Iterative Machine Learning (IterML) for Effective Parameter Pruning and Tuning in Accelerators

Xuewen Cui
Virginia Tech
Blacksburg, Virginia
xuewenc@vt.edu

Wu-chun Feng
Virginia Tech
Blacksburg, Virginia
wfeng@vt.edu

ABSTRACT

With the rise of accelerators (e.g., GPUs, FPGAs, and APUs) in computing systems, the parallel computing community needs better tools and mechanisms with which to productively extract performance. While modern compilers provide flags to activate different optimizations to improve performance, the effectiveness of such automated optimization depends on the algorithm and its mapping to the underlying accelerator architecture. Currently, however, extracting the best performance from an algorithm on an accelerator requires significant expertise and manual effort to exploit both spatial and temporal sharing of computing resources in order to improve overall performance. In particular, maximizing the performance on an algorithm on an accelerator requires extensive hyperparameter (e.g., thread-block size) selection and tuning. Given the myriad of hyperparameter dimensions to optimize across, the search space of optimizations is generally extremely large, making it infeasible to exhaustively evaluate each optimization configuration.

This paper proposes an approach that uses statistical analysis with iterative machine learning (IterML) to prune and tune hyperparameters to achieve better performance. During each iteration, we leverage machine-learning (ML) models to provide pruning and tuning guidance for the subsequent iterations. We evaluate our IterML approach on the selection of the GPU thread-block size across many benchmarks running on an NVIDIA P100 or V100 GPU. The experimental results show that our IterML approach can significantly reduce (i.e., improve) the search effort by 40% to 80%.

CCS CONCEPTS

• **Computing methodologies** → *Discrete space search*; *Machine learning approaches*; • **General and reference** → **Performance**; • **Software and its engineering** → **Software performance**.

KEYWORDS

GPU, performance, thread block, machine learning, random forest, classification and regression trees (CART), support vector machine (SVM), multi-layer perceptron (MLP), k-nearest neighbor (KNN)

CF '19, April 30-May 2, 2019, Alghero, Italy

ACM ISBN 978-1-4503-6685-4/19/05…$15.00
https://doi.org/10.1145/3310273.3321563

ACM Reference Format:
Xuewen Cui and Wu-chun Feng. 2019. Iterative Machine Learning (IterML) for Effective Parameter Pruning and Tuning in Accelerators. In *Proceedings of the 16th conference on Computing Frontiers (CF '19), April 30-May 2, 2019, Alghero, Italy*. ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/3310273.3321563

1 INTRODUCTION

Systems with accelerators, particularly GPUs, are becoming increasingly prominent on the Top500 [3] as well as in embedded high-performance computing (HPC) systems, like those found in smartphones and smart cars. In such heterogeneous systems, the host CPU manages the execution context while the computation is typically offloaded to the accelerators, like GPUs. Leveraging accelerators not only enables high performance, but it also improves energy efficiency [15]. However, extracting the optimal performance and energy efficiency from these accelerators can be extraordinarily difficult for a software developer [7]. Thus, developers need simpler abstractions and underlying mechanisms to program these accelerators [2, 12] as well as significant domain knowledge to tune the performance of the code on these accelerators [5, 6].

Because accelerator architectures expose many software and hardware parameters for developers to tune to achieve optimal performance, the different combinations of parameters typically result in a large search space, making it infeasible for developers to exhaustively test each combination of parameters. Furthermore, choosing the wrong combination of parameters can result in severe performance degradation. Obviously, efficiently finding the optimal parameter settings would be ideal [8]. As such, this paper presents IterML, our iterative parameter pruning and tuning approach with machine-learning (ML) models. During each iteration, we use ML models to assist with pruning (and tuning) the rest of the search space by their predicted performance. To demonstrate the efficacy of IterML, we apply it across 10 benchmarks and run them on NVIDIA P100 and V100 GPUs. The experimental results show that our IterML approach can significantly reduce the search space by 40%-80%, when compared to the exhaustive random search.

We make the following research contributions:

- The design of an iterative machine-learning (IterML) approach that automatically determines nearly optimal parameter settings for the GPU thread-block size to achieve high performance.
- An empirical study that demonstrates how our IterML approach consistently delivers better search speed over the non-iterative ML methods and nearly achieves optimal performance while sampling only 1.5% of the search space on average and, in turn, reducing the search effort by 40%-80%.

The rest of the paper is organized as follows. In §2, we motivate the need for this research, along with related work. Then, §3 describes the design of our iterative machine-learning (IterML) approach, followed by case studies that make use of IterML in §4. Finally, §5 evaluates our experimental results.

2 MOTIVATION AND BACKGROUND

In this section, we first present a motivating example to illustrate the importance of parameter tuning in heterogeneous computing, followed by a brief discussion on related work.

2.1 Motivating Example

Figure 1 shows a performance heatmap of our lid-driven cavity (LDC) code, where the thread-block size (i.e., blockDim.x x blockDim.y) is varied, when running on an NVIDIA K20m GPU. The x- and y-dimensions are limited to 64, and each thread block contains at most 512 threads. We observe that the performance varies significantly across different thread-block sizes. At the ideal thread-block size of 8x8 for *this* code on *this* GPU, the K20m achieves 103 GFLOPS. On the other hand, the performance can be more than 20% worse at 79 GFLOPS if the wrong thread-block size is chosen. Unfortunately, due to the size of the search space, identifying the ideal thread-block size is tedious and time-consuming. Furthermore, the GPU thread-block size is merely one parameter to be tuning; there are many other potential parameters that could be tuned, e.g., GPU block size, degree of loop unrolling, and so on. These parameters significantly increase the search space, making it infeasible to exhaustively enumerate every combination. Thus, there exists a need for a simpler and more efficient approach to identify ideal parameter settings for (near-)optimal performance.

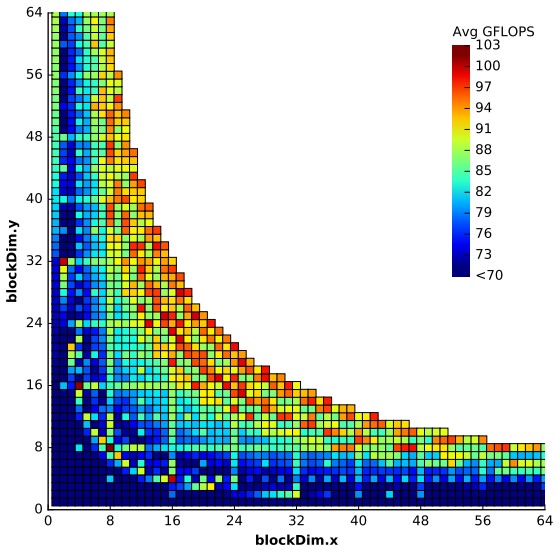

Figure 1: Performance of a Lid-Driven Cavity Code with Varying GPU Thread-Block Size on an NVIDIA K20m GPU

2.2 Related Work

In [1, 14, 19], the authors auto-tune the performance of a particular algorithm or application on an accelerator, like the GPU. The auto-tuning, however, still requires extensive expertise (or intuition) to manually select the key parameters as well as compiler flags. To address this problem, we propose an approach that automatically identifies a smaller (pruned) search space that contains the (near-)optimal setting, which, in turn, can be searched exhaustively. Specifically, given a large search space, our iterative machine-learning (IterML) approach gathers information during each iteration, builds models, and finds the best interaction between the parameters and the performance. This, in turn, provides automated guidance as to how to tune performance in the context of a large parameter search space.

Various statistical or machine-learning (ML) methods have been applied to help with auto-tuning parameters to get better performance. In [10], the authors propose a linear regression model to predict processor performance based on micro-architectural parameters, but it requires a large amount of processor profiling data as input to build the linear model. In [13], deep-reinforcement learning is used to find the optimal values of tunable parameters in computer systems — from a simple client-server system to a large data center. It can be deployed into a production system to collect training data and suggest tuning actions during the system's daily operation. However, it requires the system to be mostly static, which is not applicable to new algorithms or libraries targeting new devices like GPUs. For GPUs, there exist pheromone models based on the profiling data of GPUs [4, 18] that require large training sets across various codes and with a wide variety of performance counters. Moreover, they require developers to have intimate knowledge about the programs. Other related research focuses on design coding machines to handle the programming tasks [11].

In contrast, our goal is to help developers productively tune their algorithms (or benchmarks) to achieve near-optimal performance with the least amount of effort and domain knowledge.

3 APPROACH AND DESIGN

Here we articulate the approach and design of our iterative machine learning (IterML), including the selection of the parameter search space, the iterative machine-learning (pruning) algorithm itself, and the regression models to predict the rest of the search space.

3.1 Choosing the Parameter Search Space

For microbenchmarks or libraries, a set of hyperparameters that define the dimensions of the tuning search space must be identified. The hyperparameters may relate to software (e.g., input partition chunks and thread count) or to hardware (e.g., active core count, GPU thread-block size, and compiler optimization options). Parameters can be either binary in nature (e.g., turning on or off a compiler flag) or multi-valued across a range (e.g., thread-block size and number of partition chunks). Our iterative pruning approach builds knowledge based on machine-learning (ML) models as it uses samples from one iteration to then look for potentially better samples in subsequent iterations.

3.2 Iterative Machine-Learning (IterML) Pruning and Tuning Algorithm

We propose an iterative pruning and tuning algorithm to quickly and effectively reduce the search space, as shown in Algorithm 1. As we first define the search space D, specified by multiple design parameters (e.g., thread-block dimension), we also chose a pick-ratio, which is the sample ratio that we need to test in each iteration. Once selected, the number of samples that we pick in each iteration stays constant. For each iteration, we first apply regression models to the samples. We then predict the performance of the residual search space and drop those with lower performance by a cut-ratio, e.g., 50%. We repeat this process until we meet a stopping criteria. For example, if the cut-ratio is 50%, then after every iteration, the search space is halved, which, in turn, means that the number of iterations is log_2(size of search space). We may adjust the cut-ratio based on the size of original search space.

Algorithm 1: Iterative Pruning Algorithm (IterML)

Input : **D:** search space specified with n design parameters
Pick-ratio: sample ratio taken in each iteration
Cut-ratio: ratio of the space pruned each iteration
Model: regression model chosen for prediction

Result: Best parameter combination currently found

initialization;

while *While not meeting stopping criteria* **do**
  Pick sample set **S** randomly from remaining **D** based on pick-ratio;
  Gather performance **P** of **S**;
  Build model M_i based on **P**;
  Predict the remaining **D** based on model M_i;
  Drop the remaining **D** with low predicted performance by **cut-ratio**;

end

3.3 Regression Models

Based on our iterative pruning approach, we need to build a model between each iteration to predict the *rest* of the search space. We study and use the following five popular machine-learning (ML) models to support our IterML algorithm.

Classification and Regression Trees (CART): Decision trees can be represented as a binary tree, where each node represents a single input variable (x) and a split point on that variable (assuming the variable is numeric). CART is usually fast to train and very fast to make predictions. It requires no data pre-processing and can be accurate for a broad range of problems.

K-Nearest Neighbors (KNN): Predictions are made for a new data point by searching through the entire training set for the K most similar instances (i.e., neighbors) and summarizing the output variable for those K instances. We use the mean output variable as the result for regression problems.

Support Vector Machine (SVM) Regression: SVMs use a hyperplane to split the input variable space. The support vector regression (SVR) uses the same principles as the SVM for classification, with only a few minor differences. In the case of regression, a margin of tolerance (i.e., ε) is set in approximation to the SVM, which would have already requested from the problem.

Random Forest (RF): This type of ensemble ML algorithm is called bootstrap aggregation or bagging. Multiple samples of training data are taken; models are then built for each data sample. When a prediction for new data needs to be made, each model makes a prediction, and the predictions are averaged to give a better estimate of the true output value. Combining predictions from these models results in a better estimate of the true underlying output value.

Multilayer Perceptron: A multilayer perceptron (MLP) is a neural network that connects multiple layers in a directed graph, which means that the signal path through the nodes only goes one direction. Each node, apart from the input nodes, has a nonlinear activation function. MLP utilizes a supervised learning technique called back propagation for training, which has drawn significant interest recently due to its success in deep learning.

4 EXPERIMENTS

To evaluate our iterative machine-learning (IterML) pruning and tuning approach, we leverage different ML models while pruning the search space. To demonstrate the efficacy of our approach, we focus on the GPU thread-block size as the hyperparameter of interest. The GPU thread block is usually composed of an X-dimension and Y-dimension. Each dimension ranges between 1 and 1024, inclusive. The product of the two dimensions, which is the thread number of each GPU thread block, is also limited by 1024. To identify the GPU thread-block size that delivers the best (optimal) performance, we exhaustively test the performance of different benchmarks using all the possible combinations of GPU thread-block size. We then apply our iterative machine-learning (IterML) pruning and tuning approach on the GPU thread-block size and evaluate its subsequent performance and compare it to the optimal performance.

Table 1: Benchmark List

Language	Benchmark	Description
CUDA	2dconv	2-D convolution
CUDA	3dconv	3-D convolution
CUDA	2mm	2 matrix multiplications
CUDA	3mm	3 matrix multiplications
CUDA	gemm	Matrix-multiply C=alpha.A.B+beta.C
CUDA	gesummv	Scalar, vector and matrix multiplication
CUDA	mvt	Matrix vector product and transpose
CUDA	syr2k	Symmetric rank-2k operations
CUDA	syrk	Symmetric rank-k operations
OpenACC	epcc	EPCC 27Stencil benchmark
OpenACC	ldc	Lid-driven cavity code

4.1 Benchmarks Studied

As shown in Table 1, we use nine (9) GPU kernels from the Polybench benchmark suite [17], an OpenACC kernel from the EPCC benchmark suite [9], and an OpenACC kernel from our lid-driven cavity (LDC) code to conduct the v experiments. The kernels use various GPU functional units and exhibit diverse behavior. For CUDA benchmarks, relevant design parameters are substituted

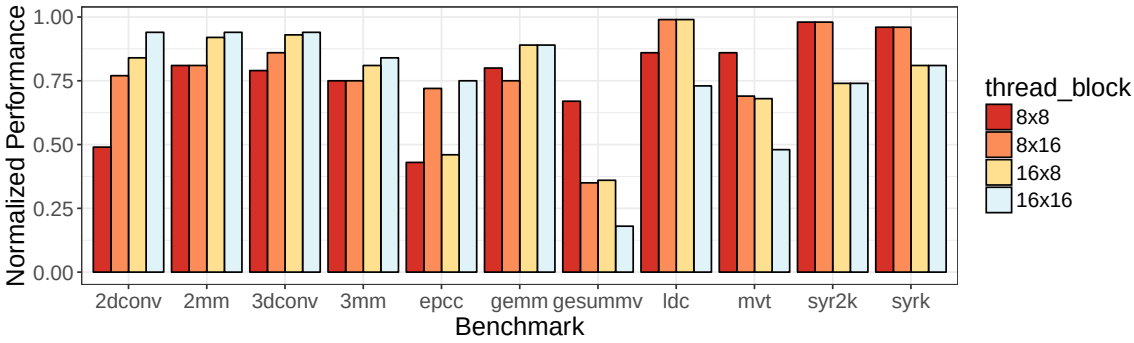

Figure 2: Normalized performance across varying thread-block sizes on the V100 GPU

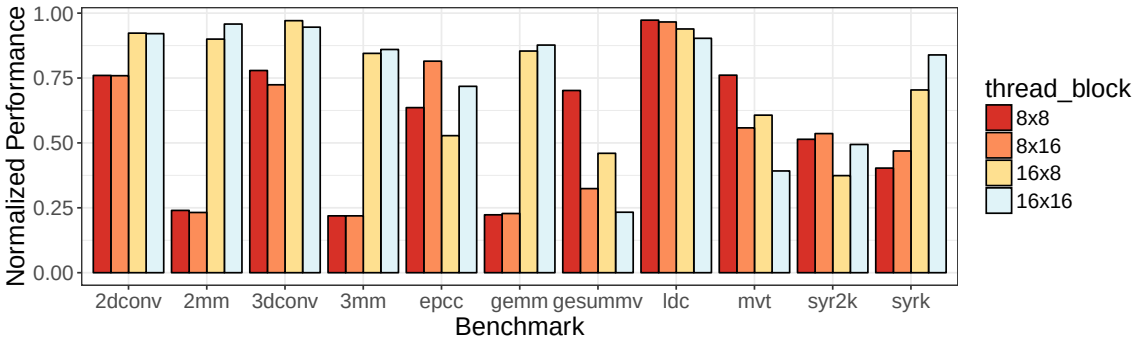

Figure 3: Normalized performance across varying thread-block size on the P100 GPU

by C macros so that our design can easily modify and recompile them. For the OpenACC benchmarks, we pass variables using the compiler flags to modify the OpenACC pragma.

Each kernel is executed 10 times and the average execution time reported. Only the GPU time of each kernel execution is measured and used, thus excluding any CPU work, data transfer, or kernel launch overhead.

4.2 Hardware Platform

We conducted our experiments on two modern NVIDIA GPUs: Tesla P100 and Tesla V100. The P100s are equipped with two 2.4-GHz Intel E5-2680v4 CPUs, resulting in 28 cores on each node. The connection between the GPU and the CPU is PCI-E 3.0. The V100 nodes are paired with two 3.0-GHz Intel Skylake Xeon Gold CPUs, resulting in 24 cores on each node. There is 384 GB of memory, and two NVIDIA V100 ("Volta") GPUs. Each of these GPUs is capable of more than 7.8 TFLOPS of double-precision performance.

4.3 Pruning Procedure

We begin to evaluate our iterative pruning approach after we collect all the performance data of the 10 benchmarks and applications by varying the GPU thread-block size. The pruning approach is implemented in Python with the Scikit-learn machine learning (ML) libraries [16]. We select five (5) commonly used models to predict the performance based on previous samples; these models include CART, KNN, SVM, RF, and MLP, as presented in §3.3. Because the thread-block search space is relatively modest, we use 0.5 as the cut-ratio, which means that we drop 50% of the search space with low predicted performance in each iteration. We can change this number depending on the scale of the search space. Each time, we pick a portion of the sample based on the pick-ratio and keep this sample number consistent until end. Due to the random selection of the samples, we repeat this process at least 100 times, thus drawing a distribution of the results using our iterative pruning approach. We then compare the result of different models based on this distribution. We normalize the samples to the result of the baseline, which entirely randomly selected the samples.

4.4 Case Studies

Here we present multiple case studies to show how the selection of the GPU thread-block size affects performance. *How does the manual selection of the thread-block size by an experienced developer impact performance?* In real-world GPU coding, developers set the GPU thread-block size based on their experience. Typically, the chosen block sizes are 64, 128, and 256. Figures 2 and 3 show the normalized performance of benchmarks, where the GPU thread-block size is set by experienced developers, relative to the optimal performance. We observe that none of the manually chosen thread-block sizes provide consistently good performance across all the benchmarks. In fact, each setting achieves only 20% to 99% of the optimal performance across the benchmarks.

Is there a universal thread-block size that "rules them all"? Figures 4 and 5 show the minimum normalized performance across all benchmarks tested on the V100 and P100 GPUs, respectively. For brevity, we only show the cases where the total number of threads is less than or equal to 512 and where each dimension is limited to 128. (Note: The unshown parts of the graphs produce similar results.) We observe that all the blocks in the heatmap achieve less than 30% of optimal performance. This result indicates that there is no such notion of a universal thread-block size that can consistently provide good performance for most applications.

Is the ideal thread-block size for one device good enough for other devices with similar architectures? We first identify the thread-block size that achieves the best performance on one device (e.g., P100) and then see if that ideal thread-block size for the one device (e.g., P100) is also good enough for another device (e.g., V100) with a similar architecture.

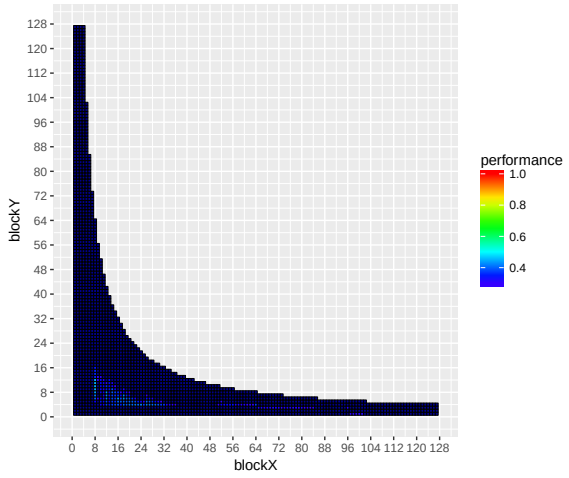

Figure 4: Minimum performance heatmap across all benchmarks on V100 GPU

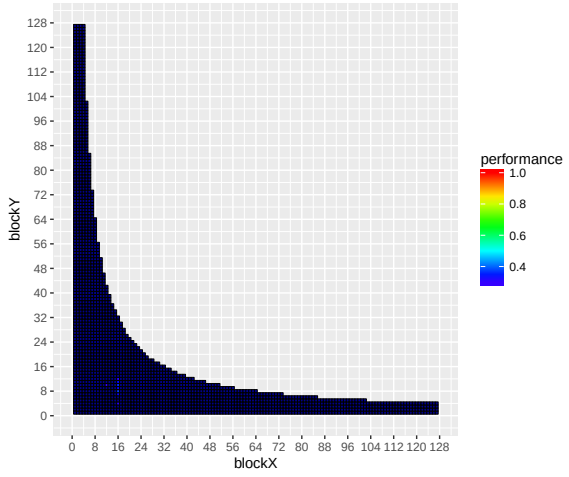

Figure 5: Minimum performance heatmap across all benchmarks on P100 GPU

Table 2 shows how the ideal thread-block size on the P100 GPU performs on the V100 GPU while Table 3 shows the converse. For some benchmarks (e.g., 2mm), the ideal block size delivers 99% of the optimal performance. However, there are still some benchmarks (e.g., syrk) that achieve only 44%-47% of the optimal performance.

Table 2: P100 ideal thread-block size performance on V100

Benchmark	Thread-X	Thread-Y	Performance
2dconv	512	2	0.974
3dconv	96	2	0.946
2mm	32	32	0.99
3mm	16	48	0.76
gemm	16	48	0.775
gesummv	16	1	0.868
mvt	14	1	0.903
syr2k	16	36	0.751
syrk	32	24	0.473
epcc	1	128	0.968
ldc	16	6	0.954

Table 3: V100 ideal thread-block size performance on P100

Benchmark	Thread-X	Thread-Y	Performance
2dconv	64	5	0.941
3dconv	88	10	0.846
2mm	64	16	0.994
3mm	32	26	0.938
gemm	32	26	0.938
gesummv	6	5	0.675
mvt	8	1	0.883
syr2k	8	124	0.601
syrk	8	124	0.447
epcc	8	64	0.778
ldc	4	32	0.945

5 EVALUATION

We evaluate the total sample ratio needed to achieve good performance using our iterative pruning and tuning approach. We vary the ML models used for prediction during each iteration and use exhaustive random search approach as our baseline. To quantify the goodness of the performance compared to the optimal, we utilize the following two standards:

(1) **Standard 1:** The sample ratio required to achieve a median that is higher than 95% of optimal performance. This means that the result is expected to be at least better than 95% of the optimal performance on average.
(2) **Standard 2:** The sample ratio required to achieve 5-percentile *higher* than 95% of the optimal performance. Standard 2 is more difficult standard to achieve than Standard 1. It means there is at least a 95% probability to get a result that is better than 95% of the optimal performance.

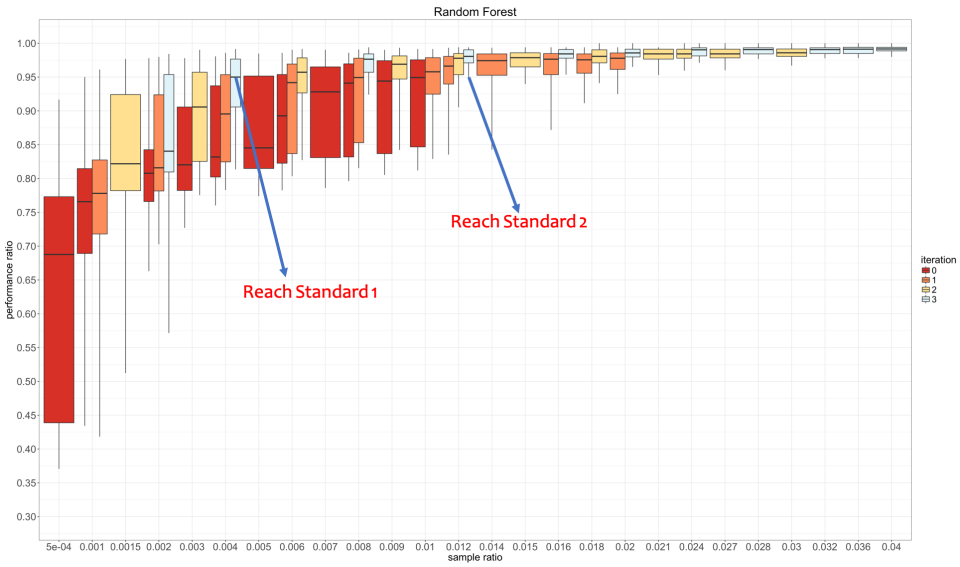

Figure 6: Performance of SYR2K benchmark with random forest (RF) and varying the number of iterations (see legend) using IterML and the total sample ratio (X-axis) on the V100 GPU

Figure 6 shows the box-plot for the performance of the SYR2K benchmark on a V100 GPU. It is tuned by our iterative pruning approach using the random forest model and varying the total sample ratio used. Each bar in the box-plot consists of at least 500 data points. The middle line of each bar represents the median (for *Standard 1*); the lower-bound labels the 5-percentile (for *Standard 2*). We only show the first three iterations in this plot since the data set is relatively small in this case. The red color represents the baseline results, which randomly selects the potential samples. We observe that, with the same number of total samples used, those with more iterations generally produce better performance. This means *our approach performs better with fewer samples in each iteration but with more iterations*. However, some models may require a certain minimum of samples in order to become accurate for prediction.

From the plot, we also observe that the light grey bar (i.e., 3 iterations) with total sample ratio 0.4% reaches *Standard 1*, while the baseline (red bar) requires at least 1%. With a total of 1.2% samples (i.e., 3 iterations of 0.4%), the light bar reaches *Standard 2*. In this paper, we generate this box-plot for all the benchmarks with different predicting models. We then compare the median or 5-percentile bar to 0.95 and collect the sample ratio required by each model to achieve *Standard 1* or *Standard 2*.

Figures 7, 8, 9, and 10 show the normalized total samples to reach *Standard 1* or *Standard 2* on the V100 and P100 GPUs, respectively. Depending on the performance distribution of different benchmarks, we need different sample ratios to achieve good performance. For comparison purposes, we normalized the number of the samples required by different models to the baseline (denoted as the "non" model, short for non-iterative ML model).

We observe that in most cases, using our iterative pruning approach saves approximately 40% to 80% search effort when choosing the best model. Due to the performance distribution in some cases, we only need a very small set of samples to achieve good performance (e.g., epcc).

We also observe that the performance of SVM and MLP, respectively, are not stable. Sometimes these two models perform even worse than the baseline, especially when the sample size is relatively small. We conjecture that these two models require a certain amount of data to be effective.

On the other hand, the other three models (CART, RF, and KNN) always require fewer samples to achieve good performance. We note that we used a cut-ratio of 50% by default in this case because the search space (i.e., GPU thread-block size) is relatively small. When dealing with a larger search space, we may increase this value to achieve higher search speed. However, this higher search speed could compromise the search quality; hence, there is a tradeoff here, which we discuss in our future work.

Figure. 11 shows the box-plot of the overall average normalized sample required to achieve *Standard 1* and *Standard 2* for different machine-learning (ML) models. The results are normalized to baseline, which exhaustively randomly chooses the parameter combinations as samples. We observe that, among the five popular ML models, random forest (RF) performs better and produces more stable results than all the other models. It only requires $\sim$ 1.5% samples to reach *Standard 1* on average.

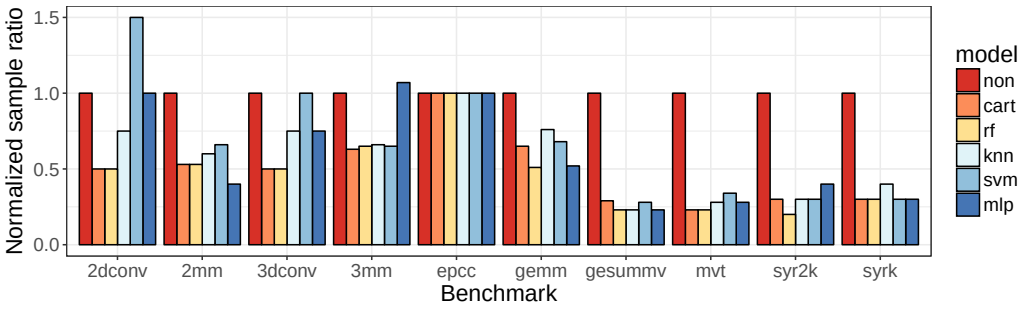

Figure 7: Normalized sample ratio to achieve Standard 1 on the V100 GPU (lower is better)

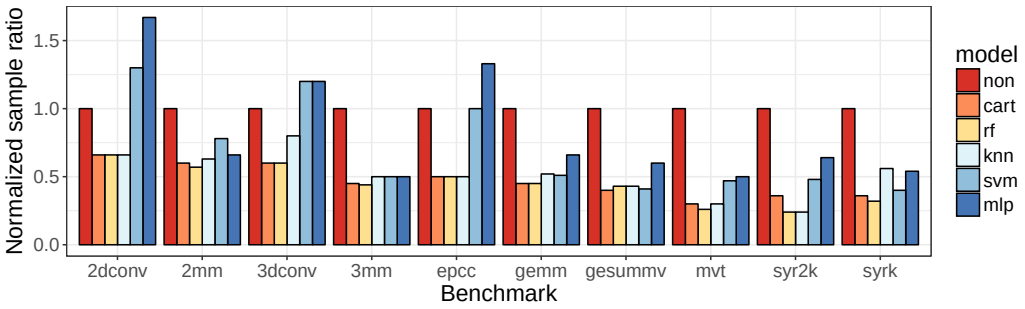

Figure 8: Normalized sample ratio to achieve Standard 2 on the V100 GPU (lower is better)

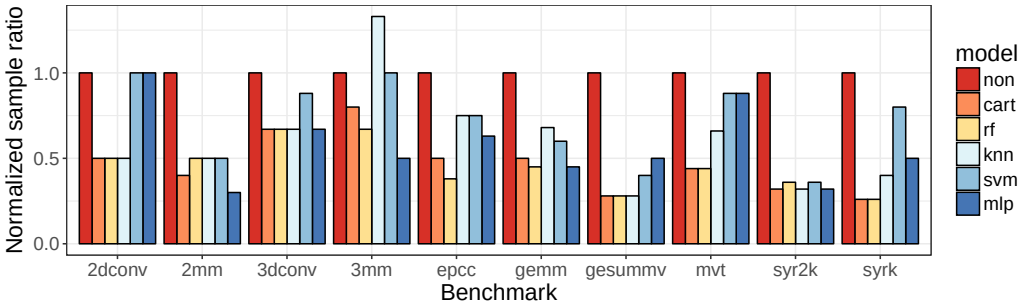

Figure 9: Normalized sample ratio to achieve Standard 1 on the P100 GPU (lower is better)

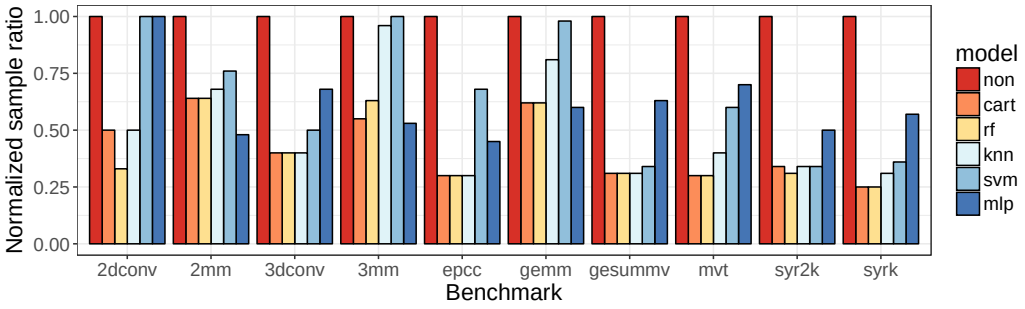

Figure 10: Normalized sample ratio to achieve Standard 2 on the P100 GPU (lower is better)

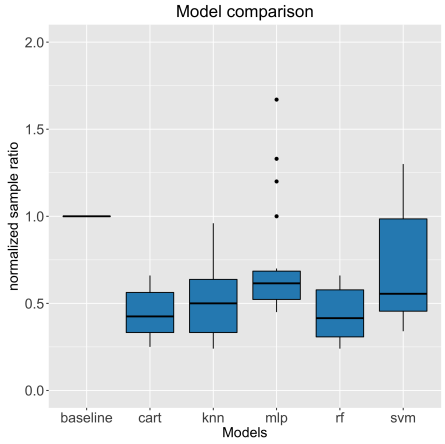

Figure 11: Comparison of machine-learning (ML) models for IterML, relative to the normalized sample ratio (lower is better)

In conducting this empirical study on our iterative machine learning (IterML) algorithm and comparing its performance to that of traditional non-iterative ML, we note that we used the default model functions provided by the scikit-learn library.

6 FUTURE WORK

In proposing our iterative machine-learning (IterML) approach to prune and tune hyperparameters for better performance, we utilized five machine-learning (ML) models for prediction in each iteration and parameterized our IterML approach with the *cut-ratio* and *pick-ratio*. The values for these two ratios should depend on the tradeoff between accuracy and search speed. Thus, future work encompasses conducting additional experiments to glean more insight as to how to adjust these two ratios under different circumstances. At present, we directly use the default ML model functions from the Python scikit-learn library. However, these ML models could benefit from extra hand-tuning to achieve better prediction results. Thus, we plan to continue studying and improving these models.

In addition, in this paper, we only evaluated our IterML approach to tune *one* system parameter of the GPU, namely the GPU thread-block size. Therefore, we plan to study additional parameters of relevance, including input data size and compiler flags, for example. Furthermore, due to the small set of benchmarks used and due to only using *one* vendor's hardware, namely NVIDIA, we plan to diversify the study further to include a broader set of benchmarks and greater breadth of hardware platforms.

7 CONCLUSION

In this work, we presented our iterative pruning approach with machine learning models (IterML) to auto-tune the performance of code running on accelerators, in particular, NVIDIA GPUs. In each iteration, we used machine-learning (ML) models to assist with pruning the rest of the parameter search space. Specifically, we focused on auto-tuning the GPU thread-block size.

Overall, our experiment results showed that IterML can significantly reduce the search effort by 40% to 80% compared to the traditional non-iterative ML approach. We also showed that the random forest (RF) model, in particular, better fits our IterML design than other models like SVM or MLP.

ACKNOWLEDGEMENT

This work was supported in part by the Air Force Office of Scientific Research (AFOSR) Computational Mathematics Program via Grant No. AFOSR Grant FA9550-17-1-0205.

REFERENCES

[1] Jee W Choi, Amik Singh, and Richard W Vuduc. 2010. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *ACM sigplan notices*, Vol. 45. ACM, 115–126.

[2] Xuewen Cui, Thomas RW Scogland, Bronis R de Supinski, and Wu-chun Feng. 2017. Directive-based partitioning and pipelining for graphics processing units. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*. IEEE, 575–584.

[3] Jack J Dongarra, Hans W Meuer, and Erich Strohmaier. 1994. Top500 Supercomputer Sites.

[4] Sunpyo Hong and Hyesoon Kim. 2009. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *ACM SIGARCH Computer Architecture News*, Vol. 37. ACM, 152–163.

[5] Kaixi Hou, Wu-chun Feng, and Shuai Che. 2017. Auto-tuning strategies for parallelizing sparse matrix-vector (spmv) multiplication on multi-and many-core processors. In *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*. IEEE, 713–722.

[6] Kaixi Hou, Hao Wang, and Wu-chun Feng. 2017. Gpu-unicache: Automatic code generation of spatial blocking for stencils on gpus. In *Proceedings of the Computing Frontiers Conference*. ACM, 107–116.

[7] Kaixi Hou, Hao Wang, Wu-chun Feng, Jeffrey S Vetter, and Seyong Lee. 2018. Highly Efficient Compensation-based Parallelism for Wavefront Loops on GPUs. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 276–285.

[8] Wenhao Jia, Kelly A Shaw, and Margaret Martonosi. 2013. Starchart: Hardware and software optimization using recursive partitioning regression trees. In *Parallel Architectures and Compilation Techniques (PACT), 2013 22nd International Conference on*. IEEE, 257–267.

[9] N Johnson. 2013. EPCC OpenACC benchmark suite.

[10] PJ Joseph, Kapil Vaswani, and Matthew J Thazhuthaveetil. 2006. Construction and use of linear regression models for processor performance analysis. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*. IEEE, 99–108.

[11] Rubao Lee, Hao Wang, and Xiaodong Zhang. 2018. Software-Defined Software: A Perspective of Machine Learning-Based Software Production. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1270–1275.

[12] Wenqiang Li, Guanghao Jin, Xuewen Cui, and Simon See. 2015. An evaluation of unified memory technology on nvidia gpus. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 1092–1098.

[13] Yan Li, Kenneth Chang, Oceane Bel, Ethan L Miller, and Darrell DE Long. 2017. CAPES: unsupervised storage performance tuning using neural network-based deep reinforcement learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 42.

[14] Yinan Li, Jack Dongarra, and Stanimire Tomov. 2009. A note on auto-tuning GEMM for GPUs. In *International Conference on Computational Science*. Springer, 884–892.

[15] Sparsh Mittal and Jeffrey S Vetter. 2015. A survey of methods for analyzing and improving GPU energy efficiency. *ACM Computing Surveys (CSUR)* 47, 2 (2015), 19.

[16] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *Journal of machine learning research* 12, Oct (2011), 2825–2830.

[17] Louis-Noël Pouchet. 2012. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/pouchet/software/polybench* (2012).

[18] Shane Ryoo, Christopher I Rodrigues, Sam S Stone, Sara S Baghsorkhi, Sain-Zee Ueng, John A Stratton, and Wen-mei W Hwu. 2008. Program optimization space pruning for a multithreaded GPU. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, 195–204.

[19] Nhat-Phuong Tran, Myungho Lee, and Jaeyoung Choi. 2017. Parameter based tuning model for optimizing performance on GPU. *Cluster Computing* 20, 3 (2017), 2133–2142.