

Utilizing GPU Performance Counters to Characterize GPU Kernels via Machine Learning

Bob Zigon¹ and Fengguang Song²

¹ Beckman Coulter Inc., Indianapolis, Indiana, USA. bob.zigon@gmail.com

² Department of Computer Science, Indiana University-Purdue University Indianapolis, Indiana, USA. fgsong@cs.iupui.edu

Abstract. GPU computing kernels are relatively simple to write if achieving the best performance is not of the highest priority. However, it can quickly become a much more daunting task when users try to tune and optimize their kernels to obtain the highest performance. This is due to GPUs' massive degree of parallelism, complex memory hierarchy, fine grain synchronization, and long memory access latency. Hence, users must carry out the complex tasks of profiling, analyzing, and tuning to reduce performance bottlenecks. Today's GPUs can generate hundreds of performance events that comprehensively quantify the behavior of a kernel. Instead of relying on experts' manual analysis, this paper targets using machine learning methods to generalize GPU performance counter data to determine the characteristics of a GPU kernel as they will reveal possible reasons for low performance. We choose a set of problem-independent counters as our inputs to design and compare three machine learning methods to automatically classify the execution behavior of a kernel. The experimental results on stencil computing kernels and sparse matrix multiplications show the machine learning models' good accuracy, and demonstrate a feasible approach that is capable of classifying a kernel's characterizations and suggesting changes to a skilled user, who can subsequently improve kernel performance with less guessing.

Keywords: GPU computing · hardware performance counters · automatic performance analysis · machine learning for HPC

1 Introduction

When writing high performance kernels for a modern GPU, several guidelines must be followed. For instance, utilization of both host and GPU memory bandwidth should be maximized. The idle time of parallel computing resources within the GPU should be minimized. Finally, instruction and memory access latency should be either hidden or minimized.

After successfully writing a kernel, most often a GPU programmer is faced with optimizing the kernel's performance. GPU code, however, is notoriously difficult to optimize given the complexity of the underlying hardware. This begs an interesting research question: "*Can machine learning be used to aid in improving GPU kernel performance?*" We have seen that optimizing GPU kernels

is sufficiently difficult that only experts engage in the activity. For the average researchers it is both tiresome and tedious. In this paper, we design and develop a process that utilizes different machine learning (ML) techniques to generate insight into GPU performance tuning.

Our approach carefully creates ten classes of GPU kernels which have various performance patterns. Great numbers of executions with the ten classes using different parameters are then used as a dataset to train three distinct machine learning (ML) methods. The three methods include a deep neural network, a random forest, and a naive Bayes classifier. As for the particular machine learning *inference* stage, we use three other new kernels that the ML models have never seen so that the inference results can be compared and a good or poor ML model can be identified.

To the best of our knowledge, the main contributions of this paper are presented as follows.

1. A comprehensive machine learning software framework dedicated to evaluating the characteristics of GPU kernels using GPU hardware counters.
2. A software toolkit implemented for generating large amounts of training data for performance modeling.
3. Different machine learning models tailored and optimized to classify GPU kernels to help users tune kernel performance.

The rest of this paper is organized into the following sections. Section II describes related work while section III discusses background and GPU performance bottlenecks. Section IV describes our kernel classification and optimization process. Section V describes the three machine learning models and section VI presents experimental results. Finally, section VII discusses our conclusions.

2 Related Work

The process of program optimization requires first identifying the type of optimization to be performed. We choose to optimize execution time, that is, we strive to produce kernels that execute the fastest. However, optimization can also be performed with respect to power consumption. For instance, Antz [2] focused on optimizing matrix multiplication with respect to energy for GPUs.

A tool named BEAST (Bench-testing Environment for Automated Software Tuning) [9] was designed to autotune dense matrix-matrix multiplication. Using BEAST requires manually annotating a user's GPU program with the new language (The BEAST project has since been renamed BONSAI [12]). BEAST uses the following recipe for code optimization. First, a computational kernel is parameterized and implemented with a set of tunable parameters (e.g., tile sizes, compiler options, hardware switches), which generally defines a search space. Next, a number of pruning constraints are applied to trim the search space to a manageable size [6, 9]. Finally, those kernel variants that have passed the pruning process are compiled, run, benchmarked, and then the best performers are identified.

Abe [1] studied the problem of modeling performance and power by using multiple linear regression. They quantified the impact of voltage and frequency scaling on multiple GPU architectures. Their approach used power and performance as the dependent variables while using the statistical data obtained from performance counters as the source of the independent variables.

Lai [8] took a different approach. An analytical tool called TEG (Timing Estimation tool for GPU) was developed to estimate GPU performance. TEG takes the disassembled CUDA kernel opcodes as input, along with an instruction trace from a GPU simulator named *Barra*, and generates a predicted execution time. TEG is a static analysis tool and does not need to execute the user code.

In reflection, our work is different from the previous work. Instead of predicting execution time or power consumption, we aim to classify GPU kernels in order to aid users in improving performance. We achieve the goal by running a target GPU kernel, extracting all performance counters, classifying the kernel to a similar kernel class by ML models, and finally suggesting an optimization strategy.

3 Background on Performance Metrics

Given our use of performance counters and performance bounds, we start with some definitions. Modern GPUs provide between 100 and 200 hardware counters that can be collected during kernel execution. These counters are also referred to as *events*. On the other hand, another term called *metric* is used to represent a kernel's characteristic, which is calculated based upon one or multiple *events*. In this paper, we use the *metrics* as the training data input to our ML methods.

In general, the first step in analyzing a GPU kernel is to determine if its performance is bounded by memory bandwidth, computation, or instruction/memory latency.

A *memory bound* kernel reaches the physical limits of a GPU device in terms of accesses to the global memory.

A *compute bound* kernel is one in which computation dominates the kernel time, under the assumption that there is no issue feeding the kernel with memory and there is good overlap of arithmetic and latency.

Finally, a *latency bound* kernel is one whose predominant stall reason is due to memory access latency. The global memory bus is still not saturated. An example is when a kernel has to wait to retrieve an operand due to an inadequate number of kernel threads [7].

4 The Process of GPU Kernel Classification and Optimization Recommendation

Our paper focuses on three commonly used kernel optimization methods (as our first step). The methods are: 1) efficient use of memory bandwidth, 2) efficient use of compute resources, and 3) efficient hiding and reduction of instruction

and memory latency. Our future work plans to add more kernel optimization methods.

The whole process consists of six steps.

1. Design ten different kernel classes that exhibit different properties of GPU bottlenecks.
2. Generate the ten kernel classes' code samples.
3. Collect GPU performance counter data from running the generated code samples.
4. Create three ML classifiers.
5. Train the three ML classifiers using the performance counters.
6. Apply the best ML classifier to a new GPU kernel that needs to be optimized.

Steps 1 – 3 are mainly used for our training data generation. Each of the first three steps is described in the subsections that follow. Steps 4–6 are for our ML design and implementation and will be described in Section 5.

4.1 Designing Different Kernel Classes for Classification

Many problems in scientific computing can be described in terms of matrix computations. Popular operations include matrix-matrix multiplication, matrix-vector multiplication, and vector normalization, as well as stencil processing for Laplace partial differential equations.

The ten mathematical functions listed in Table 1 represent a list of commonly used GPU kernels, present in both matrix computations and scientific computing, that can be used to demonstrate memory bound, compute bound and latency bound behaviors.

The first six kernels (i.e, K1–K6) are revised from the BLAS Level 1 SAXPY functions but with increasing computation intensities. BLAS is a collection of functions, grouped into three sets, called Level 1, Level 2, and Level 3. Level 1 functions perform scalar, vector and vector-vector operations and have a computational complexity of $O(n)$. BLAS Level 2 functions perform matrix-vector operations of the form $y \leftarrow \alpha Ax + \beta y$, and have a computational complexity of $O(n^2)$. Finally, BLAS Level 3 performs matrix-matrix operations of the form $C \leftarrow \alpha AB + \beta C$, and have a computational complexity of $O(n^3)$.

The seventh kernel K7 in Table 1, naive matrix multiplication, is an example of a function that exhibits a memory access pattern more complex than the previous six. This kernel does not use shared memory (SMEM) on the GPU.

The eighth kernel K8 in Table 1 is an example of a BLAS Level 3 function that exhibits a memory access pattern more complex than the first six. This kernel tiles the shared memory over the matrix for higher performance than kernel K7.

Finally, kernels K9 and K10 in Table 1 are examples of a 5-point stencil. They represent an $O(\Delta x^2)$ finite difference approximation to the Laplacian of a function. Kernel K9 does not use shared memory while kernel K10 does.

The last column in Table 1, *Strategy Id*, shows the recommended strategies that can be used to improve kernel performance. Table 2 lists the strategy IDs and their corresponding descriptions.

Table 1: Functions of the ten classes of kernels that are used to train ML models along with the number of *training*, *validation* (val) and *test* samples. *SM* represents the percentage of the streaming multiprocessors utilized, *Mem* represents the percentage of memory bandwidth consumed, and *Strategy Id* describes the technique recommended to apply to a kernel to improve performance. The *SM* and *Mem* percentages were reported by NVidia’s NSight Compute tool.

Kernel Num	Kernel function	Train	Val	Test	SM	Mem	Strategy Id
K1	$z_i = x_i + y_i$	609	63	672	43%	84%	S4, S10, S20
K2	$z_i = K_1 x_i + K_2 y_i$	600	72	672	49%	83%	S4, S10, S20
K3	$z_i = \sin(K_1)x_i + \cos(K_2)y_i$	604	68	672	50%	24%	S4, S200
K4	$z_i = \sin(K_1)x_i + \cos(K_2)y_i$, ILP2, 32 bit read	606	66	672	49%	22%	S4, S200
K5	$z_i = \sin(K_1)x_i + \cos(K_2)y_i$, ILP2, 64 bit read	612	60	672	52%	52%	S1,S200
K6	$z_i = \sin(K_1)x_i + \cos(K_2)y_i$, ILP4, 128 bit read	611	61	672	45%	60%	S1,S200
K7	$C = AB$, not using shared memory (smem)	543	64	529	95%	22%	S100
K8	$C = AB + C$, uses smem, a tiled GEMM	523	63	528	92%	6%	S100
K9	$U_c^{t+1} = 0.25 \cdot \{U_n^t + U_s^t + U_e^t + U_w^t\}$ no smem	450	50	621	80%	43%	S20
K10	$U_c^{t+1} = 0.25 \cdot \{U_n^t + U_s^t + U_e^t + U_w^t\}$ smem	667	81	548	60%	63%	S1, S10, S30

Table 2: Suggested modifications that can be applied to a kernel to improve its performance.

Strategy ID	Strategy Description
S1	Low GPU occupancy – vary thread block size, vary shared memory usage
S4	Unroll loops and improve ILP
S10	Improve read and write memory coalescing
S20	Improve cache locality using registers or shared memory
S30	Reduce bank conflicts and cache into registers
S100	Save results to shared memory or global memory
S200	Use low occupancy and high ILP

Design Rationale behind Our Kernel Classes: The first kernel class K1 in Table 1 adds two single precision floating point vectors and stores the result. Note that K1 can be used to generate many K1 sample instances that use different parameters such as thread block size, input vector size and number of thread blocks. The simplicity of the function combined with the low arithmetic intensity causes the kernel to be memory bound.

The second kernel class K2 extends K1 by multiplying the vectors by two floating point constants. Since K1 is memory bound, the multiplications in K2 come at virtually no cost because they can be hidden by the memory accesses. K2 demonstrates how to get more work done per clock cycle [13]. If the PTX (GPU virtual machine language) for kernel K2 is generated as in Listing 1.1, the opportunity for overlapping instruction execution is lost. The GPU stalls when an operand is not available and not on a memory read. This means that after

line 1 issues, line 2 will stall because T1 is not available yet. The same pattern exists for lines 3 and 4.

Listing 1.1: Non Overlapping PTX

1.	T1 = X[i]
2.	T2 = T1*K1
3.	T3 = Y[i]
4.	T4 = T3*K2
5.	T5 = T2+T4
6.	Z[i] = T5

Listing 1.2: Overlapping PTX

10.	T1 = X[i]
20.	T2 = Y[i]
30.	T3 = T1*K1
40.	T4 = T2*K2
50.	T5 = T3+T4
60.	Z[i] = T5

To increase instruction throughput, the PTX for K2 needs to look like Listing 1.2. Line 10 will issue the memory read. Since the GPU does not block on memory reads, line 20 will then issue. Line 30 will stall because T1 is not likely to be available after line 20 issues. However, when T1 is available, it is likely that T2 is also available, so lines 30 and 40 will issue in parallel and line 50 will stall. As numeric expressions become more complex there can be many opportunities to overlap memory reads with arithmetic instruction execution.

In the third kernel class K3, the multiplicative constants K_1 and K_2 are replaced with the evaluation of two trigonometric functions. The result is that K3 stresses the floating point hardware with an arithmetic intensity level higher than kernel K2.

The fourth kernel class K4 extends K3 by exposing loop unrolling to instruction level parallelism (ILP). The loop in this SAXPY-like K4 kernel is first unrolled twice. The two vector accesses are then interleaved while performing 32-bit operand fetches. In fact, kernels K3-K6 each run faster than the previous kernel by exploiting different degrees of ILP and utilizing more memory bandwidth [13].

The fifth kernel class K5 is similar to K4 by way of loop unrolling. However, kernel K5 issues half as many memory instructions as K4 because the hardware is forced to generate 64-bit reads. This twice reduction in instruction count is important to instruction-bound or latency-bound kernels.

Kernel class K6 is similar to K5. However, K6 issues half as many memory instructions as K5 because the hardware is forced to generate 128-bit reads. Like kernel K5, this 2x reduction in instruction count is important to instruction-bound or latency-bound kernels because it increases memory bandwidth utilization.

Kernel K7, the naive matrix multiplication kernel, is needed because the memory access pattern aids in differentiating the kernel from the SAXPY like kernels. The addition of this kernel is consistent with the vector and matrix processing needs of scientific applications.

Kernel K8, the blocked BLAS Level 3 generalized matrix multiplication kernel, is added because it gave us an opportunity to use tile shared memory over the kernel arguments. The resulting memory access pattern is similar to kernel K7, but with much higher throughput, because the DRAM reads are cached in

shared memory. Like kernel K7, this kernel is consistent with the matrix processing needs of scientific applications.

The ninth kernel class K9, a 5-point stencil kernel, is a memory bound kernel with no reuse of its operands. For that reason, we added the tenth kernel class K10, which is a 5-point stencil that caches its operands in the GPU shared memory.

Our initial work in this paper studied ten classes of kernels related to general matrices, but new classes can be added to support other problem domains. For example, if users were to explore social relationships between individuals, we may add graph kernels to classify graph domain related kernels.

4.2 Generating the Training Dataset

All three of our machine learning models (to be described in Section V) require (\bar{X}_i, Y_i) pairs in order to be trained on. The vector \bar{X}_i consists of the GPU performance counters from the i^{th} run of some kernel sample. The value Y_i is the label associated with \bar{X}_i . Each label Y_i is one of the ten kernel numbers: K1 to K10.

Our ML models will be trained by using the performance counters collected from hundreds of runs of the ten kernel classes. To automate the process, we created a toolkit called *Datagen* that performed two functions. First, when *Datagen* was launched with the `-q` parameter, a function was called to write all of the different ways a given kernel could be launched, to a file. Table 3 shows an example of the output from that function. Lines 1 through 3 show *kernel_one* being launched with different block sizes (`-bs`) and grid sizes (`-gs`). The `-numele` parameter describes the size of the vector that will be manipulated by the kernel.

The second function of *Datagen* is to automatically execute a kernel with the parameters that were passed in as shown in Table 3. The GPU profiler would launch *Datagen*, which in turn launched the target kernel, so that the performance counters could be collected.

Table 3: Example output from the *Datagen* query functions.

```
kernel_one, params -bs 16,16 -gs 10,1 -numele 10000
kernel_one, params -bs 16,18 -gs 10,1 -numele 10000
kernel_one, params -bs 16,20 -gs 20,2 -numele 10000
```

4.3 Collecting Performance Counter Data

Figure 1 shows a flow chart that describes our process to collect the performance counter data. First, all of the kernels were queried so that their launch parameters could be written to a file called `TestConfig.txt`. Then, a python script opened `TestConfig.txt`, parsed each line, and launched *nvprof* with the *Datagen* executable, the kernel name, and the kernel configuration.

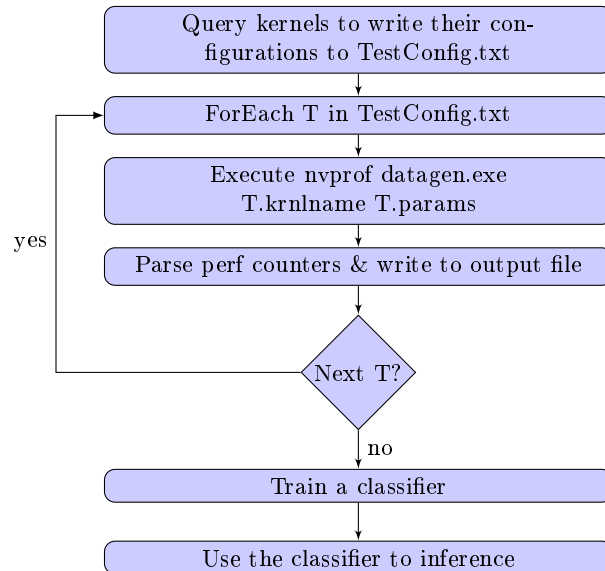


Fig. 1: Flow chart for performance counter data collection process.

5 Designing Machine Learning Models

In this section, we introduce the libraries we have used, and the three different ML methods we designed to perform GPU kernel classifications, which consist of the neural network model, the random forest model, and the naive Bayes model.

5.1 Performance Counters Used

In our experiments, we use the TITAN V GPU, which exposes 158 performance counters. Our Datagen can collect all of the counters, but the nature of our ten kernel classes means that only 95 of them would be non-zero. For example, our kernels operate on single precision floating point numbers and do not call atomic operations. As a result, the double precision floating point counters and atomic transaction counters are all zero.

There are three types of performance counters [7]. The first type is *absolute count*. These counters, for example, will record the total number of memory read and write requests. The second type is efficiency or *utilization counters*. These counters typically have a value between 0 and 1. They summarize the behavior of the hardware. For example, the `sm_efficiency` counter measures the overall activity of the SM's. Finally, there are *throughput counters*. These throughput counters typically measure the throughput of the various memory subsystems within the GPU. For example, the `gld_requested_throughput` counter measures the throughput of requested global loads in gigabytes per second.

5.2 Software Libraries Used

We designed and implemented our neural network model with Python 3.6.8, Keras, and Tensorflow 1.10.0 [10]. Our random forest and naive Bayes model are implemented with scikit-learn 0.20.2 [5]. Scikit-learn supports classification, regression, clustering, dimensionality reduction, and model evaluation.

5.3 Normalization of Performance Counters

In preparation for ML model training and accommodating various scales of metric values, we normalized our collected performance counters to the interval $[-1, 1]$ using the transform $F(x_{i,j}) = 2(x_{i,j} - \mu_j)/\sigma_j - 1$, where $x_{i,j}$ is the i^{th} measurement of the j^{th} feature, μ_j is the mean of the j^{th} feature, and σ_j is the standard deviation of the j^{th} feature. The random forest and naive Bayes methods do not require this step. The neural network method, however, must have all of the training data on the same scale.

5.4 The Neural Network Model Design

When doing experiments, we found that the efficiency and throughput performance counters produced better modeling results while the absolute counters negatively impacted our models. In other words, summarization information that efficiency and throughput counters offered caused ML models to generalize better. The absolute counters, on the other hand, mislead the ML classifiers. We therefore modified our methods to train on the union of the non-zero counters that only belong to the efficiency or throughput type. The result is a neural network with 48 inputs instead of 158.

The designed neural network is shown in Figure 2. The input layer, consisting of 48 performance counters, was followed by a fully connected layer of 48 neurons and a ReLU activation function, another fully connected layer of 24 neurons and a ReLU, and finally a softmax activation function that generates the probability for each of the 10 classes [3, 11].

We elected to train against all of the performance counter data using 7-fold cross validation for each of 7 different optimizers: RMSprop, SGD (stochastic gradient descent), Adagrad, Adadelta, Adam, Adamax and Nadam. Figure 3 shows that Nadam produced the best results.

5.5 The Random Forest Model Design

Our second machine learning model is a random forest classifier [4], which is based upon a collection, or *ensemble*, of binary decision trees where the probability of each class is the average of the probabilities across all trees.

Decision trees are easy to interpret and understand. They can be visualized, and unlike neural networks, the training data does not need to be normalized before being processed by the algorithm. The drawback to a decision tree is that

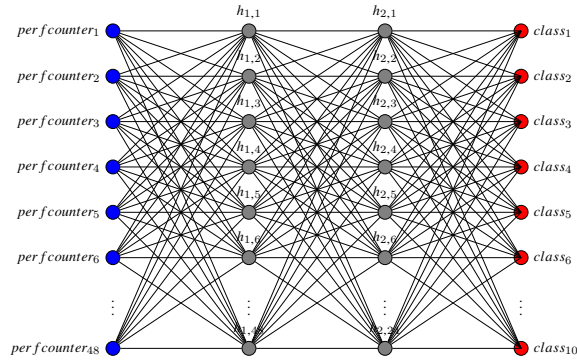


Fig. 2: Our neural network consists of 48 inputs, 2 hidden layers fully connected, and an output layer with 10 classes.

they can become overly complex which results in overfitting and poor generalization. Random forests manage the overfitting by utilizing hundreds or thousands of relatively simple decision trees. Our random forest consisted of 500 decision trees, each with a maximum depth of 6.

5.6 The naive Bayes Model Design

The naive Bayes classifier uses Bayes theorem to predict the class for a new test instance \mathbf{x} . The predicted class for \mathbf{x} is given as

$$\hat{y} = \arg \max_{c_i} \{P(c_i|\mathbf{x})\}, \quad (1)$$

where a training dataset \mathbf{D} consists of n points \mathbf{x}_i in a d -dimensional space, and y_i denote the class for each point, with $y_i \in \{c_1, c_2, \dots, c_k\}$. The posterior probability $P(c_i|\mathbf{x})$ for each class c_i is given by

$$P(c_i|\mathbf{x}) = \frac{P(\mathbf{x}|c_i) \cdot P(c_i)}{P(\mathbf{x})} \quad (2)$$

where $P(\mathbf{x}|c_i)$ is the *likelihood*, $P(c_i)$ is the *prior probability* of class c_i , and $P(\mathbf{x})$ is the probability of observing \mathbf{x} from any of the k classes, given as

$$P(\mathbf{x}) = \sum_{j=1}^k P(\mathbf{x}|c_j) \cdot P(c_j). \quad (3)$$

6 Experimental Results with the Three Different ML Methods

In this section we show the results from the experiments conducted using the three models. Table 1 shows the distribution of training, validation and testing

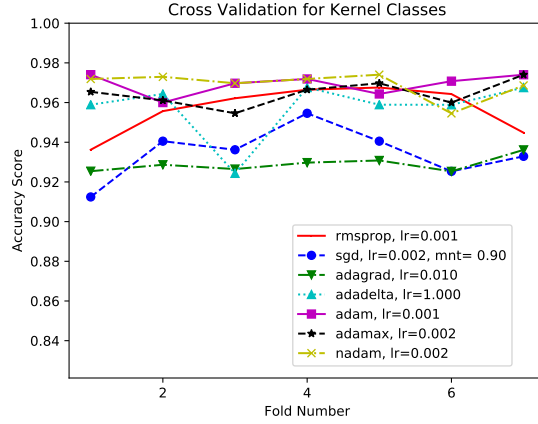


Fig. 3: 7-fold cross validation with 7 optimizers shows that the Nadam optimizer produced the best results.

samples that were used. Each of the GPU kernels was compiled using Visual Studio 2015 and CUDA 10.0 on a quadcore, Lenovo D30 ThinkStation running Windows 7/64. NSight Visual Studio 6.0.0.18227 was the visual profiler used. The GPU kernels were subsequently executed on an NVIDIA TITAN V GPU with device driver version 417.01.

For each ML model we generate a *confusion matrix*. The confusion matrix shows the ways in which a classification model fails when it makes predictions. The confusion matrices are shown in Figures 4, 5a and 5b. Based on the confusion matrices, we observe that all three models are nearly identical with the exception of kernels K3 and K4. In this case, the random forest has the largest normalized score.

In Table 4 the F1-score for the random forest is mostly larger when compared to the other classifiers. The F1-score [11] measures a model’s accuracy as function of precision and recall.

6.1 Evaluation of the Inference Results

Inferencing is the process of using a ML model to generate a category for an input the ML model has never seen. In our experiments, we use three test kernels that our models have never seen.

The first test kernel is a 9-point stencil shown in equation 4. This equation represents a $O(\Delta x^4)$ finite difference approximation to the Laplacian.

$$U_{center}^{t+1} = \frac{1}{120} \{4U_n^t + 4U_s^t + 4U_e^t + 4U_w^t + U_{ne}^t + U_{nw}^t + U_{se}^t + U_{sw}^t\} \quad (4)$$

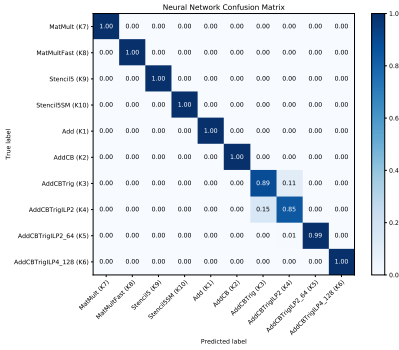


Fig. 4: Neural network confusion matrix.

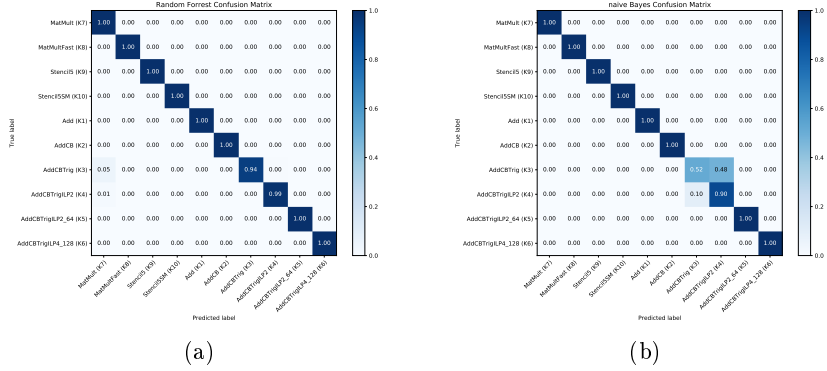


Fig. 5: Random forest and naive Bayes confusion matrices.

The second test kernel implements a sparse matrix vector multiplication (SpMV). The matrix is in compressed sparse row format and has a sparsity of 5% (i.e. 95% of the values are non-zero).

The third test kernel is a version of the 9-point stencil kernel that uses shared memory to cache variables from DRAM. If the models are doing a good job of generalizing, then we are postulating that the 9-point kernels will be mapped to the 5-point kernel class used in training.

Table 5 shows the inference results for our three ML models given the three test kernels. Each ML model was presented with ten different instances of the three kernels. All three models classified the 9-point stencil as the 5-point stencil, and the 9-point stencil with shared memory as the 5-point stencil with shared memory.

For the third test kernel, SpMV, the behavior of the models starts to diverge. The naive Bayes model classified SpMV as the 5-point stencil with shared memory. The random forest model classified SpMV as a matrix multiplication and a

Table 4: F1 score results for the three classifiers.

Kernel Name	NN	Random Forest	naive Bayes
MatMult (K7)	1.00	0.96	1.00
MatMultFast (K8)	1.00	1.00	1.00
Stencil5 (K9)	1.00	1.00	1.00
Stencil5SM (K10)	1.00	1.00	1.00
Add (K1)	1.00	1.00	1.00
AddCB (K2)	1.00	1.00	1.00
AddCBTrig (K3)	0.87	0.97	0.64
AddCBTrigILP2 (K4)	0.86	0.99	0.76
AddCBTrigILP2_64 (K5)	1.00	1.00	1.00
AddCBTrigILP4_128 (K6)	1.00	1.00	1.00

5-point stencil with shared memory. Finally, the neural network model classified SpMV as being predominantly similar to kernel K3, $z_i = \sin(K_1)x_i + \cos(K_2)y_i$. The divergent results on SpMV may be caused by the specialty of the SpMV kernel, which is not only similar to matrix multiplication (K7), but also similar to K3 because SpMV consists of a collection of vector-vector dot products. Based on the result of SpMV, we can say that the neural network and random forest models perform better than the naive Bayes model.

7 Conclusion

In this paper, we have described a machine learning framework that was used to classify unseen GPU kernels into one of ten classes using machine learning and GPU hardware performance counters. With that classification information, users are able to pursue optimization strategies for their target kernel based on the strategies for the learned kernels in Tables 1 and 2. What is critical in this paper is that each of the ten kernel classes we selected for training purposes was each slightly more complex than the previous kernel. This devised additional

Table 5: Inference results for our three ML models on three new test kernels. The green values show where the ML algorithms agree in their classification. The red values show where they disagree.

ML Model	Test Kernel	MM (K7)	K8	Sten5 (K9)	Sten5SM (K10)	K1	K2	AddCBTrig (K3)	AddCBTrigILP2 (K4)	K5	K6
naive Bayes	Stencil9			10							
Random Forest	Stencil9			10							
Neural Net	Stencil9			10							
naive Bayes	Stencil9SM				10						
Random Forest	Stencil9SM				10						
Neural Net	Stencil9SM				10						
naive Bayes	SpMV05				10						
Random Forest	SpMV05	5			5						
Neural Net	SpMV05						8		2		

complexity gradually utilized more and more of memory bandwidth and parallel resources, while minimizing the instruction and memory access latencies. We applied our framework to three new GPU kernels that are widely used in many domains, i.e. the 9-point stencil (with and without shared memory) used to discretize the Laplacian operator in differential equations, and the sparse matrix vector multiplication procedure. We also found that the random forest model and the neural network model performed similarly with respect to the confusion matrix, the F1-score and the actual classification results.

Finally, this framework can be extended by adding more kernel classes in our training dataset to support classifying larger, more complex GPU kernels or kernel sections. Our future work, on the other hand, will seek to improve and extend the three ML models to support new types of GPU applications.

References

1. Abe, Y., Sasaki, H., Kato, S., Inoue, K., Edahiro, M., Peres, M.: Power and performance characterization and modeling of gpu-accelerated systems. In: IPDPS. pp. 113–122 (2014)
2. Anzt, H., Haugen, B., Kurzak, J., Luszczek, P., Dongarra, J.: Experiences in autotuning matrix multiplication for energy minimization on gpus. *Concurrency and Computation: Practice and Experience* **27**(17), 5096–5113 (2015)
3. Bishop, C.M., et al.: *Neural networks for pattern recognition*. Oxford university press (1995)
4. Bowles, M.: *Machine learning in Python: essential techniques for predictive analysis*. John Wiley & Sons (2015)
5. Géron, A.: *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems*. " O'Reilly Media, Inc." (2017)
6. Haugen, B., Kurzak, J.: Search space pruning constraints visualization. In: *Software Visualization (VISSOFT), 2014 Second IEEE Working Conference on*. pp. 30–39. IEEE (2014)
7. Kim, H., Vuduc, R., Bagsorkhi, S., Choi, J., Hwu, W.m.: Performance analysis and tuning for general purpose graphics processing units (gpgpu). *Synthesis Lectures on Computer Architecture* **7**(2), 1–96 (2012)
8. Lai, J., Sez nec, A.: Break down gpu execution time with an analytical method. In: *Proceedings of the 2012 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*. pp. 33–39. ACM (2012)
9. Luszczek, P., Gates, M., Kurzak, J., Danalis, A., Dongarra, J.: Search space generation and pruning system for autotuners. In: *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*. pp. 1545–1554. IEEE (2016)
10. McClure, N.: *TensorFlow machine learning cookbook*. Packt Publishing Ltd (2017)
11. Murphy, K.P.: *Machine learning: a probabilistic perspective*. MIT press (2012)
12. Tsai, Y.M., Bachstein, M., Luszczek, P., Kurzak, J., Anzt, H., Gates, M., Dongarra, J.: Bonsai. In: *ISC 2018, Frankfurt Am Main, Germany (Jun 24-28 2018)*, https://ssl.linklings.net/conferences/isc_hpc/assets/2018/posters/post101.pdf
13. Volkov, V.: Better performance at lower occupancy. In: *NVIDIA GPU Conference 2010*. NVidia (Sep 2010)