Cross-GPU Performance Prediction Using Analytical and Machine Learning Models

Arushi Srivastava as19341 | Debdeep Naha dn2491 | Eva Gupta eg3900 | Shika Rao sr7463

1 Abstract

Predicting GPU performance across platforms traditionally requires executing benchmarks on each target device, a process that is expensive, time-consuming, and impossible for unreleased hardware. This project addresses cross-GPU performance prediction: given kernel measurements on GPU-A, predict runtime on GPU-B without target execution. Our analytical model extends the Roofline framework, decomposing performance into hardware capability, resource utilization (occupancy), and kernel efficiency. Our machine learning approach employs Random Forest models trained on kernel features and GPU architecture specs. We evaluate both methods on 16 diverse CUDA kernels (71 configurations each) across four NVIDIA architectures (Maxwell, Volta, Turing, Ada Lovelace). Our results show that analytical models can handle global generalization (new architectures, new kernels) through physics-based reasoning, while an ML model is able to handle local generalization (scaling configuration patterns) through learned relationships.

Keywords: GPU Performance Prediction, Cross-Architecture Modeling, Roofline Model, Machine Learning, CUDA Kernels, Zero-Shot Transfer

2 Introduction

Motivation

The rapid evolution of GPU architectures presents a fundamental challenge for performance engineering: how can developers and researchers predict application performance on new or unavailable hardware without direct execution? This problem has become increasingly critical as:

- **Procurement decisions** need performance forecasts before hardware purchase
- **Design space exploration** requires evaluating hundreds of configurations across multiple platforms
- **Hardware availability** limits access to latest architectures during development cycles
- **Cloud GPU costs** make extensive benchmarking expensive

Traditional approaches require executing benchmarks on every target platform, making comprehensive performance analysis infeasible for resource-constrained teams and impossible for unreleased hardware.

Problem Statement

Given:

- Performance measurements of CUDA kernels on a source GPU (GPU-A)
- Hardware specifications of both source and target GPU (GPU-B)
- Kernel characteristics (FLOPs, memory accesses, arithmetic intensity, etc.)

Objective: Predict the execution time of kernels on the target GPU without running them on that hardware.

Approach Overview

Our solution has three key components:

1. **Feature Engineering:** We extract comprehensive kernel characteristics including arithmetic intensity, memory access patterns, occupancy metrics, and resource utilization to create transferable performance signatures.
2. **Analytical Modeling:** We develop a roofline model that captures both memory bandwidth and computational throughput limits, incorporating GPU-specific parameters like SM count, warp schedulers, and memory hierarchy.
3. **Machine Learning:** We employ machine learning models trained on source GPU data with architecture-aware normalization to enable zero-shot prediction on target GPUs.

Contributions

- A comprehensive benchmark suite of 16 diverse CUDA kernels with 71 configurations per GPU.
- An analytical model that accounts for architectural differences in memory systems, compute units, and scheduling policies
- Empirical validation across 4 generations of NVIDIA GPUs (Volta, Turing, Ada Lovelace, Maxwell)
- Open-source implementation and dataset for reproducibility

Organization

The remainder of this report is structured as follows: Section 3 reviews related work in performance modeling and prediction. Section 4 describes our proposed methodology including data collection, analytical models, and ML approach. Section 5 details the experimental setup and benchmark kernels. Section 6 presents results and analysis. Section 7 concludes with future directions.

3 Literature Survey

Analytical Performance Models

3.0.1 Roofline Model

The Roofline model, introduced by Williams et al. [1], provides an insightful visual performance model for multicore architectures. The model establishes performance bounds based on two fundamental hardware limits:

$$\text{Performance} = \min\left(\text{Peak Compute}, \text{Bandwidth} \times \text{Arithmetic Intensity}\right)$$

where Arithmetic Intensity is defined as:

$$\text{Arithmetic Intensity} = \frac{\text{FLOPs}}{\text{Bytes Transferred}}$$

Although it provides intuitive upper bounds on achievable performance, it does so by assuming perfect utilization and no overhead. Moreover, it does not capture cache effects or memory hierarchy. Several enhancements were made with varying levels of success.

Yang et al. [2] extended the Roofline model to account for GPU memory hierarchy, creating separate rooflines for DRAM bandwidth, L2 cache bandwidth, L1 cache bandwidth and shared memory bandwidth. Ding and Williams [3] further refined this with instruction-level rooflines that model different arithmetic operations separately. Konstantinidis and Cotronis [4] developed a quantitative roofline model using micro-benchmarks to calibrate actual achievable bandwidths rather than theoretical peaks, improving prediction accuracy.

These hierarchical models improve fidelity but still require profiling or micro-benchmarks on each target GPU, limiting their zero-shot transfer capability.

3.0.2 Pipeline-Based Analytical Models

Hong and Kim [5] developed analytical models that account for memory-level and thread-level parallelism by modeling the GPU execution pipeline:

$$T_{\text{kernel}} = \max\left(T_{\text{compute}}, T_{\text{memory}}\right) \tag{1}$$

Lemeire et al. [6] extracted and analyzed underlying pipeline models from various GPU analytical approaches, identifying common patterns and limitations. However, the major drawback is that these require detailed micro-architectural knowledge and often target-specific profiling data.

Machine Learning Approaches

3.0.3 Early ML for Heterogeneous Systems

Grewe and O'Boyle [7] pioneered ML for heterogeneous system performance prediction, using static code features to predict task partitioning between CPUs and GPUs. They could achieve 77-90% accuracy for partitioning decisions by using decision trees to classify features from OpenCL source code. Kothapalli et al. [8] and Baghsorkhi et al. [9] used regression models on dynamic profile data to predict GPU acceleration potential.

These approaches show good results, but require extensive training data for each target architecture, lacking cross-platform generalization.

3.0.4 Deep Learning Models

Yu et al., 2021 [10] developed Habitat, a deep learning workload prediction program based on hardware features (SM count, memory bandwidth, etc.), execution patterns (kernel launch frequencies, tensor sizes) and runtime statistics. It had good accuracy on training data but exhibited high percentage errors ($> 100\%$) for newer models and unseen GPUs. Li et al. [11] proposed similar learning-based models for DNN workloads but faced comparable limitations.

One interesting observation came from the Lee et al. [12] who demonstrated that naively using machine learning techniques to predict kernel latency does not suffice, as they fail to capture the continuous performance improvements of GPUs resulting from both software and hardware optimizations.

Hybrid Approaches

Ardalani et al. [16] developed XAPP (Cross-Architecture Performance Prediction). Using CPU code features as input to a two-level ML pipeline, the transfer to GPU performance was calculated. However, this model did not perform well while calculating GPU→GPU transfer.

Zheng et al. [17] developed LACross and Qi et al. [18] developed CP^3 which incorporated transfer learning, allowing partial reuse of models on new devices. However, they still require *some profiling* on the target GPU to calibrate the model.

The most promising hybrid model was presented by Lee et al. [12], called NeuSight. It operated by decomposing workloads into tiles and predicting per-tile utilization with ML. It also takes into consideration GPU performance laws to bound final latency. It achieved state-of-the-art accuracy (2-15%) on unseen GPUs. However, it mainly applies to deep learning operators with known execution structure, not general CUDA kernels.

4 Proposed Methodology

We use kernel-level quantities such as FLOPs, bytes transferred, arithmetic intensity, register usage, shared memory usage, and block size to capture how each workload stresses the compute units, memory system, and occupancy. We also give the model hardware-level attributes such as SM count, peak and sustained FP32 throughput, memory bandwidth, warp size, and per-SM resource limits to capture capabilities of each GPU. Together, these features represent the two primary determinants of GPU performance, the computational and memory demands of the kernel, and the architectural capacity of the device executing it. For further details on the data collected and the setup, refer to 5

Analytical Model

We decompose performance of a kernel into three components that can be independently analyzed and transferred:

$$\text{Runtime} = f(\text{Hardware Capability}, \text{Resource Utilization}, \text{Kernel Efficiency})$$

While GPUs have different absolute performance capabilities (e.g., TITAN V has 14.9 TFLOPS vs RTX 2080 Ti has 13.5 TFLOPS), the *efficiency* with which a kernel uses available resources tends to remain stable across architectures. A kernel that achieves 70% of theoretical peak on one GPU will likely achieve similar relative efficiency on another. This is the base hypothesis of our model.

4.0.1 The Roofline Foundation (Hardware Capability)

The Roofline model [1] provides a physics-based upper bound on performance. Every kernel is limited by either **Compute** (how fast the GPU can execute arithmetic operations) or **Memory** (how fast data can be moved to/from DRAM).

The Model:

$$\begin{aligned}\text{Max Performance} &= \min\big(\text{Compute Limit, Memory Limit}\big)\\ &= \min\big(\text{Peak Compute, Arithmetic Intensity} \times \text{Peak Bandwidth}\big)\end{aligned}$$

where Arithmetic Intensity (AI) is:

$$\text{AI} = \frac{\text{FLOPs}}{\text{Bytes Transferred}}$$

This model gives us a theoretical upper bound that accounts for both compute and memory capabilities. It automatically identifies which resource limits performance for each kernel. However, real kernels never achieve theoretical peaks. We need to model this gap with two factors:

$$\text{Actual Performance} = \eta \times O \times \text{Peak Performance}$$

where:

- O = **Occupancy** (0 to 1): Fraction of GPU parallelism utilized, architecture-dependent (varies with SM resources)
- η = **Efficiency** (0 to 1): Quality of resource usage given the parallelism, kernel-dependent (stays relatively constant across GPUs)

This separation allows us to adjust for different GPU designs while preserving kernel characteristics.

4.0.2 Occupancy (Resource Utilization)

Occupancy measures what fraction of a GPU's parallel execution capacity is utilized:

$$O = \frac{\text{Active Warps per SM}}{\text{Max Possible Warps per SM}}$$

Since each SM has limited resources, the number of concurrent thread blocks is constrained by the minimum of the following factors:

$$
\begin{aligned}
B_{\text{reg}} &= \left\lfloor \frac{\text{Registers per SM}}{\text{Regs per thread} \times \text{Threads per block}} \right\rfloor \\
B_{\text{smem}} &= \left\lfloor \frac{\text{Shared mem per SM}}{\text{Shared mem per block}} \right\rfloor \\
B_{\text{threads}} &= \left\lfloor \frac{\text{Max threads per SM}}{\text{Threads per block}} \right\rfloor \\
B_{\text{hardware}} &= \text{Max blocks per SM (hardware limit)}
\end{aligned}
$$

$$
B_{\text{active}} = \min(B_{\text{reg}}, B_{\text{smem}}, B_{\text{threads}}, B_{\text{hardware}})
$$

A kernel using 64 registers/thread might be register-limited on one GPU but shared-memory-limited on another, resulting in different occupancies. By computing occupancy separately for each GPU using its specific resource limits, we account for architectural differences.

4.0.3 Kernel Efficiency Factor

Kernel efficiency captures how effectively a kernel utilizes the available computational resources, independent of specific hardware capabilities. This efficiency factor accounts for all microarchitectural effects not captured by the roofline model or occupancy calculation.

The critical assumption enabling cross-GPU prediction is that kernel efficiency remains relatively stable across GPU architectures. We calculate a different Kernel Efficiency factor for every kernel and configuration used (please refer to Table 2).

We compute efficiency on the source GPU:

$$
\eta_{\text{src}} = \frac{\text{Measured Performance}}{O_{\text{src}} \times \text{Roof}_{\text{src}}}
$$

This factor captures all inefficiencies: non-coalesced accesses, branch divergence, cache misses, etc.

Given measurements on source GPU, we predict target GPU time:

$$
T_{\text{tgt}} = T_{\text{src}} \times \frac{\text{Roof}_{\text{src}}}{\text{Roof}_{\text{tgt}}} \times \frac{O_{\text{src}}}{O_{\text{tgt}}} \times \underbrace{\frac{\eta_{\text{src}}}{\eta_{\text{tgt}}}}_{\approx 1}
$$

Key Assumption: Efficiency transfers, i.e., $\eta_{\text{src}} \approx \eta_{\text{tgt}}$

The roofline ratio accounts for different peak capabilities. The occupancy ratio accounts for different resource utilization. And the efficiency ratio corrects for anything the theoretical roofline bound doesn't capture.

4.0.4 Handling Different Kernel Types

Case 1: Pure Memory Operations (FLOPs = 0)

Examples: vector_add, naive_transpose, strided_copy

These kernels are purely memory-bound, compute capability is irrelevant. Hence, we model bandwidth efficiency. This captures memory access pattern quality (coalesced vs. strided vs. random).

Prediction:

$$BW_{\text{measured}} = \frac{\text{Bytes}}{T_{\text{src}}}$$
$$\eta_{BW} = \frac{BW_{\text{measured}}}{O_{\text{src}} \times BW_{\text{peak,src}}}$$
$$T_{\text{tgt}} = \frac{\text{Bytes}}{\eta_{BW} \times O_{\text{tgt}} \times BW_{\text{peak,tgt}}}$$

Case 2: Compute Operations (FLOPs > 0)

Examples: matmul, conv2d, saxpy, dot_product

These kernels have arithmetic operations. The Roofline identifies the bottleneck and determines if they are compute or memory-bound.

Prediction:

$$\text{Roof} = \min(\text{Peak Compute}, \text{AI} \times \text{Peak BW})$$
$$P_{\text{measured}} = \frac{\text{FLOPs}}{T_{\text{src}}}$$
$$\eta = \frac{P_{\text{measured}}}{O_{\text{src}} \times \text{Roof}_{\text{src}}}$$
$$T_{\text{tgt}} = \frac{\text{FLOPs}}{\eta \times O_{\text{tgt}} \times \text{Roof}_{\text{tgt}}}$$

4.0.5 The Complete Prediction Algorithm

Algorithm 1 Cross-GPU Performance Prediction

Require: T_{src}, FLOPs, Bytes from source GPU
Require: Source GPU specs: C_{src}, BW_{src}, resources
Require: Target GPU specs: C_{tgt}, BW_{tgt}, resources
Ensure: T_{tgt} (predicted time on target GPU)

```
// Compute occupancy on both GPUs
O_src ← ComputeOccupancy(kernel, src_resources)
O_tgt ← ComputeOccupancy(kernel, tgt_resources)
if FLOPs = 0 then                                   ▷ Pure memory case
    BW_measured ← Bytes/T_src
    η ← BW_measured/(O_src × BW_src)
    T_tgt ← Bytes/(η × O_tgt × BW_tgt)
else                                                ▷ Compute or mixed case
    AI ← FLOPs/Bytes
    Roof_src ← min(C_src, AI × BW_src)
    Roof_tgt ← min(C_tgt, AI × BW_tgt)
    P_measured ← FLOPs/T_src
    η ← P_measured/(O_src × Roof_src)
    T_tgt ← FLOPs/(η × O_tgt × Roof_tgt)
end if
return T_tgt
```

Machine Learning Model

We organize our data such that each training sample carries both workload features (FLOPs, bytes, occupancy inputs, etc) and hardware features (compute throughput, memory bandwidth, SM resources, etc). We train the model in a supervised learning setup; we predict the runtime on a target GPU based on other sources GPUs runtime features and source and target GPUs architectural characteristics.

For modeling, we used a Random Forest Regressor as the primary method. Random Forests are well-suited here because they can handle non-linearities in data and GPU performance is influenced by non-linear factors like memory bandwidth, SM count, register pressure, arithmetic intensity, and kernel size parameters. We also tried some more models, please refer to Table 6.

We train 1 ML model per experimental configuration, please see Section 6.

5 Experimental Setup - Hardware and Kernel Data Collection

Hardware Platforms

We conducted experiments on 4 NVIDIA GPU architectures representing different generations and market segments. We collected the data for each GPU using Nsight Compute CLI (NCU) and our own programs for profiling the rest.

Table 1: GPU Hardware Specifications

Specification	**TITAN V**	**RTX 2080 Ti**	**RTX 4070**	**GTX TITAN X**
Architecture	Volta	Turing	Ada Lovelace	Maxwell
Compute Capability	7.0	7.5	8.9	5.2
Launch Year	2017	2018	2023	2015
SM Count	80	68	46	24
CUDA Cores	5120	4352	5888	3072
Tensor Cores	640 (1st gen)	544 (2nd gen)	184 (4th gen)	–
Base Clock	1200 MHz	1350 MHz	1920 MHz	1000 MHz
Boost Clock	1455 MHz	1545 MHz	2475 MHz	1075 MHz
Memory Size	12 GB HBM2	11 GB GDDR6	12 GB GDDR6X	12 GB GDDR5
Memory Bus	3072-bit	352-bit	192-bit	384-bit
Memory Bandwidth	652 GB/s	616 GB/s	504 GB/s	336 GB/s
L2 Cache	4.5 MB	5.5 MB	36 MB	3 MB
Shared Memory/SM	96 KB	64 KB	100 KB	96 KB
Registers/SM	65536	65536	65536	65536
Peak FP32 (TFLOPS)	14.9	13.5	29.1	7.47
Peak FP16 (TFLOPS)	29.8	27.0	116.4	–
TDP	250W	250W	200W	250W

Benchmark Kernels

We developed a comprehensive benchmark suite of 16 CUDA kernels covering diverse computational patterns.

Category	**Kernels**
Memory-bound	vector_add, saxpy, strided_copy_8, naive_transpose, shared_transpose, random_access
Compute-bound	matmul_naive, matmul_tiled, conv2d_3x3, conv2d_7x7
Reductions	reduce_sum, dot_product
Atomic ops	histogram, atomic_hotspot
Special cases	vector_add_divergent, shared_bank_conflict

Problem Sizes and Configurations

Each kernel was tested at multiple problem sizes to capture scaling behavior.

Table 2: Problem Size Categories

Category	**1D (N elements)**	**2D (rows×cols)**	**MatMul (N×N)**
Small	262,144	512×512	256×256
Medium	1,048,576	1024×1024	512×512
Large	4,194,304	2048×2048	1024×1024
XLarge	16,777,216*	4096×4096*	2048×2048

*Reduced for RTX 4070 due to memory constraints

Total configurations per GPU: 71 (varying by kernel type and size)

Measurement Methodology

For each kernel configuration, we performed the following experiments:

- **warmup** : 10–20 iterations, kernel-dependent
- **measurement** : 50–100 iterations, kernel-dependent
- **trials** : 10 independent runs
- **metric** collection : mean execution time in ms and standard deviation

Feature Extraction

For each kernel configuration, we extracted the following features:

Kernel Characteristics	FLOPs (floating-point operations); Memory bytes transferred; Arithmetic intensity (FLOPs/Byte); Working set size; Shared memory usage; Register usage per thread
Memory Access Patterns	Coalesced vs. uncoalesced; Stride information; Random access indicators; Bank conflict potential
Control Flow	Branch divergence flags; Atomic operation counts; Synchronization points
Occupancy Metrics	Theoretical occupancy; Blocks per SM; Threads per block; Grid dimensions

6 Experiments and Analysis

We evaluate two models: (1) an analytic roofline based predictor and (2) a Random Forest ML model (3 models trained on the 3 experimental configurations), under three generalization settings. These experiments test the ability to transfer performance information across GPUs, across new problem configurations, and across entirely new kernels.

We have picked 3 experimental configurations:

1. Testing the same kernel and the same configuration on a new GPUs: To test the generalization ability of our model to new GPUs.
2. Testing a new configuration of the same kernel on the same GPU: To test the generalization ability of our model to predict performance of new kernel configurations.
3. Testing an entirely new kernel: To test the generalization ability of our model to entirely new kernels on same and different GPUs.

We report the following metrics to assess prediction quality:

- **Mean Absolute Percentage Error (MAPE)** : for overall prediction accuracy, calculated as -

$$\text{MAPE} = \frac{100\%}{n} \sum_{i=1}^{n} \left| \frac{t_{\text{predicted},i} - t_{\text{actual},i}}{t_{\text{actual},i}} \right|$$

  Here, n is the number of predictions. Lower MAPE indicates better accuracy. A MAPE of 20% means predictions are off by 20% on average. MAPE is also sensitive to outliers.

- **Median pred/true ratio** : The median of $\frac{t_{\text{predicted}}}{t_{\text{actual}}}$ across all predictions. This metric indicates systematic bias (calibration):
  - Ratio ≈ 1.0: Well-calibrated (no systematic over/under-prediction)
  - Ratio > 1.0: Systematic over-prediction (model predicts slower runtimes)
  - Ratio < 1.0: Systematic under-prediction (model predicts faster runtimes)

  The median ratio is robust to outliers and reveals whether a model consistently overestimates or underestimates performance.

- **Within X% accuracy**: The percentage of predictions where:

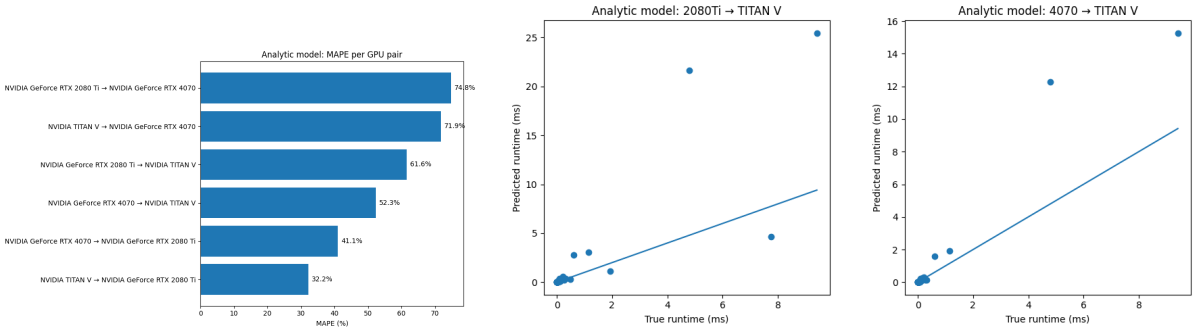
$$\left| \frac{t_{\text{predicted}} - t_{\text{actual}}}{t_{\text{actual}}} \right| \leq \frac{X}{100} \tag{2}$$

We report this for $X = 10\%, 25\%$, and 50%. For example, "Within 25%" means the fraction of predictions that are within $\pm 25\%$ of the true runtime. This metric provides insight into the distribution of errors: a model might have low median error but still produce many large outliers, or vice versa.

Experiment 1: New GPU (Same Kernel and Configuration)

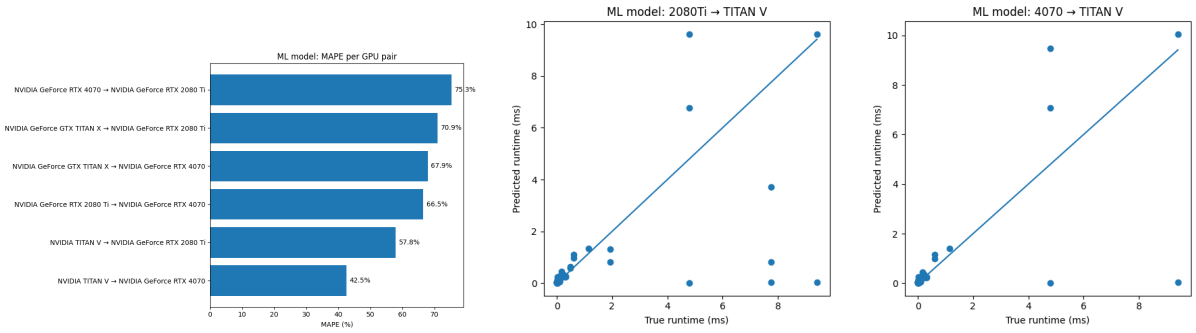
Goal: Predict runtime on a previously unseen GPU, given identical kernels and configurations measured on other GPUs.

The plots in Figs. 1a and 2a indicate the performance of each model when we use calibrations and values from GPU A to predict performance on GPU B. We use all the configurations in 2.



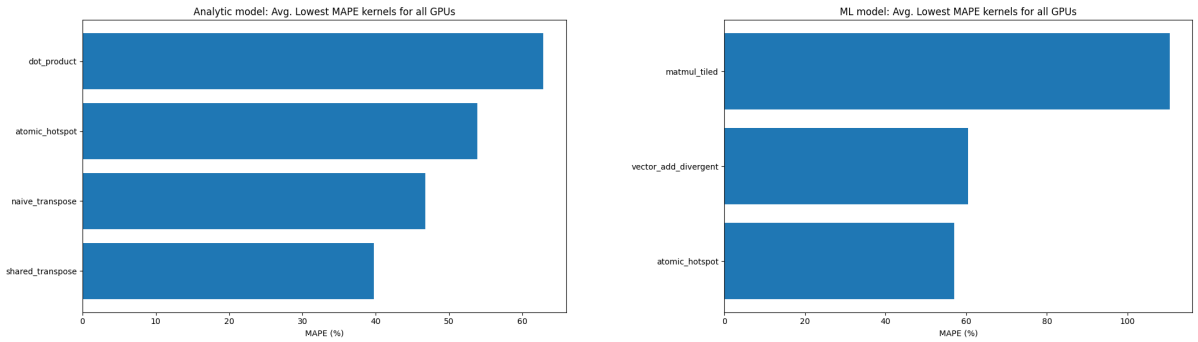
(a) Analytic MAPE per GPU pair (b) 2080Ti → TitanV (c) 4070 → TitanV

Figure 1: Analytical model performance.



(a) ML MAPE per GPU pair (b) 2080Ti → TitanV (c) 4070 → TitanV

Figure 2: ML model performance.



(a) (b)

Figure 3: Kernels with least MAPE error when averaged across GPUs for Analytical and ML model.

For TitanV architecture specifically, metrics for both models are summarized in Table 3.

Table 3: Performance metrics for TitanV architecture for analytic and ML models.

Metric	**Analytic Model**	**ML Model**
MAPE	86.62%	355.97%
Median pred/true	1.03	2.05
Within 10/25/50%	16.3% / 30.37% / 51.11%	6.57% / 19.71% / 27.74%

The analytical model is well-calibrated but has high variance. This is because the roofline model captures the average behavior correctly but misses architecture-specific nuances (cache hierarchies, memory controllers, warp scheduling differences).

The ML model systematically over-predicts runtimes. In the scatterplots, we show how close or far the predicted runtimes are for all 16 kernels on TitanV from 2080Ti and 4070 GPU. This shows that the ML model cannot learn the mapping between hardware features and actual performance. It's essentially guessing based on patterns from other GPUs that don't transfer.

The best performing kernels (with lowest MAPE when averaged across all GPU permutations, not just TitanV) are shown in Figs. 3a and 3b for the analytical model and ML model respectively. Simple, regular memory access patterns (like vector_add, dot_product) work better for physics-based models because they match roofline assumptions. The ML model struggles with everything because it has no architectural grounding.

Experiment 2: New Configurations (Same Kernels and GPUs)

Goal: Predict performance for new problem sizes for known kernels on known GPUs. So it's basically the same kernel, testing on the same GPU, but predicting the runtime when varying the problem size as seen in Table 2. We use the Small, Medium, and Large category to predict XLarge.

The reported MAPE below is an average across all included kernel/config/GPU-pairs, not limited to a single target GPU.

Table 4: Performance metrics for new configurations for analytic and ML models across all GPU/kernel/config pairs.

Metric	**Analytic Model**	**ML Model**
MAPE	871.52%	88.02%
Median pred/true	1.00	0.04
Within 10/25/50%	17.24% / 22.41% / 46.55%	0% / 6.9% / 10.34%

This is the analytic model's weakest setting. Even though it is well-calibrated, the MAPE is catastrophic! Since configuration changes alter memory behavior, occupancy, and caching drastically, the model breaks down entirely.

The ML model performs substantially better than the analytic model in this case, but is still lacking. It is able to learn kernel-specific scaling patterns from FLOPs/BYTES/resources, enabling better generalization across problem sizes. However, it fails in predicting the influence of non-linear effects of cache-thrashing, occupancy limits at SM capacity and memory-bandwidth saturation.

Predicting scaling behavior requires understanding phase transitions (when a kernel switches from compute-bound to memory-bound, or from L1-cached to DRAM-bound). Neither approach captures this well, but ML at least learns partial patterns from training data

Experiment 3: New but Related Kernels

Goal: Predict performance for entirely new kernels based only on runtimes of other kernels on the same GPUs.

We partitioned our 16 kernels into strategically chosen train and test sets:

- Train kernels (12): vector_add, saxpy, strided_copy_8, random_access, reduce_sum, dot_product, histogram, matmul_naive, naive_transpose, conv2d_3x3, conv2d_7x7, shared_bank_conflict
- Test kernels (4): matmul_tiled, shared_transpose, atomic_hotspot, vector_add_divergent

The test kernels were specifically chosen to be related but distinct from kernels in the training set. This design tests whether models can generalize from basic patterns to their optimized or pathological variants.

The analytic model does not require training as such. Since it uses FLOPs, BYTES, occupancy, and GPU characteristics to predict runtimes, calibration is performed on the train kernels to compute the efficiency factor. This factor is then applied to both the train and test kernels. The ML model is trained only on the 12 train kernels and only the Small configuration from Table 2.

Table 5: Experiment 3 Results: Analytic vs ML Model

Model	MAPE (%) on the 4 test kernels
Analytic	9.8
ML Model	85.1

The analytical model performs very well on new kernels. This shows that kernel identity is largely irrelevant to the roofline model as long as FLOPs/BYTES and resource usage are accurate. ML models excel at predicting within the convex hull of training examples, but they fail at predicting beyond training distribution boundaries.

This shows that efficiency factors transfer across architectures because they capture fundamental resource utilization patterns, not implementation-specific details.

Why Random Forest ML Model?

Model	**Exp1**	**Exp2**	**Exp3**
Linear	1390.91	416.62	1253.85
Ridge	1390.21	402.93	1253.37
Lasso	1344.96	425.75	1242.31
SVR (RBF)	**78.01**	88.00	120.62
Random Forest	355.97	88.02	**85.08**
Gradient Boosting (XGBoost)	336.25	**87.74**	107.82
kNN	287.08	88.55	186.38
Best Model per Experiment	SVR-RBF	XGBoost	Random Forest

Table 6: MAPE (%) for all models across the three experiments. Lower is better.

Overall from Table 6, we can see that SVR does the best followed by Random Forest when we take mean MAPE across all 3 experiments. We used Random Forest as the main ML model in our experiments as it is faster than SVR (RBF) to train and test.

7 Conclusion

Our analytical model is able to handle "global" generalization (new architectures, new kernels) and the ML model is able to handle "local" generalization (new sizes of the same kernel) from our 3 experiments. Also one main thing to note is, the same analytical model is being used for all 3 experimental configurations. However, we train 3 different ML models to evaluate each of the 3 experimental configurations we have come up with. So overall, the analytical model is a better choice for us.

References

[1] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65-76, 2009.

[2] C. Yang, T. Kurth, and S. Williams, "Hierarchical roofline analysis for GPUs: Accelerating performance optimization for the NERSC-9 Perlmutter system," *Concurrency and Computation: Practice and Experience*, vol. 32, no. 20, 2020.

[3] N. Ding and S. Williams, "An instruction roofline model for GPUs," in *Proc. IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2019, pp. 7-18.

[4] E. Konstantinidis and Y. Cotronis, "A quantitative roofline model for GPU kernel performance estimation using micro-benchmarks and hardware metric profiling," *Journal of Parallel and Distributed Computing*, vol. 107, pp. 37-56, 2017.

[5] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," in *Proc. 36th Annual International Symposium on Computer Architecture*, pp. 152-163, 2009.

[6] J. Lemeire et al., "Analysis of the analytical performance models for GPUs and extracting the underlying Pipeline model," *Journal of Parallel and Distributed Computing*, vol. 181, 2023.

[7] D. Grewe and M. F. P. O'Boyle, "A static task partitioning approach for heterogeneous systems using OpenCL," in *Proc. 20th International Conference on Compiler Construction (CC)*, pp. 286-305, 2011.

[8] K. Kothapalli et al., "A performance prediction model for the CUDA GPGPU platform," in *Proc. International Conference on High Performance Computing (HiPC)*, 2009.

[9] S. S. Baghsorkhi et al., "An adaptive performance modeling tool for GPU architectures," in *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2010.

[10] G. X. Yu, Y. Gao, P. Golikov, and G. Pekhimenko, "Habitat: A runtime-based computational performance predictor for deep neural network training," in *Proc. USENIX Annual Technical Conference (ATC)*, 2021.

[11] J. Li et al., "Learning-based performance prediction for data-intensive applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 5, 2023.

[12] S. Lee et al., "Forecasting GPU performance for deep learning training and inference: NeuSight and beyond," in *Proc. 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, vol. 1, pp. 493-508, 2024.

[13] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Proc. IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 163-174, 2009.

[14] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-sim: An extensible simulation framework for validated GPU modeling," in *Proc. 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 473-486, 2020.

[15] D. Avalos Baddouh et al., "Sampled simulation of GPU kernels," *IEEE Computer Architecture Letters*, 2021.

[16] N. Ardalani, C. Lestourgeon, K. Sankaralingam, and X. Zhu, "Cross-architecture performance prediction (XAPP) using CPU code to predict GPU performance," in *Proc. 48th International Symposium on Microarchitecture (MICRO)*, pp. 725-737, 2015.

[17] X. Zheng, L. K. John, and A. Gerstlauer, "LACross: Learning-based analytical cross-platform performance and power prediction," *International Journal of Parallel Programming*, vol. 45, pp. 1488-1514, 2017.

[18] X. Qi, J. Chen, and L. Deng, "CP$^{\mathbf{3}}$: Hierarchical cross-platform power/performance prediction using a transfer learning approach," in *Proc. International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*, pp. 119-134, 2023.