Predicting Kernel Performance: A Machine Learning Analysis

Claire Chen
hc4592@nyu.edu

Alex Manko
asm8755@nyu.edu

Rachel Ren
rr4000@nyu.edu

Yiran Wan
yw7153@nyu.edu

November 20, 2025

1 Abstract

GPU kernel performance varies across operations and hardware, and developers often rely on costly experimentation or intuition to optimize performance. We addressed this problem by generating a dataset containing the speed-up of thousands of CUDA kernel executions. These executions deterministically span several GPU architectures and kernel operation types, and have random problem sizes and kernel-launch parameters. Using this dataset, we trained both linear and nonlinear models to predict GPU speedup. Across all experiments, Random Forests consistently outperformed Linear Regression, achieving $R^2 > 0.99$ for addition kernels and > 0.98. The analysis reveals that predictor importance depends strongly on computational pattern: matrix multiplication is dominated by problem size due to its $O(n^3)$ complexity; addition and subtraction have heavy reliance on GPU architecture at high workloads; and reduction kernels exhibit highly nonlinear synchronization-driven behavior that our models could not capture.

2 Introduction

People need guidance on which kernel and GPU combinations yield the best speedup given a particular problem (addition, matrix multiplication, etc.) There are many factors that shape execution time such as thread-block configuration, memory behavior, and GPU-specific architecture limits. Without a proper way to estimate speedup, developers must either experiment

or rely on intuition, or simply select a configuration at random. This is inefficient and often leads to poor performance. A more methodical approach is important for reducing this trial and error process and for understanding which aspects of the kernel's design are more consequential across different workloads.

Existing literature in an attempt to model a GPU's performance mostly focused on a narrow subset of the problem, such as using analytical models that rely on detailed hardware equations, static methods that use instruction sequences without execution data, and runtime models that target a specific GPU or benchmark suite. We found limited approaches that combined kernel configuration, hardware characteristics, and operation type into a single predictive framework.

Hence, in this project, we approached the problem using machine learning by generating thousands of randomized kernel executions across four GPU architectures and across four operations, namely: vector addition, vector subtraction, matrix multiplication, and minimum reduction. Each kernel run records both the GPU hardware features, kernel parameters, CPU time, and the GPU time and speedup. Then we trained regression models that learn how these factors influenced performance and compared individual operation models with the combined model of all the operations trained across all of the kernels.

3 Literature Survey

Prior work to solve this problem can be divided into several categories:

- Analytical Math Models: rely on architecture-specific equations and quantitative modeling to estimate execution time.
- Static Non-Run-Time Models: Use static code characteristics, such as branching, loops, and instruction type, to predict performance.
- Run-Time Models: Use runtime data (memory calls, caching, etc.) to predict performance.
- Kernel and GPU-Aware Models: Use kernel-launch and GPU architecture to predict performance.

3.0.1 Analytical Math Models

Hong and Kim (2009) proposed one of the first analytical models. Unlike later data-driven models that learn empirical mappings using observed ker-

nel behavior, this research group derives an analytical expression for GPU execution time using architectural parameters, active warp count, memory bandwidth utilization, and two novel metrics (MWP, CWP). Their work is purely analytical, and predicts performance without machine learning training.

Analytical frameworks such as these offer interpretability by decomposing performance into memory and computation overlap, but they rely on many assumptions. Later works move beyond these methodologies, learning performance patterns from empirical data across kernels or GPUs. In this sense, Hong and Kim's model represents the analytical antecedent to modern data-driven prediction methods, which attempt to generalize across architectures and workloads without handcrafted latency terms.

3.0.2 Static Non-Run-Time Models

Guerreiro et al. (2019) propose a static GPU modeling framework that predicts execution time, power, and energy without requiring any runtime profiling. Their approach uses PTX assembly code as input and applies a deep recurrent neural network (LSTM) to model the sequence of GPU instructions. The model learns how the instruction sequence stresses GPU components, enabling accurate predictions of how performance and power scale under different voltage–frequency (DVFS) settings. Their model achieved errors of roughly 5–10 %, comparable to counter-based runtime models — despite being entirely static.

This work establishes that static code characteristics encode predictive information about runtime performance. However, their framework did not analyze this across multiple kernels (i.e., they did not alter the number of threads per block, number of blocks, etc.). As such, in our study, we seek to incorporate kernel-related predictors in our models.

3.0.3 Run-Time Models

Baldini et al. (2014) proposed one of the earliest machine learning frameworks for predicting GPU performance before porting applications. Their model collects dynamic instruction profiles from CPU executions and uses them to predict whether a given application would benefit from GPU acceleration. Using supervised classifiers such as SVM and NNGE, they achieved 77–90% accuracy in classifying beneficial GPU speedups across Rodinia and Parboil benchmarks. However, similarly to the non-run-time literature review, their features are extracted solely from CPU runs, and are not kernel-

specific. Additionally, they performed two separate models on two separate GPUs, rather than incorporating the GPU architecture into the model.

3.0.4 Kernel and GPU-Aware Models

While early works focused either on analytical or instruction-level features, several more recent studies explicitly integrate GPU architectural parameters and kernel configuration into predictive modeling.

Mahmood et al. (2024) proposed a sequence-to-sequence framework that predicts optimal kernel tuning parameters by treating the problem as a translation task. Their model takes input tensor descriptors as input and outputs kernel configuration parameters such as thread block dimensions and vectorization widths. By incorporating GPU hardware constraints through a constrained beam search, they achieved over 90% accuracy on convolutional kernels from AMD's MIOpen library. This work demonstrates that neural networks can learn the complex mapping between problem configurations and GPU-specific kernel parameters.

Lee et al. (2024) introduced NeuSight, which decomposes kernel execution into smaller working sets called tiles that execute independently on GPU streaming multiprocessors. Their framework trains a multi-layer perceptron using both GPU hardware characteristics and kernel execution patterns at tile granularity, then aggregates predictions while bounding them by fundamental performance laws. This approach reduced prediction error from 121.4% to 2.3% when forecasting GPT-3 performance on unseen H100 GPUs, demonstrating that explicit modeling of how kernel configuration interacts with GPU microarchitecture improves accuracy.

Despite these advances, existing kernel and GPU-aware models face several limitations. They typically focus on specific operation types rather than providing a unified framework across diverse operations. Most frameworks either require extensive profiling data or operate as black-box predictors without interpretable insights into performance drivers. Prior work has not systematically investigated how the relative importance of kernel configuration versus GPU architecture shifts across operation categories.

Our work addresses these gaps by building both a universal model across all operations and GPUs, and operation-specific models that reveal how predictor importance varies by computational pattern. This approach bridges interpretability with generality while quantifying whether kernel configuration or GPU architecture dominates performance for different operation types.

4 Proposed Idea

Across these categories, one consistent limitation is the lack of a unified regression framework that can *jointly* learn how kernel configuration, operation type, and GPU architecture interact to influence speedup. Existing models are typically:

- architecture-specific (analytical models),
- instruction-specific but architecture-agnostic (static models),
- or GPU-specific but not kernel-aware (runtime models).

We hypothesize, based on the surveyed literature, that kernel speedup depends *nonlinearly* on factors such as problem size, thread/block configuration, and operation type—each interacting differently with GPU occupancy, memory bandwidth, and instruction throughput. To capture these relationships, we frame our task as a regression problem in which each data point represents one kernel execution with measured speedup as the target variable.

Our approach builds two complementary models:

1. a **universal model** that learns from all kernels, GPUs, and operations to quantify global predictor importance;
2. and a set of **operation-specific models** that isolate how feature relevance shifts across computational patterns (e.g., vector addition vs. matrix multiplication).

By comparing these models, we aim to identify both generalizable performance drivers and operation-specific sensitivities—bridging the gap between interpretability (analytical tradition) and generality (data-driven methods).

5 Experimental Setup

In this section, we describe the data collection procedure, feature preprocessing pipeline, model structure, and validation strategies used to train and evaluate our prediction techniques. Our goal is to build a reproducible and architecture-independent framework that can be applied to any problem, kernel, GPU, and workload size.

Our experimental pipeline consists of three tightly integrated components:

1. a configurable CUDA benchmarking program
2. a randomized parameter–generation script, and
3. an automated experiment runner that executes thousands of kernel launches across GPUs and records their performance.

The output of this pipeline becomes the dataset used for all subsequent machine learning models.

5.1 CUDA Benchmark Program

To systematically evaluate performance/speed-up across diverse kernels, we implement several basic GPU problems inside a single CUDA executable:

- Vector Addition (addvector)
- Vector Subtractor (subtractvector)
- Matrix Multiplication (MatrixMulKernel)
- Minimum Reduction (minReductionKernel)

The program accepts command line arguments in this layout:

```
./gpu_project <ProblemSize> <ThreadsPerBlock> <NumBlocks> <Operation>
```

Once initiated, the program automatically:

- allocates host/device memory
- initializes input variables
- selects appropriate kernel based on command-line argument
- runs a CPU version to verify correctness and collect a sequential execution time
- enforces threading and block constraints (i.e., ensuring there are at least as many threads as there are elements in the problem size)
- measures GPU execution time
- computes and logs the speedup

Each run yields three measurements:

- SingleThreadTime (CPU time)
- KernelTime (GPU time)
- Speedup (ratio)

5.2 Random Configuration Generation

To ensure broad coverage and a robust dataset, we constructed a parameter generator that samples random, but valid, combinations of :

- Problem Size:
  - Matrix Multiplication: Log-uniform sampling between 100 and 2048 (due to computational constraints)
  - Other: Log-uniform sampling between 10^3 and 10^8. Log sampling avoids the high-end clustering that occurs with uniform sampling and ensures problem sizes are evenly distributed across the entire range.
- Threads per Block:
  - Values between 32 and 1024, noting which samples are aligned vs. misaligned (multiples of 32 vs. not), hoping to leverage warp alignment in the model.
  - Each sample is assigned good vs. bad alignment with $\approx 50 - 50$ probability, ensuring balanced representation.
- Grid/Block Dimensions:
  - Matrix Multiplication: 2D grid/block
  - Other: 1D grid/block
- Alignment with a Good/Bad Tag:
  - Good: $ThreadsPerBlock\%32 = 0$
  - Bad: $ThreadsperBlock\%32 \neq 0$
- Number of Blocks
  - Automatically chosen to ensure coverage: $NumBlocks \geq [\frac{ProblemSize}{ThreadsPerBlock}]$
  - If the initial threads or blocks are insufficient or exceed CUDA limits, the generator adjusts them to guarantee coverage while remaining valid.

- Repetition (Noise Reduction):
  - Each configuration is executed multiple times, and the timings are averaged to reduce noise from GPU warm-up and scheduling variability.

The script generates thousands of unique configurations and produces a .csv to dictate the kernel launches and provide a machine learning dataset structure.

5.3 Automated Experiment Runner

We wrote an automation script (written in Python) to parse through all the random configurations (one per row) in the .csv, and run any specified operation (addition, subtraction, multiplication, reduction), collecting speed-up data.

For each configuration:

- the CUDA executable is run multiple times to reduce noise
- CPU time, GPU time, and speedup are averaged
- results are appended to an operation and GPU-specific CSV
- Each row includes:
  - GPU identifier
  - Problem Size
  - Threads per Block
  - Number of blocks
  - Alignment tag
  - Operation type
  - Average CPU time
  - Average kernel time
  - Average speedup

When run across all GPUs and operations, this pipeline produces several thousand labeled data points to train ML models.

5.4 Feature Engineering and Pre-Processing

5.4.1 CUDA GPU Features

In order to convert the GPU specs into predictors, we examined each GPU's hardware:

- Name:
  - CUDA 2: NVIDIA GeForce RTX 2080 Ti
  - CUDA 3: NVIDIA TITAN V
  - CUDA 4: NVIDIA GeForce GTX TITAN X
  - CUDA 5: NVIDIA GeForce RTX 4070
- Total Global Memory(KB):
  - CUDA 2: 11081664
  - CUDA 3: 12339520
  - CUDA 4: 12493056
  - CUDA 5: 12158272
- Shared Memory Per Block:
  - CUDA 2-5: 49152
- Warp Size:
  - CUDA 2-5: 32
- Clock Rate(KHz):
  - CUDA 2: 7000000
  - CUDA 3: 850000
  - CUDA 4: 1215500
  - CUDA 5: 10501000
- Max Threads per SM:
  - CUDA 2: 1024
  - CUDA 3: 2048
  - CUDA 4: 2048
  - CUDA 5: 1536

- L2 Cache Size (bytes):
  - CUDA 2: 5767168
  - CUDA 3: 4718592
  - CUDA 4: 3145728
  - CUDA 5: 37748736
- Memory Bus Width:
  - CUDA 2: 352
  - CUDA 3: 3072
  - CUDA 4: 384
  - CUDA 5: 192
- Multiprocessor Count:
  - CUDA 2: 68
  - CUDA 3: 80
  - CUDA 4: 24
  - CUDA 5: 46

Based on the above specs, we converted each categorical GPU entry (2-5) some its numerical GPU specs. We chose to only include clock rate and global memory because we wanted to avoid having redundant, correlated predictors (i.e., higher global memory usually coincides with higher cache memory, so no need to include both). The overarching goal was to help the models learn the relationships between hardware and speedup.

5.4.2 One-Hot Encoding

We one-hot encoded the categorical variables to ensure they were treated as equal in the regression models.

5.4.3 Scaling and Cleaning

We used StandardScaler to make sure all predictors (besides one-hot encoded ones) were on the same scale, and none had more impact on the model than others due to their magnitude.

5.4.4 Cross-Validation

On all models, we implemented an 80%-20% train-test split, and evaluated all models on the test set to see how the models perform on unseen data and to assess their ability to generalize.

5.4.5 ML-Ready Dataset

Each row in each sub-dataset corresponds to one unique kernel launch containing these predictors:

- Numerical: ProblemSize, ThreadsPerBlock, NumBlocks, Total Global Memory, Shared Memory Per Block, Clock Rate
- Categorical: OperationType, Alignment
- Target: Speedup

5.5 Model Design/Rationale

Since GPU speed-up is shaped by complex interactions between problem size, kernel characteristics, and architecture, we adopted a multi-stage modeling strategy. Our first goal was to (1) evaluate how well simple vs. nonlinear models capture the structure in the data. To achieve this, we trained two model "families":

1. Linear Regression: to measure whether a simple linear mapping can capture high-level trends
2. Random Forests Regression: to capture nonlinear dependencies and interactions between the predictors

One initial concern we had was that problem size would dominate the feature importances in all models. To isolate this, our second goal was to (2) understand how each predictor's importance changes once kernel executions become dominated by computation rather than overhead (i.e., when there is a high enough problem size). So among every model above, we created two versions:

1. Include All Problem Sizes: include problem sizes with significant overhead and problem sizes with intense computation
2. Large-problem-only: Trained only on kernels above certain size thresholds (chosen per operation).

By holding problem sizes high in the large-problem-only model, we would decrease the problem size variance and force the model to learn the other predictors' impact. In doing so, we could also learn which predictors become more important when the GPU is saturated and the problem size becomes less relevant.

6 Experiment & Analysis

6.0.1 "Everything" Model

Results:

Table 1: "Everything" Model Performance

Model	**Dataset (Problem Sizes)**	**MSE**	**R^2**
Linear Regression	All Sizes	440374887791.511	0.232
Linear Regression	> 1000 with mult, $> 1mil$ with others	494766829824.852	0.649
Random Forests	All Sizes	10509528648.414	0.982
Random Forests	> 1000 with mult, $> 1mil$ with others	20945794833.488	0.985

Table 2: "Everything" Model: Linear Regression *Coefficients*

Feature	**All Sizes Model**	**Big Sizes Model**
ProblemSize	-268.903	10339.006
ThreadsPerBlock	640.839	-27602.516
NumBlocks	802.910	-23005.791
Alignment	-6351.305	5350.514
global_mem	-4815.720	-4935.834
clock_rate	-3293.107	543.257
Operation_addition	-86773.930	-210307.107
Operation_multiplication	298175.181	787209.884
Operation_reduction	-87795.296	-211416.760
Operation_subtraction	-87269.148	-212140.981

Table 3: "Everything Model": Random Forests *Importances*

Feature	**All Sizes Model**	**Big Sizes Model**
ProblemSize	0.735	0.393
ThreadsPerBlock	0.004	0.004
NumBlocks	0.003	0.516
Alignment	0.000	0.001
global_mem	0.009	0.009
clock_rate	0.003	0.003
Operation_addition	3.233e-10	7.093e-11
Operation_multiplication	0.246	0.246
Operation_reduction	2.389e-10	2.398e-10
Operation_subtraction	3.864e-11	3.864e-11

Analysis: The "Everything" model provides insight into whether a single regressor can learn GPU performance across several operation types. The results reveal several important properties about GPU performance predictions and the limits of model generalization.

1. **Random Forests achieve near-perfect performance because they can capture interaction effects between operation type and predictors.**
    - Random Forests reached $R^2 \approx 0.98$ in both the unrestricted problem-size version and the restricted one. This suggests that despite mixing operations, there are consistent nonlinear interactions between our predictors that the trees were able to learn.
    - Linear Regression, in comparison, performed poorly. This indicates that the relationship between our predictors (when including operation) is largely nonlinear. As an example, the theory states that in multiplication, problem size more dramatically improves speed-up than addition (which is a non-linear trend).
2. **Problem Size dominates performance in the unrestricted dataset**
    - In the all-problem-sizes dataset, ProblemSize becomes the strongest signal. There are two ideas playing into this. Firstly, within every operation, problem size means a higher workload and higher speed-up because there is more work to separate the GPU from the CPU. Secondly, multiplication, which is significantly more intense than the others, operated on smaller problem sizes (due

to computation/time constraints). As such, the model learned and relies on the premise that low problem sizes correlate with multiplication's high intensity and speed-up.

3. **Operation as a predictor was simply a way to distinguish multiplication and the rest**

    - Operation_multiplication received substantial importance ≈ 0.25 while other operations contributed almost nothing.

4. **Inability to learn Kernel characteristics and GPU-related trends**

    - Even in the large-problem-size model, the model still relied heavily on problem size, or problem-size related proxies (NumBlocks, Operation_multiplication), and barely used the kernel characteristics or GPU architecture

Overall Takeaways: The everything model shows that a universal nonlinear model can learn cross-operation performance trends, but only between operations that have high intensity gaps. This is a significant limitation in including the operation type in a model.

6.0.2 Addition-Only Model

Results:

Table 4: Addition Model Performance

Model	Dataset (Problem Sizes)	MSE	$\mathbf{R^2}$
Linear Regression	All Sizes	778.852	0.666
Linear Regression	$> 1mil$	417.527	0.830
Random Forests	All Sizes	16.896	0.993
Random Forests	$> 1mil$	20.961	0.992

Table 5: Addition: Linear Regression *Coefficients*

Feature	**All Sizes Model**	**Big Sizes Model**
ProblemSize	33.981	7.377
ThreadsPerBlock	-0.285	-0.411
NumBlocks	-2.252	-3.886
Alignment	-0.526	-0.401
global_mem	-21.050	-40.549
clock_rate	-25.433	-48.786

Table 6: Addition: Random Forests *Importances*

Feature	**All Sizes Model**	**Big Sizes Model**
ProblemSize	0.544	0.029
ThreadsPerBlock	0.004	0.006
NumBlocks	0.007	0.014
Alignment	0.000	0.000
global_mem	0.108	0.256
clock_rate	0.337	0.695

Analysis: The addition-only experiments provide a cleaner environment for understanding how our predictors behave when the computational pattern is simple and consistent across all runs. The results reveal several insights:

1. **Random Forest outperforms Linear Regression because addition speed-up exhibits some non-linear patterns.**
    - Random Forest achieves $R^2 \approx 0.99$ on both the unrestricted and restricted datasets, indicating that it captures nonlinear trends in all cases.
    - Linear Regression does reasonably well on the restricted, large-workload dataset. This is because small inputs have somewhat erratic, unpredictable, nonlinear speedup due to overhead costs.
2. **Problem size appears dominant in the unrestricted dataset, because it separates overhead-dominated runs from compute-dominated ones.**
    - When the variance in problem size is large, it is a way stronger predictor than anything related to the kernel or GPU, because workload is most important in GPU performance.

3. **On the contrary, at large problem sizes, architectural gaps, such as global memory and clock rate have a much larger impact.**
    - Once small problem sizes are removed, the Random Forest assigns most importance to global_mem and clock_rate, and the linear model assigns larger coefficients.
    - Kernel-related predictors' importances only changed marginally.
    - This change suggests that when the workload is high, speed-up differences arise mainly due to GPU architectural gaps, rather than problem size and kernel parameters.
4. **Counterintuitive negative weights on clock rate**
    - The linear model assigns a negative weight to the clock rate in both datasets. This does not necessarily mean that clock rate reduces performance. Rather, clock rate can be a proxy variable, hiding other architectural gaps between the GPUs. As an example, the lower-clock-rated GPUs might have higher bandwidth and throughput.

Overall Takeaway: Addition kernels exhibit predictable and simple performance behavior, making them easy to learn, especially by nonlinear models. Once overhead is removed at high problem sizes, GPU architectural gaps, not kernel launch characteristics, have the strongest impact on speed-up, due to GPU saturation.

6.0.3 Matrix Multiplication-Only Model

Results:

Table 7: Multiplication Model Performance

Model	Dataset (Problem Sizes)	MSE	R^2
Linear Regression	All Sizes	464478804081.793	0.778
Linear Regression	$> 1k$	353467221752.143	0.883
Linear Regression	$> 1.5k$	358277685491.741	0.793
Random Forests	All Sizes	36432659533.218	0.983
Random Forests	$> 1k$	138761335397.814	0.954
Random Forests	$> 1.5k$	208731169536.907	0.880

Table 8: Multiplication: Linear Regression *Coefficients*

Feature	**All Sizes Model**	**Big Sizes Model**	**Bigger Sizes Model**
ProblemSize	1401778.261	1583257.249	1195792.408
ThreadsPerBlock	-60181.559	-39098.320	-69282.335
NumBlocks	-80376.192	-130740.245	-11834.298
Alignment	10830.573	5262.832	-48006.944
global_mem	-14128.413	-48818.359	26421.644
clock_rate	-2110.556	-21366.058	-38462.945

Table 9: Multiplication: Random Forests *Importances*

Feature	**All Sizes Model**	**Big Sizes Model**	**Bigger Sizes Model**
ProblemSize	0.974	0.935	0.830
ThreadsPerBlock	0.006	0.013	0.021
NumBlocks	0.004	0.010	0.020
Alignment	0.001	0.002	0.003
global_mem	0.011	0.030	0.097
clock_rate	0.005	0.011	0.028

Analysis:

Matrix multiplication behaves very differently from the element-wise addition kernel, and the results show this distinction. Since workload scales cubically with matrix size, GPU workload increases rapidly, separating the CPU and GPU execution times dramatically.

1. **Random Forest again yields better results than Linear Regression because multiplication introduces strong nonlinear scaling with problem size.**
    - Random Forests achieve high accuracy across all three size restrictions.
2. **Problem Size overwhelming dominates prediction, even when restricting to high problem sizes only**
    - Even in the highest restrictive Random Forests model, the problem size explained 83% of the variance in speed-up.
    - Matrix multiplication is $O(n^3)$, so any small change in problem size will impact CPU speed (and GPU speed-up) way more than any other predictor.

3. **Coefficient patterns in the linear models reveal instability**

    - ThreadsPerBlock, NumBlocks, and Alignment have weights that change sign across the three linear models.
    - This occurs because the linear model tries to model a straight line for behavior that has very non-linear behavior.

Overall Takeaways: Matrix multiplication exhibits very unique results compared to the element-wise kernels. The cubic complexity makes ProblemSize the most important predictor, no matter how little its variance is. Linear models struggle to capture these trends.

6.0.4 Minimum Reduction-Only Model

Results:

Table 10: Reduction Model Performance

Model	**Dataset (Problem Sizes)**	**MSE**	$\mathbf{R^2}$
Linear Regression	All Sizes	1015.969	0.565
Linear Regression	$> 1mil$	14918	0.442
Random Forests	All Sizes	803.059	0.656
Random Forests	$> 1mil$	2255.478	0.155

Table 11: Reduction: Linear Regression *Coefficients*

Feature	**All Sizes Model**	**Big Sizes Model**
ProblemSize	28.169	6.700
ThreadsPerBlock	-4.369	-14.207
NumBlocks	3.827	-3.172
Alignment	-0.473	0.423
global_mem	0.845	2.300
clock_rate	16.131	34.480

Table 12: Reduction: Random Forests *Importances*

Feature	**All Sizes Model**	**Big Sizes Model**
ProblemSize	0.553	0.078
ThreadsPerBlock	0.085	0.230
NumBlocks	0.134	0.216
Alignment	0.008	0.019
global_mem	0.123	0.226
clock_rate	0.096	0.230

Analysis:

Minimum reduction behaves very uniquely compared to the previous kernels because its computation is $log(N)$ complexity, with half of the threads becoming inactive at every step. This idea, coupled with synchronization overhead, leads to complex performance dynamics.

1. **Model performance collapses at large problem sizes because reduction exposes nonlinear overhead patterns that our predictors do not capture.**
   - Both model types have mediocre R^2 on the unrestricted datasets, which crash on the restricted ones.
   - This is because at high problem sizes, the performance relies heavily on thread activity, memory usage, and synchronization points, which our model does not include.
2. **Feature importance changes dramatically between small and large problem sizes.**
   - The kernel and GPU characteristics become more important than the problem size due to the dramatic overhead and synchronization.
3. **Sign changes in linear-regression weights reveal unstable relationships**
   - NumBlocks and Alignment both switched signs in the Linear models.
   - These reversals signal that what improves speed-up at smaller sizes can worsen performance at larger ones.

- This makes sense because in reduction kernels, there are more synchronization points, warp scheduling, and complicated memory accesses, which interact with the predictors in complex ways.

Overall Takeaway: Minimum reduction exhibits distinct behavior compared to the mathematical kernels. Since threads progressively become idle and synchronization overhead accumulates, both our linear and nonlinear models struggle. It can be reasoned that the reduction speed-up depends heavily on the algorithm, which is not related to our predictors. As such, reduction kernels require additional predictors to achieve high accuracy.

6.0.5 Subtraction-Only Model

Results:

Table 13: Subtraction Model Performance

Model	**Dataset (Problem Sizes)**	**MSE**	$\mathbf{R^2}$
Linear Regression	All Sizes	1517.672	0.439
Linear Regression	$> 1mil$	1399.983	0.351
Random Forests	All Sizes	353.783	0.869
Random Forests	$> 1mil$	1134.031	0.475

Table 14: Subtraction: Linear Regression *Coefficients*

Feature	**All Sizes Model**	**Big Sizes Model**
ProblemSize	35.176	-0.634
ThreadsPerBlock	-2.424	-4.045
NumBlocks	-3.697	-6.934
Alignment	-1.615	-0.354
global_mem	-10.288	-31.385
clock_rate	2.249	-6.728

Table 15: Subtraction: Random Forests *Importances*

Feature	**All Sizes Model**	**Big Sizes Model**
ProblemSize	0.769	0.202
ThreadsPerBlock	0.032	0.091
NumBlocks	0.048	0.140
Alignment	0.003	0.008
global_mem	0.097	0.487
clock_rate	0.051	0.071

Analysis: Compared to addition, the subtraction kernel exhibits more variability and weaker predictive performance across all models. While the Random Forests model achieved an R^2 of 0.869 on the full dataset, this represents a significant degradation from the $R^2 \approx 0.99$ observed for addition operations. More concerning, when restricting to large problem sizes, the Random Forests model performance dropped dramatically to $R^2 = 0.475$, whereas the addition model maintained strong performance at $R^2 = 0.992$. Linear Regression similarly showed weaker results, with R^2 values of 0.439 and 0.351 for all sizes and large sizes, respectively, compared to 0.666 and 0.830 for addition.

This disparity is particularly puzzling because vector addition and vector subtraction are mathematically equivalent operations from a computational perspective. Both operations involve identical memory access patterns, require the same number of arithmetic operations per element, and should exhibit similar parallelization characteristics. Theoretically, the models trained on subtraction should yield performance metrics comparable to those trained on addition, yet our results show substantial divergence.

We hypothesize several possible explanations for this unexpected behavior. First, resource contention may have played a role during data collection. Since all experiments were conducted on shared GPU infrastructure, subtraction kernel executions may have experienced more variable background load compared to addition runs, introducing noise into the training data that degraded model quality. Second, the training data collected for subtraction operations may have inadvertently sampled less representative regions of the configuration space, leading to poorer generalization despite similar sample sizes.

Unfortunately, due to time constraints, we were unable to conduct additional experiments to verify these hypotheses. A rigorous investigation would require re-collecting subtraction data under controlled conditions, ensuring equivalent system load during both addition and subtraction mea-

surements, and running multiple training iterations with different random seeds to isolate the effect of training stochasticity from genuine performance differences. Such validation remains important future work to understand whether the observed gap reflects genuine computational differences or artifacts of the experimental setup.

7 Conclusions

Our project successfully demonstrated that machine learning models can effectively predict GPU kernel speedup across different operations and hardware configurations without requiring the code's characteristics as an input. By generating a comprehensive dataset of kernel executions across four GPU architectures and four operation types, we established that speedup prediction is fundamentally a nonlinear problem requiring sophisticated modeling approaches. The most important findings and conclusions are as follows:

- Random Forests significantly outperformed Linear Regression across all operations, achieving R^2 values exceeding 0.99 for addition kernels and 0.98 for matrix multiplication. This substantial performance gap confirms that the relationship between kernel configuration, GPU architecture, and speedup is highly nonlinear, making tree-based ensemble methods essential for accurate prediction rather than simple linear approximations.

- The importance of the predictor varies dramatically by operation type, revealing fundamental differences in how computational patterns interact with hardware. For matrix multiplication, problem size alone explained 83-97% of variance due to its $O(n^3)$ complexity, while for addition and subtraction operations, GPU architectural features such as global memory and clock rate became increasingly dominant at large problem sizes. Reduction operations exhibited particularly complex behavior, with model performance collapsing at large problem sizes, indicating that synchronization overhead and progressive thread idling create performance characteristics not captured by standard kernel configuration features.

- The universal model successfully generalized across all operations with $R^2 = 0.985$, demonstrating that a single framework can learn cross-operational performance patterns. However, the operation-specific models revealed that the relative importance of kernel configuration

versus GPU architecture shifts substantially depending on computational pattern, with memory-bound operations showing greater sensitivity to architectural parameters while compute-bound operations remaining dominated by problem size.

These results bridge the gap between analytical performance models and empirical prediction methods, providing developers with a practical tool for estimating kernel speedup without extensive experimentation. Future work could extend this framework to include additional GPU architectures, more complex operations such as convolution and attention mechanisms, and incorporate additional features such as memory access patterns and warp divergence metrics to improve prediction accuracy for challenging operations such as parallel reduction.

8 References

- Sunpyo Hong and Hyesoon Kim, **An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness**, Technical Report, Electrical and Computer Engineering/School of Computer Science, Georgia Institute of Technology, 2009
- JOÃO GUERREIRO, ALEKSANDAR ILIC, NUNO ROMA, and PEDRO TOMÁS, **GPU Static Modeling Using PTX and Deep Structured Learning**, IEEE Access, vol. 7, pp. 153091-153102, Nov. 2019
- Ioana Baldini, Stephen J. Fink, and Erik R. Altman, **Predicting GPU Performance from CPU Runs Using Machine Learning**, IEEE 26th International Symposium on Computer Architecture and High Performance Computing, 2014
- Khawir Mahmood, Huanqi Cao, and Alex K. Jones, **Optimal Kernel Tuning Parameter Prediction using Deep Sequence Models.** arXiv preprint arXiv:2404.10162, May 2024.
- Seonho Lee, Amar Phanishayee, and Divya Mahajan, **Forecasting GPU Performance for Deep Learning Training and Inference.** arXiv preprint arXiv:2407.13853v3, December 2024.