Cross-GPU Performance Prediction Using Analytical Models and Machine Learning

Team Members:
Arushi Srivastava - as19341
Debdeep Naha - dn2491
Eva Gupta - eg3900
Shika Rao - sr7463

1 Abstract

Selecting the right GPU for a workload or predicting application performance across platforms requires executing kernels on each target device—a process that is expensive (cloud GPU costs), time-consuming (hours for comprehensive evaluation), and impossible when hardware is unavailable or unreleased. Through this project, we want to attempt to solve the cross-GPU performance prediction problem: given kernel measurements on GPU-A, predict runtime on GPU-B using only device specifications, without executing on the target platform.

We present 2 approaches: analytical performance modeling and machine learning techniques to enable zero-shot performance prediction across GPU architectures. Our methods leverage hardware specifications, kernel characteristics, and performance patterns observed on a source GPU to predict execution time on unseen target GPUs. We evaluate our approach on a comprehensive benchmark suite of 16 diverse CUDA kernels spanning memory-bound, compute-bound, and mixed workloads across three GPU architectures: NVIDIA TITAN V (Volta), GeForce RTX 2080 Ti (Turing), and GeForce RTX 4070 (Ada Lovelace).

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

2 Introduction

2.1 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:

- **Cloud GPU costs** make extensive benchmarking prohibitively expensive ($1-5/hour for high-end GPUs)
- **Hardware availability** limits access to latest architectures during development cycles
- **Design space exploration** requires evaluating hundreds of configurations across multiple platforms
- **Procurement decisions** need performance forecasts before hardware purchase

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

2.2 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.

2.3 Approach Overview

Our solution combines three key components:

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

2.4 Contributions

- A comprehensive benchmark suite of 16 diverse CUDA kernels with 71 configurations per GPU, totaling over 200 data points per architecture
- An extended analytical model that accounts for architectural differences in memory systems, compute units, and scheduling policies
- Empirical validation across three generations of NVIDIA GPUs (Volta, Turing, Ada Lovelace)
- Open-source implementation and dataset for reproducibility

2.5 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 analytical models and ML approaches. Section 7 details the experimental setup and benchmark kernels. Section **??** presents results and analysis. Section 8 concludes with future directions.

3 Literature Survey

3.1 Analytical Performance Models

3.1.1 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:

$$\text{Performance} = \min\left(\text{Peak Compute}, \text{Bandwidth} \times \text{Arithmetic Intensity}\right) \tag{1}$$

where Arithmetic Intensity is defined as:

$$\text{Arithmetic Intensity} = \frac{\text{FLOPs}}{\text{Bytes Transferred}} \tag{2}$$

Strengths:

- Provides intuitive upper bounds on achievable performance
- Identifies whether kernels are compute-bound or memory-bound
- Requires only peak specifications (no execution needed)

Limitations:

- Assumes perfect utilization and no overhead
- Does not capture cache effects or memory hierarchy
- Single-level model insufficient for modern GPUs

3.1.2 Hierarchical Roofline Extensions

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
- 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 microbenchmarks to calibrate actual achievable bandwidths rather than theoretical peaks, improving prediction accuracy.

Key Insight: Hierarchical models improve fidelity but still require profiling or microbenchmarks on each target GPU, limiting their zero-shot transfer capability.

3.1.3 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:

$$T_{\text{kernel}} = \max\left(T_{\text{compute}}, T_{\text{memory}}\right) \tag{3}$$

Lemeire et al. [6] extracted and analyzed underlying pipeline models from various GPU analytical approaches, identifying common patterns and limitations.

Limitations: These models require detailed microarchitectural knowledge and often target-specific profiling data.

3.2 Machine Learning Approaches

3.2.1 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. Their approach:

- Extracted features from OpenCL source code
- Used decision trees for classification
- Achieved 77-90% accuracy for partitioning decisions

Kothapalli et al. [8] and Baghsorkhi et al. [9] used regression models on dynamic profile data to predict GPU acceleration potential.

Key Limitation: These approaches required extensive training data for each target architecture, lacking cross-platform generalization.

3.2.2 Deep Learning Models

Habitat (Yu et al., 2021) [10]: Developed multi-layer perceptrons for deep learning workload prediction using:

- Hardware features (SM count, memory bandwidth, etc.)
- Execution patterns (kernel launch frequencies, tensor sizes)
- Runtime statistics

Results: 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.

Critical Observation from NeuSight (2024): Lee et al. [12] 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.

3.3 Simulation-Based Methods

3.3.1 GPGPU-Sim

Bakhoda et al. [13] introduced GPGPU-Sim, a cycle-accurate GPU simulator that models:

- Warp scheduling and divergence
- Memory coalescing
- Cache behavior
- Interconnect contention

Accuracy: Achieved 5-10% prediction error for older GPU architectures.

3.3.2 Accel-Sim

Khairy et al. [14] modernized simulation with Accel-Sim, adding support for Volta/Ampere architectures and improving PTX-level fidelity.

3.3.3 Sampled Simulation

Avalos Baddouh et al. [15] developed sampled-simulation techniques that simulate only representative kernel slices, reducing runtime from days to hours.

Fundamental Limitation: Even optimized simulators require hours for realistic workloads because every instruction is modeled cycle-by-cycle. While simulation provides highest accuracy before hardware availability, its extreme computational cost makes it unsuitable for rapid design-space exploration or predicting performance for many kernels.

3.4 Hybrid Approaches

3.4.1 XAPP

Ardalani et al. [16] developed XAPP (Cross-Architecture Performance Prediction) using:

- CPU code features as input
- Two-level ML pipeline
- Transfer to GPU performance

Limitation: Limited to CPU$\rightarrow$GPU transfer, not GPU$\rightarrow$GPU.

3.4.2 LACross and CP^3

Zheng et al. [17] (LACross) and Qi et al. [18] (CP^3) 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.

3.4.3 NeuSight

Lee et al. [12] presented the most advanced hybrid method, NeuSight, which:

1. Decomposes workloads into tiles
2. Predicts per-tile utilization with ML
3. Bounds final latency using GPU performance laws

Results: Achieved state-of-the-art accuracy (2-15%) on unseen GPUs.
Limitation: Applies mainly to deep learning operators with known execution structure, not general CUDA kernels.

3.5 Research Gap

Hybrid methods offer the best balance of accuracy and generality, but existing work still lacks a **true zero-shot approach for general CUDA kernels**. Our work addresses this gap by:

- Developing architecture-aware analytical models that transfer without target execution
- Creating comprehensive kernel characterizations that capture performance-critical features
- Validating on diverse workload types beyond deep learning

4 Proposed Methodology

4.1 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.

4.1.1 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
- **Memory:** How fast data can be moved to/from DRAM

The Model:

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

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.

4.1.2 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, hence, 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.

5 Kernel Efficiency

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, including:

- Instruction mix and scheduling: Pipeline utilization, instruction-level parallelism, and warp scheduling efficiency
- Memory access patterns: Cache hit rates, coalescing effectiveness, and bank conflict avoidance
- Control flow: Branch prediction accuracy and warp divergence penalties
- Latency hiding: Effectiveness of interleaving computation with memory operations

The critical assumption enabling cross-GPU prediction is that kernel efficiency remains relatively stable across GPU architectures.

5.1 The Cross-GPU Prediction Model

5.1.1 Core Prediction Formula

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

$$T_{\text{tgt}} = T_{\text{src}} \times \frac{\text{Roof}_{\text{src}}}{\text{Roof}_{\text{tgt}}} \times \frac{O_{\text{src}}}{O_{\text{tgt}}} \times \underbrace{\frac{\eta_{\text{src}}}{\eta_{\text{tgt}}}}_{\approx 1} \tag{4}$$

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

1. **Roofline ratio:** Accounts for different peak capabilities
2. **Occupancy ratio:** Accounts for different resource utilization
3. **Efficiency ratio $\approx$ 1:** Same kernel characteristics (memory patterns, branch behavior) produce similar efficiency

We compute efficiency on the source GPU:

$$\eta_{\text{src}} = \frac{\text{Measured Performance}}{O_{\text{src}} \times \text{Roof}_{\text{src}}} \tag{5}$$

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

5.2 Handling Different Kernel Types

5.2.1 Case 1: Pure Memory Operations (FLOPs = 0)

Examples: vector_add, naive_transpose, strided_copy
These kernels are purely memory-bound – compute capability is irrelevant.
Prediction:

$$BW_{\text{measured}} = \frac{\text{Bytes}}{T_{\text{src}}} \tag{6}$$

$$\eta_{BW} = \frac{BW_{\text{measured}}}{O_{\text{src}} \times BW_{\text{peak,src}}} \tag{7}$$

$$T_{\text{tgt}} = \frac{\text{Bytes}}{\eta_{BW} \times O_{\text{tgt}} \times BW_{\text{peak,tgt}}} \tag{8}$$

Reasoning: For memory-only operations, we model bandwidth efficiency. This captures memory access pattern quality (coalesced vs. strided vs. random).

5.2.2 Case 2: Compute Operations (FLOPs > 0)

Examples: matmul, conv2d, saxpy, dot_product
These kernels have arithmetic operations. The Roofline determines if they're compute- or memory-bound.

Prediction:

$$\text{Roof} = \min(\text{Peak Compute}, \text{AI} \times \text{Peak BW}) \tag{9}$$

$$P_{\text{measured}} = \frac{\text{FLOPs}}{T_{\text{src}}} \tag{10}$$

$$\eta = \frac{P_{\text{measured}}}{O_{\text{src}} \times \text{Roof}_{\text{src}}} \tag{11}$$

$$T_{\text{tgt}} = \frac{\text{FLOPs}}{\eta \times O_{\text{tgt}} \times \text{Roof}_{\text{tgt}}} \tag{12}$$

Reasoning: The min() in the Roofline automatically identifies the bottleneck. This can even change across GPUs – a kernel might be compute-bound on an older GPU but memory-bound on a newer GPU with higher compute capability.

5.3 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)
 1: // Compute occupancy on both GPUs
 2: $O_{src} \leftarrow$ ComputeOccupancy(kernel, src_resources)
 3: $O_{tgt} \leftarrow$ ComputeOccupancy(kernel, tgt_resources)
 4: if FLOPs = 0 then                                  ▷ Pure memory case
 5:     $BW_{measured} \leftarrow$ Bytes/$T_{src}$
 6:     $\eta \leftarrow BW_{measured}/(O_{src} \times BW_{src})$
 7:     $T_{tgt} \leftarrow$ Bytes/$(\eta \times O_{tgt} \times BW_{tgt})$
 8: else                                               ▷ Compute or mixed case
 9:     AI $\leftarrow$ FLOPs/Bytes
10:     $\text{Roof}_{src} \leftarrow \min(C_{src}, \text{AI} \times BW_{src})$
11:     $\text{Roof}_{tgt} \leftarrow \min(C_{tgt}, \text{AI} \times BW_{tgt})$
12:     $P_{measured} \leftarrow$ FLOPs/$T_{src}$
13:     $\eta \leftarrow P_{measured}/(O_{src} \times \text{Roof}_{src})$
14:     $T_{tgt} \leftarrow$ FLOPs/$(\eta \times O_{tgt} \times \text{Roof}_{tgt})$
15: end if
16: return $T_{tgt}$
```

Why this works:

- Hardware differences captured by roofline and occupancy
- Kernel characteristics captured by efficiency factor
- Separation of concerns enables accurate cross-GPU transfer

5.4 Model Calibration Strategy

Peak vs. Sustained Performance: GPU datasheets list theoretical peaks, but real applications achieve 85-95% due to memory controller overhead, scheduling inefficiencies, and power limitations.

Our approach: Use *sustained* (measured) performance from micro-benchmarks:

- Sustained bandwidth from optimized memory copy
- Sustained compute from optimized matrix multiplication

Reasoning: This accounts for inherent hardware limitations that affect all kernels equally, improving prediction accuracy.

5.5 Key Assumptions and Validity

Our model assumes:

1. **Efficiency transfers ($\eta_{\mathbf{src}} \approx \eta_{\mathbf{tgt}}$)**

   Valid when:

   - Same memory access patterns
   - Similar cache behavior (relative to each GPU's capacity)
   - No architecture-specific optimizations (e.g., Tensor Cores)

   Breaks when:

   - Workload fits in cache on one GPU but not another
   - Specialized hardware units used (RT cores, Tensor cores)
   - Different memory coalescing behavior

2. **Occupancy captures parallelism**

   Valid when:

   - Static resource allocation
   - Regular workload distribution
   - Steady-state dominates runtime

   Breaks when:

   - Severe load imbalance
   - Dynamic parallelism
   - Tail effects dominant (tiny final batch)

3. **Roofline captures bottleneck**

   Valid when:

   - Kernel clearly compute- or memory-bound
   - Minimal compute-memory overlap

- Steady-state behavior

Breaks when:

- Complex instruction mix
- Significant pipeline overlap
- Launch overhead dominates (very short kernels)

These assumptions are validated empirically in Section **??**.

6 Experimental Setup

6.1 Hardware Platforms

We conducted experiments on three NVIDIA GPU architectures representing different generations and market segments:

Table 1: GPU Hardware Specifications

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

6.2 Benchmark Kernels

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

6.2.1 Memory-Bound Kernels

1. **vector_add**: Element-wise vector addition (coalesced access)
2. **saxpy**: Scalar-alpha X plus Y operation
3. **strided_copy_8**: Strided memory access (stride=8)
4. **naive_transpose**: Matrix transpose without shared memory
5. **shared_transpose**: Optimized transpose with tiling
6. **random_access**: Random gather/scatter pattern

6.2.2 Compute-Bound Kernels

7. **matmul_naive**: Naive matrix multiplication
8. **matmul_tiled**: Tiled matrix multiplication with shared memory
9. **conv2d_3x3**: 3×3 convolution operation
10. **conv2d_7x7**: 7×7 convolution operation

6.2.3 Reduction Operations

11. **reduce_sum**: Sum reduction with shared memory
12. **dot_product**: Vector dot product

6.2.4 Atomic Operations

13. **histogram**: 256-bin histogram computation
14. **atomic_hotspot**: Contention on single atomic counter

6.2.5 Special Cases

15. **vector_add_divergent**: Vector add with branch divergence
16. **shared_bank_conflict**: Demonstration of bank conflicts

6.3 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)

6.4 Measurement Methodology

6.4.1 Execution Protocol

For each kernel configuration:

- **Warmup:** 10-20 iterations (kernel-dependent)
- **Measurement:** 50-100 iterations (kernel-dependent)

- **Trials:** 10 independent runs
- **Metric:** Mean execution time (ms) and standard deviation

6.4.2 Timing Instrumentation

We used CUDA events for precise kernel timing:

Listing 1: Timing Code Example

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start);
kernel<<<grid, block>>>(args);
cudaEventRecord(stop);

cudaEventSynchronize(stop);
float milliseconds = 0;
cudaEventElapsedTime(&milliseconds, start, stop);
```

6.5 Feature Extraction

For each kernel configuration, we extracted the following features:

6.5.1 Kernel Characteristics

- FLOPs (floating-point operations)
- Memory bytes transferred
- Arithmetic intensity (FLOPs/Byte)
- Working set size
- Shared memory usage
- Register usage per thread

6.5.2 Memory Access Patterns

- Coalesced vs. uncoalesced
- Stride information
- Random access indicators
- Bank conflict potential

6.5.3 Control Flow

- Branch divergence flags
- Atomic operation counts
- Synchronization points

6.5.4 Occupancy Metrics

- Theoretical occupancy
- Blocks per SM
- Threads per block
- Grid dimensions

6.6 Dataset Statistics

Table 3: Benchmark Dataset Summary

GPU	**Configurations**	**Total Runs**	**Kernel Launches**
TITAN V	71	710	85,200
RTX 2080 Ti	71	710	85,200
RTX 4070	71	710	85,200
Total	**213**	**2,130**	**255,600**

7 Experimental Setup

7.1 Hardware Platforms

We conducted experiments on three NVIDIA GPU architectures representing different generations and market segments:

Table 4: GPU Hardware Specifications

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

7.2 Benchmark Kernels

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

7.2.1 Memory-Bound Kernels

1. **vector_add**: Element-wise vector addition (coalesced access)
2. **saxpy**: Scalar-alpha X plus Y operation
3. **strided_copy_8**: Strided memory access (stride=8)
4. **naive_transpose**: Matrix transpose without shared memory
5. **shared_transpose**: Optimized transpose with tiling
6. **random_access**: Random gather/scatter pattern

7.2.2 Compute-Bound Kernels

7. **matmul_naive**: Naive matrix multiplication
8. **matmul_tiled**: Tiled matrix multiplication with shared memory
9. **conv2d_3x3**: 3×3 convolution operation
10. **conv2d_7x7**: 7×7 convolution operation

7.2.3 Reduction Operations

11. **reduce_sum**: Sum reduction with shared memory
12. **dot_product**: Vector dot product

7.2.4 Atomic Operations

13. **histogram**: 256-bin histogram computation
14. **atomic_hotspot**: Contention on single atomic counter

7.2.5 Special Cases

15. **vector_add_divergent**: Vector add with branch divergence
16. **shared_bank_conflict**: Demonstration of bank conflicts

7.3 Problem Sizes and Configurations

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

Table 5: 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)

7.4 Measurement Methodology

7.4.1 Execution Protocol

For each kernel configuration:

- **Warmup:** 10-20 iterations (kernel-dependent)
- **Measurement:** 50-100 iterations (kernel-dependent)

- **Trials:** 10 independent runs
- **Metric:** Mean execution time (ms) and standard deviation

7.4.2 Timing Instrumentation

We used CUDA events for precise kernel timing:

Listing 2: Timing Code Example

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start);
kernel<<<grid, block>>>(args);
cudaEventRecord(stop);

cudaEventSynchronize(stop);
float milliseconds = 0;
cudaEventElapsedTime(&milliseconds, start, stop);
```

7.5 Feature Extraction

For each kernel configuration, we extracted the following features:

7.5.1 Kernel Characteristics

- FLOPs (floating-point operations)
- Memory bytes transferred
- Arithmetic intensity (FLOPs/Byte)
- Working set size
- Shared memory usage
- Register usage per thread

7.5.2 Memory Access Patterns

- Coalesced vs. uncoalesced
- Stride information
- Random access indicators
- Bank conflict potential

7.5.3 Control Flow

- Branch divergence flags
- Atomic operation counts
- Synchronization points

7.5.4 Occupancy Metrics

- Theoretical occupancy
- Blocks per SM
- Threads per block
- Grid dimensions

7.6 Dataset Statistics

Table 6: Benchmark Dataset Summary

GPU	Configurations	Total Runs	Kernel Launches
TITAN V	71	710	85,200
RTX 2080 Ti	71	710	85,200
RTX 4070	71	710	85,200
Total	**213**	**2,130**	**255,600**

8 Conclusions and Future Work

8.1 Summary of Contributions

[Space for summary]

8.2 Key Findings

[Space for key findings]

8.3 Limitations

[Space for limitations]

8.4 Future Directions

[Space for future work]

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^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.

A Kernel Implementation Details

[Space for kernel code snippets and implementation notes]

B Complete Benchmark Results

[Space for detailed performance tables]

C Statistical Analysis

[Space for additional statistical analysis]