Modeling the Impact of Thread Configuration on Power and Performance of GPUs

Tiffany Connors
Texas State University
Email: tac115@txstate.edu

Abstract—The use of graphics processing units (GPUs) has become more widespread due to their high computational power. However, GPUs consume large amounts of power. Due to the associated energy costs, improving energy-efficiency has become a growing concern.

By evaluating the impact of thread configuration on performance and power trade-off, energy-efficient solutions can be identified. Using machine learning, the effect of applying a given thread configuration to a program can be predicted in terms of the relative change in performance and power trade-off of a GPU kernel. This enables us to establish which dynamic program features are used to predict the impact of a thread configuration on a program’s performance and how these features are related to the overall effectiveness of an applied configuration. Using these program features, machine learning can be used to assist in determining the most effective thread configuration to be applied based on a given code.

I. INTRODUCTION

There has been an increasing demand for high performance systems for the processing of large sets of data and complex scientific computations. However, performance increase typically results in greater levels of power consumption. The consequence is increased energy costs. Through the modeling of performance and power consumption, it is possible to identify a correlation between the two and determine ways in which to make systems more energy-efficient while continuing to provide high levels of performance speedup.

Because GPUs are a low-cost option for achieving high computational power, they have become widely used in high-performance computing. Although able to provide high levels of performance speedup, GPUs can have a power consumption of up to 300W [1]. Due to the prevalence of GPUs in computational intensive work, there is a need for solutions that will decrease the associated energy costs of GPUs while continuing to provide performance speedup.

II. PROBLEM DEFINITION

The main goal of this research is to improve both performance and power consumption of GPUs through the selection of optimal thread configurations by creating a predictive model using machine learning. This model can then be used to assist in the selection of thread configurations that will produce the greatest improvement in execution time while maintaining a minimal increase in power consumption. In order to achieve this, the impact of various commonly used thread configurations on power consumption and performance values will be observed. Specifically, the compiler optimizations which this work will concentrate on are thread configurations. Additionally, our work will focus on optimizing search algorithms for use on GPUs.

One problem which arises when attempting to optimize code through either the use of compiler optimizations or a change in thread configuration is that while an optimization may produce substantial improvements for one application, the same optimization may be detrimental for another application [2]. For this reason, it is important to create a model that, when provided with the code to be optimized, will predict the impact that an optimization will have on execution time and energy usage.

The framework and overview of the machine learning process proposed in this paper can be seen in Figure 1 and is explained in further detail in the following section.

Fig. 1. Overview of the machine learning process proposed in this paper.
In CUDA, threads are organized into blocks, which are then organized into groupings called grids. Block size refers to the number of threads per thread block. Grid size is the number of block per grid is referred to as the thread configuration. A set of feasible thread configurations, shown below in Table III, is used and the impact of each thread configuration on performance and power is evaluated.

<table>
<thead>
<tr>
<th>Block Size</th>
<th>Threads per Block</th>
<th>Blocks per Grid</th>
</tr>
</thead>
<tbody>
<tr>
<td>64x32</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>64x64</td>
<td>128</td>
<td>16</td>
</tr>
<tr>
<td>64x96</td>
<td>512</td>
<td>8</td>
</tr>
<tr>
<td>64x128</td>
<td>1024</td>
<td>4</td>
</tr>
</tbody>
</table>

**TABLE I**

**SET OF THREAD CONFIGURATIONS**

In this work, we focus specifically on optimizing the performance and power trade-off of search algorithms for use on GPU systems. The programs included in our study as sample code are quadratic assignment problem (QAP) solvers implemented with tabu, simulated annealing, and three variations of 2opt. In addition to the varying algorithms, multiple input datasets from QAPLIB were used. The datasets which were selected are lipa20, lipa30, and tai25a. Each dataset varies in size and structure, and thus affects the program’s behavior.

### A. Feature Extraction

In order to create a machine learning model that works with more than just one type of code, key features of the code which provide a good description of the program’s characteristics must be determined. To do this, the source code is analyzed at runtime and a set of dynamic features is extracted. Each thread configuration from our set of commonly used configurations was applied to all of our sample source code.

The modified code was executed and the programs runtime behavior was recorded using NVIDIA’s command-line GPU profiler, nvprof. It has been shown that by taking code features into account when selecting compiler optimizations, significant improvement in performance can be achieved [4]. The fifty-two features which were extracted are shown in Table II. To normalize the values collected, each feature was divided by the number of instructions executed.

### B. Collection of Execution Time and Power Consumption

In addition to dynamic features, the programs performance and energy usage were also collected. These values were obtained by using the built-in power sensor of the Tesla K20c GPU. During execution of each modified program, the average power consumption and the program’s total execution time were recorded. Next, the performance and power values for each thread configuration were compared to all other thread configurations of the same program.

The increase in execution time and power consumption was calculated by dividing the original thread configuration performance and power values to those of the new target configurations. The performance and power trade-off was then computed by taking the ratio of the performance and power increases, as shown in the following equation:

\[
\text{trade-off} = \frac{\Delta \text{execution time}}{\Delta \text{power consumption}}
\]

Based off this trade-off value, the row of data was assigned a class of either good or bad. These two classes describe the relative change in performance and power of the program. If a thread configuration produced a trade-off of 1.05 or greater in a program, it was classified as ”good”. Otherwise, the row of data was assigned to the ”bad” class. The result is a file for each target thread configuration in which each row in the dataset contains a set of features, the corresponding original thread configuration, and a class label.

Next, all numeric values of the dataset were standardized by using the following formula to calculate the z-score:

\[
z = \frac{X - \mu}{\sigma}
\]

### C. Feature Selection

To reduce our set of features down to only those with the highest predictive power, feature selection was performed. First, any features which had zero variance were removed. Machine learning models were then built and the variable importance of each model was calculated, Figure 6. Next, features were analyzed using correlation matrices, see in

<table>
<thead>
<tr>
<th>Feature Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_latest blockbuster</td>
<td>Number of instructions sent to 32 byte global memory</td>
</tr>
<tr>
<td>get_latest blockbuster</td>
<td>Number of instructions sent to 32 byte global store</td>
</tr>
<tr>
<td>write Barrier</td>
<td>Number of writes launched in a SM</td>
</tr>
<tr>
<td>num_threads_launch</td>
<td>Number of threads launched in a SM</td>
</tr>
<tr>
<td>num_threads_launch</td>
<td>Number of cycles that issue instructions</td>
</tr>
<tr>
<td>num_instructions_executed</td>
<td>Number of instructions executed</td>
</tr>
<tr>
<td>num_threads_executed</td>
<td>Number of threads instructions executed</td>
</tr>
<tr>
<td>get_latest blockbuster</td>
<td>Number of executed global load instructions per warp in a SM</td>
</tr>
<tr>
<td>get_latest blockbuster</td>
<td>Number of instructions executed by all threads</td>
</tr>
<tr>
<td>num_threads_launch</td>
<td>Number of threads blocked in issued in a SM</td>
</tr>
<tr>
<td>num_instructions_executed</td>
<td>Number of executed global load instructions</td>
</tr>
<tr>
<td>global_transactions</td>
<td>Number of global store transactions</td>
</tr>
<tr>
<td>global_transactions</td>
<td>Number of global loads in L1 cache</td>
</tr>
<tr>
<td>global_transactions</td>
<td>Number of global stores in L1 cache</td>
</tr>
<tr>
<td>solutions</td>
<td>Number of threads * number of blocks</td>
</tr>
<tr>
<td>num_threads_block</td>
<td>Number of threads per block in a program</td>
</tr>
<tr>
<td>num_instructions_executed</td>
<td>Number of instructions per grid in a program</td>
</tr>
</tbody>
</table>
Figure 2, and those which were highly correlated to one another were identified.

Fig. 2. Correlation matrix depicting the level of correlation between each of the 52 features.

Any features which had a correlation of 95% or higher were removed from the feature set and a new correlation matrix, Figure 3, was produced. Principle component analysis (PCA) was performed on the remaining features in order to evaluate the active variables’ degree of correlation. Additionally, the features were compared with each of the models’ variable importance values to determine if the features identified as significant factors in the models remained in the new set of features.

D. Machine Learning Methods

Nine different machine learning algorithms, listed in Table 6, were used in this work. The algorithms selected were those which supported binary classification and are available in R’s Caret package. The purpose of using varied machine learning methods was to determine if the same features were significant factors across all models, as well as identify which machine learning algorithms worked best with our data. Each target thread configuration was treated independently and separate models were built, trained, and tested for each of these configurations. The algorithm which performed the best was then selected to be used for building the final predictive model.

Table III

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>boosted C5.0</th>
<th>bagged CART</th>
<th>ctree</th>
</tr>
</thead>
<tbody>
<tr>
<td>random forest</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>naive bayes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>svmRadial</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>flexible discriminant analysis</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>neural network</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k-nearest neighbors</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TABLE III

THE NINE MACHINE LEARNING ALGORITHMS USED IN THIS WORK.

IV. RESULTS

For each target thread configuration file, the data was randomly partition and 60% of the data was used for training and the remaining 40% was reserved for testing. Additionally, repeated k-fold cross-validation was used with k=10 and repeats set to 5. This partitioned data was then used in each of the nine models.

The machine learning models performed quite well, with most models having an accuracy in the high eighties to mid-nineties. The boosted C5.0 tree algorithm was selected as the final model for predicting the impact of modifying the thread configuration of a program. Models built using this algorithm achieved the highest levels of accuracy among
the nine machine learning algorithms used, with an average accuracy rate of 94.9%.

Through feature selection, the initial set of features was reduced from fifty-two down to the fourteen features identified in Figure 3. When principle component analysis is performed on the remaining fourteen factors and the variables factor map is generated, Figure 5, we can see that OrigThreads, OrigBlocks, Solutions, and fb_subp0_read_sectors are highly correlated to each other. not_predicted_off_thread_inst_executed is not strongly correlated to anything else and has negative correlation to inst_issued2. In addition to having a correlation of less than 95% to one another, these fourteen features had also been identified as significant factors in all nine of the machine learning models, as seen in Figure 6. These results were the same across all target thread configuration files and models.

Fig. 5. In the variable's graph, the angle between two arrows depicts the correlation of the two variables. If the arrows are at a 90 degree angle, there is no correlation. Two arrows that are on a near linear line are negatively correlated.

As seen in Figure 4, limiting the feature set to only those features which were not highly correlated improved the accuracy of the models. The feature most frequently identified as the factor with the greatest significance was original threads. It appears that if the original thread count is large, then reducing the thread count to a lower number will be more beneficial.

The two thread configurations which had the greatest number of good instances were 64x128 and 128x16. Changing the thread configuration to 64x128 was good 57.5% of the time, while changing the thread configuration to 128x16 resulted in desirable tradeoff 89.58% of the time. The visualization of these two trees can be seen in Figure 7. Changing the thread configuration to 1024x8 was always bad, therefore we were unable to use C5.0 for this thread configuration since there was no variance in the response label.

The C5.0 tree model predicted that changing the thread configuration to 64x128 would result in improved performance/power trade-off when the program had a high original thread count in conjunction with a large number of thread instructions executed. If a lower amount of thread instructions are executed, then improvement will be seen if this is coupled with a smaller number of write requests sent to sub-partition 0.

When the thread configuration is modified to 128x16, the C5.0 tree predicted that the change would have the highest probability of being bad if the original thread count is small, the number of instructions sent to 32-bit global memory is low, and a smaller number of read requests are sent to sub-partition 0. Otherwise, the change will likely result in improved performance/power trade-off.

V. RELATED WORK

Ukidave et al. studied the effects of optimizations and algorithm design on power/performance trade-offs of GPUs,
APUs, and SoCs. The compiler optimizations investigated were loop unrolling, data transformation, and local memory optimizations [6]. In contrast, this research concentrates on thread configuration. Additionally, the paper by Ukidave et al. did not attempt to create a machine learning model for the implementation of these optimizations.

Agakov et al. used iterative search to select good compiler optimizations for increasing performance. Program features were taken into account and Agakov et al. identified thirty-six loop-level features that described a program’s characteristics [7]. In the work presented in this paper, we did not limit ourselves to loop-level features. Another difference is that our work uses CUDA programs, while their work focused on C programs running on CPU systems.

There has been much work exploring the use of machine learning for selecting compiler optimizations. Research performed by Magni et al. aimed at building a machine learning model for automatic optimization for GPUs by using thread-coarsening [8]. Similarly, a technique has been proposed by Cavazos et al. which uses machine learning to automatically select the best optimizations to increase GPU performance [9]. Liang et al. has proposed a joint register and thread structure optimization framework that achieves considerable increase in speedup, showing that the impact of thread structure and register allocation on performance are related [10]. However, unlike the research proposed in this paper, the work mentioned above did not take power consumption into account.

VI. Future Work

This work currently includes only two classes used in the machine learning models. It is our goal to expand these models to include up to eight different classes in order to give more detailed results of the relative change in performance/power trade-off.

We intend to expand this work by investigating performance and power consumption of stencil code on the GPU. By using a stencil code generator, a large number of programs can be produced in a short amount of time. Dynamic features, execution times, and power consumption will be collected using the same methods already outlined in this paper.

Including stencil code will enable us to create a larger training dataset. This will also allow us to see if the same dynamic features important for predicting the impact of a thread configuration on QAP algorithms are significant factors for stencil code.

In addition to using machine learning to predict the relative change in performance and power trade-off, we also intend to create a model that will predict which thread configuration should be applied in order to obtain optimal trade-off. Provided with a set of program features, the model will output the thread configuration predicted to result in the best performance and power trade-off.

VII. Conclusion

Due to their high computational power, GPUs have become an increasingly popular choice for high performance computing. While GPUs provide exceptional performance speedup, they also consume a large amount of power, resulting in higher energy costs. In order to make these systems more energy-efficient, this paper proposes using machine learning to select the thread configuration which will provide greater speedup while maintaining minimal power consumption.

Using dynamic feature extraction on a given code and selecting the features which are most closely related to thread configuration, a machine learning model can accurately predict the impact that a new thread configuration will have on the performance/power trade-off of the program. In turn, this can be used to assist programmers in the proper selection of thread configurations which will give increased performance while maintaining minimal increase in power consumption.

VIII. Acknowledgment

This material is based upon work supported by the National Science Foundation Research Experiences for Undergraduates program under Grant No. 1359275.

References


