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

ScalA'17, Denver, CO, USA, November 13, 2017

Ignacio Laguna

Lawrence Livermore National Laboratory

Ranvijay Singh, Paul Wood, Ravi Gupta, Saurabh Bagchi

PURDUE UNIVERSITY

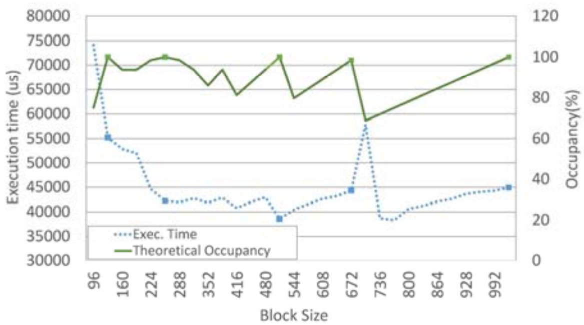
This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DEAC52-07NA27344 (LLNL-CONF-678315). Lawrence Livermore National Security, LLC

Background

- GPUs are increasingly being used for High Performance Computing applications
  - Well suited for highly parallel and scalable workloads
- NVIDIA's CUDA programming model
  - Kernels execute in multiple threads
  - Threads grouped as blocks
  - Programmers need to specify the block size for each kernel call. Threads within the same block can use shared memory and some synchronization primitives among themselves.
  - Total number of threads is grid size × block size. We call this value the *input size* of the kernel.
    - Typically dependent on the input to the program
    - Can change for different invocations
- We propose a model which predicts the optimal block size by considering various static features of a kernel along with a dynamic feature, namely, input size

Current Approaches

- NVIDIA Occupancy Calculator:



- Autotuning
  - Involves compiling and executing the kernel for different block sizes
  - Search through the space of possible block sizes
  - Time involved can be significant
    - Involves executing the program multiple times
  - Not feasible to run in a live environment
    - Performs multiple executions to determine the best runtime
    - Required result is already available at the end of the first execution

Snowpack: Statistical determination of optimal parameter configurations for GPU kernels

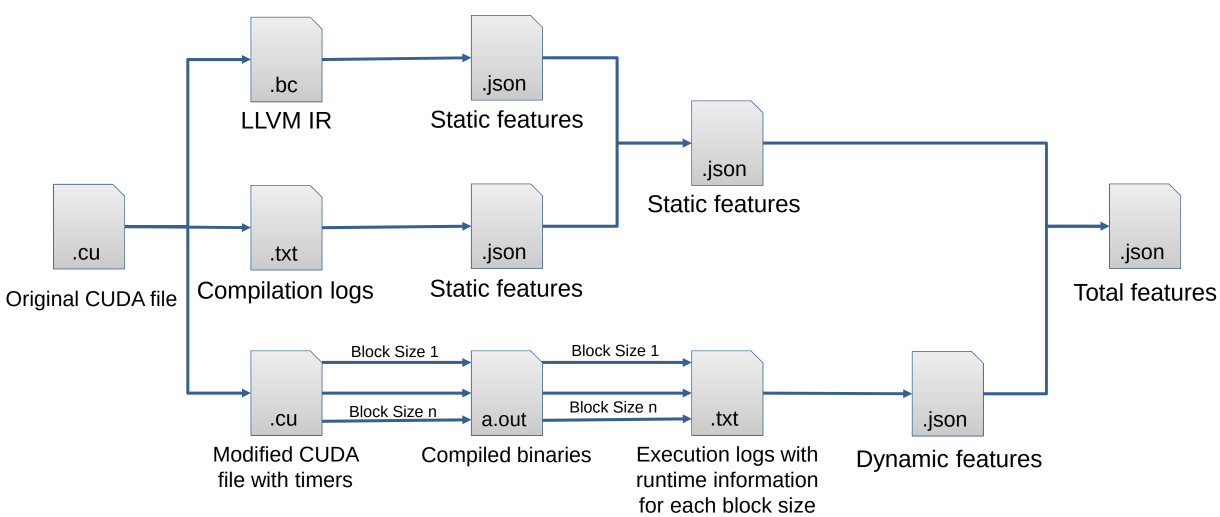
- We use an Support Vector Regression (SVR)-based model to predict the runtime of a kernel given some static features of the kernel, block size, and input size
  - We then use this model to predict the runtime of the different block sizes
  - The block size with the minimum predicted runtime is the predicted block size
- We use an SVR due to the fairly large number of features involved in the prediction, which leads to a very high dimensional feature space

Features

- We use features we expect them to have a significant impact on the runtime such as:
  - Number of various arithmetic instructions and memory operations since
  - Number and depth of loops since we expect programs to spend a large amount of time there
- Besides the static features, we also collected the input size for each block size along with its runtime in order to facilitate input size based prediction

#	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

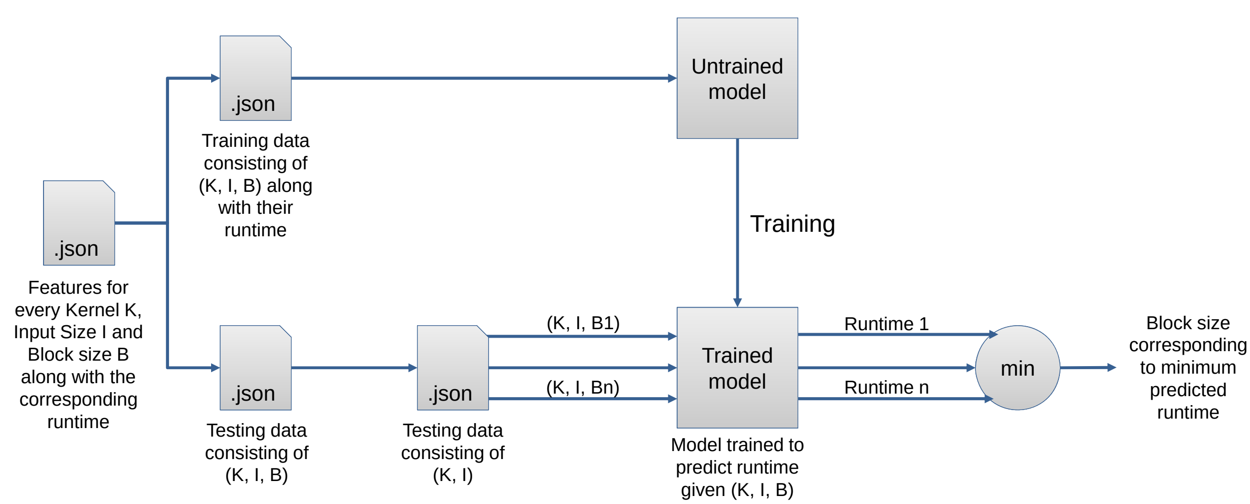
Data Collection Workflow



Toolchain

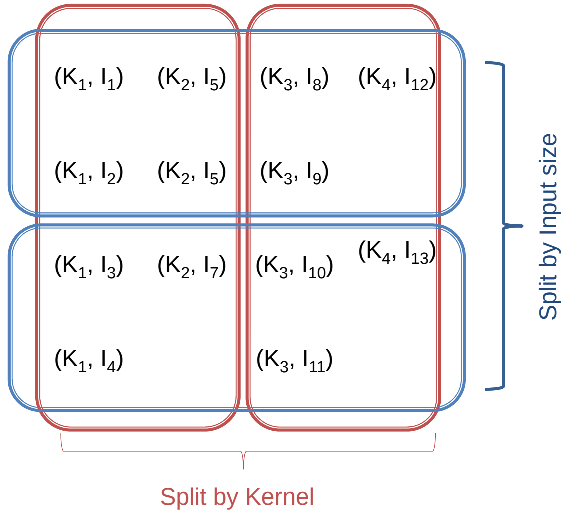
- LLVM
  - clang
    - Compile CUDA to IR
  - opt
    - Obtain some static features
- NVCC
  - Compiler logs
    - Remaining static features
  - Compilation of binary
    - Based on modified file with timers
- We also instrument the code to add CUDA timers immediately before and after the kernel calls in order to obtain the runtime of the kernels and to print it to stdout along with the kernel name and block size
  - Compile and run the modified file for different block sizes and obtain the dynamic features
  - Combine the dynamic features with the static features and used for prediction

Experiment Workflow



Train/Test Split methodology

- We split our data in two ways, by kernel and by input size
  - 75:25 ratio between train/test
- Split by kernel
  - Training and the test set are mutually exclusive with respect to kernels, ie, any given kernel belongs to exactly on of the 2 sets
  - Done to show the performance of Snowpack when it encounters a new, unseen kernel
- Split by input size,
  - Training and test set both contained the same kernels, but different inputs sizes for these kernels
  - Done to test the performance of the model for a kernel which it has already seen, but with a different input size



Evaluation Metric

- We use *performance suboptimality* as our evaluation metric. It is defined as:

$$S = \frac{R_c - R}{R}$$

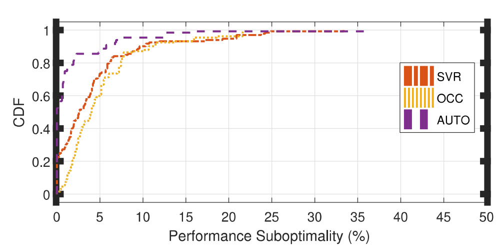
  - R_c is the actual runtime observed with our predicted block size
  - R is the best possible runtime for the given Kernel and Input Size combination (found through an exhaustive search, for evaluation)
- We use this since some kernels did not have much variation in runtime with block size
- Hence, a mis-predicted block size would not have a significant impact on the performance degradation on such a kernel. The sub-optimality metric we define above captures the degradation in performance without penalizing situations where the effect is insignificant.

Evaluation Methodology

- Predict optimal block size on testing data
- Baseline comparisons: NVIDIA Occupancy Calculator (OCC) and an Autotuner based on the Nelder–Mead method
  - For OCC: If multiple block sizes give highest occupancy, we observed the runtime for all the block sizes and then classified the predictions as best, median and worst
- We used the median case OCC prediction for our comparison since in the absence of additional knowledge
  - It would be unreasonable to assume that a programmer always chooses the best or the worst block size from amongst the ones predicted by OCC

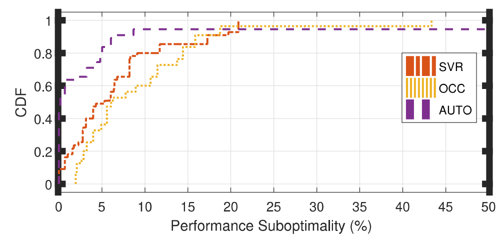
New Kernel Prediction

- The graph on the right shows the fraction of samples below a any given suboptimality.
  - higher is better
- Our model performs better than the OCC case though the performance is worse than Autotuning
- Better performance of Autotuning comes at the expense of a higher prediction time
  - 18.1 ms for Snowpack vs 28.4 ms for Autotuning



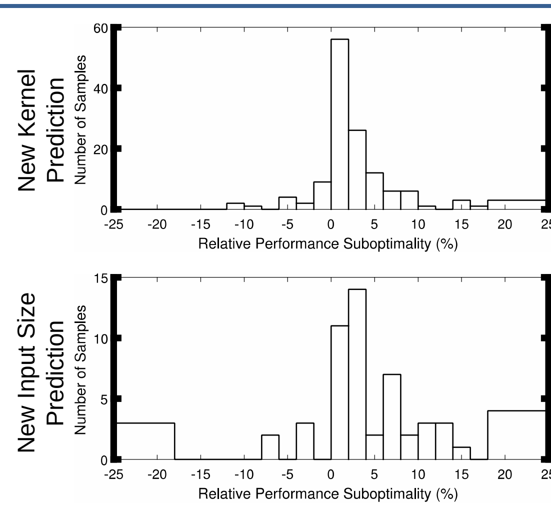
New Input Size Prediction

- For New Input Size Prediction too, a similar ranking is observed
  - Autotuning being the best and our model being better than OCC
- Here too, the prediction time for our model was significantly lower than Autotuning
  - 6.91 ms for Snowpack vs 19.1 ms for Autotuning



Comparison with Autotuning

- Our model performs worse than Autotuning, but has a faster prediction time
- Important for long running kernels
  - Autotuner prediction time depends on the runtime of the kernel while our model's prediction doesn't
- Snowpack uses dynamic inputs
- When compared directly with Autotuning, we see that a large fraction of the prediction differences are close to 0



Conclusion

- Execution time of a kernel is dependent on the block size
  - Determining the optimal block size is, thus, an important problem.
- We propose a model, Snowpack, which predicts a block size with a mean performance penalty of 5.24% when compared to the best possible case for an unseen kernel. For a previously seen Kernel with a different Input, the mean suboptimality is 6.67%
- Better than selecting the median OCC value in our experiment
- Worse than Autotuning in our experiment,
  - But has the advantage that it has a faster prediction
  - And the prediction time doesn't grow with kernel runtime, unlike Autotuning