Estimating the Execution Time of CUDA Programs

Gabriel WARDE

CRIStAL Laboratory, University of Lille, France

Supervised by:

Pr. Giuseppe LIPARI
Nordine FEDDAL

Abstract

The growing use of GPUs in safety-critical real-time systems, such as autonomous driving, demands precise estimation of CUDA kernel execution times to meet strict timing constraints. In this report, we tackle challenges posed by NVIDIA GPU architectural complexity, including hierarchical memory, thread scheduling, and kernel configuration variability by proposing machine learning (ML) models to predict both average-case execution time (ACET) and worst-case execution time (WCET). Using the Rodinia benchmark, we evaluate different ML models like Linear Regression, Random Forests, Neural Networks, Support Vector Machines, and Curve-fitting techniques. Our experiments were conducted with 12 CUDA kernels which belong to 6 distinct applications of the Rodinia benchmark suite, systematically testing their generalization across unseen applications and different computational resource assignments. We propose calibration strategies that enforce safety margins using minimal assigned resources measurements, achieving robust WCET guarantees with reduced profiling overhead. Our results demonstrate that the Curve_fit (with Minimal Offset) model outperforms others, balancing high accuracy with conservative overestimation, even for untested applications like Heartwall and SRAD.

April 1, 2025

CRIStAL Centre de Recherche en Informatique, Signal et Automatique de Lille
Université de Lille
centralelille ÉCOLE CENTRALE DE LILLE
IMT Atlantique Bretagne-Pays de la Loire École Mines-Télécom

Contents

1. Introduction

1.1. Motivation

Modern computing systems rely heavily on GPU acceleration for high-performance tasks like deep learning, computer vision, and scientific simulations. Accurately estimating the execution time of CUDA processes is crucial for optimizing resource usage and ensuring real-time performance in critical applications such as in autonomous driving and automatic control. However, providing accurate upper bounds on execution times is challenging due to the complexity and heterogeneity of modern GPUs, including hierarchical memory structures, diverse kernel configurations, and varying workload patterns.

This report investigates two key methodologies for GPU kernel execution time prediction: analytical modeling and machine learning based approaches. Analytical models provide a deterministic framework that simplifies the relationship between hardware and software. Whereas, ML models utilize data-driven methods to adapt to diverse workloads and architectures. By reviewing state-of-the-art approaches, this report aims to highlight their strengths, limitations, and potential avenues for improvement.

This chapter is organized as follows: In Section 1.2, I will explain the basic functioning of modern GPUs. In Section 1.3, I will outline the main objectives of this project. In Section 2, I will discuss three state-of-the-art papers related to our project to gain a deeper understanding of the problem and explore what has already been accomplished to achieve promising results. In Section 3, I will provide a conclusion based on the insights from these papers. Finally, in Section 4, we will present the future work planned for this project.

1.2. How GPUs Work

GPUs (Graphics Processing Units) [6] are specialized hardware designed to handle massive parallel tasks efficiently. Unlike CPUs, which excel at sequential processing, GPUs are optimized to perform many similar operations simultaneously. They consist of thousands of small cores organized into streaming multiprocessors (SMs), which execute tasks divided into threads and grouped into blocks. This parallel architecture makes GPUs ideal for workloads like deep learning, scientific simulations, and real-time graphics rendering (e.g., in video games), where large amounts of data need to be processed quickly.

1.2.1 CUDA in Machine Learning

CUDA [18] is a parallel computing platform provided by NVIDIA that revolutionized machine learning by enabling efficient GPU utilization for fast computing. Its ability to handle matrix-heavy computations and perform parallel processing makes it ideal for training and serving AI models. While high-level frameworks like TensorFlow and PyTorch rely on CUDA as a back-end, CUDA also enables researchers to design custom, optimized functionality for state-of-the-art innovations. This makes CUDA an essential tool for improving model efficiency, scalability, and performance.

1.2.2 Kernels

In GPU programming, a kernel [5] is a function designed to execute in parallel on the GPU, typically marked with the __global__ specifier in CUDA. When launched, a kernel runs across multiple threads, each with a unique ID used to handle a specific portion of the input data. Threads are grouped into blocks, and blocks form a grid, enabling large-scale parallelism. Kernel launches specify the configuration with syntax like fun<<<blocks, threads>>>(args), allowing efficient computation on massive datasets.

1.2.3 Threads

A thread is the smallest unit of work in a GPU [15]. We can think of it as a single worker performing a specific task. In a GPU, thousands of threads run simultaneously *(see Figure 2)*, each working on a small piece of the overall problem. For example, in image processing, each thread might handle one pixel.

1.2.4 Blocks and Threads per Block

Threads are grouped into blocks. A block is like a team of threads that can work together and share resources. Each block contains a fixed number of threads, usually between 32 and 1024, depending on the task. Blocks are organized into a larger structure called a grid, which represents the entire workload. The way we divide threads into blocks can affect the performance a lot.

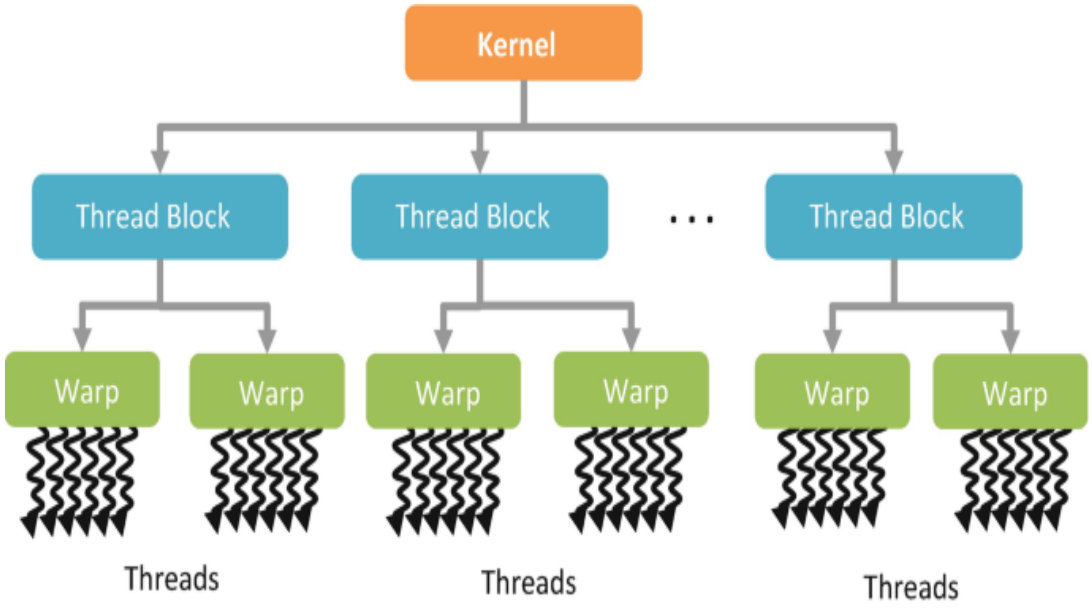

Figure 2: GPU Thread Hierarchy: Threads, Warps, and Blocks. (Image credits: [9])

1.2.5 Shared Memory and Global Memory

GPUs have different types of memory, each with its own purpose [12]:

Shared Memory: This is a fast, small memory that all threads in a block can use. It is useful for storing data that threads need to share or reuse frequently. However, it is limited in size, so it must be used carefully.

Global Memory: This is the main memory of the GPU, accessible by all threads. It's much larger but slower than shared memory. Efficient use of global memory is critical for performance, since poor access patterns can slow processing.

1.2.6 Warps

A warp [10] is a group of 32 threads that execute instructions together in a lock-step fashion. Warps are the basic units of work scheduled by the GPU. If threads in a warp are doing the same thing, the GPU runs efficiently.

Warp divergence [19] happens when threads in a warp take different paths in the code, like in an if-else statement *(see Figure 3)*. Since all threads in a warp must execute the same instruction at the same time, the GPU has to run each path one after the other. This can reduce performance, so minimizing the warp divergence is key to optimizing GPU code.

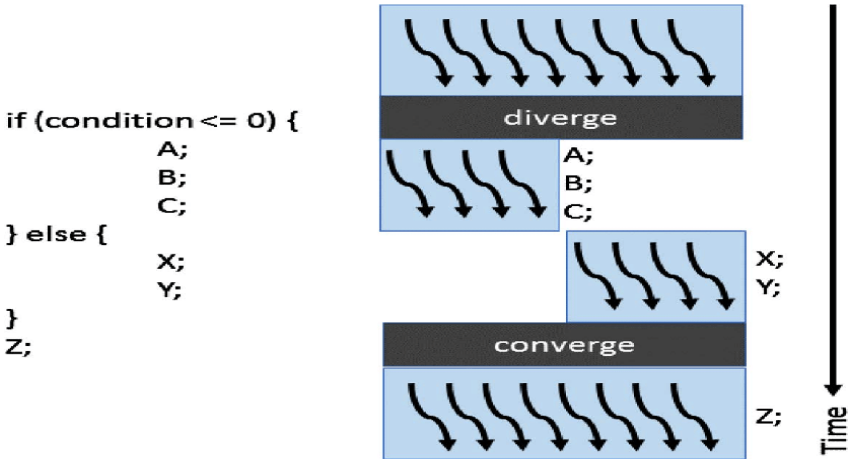

Figure 3: Warp divergence example. (Image credits: [16])

1.2.7 Coalesced Memory Access

Coalesced memory access [13] refers to a memory access pattern where threads within a warp access consecutive memory addresses. This enables the GPU to combine these accesses into a single memory transaction, significantly improving memory throughput. When memory accesses are scattered or non-sequential, multiple transactions are required, which reduces efficiency.

1.2.8 Registers and Occupancy

Each thread has its own set of registers [4] allocated by the compiler, which are the fastest memory available and store operands for operations and intermediate results. Registers

are local to each thread, meaning they are not accessible by other threads or the host. However, the number of registers per thread is limited, and using too many can reduce the number of threads that can run concurrently.

Occupancy [14] measures how well the GPU's resources are being used. It is the ratio of active warps to the maximum number of warps the GPU can handle. High occupancy doesn't always mean better performance, but It is a good indicator of how efficiently the GPU is being utilized.

1.2.9 Atomic Operations

Atomic operations [7] are used to ensure that multiple threads can safely update the same memory location without causing conflicts, such as race conditions. In CUDA, atomic operations prevent race conditions by performing read-modify-write operations without interference from other threads. They can be used in both shared and global memory, with shared memory being faster but limited to threads within the same block. While atomic operations are crucial for preventing errors, they can reduce performance because only one thread can perform the operation at a time.

1.2.10 Streams

Streams in CUDA [11] are sequences of kernels and copy operations that execute in order. Different streams can execute concurrently or out of order with respect to each other. By using non-default streams (instead of the default stream, which is synchronized), we can overlap computations and data transfers, which helps improve performance in CUDA programs. This allows for a concurrent execution model, where kernels can run while memory copies are being made.

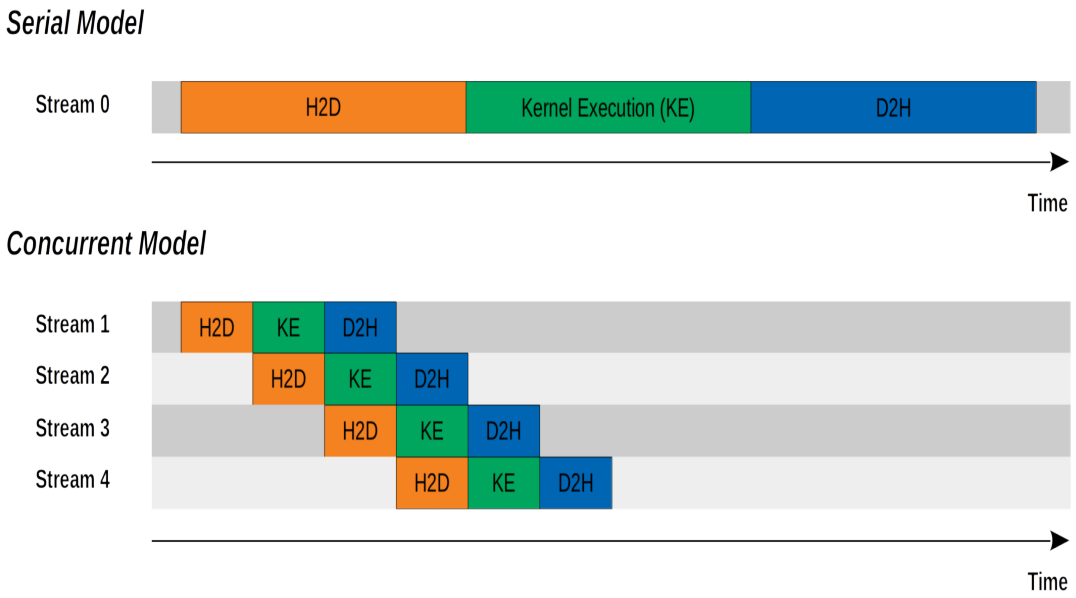

Figure 4: CUDA Stream: Serial Model vs Concurrent Model. (Image credits: [11])

In Figure 4, the difference between a Serial Model and a Concurrent Model in CUDA using streams is illustrated. In the Serial Model, operations such as Host-to-Device (H2D) transfers, Kernel Execution (KE), and Device-to-Host (D2H) transfers are executed sequentially. In contrast, the Concurrent Model leverages multiple streams to overlap these operations, allowing for more efficient use of resources. For example, while one stream performs a kernel execution, another stream can handle a data transfer, leading to improved performance.

1.2.11 Streaming Multiprocessors (SMs)

A Streaming Multiprocessor (SM) [6] is a fundamental processing unit in NVIDIA GPU architectures designed for parallel computing. It manages and executes multiple threads concurrently, utilizing resources such as registers, shared memory, and warp schedulers to optimize instruction throughput. SMs are responsible for dividing workloads into smaller thread blocks and executing them in parallel, enabling high-performance computation for tasks like graphics rendering and general-purpose GPU (GPGPU) applications. SMs also incorporate physical compute units (such as CUDA cores, Tensor cores, etc.) on which the threads are executed.

1.2.12 Texture Processing Cluster (TPC)

A Texture Processing Cluster (TPC) is a fundamental architectural unit in NVIDIA GPUs that groups several Streaming Multiprocessors (SMs) together with dedicated texture units and control logic [8]. Each TPC is responsible for managing texture mapping, shading tasks, and computational tasks by coordinating the work of its resident SMs, thereby enhancing parallel processing and overall rendering efficiency. For instance, a GPU with two TPCs (TPC2) would have half the number of these clusters compared to a GPU with four TPCs (TPC4), which typically translates to roughly double the texture and shading throughput assuming similar SM configurations and clock speeds.

A TPC is essentially a group of SMs. For example, in a given architecture, one TPC may contain two SMs. Consequently, a machine with 16 SMs would consist of 8 TPCs. Through the use of a specific library [2], it is possible to reserve a certain number of TPCs for a kernel, thereby defining a "TPC configuration." For instance, a kernel may be allocated a maximum of 4 TPCs, allowing it to exploit parallelization. The more TPCs reserved for a kernel, the greater the degree of parallelism, and therefore, the shorter the execution time.

Ideally, the execution time is inversely proportional to the number of reserved TPCs, following the relation:

$$\text{Time} = \frac{C}{\text{Number of TPCs}}$$

where **C** is a constant representing the ideal execution time on one TPC (unique to each kernel). In practice, shared resource interference may cause deviations from this theoretical relationship. We will attempt to determine the empirical relationship for a given kernel.

1.3. Objectives

The primary objective of this project is to develop a robust and accurate method to predict the average case and the worst-case execution times (WCET) of CUDA kernels across different TPC configurations.

1.3.1 Data Collection and Preprocessing

The project will begin by collecting data on CUDA kernel execution times through benchmarks executed on NVIDIA GPUs under controlled conditions. Parameters such as thread block sizes, threads per block, and shared memory allocations will be varied to create a comprehensive dataset. The data will then be pre-processed to handle missing values, normalize features, and remove outliers, ensuring high-quality input for machine learning models. This step is crucial for building reliable predictive models that can generalize across different GPU configurations.

1.3.2 Feature Extraction and Selection

Feature extraction will involve collecting thousands of kernel execution metrics using tools like NVIDIA's `ncu` profiler. To reduce dimensionality and avoid overfitting, feature selection techniques such as Pearson's correlation and Spearman's correlation will be applied, and the correlation matrix will be plotted to visualize relationships between features. These methods will help identify the most relevant features, improving model efficiency and accuracy. This step ensures that the models generalize well across different GPU configurations, which is essential for practical deployment in real-world systems.

1.3.3 Model Design, Training, and Evaluation

Machine learning models, including Support Vector Regression (SVR), Random Forest, and Neural Networks, will be designed and trained on the preprocessed dataset. Hyperparameter optimization (fine-tuning) and cross-validation will be used to enhance predictive accuracy and generalization. The models will be evaluated on unseen GPU configurations.

1.3.4 Expected Results

The expected outcome of this project is a robust machine learning model capable of accurately predicting CUDA kernel execution times on both average-case and worst-case scenarios. We will determine the best model for average-case predictions and the best for worst-case predictions. This will enable system designers to optimize performance and meet critical timing needs across diverse GPU configurations, ensuring reliability and efficiency in real-world applications.

2. Related Work and Analysis

This section examines significant research on predicting GPU kernel performance and power consumption. It discusses various approaches, including analytical frameworks and machine learning (ML) methods, emphasizing their advantages, challenges, and practical use cases.

2.1. *Evaluating Execution Time Predictions on GPU Kernels Using an Analytical Model and Machine Learning Techniques* [1]

The study investigates GPU kernel execution time predictions by comparing the Bulk Synchronous Parallel (BSP) analytical model and machine learning (ML) techniques: Linear Regression, Support Vector Machine (SVM), and Random Forest. The BSP model, which divides GPU execution into compute and memory access phases, uses a calibration parameter (λ) for architecture-specific optimizations. It effectively predicts performance for regular workloads but struggles with irregular patterns, such as thread divergence and varying kernel configurations.

The ML models use profiling data and feature selection to enhance prediction accuracy. Two scenarios were explored: first, ML models used the same input features as the BSP model, derived from profiling data and computation parameters; second, a feature extraction process employed techniques like hierarchical clustering and Spearman correlation on the input features, keeping only those with a high correlation to execution times. A dataset of 20 CUDA kernels was tested, comprising 9 classical matrix-vector algorithms and 11 kernels from six applications in the Rodinia benchmark, evaluated on NVIDIA GPUs from Kepler, Maxwell, and Pascal architectures.

Results revealed that the BSP model achieved a reasonable accuracy (MAPE 10.39%) for regular workloads but exhibited errors with irregular ones, where errors reached 28.02%. In contrast, ML models, especially Random Forest, outperformed the BSP model, achieving MAPEs as low as 1.54% for unseen GPUs and 2.71% for unseen kernels. However, for irregular kernels like Maximum Sub-Array (MSA), ML errors remained higher (up to 24.43%), highlighting the challenges posed by irregular workload patterns.

The study highlights the trade-offs: the BSP model offers simplicity but limited flexibility, while ML models provide higher accuracy by capturing non-linear data relationships but require substantial training data and computational resources for training.

2.2. *A Simple Model for Portable and Fast Prediction of Execution Time and Power Consumption of GPU Kernels* [3]

This study introduces a Random Forest-based model designed for efficient and portable prediction of GPU kernel execution times and power consumption. The model's reliance on hardware-independent features, such as PTX-level instruction counts, kernel launch configurations, and memory access volumes, makes it adaptable across different GPU architectures. By employing features derived through the CUDA Flux profiler, the model avoids dependencies on hardware-specific metrics, ensuring flexibility and ease of retraining for new GPUs.

The methodology involves grouping features into categories (e.g., arithmetic, memory, and control flow) to maintain a meaningful yet concise representation. A diverse dataset of 189 kernels derived from benchmark suites including Parboil, Rodinia, Polybench-GPU, and SHOC was used, including both structured and irregular workloads across various GPU architectures, from Kepler to Turing. To minimize noise, data collection involved multiple measurements, while overrepresented kernels were limited to avoid bias. Hyperparameter tuning through nested cross-validation optimized the Random Forest model, ensuring a balance between both, accuracy and computational efficiency.

Results demonstrated a strong performance, with a median MAPE between 8.86% and 13.86% for execution time predictions on professional GPUs such as the K20, Titan Xp, P100, and V100. However, accuracy for the consumer-class GTX1650 decreased significantly, with a MAPE of 52%, due to dynamic frequency variations. Power consumption predictions achieved a median MAPE below 3% across all tested GPUs, highlighting the model's robustness in this area. The prediction latency ranged from 15 to 108 milliseconds, making the model suitable for runtime scheduling. However, the underrepresentation of irregular workloads and long-running kernels in the dataset posed challenges. Expanding dataset diversity, potentially with synthetic workloads, was suggested as a solution.

The strengths of the model include its portability, computational efficiency, and minimal profiling overhead, making it highly applicable for scheduling and optimization tasks in heterogeneous GPU environments. However, the dependence only on static features limits its ability to capture dynamic execution behaviors, such as cache utilization and thread divergence.

2.3. *GPGPU Performance and Power Estimation Using Machine Learning* [20]

The study presents a machine learning-based model for predicting the performance and power consumption of GPU kernels across different hardware configurations. The model uses real hardware measurements to create scaling surfaces that describe how performance and power change with different configurations, such as the number of compute units (CUs), core frequency, and memory bandwidth. By clustering kernels with similar scaling behaviors and using neural networks to classify new kernels, the model provides fast and accurate predictions without needing extensive simulations.

The model operates in two phases: a training phase and a prediction phase. During training, a collection of OpenCL kernels is executed on a real GPU across multiple hardware configurations, and performance counters, execution times, and power consumption data are collected. These data are used to create scaling surfaces and cluster kernels based on their scaling behaviors. In the prediction phase, a new kernel is executed once on a base hardware configuration, and its performance counters are given to a neural network to determine which cluster it belongs to. The scaling surface of the selected cluster is then used to predict the kernel's performance and power consumption on other configurations.

The model was validated using 108 kernels from different benchmark suites, including Rodinia, SHOC, and Parboil, across 448 hardware configurations. The results showed that the model achieved an average prediction error of 15% for performance and 10% for power consumption. These predictions are significantly faster than running the kernels on real hardware or using low-level simulators, facilitating quick design space exploration and real-time operational adjustments.

The study highlights the limitations of traditional methods, such as low-level simulators, which are accurate but usually slow, and analytic models, which struggle with complex scaling behaviors. However, the ML-based model captures the non-linear scaling patterns and hardware interactions, such as cache conflicts, which are difficult to model analytically. But the model requires a substantial training phase and assumes that new kernels will scale similarly to those in the training set. The study demonstrates that ML-based models can effectively balance accuracy and speed, offering a powerful tool for GPU performance and power estimation in both design and runtime optimization scenarios.

3. Methodology

This section outlines the methodology used to estimate the average-case and worst-case execution times of CUDA programs and details our experimental setup and measurement techniques. We combine systematic benchmarking with profiling tools and statistical analysis to ensure accurate and reliable performance evaluations.

3.1. Data Collection And Training

Our study utilized an extensive dataset comprising 767,655 records and 32 features, generated through rigorous execution and profiling applications in the Rodinia Benchmark (citation)[17]. The dataset was constructed by repeatedly running 6 real-world computational applications: needle (bioinformatics), backprop (machine learning), gaussian (linear algebra), bfs (graph traversal), hotspot (thermal simulation), and lud_cuda (matrix decomposition), each of these launched one or more of the 12 unique kernels (see Table 1) hundreds of times to simulate diverse GPU workloads. Performance metrics were systematically collected using the NCU (NVIDIA Nsight Compute CLI) profiling tool, capturing kernel execution times, resource utilization, and hardware counter data. This comprehensive profiling process yielded the 32 features characterizing GPU behavior.

Application Name	Number of Kernels	Associated Kernels
needle	2	needle_cuda_shared_1, needle_cuda_shared_1
backprop	2	bpnn_layerforward_CUDA, bpnn_adjust_weights_cuda
gaussian	2	Fan1, Fan2
bfs	2	Kernel, Kernel2
hotspot	1	calculate_temp
lud_cuda	3	lud_diagonal, lud_perimeter, lud_internal

Table 1: Table Showing The Kernel Count and Names for Each Application

For GPU average-case time estimation (ACET), we split the dataset into 80% for training and 20% for testing to enable robust evaluation of ML and DL models.

On the other hand, for GPU worst-case time estimation (WCET), models were trained on the full dataset to maximize generalizability and we later tested them on two previously unseen applications, Heartwall and SRAD, which exhibit distinct computational patterns and kernel behaviors.

These strategies ensures thorough evaluation: for average-case predictions, models are tested on a reserved portion of the dataset (20%), simulating real-world scenarios where data patterns are familiar. For worst-case scenarios, models are challenged with entirely new applications (e.g., Heartwall and SRAD) which may operate differently and have a different distribution from the training data.

Below (In Figure 5) we can see a detailed correlation matrix between the 32 different features:

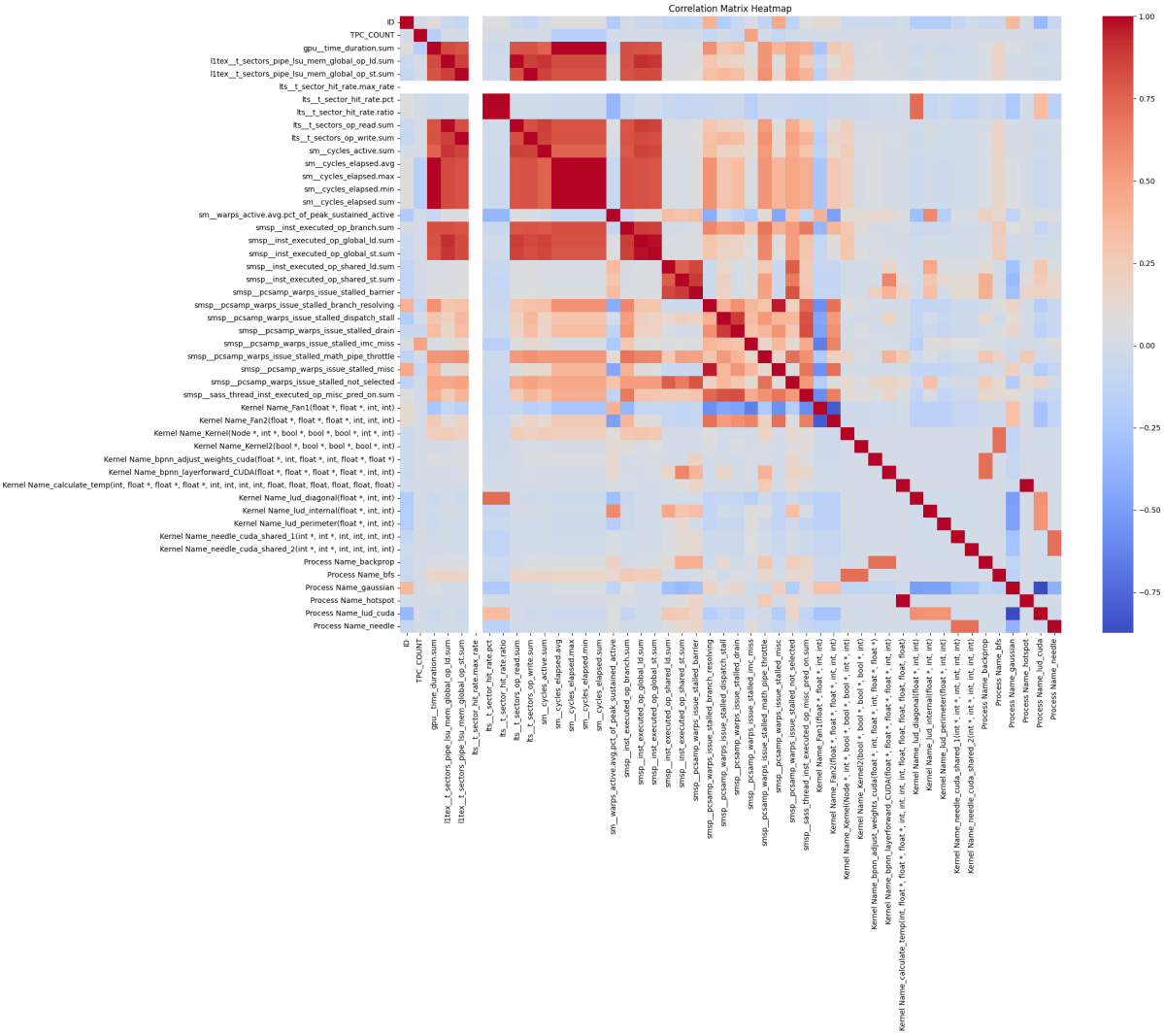

Figure 5: Correlation Matrix of the 32 different features

3.2. Feature Selection

Our feature selection process started with a comprehensive data exploration that involved generating descriptive statistics and visualizing correlation matrices. This helped us identify variables that were strongly correlated with our target, `gpu__time_duration.sum`. We converted categorical data into numerical form using one-hot encoding, applied a correlation threshold of 0.7 to filter out weaker predictors, and then computed the Variance Inflation Factor (VIF) to reduce multicollinearity (we kept the ones that were below VIF 10). After iteratively removing redundant features, we narrowed the selection down to five key features:

- `sm__cycles_elapsed.sum`: Total cycles elapsed, indicating overall workload.
- `smsp__inst_executed_op_branch.sum`: Counts branch instructions, reflecting control flow complexity.
- `smsp__inst_executed_op_global_ld.sum`: Captures global memory load operations, a key factor in latency.

- `lts__t_sectors_op_read.sum`: Measures memory read throughput, indicating data retrieval efficiency.
- `sm__cycles_active.sum`: Represents active cycles, showing the effective computational effort.

3.3. Worst-Case Prediction Used Methods

Multiple strategies were implemented to ensure conservative predictions (worst case), which negatively impacted the accuracy of the models as expected due to overestimation guarantees. There is a trade-off between accurately predicting the worst-case execution time and obtaining good/satisfying average-case results. These approaches combine mathematical calibration at known TPC configurations (e.g., TPC 1 & 7) with safety margins for unseen/intermediate cases. The methods range from fixed percentage buffers to adaptive machine learning techniques, leveraging both model-specific behaviors and post-prediction adjustments. Each strategy strikes a balance between accurate calibration and worst-case safety by using techniques such as setting parameter limits, applying statistical thresholds, or gradually increasing error margins.

Predicting the Worst-Case Execution Time (WCET) is crucial for guaranteeing system reliability and safety, especially in critical applications, as it provides a safeguard against unexpected delays that might not be captured by average-case estimates. However, the main challenge in learning these predictions lies in the high variability and non-linear behavior of GPU performance, which complicates the modeling process.

3.4. Machine & Deep Learning Models Used

Both traditional Machine Learning (ML) and Deep Learning (DL) approaches were applied to effectively predict the Average-Case Execution Time (ACET) and Worst-Case Execution Time (WCET) of CUDA programs. The Average-Case Execution Time (ACET) reflects the typical performance under normal operating conditions, providing insight into everyday efficiency. In contrast, Worst-Case Execution Time (WCET) represents the maximum time a program might take to execute, ensuring safety and reliability even under peak loads.

3.4.1 Models Used On Average-case Execution Time

- **Linear Regression**: Employed default scikit-learn parameters with StandardScaler normalization for both features and targets.
- **MLP Regressor**: Used 30 neurons architecture with ReLU activation, Adam optimizer, and adaptive learning rate over 500 iterations.
- **Random Forest**: Combined 150 estimators with full CPU parallelism (`n_jobs=-1`).
- **Support Vector Machines**: Included both linear (unscaled features) and RBF kernel variants with standardized data.

3.4.2 Models Used On Worst-case Execution Time

- **Linear Regression**: Employed default scikit-learn parameters with StandardScaler normalization for both features and targets.
- **SciPy's curve_fit**: We implemented 3 different approaches using curve_fit from SciPy (later detailed in section **3.6**) on standardized features, with a linear model consisting of 6 coefficients (including an intercept).
- **Random Forest**: Combined 150 depth-constrained trees (`max_depth=8`) with adaptive 3-13% safety margins, testing multiple TPC calibration configurations.
- **3-Layer Neural Network**: Processed scaled features through 64 ReLU input neurons, extracted patterns via 32 hidden units, and output conservative estimates through final linear layer.

3.5. Fine-tuning the models

We employed a trial-and-error approach to optimize the models parameters for improved performance. By experimenting with different hyperparameter values, we aimed to find the best combination that would produce more accurate predictions and better models fitting. This iterative process involved adjusting one or more parameters, observing the results, and refining the choices until satisfactory results were achieved. This method helped us select good settings for each model.

3.6. Inference Calibration

Calibration serves as a critical mechanism to align model predictions with real-world execution behavior, particularly when deployed on applications not encountered during training. By adjusting predictions using a minimal set of observed data points, calibration compensates for distributional mismatches between training and inference scenarios, ensuring outputs remain grounded in empirical measurements.

This process also addresses the worst-case prediction challenge: calibrated models dynamically scale predictions with safety margins, enforcing conservative estimates that reliably bound actual execution times. The calibration framework thus balances accuracy and reliability, enabling robust generalization while maintaining essential safety guarantees for real-time systems. Here are the different calibration strategies that we have used to ensure worst-case predictions on each model:

Linear Regression: Calibration was achieved by enforcing exact alignment between predictions and ground-truth measurements at TPC1 and TPC7, then linearly scaling predictions for intermediate TPC configurations (2–6). A fixed 5% safety margin was applied to these intermediate values by multiplying the raw predictions by a conservative factor of 1.05, ensuring worst-case overestimation. The model used standardized features and targets.

Curve_fit: We evaluated 3 calibration strategies using SciPy's curve_fit:

- **Fixed 3% Margin:** A 6-coefficient model is fitted using TPC1 and TPC7 as anchors, with a fixed 3% inflation applied to other TPC predictions for worst-case estimates.
- **Incremental Margin Adjustment:** Starting with a 3% base margin, the margin is then increased by 2% steps (up to 13%) until every prediction exceeds its actual value on a manual selection of TPCs combinations.
- **Combination Calibration with Minimal Offset:** Linear scaling is applied over various TPC combinations from one measurement per TPC up to nearly all available (using "combinations" method from itertools), then the minimum offset is computed and applied to shift all predictions upward so they always surpass the actual measurements.

Random Forest Regressor: An adaptive calibration strategy tested multiple combinations of TPC anchors (e.g., [1,7], [4,7], etc.), dynamically adjusting safety margins through iterative inflation (3% base + 2% increments per iteration) until all predictions for non-calibration TPCs exceeded actual measurements. Predictions were first generated using an ensemble of 150 trees and then post-processed to enforce exact matches at the calibration TPCs while applying the adjusted margins to other configurations. This hybrid approach combined tree-based non-linearity with explicit worst-case enforcement.

3.7. Evaluation Metrics

We used six metrics to evaluate model performance on evaluation TPCs configurations, balancing the accuracy assessment with conservative prediction analysis:

- **MAE** (Mean Absolute Error): $\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$

  It quantifies average absolute deviation between predictions ($\hat{y}$) and observations (y). It is robust to outliers, directly interpretable in original units (here nanoseconds).

- **Max AE** (Maximum Absolute Error) $\text{Max AE} = \max_{1\leq i\leq n}|y_i - \hat{y}_i|$

  It denotes the worst single prediction error, ensuring that even the most extreme deviation remains within acceptable safety limits.

- **MSE** (Mean Squared Error): $\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$

  It measures squared error average, emphasizing larger deviations through quadratic weighting. It is sensitive to extreme values, used for gradient-based optimization.

- **RMSE** (Root Mean Squared Error): $\text{RMSE} = \sqrt{\text{MSE}}$

  It rescales MSE to original units while preserving error magnitude relationships. It is critical for understanding practical significance of errors in timing predictions.

- $\mathbf{R^2}$ (Coefficient of Determination): $R^2 = 1 - \frac{\sum(y_i-\hat{y}_i)^2}{\sum(y_i-\bar{y})^2}$

  It is a statistical measure that quantifies the proportion of variance in the dependent variable predictable from the independent variables in a regression model. It ranges

from 0 (no explanatory power) to 1 (perfect explanation), indicating how well the model captures the data's variability.

- **MAPE** (Mean Absolute Percentage Error): $\text{MAPE} = \frac{100\%}{n} \sum_{i=1}^{n} \left| \frac{y_i - \hat{y}_i}{y_i} \right|$

  It expresses the average prediction error as a percentage of actual values. It is particularly relevant for evaluating conservative overestimation - lower MAPE indicates better precision while maintaining safe margins.

We primarily relied on Max AE, MAPE, and RMSE to evaluate worst-case GPU time predictions.

MAX AE captures the worst-case deviation, ensuring that even the most extreme prediction error remains within acceptable safety limits.

MAPE, with its relative error interpretation, accounts for execution time scales and ensures conservative, proportional overestimations across different TPC configurations.

RMSE heavily penalizes large errors (due to squaring), helping balance overall accuracy while catching extreme mistakes that could cause issues in GPU workloads.

Together, these metrics provide a comprehensive view of both overall precision and strict worst-case safeguards.

3.8. Minimal TPCs Measurements

We developed a systematic methodology to determine the minimal set of TPC measurements required for secure GPU time estimation through three key steps:

1) **Model Calibration**: Fit affine scaling parameters (a, b) via linear regression between raw model predictions and actual measurements at selected TPC configurations, adapting the general model to application-specific characteristics.

2) **Safety Enforcement**: Calculate a worst-case adjustment offset $\text{offset} = \max(0, -\min(\Delta))$ from prediction residuals Δ, ensuring calibrated estimates conservatively exceed actual values.

3) **Configuration Optimization**: Select the minimal TPC subset by prioritizing: fewest TPCs measurement points, zero-offset security (exact or overestimated predictions), and minimized mean absolute percentage error (MAPE). If multiple configurations perform equally well, the one with the smallest maximum prediction error is chosen.

This framework enables efficient GPU timing characterization through strategic calibration points, balancing measurement economy with robust safety guarantees.

4. Results

This section shows the results of our experiments on CUDA applications. We tested our methods to estimate both the overall execution time and the worst-case execution time. By comparing different models, we identified which approach works best and gained useful insights for further optimizing CUDA programs.

4.1. Average-Case GPU Time Estimation

4.1.1 Linear Regression

Evaluation Metrics:

- MAE: 537.5999
- MAPE: 2.79%
- MSE: 3898270.3828
- RMSE: 1974.4038
- R^2: 0.9999

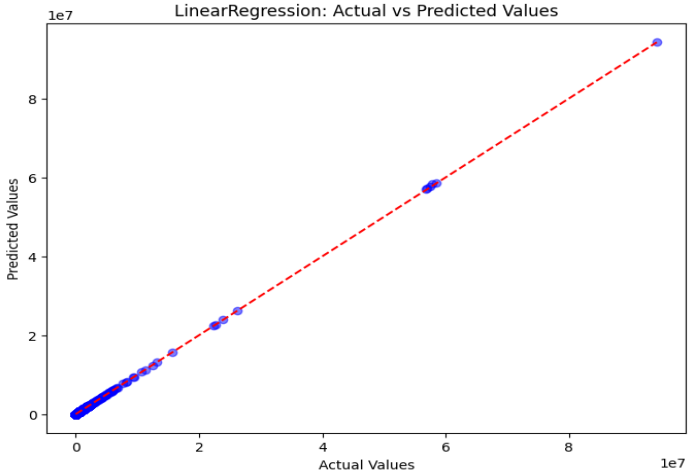

Figure 5: Linear Regression Performance

This graph shows highly accurate predictions using Linear Regression, with the data points closely following the diagonal line that represents a perfect fit.

4.1.2 MLP Neural Network

Evaluation Metrics:

- MAE: 2944.2841
- MAPE: 13.29%
- MSE: 142030287.4442
- RMSE: 11917.6461
- R^2: 0.9995

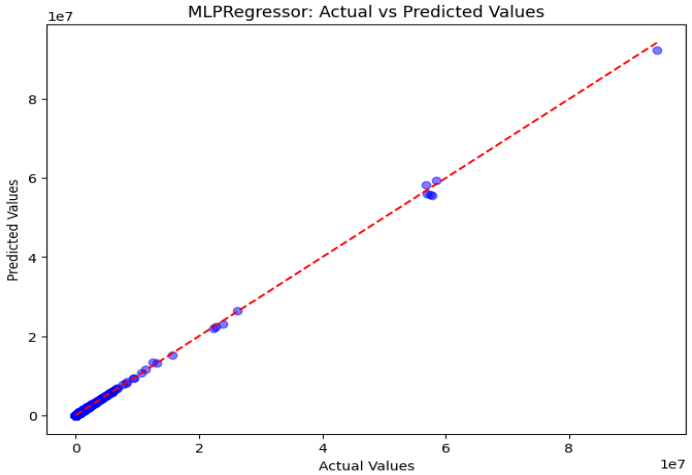

Figure 6: MLP Neural Network Performance

This graph shows accurate predictions using a MLP Neural Network, with data points closely following the diagonal line representing a perfect fit, though the previous graph appeared to fit better.

4.1.3 Random Forests

Evaluation Metrics:

- MAE: 132.8470
- MAPE: 0.15%
- MSE: 35009591.4810
- RMSE: 5916.8904
- R^2: 0.9999

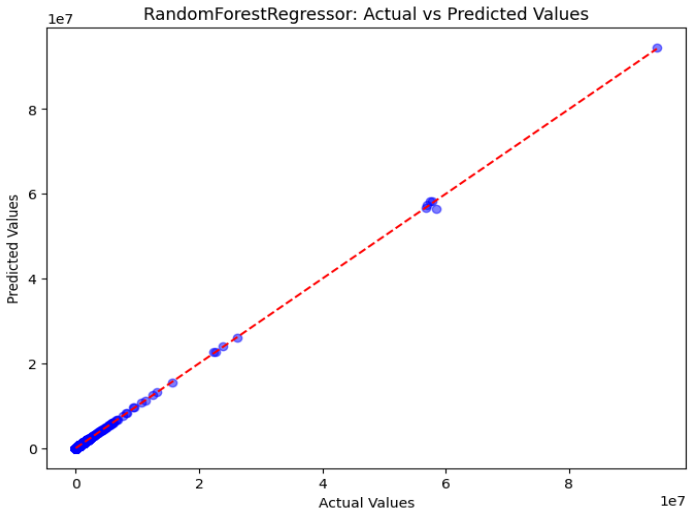

Figure 7: Random Forests Performance

This graph shows highly accurate predictions using Random Forests, with most data points closely following the diagonal line that represents a perfect fit. Although we can see one point that deviates slightly and increases the squared error, the overall fit remains excellent.

4.1.4 LinearSVR

Evaluation Metrics:

- MAE: 2982.5293
- MAPE: 0.92%
- MSE: 469987884.8465
- RMSE: 21679.2040
- R^2: 0.9984

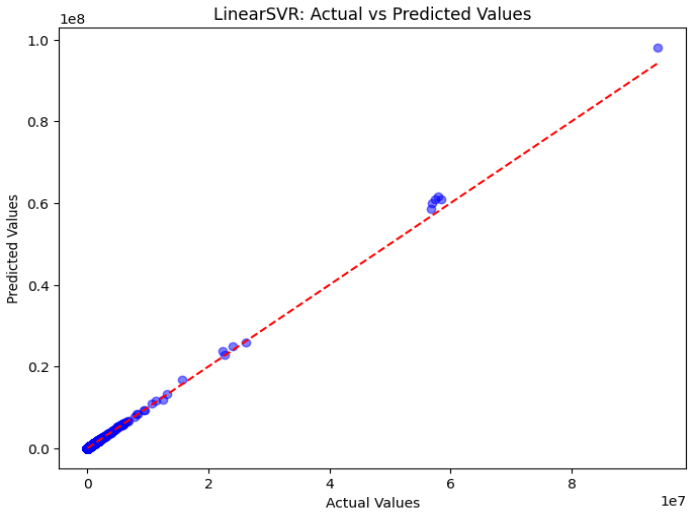

Figure 8: LinearSVR Performance

This graph shows moderate accurate predictions using LinearSVR, with most data points near the diagonal line that represents a perfect fit. Here we can see more point slightly deviating, which will increase the squared errors accordingly, but the overall fit remains good.

4.1.5 SVR(Kernel='rbf')

Evaluation Metrics:

- MAE: 47822.6202
- MAPE: 1010%
- MSE: 158565128015.9958
- RMSE: 398202.3707
- R^2: 0.4675

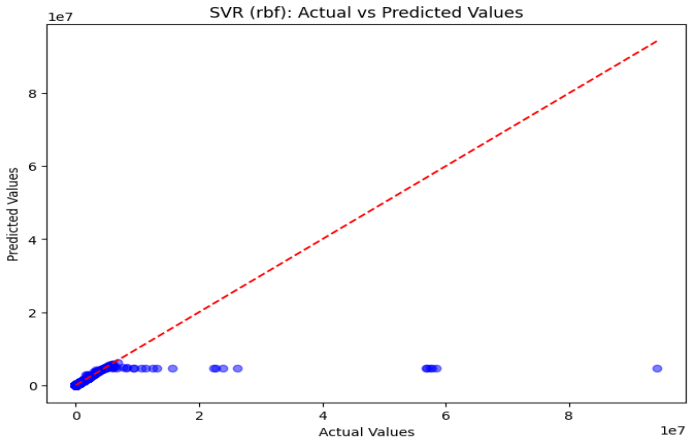

Figure 9: SVR(Kernel='rbf') Performance

This graph shows poor predictions, with most points lying far from the diagonal (ideal fit). Only a few points match the curve well, while the majority deviate substantially. One reason could be that SVR (RBF kernel) maps the data into a high-dimensional space to model complex nonlinear relationships. If the pattern is linear, this extra complexity can degrade performance, which would explain why linear models (e.g., Linear Regression, LinearSVR) performed better.

Method	MAE	MSE	RMSE	R^2	MAPE (in %)
Linear Regression	537.60	3,898,270.38	1,974.40	0.9999	2.79
MLP Neural Network	2,944.28	142,030,287.44	11,917.65	0.9995	13.29
Random Forest	132.85	35,009,591.48	5,916.89	0.9999	0.15
LinearSVR	2,982.53	469,987,884.85	21,679.20	0.9984	0.92
SVR(kernel='rbf')	47,822.62	158,565,128,016	398,202.37	0.4675	1010

Table 2: Table summarizing average-case results for different ML models

The results highlight a trade-off between Random Forest and Linear Regression for average GPU time estimation. Random Forest consistently achieves the lowest average error, making it exceptionally good at minimizing both absolute and relative errors; however, it shows occasional larger deviations (higher MSE). Conversely, Linear Regression offers more consistent absolute errors (lower MSE/RMSE) but a slightly higher relative error.

- MAE is critical if you want to minimize average prediction error (e.g., real-time scheduling).
- MSE is more important if penalizing large outliers (e.g., catastrophic delays) is a priority.

Since average-case estimation typically emphasizes MAE, Random Forest emerges as the optimal choice for its strong balance of precision and robustness, further supported by an excellent R^2 and MAPE.

4.2. Worst-Case GPU Time Estimation

For the new untrained applications (heartwall, srad), the curves (actual and predicted) validate whether models can generalize from minimal calibration data (specific number of points) to predict the full performance trend. The resulting plots visually balance accuracy (proximity to the actual measurements) and safety (systematic overestimation).

4.2.1 Linear Regression + Fixed Margin

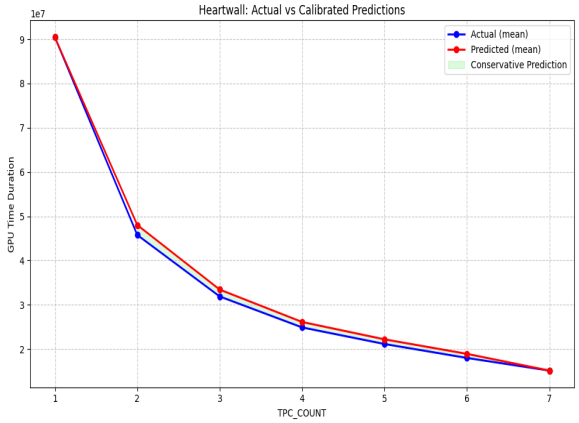

Figure 6: [Heartwall] Application Performance on TPCs 2-6

- MAE: 1412090.2741
- MAX AE: 2900992.2655
- MSE: 2268213151722.1377
- RMSE: 1506058.8142
- R^2: 0.9804
- MAPE: 22.80%

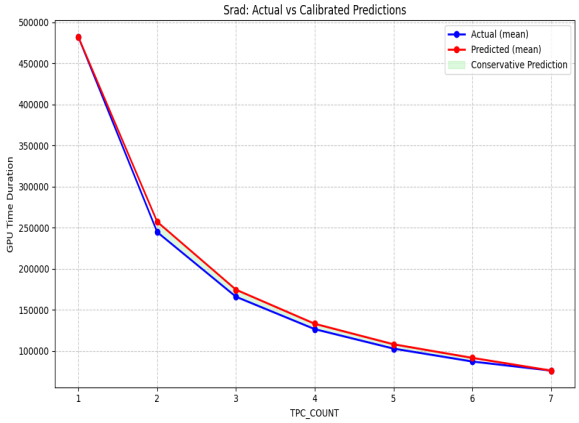

Figure 7: [SRAD] Application Performance on TPCs 2-6

- MAE: 7462.5340
- MAX AE: 107678.9951
- MSE: 211699682.2284
- RMSE: 14549.9032
- R^2: 0.9968
- MAPE: 11.62%

Observation: The conservative predictions for TPCs 2-6 closely follow the actual performance curves for both Heartwall and SRAD applications. The worst-case estimates align tightly with observed trends, reflecting reliable calibration across varied configurations. Overall, it is good results on both.

4.2.2 Neural Network Quantile Regression

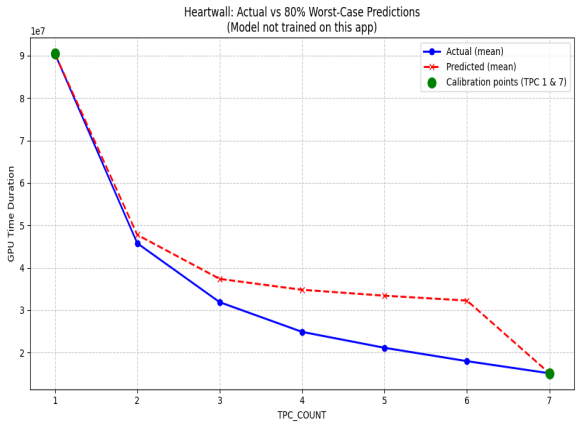

Figure 8: [Heartwall] Application Performance on TPCs 2-6 - 80% Quantile

- MAE: 8792919.6218
- MAX AE: 14755267.6000
- MSE: 98473701061051.4
- RMSE: 9923391.6108
- R^2: 0.1484
- MAPE: 317.39%

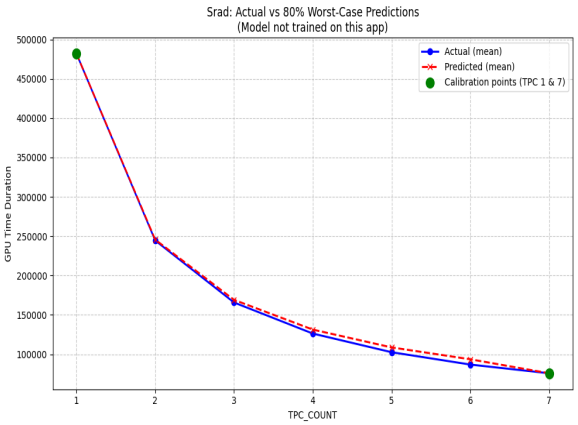

Figure 9: [SRAD] Application Performance on TPCs 2-6 - 80% Quantile

- MAE: 5611.0448
- MAX AE: 105963.9588
- MSE: 211450411.9688
- RMSE: 14541.3346
- R^2: 0.9968
- MAPE: 15.42%

Observation: For Heartwall, the predicted curve is far off from the actual curve, resulting in very large errors. Meanwhile, for SRAD, the predictions closely follow the actual performance. This shows that the model is unstable; working poorly on Heartwall but performing well on SRAD, so it is not reliably applicable to all workloads.

4.2.3 Curve_fit + Fixed Margin

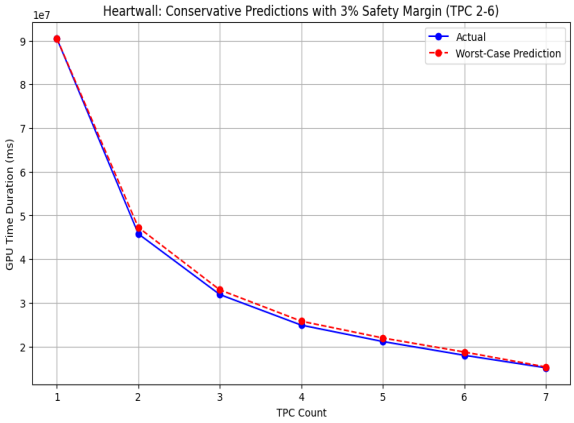

Figure 10: [Heartwall] Application Performance on TPCs 2-6 with Safety Margin

- MAE: 999324.5014
- MAX AE: 2093013.9385
- MSE: 1102506511746.3833
- RMSE: 1050003.1008
- R^2: 0.9905
- MAPE: 3.62%

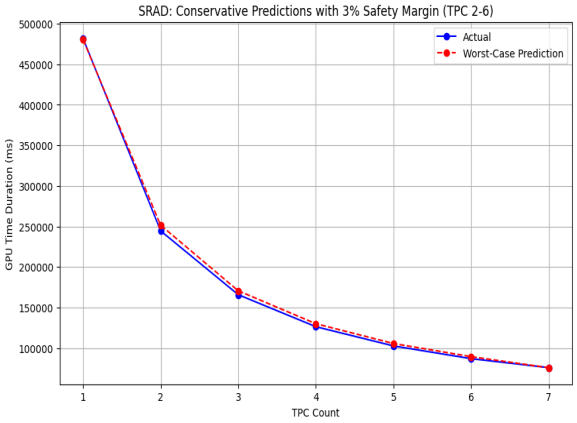

Figure 11: [SRAD] Application Performance on TPCs 2-6 with Safety Margin

- MAE: 4179.5187
- MAX AE: 57799.6478
- MSE: 60660290.9071
- RMSE: 7788.4717
- R^2: 0.9991
- MAPE: 9.71%

Observation: As expected, it is working as good as Linear Regression + Fixed Margin, since they both work similarly in terms of algorithm. Overall it is a good model.

4.2.4 Curve_fit + Adaptive Margin Optimization

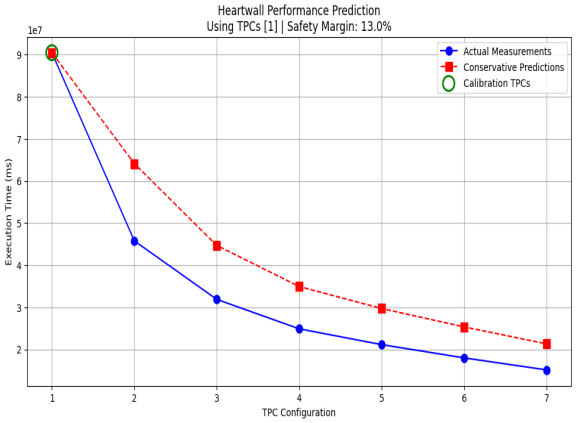

Figure 12: [Heartwall] Application Performance on TPCs 2-7 with Adaptive Margin

- MAE: 10559573.0667
- MAX AE: 19339729.0171
- MSE: 130588200402315.5469
- RMSE: 11427519.4335
- R^2: -0.0763
- MAPE: 50.8%

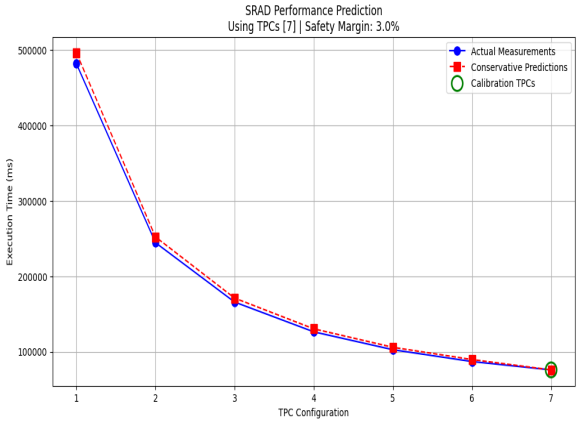

Figure 13: [SRAD] Application Performance on TPCs 1-6 with Adaptive Margin

- MAE: 6210.9353
- MAX AE: 125759.1989
- MSE: 182483947.4667
- RMSE: 13508.6619
- R^2: 0.9990
- MAPE: 9.3%

Observation: In this approach, we manually specified certain TPC configurations to evaluate performance when using only one TPC measurement. For the Heartwall application, providing only TPC1 resulted in predictions that were actually too high compared to the actual curve, ensuring worst-case estimates but sacrificing efficiency. In contrast, for the SRAD application, using TPC7 achieved a good balance between safety and performance. The model is not reliable. The adaptive margin optimization may have contributed to the less stable performance observed in Heartwall, suggesting that the fixed margin approach seemed to be much more reliable and stable.

4.2.5 Curve_fit + Minimal Offset + Full TPCs Combination

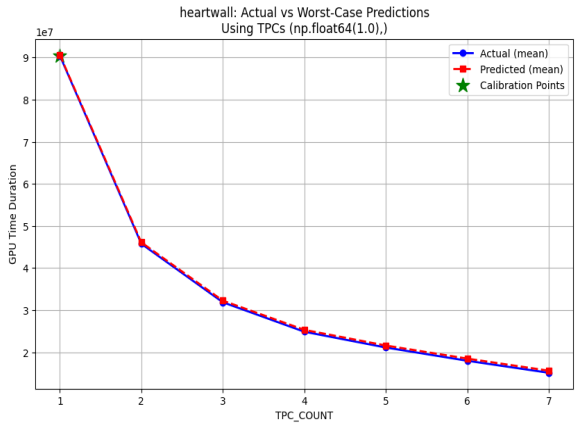

Figure 14: [Heartwall] TPC1 combination chosen as best for a decent evaluation of other TPCs

- MAE: 210162.5796
- MAX AE: 706827.8414
- MSE: 60953450583.2668
- RMSE: 246887.5262
- R^2: 0.9995
- MAPE: 10.86%

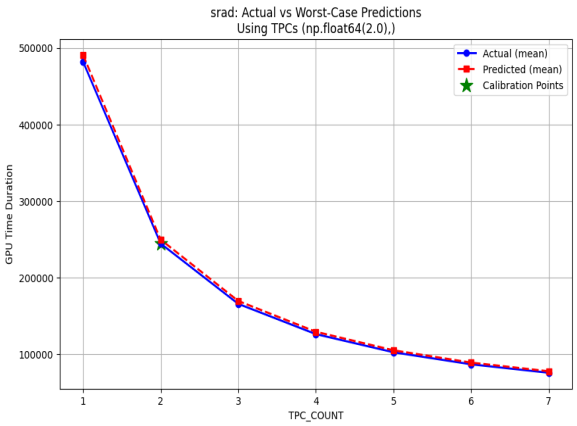

Figure 15: [SRAD] TPC2 combination chosen as best for a decent evaluation of other TPCs

- MAE: 3755.2118
- MAX AE: 70446.1793
- MSE: 51596748.3681
- RMSE: 7183.0877
- R^2: 0.9997
- MAPE: 12.65%

Observation: For this approach, it is interesting to note that after evaluating all possible TPC sets (using the 'combinations' method), the model selects the minimal combination of TPC measurements needed to achieve a prediction that is both safely above the actual values (for worst-case) and very close to the actual curve. It does so by choosing the combination with a zero or nearly zero offset, while prioritizing a small number of TPCs and simultaneously minimizing both the MAPE and Maximum Absolute Error. For the Heartwall application, the model ultimately selected only one TPC measurement (TPC1), ensuring a highly accurate prediction curve that remains safely above and near the actual curve. Similarly, for the SRAD application, it chose only one TPC (TPC2) and successfully adjusted the predictions for the other six TPCs in a very efficient way. It performed even better than Linear Regression or the curve_fit with fixed margin.

4.2.6 Random Forest + Adaptive Margin Optimization

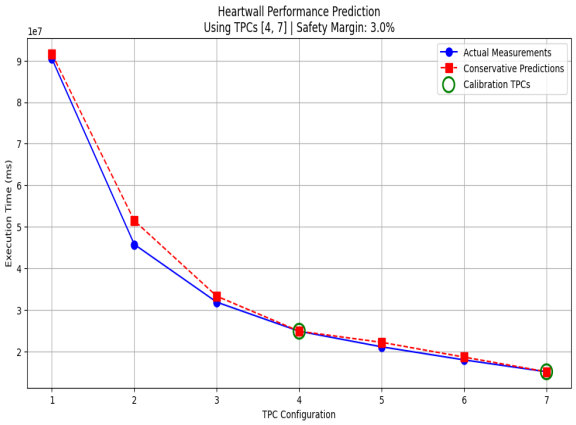

Figure 16: [Heartwall] Application Performance on TPCs 1,2,3,5,6 with Adaptive margin Optimization

- MAE: 2013009.5301
- MAX AE: 6460025.2843
- MSE: 7683274918544.5449
- RMSE: 2771872.0964
- R^2: 0.9897
- MAPE: 13.2%

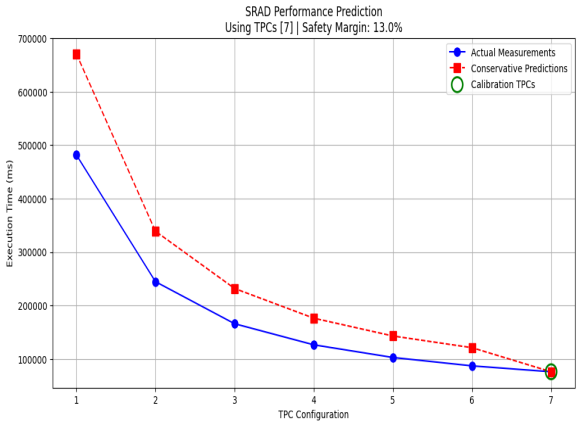

Figure 17: [SRAD] Application Performance on TPCs 1-6 with Adaptive Margin Optimization

- MAE: 78951.6325
- MAX AE: 1730331.2238
- MSE: 32680453899.1507
- RMSE: 180777.3600
- R^2: 0.8163
- MAPE: 42.0%

After manually evaluating various combinations for this approach, we found that using measurements from TPCs 4 and 7 to predict the remaining TPCs produced a good prediction curve for the Heartwall application. For SRAD, using one TPC measurement from TPC7 resulted in a prediction curve that focused more on overestimation than on precision. Some previous models performed much better in terms of both (safety and accuracy) when using a single TPC measurement.

Heartwall	MAE	MAX AE	MSE	RMSE	R^2	MAPE (in %)
Linear Regression (Fixed)	1,412,090.27	2,900,992.27	2,268,213,151,722.14	1,506,058.81	0.9804	22.8
Neural Network (Quantile)	8,792,919.62	14,755,267.60	98,473,701,061,051.40	9,923,391.61	0.1484	317.39
Curve_fit (Fixed)	999,324.50	2,093,013.94	1,102,506,511,746.38	1,050,003.10	0.9905	3.62
Curve_fit (Adaptive)	10,559,573.07	19,339,729.02	130,588,200,402,315.55	11,427,519.43	-0.0763	50.8
Curve_fit (Minimal Offset)	210,162.58	706,827.84	60,953,450,583.27	246,887.53	0.9995	10.86
Random Forest (Adaptive)	2,013,009.53	6,460,025.28	7,683,274,918,544.54	2,771,872.10	0.9897	13.2

Table 3: Heartwall Application Results

SRAD	MAE	MAX AE	MSE	RMSE	R^2	MAPE (in %)
Linear Regression (Fixed)	7,462.53	107,679.00	211,699,682.23	14,549.90	0.9968	11.62
Neural Network (Quantile)	5,611.04	105,963.96	211,450,411.97	14,541.33	0.9968	15.42
Curve_fit (Fixed)	4,179.52	57,799.65	60,660,290.91	7,788.47	0.9991	9.71
Curve_fit (Adaptive)	6,210.94	125,759.20	182,483,947.47	13,508.66	0.9990	9.3
Curve_fit (Minimal Offset)	3,755.21	70,446.18	51,596,748.37	7,183.09	0.9997	12.65
Random Forest (Adaptive)	78,951.63	1,730,331.22	32,680,453,899.15	180,777.36	0.8163	42

Table 4: SRAD Application Results

The results from both applications demonstrate the superiority of the Curve_fit (Minimal Offset) model across almost all evaluation metrics.

For the Heartwall application, this approach achieved exceptional performance, with the lowest MAE (210,162.58), MAX AE (706,827.84) MSE (60.95 billion), and RMSE (246,887.53), alongside a near-perfect R^2 of 0.9995 indicating almost a perfect alignment with the ground truth.

Similarly, in the SRAD application, it outperformed all other models with an MAE of 3,755.21, MSE of 51.59 million, RMSE of 7,183.09, and the highest R^2 (0.9997), and a second-best Max AE of 70,446.18. Curve_fit with a fixed margin performed well, achieving the best Max AE (57,799.65) among all models. However, it did not perform as well on the Heartwall application, which is why we prefer choosing Curve_fit with the minimal offset for better consistency and reliability.

Graphical analysis further validated these results, showing that the Minimal Offset predictions closely followed the actual values while maintaining a slight, consistent overestimation. This minor bias did not compromise accuracy, as reflected in its unmatched metrics, making it the most robust and precise model for both tasks.

Conversely, the Curve_fit (Adaptive) model performed horribly in the Heartwall application, recording the highest errors: an MAE of 10,559,573.07, MSE of 130.59 trillion, and a negative R^2 (-0.0763), which suggests it failed to capture even basic data trends.

For SRAD, the Random Forest (Adaptive) model was the weakest, with an MAE of 78,951.63, MAX AE of 1,730,331.22, and an R^2 of 0.8163, highlighting severe overfitting or an inability to generalize. These results underscore the critical importance of model selection, with Curve_fit (Minimal Offset) emerging as the gold standard for high-precision

applications, while adaptive or overly complex models like Random Forest and Curve_fit (Adaptive) require rigorous tuning to avoid poor performance.

5. Conclusion

This research successfully bridges the gap between precision and safety in estimating CUDA kernel execution times, demonstrating that machine learning can deliver reliable worst-case predictions while minimizing data requirements. By rigorously exploring diverse ML/DL approaches, we identified the Curve_fit (Minimal Offset) model as a standout solution, capable of extrapolating performance across unobserved Thread Processing Clusters (TPCs) using only 1–2 strategic TPC measurements. Remarkably, even for applications like Heartwall and SRAD being excluded entirely from training, the model generalized effectively, proving that sparse, targeted data can capture essential behavioral patterns without compromising safety margins. This breakthrough drastically reduces profiling overhead, making it feasible to deploy in resource-constrained embedded systems where efficiency is paramount.

We are immensely encouraged by these outcomes, which hold transformative potential for safety-critical domains such as autonomous driving, medical diagnostics, and aerospace engineering. The model's ability to balance simplicity with robustness challenges the notion that complex frameworks are necessary for high-stakes applications. Instead, it highlights the power of adaptable, lightweight solutions that prioritize reliability and practicality. These findings not only advance the deployment of trustworthy AI in real-time systems but also pave the way for smarter, safer technologies in mission-critical environments. By proving that minimal input data can yield precise, actionable insights, this work sets a new standard for efficiency in AI-driven systems.

6. Future Works

Future research could extend this methodology to dynamic GPU architectures and multi-kernel interference scenarios, enhancing the adaptability and scalability of ML-driven timing analysis frameworks. Investigating state-of-the-art models and techniques with systematic parameter tuning, may unlock performance gains while preserving safety guarantees. Critically, try to develop specific evaluation metrics tailored to worst-case scenarios would better quantify robustness in safety-critical systems, addressing gaps in current benchmarks. Additionally, exploring the trade-offs between sparse TPC measurements and prediction confidence could refine the balance between minimal profiling overhead and reliability. Finally, validating these frameworks on real-world embedded platforms with heterogeneous workloads would solidify their role in real-time systems, where efficiency, reliability, and safety margins are non-negotiable.

References

[1] Marcos Amaris, Raphael Camargo, Daniel Cordeiro, Alfredo Goldman, and Denis Trystram. Evaluating execution time predictions on gpu kernels using an analytical model and machine learning techniques. *Journal of Parallel and Distributed Computing*, 171:66–78, 2023.

[2] Joshua Bakita and James H. Anderson. Hardware compute partitioning on nvidia gpus. In *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 54–66, 2023.

[3] Lorenz Braun, Sotirios Nikas, Chen Song, Vincent Heuveline, and Holger Fröning. A simple model for portable and fast prediction of execution time and power consumption of gpu kernels. *ACM Trans. Archit. Code Optim.*, 18(1), December 2021.

[4] Carpentries. Gpu programming: Registers, global, and local memory. `https://carpentries-incubator.github.io/lesson-gpu-programming/global_local_memory.html`, 2025.

[5] Cornell Virtual Workshop. Understanding gpu architecture ¿ gpu characteristics ¿ kernels and sms. `https://cvw.cac.cornell.edu/gpu-architecture/gpu-characteristics/kernel_sm`, 2025.

[6] NVIDIA Corporation. Nvidia cuda c programming guide. 2023.

[7] Zhiyuan Huang. Atomic operations in cuda, may 2020. Mi Bu You Chu Xian Ke You Zhong.

[8] JeGX. What is a texture processor cluster or tpc? — geeks3d. `https://www.geeks3d.com/20100318/tips-what-is-a-texture-processor-cluster-or-tpc/`, March 2010. 18 Mar. 2010.

[9] Hyeran Jeon. *GPU Architecture*, pages 1–29. Springer Nature Singapore, Singapore, 2022.

[10] Erik Lindholm, John Nickolls, Scott Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[11] Lei Mao. Cuda stream, feb 2020. Lei Mao's Log Book.

[12] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *ACM Queue*, 6(2):40–53, 2008.

[13] NVIDIA. How to access global memory efficiently in cuda c/c++ kernels, jan 2013. NVIDIA Technical Blog.

[14] NVIDIA. Achieved occupancy. `https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm`, 2025.

[15] NVIDIA Technical Blog. Cuda refresher: The cuda programming model, June 2020.

[16] Vinay Ramakrishnaiah, Suresh Muknahallipatna, and Robert Kubichek. Adaptive region construction for efficient use of radio propagation maps. *Journal of Computer and Communications*, 05:21–51, 01 2017.

[17] Rodinia Benchmark Suite. Start [rodinia]. `https://rodinia.cs.virginia.edu/doku.php?id=start`. Accessed: 2025-03-18.

[18] Daniel Warfield. Cuda for machine learning — intuitively and exhaustively explained, jun 2024. Intuitively and Exhaustively Explained.

[19] Wikipedia contributors. Thread block (cuda programming) — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Thread_block_(CUDA_programming)&oldid=1203479517`, 2024.

[20] Gene Wu, Joseph L. Greathouse, Alexander Lyashevsky, Nuwan Jayasena, and Derek Chiou. Gpgpu performance and power estimation using machine learning. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 564–576, 2015.