Project Report

CSCI-GA.3033-025
Graphics Processing Units (GPUs): Architecture and Programming
Prof. Mohamed Zahran
Courant Institute of Mathematical Sciences
New York University

Analytical Model for Kernel Performance Prediction

Name: Karan Kumar G
N10704874
kk5409

Table of Contents

Abstract **3**
Introduction and Motivation **4**
Problem Statement and Contribution **6**
Literature Survey **7**
GPU architecture and Influencing Parameters **9**
GPU Overview 9
Influencing Parameters 11
1. Occupancy 11
2. Coalesced vs Uncoalesced Accesses 15
3. Divergence 16
4. Cache hits and misses 17
The Analytical Model **18**
CUDA Input Kernels **20**
Results **23**
Conclusion and Final Thoughts **32**
Bibliography **33**

Abstract

With the deceleration of Moore's Law and the attenuated progress in hardware technology, the imperative now lies in the development of efficient and parallel software to mitigate the scarcity in hardware capabilities. In recent years, the demand for processing vast amounts of data across diverse domains has attained paramount significance. A considerable focus has been directed towards advancements in Machine Learning and Deep Learning, necessitating intensive data computations. Simultaneously, the scientific domain has experienced a substantial surge in data collection methodologies, precipitating a necessity for intricate data calculations. Consequently, there has been an exponential surge in the utilization and evolution of Graphics Processing Units (GPUs).

While Central Processing Units (CPUs) were hitherto the benchmark for diverse computations, their inherent limitations in handling massive parallel applications have paved the way for General Purpose GPUs (GPGPUs). GPU applications are now pervasive and find application across various domains, yielding unprecedented performance enhancements and significant cost reductions due to GPUs' superior power efficiency per computation compared to CPUs.

Nevertheless, the development of GPU code is a non-trivial undertaking. Software developers, accustomed to learning algorithms and general software development in a sequential manner, must invest time and effort in transitioning to parallel thinking. Consequently, planning and developing massively parallel GPU code entail substantial cost implications. Furthermore, GPUs are applicable only to specific problem types that exhibit notable characteristics conducive to parallelization.

Identifying a problem suitable for GPU utilization raises questions regarding the potential performance improvement compared to traditional CPU execution without parallelism. Thus, predicting the performance of GPU code (analogous to kernels on GPUs) becomes crucial in quantifying potential enhancements resulting from the parallelization of software for GPUs. This report introduces an analytical model designed to predict the execution times of GPU kernels for unknown problem sizes without actual execution. The model considers various metrics involved in launching kernels, including problem size, block size, and grid size configurations, enabling the prediction of expected execution times. It is also shown that the prediction can be extended to different GPUs, considering their own hardware characteristics. Consequently, the model instills a degree of confidence without necessitating an in-depth understanding of kernel intricacies, their functionality, or how they scale with increasing problem sizes. The model automates this process, providing a streamlined approach to performance prediction. For the simplicity of the model, it offers adequate prediction accuracy and provides good scope for future expansion to consider more niche parameters affecting all kinds of kernels and subsequently their performance.

Introduction and Motivation

The art of programming applications for GPUs has never been more important. Every other application now needs to be analyzed and studied before parallelising it for GPUs. This study requires insight into the nature of the application. An application would be a suitable fit for parallelisation if it ticks off a few standard characteristics. These include the following:

1. The application should be computation intensive. The overheads of parallelization shouldn't consume the majority of execution time, it should be the actual computations.
2. There must be a lot of independent computations. GPUs are suitable for massively parallel applications. If there is no scope for parallelization, then it might be prudent not to consider the application for parallelization at all.
3. There must be many similar computations such that parallelization is data driven, that is, many processors execute the same kind of computations, but on different data.
4. The problem size must be large enough to see the true potential of GPUs. Small problem sizes might be better off being executed sequentially on CPUs to counteract the overheads.

Once the application is identified as a right candidate for parallelization, there comes another bigger task of actually writing efficient and effective GPU code for the same. Understanding of GPU hardware and techniques to write efficient and optimized code is necessary. This is not an easy task by any means. To help with this process, being able to know how a particular program behaves for different input sizes without having to thoroughly understand the program itself, or even execute it can be very useful. It can also be very handy if we are able to understand the nature of behavior of programs on not just one GPU, but multiple other types of GPUs with different hardware characteristics.

To understand why prediction of execution times of GPU kernels would be helpful, we can look at the following points:

- **Resource Planning and Optimization**:
  - Resource Allocation: Predicting GPU execution times aids in allocating computational resources effectively. Developers can distribute resources based on the anticipated demands of GPU-accelerated code.
  - Optimization Strategies: Understanding expected execution times enables developers to prioritize optimization efforts on the most critical sections of code, ensuring that time and resources are invested where they will yield the maximum benefit.

- **Cost Savings**:
    - Resource Efficiency: Predictive modeling helps avoid unnecessary resource consumption during trial-and-error testing. This efficiency leads to cost savings by reducing the need for extensive experimentation to achieve optimal performance.
    - Strategic Investments: Developers can strategically invest time and resources in optimizing sections of code that offer the greatest potential for performance improvement, minimizing wasteful expenditures.

- **Time Efficiency**:
    - Accelerated Development: Predicting GPU execution times accelerates the development cycle by providing insights early in the process. Developers can make informed decisions and focus efforts on areas that contribute most significantly to overall performance.

- **Early Performance Assessment**:
    - Informed Decision-Making: Predictive models enable early assessment of the viability of GPU acceleration for specific code segments. This information assists developers in making informed decisions about whether to proceed with GPU implementation or explore alternative solutions.

- **Algorithmic and Architectural Exploration**:
    - Exploratory Analysis: Predictive models facilitate exploratory analysis of different algorithms and architectural choices. Developers can assess the potential impact on performance without committing to full implementations, guiding the selection of the most promising approaches.

- **Scalability Analysis**:
    - Optimal Problem Size: Predictive models assist in determining the optimal problem size for efficient GPU utilization. Developers can analyze how execution times scale with varying input sizes, aiding in the selection of problem sizes that balance performance and resource utilization.

- **Decision Support**:
    - Performance Trade-offs: Predictive models provide valuable decision support by quantifying the expected performance trade-offs between GPU and CPU implementations. Developers can choose the most appropriate solution based on the predicted gains and associated complexities.

- **Documentation and Reporting**:
    - Quantitative Insights: Predictive models contribute quantitative insights into the expected performance characteristics of GPU-accelerated code. This information is valuable for documenting expectations, reporting progress, and communicating effectively with stakeholders throughout the development process.

In essence, predicting GPU execution times without program execution empowers developers with foresight, allowing for strategic decision-making, efficient resource utilization, and effective optimization efforts in GPU-accelerated software development.

Problem Statement and Contribution

In this section, the clear definition of the problem statement, the analytical model and what it takes as input and what it outputs is provided.

Determining the absolute execution time of a kernel is not a straightforward task, as a cursory examination of the kernel does not yield precise insights into its temporal characteristics. While it is evident that larger input sizes generally incur lengthier execution times, deriving an exact, quantifiable metric for execution time proves elusive. It is also known that the underlying GPU affects the total execution time. Recent GPUs tend to get more powerful and therefore are able to execute code faster. However, the precise prediction of execution time remains a formidable challenge.

The primary objective in this context is to undertake an analysis of a specific kernel, particularly a CUDA kernel, and its associated launch parameters which at least includes input size, standard block and grid sizes. The inherent characteristics of the underlying GPU are also considered input parameters. The aim is to furnish an estimation of execution times contingent upon variations in problem size, alterations in kernel launch parameters (specifically, grid size and block size), or differences in the underlying GPU hardware.

To validate the predictive accuracy of the proposed model, a series of experiments is conducted across diverse standard kernels. The resultant predictions are juxtaposed against the actual execution times to discern the efficacy and precision of the predictive model. Moreover, a comprehensive investigation is undertaken to survey the landscape of available models and techniques for GPU performance prediction. A meticulous examination of various GPU parameters and an exploration of their impact on prediction accuracy, coupled with an

assessment of the relative feasibility of their integration into the predictive model is also performed as part of this project.

Literature Survey

There has been previous work in trying to predict GPU performance. Amaris[1] worked on a thesis about predicting execution times of GPU kernels. This is a very simplified model which doesn't take into consideration parameters such as occupancy, or coalesced and uncoalesced memory accesses or branch prediction. It assumes every thread does equal work and begins and finishes execution at the same time. It is inspired by a Bulk Synchronous Parallel (BSP) model which takes into account the number of computations and memory accesses to roughly determine the total execution times. The thesis also implements a Machine learning based model which assumes no prior knowledge about the kernel or the GPU hardware and tries to extract features out of multiple parameters which are recorded with the use of a dynamic profiler. These ML models are able to overcome the shortcomings of the analytical model for a few certain kernels, but the overall accuracy is closely matched with that of the analytical model. For the ML models, standard practices of correlation analysis and hierarchical clustering are performed. The final conclusion of the thesis suggests that analytical models do perform better for applications which uniformly scale based on multiple parameters since it utilizes knowledge about impacting parameters and how they affect the performance.

Another paper which has looked into performance prediction is the paper. In Hong et al.[2], the onus is on determining the so-called Memory Warp Parallelism (MWP) which considers the number of running threads and memory bandwidth. They do a thorough case study looking into different cases and determining the corresponding MWP. Once the MWP is determined, they also determine the number of warps that can be executed while threads are waiting for memory and this metric is called the Computation Warp Parallelism (CWP). Together with these two parameters and other statically fetched parameters from the PTX code, they build a complicated model designed to predict the execution times. A technique called microbenchmarking is used to determine the weightage for every parameter contributing to the model. This model is able to deliver decent accuracy and can be considered as an important contribution to this field.

Another really good paper by Sara et al.[3] provides a modeling tool for GPU architectures. They develop a work flow graph to determine the control flow patterns, loop bounds and data access patterns which help in estimating the divergence, coalescing and bank conflicts. A unique aspect of their work is the power to identify bottlenecks which are negatively impacting performance and this information can be used by compilers for optimization. There has also been work which used the intermediate version of CUDA code, the PTX representation to extract information and

accordingly predict the execution times. One such work is by Alavani [4] where all necessary information for prediction is fetched from the intermediate representation. Their prediction pipeline contains two phases, namely the "Delay Computation Algorithm" which calculates the execution time of a single thread by individually adding the delay introduced by every single instruction in the program. These delays are derived from a pre-calculated external parametric model. Then to determine the total execution time, a so-called simulation of scheduling of threads across different Streaming Multiprocessors (SMs) is done, along with considering kernel launch overheads and microbenchmarking details.

Another impressive thesis work done by Singhania [5] individually addresses influencing parameters such as uncoalesced memory access overheads, block size independence of GPU programs and cache reuse optimization which all have impacts on the final execution time. In the further study on each of these parameters in the coming sections, a deep dive into understanding how they influence the execution times is performed. One of the Machine Learning approaches out there was proposed by Wang et al. [6], where they created their own dataset with synthetic kernels with a good spread of different kinds of instructions (which they refer to as cross-benchmarking) to improve prediction accuracy of their ML models. They use these predictions and extend them to determine the right memory frequency scaling for best performance given power requirements. They use dynamic profilers to extract information out of the kernels and use them as features in their ML models, so this isn't exactly a means to predict execution times without actually executing the program. The main idea behind this paper is their application into determining frequency scaling.

GPU architecture and Influencing Parameters

In this section, we will take a look at the basics of the standard GPU architecture and understand how parallel processing works inside a GPU. With the basics set, a deep dive into particular parameters and how they affect the total execution time ensues.

GPU Overview

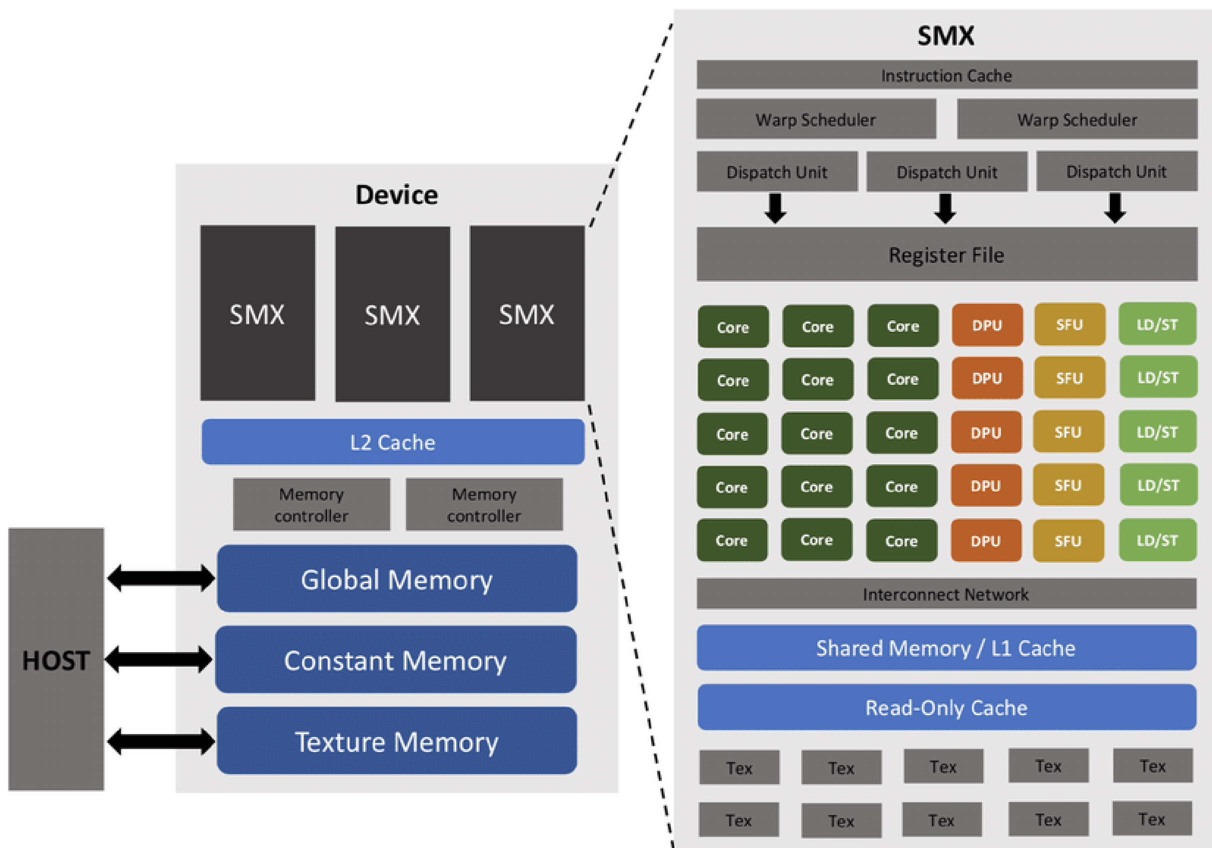

Figure 1: GPU Architecture Overview

Figure 1 shows the construction of a GPU system. Streaming Processors (SMs) are essentially where the computation takes place. Every SM consists of multiple cores/Streaming Processors (SPs), which are responsible for individual thread execution. Recent GPUs contain more than 100 SMs (NVIDIA [7]), with several tens of SPs inside each SM. This very large number of hardware pieces are responsible for the massive parallelism that GPUs bring about. Inside every SM, apart from the SPs themselves, there are other components as well. The most important ones include:

- Registers - the lowest component in the memory hierarchy
- L1 cache/Shared Memory - The cache and the shared memory share a common hardware memory. These occupy one level higher on the memory hierarchy than the registers.
- Warp schedulers - A warp is the elemental thread dispatch unit. These are scheduled onto an SM using these schedulers

There resides a L2 cache outside the SMs, in between the SMs and the other memories. The other memories include the global memory, constant memory and texture memory. Global memory is the main memory (which is also the largest memory on a GPU). Being the largest, it also is the slowest memory on a GPU. Constant memory is smaller but faster than global memory. In Figure 2, from top to bottom, the size of the memory increases, but they are further and further away from the execution units, so they are slower.

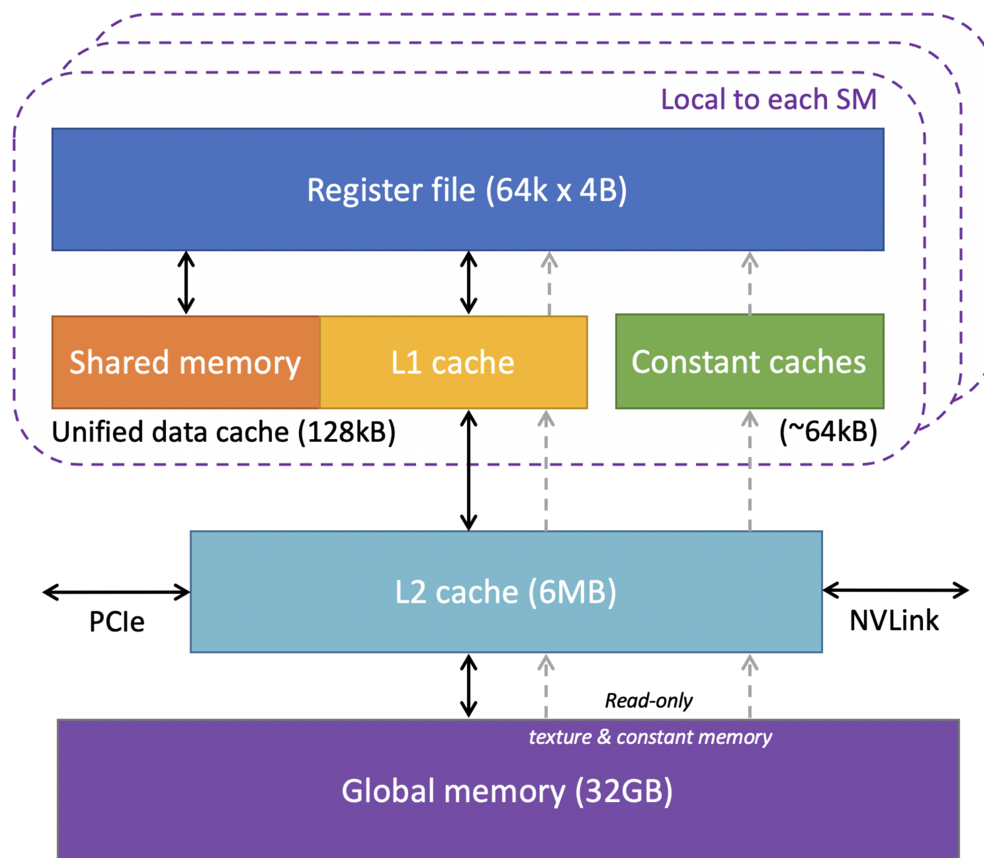

Figure 2: Memory Hierarchy on a GPU System

Relevant to the report here, it is important to know that when an application is parallelised using a CUDA program, it is launched as a kernel. The number of threads that get created by the kernel

is something the user defines. The multiple threads are grouped into blocks, and multiple blocks are grouped into a grid. A single kernel has a single grid. The blocks are scheduled to execute on SMs. To maximize parallelization, we need to ensure we make use of all the available hardware resources. This means depending on our input size for the problem, we need to ensure we have enough threads to occupy every possible SP on every possible SM. Depending on the nature of the kernel or the problem, this might not be possible sometimes, and it is important to identify this to correlate to occupancy which will impact the overall execution time.

Influencing Parameters

1. Occupancy

Occupancy is defined as the ratio of active warps to the maximum number of warps that a GPU can simultaneously execute. This definition is keeping the entire GPU in mind. We can talk about occupancy at a SM level as well, considering individual SPs.
For example:
If there are 64 SPs inside a SM. And at a time, only 48 of those SPs have threads executing on them. In this scenario, the occupancy is simply the ratio of 48:64, which gives us a number of 75%. So in a way, only 75% of the total hardware available inside a SM is being utilized. It is important to understand why there might be less than 100% occupancy. This is due to the following reasons.

Factors that influence occupancy include “resources” available inside each SM and its capacity to have a predetermined number of blocks assigned to it based on these resources. These resources include the number of threads per block, the total number of registers used by each thread, and the amount of shared memory utilized. Optimizing these parameters helps maximize occupancy and, consequently, the overall throughput of the GPU.

Occupancy is a critical consideration when developing GPU applications, as it directly affects the efficiency of parallel execution. Balancing the occupancy involves finding the right combination of thread block size, register usage, and shared memory to fully exploit the parallel processing capabilities of the GPU. In fact this can be accurately calculated for a given GPU knowing its hardware characteristics, and knowing the kernel parameters. Below is an example of the said calculation.

Consider a GPU with a compute capability of 8.6. Compute capability is simply a numeric number denoting the inherent hardware capabilities of the GPU. A higher

number generally denotes a more recent GPU with greater hardware resources and additional features compared to a smaller number. Now with this GPU, let's say we launch a kernel with block size (number of threads per block) as 320. Each thread takes up 10 registers of memory space (Local variables defined inside kernels occupy register space, only the ones which can fit). With respect to the shared memory, an entire block requires 1024 bytes or 1 KB of shared memory. Given these values, the occupancy can be calculated.

Version	8.6
Threads per Warp	32
Warps per Multiprocessor	48
Threads per Multiprocessor	1536
Thread Blocks per Multiprocessor	16
Total # of 32-bit registers per Multiprocessor	65536
Register allocation unit size	256
Register allocation granularity	warp
Max registers per Block	65536
Max registers per thread	255
Shared Memory per Multiprocessor (bytes)	102400
Shared Memory Allocation unit size	128
Warp allocation granularity (for register allocation)	4
Max thread block size	1024

Table 1: Physical limits for the given compute capability of 8.6

Active Threads per Multiprocessor	1280
Active Warps per Multiprocessor	40
Active Thread Blocks per Multiprocessor	4
Occupancy of each Multiprocessor	0.8333333333333334

Table 2: Calculated occupancy data

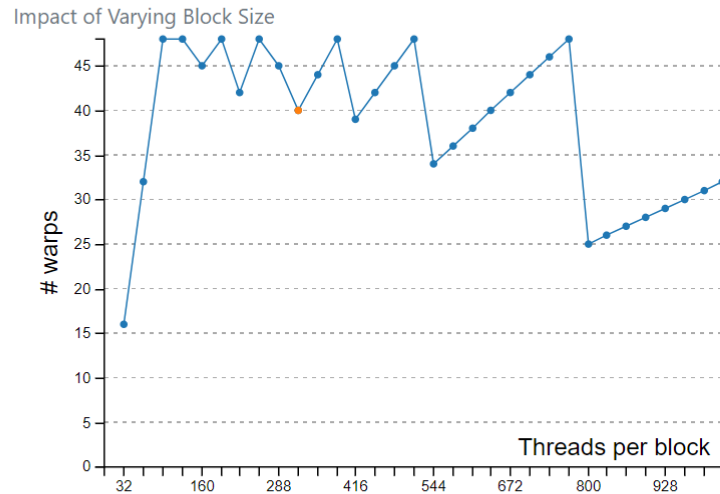

Figure 3: Variation of occupancy with changing block size

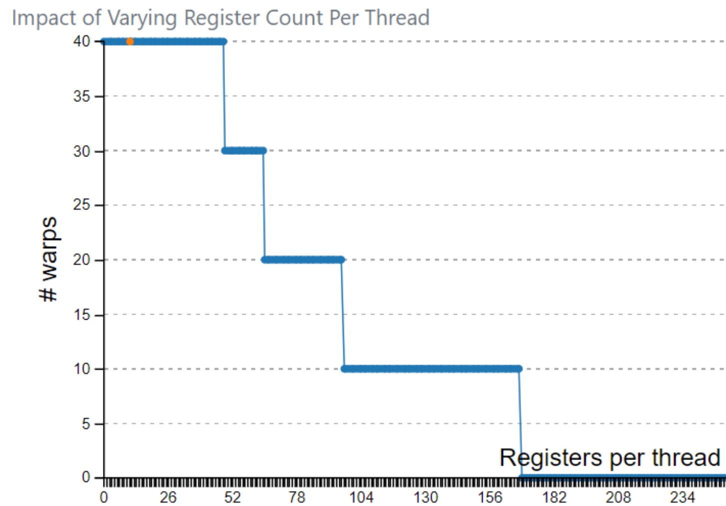

Figure 4: Variation of occupancy with changing requirements of registers per thread

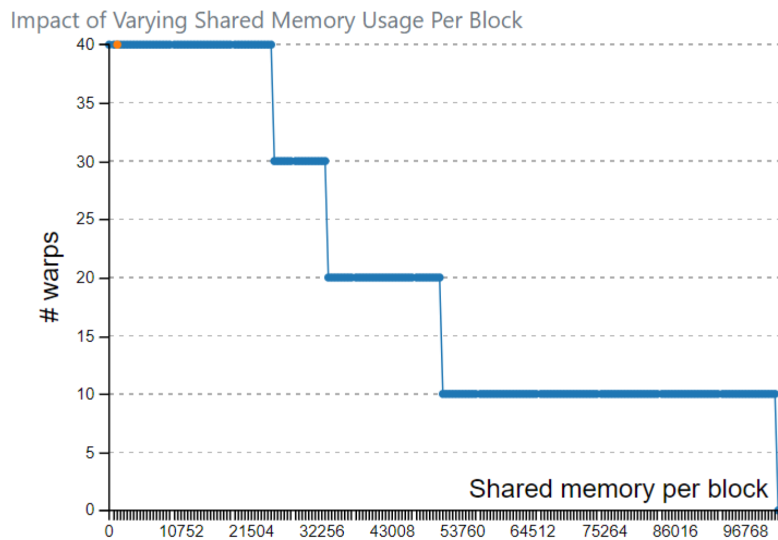

Figure 5: Variation of occupancy with changing requirements of shared memory per block

Table 1 shows the limits defined by the compute capability value. This can be used in accordance with the input values of block size, register size per thread and shared memory requirement per block and calculate the occupancy as is noted in Table 2. A maximum of 1280 threads can be scheduled due to the limited resources in a SM. The important caveat is that a block cannot be divided and scheduled across multiple SMs, the entire block must be scheduled within one SM. Therefore if this is the case, we might have scenarios where one block might take up more than half the space on a SM, leaving no space for another block to be scheduled. Using similar logic, we see here that 1280 threads (or 40 warps) can be scheduled, however 48 warps are technically possible to be scheduled inside a SM. Therefore the occupancy here is 40/48 which roughly equates to 83%.

We can correlate the input parameters to the occupancy in Figure 3, 4 and 5. We see the occupancy generally reduces in steps as we increase the register size requirement per thread and shared memory per block requirement. This is expected as we have only limited resources inside each SM and greater values mean additional blocks cannot be scheduled. Block size also matters as can be seen in Figure 3.

Therefore we see that occupancy values are closely related to all of the above mentioned parameters. It so happens that these parameters are taken as input into our analytical model. Since these affect the occupancy, it directly affects the execution time as well. A higher occupancy generally implies lower execution time, and lower occupancy implies higher execution times. That is, the total execution time is inversely proportional to the occupancy.

There could be a lot more additional parameters affecting the execution times and may overshadow the occupancy effect. However it can be seen if each thread takes a considerable amount of time to execute, occupancy effect can be clearly seen. This can be seen in experiments later on and how they affect the execution times.

2. Coalesced vs Uncoalesced Accesses

Coalesced and uncoalesced memory accesses refer to the patterns in which threads in a GPU access memory. These terms are particularly relevant when discussing global memory accesses CUDA. These can be defined as follows:

Coalesced memory access involves threads within a warp accessing consecutive or contiguous memory locations in a single memory transaction. The advantage here is that it is possible to fetch a lot of data which is used by multiple threads in one go, thereby saving trips to the global memory. Fetching data from the global memory is a costly affair, in fact the costliest memory operation in a GPU since the global memory comes at the bottom of the memory hierarchy.

It is easy to see coalesced memory access in action. If each thread in a warp accesses an element in an array, and these elements are stored consecutively in memory, it results in a coalesced memory access pattern. We will see an example where we access memory both in a coalesced and uncoalesced manner later in one of the test CUDA kernels used for experimentation.

Uncoalesced memory access occurs when threads within a warp access non-contiguous or scattered memory locations in a way that does not allow for a single, efficient memory transaction. These lead to suboptimal memory throughput, resulting in additional memory transactions, increasing latency and reducing overall performance.

Again it is easy to see this in action. If each thread in a warp accesses elements in an array but the elements are not stored consecutively in memory, it results in an uncoalesced memory access pattern. For example, assuming a two dimensional array is stored in row major format, if we were to access the column elements of the array, then it wouldn’t be contiguous accessing. This leads to a lot more memory accesses compared to coalesced access.

The primary impact of this nature of memory accesses is its impact on performance and consequent impact on the final execution time as well. Coalesced memory accesses are crucial for achieving high memory bandwidth and optimal GPU performance. Uncoalesced memory accesses can significantly impact the efficiency of memory transactions and result in performance bottlenecks.

For example, if the ratio of uncoalesced vs coalesced memory transactions in a kernel was given by r1:r2. Here it is obvious that the total execution time is largely impacted by the r1 number of uncoalesced memory accesses. It can be roughly seen that total execution time is directly proportional to r1.

However, including this into an analytical model to accurately predict its quantifiable metric on the total execution time is quite challenging. It is possible to identify which instructions in a kernel result in uncoalesced or coalesced memory accesses. The GPUDrano[8] tool does precisely this. But just the mere number of such transactions isn't sufficient. Let's say there were coalesced accesses. We cannot just assume that this means a single memory trip to the global memory. We will have to consider the size of data being fetched, the memory bandwidth, the cache sizes and so on to determine exactly the number of trips to the global memory. Doing so would provide us a metric to include into the analytical model. This is reserved for future scope of the project. Instead different kernels showcasing coalesced and uncoalesced memory accesses are treated separately and execution times are predicted for the individual kernels.

3. Divergence

Branch divergence is a big problem when it comes to CUDA programs. There is no clear way to counteract the nature of divergences. By design, if there are two branches present and we have a part of the threads executing the first branch and the rest of the threads executing the other branch, both of these cannot execute simultaneously. One of the branches gets executed first and then waits until the other set of threads execute the other branch. So as can be seen the total number of instructions executed is a sum of the instructions present in both branches. This clearly affects the executed time in the final calculations, simply because there are more computations being performed than expected due to the nature of the program. If we had all threads inside a warp execute only one of the branches, then this would have resulted in lesser number of computations and thereby lesser execution time as well.

Therefore to incorporate this influence into an analytical model, we should identify all cases of divergence and figure out which set of threads are being affected by the divergence and accordingly include the excess bit of computations and memory accesses that come along with it for the said threads. This is explored in depth with an example kernel later on in the experiments. But to see how it works consider the following bit of kernel code

```
Begin kernel code
if(threadId is “even”) {
 Do 5 computations
 Access memory 3 times
}
if(threadId is “odd”) {
 Do 10 computations
 Access memory 2 times
}
End kernel code
```

Let’s say there are 320 threads executing this piece of kernel code. It is clear that within every warp (there would be 10 warps in this case since warp size is 32), we have half of the threads in a warp executing the one half of the branch while the rest execute the other branch.

Hypothetically let’s say every computation is similar and takes 5 cycles and every memory access is also similar and takes 100 cycles. The first branch would therefore take: 5*5 + 3*100 = 325 cycles. In a similar fashion, the second branch would take a total of 250 cycles. Due to thread divergence here, what happens here is half of the threads execute the first branch of 325 cycles. Once done, they wait till the other branch of 250 cycles is done executing. These two branches thus execute serially in some sense and end up taking a sum of the two cycles count, in this case it is 575 cycles. If the computations within each branch were independent of each other, we would think that these be executed simultaneously. But this is not the case and we see this inefficiency. This can be accounted for in the analytical model by taking the additional computations and memory accesses count.

4. Cache hits and misses

A big part of memory transactions is the behavior of caches. Good cache management can bring about a lot of improvements and drastically improve the performance. If we are

able to fetch data from cache, whose access latencies are much faster than those of global memory, that will boost performance and reduce execution time considerably. So it would be prudent to consider the effect of cache as well.

Speaking of caches, there are two types of caches. One resides inside every SM, called the L1 cache. It might optionally share the same hardware with the shared memory. This would be the fastest cache available and being able to get cache hits on this cache would be the quickest way to access data (of course after being able to fetch small data from registers). The second level of cache, called the L2 cache is available outside of the SMs. This is larger but slower than the L1 cache.

However determining cache hits and misses would be an uphill task without having to execute programs. Therefore we would have to run programs and use a dynamic profiler to calculate the cache hits and misses. This statistic, if procured, can be simply subtracted from the total global memory accesses which was calculated earlier and the time can be adjusted accordingly. However, as stated before, this cannot be done statically.

The Analytical Model

The analytical model is primarily based on counting the number of instructions and determining their nature and how much they contribute to the overall execution time. This simple yet effective metric forms the crux of the whole execution time calculation.

It is possible to extract the total number of computational instructions present in a kernel code. These simply constitute the non-memory transactional instructions. For example, floating point additions, subtractions, divisions and so on. Every type of instruction carries with itself a certain weight, in this case it is simply the total number of cycles these standard instructions would take to finish execution. For example, from literature, through the use of micro-benchmarking it can be seen that Fused Multiply Add (FMAs), additions, multiplications, divisions take approximately 2, 24, 32, and up to 96 clock cycles (Mei et al. [9]). These values will be used as weights when summing together the total cycles required for computational instructions.

Another basic type of instructions contributing to the execution times are the memory transactions. These could be either load or store instructions. Since we have to deal with a multi-layer memory hierarchy as discussed earlier, we have to be wary of the fact that the type of memory differentiates the total number of cycles it would take to perform a memory transaction with the said memory. It is known that accessing global memory is around 100x times slower than accessing the shared memory (or L1 cache since they share the same hardware). Therefore

for the simulations, it can be considered that accessing shared memory takes around 5 cycles compared to 500 cycles that would be required for accessing the global memory. Similarly it can be considered for the L2 cache, which lies in between L1 cache/ Shared memory and the Global memory in the memory hierarchy that it would take around 250 cycles for memory transactions on it.

These instructions can very well be automatically detected using a kernel code parser. But for the sake of simplicity of the project, these values were determined by hand by simply perusing through the kernel code for multiple different kernels.

Let $n_{computations}$ be the number of computations as calculated by hand. Let $memAccess_{GM}$ be the total clock cycles taken by global memory accesses and $memAccess_{SM}$ be the total clock cycles taken by shared memory accesses. If we were not to consider the caches, we have:

1. $memAccess_{GM} = (loads_{GM} + stores_{GM}) * latency_{GM}$
2. $memAccess_{SM} = (loads_{SM} + stores_{SM}) * latency_{SM}$

We could disable caches during compilation and the above equations would suffice in such a case. If we had to consider the effect of caches, then we would have to slightly modify the above equation as follows:

3. $memAccess_{GM} = (loads_{GM} + stores_{GM} - L1_{hits} - L2_{hits}) * latency_{GM} + L1_{hits} * latency_{L1} + L2_{hits} * latency_{L2}$

Where the loads and stores are those as calculated by hand and the latency values are as mentioned earlier in the section.

Similarly if $N_{threads}$ is the total number of threads spawned as the kernel is launched, F is the processor clock frequency and O is the occupancy and $adjustingWeight$ denote a value which accounts for other parameters influencing the total execution time, then we can denote the total execution time as below:

4. $E.T_{total} = \dfrac{N_{threads}(n_{computations} + memAccess_{GM} + memAccess_{SM})}{F * O * adjustingWeight}$

Where occupancy O is determined by the compute capability, block size, register usage and shared memory usage as described earlier.

The $adjustingWeight$ is catering to all the additional parameters and weights that haven't been considered in the equation but previously discussed. These include divergences, uncoalesced vs coalesced memory accesses, bank conflicts, etc. This parameter is calculated as the ratio of estimated execution time to the actual execution time. This is only calculated once by executing the kernel for some arbitrary input size. However this parameter can serve for other input sizes and we wouldn't need to execute the kernel for such instances. This behaves as a scaling factor in addition to the other parameters defined as part of equation 4. This is quite effective when we observe that execution times scale with increasing input sizes. The parameter remains constant and doesn't change for that specific GPU and problem statement, but it changes for different kernels and different GPUs. Note that this parameter alone wouldn't suffice since the correlation of execution times to the input size is not always simple. The understanding of other contributing factors such as both memory and non-memory computations and occupancy provide the necessary modeling to accurately determine the execution times.

It is important to note that this analytical model can be used on any GPU platform. The only things changing would be the hardware characteristics of the GPU, which may change the latency values and clock rate, but the primary scaling factors depend on the nature of the algorithm. The constants which determine the actual execution time is just taken into account by the adjusting factor. Therefore for simplicity, experimentation has been performed on a single GPU across 4 different kernels of unique computations, to show a metric for accuracy.

CUDA Input Kernels

With the analytical model defined in the previous section, the prediction of execution times for multiple kernels was performed. The 4 different kernels included are:

1. **Matrix Vector Multiplication**: This is a simple CUDA program to multiply a N * N dimensional matrix containing floating point numbers with a vector of dimensions N. The CUDA kernel itself uses N threads to compute the final result, where each thread computes the corresponding element on the product vector.

$$\begin{pmatrix} 1 & 1 & 2 \\ 2 & 1 & 3 \\ 1 & 4 & 2 \end{pmatrix}\begin{pmatrix} 3 \\ 1 \\ 2 \end{pmatrix} = \begin{pmatrix} 1\cdot3 + 1\cdot1 + 1\cdot2 \\ \\ \end{pmatrix}$$ First row,

$$\begin{pmatrix} 1 & 1 & 2 \\ 2 & 1 & 3 \\ 1 & 4 & 2 \end{pmatrix}\begin{pmatrix} 3 \\ 1 \\ 2 \end{pmatrix} = \begin{pmatrix} 1\cdot3 + 1\cdot1 + 1\cdot2 \\ 2\cdot3 + 1\cdot1 + 3\cdot2 \\ \end{pmatrix}$$ next row,

$$\begin{pmatrix} 1 & 1 & 2 \\ 2 & 1 & 3 \\ 1 & 4 & 2 \end{pmatrix}\begin{pmatrix} 3 \\ 1 \\ 2 \end{pmatrix} = \begin{pmatrix} 1\cdot3 + 1\cdot1 + 1\cdot2 \\ 2\cdot3 + 1\cdot1 + 3\cdot2 \\ 1\cdot3 + 4\cdot1 + 2\cdot2 \end{pmatrix} = \begin{pmatrix} 6 \\ 13 \\ 11 \end{pmatrix}$$ last row, then do the addition.

Figure 6: Matrix-Vector multiplication of size 3*3

In Figure 6, we can see the multiplication in action. Thread 1 would be used to compute the first element of the resultant vector, thread 2 for the second one and thread 3 for the third element. In each thread, the computation involves multiplication and addition of the elements, which are essentially fused add multiply(FMA) operations. Each operation can be considered to take 2 clock cycles. The value of the model parameters for this problem statement and for the others can be seen in table 3.

2. **Dot Product**: Here, the dot product of two vectors is performed. We have one thread to compute for every multiplication of two elements from the two vectors and we use a divide and conquer technique to add up these values to generate the final dot product itself. Importantly, shared memory has been used here and it is clear from this example the differentiation of shared memory vs global memory accesses and the changes in latency that this incurs.

3. **Vector Addition**: A simple operation of adding two N-dimensional vectors to generate a N-dimensional vector. With every thread computing every element, there would be N threads spawned in total, contributing to N additions of floating point numbers in the process.

4. **Matrix Addition**: This is an extension of vector addition to matrix addition, where we use N*N number of threads to compute for every N*N element in the final matrix. Here, it is ensured to access the memory in a coalesced manner to improve performance. The model is able to pick up on this as well and generate accurate results.

Influencing Factors	Matrix-Vector Multiplication	Dot Product	Vector Addition	Matrix Addition
$n_{computations}$	$2 * N$	$96 * 1$	$24 * 1$	$24 * 1$
$loads_{GM}$	$2 * N$	2	2	2
$stores_{GM}$	1	$1/(Grid\ Size)$	1	1
$loads_{SM}$	-	$log_2(Block\ Size)$	-	-
$stores_{SM}$	-	$log_2(Block\ Size)$	-	-
$N_{threads}$	N	N	N	N^2

Table 3: Parameter values for every kernel function

Since the inclusion of the effect of caches requires the program to be actually executed for every input size, it hasn't been taken into consideration. Every kernel was compiled keeping L1 cache disabled using the flag -Xptxas -dlcm=cg. All optimisations have also been turned off by setting the -O parameter to zero.

Results

The experiments were run on the CUDA3 CIMS machine with the following properties:

Device 0:

name: NVIDIA TITAN V

Compute capability 7.0

total global memory(KB): 12356288

shared mem per block: 49152

regs per block: 65536

warp size: 32

max threads per block: 1024

max thread dim z:1024 y:1024 x:64

max grid size z:2147483647 y:65535 x:65535

clock rate(KHz): 1455000

total constant memory (bytes): 65536

multiprocessor count 80

integrated: 0

async engine count: 7

memory bus width: 3072

memory clock rate (KHz): 850000

L2 cache size (bytes): 4718592

max threads per SM: 2048

Figure 7: GPU Testbed characteristics

As can be seen, the machine is of a 7.0 compute capability with 145500 KHz clock rate. These values are used to compute the occupancy values as well as the final execution times.

1. **Matrix-Vector Multiplication Results**
   The values reported can be seen in Table 4. The Mean Absolute Error (MAE) was calculated to be 16.37 and the Root Mean Squared Error (RMSE) was calculated to be 24.9. Note that the times were calculated in ms. The most important metric to take away

from here is that the predicted values deviated by less than **3.52%** compared to the actual execution times.

Input Size (N)	Actual Execution Time (in ms)	Predicted Execution time (in ms)	Ratio of Actual to Predicted Execution Time
1024	1.351	1.253382165	0.9277440153
3072	11.185	11.27677658	1.008205327
5120	31.723	31.32234444	0.9873701869
7168	58.963	61.39008574	1.04116286
9216	101.754	101.4800005	0.997307236
11264	151.289	151.5920887	1.002003376
13312	210.26	211.7263503	1.006973986
15360	260.85	281.8827854	1.080631725
17408	410.27	362.061394	0.8824954151
19456	469.544	452.2621759	0.9631944523
21504	590.87	552.4851314	0.935036694
23552	666.611	662.7302602	0.9941784042
25600	772.193	782.9975625	1.013992049
27648	880.77	913.2870383	1.036918876
29696	992.858	1053.598688	1.061177618
31744	1171.141	1203.93251	1.027999626
32768	1290.45	1282.857737	0.9941165768

Table 4: Matrix-Vector Multiplication Execution Times

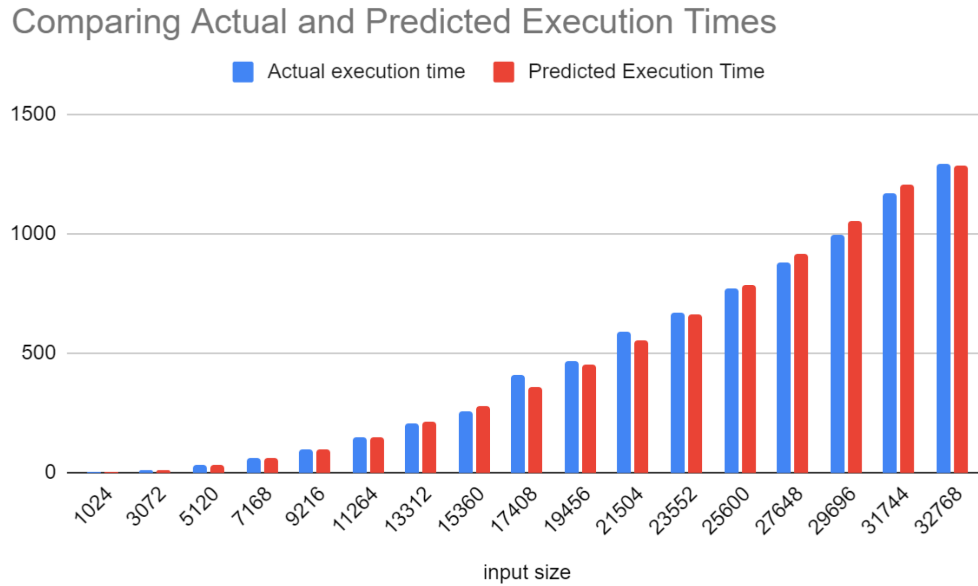

Figure 8: Side to Side Comparison of Actual vs Predicted Execution Times for Matrix-Vector Multiplication

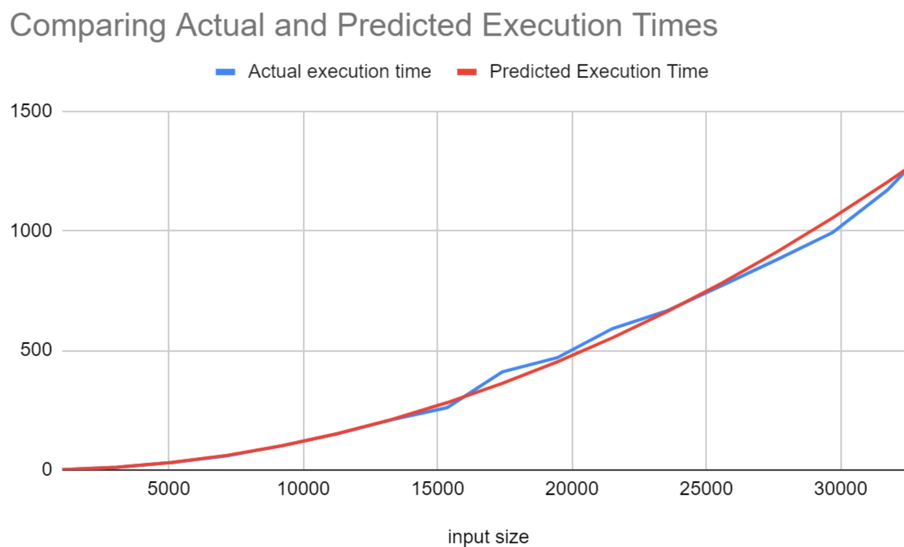

Figure 9: Execution time trends for Matrix-Vector Multiplication

2. **Dot Product Results**

The values reported can be seen in Table 4. The Mean Absolute Error (MAE) was calculated to be 58.719 and the Root Mean Squared Error (RMSE) was calculated to be 78.1. Note that the times were calculated in ms. The most important metric to take away

from here is that the predicted values deviated by less than **8.75%** compared to the actual execution times.

Input Size (N)	Actual Execution Time (in ms)	Predicted Execution time (in ms)	Ratio of Actual to Predicted Execution Time
100000000	121.875	131.8401968	1.081765718
150000000	158.758	197.760277	1.245671254
200000000	265.142	263.6803572	0.9944873206
250000000	355.15	329.6004373	0.9280597982
300000000	556.109	395.5205175	0.7112284058
350000000	528.565	461.4405977	0.873006343
400000000	675.105	527.3606778	0.7811535655
450000000	588.392	593.280758	1.008308675
500000000	686.736	659.2008382	0.9599042983
550000000	719.41	725.1209183	1.007938336
600000000	853.733	791.0409985	0.9265672037
650000000	956.576	856.9610787	0.8958630351
700000000	735.628	922.8811588	1.254548711
750000000	923.623	988.801239	1.070568012
800000000	988.11	1054.721319	1.067412858
850000000	1066.942	1120.641399	1.050330195
900000000	1075.672	1186.56148	1.103088562
950000000	1248.27	1252.48156	1.003373917
1000000000	1362.149	1318.40164	0.9678835721
1050000000	1338.466	1384.32172	1.034259907
1100000000	1445.738	1450.2418	1.003115226
1150000000	1469.911	1516.16188	1.031465089

Table 5: Dot Product Execution Times

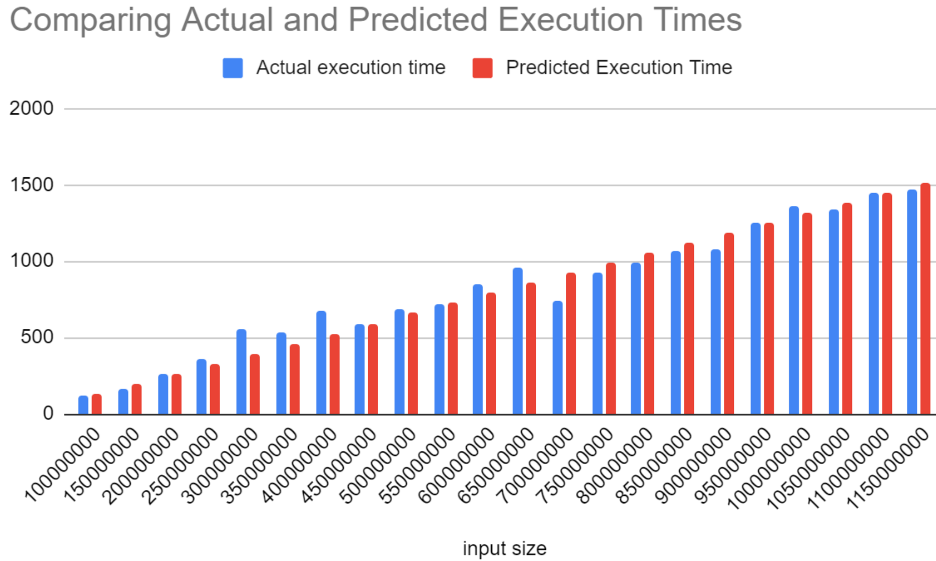

Figure 10: Side to Side Comparison of Actual vs Predicted Execution Times for Dot Product

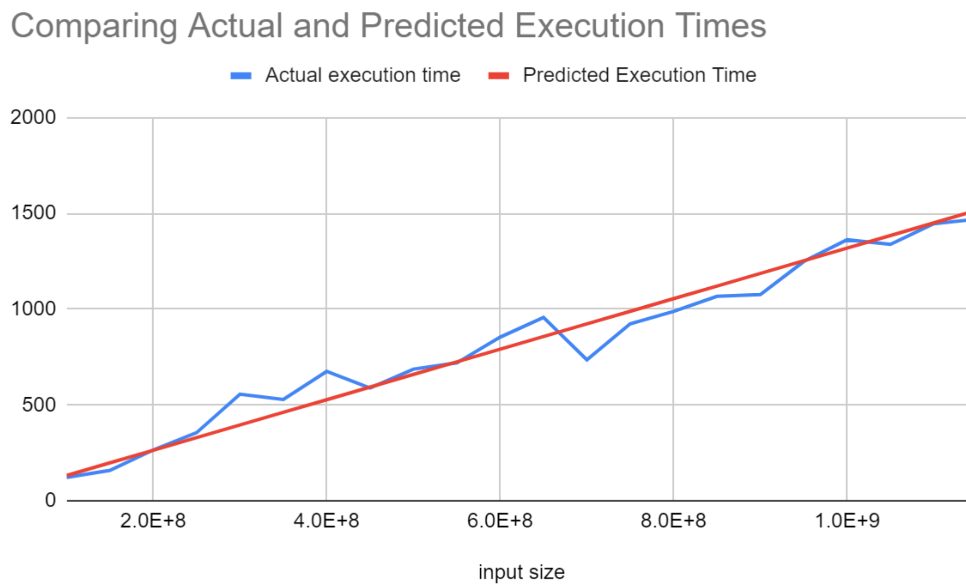

Figure 11: Execution time trends for Matrix-Vector Multiplication

3. **Vector Addition Results**

The values reported can be seen in Table 4. The Mean Absolute Error (MAE) was calculated to be 8.28% and the Root Mean Squared Error (RMSE) was calculated to be **11.96%**. Note that the times were calculated in ms. The most important metric to take

away from here is that the predicted values deviated by less than 5.03% compared to the actual execution times.

Input Size (N)	Actual Execution Time (in ms)	Predicted Execution time (in ms)	Ratio of Actual to Predicted Execution Time
10000000	24.5162	24.20043147	0.9871200051
15000000	37.9632	36.3006472	0.9562061998
20000000	48.2768	48.40086294	1.002569825
25000000	58.667	60.50107867	1.031262527
30000000	72.6608	72.60129441	0.9991810496
35000000	80.7094	84.70151014	1.049462766
40000000	91.4772	96.80172588	1.058206043
45000000	109.6958	108.9019416	0.9927630922
50000000	116.1388	121.0021573	1.041875388
55000000	124.1798	133.1023731	1.071852049
60000000	131.7566	145.2025888	1.102051729
65000000	162.2946	157.3028046	0.9692423812
70000000	164.4648	169.4030203	1.030026001
75000000	211.817	181.503236	0.8568870111
80000000	211.6656	193.6034518	0.9146665862
85000000	217.141	205.7036675	0.9473276235
90000000	213.6954	217.8038832	1.019225885
95000000	255.9418	229.904099	0.898267102
100000000	225.791	242.0043147	1.071806736

Table 6: Vector Addition Execution Times

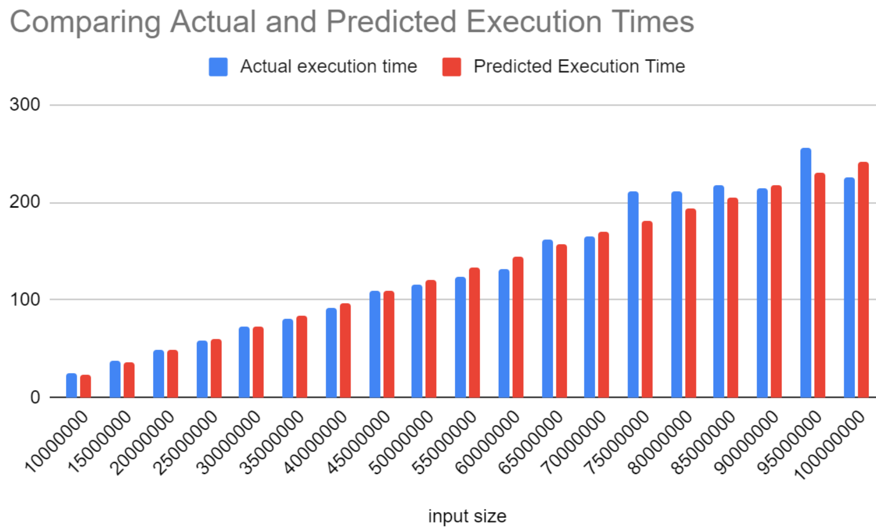

Figure 12: Side to Side Comparison of Actual vs Predicted Execution Times for Vector Addition

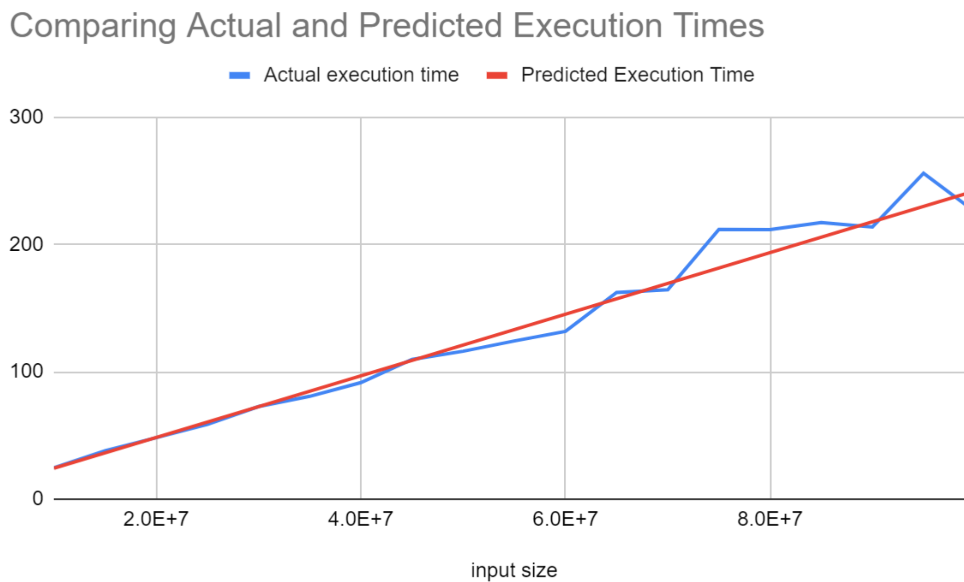

Figure 13: Execution time trends for Vector Addition

4. **Matrix Addition Results**
   The values reported can be seen in Table 4. The Mean Absolute Error (MAE) was calculated to be 6.96% and the Root Mean Squared Error (RMSE) was calculated to be **8.73%**. Note that the times were calculated in ms. The most important metric to take

away from here is that the predicted values deviated by less than 10.3% compared to the actual execution times.

Input Size (N)	Actual Execution Time (in ms)	Predicted Execution time (in ms)	Ratio of Actual to Predicted Execution Time
1000	4.5966	1.581049168	0.3439605726
2000	8.9644	6.324196673	0.7054790809
3000	19.8556	14.22944251	0.7166463121
4000	31.9958	25.29678669	0.7906283541
5000	43.5154	39.52622921	0.9083273785
6000	66.3614	56.91777006	0.8576939313
7000	82.8394	77.47140924	0.9352000285
8000	107.5648	101.1871468	0.9407087334
9000	125.315	128.0649826	1.021944561
10000	154.418	158.1049168	1.023876211
11000	191.8944	191.3069494	0.9969386775
12000	226.9964	227.6710802	1.002972207
13000	259.5502	267.1973094	1.02946293
14000	302.1154	309.885637	1.025719434
15000	341.0948	355.7360628	1.042924321
16000	395.661	404.7485871	1.022968114
17000	455.1898	456.9232096	1.003808103
18000	502.2988	512.2599305	1.019831086
19000	549.2808	570.7587497	1.039101949
20000	616.3246	632.4196673	1.026114595

Table 7: Matrix Addition Execution Times

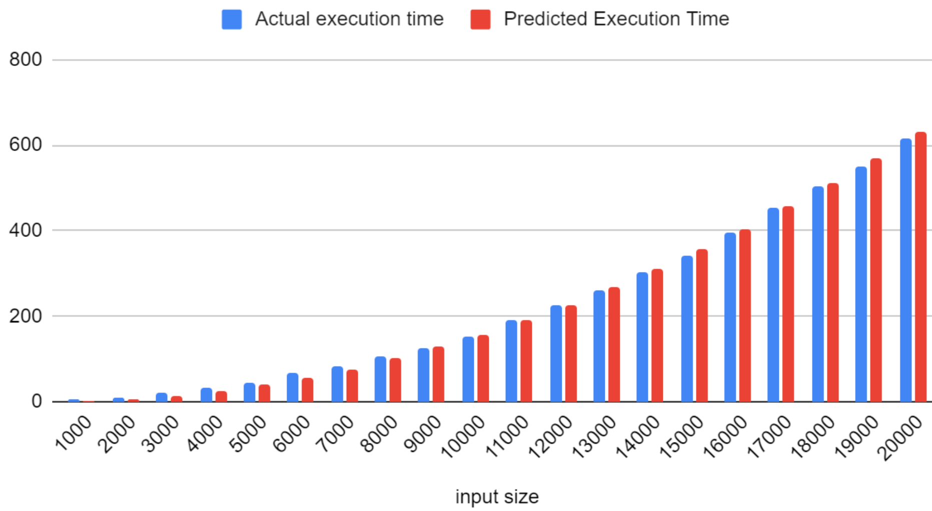

Figure 14: Side to Side Comparison of Actual vs Predicted Execution Times for Matrix Addition

Figure 15: Execution time trends for Matrix Addition

Error metric	Matrix Vector Multiplication	Dot Product	Vector Addition	Matrix Addition
Mean Absolute Error (MAE) (in ms)	16.37	58.18	8.28	6.96
Root Mean Squared Error (RMSE) (in ms^2)	24.92	78.09	11.96	8.73
Deviation (in percentage)	**3.51**	**8.74**	**5.03**	**10.3**

Table 8: Error Metrics for predictions on all kernels

Combining all results, it can be seen that the percentage deviation of the predicted execution times with respect to the actual execution times comes under 6.9%. This error creeps in due to the nondeterministic nature of the actual execution times itself, where a lot of dynamic parameters are at play. However, for the simplicity of the model, it provides a good estimate. This can be very well extended to more kernels and across different GPU systems.

Conclusion and Final Thoughts

In this project, an analytical model for estimating GPU kernel execution times has been proposed. It has been verified against 4 standard kernel functions and the relevant data has been collected and analyzed. Machine learning models require data to estimate a prediction function. In this case, the data required are the execution times of the input program across different input sizes. However, we cannot be expected to execute the program on multiple different inputs and gather execution times, that defeats the purpose of trying to estimate the execution times without actually executing the programs themselves. Thus, in this way, the analytical model comes very handy since we only execute the program once to estimate the adjusting factor value. With that value, we are able to correctly relate how the execution time of a program varies with varying input size, as can be seen from the experimentation and results. Thus, this approach is suitable for an analytical model which utilizes some degree of GPU knowledge. A machine learning model would just assume the underlying function as a black box and wouldn't even care to understand that it is a GPU kernel in execution.

However, there are other techniques as seen from literature surveys which can be consolidated and extended to support machine learning based approaches. With respect to the model proposed,

there still is a lot of scope for improvement. The elimination of the adjusting factor would require taking into account a lot more influencing factors and use the techniques of micro-benchmarking to correctly estimate the relevant weights for the parameters.

Bibliography

1. Gonzalez, Marcos Tulio Amarez. “Performance Prediction of Applications Executed on GPUs Using a Simple Analytical Model and Machine Learning Techniques.” *Institute of Mathematics and Statistics of the University of Sao Paulo*, 2018.
2. Sunpyo Hong and Hyesoon Kim. 2009. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In Proceedings of the 36th annual international symposium on Computer architecture (ISCA '09). Association for Computing Machinery, New York, NY, USA, 152–163. https://doi.org/10.1145/1555754.1555775
3. Sara S. Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, William D. Gropp, and Wen-mei W. Hwu. 2010. An adaptive performance modeling tool for GPU architectures. In Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '10). Association for Computing Machinery, New York, NY, USA, 105–114. https://doi.org/10.1145/1693453.1693470
4. G. Alavani, K. Varma and S. Sarkar, "Predicting Execution Time of CUDA Kernel Using Static Analysis," 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), Melbourne, VIC, Australia, 2018, pp. 948-955, doi: 10.1109/BDCloud.2018.00139.
5. Singhania, Nimit. “Static Analysis for GPU Program Performance” *University of Pennsylvania*, 2018.
6. Qiang Wang, Chengjian Liu, and Xiaowen Chu. 2020. 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). Association for Computing Machinery, New York, NY, USA, 31–40. https://doi.org/10.1145/3366428.3380767
7. “NVIDIA H100 Tensor Core GPU Architecture Overview.” NVIDIA, resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper.
8. Rajeev Alur, Joseph Devietti, Omar S. Navarro Leija and Nimit Singhania. GPUDrano: Detecting Uncoalesced Accesses in GPU Programs. In Proceedings of the 29th

International Conference on Computer-Aided Verification, CAV 2017, pages 507-525. Springer, 2017.

9. Mei, X., Zhao, K., Liu, C., Chu, X. (2014). Benchmarking the Memory Hierarchy of Modern GPUs. In: Hsu, CH., Shi, X., Salapura, V. (eds) Network and Parallel Computing. NPC 2014. Lecture Notes in Computer Science, vol 8707. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-662-44917-2_13