2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems

# Automatically Selecting Profitable Thread Block Sizes for Accelerated Kernels

Tiffany A. Connors
Computer Science Dept.
Texas State University
San Marcos, TX 78666-4684
Email: tac115@txstate.edu

Apan Qasem
Computer Science Dept.
Texas State University
San Marcos, TX 78666-4684
Email: apan@txstate.edu

***Abstract*—Graphics processing units (GPUs) provide high performance at low power consumption as long as resources are well utilized. Thread block size is one factor in determining a kernel's occupancy, which is a metric for measuring GPU utilization. A general guideline is to find the block size that leads to the highest occupancy. However, many combinations of block and grid sizes can provide highest occupancy, but performance can vary significantly between different configurations. This is because variation in thread structure yields different utilization of hardware resources. Thus, optimizing for occupancy alone is insufficient and thread structure must also be considered. It is the programmer's responsibility to set block size, but selecting the right size is not always intuitive. In this paper, we propose using machine learning to automatically select profitable block sizes. Additionally, we show that machine learning techniques coupled with performance counters can provide insight into the underlying reasons for performance variance between different configurations.**

## I. Introduction

Graphics Processing Units (GPUs) can provide great performance at low power consumption as long as there is good utilization of resources. Thread block size is a key factor in determining a kernel's occupancy. Occupancy is the ratio of the number of active warps running on a GPU to the maximum number of warps that can be scheduled. Occupancy provides intuition into how well a parallel kernel utilizes the GPU and is closely related to resource allocation. A general guideline is to find the thread configuration that leads to the highest occupancy. However, it has been shown that for some kernels the highest occupancy does not always yield the best performance[22]. High occupancy leads to increased resource contention, as more threads compete for limited hardware resources such as registers and shared memory. Low occupancy provides each thread with more resources but this can have a negative impact due to low latency hiding. Furthermore, multiple block sizes can provide highest occupancy for a given kernel, but their performance can vary at these different configurations. This is because variation in thread configuration yields different utilization of hardware resources. Thus, optimizing for occupancy alone is insufficient and the thread geometry must also be taken into consideration.

This work was supported by the National Science Foundation through awards CNS-1305302 and CNS-1253292.

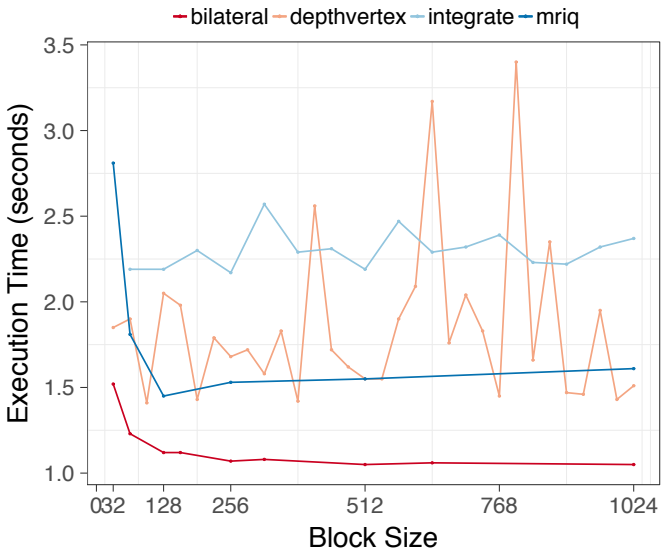


Fig. 1. Execution time of four applications with varying block sizes.

Current practice dictates that programmers choose the grid and block size to optimize their GPU applications. Selecting a good thread configuration is not always intuitive. Small variations in the thread block size can have huge performance impact. Consider the performance variations of four kernels shown in Fig. 1. The performance can vary by as much as a factor of three when selecting different block sizes for the same kernel (`depthvertex`). Although larger block sizes yields better performance on average, the largest block sizes do not necessarily produce the best results. For instance, for`mriq`, it is most profitable to select a relatively smaller block size of 128.

Navigating the different choices for thread block configuration can prove time consuming for the programmer. It may require the programmer to manually change the thread configuration, re-run the program, and collect performance results for each change until the desired performance level has been reached. Additionally, the space that needs to be considered when finding an optimal thread block size is multi-

978-1-5386-2588-0/17 $31.00 © 2017 IEEE
DOI 10.1109/HPCC-SmartCity-DSS.2017.58

dimensional. This complex search space can prove to be difficult to evaluate. As the size of this search space increases, it soon becomes unfeasible to perform an exhaustive search and many heuristic searches can easily become stuck in local optima. Features of this search space include the size of input data and the number of registers allocated, both of which are correlated to the kernel's performance under a given block size. Another factor that can cause variance is the grid size. Grid size reflects the total amount of work to be done in terms of the number of threads launched. When a large grid size is used, this results in less work for each thread to perform and increased contention for limited hardware resources.

Using machine learning (ML), performance and power consumption of GPU kernels can be improved through automatic selection of profitable thread configurations. This reduces the number of kernel runs necessary and allows for a more efficient evaluation of the complex search space. In addition, ML techniques coupled with hardware performance counters can help provide insight into the underlying reasons for performance variance between different thread configurations.

In this paper we present a strategy for selecting profitable blocks sizes in GPU kernels using supervised ML. Our ML model uses dynamic performance events as features. Given a GPU kernel, our framework profiles the kernel and extracts the relevant dynamic features. The model then predicts if a change in block size will improve the performance of the given kernel.

Our framework automates all major steps in the ML workflow, including feature extraction, feature selection, and training data labeling. In order to ensure a sufficient sample size for the training data, we generate multiple code variants from a single base program. These variants all exhibit distinct behavior on the target platform, allowing for a range of program characteristics for the ML model to learn from.

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

- construction of a novel machine learning based heuristic for selecting thread block sizes that accounts for multiple performance trade-offs
- a general framework for automatically developing supervised classifiers for platform-specific performance modeling and automating the machine learning workflow
- an analysis of the underlying causes of performance anomalies due to thread block variation

## II. Background

A GPU is a highly parallel processor that is traditionally used for rendering computer graphics. However, modern GPUs are commonly used for performing computations in scientific and engineering applications. A GPU consists of a set of Streaming Multiprocessors (SMs), and each SM contains a number of execution units called Stream Processors (SPs). Modern GPUs contain thousands of SPs. An SM is designed to execute hundreds of threads concurrently and follows the single instruction, multiple data (SIMD) model of execution. The compute capability of a NVIDIA GPU identifies the features supported by the GPU hardware [2].

### A. CUDA

CUDA is a programming interface which allows direct programing of NVIDIA GPUs. CUDA C is an extension to the C programming language that allows developers to write parallel functions, called kernels, for execution on the GPU. In the CUDA programing model, GPUs can achieve high-performance by executing massively parallel threads simultaneously.

### B. Thread Hierarchy

The most basic unit of execution in CUDA is a thread. Warps, which are sets of 32 threads that are simultaneously executed together, are divided into thread blocks. Thread blocks execute independently of one another, allowing them to be scheduled in any order across any number of cores. Warps within the same thread block are executed on the same multiprocessor and access the same shared memory unit. Each thread block is assigned to a single SM during the execution of a kernel. A grid is a collection of thread blocks. The number of thread blocks in a grid is typically based on the size of the data being processed. The thread blocks within a grid are mapped across multiple SMs. The maximum number of threads which can be assigned to each block varies depending on the GPU's architecture and compute capability. Likewise, the maximum blocks per SM and maximum threads per SM also depends on the compute capability. Limiting factors include the number of registers and shared memory required by the kernel and the number of registers and amount of shared memory available on the multiprocessor [2].

### C. Memory Hierarchy

A CUDA enabled GPU has six different memory components: register, shared memory, local memory, global memory, texture memory and constant memory. Every thread has its own private local memory. Each block has its own shared memory, which is shared among all the threads within that block. Global memory, constant memory, and texture memory can be accessed by all threads. Constant and texture memory are read-only, while the other memory types are read/write. A generalized diagram of this memory hierarchy is depicted in Fig. 2.

### D. Machine Learning

Machine learning (ML) is a method of data analysis that uses algorithms which iteratively learn from data, allowing computers to find hidden patterns without being explicitly programmed. If a computer program is able to improve its performance of accomplishing a task by using previous experience then it is said to have learned[17]. One of the biggest strengths of ML is the ability to automatically apply complex mathematical calculations to large sets of data with minimal effort from the user.

The two most commonly used ML methods are supervised and unsupervised learning. In unsupervised learning, the input

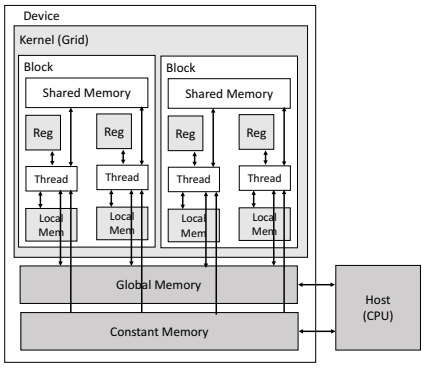


Fig. 2. Generalized GPU memory diagram

data is not labeled with the correct output. The goal of unsupervised learning is to discover similarities and find structure within the data.

Supervised learning ML algorithms are trained using labeled instances. The learning algorithm is provided with a set of inputs and the corresponding correct output, typically referred to as the training set. The goal of the learning algorithm is to infer a function that minimizes the error with respect to these inputs. Put briefly, the purpose of supervised ML algorithms is to learn a mapping $X \mapsto Y$, where $x \epsilon X$ is some instance and $y \epsilon Y$ is a class label.

A decision boundary is the hyper-surface that partitions the learning space into sets, one for each class. A decision boundary is the region of the learning space in which the output label is ambiguous. The learning space is linearly separable if the classes of the space can be separated with a single linear surface.

## III. Related Work

## IV. Optimizations for Thread Configuration

Seo *et al.* developed a heuristic for work group size selection for OpenCL kernels running on multicore processors [20]. They use static estimation and runtime feedback to fine-tune the workgroup size for improved locality and load balancing. They compare their numbers to an exhaustive search of all possible workgroup configurations. These results show that their strategy can get the same performance at a much lower cost. Their experiments do show significant variation in performance for the NAS SP kernel for different workgroup sizes. They do not extend this technique to GPUs, where the performance issues are much different.

Tran *et al.* proposed a tuning model for calculating candidate grid and block sizes to achieve optimal performance based on highest occupancy [21]. Their approach is able to calculate a set of candidate grid and block sizes faster than using exhaustive search. However, their model relies solely on the thresholds of the block and grid sizes enforced by a GPU architecture. They do not consider the characteristics of the kernel, which is be essential in determining optimal thread configuration. Their model is mainly used to reduce the search space rather than using a ML predictor and may output a list of multiple candidate configurations.

Magni*et al.* implemented thread-coarsening compiler transformations by developing a LLVM-based OpenCL compiler [14]. Additionally, they utilized regression trees and hardware performance counters to identify performance features that are affected by thread-coarsening. They evaluated the effect of the coarsening factor on performance and found that regression trees are able to identify the hardware features relevant to performance. Magni*et al.* conducted another study where they use source-level directives to tune thread configuration parameters. However, their tuning approach does not explicitly model the resource usage of the kernel [15].

Gupta *et al.* designed STATuner, which identifies a feature set of static metrics that characterize a CUDA kernel and builds a Support Vector Machine classifier to predict which block size provides the best performance [9]. Static metrics are obtained by compiling CUDA kernels in LLVM. Static analysis of the generated LLVM binary code and IR is performed to get metrics for instruction mix, loops, register usage, shared memory per block, and thread synchronization. Our approach differs in that our framework uses dynamic kernel features as input to the ML model.

## V. Machine Learning in Performance Modeling

A study of recent applications of ML 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 machine learning modeling and tuning (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 [23]. 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 [3]. 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 [8].

Similar to many ML problems, success of ML techniques hinges on accurate input characterization. Researchers have

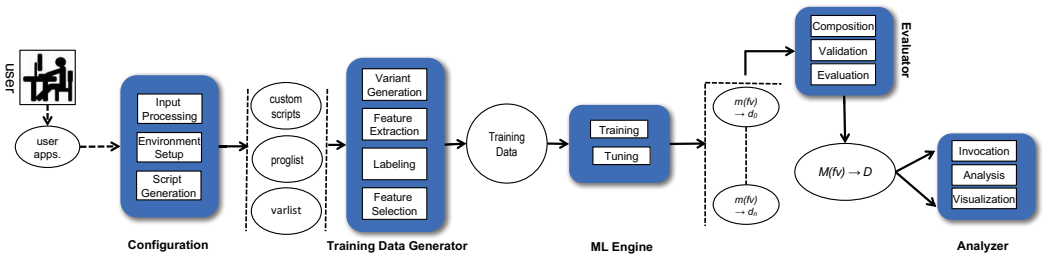


Fig. 3. Overview of our machine learning framework

attempted to characterize programs using program control flow graph [6] static program features [8] and hardware performance counters [19]. 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 [16] while some have employed statistical methods to select events that vary most across different program executions [11], [19]. Wu *et al.* designed a model that uses neural networks and k-means clustering to estimate the performance and power of a kernel on other hardware configurations [24]. Hardware performance counter values collected on one hardware configuration are used as input to the model for predicting performance of the kernel on the other configurations. The focus not on optimizing a kernel for a given system, but rather determining how well the kernel will perform on other systems.

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 [5] predicting optimal build configurations [12], runtime configurations [7], [13], identifying performance bottlenecks [10], [11] and recently, also in efficient energy management [4]

## VI. Design and Implementation

Fig. 3 gives an overview of our framework. To begin, our framework generates custom scripts that drive the tasks of feature extraction, feature selection, training data generation, model training, evaluation, and selection. The newly created model is stored as an R script and provides an interface for the user to invoke it on unseen programs. An interactive mode is also supported to perform subtasks selectively.

### A. Configuration

We provide a simple interface which allows users to specify the directory of the programs to be used as input for the training data generation. This configuration interface sets environment paths, detects CUDA enabled devices, and creates customized build and execute scripts that are tailored to the user's environment. In this phase, instructions for generating training data are specified in a file called *proglist*.

### B. Training Data Generatator

After generating the custom makefiles and execute scripts the configurer creates a *proglist* file that encapsulates necessary information for generating training data on the target platform. Each line of the file contains information for executing each program that is to be used in the training set. This file serves as input into a script called *varlist_gen.sh*. *varlist_gen.sh* reads each line of the *proglist* and outputs a file, named *varlist*, containing instructions for creating program variants for each baseline program that was listed in the *proglist*. These variants include modifying the `-maxrregcount` flag, thread block configuration in the kernel launch, and differing program input data (when available). The *varlist* is sent to a script that generates, builds, and executes each program variant. In this phase, the runtime features of each program contained in *varlist* is collected using `nvprof`. The collection and processing of data in this phase is explained in more detail in Section X.

### C. ML Engine

In the Machine Learning (ML) Engine phase, the training data is supplied to an R script. Within this script, the training dataset is randomly partitioned into training and testing sets. An SVM model is trained using the training set and its performance is evaluated using cross-fold validation. This process is repeated 10 times, adjusting tuning parameters each time, and the model yielding the highest accuracy during validation is selected. The final model's performance is further evaluated using the testing set in order to ensure that overfitting has not occurred.

### D. Analyzer

The framework currently supports three types of analysis visualizations to provide insight to the user about the training data and the generated model.

*1) Cluster-PCA plots:* Cluster-PCA plots are used to examine properties of the training data. *k-means* clustering is applied on the feature space, where the value of $k$ is determined via the silhouette method. We perform principal component analysis (PCA) on the feature space and the clustering results are visualized on scatter plots by projecting the clusters onto the two principal components (PCs) that explain the most variation in the data. A point in the plot represents a code variant. Points can be annotated to show base program, class

label, or threshold-delineated PCA values. Ellipses represent clusters and two points falling within the same cluster indicates that they exhibit similar behavior. Cluster-PCA plots can provide intuition about the training data in several ways. The number of clusters is a reflection of the the number of different types of codes present.

*2) PCA-VR segment plots:* Although PCAs are primarily used for dimensionality reduction, in MLMT they can be useful in other ways. We can think of a PC as a compound feature that describes a broad performance pattern. For instance, a PC might represent memory-bound behavior and contain related features such as LLC miss rate, DRAM accesses and stalled cycles. In general, however, the relationship between many of the performance events is either unknown or not obvious to the user. Identifying major performance events that comprise a PC can provide valuable insight about both program performance and architectural characteristics. The challenge, however, is that PCs are not amenable to direct visualization. To address this, 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. These segment plots provide the practitioner with a quick way to identify related features (although the nature of the relationship is not revealed) in the feature space. This knowledge can be used to optimize code independent of the model being generated.

*3) Decision tree analysis:* Decision trees are prone to overfitting and their ability to learn complex spaces is limited. Despite these shortcomings, decision trees are easy to visualize and can provide an intuitive way to understand the learning behind the predictions.

## VII. Model Formulation

Our model uses machine learning to provide the user with suggestions on how to modify the thread block size of their code. Given a kernel, our model will determine if the thread block size should be increased or decreased to achieve better performance.

## VIII. Determining Legal Thread Block Dimensions

When determining legal thread block dimensions, several factors need to be taken into consideration:

- The hardware constraints of the GPU
- The original thread block dimensions
- The correctness of the kernel's results

Often the kernel has been coded such that the correctness of the results is dependent on the block size. This means that while a given block size is legal in CUDA, it may not be valid in context of the program in question. It can be determined whether or not a block size is valid by checking the results of a program run using the new block size with the original results. Note that this approach only works for programs whose output is deterministic. For these reasons, we have chosen to create a model that suggests relative changes in block size rather than giving absolute numbers. The programmer can then select the next valid block size in the direction of the change.

## IX. ML Algorithm Selection

To make predictions on the direction of change in block size for a given kernel, we use Support Vector Machines (SVMs). In selecting which machine learning algorithm to use, we took into consideration what type of decision boundaries we expected in our feature space. Specifically, whether or not the feature space is linearly separable is important in selecting which machine learning model to use.

Although our data includes only three classifications, the feature space is much more complex. Because of this, our data does not exhibit a linearly separable decision boundary. Thus, we opted to use an algorithm capable of learning complex spaces that are not linearly separable. We selected SVMs due to their high accuracy and ability to learn complex search spaces. Other strengths of SVMs are that they aren't overly influenced by noisy data and are not prone to overfitting. We use the kernel trick, which maps the feature space into higher dimensional space, in order to enable learning of nonlinearly separable decision boundaries. We also employ the all-versus-all strategy, which combines several binary SVMs, to allow for multi-class predictions.

Additionally, we also rely on clustering to evaluate the feature space and decision trees to provide meaningful insight into the reasons why some kernels perform better with smaller block sizes over larger block sizes.

## X. Training Data Generation

Our framework is able to manipulate the max register allocation and block size of CUDA kernels in order to generate multiple code variants from the same base program. Next, dynamic metrics of each of the kernel variations are collected using performance counters. This set of metrics becomes the input feature vector for the ML model.

### A. Feature Extraction

Our framework uses runtime events as features. To collect runtime events, we read values from hardware performance counters using nvprof. We created a shell script which reads a list of the selected events from a text file and passes them to nvprof. To reduce the time required for collection of these events, we take of advantage of multiplexing and divide the events into groups that can be measured during a single program run without causing conflicts in hardware counters. Based on expert knowledge and analysis, we selected events which are closely related to thread block size. The selected events are shown in Table I. In addition to considering resource utilization, we model the following program characteristics.

*1) Memory Divergence:* Memory access can greatly impact a kernels performance. Coalescing is a memory access technique in which memory requests to the same cache line are grouped together to create a single transaction. Coalescing is typically performed at the warp level. Address-aligned requests to contiguous memory locations from threads in the same warp are combined into a single transaction, greatly reducing memory traffic. With a greater number of warps, more coalesced memory accesses can occur at a time. However, too

TABLE I
NVPROF EVENTS COLLECTED

| Events Collected | |
|---|---|
| gld_request<br>gst_request | fb_read_sectors<br>fb_write_sectors |
| l1_local_load_hit<br>l1_local_store_hit<br>l1_global_load_hit | l1_local_load_miss<br>l1_local_store_miss<br>l1_global_load_miss |
| uncached_gld_transaction | global_store_transaction |
| gld_inst_32bit<br>gst_inst_32bit | inst_issued<br>inst_executed |
| not_predicated<br>_off_thread_inst_executed | thread_inst_executed |
| l2_write_sector_misses<br>l2_read_l1_hit_sectors<br>l2_total_write_sector_queries | l2_read_sector_misses<br>l2_total_read_sector_queries<br>shared_load_replay |
| global_ld_mem<br>_divergence_replays | global_st<br>_mem_divergence_replays |

many warps per SM can degrade performance due to increased resource contention. For these reasons, kernels which are memory-bound tend to be more sensitive to changes in block size.

*2) Control Divergence:* Control divergence is another factor that can influence a kernels performance. Frequent branch instructions and branch divergence can degrade performance. A control instruction is divergent if it forces threads within a warp to take different execution paths. In CUDA, divergence results in serialization of the execution paths, thereby increasing the total number of instructions executed. Additionally, threads within a warp cannot continue until all threads of the warp have exited the conditional path. Smaller block sizes can reduce the overhead of control divergence by reducing the number of instructions executed per warp and limiting the number of threads that must wait due to divergence. However, if too few threads are launched, it may be insufficient to hide instruction latency.

### B. Event Collection

The baseline version of each kernel is considered as the execution using the default thread block size and register pressure. For each baseline version, we modified the block size in the kernel launch configuration of the code. We executed the baseline and all variants and collected runtime events and kernel execution time using nvprof. Next, we computed the speedup of each instance over the baseline version. Labels were added to each instance in the dataset based on the speedup and block size.

### C. Data Labeling

To train the ML model, we must provide it with labeled instances that it will learn from. Manually labeling each instance in the training dataset is time consuming. To alleviate the user of this task, our framework automates the process using scripts and a simple algorithm to determine the labels. For each instance in the training data set, the speedup over the baseline is computed. We consider a speedup $< 1$ to be bad, a speedup = 1 to be neutral, and a speedup $> 1$ to be good. Next, the block size of the variant is compared to the baseline block size and the label is assigned using Algorithm 1.

**Algorithm 1** Labeling algorithm

```
for all d ∈D do
    if speedup < 1 and newBlock < origBlock then
        new.Label ← increase.
        orig.Label ← noChange.
    else if speedup < 1 and newBlock > origBlock then
        new.Label ← decrease.
        orig.Label ← noChange.
    else if speedup > 1 and newBlock < origBlock then
        new.Label ← noChange.
        orig.Label ← decrease.
    else if speedup > 1 and newBlock > origBlock then
        new.Label ← no Change.
        orig.Label ← increase.
    end if
end for
```

### D. Feature Selection

Feature selection is important for improving accuracy of a ML model. Features which are redundant or provide no additional information to the model should be removed. First, we removed any events that had a value of 0 for all program runs. Next, we evaluated the association between the remaining features by calculating the correlation coefficients using the Pearson correlation formula, which measures a linear dependence between two variables, $X$ and $Y$:

$$r = \frac{\sum(X - \overline{X})(Y - \overline{Y})}{\sqrt{\sum(X - \overline{X})^2 \sum(Y - \overline{Y})^2}}$$

Features with a correlation coefficient greater than 0.9 were removed from the set. Features which are highly correlated to all other features do not add any additional information to the data, hence they are redundant and can reduce model prediction accuracy. The remaining features and their correlation are shown in figure 4. These features are estimated to provide the highest predictive power.

## XI. Experimental Setup

We evaluated our model using a Nvidia Tesla K40c GPU on a linux system that had CUDA 7.5 installed. The K40c has a compute capability of 3.5, supports a maximum of 1024 threads per block, and a maximum of 2048 threads per SM.

### A. Benchmarks

We used kernels from the Parboil [1] and SLAMBench [18] benchmark suites to generate training data.

## XII. Results

### A. Training Space Characterization

The complexity of the feature space can be evaluated by performing principle component analysis (PCA) and k-means clustering on our training dataset. As we can see in Fig. 5, in which many different block sizes are contained within the same cluster, the best thread block size is not always easy to

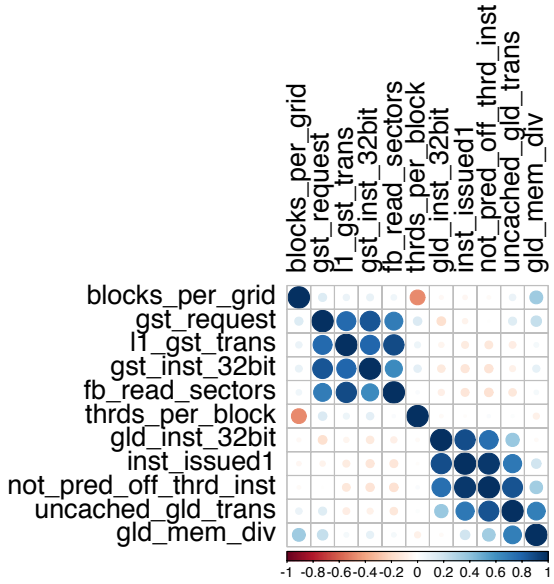


Fig. 4. The correlation matrix of the remaining features after feature selection is performed.

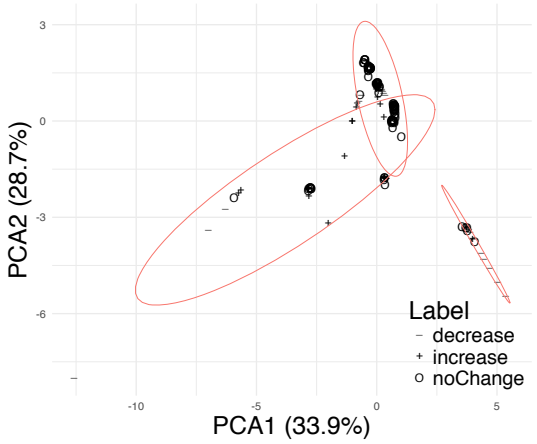


Fig. 5. A Cluster-PCA plot analysis on our training dataset.

determine. This implies that even though two programs may be very similar, subtle differences can lead to variance in resource utilization and the need for different block sizes.

Fig. 6 shows a VR-PCA segment plot for our training dataset. We can see that the fourth principle component, shown in red, is dominated by features related to global load memory transactions. This further demonstrates that memory access patterns and memory divergence is a primary factor in determining a kernel's classification and selecting a good block size. Another principal component worth noting is the second principal component, shown in blue, in which features related to instruction execution have the most contribution. The first and third principal components, in purple and green respectively, appear to be less significant.

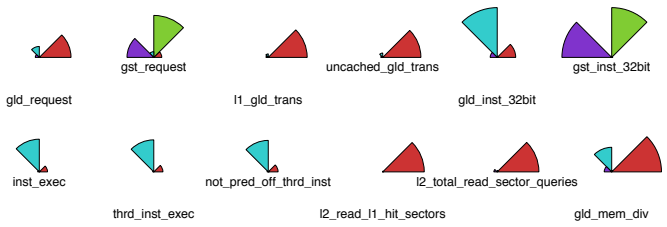


Fig. 6. The segment plot shows the contribution of each attribute to the principal components.

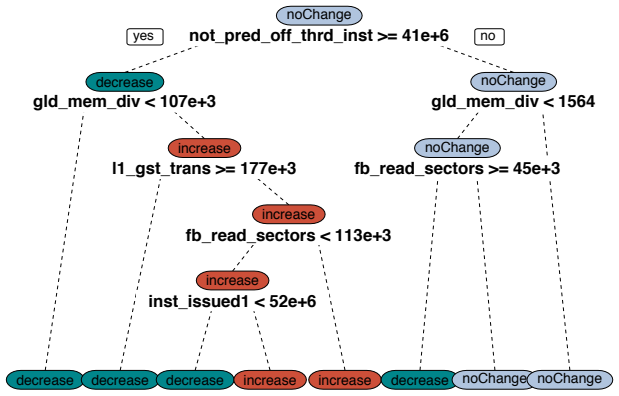


Fig. 7. The decision tree's splitting criteria.

### B. Model Evaluation

We evaluated our model's accuracy using 10-fold cross validation. We split the training set in 10 groups of approximately the same size, then iteratively train a SVM using 9 groups and make a prediction on the group which was excluded. We set the value of k to be 10. Our SVM model had an accuracy rate of 83.7%. Our decision tree model had an accuracy rate of 81.4%.

We created a visualization of the splitting criteria used by the decision tree in order to understand which variables of the feature vector were used to make predictions. As seen in Fig. 7, the choice of the thread block size is sensitive to memory divergence, L1 cache behavior, and reading from DRAM.

### C. Performance and Energy Gains

We used our model to tune 6 kernels that were not contained in the training or validation data. For these kernels, we followed the model's suggestions of adjusting the block size, selecting the next valid size and reinvoking the model until a "no change" suggestion was provided. We then compared the execution time of the unmodified kernel to that of the modified kernel to determine the speedup.

When we adjusted the block size in accordance with the suggestions provided by our model, we were able to obtain up to 1.8x speedup over the baseline versions. The tuning results of 6 programs is shown in Fig. 8. In regards to energy,

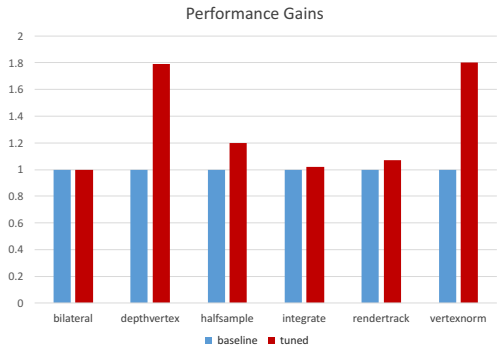


Fig. 8. Speedup gained in relation to the baseline from using our model to tune 6 programs.

we found that adjustments in thread block size provided no significant change in a kernel's power consumption.

## XIII. Conclusion

This paper presents the construction of a ML-based heuristic for selecting profitable block sizes. Using supervised ML algorithms and dynamic performance events as features, our ML model predicts if a change in block size will improve the performance of a given kernel. The framework presented in this paper introduces strategies for automating time consuming aspects of training data generation and building a ML model, such as feature extraction, feature selection and labeling. We address the common issue of not having enough programs to build a sufficiently large and diverse training dataset by generating multiple code variants for a single base program.

We demonstrated the effectiveness of our ML model on a mix of programs from the SLAMBench and Parboil benchmark suites. We show that our framework can produce accurate models for making predictions. The visualizations allowed us to better analyze the training dataset and results of the ML models in order to identify underlying causes of performance anomalies when varying thread block size. We found that subtle differences in a kernel's runtime behavior can result in the need for different block sizes. Additionally, the choice of thread block size is sensitive to memory access patterns, especially memory divergence.

By using our machine learner on 6 unseen kernels that were excluded from the training data generation phase, we were able to achieve up to 1.8x speedup over the baseline versions.

## References


[1] Parboil Benchmark Suite. http://impact.crhc.illinois.edu/parboil.php.

[2] *CUDA Programming Guide, Version 3.0*. NVIDIA, 2010.

[3] 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 *Proc. of the International Symposium on Code Generation and Optimization (CGO '07)*, Washington, DC, USA, 2007.

[4] R. Cochran, C. Hankendi, A. K. Coskun, and S. Reda. Pack & cap: adaptive dvfs and thread packing under power caps. In *Proc. of the 44th annual IEEE/ACM international symposium on microarchitecture*, 2011.

[5] 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.

[6] J. Demme and S. Sethumadhavan. Approximate graph clustering for program characterization. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4), 2012.

[7] M. K. Emani and M. F. P. O'Boyle. Celebrating diversity: a mixture of experts approach for runtime mapping in dynamic environments. In *Proc. of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA*, 2015.

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

[9] R. Gupta, I. Laguna, D. Ahn, T. Gamblin, S. Bagchi, and F. Lin. Statuner: Efficient tuning of cuda kernels parameters. In *Supercomputing Conference (SC 2015), poster*, Nov 2015.

[10] N. Jain, A. Bhatele, M. P. Robson, T. Gamblin, and L. V. Kale. Predicting application performance using supervised learning on communication features. In *Proc. of the International Conference on High Performance Computing, Networking, Storage and Analysis*.

[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 *Proc. of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013.

[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, 2012, Revised Selected Papers*, 2012.

[13] 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 *Proc. of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009.

[14] A. Magni, C. Dubach, and M. F. P. O'Boyle. A large-scale cross-architecture evaluation of thread-coarsening. In *Proc. of the 2013 ACM/IEEE conf. on Supercomputing*, 2013.

[15] A. Magni, D. Grewe, and N. Johnson. Input-aware auto-tuning for directive-based gpu programming. In *Proc. of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, 2013.

[16] C. McCurdy, G. Marin, and J. S. Vetter. Characterizing the impact of prefetching on scientific application performance. In *High Performance Computing Systems. Performance Modeling, Benchmarking and Simulation*. Springer, 2013.

[17] T. M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[18] L. Nardi, B. Bodin, M. Z. Zia, J. Mawer, A. Nisbet, P. H. J. Kelly, A. J. Davison, M. Luján, M. F. P. O'Boyle, G. Riley, N. Topham, and S. Furber. Introducing SLAMBench, a performance and accuracy benchmarking methodology for SLAM. In *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, May 2015.

[19] 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.

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

[21] N. P. Tran and M. Lee. Parameter tuning model for optimizing application performance on gpu. In *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, Sept 2016.

[22] V. Volkov. Better performance at lower occupancy. 2010.

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

[24] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou. Gpgpu performance and power estimation using machine learning. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2015.