SNOWPACK: Efficient Parameter Choice for GPU Kernels via Static Analysis and Statistical Prediction

Ranvijay Singh
Purdue University
singh406@purdue.edu

Paul Wood
Purdue University
pwood@purdue.edu

Ravi Gupta
Intel Corporation
ravigupta1989@gmail.com

Saurabh Bagchi
Purdue University
sbagchi@purdue.edu

Ignacio Laguna
Lawrence Livermore National Laboratory
ilaguna@llnl.gov

ABSTRACT

The running time of GPU kernels depends on an invocation parameter, the number of threads in each thread block. Sometime the dependence is quite strong leading to 50-100% change in execution time for long-running kernels. Till now, it has been an art form to decide on the optimal setting for this parameter. NVIDIA provides a tool for CUDA kernels, called OCC, that guides a developer toward this goal. In this paper, we show that OCC maximizes occupancy of GPU cores but does not meet the performance goal in a wide class of applications. We develop a solution called SNOWPACK that uses static features in a statistical learning framework to choose the optimal block size parameter. It does this without needing to execute the kernel multiple times, as a possible alternate solution Autotuner does. We evaluate our solution, SNOWPACK on 89 kernels of 10 applications.

CCS CONCEPTS

• **Computing methodologies** → *Support vector machines*; Graphics processors; • **Software and its engineering** → *Massively parallel systems*; *Software performance*;

KEYWORDS

GPGPU, Parameter Estimation, CUDA, Block Size, Machine Learning, Support Vector Machine

ACM Reference Format:
Ranvijay Singh, Paul Wood, Ravi Gupta, Saurabh Bagchi, and Ignacio Laguna. 2017. SNOWPACK: Efficient Parameter Choice for GPU Kernels via Static Analysis and Statistical Prediction. In *Proceedings of ScalA17: 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA17)*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3148226.3148235

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ScalA17, November 12–17, 2017, Denver, CO, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5125-6/17/11...$15.00
https://doi.org/10.1145/3148226.3148235

1 INTRODUCTION

Modern day heterogeneous computing often employs accelerators in order to offload and accelerate computation with GPUs being the most popular accelerators in use today in supercomputing clusters. Unlike CPUs, which have fewer cores that are optimized for serial execution, GPUs have a large number of cores (a few thousands) that are suited for highly parallel tasks. Program blocks that need to be executed in parallel on GPUs are typically divided into special functions called *kernels*, which are invoked from the CPU. In the CUDA programming model devised by NVIDIA, programmers must provide two parameters to each kernel invocation, namely, block size and grid size. These two parameters together determine the total number of parallel threads that will be executed on the GPU. The number of threads that are to be grouped together into one block is called the thread block size, or simply, the *block size*[1].

The thread block is an important unit of execution in GPUs—all the threads in a block run on a single GPU multiprocessor (the NVIDIA-specific term is "streaming multiprocessor"), can access a common shared memory, and can synchronize easily. As all the threads in a block share certain resources, such as shared memory and registers, varying the block size can result in huge performance variation. For example, on a sample PathFinder kernel from the popular Rodinia benchmark, we observe an execution time increase of 87.5% between an optimal block size selection (480 threads) and a non-optimal selection (96 threads). Most applications have one or a few key kernels, where the application spends most of its time. For such key kernels, it is important to understand the characteristics of the kernel, the GPU it is running on, and the input data size it is being run on to choose a block size that will result in optimal performance. This is a daunting programmability challenge, as has been expressed multiple times in developer forums [5, 6].

Need for tuning block size

There are two competing pulls that determine an optimal block size. Increasing the number of threads in a block helps to keep all the cores in a single GPU multiprocessor busy, which increases utilization of the cores and is thus desirable. Further,

[1] The other parameter, grid size, is the arrangement of blocks in a grid, such as, 16×1×1 and 4×4×1 are respectively a one-dimensional and a two-dimensional arrangement of 16 blocks in the grid.

with the possibility of scheduling multiple such thread blocks on a multiprocessor, it is possible to hide the effect of memory latency if a thread accesses far-off memory. On the other hand, threads in the same block share some resources, primarily the register file and the shared memory, a small software-managed data cache attached to each multiprocessor, shared among its cores. Thus, beyond a point, higher occupancy of the multiprocessor does not translate to higher performance.

Furthermore, for applications executing on a large GPU cluster, determining the optimal block size is important since even small differences in the kernel runtime might lead to significant divergences in application runtime between applications with and without optimal block sizes. Also, the prediction itself needs to be made without a significant overhead lest the benefit of faster executing kernels be lost due to the long time required for predictions. It would also free the programmer writing the kernel from choosing the block size, which can be cumbersome, especially for applications with many kernels.

Current Solutions

The two methods for determining a suitable block size for CUDA kernels available to a developer today are the NVIDIA Occupancy Calculator and the more research-based approach of Autotuning.

NVIDIA Occupancy Calculator (OCC). This tool, which is an Excel spreadsheet, allows the developer to compute the multiprocessor occupancy of a GPU by a given CUDA kernel. The multiprocessor occupancy is defined as the ratio of active warps[2] to the maximum number of warps supported on a multiprocessor of the GPU. With user input about the amount of shared memory used by a thread block, the number of registers used by each thread in a block, and the generation of GPU being used (which maps to the resources available on a multiprocessor), the OCC gives the occupancy as a function of the block size. The guidance to the developer is to select a block size that maximizes the occupancy. For example, we show in Figure 1, the OCC output for the PathFinder kernel in the Rodinia benchmark. This graph has 5 points where the peak occupancy is achieved and there is no method available to choose among these block sizes to achieve the highest performance. Tellingly, a block size corresponding to maximum occupancy (block size = 128) gives an execution time 37.5% higher than with the optimal block size.

Autotuning. Autotuning is a general optimization technique, which is used to find the optimal parameters in a complex system through directed search [10]. This can be adapted to our problem definition, to find the optimal block size that will minimize the kernel running time. However, a significant caveat is that Autotuning requires the kernel to be executed multiple times when it is searching for the optimal parameter value. This might cause unacceptable delays at runtime, especially if the program calls the same kernel multiple times with minor variations, *e.g.*, by changing the input problem size to the kernel. If on the other hand, the kernel is being

[2] A warp is a set of threads that execute in lock step on one multiprocessor. A typical size is 32 threads.

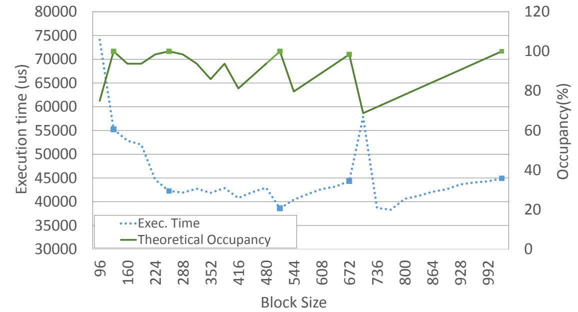


Figure 1: Output of the NVIDIA Occupancy Calculator for the PathFinder kernel from the Rodinia benchmark, plus the execution time of the kernel, for different block sizes. It shows that the block sizes that maximize occupancy do not lead to best performance.

invoked multiple times with the same parameters, then the cost of auto-tuning in the initial invocation can be amortized through the use of the optimal configuration in subsequent invocations.

Our Solution: SNOWPACK

We are motivated by the insight that to determine the optimal block size, we need to capture the characteristics of the work done by the kernel. It is attractive to use static features because they can be gathered offline and a model built on these features. Then, at runtime, the model can be used to predict what is the optimal block size to use to minimize running time of the kernel. Our solution SNOWPACK (**S**tatistical determinatio**n** of **o**ptimal **(w) pa**rameter **c**onfigurations for GPU **k**ernels) precisely takes this approach collecting a rich set of kernel program characteristics as static features, through a pass to an LLVM compiler and from the CUDA compiler. It is also natural that the size of the problem that the kernel has to work on affects the optimal block size. Therefore, SNOWPACK augments the static features with this dynamic feature. The solution has the potential to improve execution times for large-scale executions, where a kernel (which is typically short running) is executed many times. Finally, this solution direction is important from a performance portability perspective whereby SNOWPACK can decide the optimal block size for the kernel when it is moved to run on a different GPU, given some specification of the new GPU architecture.

Results summary

We evaluate SNOWPACK, OCC, and Autotuner for 89 kernels of 10 applications (see Table 2). Our statistical model is *not* very accurate in predicting the running times of a kernel across the range of block sizes—median prediction error is 41.5%. However, it is reasonably accurate in the simpler problem, predicting the block size that will give the minimum running time—it is within 2.56% of what an exhaustive search will achieve, when run with unseen kernels, and within 5.30% when run with unseen input sizes. Further, SNOWPACK outperforms OCC by 35% and 13% for unseen kernels and unseen input sizes. Autotuner shows better performance than

SNOWPACK, outperforming in accuracy nearly 80% of the cases. However, that comes at the cost of a much higher overhead in making each prediction—1.56X and 2.76X for unseen kernels and unseen input sizes. This overhead can be significant, especially for large-scale applications with many kernel calls with slightly different parameters such as input size (which would each require a new run of the Autotuner), where the additional overhead introduced by Autotuner will lead to a significant difference in application running time.

2 SOLUTION OVERVIEW

We first present a high-level overview of our solution, SNOWPACK. In the next section, we present the details of each aspect of our design. SNOWPACK is passed an application written in the CUDA language. It collects the static features of each kernel in the application through an *opt* pass of LLVM plus resource usage of the kernel through the NVIDIA CUDA compiler. Using training data which comprises kernels $\times$ input sizes $\times$ block sizes as input and execution kernel as output, SNOWPACK generates our models, SNOWPACK. At runtime, SNOWPACK just before a kernel is launched, using the static features plus the input size to the kernel, predicts the execution time of the kernel for a host of candidate block sizes. It then selects the block size for which the predicted execution time is minimum and launches the kernel with that block size. To benchmark SNOWPACK, we measure the theoretical best achievable execution time, which is obtainable by doing an exhaustive search through the parameter space, and compare it to the execution time with the block size chosen by SNOWPACK. We also compare how far Autotuner and OCC are from the theoretical best. The evaluation set of applications and kernels are listed in Table 2.

3 METHODOLOGY

3.1 Features

SNOWPACK utilizes prediction models that make predictions based on certain features of the kernel. The features were chosen so as to accurately characterize the kernel and its behavior when executed on a GPU. The list of features is listed in Table 1. In addition to the static features which can be collected at compile time, we also include as a feature, the input size, *i.e.*, the total number of threads that the kernel has been called with. This was done in order to capture the naturally expected dependency that the kernel running time, and hence, the optimal block size, may have on it.

For example, the CloverLeaf kernel `device_advec_mom_yvel_kernel_cuda(...)` has an optimal block size of 320 for input size 1857600, with a running time of 0.36 ms while the optimal block size for input size 59059200 is 128, with a running time of 10.92 ms.

This comprehensive list of static features includes all those which are well known in the programming languages community to be important indicators of program behavior [9, 11]. Added to this are some parameters that are known to affect GPU kernel performance such as memory access patterns (features 34, 35, 37), usage of scarce resources (37, 38), and synchronization overhead (36).

#	Feature description	Type
1	Number of load instructions	Int
2	Number of store instruction	Int
3	Number of branch instructions	Int
4	Number of addition instructions	Int
5	Number of subtraction instructions	Int
6	Number of multiplication instructions	Int
7	Number of division instructions	Int
8	Number of remainder instructions	Int
9	Number of logical instructions	Int
10	Number of functions calls	Int
11	Number of comparison instructions	Int
12	Number of atomic read or write instructions	Int
13	Number of pointer arithmetic instructions	Int
14	Number of stack allocation instructions	Int
15	Number of type casting instructions	Int
16	Number of integer instructions	Int
17	Number of floating point instructions	Int
18	Total number of instructions	Int
19	Number of load instructions at loop depth 1	Int
20	Number of load instructions at loop depth 2	Int
21	Number of load instructions at loop depth ¿ 2	Int
22	Number of store instructions at loop depth 1	Int
23	Number of store instructions at loop depth 2	Int
24	Number of store instructions at loop depth ¿ 2	Int
25	Number of basic blocks	Int
26	Minimum basic block size	Int
27	Maximum basic block size	Int
28	Average basic block size	Int
29	Is there a loop?	Bool
30	Number of loops	Int
31	Does the kernel have nested loops?	Bool
32	Maximum loop nesting level	Int
33	Number of arguments	Int
34	Does the kernle not access memeory?	Bool
35	Is the kernel only reading memory?	Bool
36	Number of `__syncthread()` calls	Int
37	Amount of shared memory used	Int
38	Number of registers used	Int
39	Input size	Int

Table 1: List of features used by SNOWPACK. All the features except the last are statically collected. Of the static features, all except 37-38, are obtained through our pass to the LLVM compiler.

3.2 SNOWPACK

SNOWPACK uses a Support Vector Regression (SVR) model in order to predict the running time of a kernel for different candidate block sizes and then uses these predictions to choose the optimal block size. This struck us as a promising choice because an SVM can handle high dimensional feature space, as ours is, without needing excessive amounts of training data. A soft margin SVM-based regressor was trained on the input data. The three parameters C, γ, and ϵ were tuned via

3-fold cross validation in order to ensure generalizability. The model was run multiple times, 10 times was the default in our experiments. For each run, the input data was split into training and test sets and predictions were made for the test set based on the model trained on the training set. These sets were disjoint with respect to kernel or input size for the two scenarios described in Section 3.3.

SVR Limitations

We experimentally found a limitation of SVR—it fared poorly on kernels that had data points with large input sizes, and more acutely, when there was a large range of input sizes. This included 5 kernels in CloverLeaf and one each in B^+Tree and Hotspot. The performance degradation with respect to the theoretically best achievable, when all the kernels are included (including the above problematic ones) is 3X that when the above problematic kernels are excluded.

3.3 Training and Testing Method

The models were tested for two scenarios: one where they encountered a new kernel which they had not encountered in their training dataset, and another where they encountered a new input size for a kernel which they had already encountered in the training dataset, but this specific input size was not present in training. We call these two cases, respectively, "unseen kernel" and "unseen input size".

For the unseen kernel case, the training size to the test size was 75:25 based on the number of *kernels*. So if a kernel was put in the training set, *all* the input sizes corresponding to that kernel were put in the training set. The training and test sets were disjoint with respect to kernels, that is, any kernel was present in exactly one of these sets.

For the unseen input size case, the total dataset was once again divided into a training and a test set, but this time, the division was done on the basis of input size. Thus, both the training and the test set contained samples from all kernels, but they were disjoint with respect to input size. Thus, any input size I for a kernel K would go to either the training or the test set, but not to both. For kernels with just a single input size, the corresponding samples were put in the training set. Here too, the split for training and test set was done in a 75:25 ratio based upon the number of input sizes for each kernel.

3.4 Toolchain

In order to collect the static features, two sets of tools were used. Most of the features were collected by compiling the CUDA programs using LLVM's CUDA compiler. The generated LLVM IR was then subjected to an *opt* pass in order to get most of the compile time features such as the number of loads, stores, `__syncthread()` calls, etc. In order to obtain the remaining compile time features, namely, the amount of shared memory and registers used, the programs were compiled with NVIDIA's *nvcc* compiler with the `--resource-usage` flag. The features thus collected for all the kernels was then merged into a single kernel-wise feature vector.

3.5 Practical Simplifications

It was found that across kernels, their running times spanned a large range of values, spanning across time of the order of 10 µs to times in the order of tens of seconds. This typically poses a challenge to statistical learning algorithms and we therefore logarithmically scaled the running times for both training and prediction. Furthermore, some of the kernels were found to be causing large errors in all of the models (including the OCC and Autotuning models). Such kernels were removed from all further experimetation—3 kernels from Lulesh. On investigating these further, we found that the execution time behaved in a discontinuous manner with block size. Some kernels also had extremely large values for some features and these had to be removed because of numerical stability reasons. There were 5 such kernels (1 from Heartwall, 2 from BFS, 1 from Lulesh, and 1 from NN). We think that this stage of kernel filtering can be eliminated with further engineering of the statistical models.

For obtaining the block size predictions from OCC, the amount of shared memory and the number of registers used by the kernel were fed into the OCC spreadsheet[3] and the block sizes corresponding to maximum occupancy were chosen. The runtimes corresponding to each of the these block sizes was then determined from the dataset which we had already collected (for the exhaustive executions). The block size (from amongst those predicted by OCC) which was closest to the median running time was chosen as the OCC prediction.

Similarly, for Autotuning, we used the Nelder-Mead simplex optimization algorithm to optimize the running time for a given kernel and input size. Here too, the search space was the dataset that we had already collected, restricted to that particular kernel and the first input size that is encountered for that kernel.

4 EXPERIMENTAL SETUP

4.1 Dataset Description

We had 89 kernels across 10 applications in our dataset. The runtime of these kernels spanned $[10^{-3}, 10^{2}]$ seconds, and we chose to map runtimes by $ln(x)$ during the training phase to compress the target values during mean square error calculation. Otherwise, the higher running time kernels were weighted more heavily during training. The logarithmic scale was inverted by $exp(x)$ during the prediction phase. There were 9387 samples of the 89 kernels, each having a different block size and input size.

4.2 Computing Hardware

The kernels were executed and timed on an Intel Core i7-2600 (3.4 GHz, 4-core) with an NVIDIA Tesla K40 GPU (2880 CUDA cores, 12 GB VRAM) and 24 GB of RAM, running Ubuntu 14.04. Autotuning and SNOWPACK were trained and tested on an Intel Core i5-3210M (2.5 GHz, 2-core) with 8 GB of RAM, running OS X 10.11.

[3] https://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

Applciation	# Kernels	# Input Sizes
Cloverleaf	29	7
Lulesh	7	5
B+tree	2	1
BFS	2	3
Gaussian	2	8
Heartwall	1	1
Hotspot	1	1
NN	1	1
Tealeaf-2D	21	1
Tealeaf-3D	20	1

Table 2: List of applications and constituent kernels used in our evaluation. For each application, the number of different input sizes used in our evaluation are also mentioned.

4.3 Evaluation metric

The major metric we use is based on accurately measuring the running time of a kernel. This is done as follows.

(1) An event *start* is triggered at T_s on the primary CUDA event queue whenever the test kernel is called
(2) The kernel is executed with block size b and input size i
(3) Event *end* is triggered at T_e whenever the test kernel returns
(4) The CUDA event thread and CPU threads are synchronized
(5) The CPU records $R(b,i) = T_e - T_s$ in milliseconds, where R is the running time accurate to 50 μs as per the CUDA API specifications[4].

We wish to minimize $R(b,i)$, the running time as a function of block size b and input size i for each kernel. Practical block sizes exist as a multiple of 32 through a maximum of 1024 since CUDA issues warps in sizes of 32 threads. A block size of 51 for example would yield a warp with only 19 of 32 threads active. Therefore we define the *optimal running time* for input size i as

$$\underline{R(i)} = \min_{b \in B} R(b,i) \tag{1}$$

subject to:

$$b \in \{x : 32 \leq x \leq 1024, x \bmod 32 = 0\} \tag{2}$$

where x defines the constraint on block size candidates, and $\underline{R(i)}$ is the shortest possible running time for a given input size i for the particular GPU. We define the set of all possible block sizes b as B, with $|B| = 32$. Therefore, $\underline{R}$ can be found experimentally by executing the kernel for 32 different block sizes for each (kernel, input size) pair. This is the theoretically best execution time possible.

Our major performance metric is called *performance suboptimality* that captures the adverse impact of a bad prediction[5].

$$S = \frac{R_c - \underline{R}}{\underline{R}} \cdot 100\% \tag{3}$$

[4] https://devblogs.nvidia.com/parallelforall/how-implement-performance-metrics-cuda-cc/
[5] This is specific to each input size, but we drop the notation i for clarity of the writing.

where R_c is the runtime with block size given by the algorithm being measured (SNOWPACK, Autotuning, or OCC), and $\underline{R}$ is the theoretically best runtime ($\underline{R} \leq R_c(b) \ \ \forall\, i$). The value S is the percentage increase in runtime compared to the optimal value. The lower the value of S is, the better is the algorithm.

A minor metric is optimal block size matching rate, *i.e.*, whenever the block size b from Eq. (1) matches the block size from the candidate selection algorithm. In some cases, the kernel execution is not sensitive to the block size and this metric unfairly penalizes block mismatches that have limited impact on overall application performance. For example, a prediction scheme with a match rate of 0% may still have $S = 0$ if the running time was constant across all block sizes.

4.4 Description of Applications

The applications which formed a part of our dataset are as follows:

Lulesh is a simplified hydrodynamic simulation which approximates its system by dividing it into an unstructured mesh.

CloverLeaf by UK-MAC is a mini-app that solves the compressible Euler equations on a Cartesian grid, using an explicit, second-order accurate method.

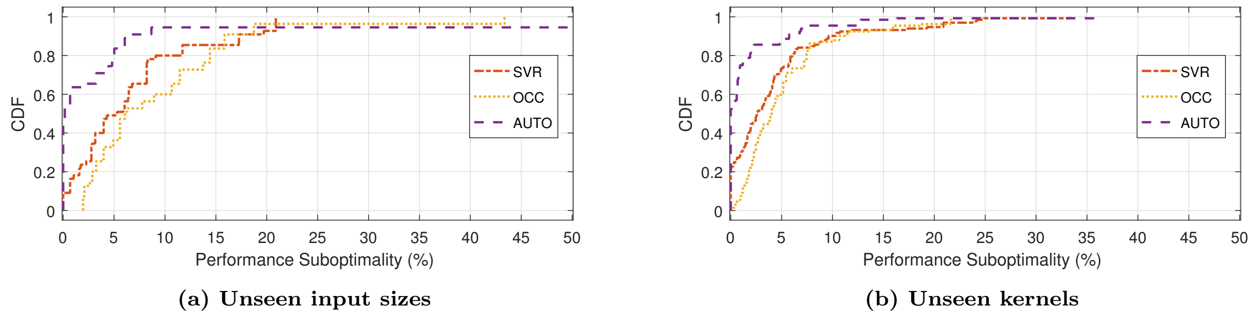
TeaLeaf , also by UK-MAC is a mini app that solve heat conduction equations by dividing it's space into a regular grid. The kernels then each update the relevant variables as they loop through the entire grid. Both the 2d and the 3d versions of this application were tested.

Rodinia is a benchmark suite designed for heterogenous computer systems. It consists of multiple simple applications, a few of which were used by us:

- **B+tree** This application is used to leverage braided parallelism to perform a traversal of a B+tree using CUDA.
- **HotSpot** This is an application used to estimate processor temperature using a simulated floor plan and power measurements.
- **Gaussian Elimination** This represents a technique used to solve linear equations. It performs the computations by rows and then synchronizes before the next iteration.
- **Heart Wall** Heart Wall tracks a mouse heart over a series of ultrasound images.
- **BFS** This application provides a GPU based implementation of breadth first graph traversal.
- **Nearest Neighbor** This application finds out the k-nearest neighbors from an unstructured data set.

5 EVALUATION OF SNOWPACK

We trained SNOWPACK and through 3-fold cross validation, determined the optimal set of parameters to be $C = 500$, $\gamma = 10^{-6}$, and $\epsilon = 10^{-6}$. Each experiment was repeated 10 times, for different splits of training and test sets (always 3:1 ratio). For each of the two scenarios (unseen kernel and unseen input size), we tested SNOWPACK along with OCC



(a) Unseen input sizes

(b) Unseen kernels

Figure 2: The cumulative distribution function (CDF) for both unseen kernels and input sizes, for our techniques (SVR), compared with existing approaches (OCC, Autotuning). Our SVR approach is comparable in performance to OCC and is inferior to Autotuning but does not require runtime searching.

and Autotuning, the latter using the Nelder-Mead simplex optimization algorithm. The runtime given by the block size predicted by these techniques was then compared to the theoretically best possible runtime which we had obtained through exhaustive runs of the kernel for different block sizes, as described in Section 4.3.

For the OCC case, multiple block size values are predicted. We evaluated the running times for each of these block sizes and chose that block size whose running time was the median. We think that this represents a fair comparison since without prior knowledge about which block size has the best running time, it is not possible for the developer to make an informed decision; thus, median would represent a good approximation of the expected performance if the developer chose at random one of the block sizes indicated by OCC.

For the Autotuning case, the Autotuner needs to actually execute the kernel multiple times with different block sizes in order to determine the optimal value. This might be helpful in determining the optimal block size during compilation and then using that value for a particular kernel, but is not as helpful if the optimum block size needs to be determined at runtime, since the Autotuner will execute the same kernel multiple times with different block sizes, but the requisite output is already available after the first run of the kernel. So, while the Autotuner might be able to predict a good block size (and empirically it does so more often than SNOWPACK), the fact that it needs to run the kernel multiple times would create a high enough overhead as to make it unusable in many practical settings.

5.1 New Kernel Prediction

We ran the experiment to determine the ability of the model to make predictions for kernels that it had not been trained upon, and thus, which it had not seen yet. The errors are shown in Table 3. It can be seen that SNOWPACK beats the OCC prediction in both the mean and the median performance.

In terms of just number of samples, SNOWPACK's predicted block sizes either matched or exceeded the performance of the OCC prediction in 60.6% of the cases while it performed better than or equal to Autotuning in 34.1% of the cases. If

	Mean	Median	Mean Prediction Time
SNOWPACK	4.37%	2.56%	18.1 ms
OCC	5.24%	3.97%	–
Autotuning	1.72%	0%	28.4 ms

Table 3: Suboptimality Comparison (validated across new kernels)

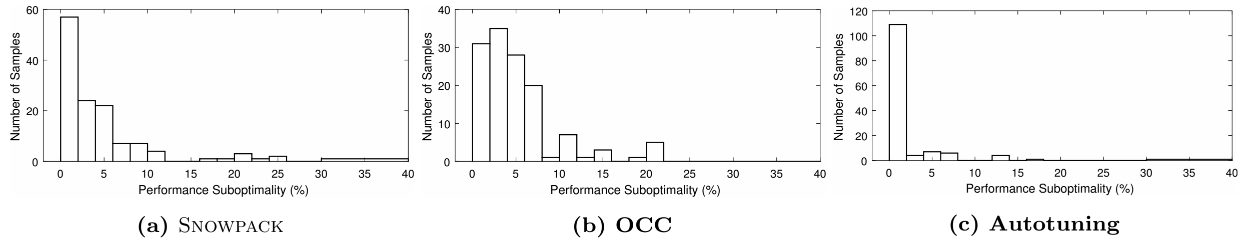
compared to the case where a programmer unluckily randomly picked the worst block from amongst OCC's suggestions, SNOWPACK matched or exceeded the performance in 76.5% of the cases.

While SNOWPACK does have worse performance than Autotuning, it does have a significantly lower prediction time , with a mean value of 28.4 ms as opposed to 18.1 ms for SNOWPACK, primarily due to the fact that SNOWPACK does *not* need to run the model multiple times in order to obtain the prediction. Furthermore, our prediction time does not depend upon the running time of the kernels, which is not the case for Autotuning, for which the prediction time would increase with an increase in kernel execution time. The mean difference between the observed runtimes given by the predictions of SNOWPACK and Autotuning was small—2.67%, with a median value of 1.87%. A majority of the kernels had very little difference, with differences close to zero, as can be seen in Figure 4.

5.2 New Input Size Prediction

For unseen input size prediction, we ran the experiment with new input sizes to determine how well the model scales with respect to new input sizes for kernels which it has already seen, in training. Here too, SNOWPACK had a better performance than OCC in both the mean and the median case, and performance comparable to Autotuning in the Mean case, as can be seen in Table 4

In terms of just number of samples, SNOWPACK's predicted block sizes either matched or exceeded the performance of the OCC prediction in 56.3% of the cases while it performed better than or equal to Autotuner in 21.8% of the cases. Furthermore, when compared to the worst OCC prediction, SNOWPACK matched or exceeded the performance in 58.2% of the cases.



(a) SNOWPACK (b) OCC (c) Autotuning

Figure 3: Histogram for suboptimality of block size prediction, validated across new kernels.

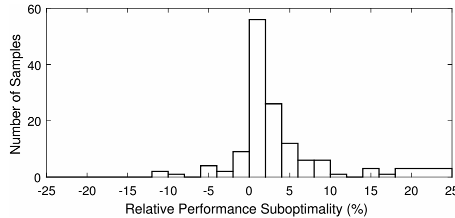


Figure 4: Histogram for relative difference in observed runtimes predicted by SVR and Autotuning, validated across new kernels.

	Mean	Median	Mean Prediction Time
SNOWPACK	6.67%	5.30%	6.91 ms
OCC	9.37%	6.06%	–
Autotuning	6.63%	0.16%	19.1 ms

Table 4: Suboptimality Comparison (validated across new input sizes)

Here too, SNOWPACK had a prediction faster (mean prediction time 6.91 ms) as compared to Autotuning (mean prediction time 19.1 ms) while having a comparable level of mean suboptimality. Also, when compared to Autotuning, SNOWPACK had a mean difference in runtime of only 2.38%, with a lot of the samples giving very little difference, as can be seen in Figure 6.

6 DISCUSSION

Here we discuss possible extensions of our approach to handle conceivable use cases.

Generalizability of the approach. We targeted a concurrency parameter, block size, that is applicable to CUDA kernels. Our approach should apply out-of-the-box to this parameter for CUDA kernels running on other GPUs since there is no GPU architecture specific feature that we use. It is however needed to train the model on the same GPU type as the one for prediction. Our approach conceptually also applies to concurrency parameters in other programming models, such as workgroup size in OpenCL since that has the same two balancing factors as for block size.

Use of dynamic features. When we do root cause analysis of kernels where SNOWPACK performs poorly ("mispredicted-kernel"), we find that these often map to where kernels very similar to the mispredicted-kernel are present in the training set. However, when compared with respect to dynamic behavior, these training kernels are very different from the mispredicted-kernel, such as, statically the number of nested levels of loops is the same but the numbers of invocations of different nesting levels are very different. This indicates that dynamic features will be beneficial, however, they have to be used sparingly so that the cost of executing the kernels for collecting these dynamic features does not drown out the benefit of using SNOWPACK.

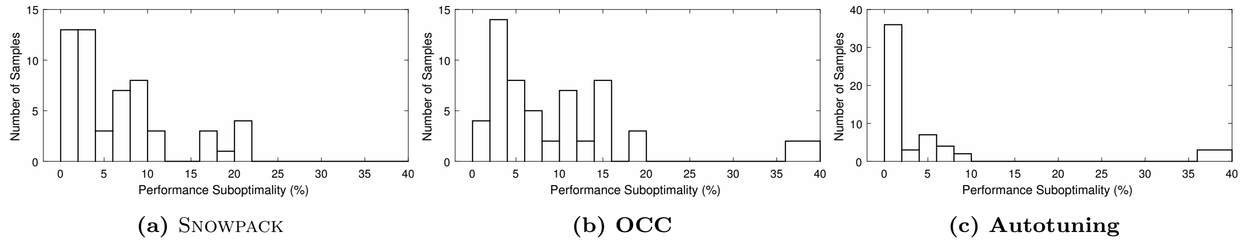
Varying 1D/2D/3D organization of threads. Another dimension for design is varying the organization of threads in a block (1D/2D/3D). However, we found that this choice was tightly coupled to the characteristic of the application, *e.g.*, processing a picture lent itself to a 2D organization, thus minimizing the scope for automatic tuning of this parameter.

7 RELATED WORK

Autotuning. Autotuning has been used before to select the optimal block size for accelerator kernels, including GPU kernels. An autotuning technique for sparse matrix-vector multiplication is shown in Guo *et al.* [2]; Nukada and Matsuoka [4] present an autotuning framework that chooses the optimal number of threads for CUDA-based 3-D FFT library automatically; Ryoo *et al.* [7] discuss a variety of tuning strategies, and show how hardware resource usage affect occupancy and ultimately performance. While most of these techniques focus on specific computational kernels (e.g., sparse matrices), our work, in contrast, is agnostic to the kernels that are tuned.

Study of Performance Features. Kerr *et al.* [3] study the features that describe kernels and hardware architectures, including accelerators. These features are used to predict more accurately the relative performance of a kernel on a CPU versus a GPU. In order to collect such features—which can be static and dynamic features—the work presents a methodology to instrument kernels at the parallel thread execution (PTX) assembly level, which is specific to CUDA kernels. In contrast, our approach collects static features at the level of source code and does not require compiling or executing kernels for prediction.

Kernel Group Size Selection. The closest work to ours is Seo *et al.* [8], where the authors propose an OpenCL work-group size selection algorithm. Their algorithm considers both cache utilization and load balancing between CPU (not GPU) cores. The study shows that the accuracy of the method across 31 kernels and four different multicore CPUs is comparable to the best case of the exhaustive search, while the selection time is much smaller. However, their analysis technique does not directly translate from CPUs to GPUs. This is because



(a) SNOWPACK (b) OCC (c) Autotuning

Figure 5: Histogram for suboptimality of block size prediction, validated across new input sizes.

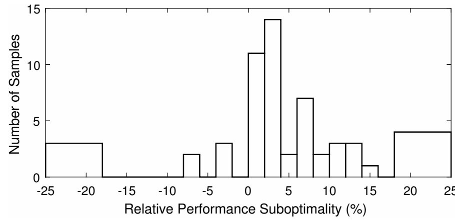


Figure 6: Histogram for relative difference in observed runtimes predicted by SVR and Autotuning, validated across new input sizes.

the resource availability and resource usage patterns are very different across CPU and GPU. For example, the high degree of parallelism and the strong need for simple control flows have significant bearing on the work-group size and there is nothing in the analysis of this prior work that can be parametrized to apply to GPUs.

Machine Learning. Machine learning has been used before to predict the optimal number of threads of a parallel application in dynamic environments [1]. This technique differs from ours in that it learns models from the application on which predictions will be made—ours learns models from different, representative applications, which makes it more generic—and it focus on dynamic features—we focus on static features.

8 CONCLUSION

In this paper, we have shown that a GPU application's execution time depends on setting suitable thread block sizes for each of its constituent kernels. This problem is challenging because of the opposing pulls of resource contention and core occupancy which determine the optimal block size to use for any given kernel. We come up with a solution, called SNOWPACK, that predicts the optimal block size given a rich set of static features and the input problem size. Our solution outperforms, in terms of closeness to optimal execution time of a kernel, NVIDIA's occupancy calculator called OCC while it underperforms the Autotuning approach. However, Autotuning incurs a significant cost at runtime while SNOWPACK does not. We identify root causes where our statistical models fail, which can serve to spur further work on this problem, such as in identifying better statistical models or more discriminating features.

ACKNOWLEDGMENTS

This work was partially performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DEAC52-07NA27344 (LLNL-CONF-678315).

REFERENCES

[1] Murali Krishna Emani and Michael O'Boyle. 2015. Celebrating Diversity: A Mixture of Experts Approach for Runtime Mapping in Dynamic Environments. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 499–508. https://doi.org/10.1145/2737924.2737999

[2] Ping Guo and Liqiang Wang. 2010. Auto-tuning cuda parameters for sparse matrix-vector multiplication on gpus. In *2010 International Conference on Computational and Information Sciences (ICCIS),*. IEEE, 1154–1157.

[3] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. 2010. Modeling GPU-CPU workloads and systems. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, 31–42.

[4] Akira Nukada and Satoshi Matsuoka. 2009. Auto-tuning 3-D FFT library for CUDA GPUs. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 30.

[5] NVIDIA. 2011. Calculating the optimal grid and block size? https://devtalk.nvidia.com/default/topic/501614/calculating-the-optimal-grid-and-block-size-/. (August 2011).

[6] Stack Overflow. 2016. How do I choose grid and block dimensions for CUDA kernels? http://stackoverflow.com/questions/9985912/how-do-i-choose-grid-and-block-dimensions-for-cuda-kernels. (March 2016).

[7] Shane Ryoo, Christopher I Rodrigues, Sara S Baghsorkhi, Sam S Stone, David B Kirk, and Wen-mei W Hwu. 2008. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. ACM, 73–82.

[8] Sangmin Seo, Jun Lee, Gangwon Jo, and Jaejin Lee. 2013. Automatic OpenCL work-group size selection for multicore CPUs. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE Press, 387–398.

[9] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically characterizing large scale program behavior. In *ACM SIGARCH Computer Architecture News*, Vol. 30. ACM, 45–57.

[10] Martin Tillmann, Thomas Karcher, Carsten Dachsbacher, and Walter F Tichy. 2013. Application-independent Autotuning for GPUs.. In *PARCO*. 626–635.

[11] Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael FP O'Boyle. 2009. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. *ACM Sigplan Notices* 44, 6 (2009), 177–187.