Predicting the Performance of a kernel in terms of Speedup

Rishit Dholakia [*1] **Sarvesh Relekar** [*1]
[1] Courant Institute of Mathematical Sciences, New York University

Abstract

Graphics Processor Units (GPUs) have become the new norm for the execution of large scale parallel applications. This is particularly true for areas of Computer Science such as Machine Learning and Computer Graphics. To quantify the impact of the usage of GPUs in the performance of a particular application, there is a dire need to predict the performance of the GPU in terms of the speed up it provides for that specific application. This project introduces one such methodology to predict the speedup of a kernel with respect to execution performance for a single thread. By utilizing statistical machine learning and in-house simulated data of some CUDA centric code over specific GPU related tasks we are able to provide an insight into some techniques to obtain these predictions.

1. Introduction

The usage of GPUs has seen an increasing trend in the number of its applications in various tasks, especially in the past few decades. Due to an increasing interest in the usage of GPUs for personal as well as industrial applications have made a bigger leap in programming and designing architectures for many tasks. With the evolving use of GPUs it is important to understand and determine whether a GPU is needed to improve the performance of an application or a CPU is sufficient. Normally the trend continues by experimenting with different tasks on GPU to identify which task benefits the most from it. This method requires the user to well versed with GPU programming technologies such CUDA API Libraries for programming languages such as C or C++. To tahe advantage of the parallel processing capabilities of GPUs and intelligently utilize them it is preferred to select metrics that can be predicted using some analytical, statistical or machine learning based model.

One such metric that can be considered is the speedup of a kernel w.r.t a single thread. This can help us to approximate the improvement in performance of an application by using a GPU against any other processor. The parameter important in helping determine speedup is the number of threads. So it would be a good option to predict the speedup of the kernel with respect to one thread.

Previous works on this topic include predicting the execution time of a particular CUDA kernel using different analytical methods. Lately, in the age of Artificial Intelligence, new techniques like Machine Learning and Deep Learning can be used to predict the execution time of kernel with greater kernel which can then be used to calculate the speedup with respect to a kernel that executes on a single thread. In this project we explore the use of Probabilistic Machine Learning and Basic Deep Learning models to predict the speedup of a kernel w.r.t. a single thread.

We have created a dataset by simulating kernels for multiple kernel tasks such as Vector Addition and Matrix Multiplication across a wide range of dimensions, blocks and threads [1]. These simulations were run on New York University's CIMS servers. In addition to this, we have utilized a Statistical Natural Language Processing paradigm known as the Bag of Words (BoW) in order to parse the code in order to obtain useful features from the input itself. The predictions for speedup were made using the Random Forest Algorithm, a popularly used Machine Learning algorithm for tabular text data.

2. Literature Survey

Multiple approaches based on both Machine Learning and non-Machine Learning techniques have been proposed to predict the time taken for a kernel to execute on a GPU. (Braun et al., 2020) describes one such approach in which a machine learning model based on the Random Forest Algorithm is used to predict the execution time of a kernel and the power consumed by the GPU during execution. The proposed model is also portable but needs to be retrained for different GPU architectures. Another approach has been described in (Wang et al., 2020) which proposes a technique called Cross Benchmarking in which a large number of instructions are simulated as parts of various kernels and then uses machine learning algorithms such as Support Vectors and Gradient Boosted Decision Trees to compute the

[1] Code: `https://github.com/SAR2652/NYU-GPU-Final-Project`

execution time of the kernel. (Agrawal et al., 2018) uses unsupervised learning techniques such as Principal Component Analysis on the input for dimensionality reduction before applying supervised machine learning techniques. An approach that is not based on Machine Learning has been described in (Alavani et al., 2018) which calculates the total execution time using not only the overall execution time of kernel but also the individual delays between instruction executions. A profiling-based approach described in (Choi et al., 2020) that makes use of GPU kernel properties of a given parallel application and MPI communication traces to profile the execution on a specific system and predict the performance on other systems.

3. Problem Definition and Challenges

3.1. Problem Definition

The aim of this project is to predict the speed up of a kernel with respect to a single thread using Analytical techniques and Statistical Machine Learning models. The Speedup is calculated as:

$$Speedup = \frac{Timetakenforsinglethreadexecution}{TimetakenforNthreadexecution} \tag{1}$$

3.2. Challenges

A major challenge that we faced was the selection of tasks to perform simulations. We selected two tasks, namely **Vector Addition** and **Matrix Multiplication** in order to run simulations due to their simplicity, ease of execution and varying dimensionality of input (vector addition requires one-dimensional input, whereas matrix multiplication requires two dimensional inputs). We also needed to consider the problem size in terms of the input domensionality and the number of threads to execute in parallel for a particular problem. This is discussed in Section 5 in detail. To obtain features from the input code file was also a difficult task. For this, we used Natural Language Processing techniques. This also involved parsing the code for loop statements which are particularly responsible for an increase in execution time. We also obtained the number of memory allocation, copy and free operations and calculated their total cost. We generated a set of 7040 examples and 62 features, from which selecting features that could be informative towards improving the model predictions was also a major challenge.

4. Methodology

We propose the following idea in this project: Initially, obtain the execution times of different processes and tasks with respect to a single thread and then generate a dataset. We used simple tasks like matrix multiplication and vector addition. These two tasks were used to run our experiments (simulations) over a number of input values for dimension, blocks and threads per block. A large number of distinct combinations of these inputs were executed to obtain their corresponding execution times. The kernel executions were simulated across five distinct GPUs with varying specifications in terms of DRAM, number of CUDA cores, etc..

[2]

In order to parse the kernel code, we use a Bag of Words method to store specific identities such as for loops within the kernel, arithmetic operation symbols such as ’+’, ’-’, etc. and create features from these values as a part of our inputs. Other features were generated using GPU specifications and profiling costs of operations such as memory allocation, memory copying and memory deallocation.

These features together in order to obtain a single data set comprising of all possible combinations of the values of individual features. Then we executed multiple machine learning algorithms on different subsets of these features to see which algorithm gives the. best estimate of kernel speedup. Finally, the trained model is tested on an unseen task (in our case, matrix sum) to verify the estimate of the model.

5. Experimental Setup

5.1. Data Set

For this project , we have generated a custom data set comprising of 7040 unique data points that were simulated for two tasks. For the vector addition task, we considered dimensions ranging from N = 1 to N = 10 million values, whereas for the matrix multiplication task, a lower dimension range from N = 1 to N = 8192 was used.

5.2. Blocks and Threads

The number of blocks used for each experiment ranged from 1 to 16 whereas the number of threads per block ranged from 1 to 1024 threads per block.

5.3. Hardware Resources

The simulations to obtain the time taken for kernel execution were executed on five distinct CIMS sockets, each with a distinct GPU of its own. The GPUs used were as follows:

1. NVIDIA GeForce GTX TITAN Black,
2. NVIDIA GeForce RTX 2080 Ti
3. NVIDIA TITAN V

[2]Student IDs: rnd7446, sr5796

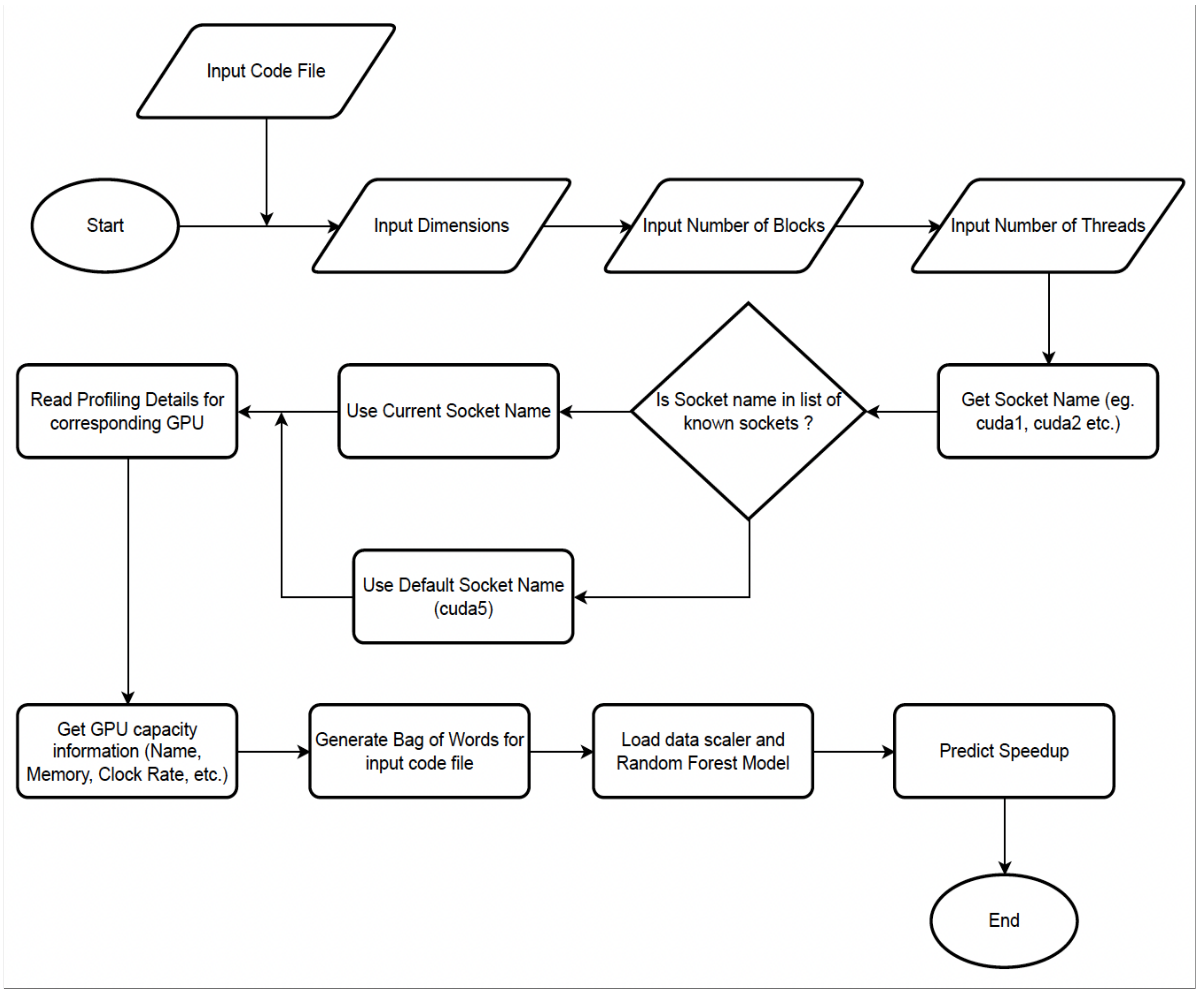
Input Code File
Start
Input Dimensions
Input Number of Blocks
Input Number of Threads
Read Profiling Details for corresponding GPU
Use Current Socket Name
Is Socket name in list of known sockets ?
Get Socket Name (eg. cuda1, cuda2 etc.)
Use Default Socket Name (cuda5)
Get GPU capacity information (Name, Memory, Clock Rate, etc.)
Generate Bag of Words for input code file
Load data scaler and Random Forest Model
Predict Speedup
End

Figure 1. System Flow

4. ”NVIDIA GeForce GTX TITAN X

5. ”NVIDIA GeForce GTX TITAN Z

We also used profiling tools such as *nvprof* to profile information with respect to the dynamic characteristics of the GPU. We also wrote a script to obtain the information of the GPU at run time.

5.4. Software

We used C and C++ Programming Languages to create programs for coding the implementations for parallelized solutions for the matrix multiplication and vector addition tasks since these two languages along with Fortran are the only three languages that have support for CUDA APIs. The Python Programming Language was used to write the code to execute simulations in order to generate our data using the *subprocess* module. For the purpose of data reorganization and cleaning, we used Python’s NumPy and Pandas libraries. Open source implementations of label encoding techniques and data scaling techniques as well as various Machine Learning algorithms such as Linear Regression, Random Forest etc. were available using Python’s Scikit-Learn library.

Dimension	Actual Speedup	Predicted Speedup
10	1.75	0.96
100	1.0	0.97
1000	1.00	1.07

Table 1. Actual vs Predicted Speed up for the Matrix Sum Task for 8 blocks and 512 threads per block.

6. Experiments and Analysis

6.1. Dimensionality

The input dimensions were initially selected as a range from 1 to 1000000 for vector addition. We also experimented

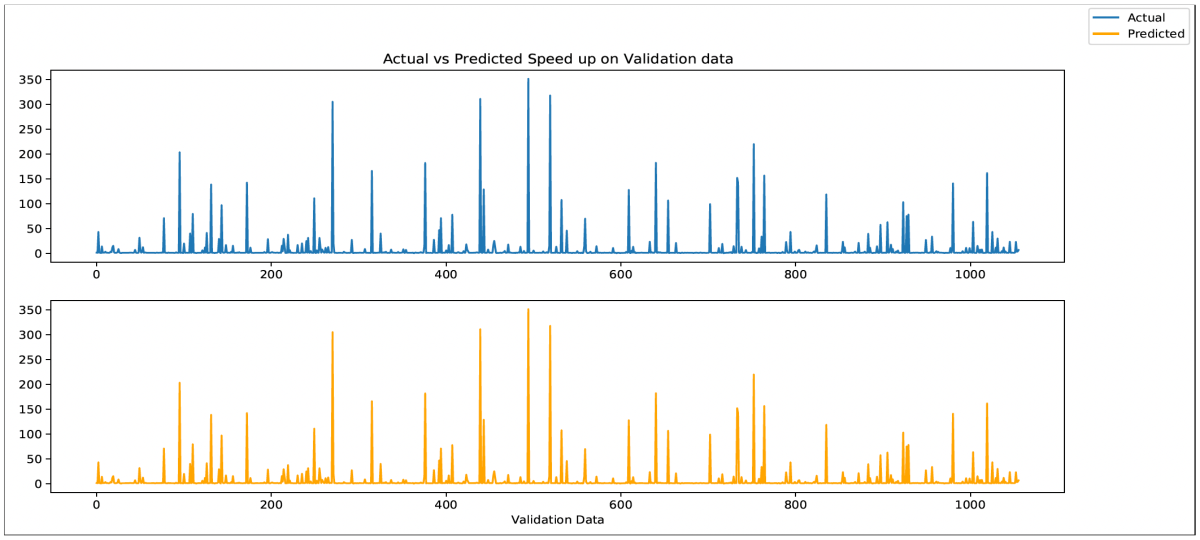

Figure 2. Speedup on Validation Data only for Static Features

Algorithm	Training RMSE	Validation RMSE	Training R^2 score	Validation R^2 score
Linear Regression	35.78	25.89	0.22	0.21
Lasso Regression	38.40	26.9	0.10	0.15
Ridge Regression	35.78	25.78	0.22	0.22
SVR	40.06	28.52	0.02	0.04
Linear SVR	40.03	28.48	0.028	0.05
XGBoost	0.43	21.30	1.00	0.91
Multi Layer Perceptron	31.20	19.95	0.38	0.43
Random Forest	10.05	17.92	0.937	0.41

Table 2. Static Feature based Model Evaluation Metrics for Training and Validation Data

with the number of threads with the values ranging from 1-1024 which resulted in a huge data set of more than 40000 samples alone without much variance in the execution time between the data that would make it viable training. To overcome this issue, we made changes in the the approach of execution by limiting the number of threads to a set of values as 2^N where $N \in \{1, 13\}$. This resulted in observations with better variance in execution time.

The number of dimensions and blocks were also provided in the similar way. In case of matrix multiplication, the dimension size ranged from 256 to 8192 since any value greater than these would cause segmentation faults during execution.

6.2. Code Parsing

To extract meaningful features from code, we needed to parse the input code using parsing techniques. We used the Bag of Words model, a popular probabilistic NLP technique where the frequencies of each word are counted. In this case, we maintained the count of CUDA kernel specific identifiers such as **cudaMalloc, cudaMemcpy (Host to Device),(device to Host),cudaFree**We also maintained the frequencies for datatypes such as **int, float*** etc. that were used as features for our predictive model.

In order to be kernel specific we consider the **for, if ,while** loops ans well as arithmetic operations like **+,-, /,*** of the kernel. We write another parsing coder to just parse the kernel in specific for further information on that the kernel is doing. ***Note: We did not consider the for loops and other arithmetic operations outside the kernel as it did not provide any contribution to the kernel speed up, so it was not considered***.

6.3. Feature Selection

In this subsection, we define two kinds of features namely *static* and *dynamic*. Based on these3 features we showcase two different approaches of feature selection in order to train the model.

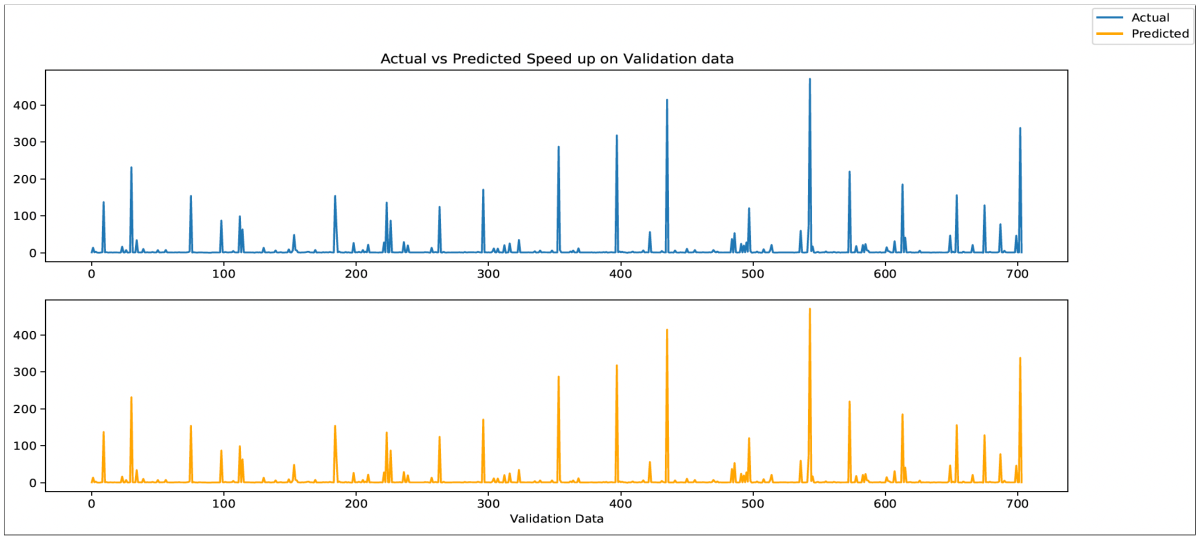

Figure 3. Speedup on Validation Data only for Dynamic Features

Algorithm	Training RMSE	Validation RMSE	Training R^2 score	Validation R^2 score
Linear Regression	34.72	32.02	0.21	0.30
Lasso Regression	37.20	35.96	0.09	0.12
Ridge Regression	34.95	32.54	0.20	0.28
SVR	38.63	37.7	0.027	0.03
Linear SVR	38.65	37.90	0.027	0.0296
XGBoost	0.42	10.47	0.99	0.92
Multi Layer Perceptron	31.67	27.63	0.35	0.49
Random Forest	9.98	8.32	0.93	0.955

Table 3. Dynamic features based Model Evaluation Metrics for Training and Validation Data

6.3.1. Static Approach

A *Static* feature is defined as a feature that is independent of kernel execution. Examples include GPU specification information as the features such as the **Clock Rate, Threads Capacity Per Block, Warp Size** etc. and were used along with the parsing features and the manually accepted features (dimensions, number of threads and blocks). Refer 5 for more information.

6.3.2. Dynamic Approach

This was a challenging approach in which we tried to retrieve the characteristics and parameters related to the operations that occur when a kernel is executes. The reason why these features are called *dynamic* is because their values are not kernel independent. We select only those operations that are not task dependent which involve calculating the cost of cudaMalloc, cudaMemcpy (both from host to device and device to host)and cudaFree. We take the average, minimum and maximum costs. We achieved this by writing scripts that would transfer large number of bytes across the kernel, after which the program would profile the code using nvprof in order to extract the timings for these fixed set of bytes. At inference time these values are used directly without the execution of the test code due to the independent nature of these features as per 5

We use the following equation to calculate the cost using the equation:

$$Cost = \frac{time\_operation}{total\_bytes\_transferred} \tag{2}$$

6.3.3. Hybrid Approach

We use a third approach as well where we combine both static and dynamic features as inputs into our model to see how well the model can approximate the speedup. The results for these are present in 4.

6.4. Model Training

The data set was split into training and validation data sets comprising of 90% or 6336 examples in the training data and 704 examples in the validation data set respectively. The data was normalized between 0 and 1 was trained on

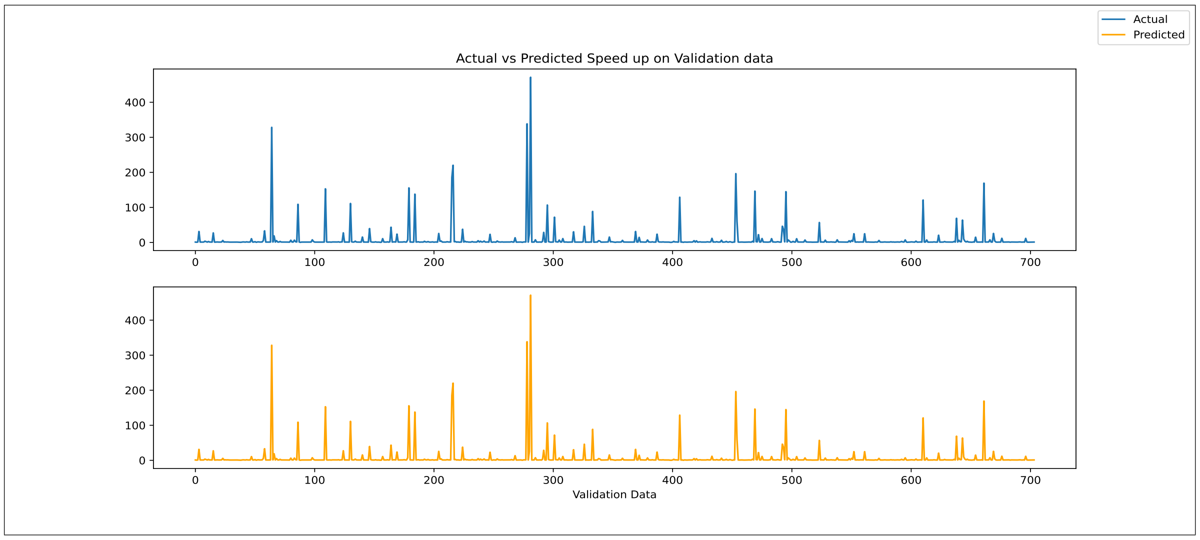

Figure 4. Speedup on Validation Data for both Static and Dynamic Features

Algorithm	Training RMSE	Validation RMSE	Training R^2 score	Validation R^2 score
Linear Regression	35.19	27.15	0.22	0.30
Lasso Regression	37.65	29.85	0.10	0.15
Ridge Regression	36.53	28.94	0.16	0.20
SVR	39.19	31.78	0.03	0.04
Linear SVR	39.45	32.08	0.02	0.02
XGBoost	0.43	9.47	1.00	0.91
Multi Layer Perceptron	30.90	23.69	0.35	0.43
Random Forest	9.88	8.52	0.94	0.93

Table 4. Model Evaluation Metrics for Training and Validation Data based on both Static and Dynamic Features

six different algorithms namely Linear Regression, Linear Regression with L1 Penalty (Lasso), Linear Regression with L2 Penalty (Ridge), Support Vector Machines, Linear Support Vector Machines and the Random Forest Algorithm. These algorithms were trained on static features 2, dynamic features 3 and a combination of both kinds of features. We observed that the Extreme Gradient Boosting and Random Forest Algorithms algorithms gave the best performances in terms of our selected metrics which were Root Mean Squared Error (RMSE) and R^2 error. We chose RMSE as our primary metric of evaluation to evaluate the bias and variance of our models since it gives us a better idea of the model's performance on a variety of inputs. The XGBoost algorithm gives an RMSE of 0.43 and 9.47 on the training and validation data sets respectively whereas the Random Forest Algorithm gives an RMSE of 9.88 and 8.52 on the same data sets.

6.5. Evaluation

For evaluation purposes, we have chosen an unseen tasks of **Matrix Addition** since we do not have any previous information regarding how the speed up will vary with an increasing number of threads. The results of our model for three different input dimension sizes are provided in 1.

6.6. Analysis

We compare the models from inclusion and exclusion of static features, dynamic features and a combination of both along with the parsing features. As it can be seen in tables above we have random forest that performs the best amongst all the other regression based algorithms. The Random Forest Algorithm has the least bias and variance which makes it the ideal choice of algorithm for this prediction task.

We observe from 2 and 2 that Static features give us a higher error rate as compared to using only dynamic features or using a combination of both static and dynamic features. Between dynamic and hybrid approaches, both being give similar results, however, on further model testing on different dimensions of the training tasks, the hybrid feature selection approach outperforms the dynamic feature selection approach.

Static Features
Compute Capability, Total Global Memory Shared Memory per Block, Registers per Block Warp Size, Maximum Threads per Block, Thread Dimension Z, Thread Dimension Y, Thread Dimension X, Grid Size Z, Grid Size Y, Grid Size X, Clock Rate, Total Constant Memory, Multiprocessor Count, Integrated, Asynchronous Engine Count, Memory Bus Width, Memory Clock Rate, L2 Cache Size, Maximum Threads per Multiprocessor, Concurrent Kernels
Dynamic Features
CUDA memcpy HtoDAvg cost, CUDA memcpy HtoDMin cost, CUDA memcpy HtoDMax cost, CUDA memcpy DtoHAvg cost, CUDA memcpy DtoHMin cost, CUDA memcpy DtoHMax cost, cudaMalloc Time cost, cudaMalloc Avg cost, cudaMalloc Min cost, cudaMalloc Max cost, cudaMemcpy Time cost, cudaMemcpy Avg cost, cudaMemcpy Min cost, cudaMemcpy Max cost, cudaFree Time cost, cudaFree Avg cost, cudaFree Min cost, cudaFree Max cost

Table 5. Feature Description

In the scenario where we test our model for Matrix Addition, we observe a good approximation of the speed ups when predicted using the hybrid random forest model which potentially indicates the fact that we can predict the speedup to some extent without executing the kernel code.

6.7. Model Performance Analysis

The Random Forest Algorithm is the best performing algorithm since it generalizes well on the training data and has low bias and variance. The XGBoost performs extremely well on the training data but fails to achieve the same level of performance as Random Forest Algorithm on the validation data set. Linear models also fail to perform well due to the large number of features. Similiar reasons apply for the inadequate performance of the SVM and Multilayer Perceptron algorithms.

7. Conclusion

Our algorithm produces robust results for most input CUDA code files. It manages to predict successfully the expected speedup of a kernel w.r.t one thread with an error of small magnitude. It showcases the application of Machine Learning models to analyse performance of kernels in terms of their execution time and speed up w.r.t a single thread. Thjis can be useful in real life applications in order to gain an estimate about the amount of time required to execute real world parallel applications and to what extent a speed up can be achieved in order to improve their performance. Further enhancements , such as considering an additional feature which is representative of the number of dimensions (1D/2D/3D) may further help in improving the performance of this algorithm. This will lead to having a lower error between the prediction and the actual speedup of the kernel.

8. Acknowledgement

This work was part of the CSCI-GA.3033-025 Special Topics: GPU:Architecture and Programming course at New York University in Spring 2022. We would like to thank Prof. Mohammad Zahran for providing us with the opportunity to work on this project.

References

Agrawal, S., Bansal, A., and Rathor, S. Prediction of sgemm gpu kernel performance using supervised and unsupervised machine learning techniques. In *2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, pp. 1–7, 2018. doi: 10.1109/ICCCNT.2018.8494023.

Alavani, G., Varma, K., and Sarkar, S. Predicting execution time of cuda kernel using static analysis. In *2018 IEEE Intl Conf on Parallel Distributed Processing with Applications, Ubiquitous Computing Communications, Big Data Cloud Computing, Social Computing Networking, Sustainable Computing Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*, pp. 948–955, 2018. doi: 10.1109/BDCloud.2018.00139.

Braun, L., Nikas, S., Song, C., Heuveline, V., and Fröning, H. A simple model for portable and fast prediction of execution time and power consumption of GPU kernels. *CoRR*, abs/2001.07104, 2020. URL https://arxiv.org/abs/2001.07104.

Choi, J., Richards, D. F., Kale, L. V., and Bhatele, A. *End-to-End Performance Modeling of Distributed GPU Applications*. Association for Computing Machinery, New York, NY, USA, 2020. ISBN 9781450379830. URL https://doi.org/10.1145/3392717.3392737.

Wang, Q., Liu, C., and Chu, X. Gpgpu performance estimation for frequency scaling using cross-benchmarking. In *Proceedings of the 13th Annual Workshop on General Purpose Processing Using Graphics Processing Unit*, GPGPU '20, pp. 31–40, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370257. doi: 10.1145/3366428.3380767. URL https://doi.org/10.1145/3366428.3380767.