IterML: Iterative Machine Learning for Intelligent Parameter Pruning and Tuning in Graphics Processing Units

Xuewen Cui · Wu-chun Feng

Received: date / Accepted: date

Abstract With the rise of graphics processing units (GPUs), the parallel computing community needs better tools to productively extract performance from the GPU. While modern compilers provide flags to activate different optimizations to improve performance, the effectiveness of such automated optimization has been limited at best. As a consequence, extracting the best performance from an algorithm on a GPU requires significant expertise and manual effort to exploit both spatial and temporal sharing of computing resources. In particular, maximizing the performance of an algorithm on a GPU 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 extremely large, making it infeasible to exhaustively evaluate.

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 models to guide the pruning and tuning for subsequent iterations. We evaluate our IterML approach on the GPU thread-block size across many benchmarks running on an NVIDIA P100 or V100 GPU. Our experimental results show that our automated IterML approach reduces search effort by 40% to 80% when compared to traditional (non-iterative) ML and that the performance of our (unmodified) GPU applications can improve significantly — between 67% and 95% — simply by changing the thread-block size.

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)

Xuewen Cui
Virginia Tech
Blacksburg, VA, USA
E-mail: xuewenc@vt.edu

Wu-chun Feng
Virginia Tech
Blacksburg, VA, USA
E-mail: wfeng@vt.edu

1 Introduction

Heterogeneous computing with accelerators, particularly GPUs, has become increasingly prominent in the Top500 List [3] as well as in embedded high-performance computing (HPC) systems, like those found in smartphones and smart cars. In such systems, the host CPU manages the execution context while computation is offloaded to an accelerator. Leveraging accelerators not only enables high performance, but it also improves energy efficiency [14]. 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 [2,11] as well as significant domain knowledge to tune the performance of the code on these accelerators [5,6].

Because heterogeneous architectures with accelerators expose many software and hardware parameters for developers to tune to achieve optimal performance, the different combinations of parameters result in an enormous 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. As such, this paper presents IterML, our iterative parameter pruning and tuning approach with machine

learning (ML). During each iteration, we use ML models to assist with the pruning (and tuning) of the search space by their predicted performance. In all, our research contributions are as follows:

- 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 illustrates how our IterML approach consistently delivers better search speed over non-iterative ML methods and achieves nearly optimal performance while sampling only 1.5% of the search space on average and, in turn, reducing the search effort by 40%-80%. In addition, when compared to the PGI 17.5 compiler, IterML also delivers about a 50% improvement in performance by automatically identifying a nearly optimal GPU thread-block size.

The rest of the paper is organized as follows. In §2, we motivate the need for this research and present 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. Next, §5 presents our experimental results. Finally, §6 outlines future work, and §7 concludes our work.

2 Motivation and Background

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,[1] where the GPU thread-block size is varied (i.e., blockDim.x $\times$ blockDim.y $\leq$ 512) when running on an NVIDIA V100 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 4x32 for *this* code on *this* GPU, the V100 achieves 893.3 GFLOPS. On the other hand, the performance can be 33% worse at 597.6 GFLOPS if the default thread-block size of the PGI 17.5 OpenACC compiler for this LDC code is chosen, namely 64x4. This 64x4 thread-block size only delivers approximately 67% of the optimal performance.

[1] A well-known computational fluid dynamics (CFD) problem for viscous incompressible fluid flow.

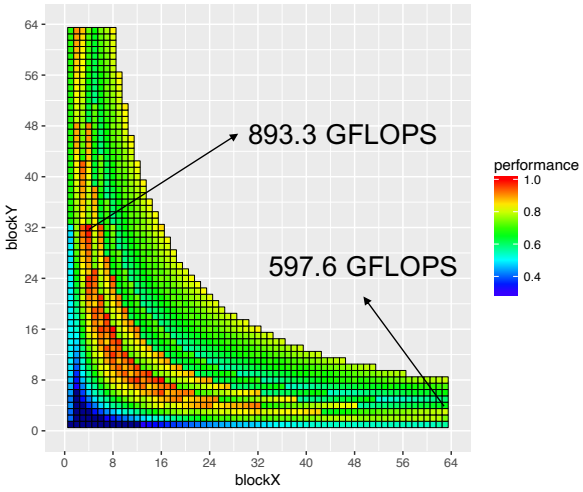

Fig. 1 Performance of a Lid-Driven Cavity Code with Varying GPU Thread-Block Size on an NVIDIA V100 GPU

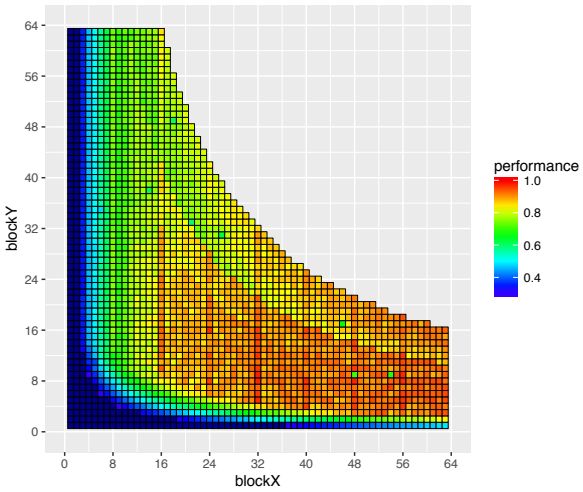

Fig. 2 Performance of the 2D Convolution Benchmark with Varying GPU Thread-Block Size on an NVIDIA V100 GPU

In addition, exhaustively generating the performance heatmap in Figure 1 is tedious and time-consuming.

Figure 2 shows the performance heatmap of a two-dimensional (2D) convolution benchmark. This heatmap looks significantly different from that in Figure 1. Thus, different codes on the same GPU can produce widely different performance characteristics, relative to GPU thread-block size.

Furthermore, the GPU thread-block size is just one parameter that can be tuned; there are many other potential parameters that could be tuned, e.g., GPU block size, degree of loop unrolling, register usage limitation, and so on. These assortment of parameters massively 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.

2.2 Related Work

In [1,18,13], the authors auto-tune the performance of a particular algorithm or application on an accelerator, like the GPU. However, their auto-tuning still requires extensive expertise (or intuition) to *manually* select the key parameters as well as the compiler flags. To address this problem, we propose an approach that automatically identifies a much smaller (pruned) search space that contains a (near-)optimal setting, which can then be searched. 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 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 [9], 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 [12], 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. While this approach can be deployed into a production system to collect training data and suggest tuning actions during the system's daily operation, it requires the system to be mostly static, which is not applicable to new algorithms or libraries that targe new devices like GPUs. While there exist pheromone models based on the profiling data of GPUs [4,17], they require large training sets across many programs and with a wide variety of performance counters. Moreover, they require developers to have intimate knowledge about the programs. Other related research focuses on designing coding machines to handle the programming tasks [10]. In contrast, our goal is to help developers productively tune their programs 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 (1) relate to software (e.g., input partition chunks and thread count) or hardware (e.g., active core count, GPU thread-block size, and compiler optimization options) and (2) be either binary in nature (e.g., turning on/off a compiler flag) or multi-valued across a range (e.g., thread-block size or number of partition chunks). Our iterative machine-learning (IterML) 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) Algorithm for Pruning and Tuning

In order to quickly and effectively reduce (i.e., prune and tune) the search space, we propose an iterative machine-learning (IterML) algorithm, as shown in Algorithm 1. As inputs, the algorithm takes the *search space D*, specified by multiple design parameters (e.g., thread-block dimension); a *pick ratio*, which is the sample ratio that needs to be tested in each iteration; a *cut ratio*, which sets the ratio of the space to be pruned in each iteration; and a *model*, which is the regression model used for prediction. Once the pick ratio is selected, the number of samples that we pick to test in each iteration stays constant. For each iteration, we first the apply the regression model to the samples. We then predict the performance of the residual search space and drop the lowest performing ones by the cut ratio, e.g., 50%. This process repeats until a stopping criteria is met. 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.

Figure 3 shows basic workflows for non-iterative ML and our iterative ML (IterML) with three iterations. We observe that a small portion of samples are randomly chosen in iteration 0 (i.e., *iter*0). Then the model is generated to use as a guide for subsequent iterations for data point selection in the residual search space. The corresponding performance of these data points are measured and then utilized to further build models for

Algorithm 1: Iterative Machine-Learning 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 the pick ratio;
  gather performance **P** of **S**;
  build model M_i, based on **P**;
  predict the remaining **D** based on model M_i;
  prune from the remaining **D** the samples with low predicted performance by an amount specified by the **cut ratio**;
end

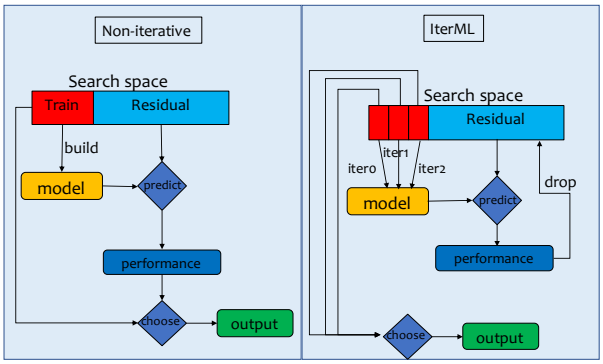

Fig. 3 Comparison of Non-Iterative and Iterative Machine Learning

next iterations. Finally, the best data point should be selected as the output result.

3.3 Regression Models

Based on our iterative machine-learning (IterML) approach, we need to build a model during each iteration to predict the *rest* of the search space. We study and explore the use of the following five popular 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 typically 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 (MLP): This 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) 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 typically 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 as a reference point to compare to, we exhaustively test the performance of different benchmarks using all the possible combinations of GPU thread-block size, a process that takes *days* to complete. We then demonstrate the speed and efficacy of our IterML pruning and tuning approach on the GPU thread-block size and evaluate its subsequent performance relative to the optimal performance.

4.1 Benchmarks Studied

As shown in Table 1, we use nine (9) GPU kernels from the Polybench suite [16], an OpenACC kernel from the EPCC benchmark suite [8], and an OpenACC kernel from our lid-driven cavity (LDC) code to conduct the our experiments. The kernels use various GPU functional units and exhibit diverse behavior. For CUDA benchmarks, relevant design parameters are substituted 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.

Table 1 Benchmarks Used

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

4.2 Hardware Platform

We conduct our experiments on two NVIDIA GPUs: Tesla P100 and Tesla V100. Our P100-based node contains two 2.4-GHz Intel E5-2680v4 CPUs for a total of 28 CPU cores per node. The V100 node pairs two 3.0-GHz Intel Skylake Xeon Gold CPUs for a total of 24 cores per node, In addition, there is 384 GB of memory and two NVIDIA V100 ("Volta") GPUs per node.

4.3 Dataset Analysis

We measure the performance of 11 benchmarks while varying the GPU thread-block size. We then generate a performance heatmap for each benchmark and conduct a preliminary data analysis. We find that the performance distribution of these benchmarks typically fall into two major categories: clustered or banded.

Clustered. Most of these benchmarks achieve higher performance when a specific GPU thread-block dimension gets higher (or lower) values, i.e., "clustered" high performance. As shown in Figure 2 and Figure 4, we observe better performance with higher blockX or blockY, respectively. In Figure 5, we see performance improvement with small blockY values.

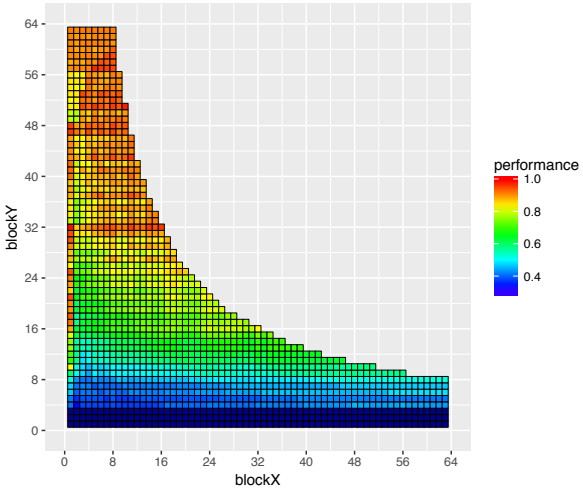

Fig. 4 Performance of the EPCC Benchmark with Varying GPU Thread-Block Size on an NVIDIA V100 GPU

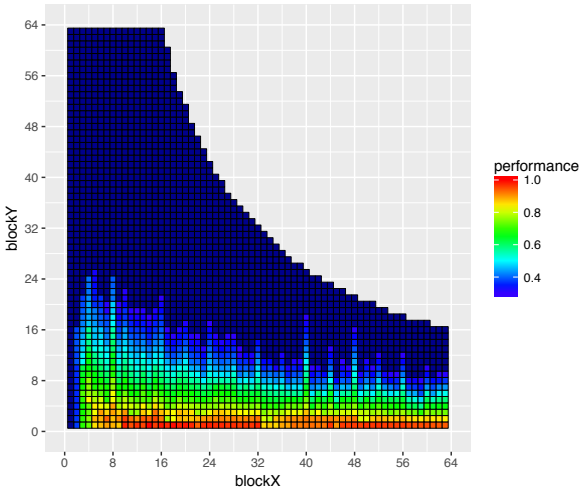

Fig. 5 Performance of the MVT Benchmark with Varying GPU Thread-Block Size on an NVIDIA P100 GPU

Figure 6 shows the SYR2K benchmark on the V100 GPU. Here the highest-performing configurations are those with a blockX equal to eight (8). The performance then degrades gradually as one moves away from the

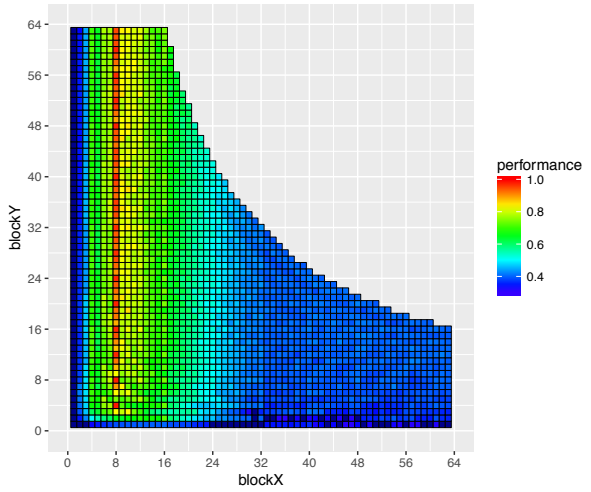

Fig. 6 Performance of the SYR2K Benchmark with Varying GPU Thread-Block Size on an NVIDIA V100 GPU

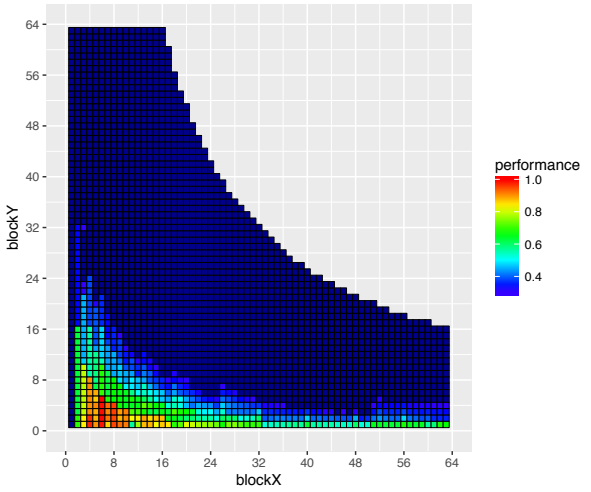

Fig. 7 Performance of the GESUMMV Benchmark with Varying GPU Thread-Block Size on an NVIDIA V100 GPU

blockX = 8 line. When the performance of a benchmark changes gradually along one axis (e.g., blockX or blockY) like this, we refer it as a "1D Cluster."

Similarly, there are other benchmarks that deliver high performance when the *product of blockX and blockY* reaches a specific number or range, meaning that the total number of GPU threads within a thread block should be limited in order to achieve the best performance. Visually, this translates into a "locus" of high performance, where performance degrades as thread-block size moves away from the "center of the locus." For example, as shown in Figure 1, the best-performing thread-block configurations occur when the total number of GPU threads in a thread block is about 128.

Finally, Figure 7 shows that the peak performance for the GESUMMV benchmark on a V100 GPU occurs near the lower-left corner of the heatmap, which means that many GPU thread blocks with a small number of GPU threads in each thread block delivers the best performance.

Banded. The banded performance distribution occurs for GPU programs that deliver peak performance when a specific GPU thread-block dimension reaches a specific number or multiple of it. As the thread-block size moves away from these specific numbers, performance degrades significantly. As shown in Figure 8 and Figure 9, the 2MM and GEMM benchmarks deliver the best performance only when blockX is a multiple of 16 and 8, respectively. For other values of blockX, the performance achieved is always below 60% of the performance achieved when blockX is a multiple of 16 and 8 for the 2MM and GEMM benchmarks, respectively.

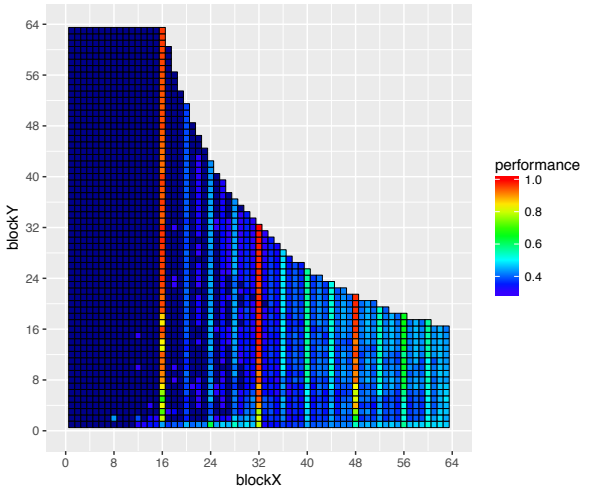

Fig. 8 Performance of the 2MM Benchmark with Varying GPU Thread-Block Size on an NVIDIA P100 GPU

Using the performance classifications of "clustered" and "banded," Table 2 shows the overall distribution of application performance when varying the GPU thread-block size. Interestingly, the performance distribution across the P100 GPU and V100 GPU is consistent. As a consequence, this feature might be useful as a guideline for further experimental work on future accelerators.

4.4 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

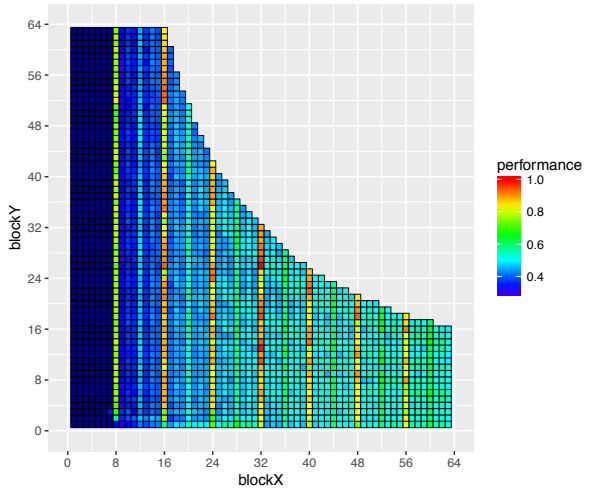

Fig. 9 Performance of the GEMM Benchmark with Varying GPU Thread-Block Size on an NVIDIA V100 GPU

Table 2 Distribution of Application Benchmark Performance

	P100	V100
Clustered	2dconv epcc 3dconv ldc gesummv mvt syrk syr2k	2dconv epcc 3dconv ldc gesummv mvt syrk syr2k
Banded	2mm 3mm gemm	2mm 3mm gemm

Python with the Scikit-learn machine learning (ML) libraries [15]. 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.

4.5 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 10 and 11 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 12 and 13 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.

Table 3 shows how the ideal thread-block size on the P100 GPU performs on the V100 GPU while Table 4 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.

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 predic-

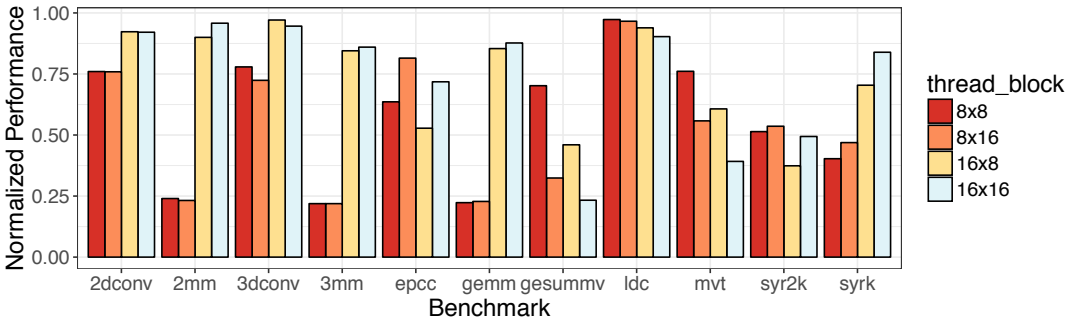

Fig. 10 Normalized Performance Across Varying Thread-Block Sizes on the P100 GPU

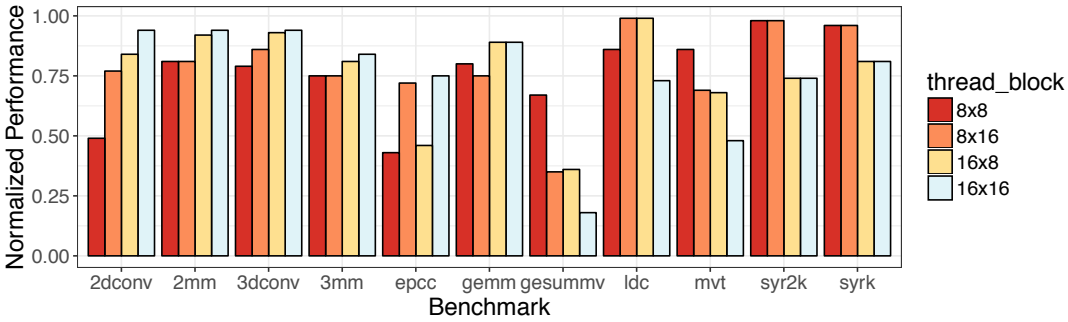

Fig. 11 Normalized Performance Across Varying Thread-Block Sizes on the V100 GPU

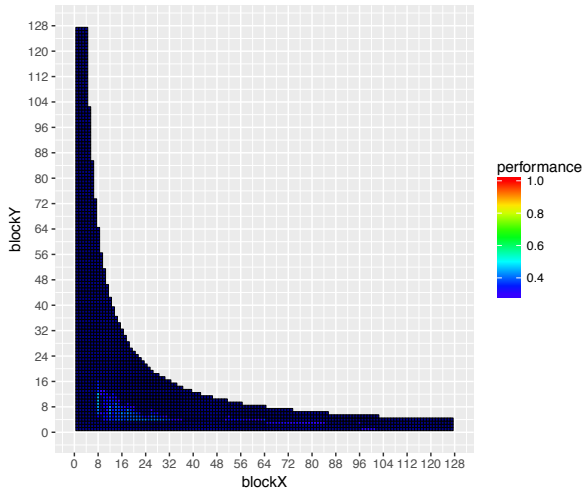

Fig. 12 Minimum performance heatmap across all benchmarks on V100 GPU

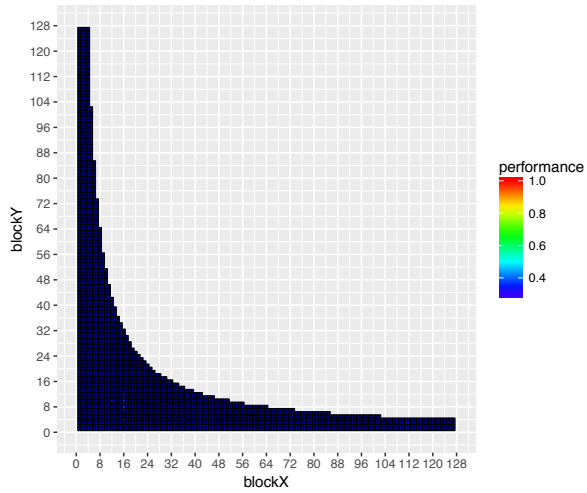

Fig. 13 Minimum performance heatmap across all benchmarks on P100 GPU

tion during each iteration and use random search sampling 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.

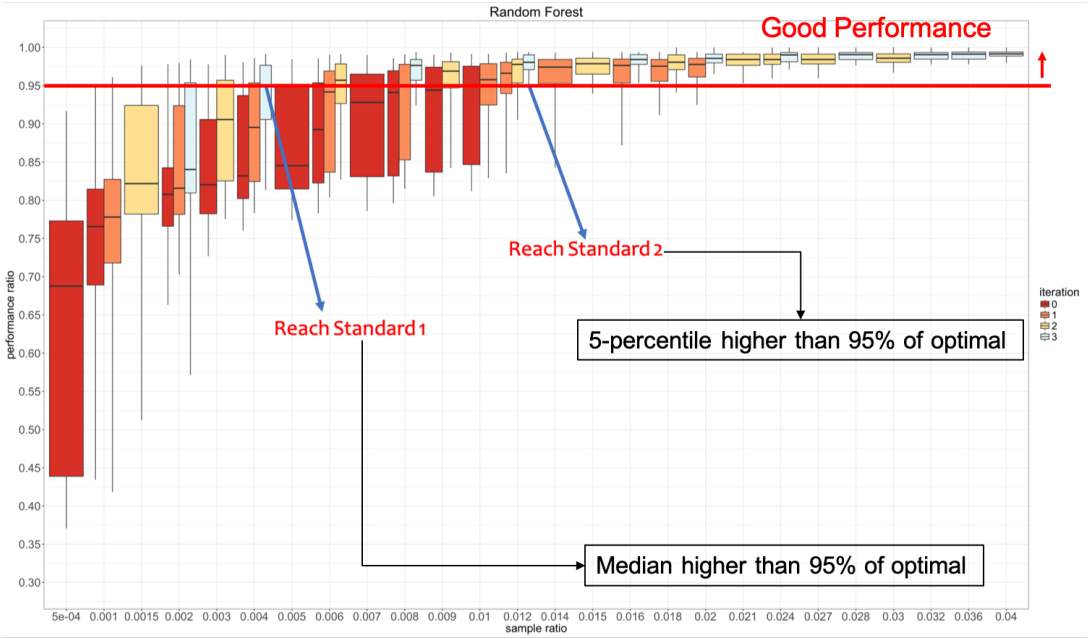

Fig. 14 Performance of SYR2K benchmark with random forest (RF) and varying the number of iterations (see legend) using IterML and the total sample ratio (X-axis) on the V100 GPU

Table 3 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 4 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

Figure 14 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 15, 16, 17, and 18 show the normalized total samples to reach *Standard 1* or *Standard 2* on the V100 and P100 GPUs, respectively. Depending on the

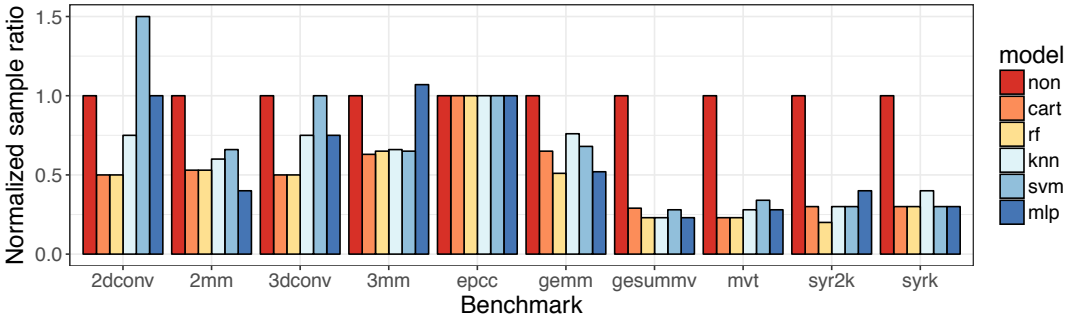

Fig. 15 Normalized sample ratio to achieve Standard 1 on the V100 GPU (lower is better)

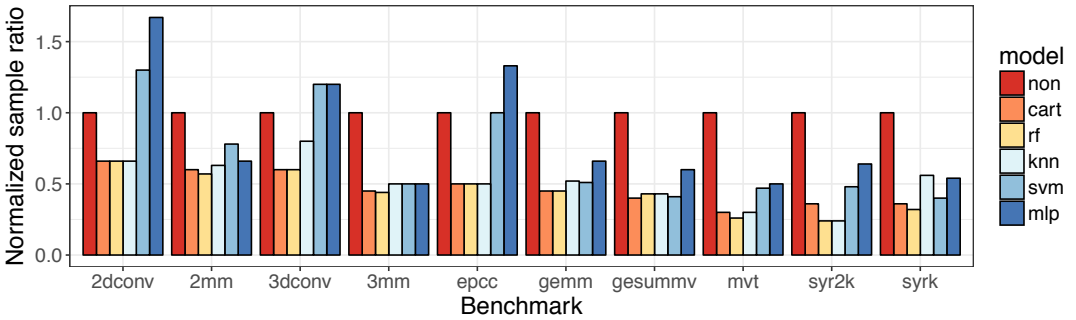

Fig. 16 Normalized sample ratio to achieve Standard 2 on the V100 GPU (lower is better)

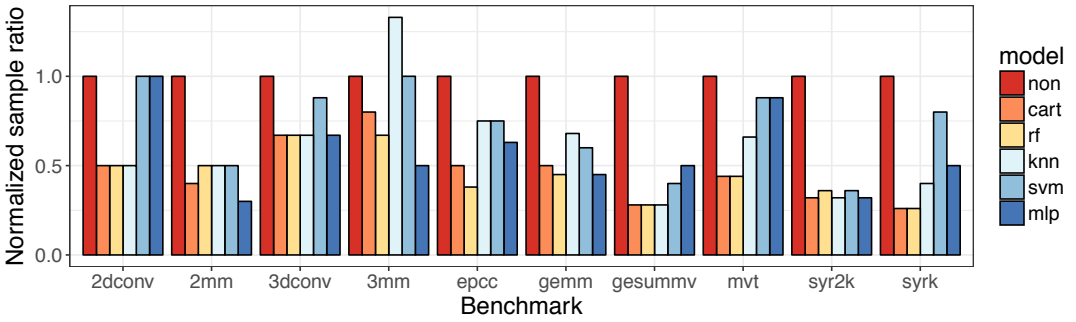

Fig. 17 Normalized sample ratio to achieve Standard 1 on the P100 GPU (lower is better)

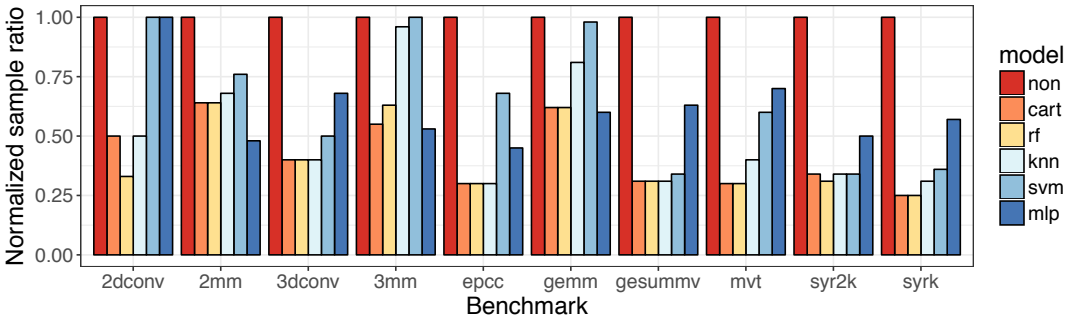

Fig. 18 Normalized sample ratio to achieve Standard 2 on the P100 GPU (lower is better)

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). In Figure. 17, we see that CART and RF models only require ∼20% samples compared to baseline. The worst case is 3MM benchmark, which takes ∼60% samples while using RF model. Overall, 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. 19 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. Compared to non-iterative approach (baseline), it saves ∼ 40% to 80% required samples to achieve good performance. Overall, it saves ∼ 60% required sample, which only requires ∼1.5% of the search space on average.

Previously, we categorized the benchmarks into two groups based on the performance distribution. Figure 20 shows their average raw sample ratio to achieve Standard 1, which means the performance expectation is higher than 95% of the optimal. We only compare between the non-iterative approach and our IterML with Random Forest model. We see that "The Band" obviously requires a lot more samples to get good performance, which is around 3% on average. Our IterML approach can significantly reduce the sample ratio , which is 1.7% on average. For both groups, the IterML always provide more stable performance than the non-iterative approach.

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.

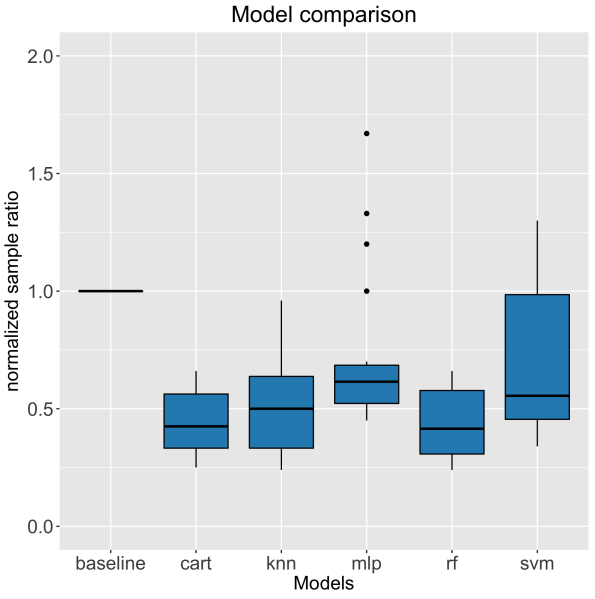

Fig. 19 Comparison of machine-learning (ML) models for IterML, relative to the normalized sample ratio (lower is better)

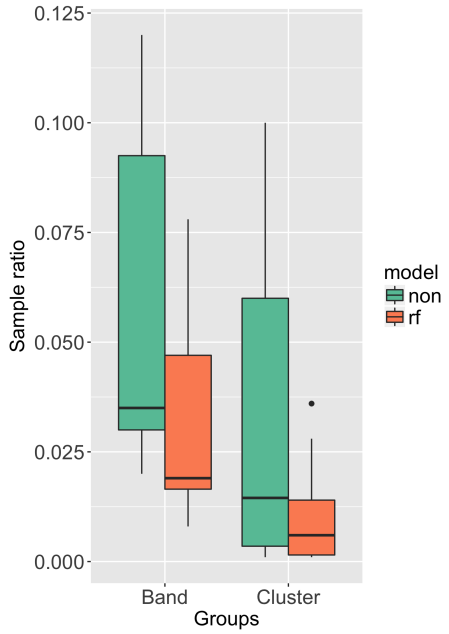

Fig. 20 Comparison of benchmark performance distribution group, relative to the raw sample ratio to achieve Standard 1 (lower is better)

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 autotune 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.

References

1. Choi, J.W., Singh, A., Vuduc, R.W.: Model-driven autotuning of sparse matrix-vector multiply on gpus. In: ACM sigplan notices, vol. 45, pp. 115–126. ACM (2010)
2. Cui, X., Scogland, T.R., de Supinski, B.R., Feng, W.c.: Directive-based partitioning and pipelining for graphics processing units. In: Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International, pp. 575–584. IEEE (2017)
3. Dongarra, J.J., Meuer, H.W., Strohmaier, E.: Top500 Supercomputer Sites (1994)
4. Hong, S., Kim, H.: An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In: ACM SIGARCH Computer Architecture News, vol. 37, pp. 152–163. ACM (2009)
5. Hou, K., Feng, W.c., Che, S.: 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, pp. 713–722. IEEE (2017)
6. Hou, K., Wang, H., Feng, W.c.: Gpu-unicache: Automatic code generation of spatial blocking for stencils on gpus. In: Proceedings of the Computing Frontiers Conference, pp. 107–116. ACM (2017)
7. Hou, K., Wang, H., Feng, W.c., Vetter, J.S., Lee, S.: Highly efficient compensation-based parallelism for wavefront loops on gpus. In: 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 276–285. IEEE (2018)
8. Johnson, N.: Epcc openacc benchmark suite (2013)
9. Joseph, P., Vaswani, K., Thazhuthaveetil, M.J.: Construction and use of linear regression models for processor performance analysis. In: High-Performance Computer Architecture, 2006. The Twelfth International Symposium on, pp. 99–108. IEEE (2006)
10. Lee, R., Wang, H., Zhang, X.: Software-defined software: A perspective of machine learning-based software production. In: 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), pp. 1270–1275. IEEE (2018)
11. Li, W., Jin, G., Cui, X., See, S.: An evaluation of unified memory technology on nvidia gpus. In: 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pp. 1092–1098. IEEE (2015)
12. Li, Y., Chang, K., Bel, O., Miller, E.L., Long, D.D.: 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, p. 42. ACM (2017)
13. Li, Y., Dongarra, J., Tomov, S.: A note on auto-tuning gemm for gpus. In: International Conference on Computational Science, pp. 884–892. Springer (2009)
14. Mittal, S., Vetter, J.S.: A survey of methods for analyzing and improving gpu energy efficiency. ACM Computing Surveys (CSUR) **47**(2), 19 (2015)
15. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al.: Scikit-learn: Machine learning in python. Journal of machine learning research **12**(Oct), 2825–2830 (2011)
16. Pouchet, L.N.: Polybench: The polyhedral benchmark suite. URL: http://www. cs. ucla. edu/pouchet/software/polybench (2012)
17. Ryoo, S., Rodrigues, C.I., Stone, S.S., Baghsorkhi, S.S., Ueng, S.Z., Stratton, J.A., Hwu, W.m.W.: Program optimization space pruning for a multithreaded gpu. In: Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization, pp. 195–204. ACM (2008)
18. Tran, N.P., Lee, M., Choi, J.: Parameter based tuning model for optimizing performance on gpu. Cluster Computing **20**(3), 2133–2142 (2017)