Predicting the performance of a kernel

Ann Xiao,[1,*] Xinyu Li,[2,*] and Yidan Qin[3,*]

[*]*New York University, Courant*
[1]*yx3493@nyu.edu*
[2]*xl5280@nyu.edu*
[3]*yq2421@nyu.edu*

Abstract: Optimizing the performance of CUDA kernels is crucial for improving the efficiency of GPU-based applications, particularly for memory-bound operations. An accurate simulation method to predict the kernel speedup can save significant computational resources and lead the way for performance optimization. In this work, we propose a machine learning-based framework that utilizes static profiling metrics, such as the Compute-to-Global-Memory-Access (CGMA) ratio and shared memory utilization, combined with static analysis to predict kernel performance. With our Random Forest regression model, We explain over 92% of the variance in kernel speedup, achieving a Mean Absolute Percentage Error (MAPE) of 55.15%. Feature importance analysis highlights CGMA as the most impactful metric for predicting performance, while metrics like occupancy showed minimal influence. Our findings highlight the practicality and scalability of machine learning for GPU performance prediction, offering insights for kernel optimization.

1. Introduction

CUDA kernel performance prediction is a critical area in GPU computing, enabling developers to optimize workloads efficiently without incurring significant computational overhead. Accurate performance estimation is essential for reducing the iterative profiling and tuning cycles required for kernel optimization, especially for memory-bound operations that dominate many real-world applications. Existing methods, such as static analysis [1], dynamic profiling [2], and machine learning-based approaches [3], have addressed challenges such as warp divergence, memory-computation overlap, and execution time prediction. While static analysis excels at providing worst-case bounds, it often lacks the flexibility to adapt to diverse workloads. Dynamic profiling, on the other hand, can provide accurate runtime information but requires kernel execution, leading to significant overhead. Machine learning approaches, while accurate, often rely heavily on runtime profiling data and large, diverse training datasets, which can limit their scalability and generalizability.

In this project, we address the challenging task of predicting the speedup of memory-bound CUDA kernels relative to single-thread performance, without requiring kernel execution. This problem is particularly important because it provides a practical means to assess kernel performance in scenarios where running the kernel on various configurations is computationally prohibitive. Our approach utilizes a combination of static analysis and runtime profiling to identify critical metrics that influence kernel performance. Specifically, we designed 11 distinct memory-bound kernels to capture diverse performance patterns, including scenarios involving shared memory usage and global memory access.

From these kernels, we identified seven critical metrics—problem size, grid configuration, occupancy, compute-to-global memory access ratio (CGMA), global memory access reduction per thread, branch divergence, and percentage of shared memory utilization—that are instrumental in determining performance. These metrics were used to create a dataset of nearly 700 samples, collected through experimentation on the CIMS CUDA2 GPU platform with an NVIDIA GeForce RTX 2080 Ti. This dataset was then used to train a Random Forest regression model, chosen for its robustness and ability to handle nonlinear interactions between features.

Our proposed approach combines the efficiency of static analysis with the adaptability of machine learning to predict kernel speedup in memory-bound scenarios. By focusing on shared memory usage, global memory access, and critical performance metrics, our framework provides a scalable and practical solution for performance prediction, significantly reducing the reliance on runtime profiling. Furthermore, the results demonstrate the effectiveness of our methodology in delivering accurate predictions while providing interpretable insights into the factors that impact kernel performance.

2. Literature Survey

CUDA kernel performance prediction is a critical area of research, with significant implications for optimizing GPU workloads. Existing approaches to address this problem can broadly be categorized into static analysis, dynamic profiling, and machine learning-based methods, each offering unique advantages and limitations.

2.1. Static Analysis Methods

Static analysis methods aim to provide worst-case or upper-bound estimates of kernel performance without executing the kernel. Recent advances include methods like the Many-BSP analytical model, which achieves a 12.33% prediction error through static analysis [4], compute-memory bound performance estimation across devices [5], and static analysis-guided optimization frameworks that predict execution costs for informed kernel improvements [6].

Muller et al. [1] utilize a static analysis framework to identify performance bottlenecks such as warp divergence and memory access conflicts. While effective at estimating resource requirements, this method lacks predictive modeling capabilities, limiting its utility for nuanced performance forecasting.

Other studies have explored static analysis models for execution time prediction. For instance, one such model uses PTX code and GPU scheduling simulation to achieve accurate predictions for compute-intensive kernels, though it struggles with memory-bound workloads due to limited memory hierarchy modeling [7]. Hong and Kim [8] propose an analytical model that incorporates memory-level and thread-level parallelism to estimate kernel performance. This approach is valuable for identifying bottlenecks but struggles with accurately modeling memory-bound workloads due to simplified assumptions about memory hierarchies. Similarly, the Many-BSP model [4] achieves a prediction error of 12.33% using static analysis but requires manual tuning for diverse workloads.

2.2. Dynamic profiling Approaches

Dynamic profiling methods utilize runtime information to analyze kernel behavior. GPUPerf, introduced by Jaewoong et al. [2], combines static and dynamic profiling to identify bottlenecks and assess memory-computation overlap. While the improved MWP-CWP model provides interpretable metrics for optimization, its reliance on runtime profiling introduces overhead and limits its applicability to real-time scenarios.

Other dynamic profiling frameworks, such as those proposed by Wang et al. [9], use runtime data to balance performance and energy efficiency through techniques like dynamic voltage and frequency scaling (DVFS). However, these methods require extensive profiling data, which can be computationally expensive to collect for large-scale applications.

2.3. Machine Learning Approaches

Machine learning-based methods have gained popularity for their ability to model complex, non-linear relationships in GPU performance. Tiwari et al. [3] utilize supervised learning to predict execution time, leveraging runtime profiling data to train the model. While effective, these methods often require large, diverse datasets and may struggle to generalize across architectures.

Hybrid approaches that combine analytical modeling with machine learning, such as those by Meng et al. [10], have demonstrated improved prediction accuracy for diverse workloads. These methods incorporate architectural details into the model but require careful feature selection and tuning, increasing development complexity.

2.4. Comparison and Limitations

The existing literature demonstrates significant progress in CUDA kernel performance prediction but highlights trade-offs between prediction accuracy, computational overhead, and generalizability. Static analysis excels at providing low-overhead, architecture-independent estimates but lacks adaptability to diverse workloads. Dynamic profiling delivers high accuracy but incurs runtime costs and is not scalable for real-time prediction. Machine learning methods offer flexibility and accuracy but depend on large, representative datasets and careful feature engineering.

2.5. Motivation for This Work

Despite these advances, a gap remains in developing a framework that balances efficiency, scalability, and accuracy for memory-bound kernels. Our work addresses this gap by integrating static analysis and machine learning, employing critical metrics such as CGMA, branch divergence, and shared memory utilization. By focusing on memory-bound workloads and minimizing profiling requirements, we aim to offer a scalable and practical solution that bridges the strengths of existing methods.

3. Proposed Idea

Our approach combines GPU-friendly kernel design with critical performance metrics to develop a predictive model for CUDA kernel speedup. We designed 11 kernels, identified 7 key metrics, and used a Random Forest model for accurate regression. Below are the details for microbenchmarks, metric definitions, and machine learning implementation.

3.1. Microbenchmarks

We implemented 11 kernels as microbenchmarks to simulate different GPU workloads, representing computational and memory access patterns. They are extracted from various real-time scenarios and are all GPU-friendly. These benchmarks demonstrate various challenges and opportunities in GPU programming.

Elementwise-Add and Elementwise-Multiply These 2 kernels are the fundamental parts in many algorithms, they are easy to implement and with scalability to add more blocks to fit for larger problem size. Elementwise multiply is a nature progression from elementwise addition, emphasizing the arithmetic operations.

Heat Diffusion/Distribution Modeling scientific simulations, this kernel evaluates memory-bound performance by computing values based on neighboring elements. It highlights the challenges of high memory access and computational dependencies.

Count Number The histogram computation involves counting the occurrences of each element in a given dataset. With the implementation on GPU, the access pattern of parallel updates to the output array can lead to a large ratio of memory conflicts, so we used atomicAdd to ensure correctness updates to global memory. While this eliminates race conditions, it introduces fully sequential updates when accessing the global memory.

Count numbers with shared memory To optimize the count number application, we utilized shared memory as an intermediate buffer to reduce global memory latency. Each thread updates shared memory using atomicAdd, and the results are then aggregated into global memory. Although there is also a partially sequential access pattern, it significantly reduced contention and improved performance compared to the previous implementation.

2D Convolutional Layer As a fundamental component of convolutional neural networks (CNNs), 2D convolutional layer is widely used in tasks like image recognition, segmentation, and object detection. This kernel calculates each output element as a weighted sum of overlapping input elements covered by a filter. While the current implementation accesses global memory directly, introducing redundant reads, future optimizations could use shared memory to cache overlapping regions, reducing latency. Despite lacking shared memory optimization, the kernel remains computationally demanding, making it an effective GPU performance benchmark.

Activation Layer (Sigmoid Function) The sigmoid function introduces non-linearity in neural networks, making it compute-intensive. This kernel demonstrates efficient thread-level parallelism while maintaining high computational demands.

Maximum Distance This kernel is inspired by real-world applications like computational geometry and spatial analysis in urban planning. It calculates the maximum distance from each point in the input array to predefined "landmark" positions. Each thread independently calculates and updates the maximum distance without requiring synchronization. Parallelism is achieved using grid-stride loops.

Array Reversal Simplified from matrix transposition, this kernel performs in-place 1D array reversal. Each thread processes elements independently, ensuring conflict-free memory updates and scalability.

Softmax Normalization Softmax normalization converts raw scores into probabilities. The implementation uses shared memory for intermediate sums, reducing global memory latency and improving performance for larger datasets.

Binary Search This kernel employs a fundamental algorithm with $O(\log n)$ complexity to efficiently locate a target value in a sorted array. Our implementation dynamically allocated shared memory for intermediate results. Each thread in the multi-threaded setup searches a segment of the array independently, followed by a reduction step to consolidate results.

3.2. Metrics

We identified 7 key metrics that capture performance characteristics, enabling comprehensive understanding and accurate prediction. Below, we provide calculation formulas and implementation details for each metric.

3.2.1. Kernel Input Parameters

Three Parameters:

Problem Size. 10000, 50000, 100000, 500000, 1000000, 5000000, 10000000

Number of Blocks. Calculated based on specific problem size and number of threads per block

Number of Threads per Block. 32, 64, 128, 200, 256, 400, 500, 512, 1024

3.2.2. Maximum Percentage of Blocks

Definition: Reflects the fraction of total blocks that can run concurrently, measuring GPU resource utilization.

Calculation Formula.

$$\text{Max Percentage of Blocks} = \frac{\text{Concurrent Blocks}}{\text{Total Blocks}} \times 100$$

Where:

$$\text{Concurrent Blocks} = \text{Blocks per SM} \times \text{Number of SMs}$$

$$\text{Blocks per SM} = \min\left(\frac{\text{Total Registers}}{\text{Registers per Block}}, \frac{\text{Total Shared Memory}}{\text{Shared Memory per Block}}, \frac{\text{Maximum Threads per SM}}{\text{Threads per Block}}\right)$$

Implementation. The custom script `maxBlock_occupancy.py` automates this calculation, processing kernel configurations and GPU constraints.

3.2.3. Occupancy

Definition: Measures GPU efficiency by calculating the fraction of active warps per SM.

Calculation Formula.

$$\text{Occupancy} = \frac{\text{Active Warps per SM}}{\text{Maximum Warps per SM}}$$

Where:

$$\text{Active Warps per SM} = \text{Blocks per SM} \times \text{Warps per Block}$$

$$\text{Blocks per SM} = \min\left(\frac{\text{Total Registers}}{\text{Registers per Block}}, \frac{\text{Total Shared Memory}}{\text{Shared Memory per Block}}, \frac{\text{Maximum Threads per SM}}{\text{Threads per Block}}\right)$$

Implementation. The script `maxBlock_occupancy.py` computes occupancy values based on kernel configurations and hardware constraints.

3.2.4. Branch Divergence

Definition: Quantifies serialization within warps caused by divergent thread execution paths.

Calculation Formula.

$$\text{Divergence Percentage} = \frac{\text{Divergent Executions}}{\text{Total Executions}} \times 100$$

Where:

- **Divergent Executions:** Difference in iterations among threads in the same warp.
- **Total Executions:** Total iterations executed by all threads in the warp.

Implementation. The script `divergence.py` simulates warp execution, identifying divergence by calculating maximum and minimum iterations within each warp.

3.2.5. CGMA (Compute-to-Global Memory Access)

Definition: Measures computational intensity relative to memory access.

Calculation Formula.

$$\text{CGMA} = \frac{\text{Number of Computations}}{\text{Number of Global Memory Accesses}}$$

Implementation. The script `autoprofiler.py` uses NVIDIA Nsight Compute (`ncu`) to extract metrics like `inst_executed` and `dram_bytes`.

3.2.6. Shared Memory Utilization

Definition: Evaluates shared memory usage and its impact on performance.

Calculation Formula.

$$\text{Shared Memory Utilization} = \frac{1024 \times \text{Number of Blocks}}{\text{Problem Size}}$$

Implementation. Metrics were calculated using kernel input parameters for each experiment.

3.2.7. Global Memory Access

Definition: Measures efficiency of global memory usage under multi-threaded execution.

Calculation Formula.

$$\text{Reduction} = (\text{dram\_read} + \text{dram\_write})_{\text{one\_thread}} - \frac{(\text{dram\_read} + \text{dram\_write})_{\text{multi\_thread}}}{\text{Number of Blocks} \times \text{Threads per Block}}$$

Implementation. Metrics `dram__bytes_read.sum` and `dram__bytes_write.sum` were collected using `ncu` for different execution scenarios.

3.3. Machine Learning Model

3.3.1. Model Selection

Our task is to address a regression problem by constructing a predictive model that maps the selected metrics to the corresponding speedup values and also provides some insights for the inner relationships across different metrics. Employing a machine learning model removes the need for manual formula derivation and enables efficient and accurate performance estimation.

Some simple regression model, such as linear and ridge regression model assume linear dependencies and treat metrics independently, making them less suitable for capturing the complexity of CUDA kernel performance.

We employed Random Forest for its ability to capture non-linear dependencies and inter-metric relationships. Hyperparameter tuning was performed using Optuna, optimizing for Mean Absolute Percentage Error (MAPE).

3.3.2. Implementation

The Random Forest model was trained on a dataset of 639 samples, extracted from kernel executions on the NVIDIA GeForce RTX 2080 Ti. The final model achieved high accuracy, with metrics indicating strong performance prediction capabilities.

4. Experimental Setup

All experiments were conducted on a single GPU environment.

4.1. Microbenchmarks

The microbenchmarks were grouped in a single `run.cu` file, which contains the entry of the comparison functions and definitions of all kernels. To reproduce the experiments, follow the steps outlined in `readme.md`, where detailed instructions are provided for compiling the source code, running the executable files, and collecting results.

4.2. Data Collection

Speedup values for each kernel configuration were directly output during execution. Additional metrics, such as memory access patterns and performance profiles, were collected using NVIDIA Nsight Compute (`ncu`) and custom scripts. These metrics were recorded across varying problem sizes and gird configurations to build a comprehensive dataset. Manual validation was conducted to ensure accuracy in the reported results.

4.3. Scripts and Tools

Several script files were written to automate metric collection, reducing human error and ensuring consistency. Table 1 summarizes the methods and tools used to calculate each performance metric.

Table 1. Methods for Calculating Metrics

Metric	Calculation Method
Occupancy	Computed with the command `python3 maxBlock_occupancy.py`.
Branch Divergence	Simulated and calculated with the command `python3 divergence.py`.
CGMA (Compute-to-Global Memory Access)	Computed with the command `python3 autoprofiler.py`.
Shared Memory Utilization	Calculated using kernel input parameters for each experiment.
Maximum Percentage of Blocks	Derived using `maxBlock_occupancy.py` with appropriate kernel configurations.
Global Memory Access	Retrieved via `ncu`. Data collected and calculated manually.
Problem Size, Number of Blocks and Number of Threads Per Block	Defined explicitly in the kernel inputs to cover varying thread and block configurations.

4.4. Hardware Specifications

- **GPU Model:** NVIDIA GeForce RTX 2080 Ti
- **CUDA Version:** 12.4
- **System Environment:** CIMS CUDA2 Server

5. Results and Analysis

5.1. Measures of Success

The primary measures of success for this project are:

- **Prediction Accuracy:** Measured using Mean Absolute Percentage Error (MAPE), Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), and R^2 scores. A lower MAPE and higher R^2 indicate better predictive performance.
- **Feature Interpretability:** Understanding which features contribute most to kernel speedup prediction using feature importance analysis.
- **Scalability:** The ability of the model to handle diverse kernels with varying configurations and problem sizes.

We expect the model to achieve a MAPE of less than 60% and an R^2 score above 0.90, indicating a high degree of prediction accuracy.

5.2. Experimental Procedure

The experiments were conducted on an NVIDIA GeForce RTX 2080 Ti GPU. The collected dataset of nearly 700 samples was then preprocessed by:

- Removing rows with missing values.
- Transforming numerical columns with non-standard formats (e.g., commas) into usable numeric data.
- Splitting the data into an 80:20 training and testing set.

A RandomForestRegressor model was trained using Optuna for hyperparameter optimization. 50 trials were conducted. The evaluation metrics were calculated on the test set for the best-performing model.

5.3. Results and Key Metrics

Table 2 summarizes the key metrics achieved by the best model.

Table 2. Performance Metrics for the Best Model

Metric	**Value**
Mean Absolute Percentage Error (MAPE)	55.15%
Mean Absolute Error (MAE)	442.78
Root Mean Squared Error (RMSE)	3767.63
R^2 Score	0.9287
Explained Variance Score (EVS)	0.9291

The results demonstrate that the model achieves high accuracy, with R^2 and EVS values exceeding 0.92, indicating that over 92% of the variance in kernel speedup is explained by the model.

5.4. Feature Importance

Feature importance analysis reveals which metrics contribute most to the model's predictions. Table 3 lists the importance of each feature.

As shown in Table 3, we can conclude:

- **CGMA Dominates:** CGMA has the highest importance (0.908) as it directly impacts performance in memory-bound kernels by balancing computation and global memory access.
- **Low Occupancy Impact:** Occupancy has minimal importance (0.0005) because memory latency, not thread scheduling, dominates performance in memory-bound scenarios.
- **Other Features:** Metrics like shared memory utilization and branch divergence show limited influence, reflecting their minor role in the tested kernels.

Table 3. Feature Importance for Kernel Speedup Prediction

Feature	**Importance**
`CGMA`	0.908
`Global Memory Access`	0.0487
`Maximum Percentage of Blocks`	0.0183
`Num of Bblocks`	0.0143
`Shared Memory Utilization`	0.0054
`Problem Size`	0.0030
`Threads per Block`	0.0024
`Branch Divergence`	0.0010
`Occupancy`	0.0005

5.5. Visual Analysis

Figure 1 visualizes the feature importance, highlighting that `CGMA` is the most influential metric for predicting kernel speedup, while other metrics like `occupancy` and `branch_divergence` have minimal impact.

Additionally, the visualization of the trained Random Forest model is presented in Appendix A.

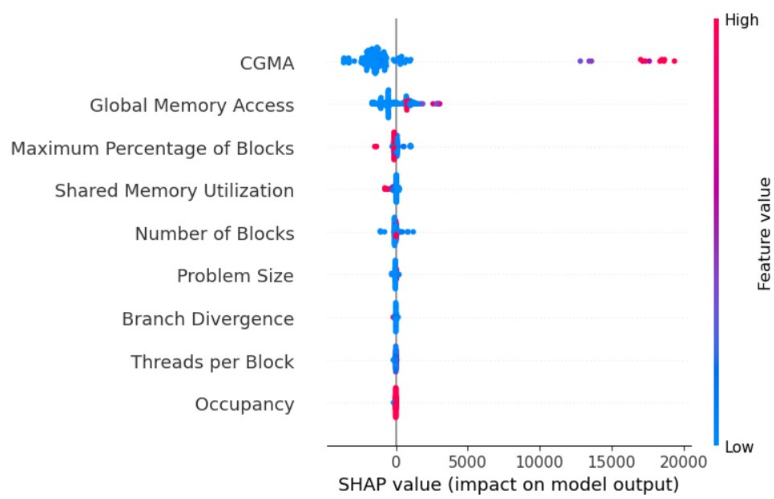

Fig. 1. Feature Importance Visualization

5.6. Discussion

- **Strengths:** The high R^2 score and low MAE demonstrate that the model accurately predicts kernel speedup. The feature importance analysis provides interpretable insights into which metrics significantly influence performance.
- **Weaknesses:** The MAPE value of 55.15% suggests that prediction errors are higher for

some kernels, particularly those with extreme configurations. This could be attributed to insufficient representation of edge cases in the dataset.

Overall, the model performs well for the majority of configurations and highlights opportunities for improving dataset diversity to handle outliers effectively.

To further enhance the results, future work could involve:

- Testing on additional datasets and architectures.
- Incorporating advanced models like Gradient Boosting or Neural Networks for comparison.
- Exploring additional metrics or transformations to improve accuracy.

6. Conclusion

In this project, we developed a machine learning-based approach to predict the speedup of different CUDA kernels using a set of microbenchmarks and performance metrics. By combining insights from static analysis with runtime profiling, we demonstrated the ability to model and predict kernel performance without requiring actual kernel execution. Our approach takes advantage of the flexibility of machine learning, providing a scalable and efficient solution for performance prediction.

The key findings of this project include:

- **Accurate Performance Prediction:** The Random Forest model achieved a high R^2 score of 0.9287 and an explained variance score of 0.9291, successfully explaining over 92% of the variance in CUDA kernel speedup. The Mean Absolute Percentage Error (MAPE) of 55.15% further demonstrates its reliability across diverse kernels.
- **Importance of CGMA:** The Compute-to-Global-Memory-Access (CGMA) ratio emerged as the most influential feature, underscoring its critical role in kernel performance. Other metrics, like memory reduction per thread, also contributed meaningfully.
- **Model Limitations:** Edge cases with high MAE highlighted areas where dataset diversity and feature representation could be improved, particularly for metrics like occupancy and branch divergence, which showed minimal impact.

Despite the model's strengths of robust predictions and interpretable feature importance metrics, its limitations also suggest opportunities for further improvement. The relatively high MAE for certain kernels may reflect difficulties in modeling irregular performance patterns or inherent variability in kernel configurations. Additionally, the current dataset may not comprehensively represent all possible configurations or GPU architectures, potentially limiting the model's generalizability.

Future work could address these limitations by:

- Expanding the dataset with diverse kernel configurations and testing across different GPU architectures to improve generalization.
- Refining profiling techniques for metrics like branch divergence and shared memory utilization to better represent their impact on performance.
- Including additional GPU-specific features, such as warp scheduling and memory bandwidth, to enhance predictive accuracy.

In conclusion, this project provides a practical and scalable framework for CUDA kernel performance prediction, reducing the reliance on runtime profiling. The results demonstrate the model's potential for optimizing and understanding GPU workloads, while its limitations outline clear directions for future advancements.

References

1. S. K. Muller and J. Hoffmann, "Modeling and analyzing evaluation cost of cuda kernels," ACM Trans. Parallel Comput. **11** (2024).
2. J. Sim, A. Dasgupta, H. Kim, and R. Vuduc, "A performance analysis framework for identifying potential benefits in gpgpu applications," SIGPLAN Not. **47**, 11–22 (2012).
3. R. Membarth, O. Reiche, F. Hannig, *et al.*, "Hipa^{cc}: A domain-specific language and compiler for image processing," IEEE Trans. Parallel Distrib. Syst. **27**, 210–224 (2016).
4. A. Riahi, A. Savadi, and M. Naghibzadeh, "Many-bsp: an analytical performance model for cuda kernels," Computing **106**, 1519–1555 (2024).
5. E. Konstantinidis and Y. Cotronis, "A practical performance model for compute and memory bound gpu kernels," in *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing,* (2015), pp. 651–658.
6. M. Lou and S. K. Muller, "Automatic static analysis-guided optimization of cuda kernels," in *Proceedings of the 15th International Workshop on Programming Models and Applications for Multicores and Manycores,* (Association for Computing Machinery, New York, NY, USA, 2024), PMAM '24, p. 11–21.
7. G. Alavani, K. Varma, and S. Sarkar, "Predicting execution time of cuda kernel using static analysis," in *2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom),* (2018), pp. 948–955.
8. S. Hong and H. Kim, "An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness," in *Proceedings of the 36th Annual International Symposium on Computer Architecture,* (Association for Computing Machinery, New York, NY, USA, 2009), ISCA '09, p. 152–163.
9. Q. Wang, C. Liu, and X. Chu, "Gpgpu performance estimation for frequency scaling using cross-benchmarking," in *Proceedings of the 13th Annual Workshop on General Purpose Processing Using Graphics Processing Unit,* (Association for Computing Machinery, New York, NY, USA, 2020), GPGPU '20, p. 31–40.
10. D. Lymberopoulos, O. Riva, K. Strauss, *et al.*, "Pocketweb: instant web browsing for mobile devices," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems,* (Association for Computing Machinery, New York, NY, USA, 2012), ASPLOS XVII, p. 1–12.

A. Additional Materials

Click here to access the tree model diagram