedup $T(d_i(v))/T(v)$; powerup and greenup computed analogously

range of values for parameters is application and platform dependent. The minimum number of alternate variants for a program in the database is 16 and the maximum is 256. Each time a variant is generated, a similarity check on the feature vector is performed to discard instances that are too similar.

VI. EVALUATION

A. ReCAP: A Model for Capping Register Allocation

Using our framework, we derived a model that recommends a cap for the number of registers to be allocated to a CUDA kernel to maximize performance and/or energy efficiency. ReCAP is mapped to (1) as shown in Table III.

We evaluate the quality of auto-generated ReCAP models in terms of prediction accuracy, portability and performance

TABLE IV
CLASSIFIER PREDICTION ACCURACY ON *Kepler*

ML Models	Prediction Accuracy	Precision	Recall	F1-Score
	Scikit-learn			
Logit	85.13	0.72	0.56	0.63
Naive Bayes	82.05	0.58	0.52	0.54
SVM	82.84	0.67	0.50	0.57
Decision Tree	80.92	0.53	0.71	0.60
MOE	**93.84**	**0.96**	**0.83**	**0.83**
	Weka			
Logit	72.50	0.31	0.52	0.37
Naive Bayes	78.87	0.56	0.67	0.61
SVM	91.60	0.71	0.59	0.64
Decision Tree	90.96	0.75	0.67	0.71
MOE	**95.77**	**0.96**	**0.92**	**0.90**

improvement on unseen kernels. We train and evaluate the models on three Nvidia GPUs from three different generations, (i) Tesla C2050 (*Fermi*) (ii) Tesla K20 (*Kepler*) and (iii) GTX Titan X (*Maxwell*).

B. Prediction Quality

Summary evaluation statistics for ReCAP derived from four classifiers in *Scikit* and *Weka* are presented in Table IV. Reported prediction accuracy is for 10-fold cross-validation and is averaged over all d_i. These numbers are comparable, and in many cases superior, to numbers that have been reported in previous MLMT work [16], [18], [19]. Notwithstanding, we observe that not all classifiers achieve the same level of accuracy. In analyzing training sets for individual d_i, we observed a greater disparity in accuracy. For instance, for *logit* from *Weka*, although average prediction accuracy is 72.5%, on individual d_i datasets, the accuracy ranged from 63.2% to 94.1%. The highly disparate accuracy levels reiterate the need for an MOE approach for MLMT, as adopted in our framework. With MOE our system selects the best classifier for each individual d_i and achieves higher overall accuracy levels of 93.84% and 95.77% for *Scikit* and *Weka*, respectively.

C. Portability

By allowing users to generate training data on the target platform, our system can produce *implicitly portable* performance heuristics. To demonstrate the significance of target-specific models, we tested each model on non-native data.

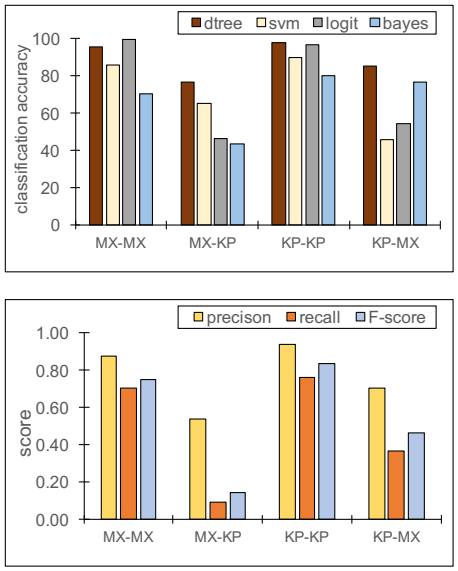


Fig. 3. Cross-platform accuracy of four classifiers. *X-Y* indicates model trained on *X* and tested on *Y*

That is, model trained on platform X is evaluated with test data generated on Y. For example, the *Maxwell* model is asked to make predictions for each instance in the training data of *Kepler* and *vice versa*. Classification accuracy is computed by taking the ratio of correctly predicted instances over total instances (since cross-validation is not applicable).

Figs. 3 shows the results of these experiments. The accuracy drops dramatically when models are asked to make cross-platform predictions. All four classifiers show a significant decline in accuracy with *logit* being affected the most. *Maxwell* models suffer a 30% decrease in accuracy, while *Kepler* models take a 25% hit. It is interesting to note that although there have been substantial changes in the *Maxwell* architecture, the register capacity and register to DRAM datapath remains unaffected (unlike the transition from *Fermi* to *Kepler*). We found that when *Maxwell* models made incorrect predictions, they typically recommended a lower register cap than one that would lead to better performance. This is evidenced by the low recall value of `MX-KP`. The low accuracy values make cross-platform models essentially invalid and clearly demonstrate the need for native training, which our system provides.

D. Performance Gains

Although performance and energy improvements were not the primary focus of this work, we did want to evaluate how the model fared in this area. From our analysis of the training data we found that kernels with *low* register pressure are insensitive to changes in the register cap. Hence, for performance experiments, we choose 12 kernels from Parboil, SLAM and Lonestar GPU suites with relatively high register pressure (¿ 30). Each kernel is run through the AutoTrainer to create four alternate variants with different register pressure and thread geometry, for a total of 48 test instances. Each instance is presented to ReCAP to obtain a prediction and then compiled and executed twice[2], once with default and once with predicted register cap. We then compute speedup over default cap.

Fig. 4 shows the performance improvements obtained from ReCAP. Not surprisingly, we find a direct correlation between the amount of performance gained via ReCAP models and the register pressure of the kernel. `sad` and `reduce`, the two kernels with the highest register pressure (44 and 51, respectively) yield the best speedups. Interestingly, however, ReCAP did not always recommend an increase in register cap. In several cases, lowering the register cap led to improved performance. On further analysis, we found that for certain kernels (e.g., `raycast`) when register cap is lowered, *nvcc* is able to generate code with more efficient use of shared memory.

We observe that in no instance the ReCAP predictions lead to performance degradation. This is a direct result of constructing conservative models (as explained in Section V). By creating a wider band of *neutral* values, we ensure that the model recommends a change only when there is a likelihood of significant (¿ 1.05 for ReCAP models) performance gains. Among the 48 test instances, ReCAP predicted a no-change cap in 13 cases.

VII. Analysis

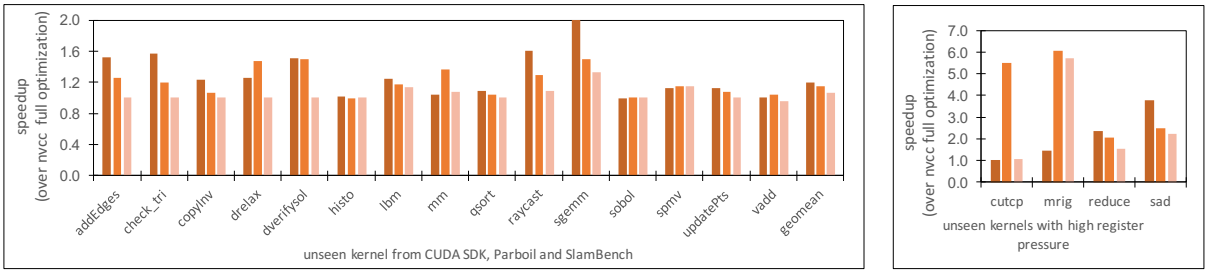
In this section, we utilize the tools available in our MLMT framework to take a more detailed look at the feature space and the generated models. The goals of this analysis is two-fold (i) gain insight into register-pressure related performance issues and improve our understanding of effective techniques for MLMT construction.

Training data for ReCAP was partitioned into eight sets to produce the eight d_i models. Each partition and model was analyzed separately. In the interest of space, we present detailed results for a subset of these training sets and models and provide summary statistics when appropriate.

A. Sample Generation

Our framework provides a mechanism to create many sample instances from the same program. To determine the utility of such an approach we analyze the training data set using *cluster-PCA bi-plots*. Fig. 5 shows a cluster-PCA plot for d_0. In this instance, the points, which represent code variants, are annotated with the base program from which they were derived. We see that for some base kernels, the variants are concentrated in the same cluster while for others they are distributed across several. For instance, $\oplus$ variants (`tacpf` from Parboil) appear in five different clusters. This distribution pattern suggests that variants derived from the same base program can exhibit very different performance characteristics and therefore can add useful information to the training set. This provides justification for an autotuner-driven sample generation approach.

[2]executions are repeated five times and only the average is considered



(a) moderate-to-low register pressure (=< 32)

(b) high register pressure (> 32)

Fig. 4. Performance improvement with RML prediction on *unseen* kernels. Speedup reported over kernels fully optimized by `nvcc`

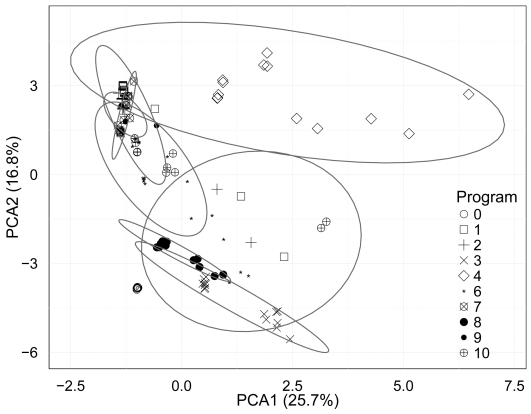


Fig. 5. Cluster-PCA plot visualizing distribution of code variants in d_0

B. Performance Insight

Analysis of the feature space also provide insight about register usage and its impact on performance. Fig. 6 shows a varimax rotation PCA segment plot for d_1 with four PCs and 20 contributing features. A VR-PCA plot is another feature of the visualization module in our framework. To create these plots, we apply Varimax Rotation on the sub-space discovered through PCA and then use a segment plot to visualize the contribution of each feature to the top k PCs. Intuitively, a PC in this context describes an observed performance pattern and collectively the VR-PCA plot gives intuition as to which specific attributes are *causing* this performance variations.

We observe the two most dominant attributes for PC4 are `gst_requested_throughput` and `gst_effective_throughput`. Thus, PC4 clearly indicates a performance problem related to increased register pressure. When register pressure increases to the point where values have to spilled to local memory (which requires global memory access on Nvidia systems), the demand for bandwidth increases. When this demand increases to the point where it cannot be satisfied by the memory subsystem, the effective bandwidth and performance start to drop. This is precisely the pattern that is being captured by PC4. PC3 also captures a similar performance problem with excessive register pressure. But it distinguishes kernels in which a higher fraction of register spill requests are serviced by the L2 cache. This can be inferred by the difference in contribution of L2 related attributes in the two PCs.

PC1 represents issues related to the arithmetic intensity (AI). When AI is high, global memory references due to register spills may not necessarily result in a performance inefficiency. This is because the kernel has sufficient computation to hide the extra latencies created by register spills. PC1 helps the model distinguish programs based on arithmetic intensity.

PC2 is dominated by IPC related features. Since IPC is another measure for performance, it is obviously a good predictor of program performance. In this context, however, it is not particularly useful because we are interested in the cause of the performance problem not the effect. Thus, this PC is an example of a special case of multicolinearity in the training data and should be eliminated in more sophisticated models. Interestingly, attributes related to L1 cache do not appear in any PC. This suggests that register spills only have a significant impact on performance when they are *not* serviced by the L1 cache.

VIII. Conclusions

This paper outlines a path towards automatically generating ML based performance models. First, we establish a set of abstractions that capture the central elements of ML-based performance modeling, and then provide software support for these abstractions to automate the workflow. We further enhance the framework by developing MLMT-specific strategies for automatic sample generation, feature extraction and selection, normalization, and labeling.

We demonstrated the utility of the framework by using it to predict the number of registers to be allocated to a GPU kernel. The constructed models achieve high degrees of prediction accuracy. For kernels with high register pressure ReCAP is able to improve the performance significantly, in some instances achieving integer factor improvements.

The analysis and visualization techniques provided useful insight into the internal decision-making process of the classifiers. We found that reasonable accuracy can be achieved using very disparate feature vectors. We also discovered that, the GPU performance metric `local_replay_overhead` captures the cost of register spills more accurately than other events that are typically used in register pressure analysis.

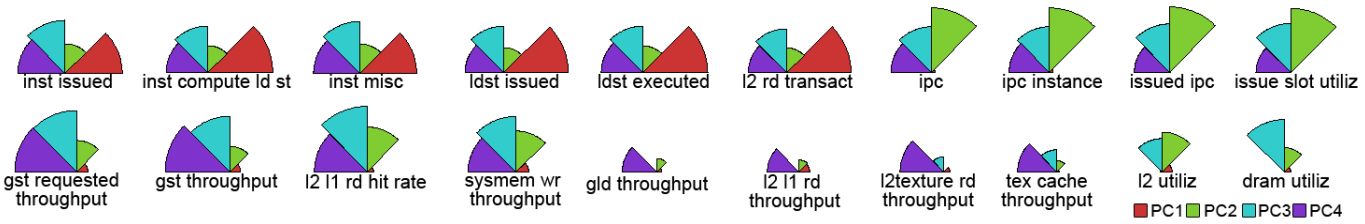


Fig. 6. PCA-VR segment plot showing the contribution of each attribute to four top PCs

We demonstrated that an autotuner can be re-purposed to generate meaningful training data from a relatively small set of programs. The generated data shows good distribution of data instances across feature values and class labels.

We substantiated the notion that cross-platform predictive models are likely to perform poorly by showing that for some classifiers the accuracy can drop by as much as 53%, when trying to make predictions across different generations of GPU architecture, making them essentially invalid.

ACKNOWLEDGEMENT

This work was supported by the National Science Foundation through awards CNS-1305302 and CNS-1253292. Biplab Saha, Saami Rahman and Tiffany Connors contributed to this work while they were students at Texas State University.

REFERENCES

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

[2] R. Cochran, C. Hankendi, A. K. Coskun, and S. Reda, "Pack & cap: Adaptive DVFS and thread packing under power caps," in *MICRO*, 2011, pp. 175–185.

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

[4] J. Demme and S. Sethumadhavan, "Approximate graph clustering for program characterization," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 4, p. 21, 2012.

[5] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U.-M. O'Reilly, and S. Amarasinghe, "Autotuning algorithmic choice for input sensitivity," in *PLDI*, 2015, pp. 379–390.

[6] M. K. Emani and M. O' Boyle, "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*, ser. PLDI 2015. New York, NY, USA: ACM, 2015, pp. 499–508. [Online]. Available: http://doi.acm.org/10.1145/2737924.2737999

[7] G. F. et al., "Milepost GCC: Machine Learning Enabled Self-Tuning Compiler," *International Journal of Parallel Programming*, vol. 39, 2011.

[8] Y. Ge and Q. Qiu, "Dynamic thermal management for multimedia applications using machine learning," in *Proceedings of the 48th Design Automation Conference*, ser. DAC '11. New York, NY, USA: ACM, 2011, pp. 95–100.

[9] Intel, *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corp, 2016.

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

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

[12] Y. Kashnikov, J. C. Beyler, and W. Jalby, "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*, 2012, pp. 32–45.

[13] H. Leather, E. V. Bonilla, and M. F. P. O'Boyle, "Automatic feature generation for machine learning-based optimising compilation," *TACO*, vol. 11, no. 1, p. 14, 2014.

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

[15] E. Park, J. Cavazos, and M. A. Alvarez, "Using graph-based program characterization for predictive modeling," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. ACM, 2012, pp. 196–206.

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

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

[18] M. Stephenson and S. Amarasinghe, "Predicting Unroll Factors Using Supervised Classification," in *CGO*, San Jose, CA, USA, March 2005.

[19] K. Stock, L.-N. Pouchet, and P. Sadayappan, "Using Machine Learning to Improve Automatic Vectorization," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 50:1–50:23, Jan. 2012.

[20] R. Vuduc, J. Demmel, and J. Bilmes, "Statistical Models for Empirical Search-Based Performance Tuning," *International Journal of High Performance Computing Applications*, vol. 18, no. 1, pp. 65–94, 2004.