A Deep Learning Approach to CUDA Kernel Launch Configuration Prediction: Challenges, Design, and Implementation

1. Introduction: The Optimization Bottleneck

1.1 The Challenge of GPU Kernel Performance Tuning

The optimization of GPU-accelerated applications is a critical and complex task, fundamentally driven by the need to maximize computational throughput and minimize latency. At the heart of this challenge lies the CUDA kernel, a function executed by many threads in parallel on the GPU. To launch a kernel, a programmer must specify its launch configuration, which includes the number of threads per block and the grid dimensions. This seemingly simple choice is, in practice, a non-trivial decision that profoundly impacts performance.[1]

Historically, this has been considered an art form requiring deep domain knowledge and intuition. The optimal configuration for a given kernel is not static; it depends on the kernel's internal logic, the input data size, and the specific hardware architecture of the target GPU.[2] A configuration that performs well on one GPU may be suboptimal on another. Traditional approaches, such as manual trial-and-error or the use of general-purpose tools like the NVIDIA Occupancy Calculator, are often insufficient. The Occupancy Calculator, for instance, focuses on maximizing the number of active warps per Streaming Multiprocessor (SM) but fails to account for other factors that limit performance, such as memory bandwidth or cache behavior.[1] In many cases, high occupancy does not directly translate to the lowest possible execution time.

Compilers like nvcc provide a range of general optimization flags and heuristics to improve performance.[4] These options, such as
--maxrregcount to limit register usage or --use_fast_math to enable faster but less precise floating-point operations, offer broad benefits but cannot fine-tune the kernel launch parameters for a specific, unique workload. The inherent complexity of this optimization problem can be conceptualized as a vast, multi-dimensional search space of possible launch configurations.[2] Manually searching this space is prohibitively time-consuming, while a simple,

brute-force approach is computationally expensive. This bottleneck in performance tuning has created a demand for an automated, intelligent solution.

1.2 The Promise of Machine Learning for Performance Prediction

The manual nature and complexity of CUDA kernel tuning make it an ideal candidate for a data-driven, machine learning-based approach. The core premise of this project is to reframe this optimization problem as a supervised learning task. In this framework, a model is trained on a dataset where each example consists of a kernel's intrinsic properties (features) and the empirically determined optimal launch configuration for that kernel (the label). The goal is to build a predictive system that can learn the intricate relationship between a kernel's characteristics and its optimal launch configuration, thereby bypassing the need for manual guesswork or exhaustive search.
A deep learning model, specifically a Multi-Layer Perceptron (MLP), is well-suited for this challenge. Unlike traditional linear models, MLPs consist of multiple layers of neurons that can learn complex, non-linear relationships between inputs and outputs.[7] This capability is essential because the mapping from a kernel's static properties to its dynamic performance is not a simple linear function. An MLP can function as a powerful "black box" that, through training on a rich dataset, uncovers the intricate dependencies that are often opaque to human intuition. By training on a diverse set of kernels and their measured performance, the model can learn to make intelligent, low-latency predictions for new, unseen kernels, making this approach a transformative alternative to traditional, manual methods.

1.3 Scope and Goals of this Report

This report serves as a detailed blueprint for the design and implementation of a deep learning-based system for predicting optimal CUDA kernel launch configurations. The analysis will progress from the foundational principles of GPU performance to the architectural design of the predictive model and conclude with a practical, actionable workflow for implementation. The report will address critical aspects such as problem formulation, feature engineering, and model selection. By grounding its recommendations in established research and profiling techniques, this document aims to provide a strategic and expert-level guide for building a robust and generalized performance prediction system.

2. Foundations of GPU Performance and Kernel Characterization

2.1 CUDA Execution Model and the Role of Launch Configurations

The CUDA programming model, a cornerstone of high-performance computing on NVIDIA GPUs, is built on a hierarchical execution structure. At the lowest level are threads, which are grouped into thread blocks. These blocks are, in turn, organized into a grid. When a kernel is launched, the programmer must specify the number of threads per block (threadsPerBlock) and the grid dimensions (gridDim). This launch configuration is the primary interface for partitioning the workload across the GPU's hardware.
The number of threads in a block has a direct impact on how the kernel utilizes GPU resources. A block is scheduled to run on a Streaming Multiprocessor (SM), a core component of the GPU architecture. The SM has finite, on-chip resources, including registers and shared memory. A programmer's choice of block size dictates the number of threads competing for these limited resources. If a kernel uses a large number of registers or a significant amount of shared memory per thread, the SM may only be able to run a small number of blocks concurrently, even if the total number of threads is below the SM's maximum.[1] This creates a critical trade-off: a larger block size may increase parallelism within the block but can restrict the number of blocks that can run simultaneously on the SM, potentially limiting the overall performance. The predictive model must learn to navigate this intricate balance to recommend a configuration that optimizes resource usage for a given kernel.

2.2 Key Performance Limiters and Profiling

Understanding the performance of a CUDA kernel requires looking beyond simple execution time. One of the most common metrics used is occupancy, which is defined as the ratio of active warps (groups of 32 threads) to the maximum number of warps that can be active on a single SM. While high occupancy is often a good indicator, as it helps hide memory latency by providing the SM with more ready-to-run warps, it is not a direct proxy for peak performance.[1] The STATuner paper explicitly notes that maximizing occupancy does not always lead to the best performance, indicating that other metrics must be considered to make an informed decision.[1]
A more fundamental way to characterize kernels is to classify them as either "compute-bound" or "memory-bound".[10] A compute-bound kernel's performance is limited by the rate at which the SMs can execute arithmetic operations, while a memory-bound kernel is bottlenecked by the speed of memory access. The optimal launch configuration and overall optimization strategy for these two kernel types are drastically different.
A detailed analysis of memory access patterns is essential for understanding and optimizing kernel behavior. Key considerations include:

- **Coalesced Global Memory Access:** To maximize memory throughput, threads within a warp should access contiguous, sequential memory addresses. Modern NVIDIA GPUs perform global memory loads in 128-byte cache lines. If a warp of 32 threads accesses

32-bit addresses in a sequential fashion, the entire load can be processed in a single transaction, a process known as memory coalescing.[9] Deviating from this pattern leads to inefficient, non-coalesced access, which can cause excessive overhead and significantly reduce the effective memory utilization.[11]
- **Shared Memory Bank Conflicts:** Shared memory, a fast, on-chip memory type, is divided into 32 banks. If multiple threads within a single warp attempt to access the same memory bank simultaneously, the accesses are serialized. This phenomenon, known as a bank conflict, can severely degrade performance.[11] Optimizing for this requires either careful memory access patterns or padding to shift accesses to different banks.

2.3 Static vs. Dynamic Kernel Features

The properties that influence a kernel's performance can be broadly divided into two categories: static and dynamic features. Static features are those that can be determined from the kernel's source code or its compiled representation (e.g., LLVM Intermediate Representation, PTX, or SASS) without requiring any execution on the hardware. As demonstrated in the STATuner paper, a powerful set of static metrics can be collected via a compiler pass, including:
- The ratio of branch, load, integer, and floating-point instructions to the total number of instructions.
- The number of loops and synchronization operations (__syncthreads).
- The total register and shared memory usage per block.[1]

Dynamic features, in contrast, are runtime metrics that capture the kernel's behavior during execution on the hardware. They include metrics such as GPU utilization, achieved occupancy, cache hit rates, memory throughput, and power utilization.[9] These metrics provide direct, empirical evidence of how the kernel is performing.

The fundamental relationship between these two categories is one of cause and effect. A kernel's static characteristics (e.g., its high ratio of load instructions or significant shared memory usage) are the *cause* that influences its dynamic behavior (e.g., its low achieved occupancy or high memory bandwidth utilization). The core challenge for a predictive model is to learn this intricate, non-linear mapping. The inclusion of both static and dynamic features in the training data allows the model to learn not just the correlation between the code's structure and performance, but the deeper causal link that connects them.

3. Architectural Design of the Predictive Deep Learning Model

3.1 Problem Formulation: Regression or Classification?

The initial impulse for a project like this might be to treat the prediction of an optimal launch configuration as a regression problem, where the model outputs a continuous, numerical value for parameters like the number of threads per block. However, a deeper analysis reveals that this approach is fundamentally flawed.

3.1.1 The Case Against Simple Regression

The output variables—the optimal number of threads per block and grid dimensions—are inherently discrete and bounded. The number of threads per block must be an integer, is typically a multiple of 32 (the warp size), and is strictly limited by the GPU's hardware to a maximum of 1024 or 512 threads.[2] A regression model, designed for continuous variables, could output non-integer values or numbers outside the physical bounds, such as a negative number of threads or a block size of 256.3, which are impossible.[16]
Furthermore, the assumptions underlying standard linear regression are violated by this data. These models assume a normal distribution of errors, linearity between features and the target variable, and uniform variance across the data (homoscedasticity).[16] The distribution of optimal launch configurations is not normal; it is discrete and truncated by the hardware limits. The variance of the output data is also not uniform, as it approaches zero at the bounds. Attempting to apply a simple linear regression model to this problem is therefore ill-advised.

3.1.2 The Superiority of a Classification or Hybrid Approach

The problem is far better framed as a classification task. Instead of predicting a continuous value, the model should be trained to classify a kernel into one of a finite set of known, high-performing launch configurations. This approach is precisely what the STATuner paper successfully demonstrated by using a Support Vector Machine (SVM) *classifier* to predict the optimal block size.[1] By defining a pool of common and effective configurations (e.g., 128, 192, 256, 512 threads per block), the model's output is constrained to a valid and meaningful set of choices, preventing impossible or nonsensical predictions.
A more sophisticated alternative is a hybrid model that combines elements of both classification and regression. The model could predict a probability distribution over a set of pre-defined "good" launch configurations, offering a nuanced recommendation that accounts for multiple potential optima. This approach respects the discrete and bounded nature of the output while providing more information than a single, hard classification.

3.2 Feature Engineering: Constructing the Training Input

The success of any supervised learning model hinges on the quality and relevance of its input features. For this project, a comprehensive feature set should be engineered to effectively characterize the kernels. This feature set must include both static and dynamic metrics to capture the full spectrum of a kernel's behavior.
Table 1: Kernel Feature Taxonomy

Feature Category	Description	Specific Metrics & Examples	Source
Static Features	Metrics that can be determined from the source code or compiled representation of the kernel without execution.	num branch instructions / num instructions num load instructions / num instructions num loops num syncthreads operations registers usage shared memory used per block num int instructions / num instructions num float instructions / num instructions	1
Dynamic Features	Runtime performance metrics collected during kernel execution on the target hardware.	achieved_occupancy dram_utilization issue_slot_utilization GPU utilization % GPU power utilization % Memory throughput Cache hit rates Warp execution efficiency	9

A specialized compiler pass, for example, built on the LLVM framework, can be used to automatically extract static features from the kernel's source code.[1] This automation is essential for building a large and diverse training dataset without manual effort. Dynamic features would be collected during the data generation phase using NVIDIA's profiling tools, capturing the kernel's real-world behavior under various configurations. The collection of these dynamic metrics is crucial because they provide the empirical data that a static-only analysis cannot.

3.3 Model Selection and Training Strategy

3.3.1 A Multi-Layer Perceptron (MLP) as the Core Predictor

For the core predictive component, a Multi-Layer Perceptron (MLP) is a highly suitable choice. The MLP's architecture, comprising an input layer, one or more hidden layers, and an output layer, allows it to learn the complex, non-linear relationships that exist between a kernel's features and its optimal launch configuration.[7] The model would be trained using an optimization algorithm like Adam, which is robust and converges quickly on large datasets. Critical hyperparameters, such as the number of hidden layers and neurons, would be tuned to find the best balance between model complexity and generalization ability.[7]

3.3.2 The NeuSight Tiling Approach: A Game-Changer for Generalization

A critical observation from recent research provides a transformative approach to this problem. Prior machine learning models for GPU performance prediction often struggled to generalize their predictions to new, unseen GPUs or kernels, leading to high error percentages.[19] This limitation stemmed from a flawed approach: they attempted to model the end-to-end latency of a full kernel, a complex, high-dimensional problem that is difficult to capture with simple models.

The NeuSight framework, in contrast, adopts a superior decomposition strategy. It leverages the fact that many popular GPU libraries and deep learning kernels are composed of smaller, repetitive working sets called "tiles" that are executed independently on the SMs.[20] Instead of predicting the behavior of the entire, complex kernel, the NeuSight approach simplifies the task by predicting the utilization for a single, standardized tile. This decomposition transforms the problem into a "more manageable sub-problem," making it far more tractable for a simple MLP to learn.[20] The predictions for individual tiles are then aggregated to estimate the end-to-end performance of the full kernel.

This strategic shift is a fundamental improvement. By focusing on the predictable, repeatable behavior of a kernel's building blocks rather than its complex, high-level execution, the model's ability to generalize is significantly improved. This is the central recommendation for this project: the model should be designed to predict the optimal configuration for a standardized tile, and these predictions should then be used to inform the overall launch configuration for the full kernel. This ensures the model's long-term viability and robustness across diverse hardware and workloads.

4. Implementation and Automated Workflow

4.1 Dataset Generation Pipeline

The most critical and resource-intensive phase of this project is the generation of a large and diverse training dataset. This process must be automated to be scalable and efficient. An excellent model for this is the automated tuning workflow used by tools like Kernel Launcher and Kernel Tuner.[22]

A robust data generation pipeline would operate as follows:

1. **Kernel Capture:** The system would capture kernels and their input data from a wide variety of applications, such as those in the Rodinia Benchmark Suite.[1]
2. **Static Feature Extraction:** For each captured kernel, an automated compiler pass would extract its static features, as detailed in Table 1.
3. **Automated Profiling:** The system would systematically launch each kernel with a broad range of valid launch configurations. For each configuration, it would use a profiling tool like NVIDIA Nsight Systems or Nsight Compute to measure a suite of dynamic performance metrics and the kernel's total execution time.[23] The CUPTI API can also be used for real-time, low-overhead metric collection.[15]
4. **Data Labeling:** For each kernel, the system would identify the launch configuration that resulted in the minimum execution time. This optimal configuration would become the "label" for the training example. The corresponding static and dynamic features would serve as the "input."

This automated process replaces manual guesswork with empirical, data-driven collection, establishing the foundation for a highly accurate predictive model.

4.2 System Architecture

The proposed system would be a modular tool that automates the entire process, from kernel analysis to launch configuration prediction. The architecture would consist of four main components:

1. **A Static Analyzer Module:** This component, built as a compiler pass (e.g., using LLVM), would automatically parse kernel source code to extract static features, such as instruction counts and register usage.
2. **A Profiling and Data Collection Module:** This module would serve as a wrapper around NVIDIA's profiling tools (Nsight Systems, Nsight Compute) and APIs (CUPTI). It would orchestrate the launching of kernels, collect dynamic performance metrics, and store the results in a structured format for dataset generation.
3. **A Deep Learning Model Module:** This component would house the MLP model. It would manage the training process on the generated dataset and store the trained model for inference.
4. **An Inference Engine:** This is the user-facing part of the tool. It would take a new kernel, pass it through the Static Analyzer, and then use the trained deep learning

model to predict the optimal launch configuration in a low-latency manner.

4.3 Practical Considerations

Several practical considerations must be addressed for a successful implementation. The combinatorial explosion of possible launch configurations can make the brute-force data generation approach challenging. This is where a smarter search strategy, such as Bayesian optimization, can be employed to efficiently find the optimal configuration during the dataset generation phase.[22] The NeuSight tiling approach is also instrumental in mitigating this issue, as it simplifies the search space from the entire kernel to a single, repetitive tile. Furthermore, a deep learning model like an MLP is sensitive to feature scaling. All input features, both static and dynamic, must be standardized to a consistent range (e.g., mean 0, variance 1) before training to ensure stable and effective learning.[7] Hyperparameter tuning is also essential to find the right balance between model complexity and performance.[7]

5. Broader Context: Beyond Supervised Learning

5.1 Comparison to Traditional Autotuners

The proposed deep learning approach represents a distinct paradigm shift from traditional autotuning systems. It is crucial to understand the fundamental difference between these two methods.

Traditional autotuners, such as Kernel Tuner and OpenTuner, are *exploratory* systems.[5] They operate by performing a search in real-time on the target hardware to find the best configuration for a given problem instance. Their strength lies in their ability to find a near-optimal solution for a specific setup. However, this process incurs a high overhead due to the repeated compilation, execution, and profiling cycles, which can be prohibitive for applications with a large number of small kernels or for dynamic, just-in-time compilation.

The proposed deep learning model, in contrast, is a *predictive* system. The computationally expensive work of generating the dataset and training the model is performed offline. Once trained, the model can perform inference in milliseconds, providing an instant prediction of the optimal launch configuration without requiring any on-the-fly search or profiling. This makes it ideal for latency-sensitive applications or for rapidly deploying workloads on new, unseen hardware where an online search is not feasible.

Table 2: Comparative Analysis of Optimization Paradigms

Criterion	Proposed Deep Learning Model	Traditional Autotuners	Emerging LLM/RL Systems

Optimization Strategy	**Predictive:** Offline training, low-latency online inference.	**Exploratory:** Real-time search and empirical testing on hardware.	**Generative/Iterative:** Generates novel code variants and refines them based on performance feedback.
Overhead	High initial training cost; near-zero per-use inference cost.	High per-use overhead due to repeated execution and profiling.	High initial training cost; high per-use overhead from iterative generation and testing.
Data Requirements	Requires a large, diverse dataset of pre-profiled kernels and their features.	Requires no pre-existing dataset; generates data through an online search.	Requires minimal human-written code and uses performance as a reward signal.
Generalization	Excellent, particularly with the NeuSight tiling approach. Can predict for unseen kernels/hardware.	Limited. Optimal configurations are specific to the hardware and problem instance.	Exceptional. Can generalize and discover novel optimization principles for new kernels and architectures.
Use Case Suitability	Latency-sensitive applications, dynamic workloads, rapid deployment on new platforms.	Offline, one-time optimization of critical kernels.	Frontier R&D, automated code generation, and complex, multi-objective optimization.

5.2 Emerging Trends: Generative AI and Reinforcement Learning

While the supervised learning approach is a powerful and practical solution, it is important to contextualize it within the broader, rapidly evolving field of automated kernel optimization. The frontier of this domain involves the use of generative AI and reinforcement learning (RL). Large Language Models (LLMs), such as DeepSeek-R1, are being used as autonomous agents that can generate, test, and iteratively refine GPU kernels.[29] These systems are far more advanced than a predictive model; they go beyond recommending a configuration and can generate entirely new, optimized code, often with a special verifier that guides the model in a closed-loop fashion.[29] Researchers have even framed these LLMs as a "GPU Kernel Scientist" that can generate hypotheses for optimization experiments and autonomously implement

them based only on end-to-end timing results.[31]
Furthermore, reinforcement learning is being applied to the problem. The CUDA-L1 framework uses a novel RL algorithm to transform a poor-performing LLM into a highly effective CUDA optimizer.[32] The model learns to optimize based on a "speedup-based reward signal" and can even uncover fundamental principles of CUDA optimization without explicit human domain knowledge.[32] This approach demonstrates that a model can learn the rules of optimization by being rewarded for performance improvements.
These emerging LLM and RL-based systems represent the next generation of automated kernel optimization, shifting the focus from predicting configurations to generating entirely new, optimized code. The supervised learning project outlined in this report is a crucial and feasible step toward this future, providing a powerful, near-term solution for a well-defined and critical problem.

6. Conclusion and Recommendations

The manual and often intuitive process of optimizing CUDA kernel launch configurations has historically been a significant bottleneck in high-performance computing. This report has demonstrated that this complex, multi-dimensional problem is a prime candidate for a data-driven, machine learning solution. The analysis has established that a simple regression model is ill-suited for this task due to the discrete and bounded nature of the output parameters and the violation of key statistical assumptions. A classification or hybrid approach is far superior and more robust.
The success of the proposed system hinges on two key factors. First, a comprehensive feature set combining both static (e.g., instruction counts, register usage) and dynamic (e.g., achieved occupancy, memory throughput) metrics is required to capture the full cause-and-effect relationship that governs a kernel's performance. Second, and most critically, the model must adopt a decomposition strategy similar to the NeuSight framework, which simplifies the problem by modeling a kernel's predictable, repeatable "tiles" rather than its complex, high-level execution. This strategic design choice will significantly enhance the model's ability to generalize to new, unseen kernels and hardware.
Based on this analysis, the recommended path forward is a phased implementation:

- **Phase 1: Foundation Building.** Prioritize the establishment of a robust, automated data collection pipeline. This involves using NVIDIA's profiling tools (Nsight Systems/Compute) and APIs (CUPTI) to systematically profile a diverse set of kernels and their various launch configurations. The output of this phase will be the high-quality, labeled dataset required for model training.
- **Phase 2: Model Prototyping.** Begin by training a simple Multi-Layer Perceptron (MLP) classifier using the static features extracted in the first phase. This will serve as a baseline model to validate the overall approach.
- **Phase 3: Refinement and Generalization.** Refine the model by integrating the NeuSight tiling decomposition and incorporating dynamic features into the training

data. This will improve the model's accuracy and ensure its long-term viability and performance across new hardware architectures.

By following this blueprint, the project can create a powerful predictive system that democratizes and accelerates a traditionally expert-driven domain, providing a significant competitive advantage for high-performance computing applications.

Works cited

1. STATuner: Efficient Tuning of CUDA Kernels Parameters ∗ - SC15, accessed August 21, 2025, https://sc15.supercomputing.org/sites/all/themes/SC15images/tech_poster/poster_files/post276s2-file3.pdf
2. How to decide the optimal block size in CUDA - NVIDIA Developer Forums, accessed August 21, 2025, https://forums.developer.nvidia.com/t/how-to-decide-the-optimal-block-size-in-cuda/14906
3. Snowpack: efficient parameter choice for GPU kernels via static analysis and statistical prediction | Request PDF - ResearchGate, accessed August 21, 2025, https://www.researchgate.net/publication/320849618_Snowpack_efficient_parameter_choice_for_GPU_kernels_via_static_analysis_and_statistical_prediction
4. CUDA Compiler Driver - nvcc - NVIDIA Documentation, accessed August 21, 2025, https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/
5. (PDF) Autotuning CUDA Compiler Parameters for Heterogeneous ..., accessed August 21, 2025, https://www.researchgate.net/publication/309735955_Autotuning_CUDA_Compiler_Parameters_for_Heterogeneous_Applications_using_the_OpenTuner_Framework
6. Show HN: Luminal – Open-source, search-based GPU compiler | Hacker News, accessed August 21, 2025, https://news.ycombinator.com/item?id=44963135
7. 1.17. Neural network models (supervised) - Scikit-learn, accessed August 21, 2025, https://scikit-learn.org/stable/modules/neural_networks_supervised.html
8. Multi-Layer Perceptron Learning in Tensorflow - GeeksforGeeks, accessed August 21, 2025, https://www.geeksforgeeks.org/deep-learning/multi-layer-perceptron-learning-in-tensorflow/
9. Profiling CUDA Kernels. This is the fifth article in the series… | by Rimika Dhara | Medium, accessed August 21, 2025, https://medium.com/@rimikadhara/cuda-4-profiling-cuda-kernels-0664252f0ac5
10. (PDF) Metric Selection for GPU Kernel Classification - ResearchGate, accessed August 21, 2025, https://www.researchgate.net/publication/330237137_Metric_Selection_for_GPU_Kernel_Classification
11. Part II - CUDA Kernel Optimization Tips - Vrushank Desai, accessed August 21, 2025, https://www.vrushankdes.ai/diffusion-policy-inference-optimization/part-ii---cud

a-kernel-optimization-tips
12. Optimize Your CUDA Applications - Essential Tools for Performance Analysis - MoldStud, accessed August 21, 2025, https://moldstud.com/articles/p-optimize-your-cuda-applications-essential-tools-for-performance-analysis
13. GPU Metrics | Modal Docs, accessed August 21, 2025, https://modal.com/docs/guide/gpu-metrics
14. CUDA Memory Architecture, accessed August 21, 2025, https://ajdillhoff.github.io/notes/cuda_memory_architecture/
15. CUPTI Profiling API Injection Tutorial - eunomia, accessed August 21, 2025, https://eunomia.dev/others/cupti-tutorial/profiling_injection/
16. Regression Models for Count Data - The Analysis Factor, accessed August 21, 2025, https://www.theanalysisfactor.com/regression-models-for-count-data/
17. Linear regression when Y is bounded and discrete - Cross Validated, accessed August 21, 2025, https://stats.stackexchange.com/questions/395548/linear-regression-when-y-is-bounded-and-discrete
18. Autotuning LLVM Optimization Passes for Matrix Multiplication in Rust, accessed August 21, 2025, https://sol.sbc.org.br/index.php/eradsp/article/download/16880/16720/
19. Forecasting GPU Performance for Deep Learning Training and Inference - arXiv, accessed August 21, 2025, https://arxiv.org/html/2407.13853v2
20. Forecasting GPU Performance for Deep Learning Training and Inference - arXiv, accessed August 21, 2025, https://arxiv.org/html/2407.13853v3
21. [Literature Review] Forecasting GPU Performance for Deep Learning Training and Inference, accessed August 21, 2025, https://www.themoonlight.io/en/review/forecasting-gpu-performance-for-deep-learning-training-and-inference
22. GPU kernel tuning — MicroHH 2.0 documentation, accessed August 21, 2025, https://microhh.readthedocs.io/en/latest/computing_systems/gpu_tuning.html
23. User Guide — NsightGraphics 2025.4 documentation, accessed August 21, 2025, https://docs.nvidia.com/nsight-graphics/UserGuide/
24. User Guide — nsight-systems - NVIDIA Documentation, accessed August 21, 2025, https://docs.nvidia.com/nsight-systems/UserGuide/index.html
25. 1. Preparing An Application For Profiling - NVIDIA Documentation, accessed August 21, 2025, https://docs.nvidia.com/cuda/profiler-users-guide/
26. NVIDIA Nsight Compute, accessed August 21, 2025, https://developer.nvidia.com/nsight-compute
27. 2. Usage — Cupti 13.0 documentation - NVIDIA Docs Hub, accessed August 21, 2025, https://docs.nvidia.com/cupti/main/main.html
28. CUDA Code Optimization and Auto-Tuning Made Easy S31429 | GTC Digital April 2021 | NVIDIA On-Demand, accessed August 21, 2025, https://www.nvidia.com/en-us/on-demand/session/gtcspring21-s31429/
29. Automating GPU Kernel Generation with DeepSeek-R1 and Inference Time Scaling, accessed August 21, 2025,

https://developer.nvidia.com/blog/automating-gpu-kernel-generation-with-deepseek-r1-and-inference-time-scaling/
30. LLM Scientist in GPU Kernel Optimization - Emergent Mind, accessed August 21, 2025, https://www.emergentmind.com/topics/llm-as-scientist
31. GPU Kernel Scientist: An LLM-Driven Framework for Iterative ... - arXiv, accessed August 21, 2025, https://arxiv.org/html/2506.20807
32. CUDA-L1: Improving CUDA Optimization via Contrastive Reinforcement Learning - arXiv, accessed August 21, 2025, https://arxiv.org/abs/2507.14111
33. CUDA-L1: Improving CUDA Optimization via Contrastive Reinforcement Learning | Cool Papers - Immersive Paper Discovery, accessed August 21, 2025, https://papers.cool/arxiv/2507.14111
34. CUDA-L1: How AI Self-Optimizes GPU Kernels for 3x Faster Performance with Contrastive RL - YouTube, accessed August 21, 2025, https://www.youtube.com/watch?v=xsEjrh0B54U