FlipFlop: A Static Analysis-based Energy Optimization Framework for GPU Kernels

Saurabhsingh Rajput
Dalhousie University
Halifax, Canada
saurabh@dal.ca

Alex Brandt
Dalhousie University
Halifax, Canada
ABrandt@dal.ca

Vadim Elisseev
IBM Research
Warrington, United Kingdom
vadim.v.elisseev@ibm.com

Tushar Sharma
Dalhousie University
Halifax, Canada
tushar@dal.ca

Abstract

Artificial Intelligence (AI) applications, such as Large Language Models, are primarily driven and executed by Graphics Processing Units (GPUs). These GPU programs (kernels) consume substantial amounts of energy, yet software developers often lack the hardware expertise and ad hoc knowledge required to optimize for power efficiency. We propose ***FlipFlop***, a framework using static code analysis to predict energy consumption and recommend Pareto-optimal thread block configurations considering both power consumption and execution time. Our framework requires no runtime execution and analyzes PTX code, a low-level instruction set for CUDA-enabled GPUs. It is validated across a diverse set of GPUs and kernels, including multi-head attention, convolution, and matrix multiplication. *FlipFlop* achieves 83% accuracy in identifying locally optimal energy-efficient configurations, while also minimizing developer effort by reducing the optimization search space by 93.4%. For multi-head attention kernels, it yields up to 79% energy savings and 106% throughput gains relative to NVIDIA's occupancy heuristic. By integrating static analysis with real-time monitoring and providing explainable optimization guidance, *FlipFlop* empowers developers to create sustainable, high-performance GPU software which minimizes environmental and computational costs.

Keywords

Green AI, GPU Power Modeling, CUDA Optimization, LLM Inference, Static Analysis, Performance Portability

1 Introduction

The rapid evolution of Artificial Intelligence (AI), particularly Large Language Models (LLMs) [8, 11, 53], has led to the emergence of AI-enabled software systems with unprecedented computational demands. Training and deploying these models consumes significant energy; the International Energy Agency notes that data center electricity—driven heavily by AI training and inference—accounts for 2% of global electricity, projected to more than double by 2026, surpassing Canada's national power consumption [19, 23, 35].

Traditional software engineering practices rely on programming abstraction layers that conceal underlying complexity. Frameworks like TensorFlow [57] and PyTorch cover hardware complexity, making AI development easier. While these abstractions provide usability, hardware-agnostic execution, and accelerated development, they create *hardware opacity* that prevents developers from leveraging architecture-specific optimizations that could dramatically reduce energy consumption.

This hardware opacity manifests concretely in common development scenarios. CUDA kernels—programs executed on NVIDIA GPUs—are critical for optimizing AI workloads but require manual tuning for efficiency [64]. Consider developing custom attention mechanisms or optimizing scientific computing workflows: developers could achieve substantial energy savings by manually tuning CUDA kernels to optimize memory access patterns. However, the specialized expertise required creates a significant barrier; most developers default to framework-provided implementations prioritizing convenience over energy efficiency. This efficiency gap compounds at scale: inefficiencies across months-long model training and millions of inference calls lead to substantial wastage in energy and time. This problem is aggravated by AI-generated code. Current LLMs are predominantly trained on open-source repositories lacking rigorous efficiency checks [59]. When these models generate new code, they replicate and amplify existing inefficiencies. As this AI-generated code enters public codebases and becomes training data for future models, we risk entering a degenerative cycle known as "*Model Collapse*" [65], where progressively inefficient implementations generate even less optimized code.

Addressing these systemic inefficiencies requires new approaches that integrate hardware-aware optimizations into software engineering workflows, enabling developers to prioritize energy efficiency without deep hardware expertise. Specialized frameworks such as CUDA [64], ROCm [67], and SYCL [68] enable direct hardware programming, but optimizing their parameters—particularly thread block configurations—remains a manual, expertise-intensive bottleneck. Each kernel must be hand-tuned by developers with specialized hardware knowledge, creating critical challenges in large-scale AI deployment pipelines. This challenge is exacerbated by heterogeneous hardware environments where modern AI systems span multiple GPU architectures (*e.g.,* mixing NVIDIA's Ampere, Hopper, and Blackwell generations), requiring per-device tuning that multiplies engineering effort.

Current approaches to automating this tuning process remain fundamentally constrained by runtime measurement reliance. State-of-the-art tools such as Kernel Tuner [58] require *execution* to measure energy usage, making comprehensive optimization impractical

at LLM scales. For an AI workload with n configuration parameters, exhaustive profiling requires $O(2^n)$ runs—wasting significant energy and time. This runtime dependency creates three prohibitive costs: (1) *profiling overhead* from executing hundreds of kernel variants; (2) *energy waste* as suboptimal configurations compound across millions of tokens; and (3) *cold-start delays* when transitioning between GPU architectures (*e.g.,* migrating from NVIDIA Ampere to Blackwell GPUs requires repeating optimization from scratch). These challenges are compounded by restricted hardware access in cloud environments and shared clusters, where power monitoring often requires privileged access. Consequently, energy inefficiencies typically surface only post-deployment, necessitating costly re-engineering cycles. Critically, preliminary energy assessment, essential for long-running training jobs, remains practically infeasible with current runtime-dependent approaches.

These limitations highlight the need for *static energy modeling* techniques that analyze and optimize code without execution. Desirable features include: (1) identifying energy hotspots in source code during development rather than deployment, (2) ranking GPU configurations by efficiency before runtime testing, and (3) providing explainable optimization guidance to developers.

In response, we propose ***FlipFlop***—a static analysis-based kernel optimization framework for energy-conscious GPU software development. *FlipFlop* analyzes NVIDIA's Parallel Thread Execution (PTX) intermediate representation [47, 66] of GPU kernels to extract memory access patterns, control flow characteristics, and instruction mix without kernel execution. These static features, combined with a calibrated hybrid performance-power model, predict energy-efficient thread block configurations and power limits. The framework enables developers to quickly identify optimal configurations for compute-intensive workloads. Unlike runtime-dependent approaches, our method provides energy-efficient comparisons through static parsing while delivering explainable hardware-aware guidance—advantages that black-box AI techniques typically lack.

Experimental results on multi-head attention (MHA) kernels show that ***FlipFlop* reduces up to 79% energy consumption** per token compared to NVIDIA's static occupancy-based heuristics **while achieving up to 106% throughput gains**, maintaining strict quality-of-service constraints. These findings, validated through a real-world case study with Code Llama, highlight *FlipFlop*'s ability to optimize code configurations for production LLMs and demonstrate practical benefits for AI-enabled software engineering workflows.

The study makes the following key **contributions**.

- A lightweight static analysis framework that predicts energy-efficient GPU kernel configurations without their exhaustive runtime execution, reducing significant profiling overhead.
- A hybrid performance-power model integrating PTX-level code analysis with hardware calibration to recommend optimal thread block shapes and power limits.
- Explainable optimization guidance for developers, addressing memory access efficiency and power scaling challenges in LLM inference kernels.
- Validation on MHA kernels and a Code Llama case study, achieving up to 79% energy savings and 105% throughput gains in production settings.

We have made **replication package** publicly available [4].

2 Background & Related Work

Transformer Architectures & LLM Computation. Modern large language models (LLMs) such as LLAMA-3 [16] rely on transformer architectures, where Multi-Head Attention (MHA) mechanisms capture contextual dependencies through learned projections of queries, keys, and values via matrix multiplications (e.g., QK^T, where Q and K are Query and Key matrices) and softmax operations [60]. These operations scale quadratically with sequence length, demanding high memory bandwidth. Autoregressive decoding, repeating computations for each token, creates irregular workloads with partial matrix-vector operations and dynamic parallelism constraints, making MHA kernels both performance-critical and energy-intensive in LLM pipelines.

GPU Architecture & CUDA Programming Model. Modern GPUs for deep learning feature thousands of cores organized into Streaming Multiprocessors (SMs) [14], each with compute units (FP32/FP64 cores, integer ALUs, tensor cores), a memory hierarchy (registers, shared memory, L1/L2 caches, global memory), and warp schedulers managing instruction dispatch [13]. CUDA organizes execution into threads, warps (32 threads executing in lockstep), thread blocks (groups of warps sharing SM resources), and grids (sets of blocks). A *kernel*—a parallel function executing across these units—implements core computational routines closer to the specialized hardware [62]. Thread block configuration including both size (total threads) and shape (dimensional arrangement—*e.g.,* 32×4 vs 16×8), impact memory coalescing (combining adjacent threads' memory accesses into fewer transactions), SM occupancy (active warps per SM), and power draw. CUDA enables developers to implement custom high-performance operations that frameworks such as PyTorch cannot automatically generate. However, this flexibility necessitates careful kernel tuning to ensure hardware resources are utilized optimally, as suboptimal thread block shapes degrades performance [43]—creating multidimensional energy optimization challenges that demand specialized solutions.

Performance & Power Modeling. LLM computational demands grow exponentially with parameter count, while hardware power constraints remain fixed [36]. Model-level optimizations like quantization, pruning, and FlashAttention reduce complexity and data movement [10, 71], while hardware techniques include predictive power modeling [3], adaptive power capping [69, 70]. However, traditional GPU optimization approaches suffer from three key limitations. First, simplified performance models such as NVIDIA's Occupancy Calculator [41] prioritize thread parallelism while overlooking power dynamics and memory hierarchy effects, affecting energy efficiency. Crucially, these tools require runtime measurements for accurate recommendations and fail to account for thread block shape effects. Second, post-hoc profiling methods, such as Boughzala et al.'s [6], rely on simplified simulations that fail to capture complex software-hardware interactions, limiting generalizability across workloads. Third, while existing analytical models such as GPUWattch [31] and the integrated power-performance model of Hong and Kim [22] provide cycle-level power estimates, their assumptions limit their accuracy and applicability on modern, heterogeneous workloads. These performance models and simulations typically ignore the importance of thread block shapes (*i.e.,*

kernel launch parameters or thread block configurations), a factor that significantly influences both energy consumption and performance [7]. Hong and Kim's memory/compute warp parallelism (MWP/CWP) framework [21] quantifies how effectively thread warps overlap memory operations or saturate compute pipelines. Recent extensions by Alavani *et al.* [2] use static analysis of PTX instructions to train ML models for program-level post-execution energy estimation; however, their approach uses total thread counts and cannot capture block shape variations that cause significant energy differences [58], and requires extensive hardware-specific training data. Cycle-accurate simulators such as *GPUWattch* [31] and *AccelWatch* [27] provide microarchitectural details, but their computational overhead is prohibitive for exploring the vast configuration space of transformer kernels.

GPU Kernel Optimization. Kernel optimization frameworks address energy-throughput trade-offs through configuration searches. For example, Kernel Tuner [54] uses runtime profiling to find optimal configurations, while KLARAPTOR [7] predicts parameters at runtime using analytical surrogates. Lou & Muller [33] apply static analysis for CUDA kernel optimization but focus solely on performance without energy considerations, while Lim *et al.* [32] use simulation-based static analysis without power modeling or shape sensitivity. Boughzala *et al.*'s approach [6] focuses on thread block counts, ignoring shape effects and micro-architectural bottlenecks such as memory divergence, bank conflicts, and synchronization delays, which critically affect energy efficiency in GPU workloads.

GPU Execution Challenges. Transformer workloads on GPUs face three challenges. First, memory coalescing efficiency [43] degrades due to non-contiguous access patterns in attention computations, increasing memory transaction costs. Second, maximizing SM occupancy through thread parallelism often conflicts with energy efficiency due to resource contention and sublinear power scaling [55]. Third, variable memory efficiency in attention kernels requires hybrid models to account for partial coalescing and warp scheduling dynamics, complicating traditional heuristic-based optimizations.

Comparison with existing work. Unlike existing approaches that rely on exhaustive runtime profiling or simplified models, *FlipFlop leverages static PTX analysis to predict energy-efficient thread block configurations without kernel execution*, reducing profiling overhead. It uniquely integrates thread block shape optimization with hardware-calibrated power modeling, addressing memory coalescing and power scaling challenges. This enables software developers to optimize kernel programs during design, offering explainable guidance that enhances energy efficiency across diverse GPU architectures.

Table 1 systematically compares FlipFlop with prior GPU energy optimization approaches across multiple dimensions: granularity, execution stage (pre-execution, runtime, or post-execution analysis), energy awareness, block shape sensitivity, explicit power modeling, pre-execution availability, and profiling requirements. We organize existing work into five categories based on their primary methodology: static analysis for GPUs, static analysis in other domains, analytical and simulation models, framework and runtime systems, and profiling-based autotuners. FlipFlop is distinguished by its unique combination of kernel-level granularity with pre-execution static analysis that simultaneously addresses energy optimization, block shape effects, and power modeling without requiring exhaustive profiling—capabilities not collectively present in any single prior approach. Notably, compilation frameworks like PyTorch Inductor [49] and TensorRT [39] operate at graph-level optimization and are complementary to our kernel-level approach: they optimize model structure while FlipFlop optimizes the launch configurations of kernels that these frameworks execute.

Table 1: Comparison of FlipFlop with Prior GPU Energy Optimization Approaches. E=Energy-aware, S=Shape sensitivity, Pw=Power modeling, Pr=Pre-execution, Pf=Profile-free.

Approach	Granularity	E	S	Pw	Pr	Pf	Type
FlipFlop	**Kernel**	✓	✓	✓	✓	✓	**Static**
Static Analysis for GPUs							
Alavani et al.[3]	Program	✓	✗	✗	✗	✗	Static+ML
Lou & Muller[33]	Kernel	✗	✗	✗	✓	✓	Static
Lim et al.[32]	Kernel	✗	✗	✗	✗	✗	Static+Sim
Other Domain Static							
Marantos et al.[37]	Program	✓	✗	✗	✓	✓	Static
Bangash et al.[5]	Program	✓	✗	✗	✗	✓	Static
Grech et al.[17]	Program	✓	✗	✗	✓	✓	Static
Analytical & Simulation							
Hong & Kim[21]	Kernel	✓	✗	✓	✓	✓	Analytical
GPUWattch[31]	Kernel	✓	✗	✓	✗	✗	Simulator
AccelWattch[27]	Kernel	✓	✗	✓	✗	✗	Simulator
Boughzala et al.[6]	Kernel	✓	✗	✗	✗	✓	Static+Sim
Frameworks & Runtime							
PyTorch Inductor [49]	Framework	✗	✗	✗	✗	✗	Dynamic
TensorRT [39]	Framework	✗	✗	✗	✗	✗	Dynamic
NVIDIA Occ. Cal.[41]	Kernel	✗	✗	✗	✓	✓	Heuristic
Zeus[69]	Framework	✓	✗	✓	✗	✗	Dynamic
Zamani et al.[70]	System	✓	✗	✓	✗	✗	Dynamic
Profiling Autotuners							
Kernel Tuner[58]	Kernel	✗	✓	✗	✗	✗	Profiling
KLARAPTOR[7]	Kernel	✗	✓	✗	✗	✗	Analytical
CLTune[38]	Kernel	✗	✓	✗	✗	✗	Profiling

3 Overview

This study aims to achieve the following three objectives: (1) identify energy-efficient GPU thread block configurations and power limits that maximize throughput, (2) develop a static analysis method to predict energy-optimal configurations, and (3) quantify reductions in developer time, computational resources, and carbon footprint achieved through static optimization. We address these objectives through three research questions (RQs), a unified experimental setup and a case study, providing actionable insights for software engineers developing energy-efficient AI systems.

3.1 Research Questions

RQ1: *How does tuning* GPU *kernel thread block configurations and power limits dynamically affect energy efficiency and throughput of* LLM *inference, and what implications do these effects have for software engineering practices in AI system development?*

Software developers typically rely on default or heuristic-driven thread block configurations when writing CUDA kernels due to limited hardware expertise and abstraction barriers imposed by existing frameworks. In AI systems, especially LLMs, this challenge

becomes acute due to highly variable sequence lengths during inference, each demanding specific kernel launch configurations for optimal performance. Multi-head attention (MHA) kernels represent a critical optimization target as they dominate computational costs in transformer architectures. Additionally, GPU power limits, which directly influence energy consumption, are usually set to vendor-provided defaults. Without proper tooling and awareness, software engineering teams inadvertently accept these defaults, potentially leading to substantial energy wastage. **RQ1** investigates how jointly adjusting thread block dimensions and GPU power limits optimizes energy efficiency and throughput for compute-intensive MHA kernels.

RQ2: *Can static analysis predict optimal configurations that match dynamically-tuned energy optima?*

Current optimization approaches require resource-intensive runtime profiling, creating barriers to energy-aware development. **RQ2** evaluates whether static analysis of PTX intermediate representations can accurately predict thread block configurations achieving energy efficiency comparable to dynamic tuning. This capability would enable developers to identify near-optimal configurations during design rather than through post-implementation profiling.

RQ3: *How much developer effort and computational resources does static shape optimization save compared to profiling-based approaches?*

Kernel optimization through exhaustive profiling creates substantial practical barriers: consuming significant time (hours to days), wasting computational resources, and increasing the carbon footprint. Provided static optimizations reduce adoption barriers (explored in RQ2), **RQ3** quantifies the savings in compute and time.

3.2 Experimental Setup

Our experiments used NVIDIA RTX 5000 Ada (24GB) and RTX 3070 Ampere (8GB) GPUs with AMD EPYC 9554P 64-core CPU, 1TB DDR4-3200 RAM, and 512MB L3 cache. Power was monitored via NVIDIA Management Library (NVML) [45] on Ubuntu 20.04 with CUDA 12.4, Python 3.8, and modified `llama2.c` [28] framework for CodeLLAMA [51]. Profiling employed Nsight Compute [41], Kernel Tuner [58] with custom energy observers [30], and `pycuda` for GPU queries. SM utilization was tracked via CUDA Performance Tools Interface (CUPTI) [44]. Each configuration ran ten independent trials for stability. GPU power limits were adjusted in 25W increments from 100W to 250W (TDP).

Kernels and Workloads: We utilized diverse kernels representing critical AI computational patterns. General-purpose benchmarks from HeCBench [72]—vector addition, matrix multiplication, convolution, reduction, scalar product, and transpose—serve as fundamental building blocks for AI workloads. LLM kernels focusing on multi-head attention (MHA) operations from LLAMA [48] exhibit quadratic complexity ($O(L^2)$ for sequence length L), making them key targets for energy optimization. Specialized kernels, such as 3D Laplace solvers and custom attention kernels, address scientific computing and emerging AI workloads.

For each experiment, we generated valid thread block configurations by enumerating combinations of `block_size_x` and `block_size_y` (14 values: 1, 2, 4, ..., 1024) and power limits (10 values: 100W to 250W). We filtered these to ensure total threads per block (`block_size_x` × `block_size_y`) ranges from 32 to 1024 and is divisible by 32, adhering to NVIDIA GPU warp-alignment constraints. This yields a subset of valid configurations per sequence length (*e.g.,* 121 for RQ1, 66 for RQ3), with example factorizations such as [1×32], [2×16] for 32 threads and [1×1024], [1024×1] for 1024 threads, enabling rigorous evaluation of geometry effects and efficient optimization.

4 RQ1: Tuning Thread Blocks and Power Limits

Methods—RQ1. We evaluate MHA kernels across sequence lengths $\{128, 256, \ldots 8192\}$. We test all valid thread block configurations with x- and y-dimensions, each drawn from $\{1, 2, 4, \ldots 1024\}$, constrained so that the total number of threads ranges from 32 to 1024 and is divisible by 32, yielding 121 configurations. Additionally, we assess seven power limits (100W-250W, at 25W increments), adjusted dynamically using NVML [45], to explore energy-performance trade-offs in realistic LLM inference settings.

To emulate real-world LLM inference, we create a partial-decoding loop where the MHA kernel is launched repeatedly as tokens are generated. For each decoding step, we exhaustively evaluate all valid block configurations and power limit combinations to identify the most energy-efficient setup. We measure energy usage, execution time, and throughput (tokens/s) over ten repeated runs for reliability. This process iterates across multiple decoding steps to improve measurement accuracy.

Data collection and metrics. We evaluate energy efficiency in terms of *Joules per token* (J/token) and assess computational performance through throughput (tokens/s) and latency per token measurements. During each kernel launch, we track GPU instantaneous power through NVML sampling at 1ms intervals. We integrate these measurements over kernel execution duration to compute total energy consumption, then normalize by processed tokens to obtain *Joules per token*:

$$\text{Joules/token} = \frac{\sum_{\text{samples}} P(t_i) \cdot \Delta t}{\text{batch size} \times \text{L} \times N_{\text{runs}}}$$

where $P(t_i)$ is instantaneous power and $N_{\text{runs}} = 10$ ensures measurement stability. CUPTI validates SM occupancy and warp scheduling efficiency through the `sm_warps_active` metric.

Additionally, we measure SM efficiency as the ratio of average active warps to the GPU's maximum supported active warps, providing insights into hardware utilization beyond traditional occupancy metrics. To contextualize findings, we benchmark results against static heuristic configurations from NVIDIA's Occupancy Calculator operating at default power limits.

Results—RQ1. Figure 1 displays energy consumed per token (J/token) as a function of block configuration. A horizontal line marks baseline energy from NVIDIA's Occupancy Calculator at default power (250W). The x-axis is organized by total threads per block in increasing order, then by increasing thread block x-dimension (*e.g.,* for 32 threads: (1×32), (2×16), (4×8), etc.). Different sequence lengths are shown using different line colors to capture how workload size impacts energy efficiency.

Comparison with Static Heuristics. Figure 1 depicts the occupancy-based heuristic for reference. At sequence length 128, the Occupancy Calculator recommends a 512-thread configuration (*e.g.,* `16x32`), yielding 100% theoretical occupancy. However, our adaptive search finds a shape using about $3.24\mu J$ *less* energy per token

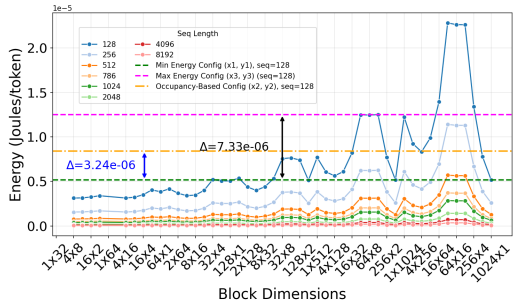

Figure 1: Energy per token vs. block dimensions. Each line represents a sequence length. The horizontal line shows the consumed energy of occupancy-based configurations

compared to that occupancy-based choice and can save up to $7.33\mu J$ relative to the *least* efficient shape tested. **Hence, while occupancy maximization may be computationally appealing, it does not reliably provide the lowest energy per token in practice.**

Per-Sequence Length Analysis. Table 2 provides detailed insights into energy efficiency across sequence lengths from 128 to 8,192, showcasing exhaustive tuning impacts on block configurations and power limits. Critically, these results demonstrate that optimal configurations fundamentally depend on both input size (sequence length) and thread block shape—not just total thread count. At sequence length 512, the optimal configuration at 100W with 2×32 block shape achieves 0.88×10^{-6} J/token, a 78.9% reduction from the 250W baseline of 3.87×10^{-6} J/token, while maintaining throughput of 1.24×10^{5} tokens/s. Similarly, at 256, the best configuration at 100W with 2×32 shape yields 1.63×10^{-6} J/token, saving 3.39×10^{-6} J/token (67.5% reduction) with throughput boost to 0.62×10^{5} tokens/s. Lower power limits (100–165W) consistently outperform the 250W baseline, with energy savings ranging from 8.3% at 128 to 78.9% at 512, demonstrating the efficacy of shape-aware optimization over static thread count maximization or occupancy-based heuristics. Although absolute energy savings vary with sequence length, the trend is clear: jointly adapting GPU power cap and block shape realizes substantial energy savings without compromising throughput. **These findings indicate that accurately predicting energy-efficient configurations requires modeling the interplay between input size, block shape dimensions, and memory access patterns**—parameters that collectively determine kernel behavior under varying workload conditions.

Table 2: Optimal Configurations per Sequence Length vs. Baseline. $\Delta E = (E_{\text{base}} - E_{\text{opt}})/E_{\text{base}} \times 100\%$; $\Delta\text{Thr} = (\text{Thr}_{\text{opt}} - \text{Thr}_{\text{base}})/\text{Thr}_{\text{base}} \times 100\%$. **Rounded mean of 10 runs($\sigma < 5\%$).**

Seq. Len	Power (W)	Block (x×y)	J/tok (J/tok)	Thr. (tok/s)	Base. (J/tok)	Δ E (%)	Δ Thr. (±%)
128	135	8×8	3.10×10^{-6}	0.33×10^{5}	3.38×10^{-6}	8.3	-1.3
256	100	2×32	1.63×10^{-6}	0.62×10^{5}	5.03×10^{-6}	67.5	+143.2
512	100	2×32	0.88×10^{-6}	1.24×10^{5}	3.87×10^{-6}	78.9	+124.1
786	135	8×8	0.51×10^{-6}	2.02×10^{5}	0.69×10^{-6}	26.7	+35.0
1024	100	2×32	0.41×10^{-6}	2.48×10^{5}	1.25×10^{-6}	67.4	+175.3
2048	165	4×16	0.12×10^{-6}	5.49×10^{5}	0.25×10^{-6}	22.3	+9.4
4096	100	2×32	0.12×10^{-6}	9.90×10^{5}	0.40×10^{-6}	74.7	+302.0
8192	100	2×32	0.06×10^{-6}	19.8×10^{5}	0.08×10^{-6}	38.5	+58.5

Understanding the Energy-Runtime-Power Relationship. Having established that joint optimization yields substantial energy savings (Table 2), we investigate the underlying relationship between energy consumption, execution time, and power draw. To rigorously examine this relationship, we conduct an expanded evaluation across 464 unique thread block configurations (ranging from 1×1 to 1024×1024 threads) tested at 8 sequence lengths without power limits, producing 11,040 measurements. This broader exploration examines whether runtime alone is sufficient to predict energy, or if additional factors need to be considered. We examine energy-runtime correlation across three progressively broader configuration subsets. **Our correlation analysis reveals a critical insight.** First, within optimal configurations (Table 2), we observe strong correlation ($R^2 = 0.91$, $p < 0.001$), where 91% of energy variance correlates with execution time. Second, examining 1,601 pairwise comparisons between baseline and candidate configurations reveals substantially weaker correlation ($r = 0.46$, $R^2 = 0.21$, $p < 0.001$), with 78.8% unexplained variance—runtime explains less than one-quarter of energy variation. Third, across the *complete* configuration space of all 11,040 measurements, correlation becomes *negative* ($r = -0.17$, $p < 0.001$), where poor configurations exhibit both high runtime *and* high power simultaneously. This three-level analysis demonstrates that while optimal configurations exhibit predictable energy-time relationships, *discovering* these configurations from the exhaustive space requires explicit consideration of factors beyond runtime. **Power draw variability** explains why runtime alone fails. Power draw exhibits substantial variability: up to 5.11× across sequence lengths and 2.07× among thread block configurations. This variation stems from three architectural mechanisms: coalesced memory accesses minimize power consumption by enhancing memory efficiency [9, 42], warp divergence and resource contention introduce power fluctuations uncorrelated with throughput due to execution unit idleness [27, 52], and dynamic voltage-frequency scaling induces power shifts as the GPU adjusts to workload intensities [18]. These mechanisms produce complex power-configuration interactions invisible to runtime-only approaches. **Greenup-Speedup analysis** [1] quantifies how energy efficiency diverges from performance. Analysis across 1,601 configuration comparisons reveals statistically significant divergence: mean 9.64% (range 0%–51.70%), with 39.7% of cases exceeding 5% divergence ($p < 0.001$) and 31.2% exceeding 10%. Critically, 18.0% of cases exhibit 20–50% divergence, with extreme cases reaching 51.70% where configurations achieve 3.85× speedup but 2.07× powerup, yielding only 1.86× greenup. This demonstrates that power varies independently of runtime, creating energy-saving opportunities invisible to time-only optimization. **Implications for optimization strategy.** The statistically significant degradation from strong correlation in optimal configurations ($R^2 = 0.91$) to moderate in pairwise comparisons ($R^2 = 0.21$) to negative in complete space ($r = -0.17$, all $p < 0.001$) demonstrates that energy-optimal configurations cannot be reliably identified through runtime analysis alone. The 78.8% unexplained variance quantifies the extent to which time-only approaches fail to capture critical power dynamics. Crucially, while runtime correlates strongly with energy for *known* optimal configurations (91% variance explained), one cannot use runtime correlation to *prospectively identify* which configurations will be optimal from thousands of candidates, where 39.7% of configuration

pairs show substantial divergence between energy and performance improvements. Our joint tuning of power limits and thread block configurations addresses this limitation, enabling navigation from the chaotic complete space ($r = -0.17$) to optimal configurations. The prevalence of substantial divergence (18.0% of cases exceeding 20%), wide power variation (up to 2.07×), and minimal variance explained by runtime alone (21%) demonstrates that energy-aware GPU optimization must treat power and time as coupled but distinct objectives.

We summarize **key observations** from the results below.

- *Non-monotonic trend:* Developers often expect higher thread block dimensions to reduce energy usage by maximizing SM occupancy, as occupancy-based studies suggest that increased parallelism hides memory latency and improves efficiency [61]. However, our results show that best energy efficiency often occurs at moderate thread counts (*e.g.,* 128–384 threads per block), suggesting configurations maximizing occupancy may not optimally exploit other resources such as shared memory or registers.
- *Adaptive advantage:* Comparing our adaptive search with NVIDIA's Occupancy Calculator (Figure 1), the occupancy-based approach (typically recommending high thread counts, *e.g.,* 256 threads per block) does not lie on the Pareto front. Instead, adaptive search frequently discovers configurations that both lower energy by 15–20% per token(representing typical mid-range savings across sequence lengths when comparing our adaptive optima against occupancy baselines in Figure 1) and improve FLOPS/Watt.
- *Throughput and utilization:* Despite lower SM occupancy at times, thread block configurations found by our adaptive approach achieve higher overall energy efficiency, indicating that maximizing occupancy does not necessarily translate into optimal energy usage.

Answer to RQ1. Yes—jointly tuning power limits and thread block configurations reveals energy-optimal operating points invisible to either parameter alone. Across sequence lengths 128–8,192, adaptive configurations (typically 100–165W with geometry-specific block shapes) reduce energy per token by up to 79% versus occupancy-based heuristics at default power, while maintaining >95% throughput. Optimal configurations depend fundamentally on both input size and thread block shape dimensions—not just total thread count—with counterintuitive patterns such as 2×32 at 100W achieving 78.9% energy savings at seq_len=512 versus occupancy's 512-thread recommendation. Statistical analysis (Section 4) validates necessity: energy-runtime correlation degrades from $R^2 = 0.91$ in optimal configs to $R^2 = 0.21$ in pairwise comparisons to $r = -0.17$ across complete design space ($p < 0.001$), with 39.7% of configuration pairs showing >5% Greenup-Speedup divergence—demonstrating that runtime-only approaches systematically fail to identify energy-efficient configurations. However, exhaustive profiling of such joint search spaces remains prohibitively expensive, highlighting the need for predictive approaches.

5 RQ2: Predict and Optimize Power and Energy

Methods—RQ2. To enable developers to create energy-efficient GPU applications without extensive runtime profiling, we developed the *FlipFlop* framework. The framework predicts optimal kernel configurations using static analysis of PTX representation of GPU kernel code, tackling the challenge of balancing energy consumption and performance in GPU-based AI software systems where power efficiency is critical. By analyzing PTX and combining it with hardware-specific calibration, *FlipFlop* abstracts complex hardware details, allowing developers to optimize energy usage with minimal hardware expertise.

Algorithm 1 FlipFlop Energy Prediction Model

Require: PTX code, GPU architecture $\mathcal{A}$, configuration $C = (\texttt{block\_x}, \texttt{block\_y}, P_{\text{cap}})$,
input resources $\mathcal{R}$ (shared_mem, grid_dims)
Ensure: Predicted energy E_{pred}
// **Calibration Phase** $\{\beta_u, L^{\text{coal}}_{\text{mem}}, \ldots\}$
// **Phase 1: Static Feature Extraction**
$\mathcal{F} \leftarrow$ ExtractFeatures(PTX, $C, \mathcal{R}$)
$\mathcal{F}.\eta_{\text{coal}} \leftarrow \min\left(1, \frac{C.\texttt{block\_x}}{32}\right) \times \frac{\text{aligned accesses}}{\text{total accesses}}$ ▷ Eq. (1)
$\mathcal{F}.N_{\text{mem}} \leftarrow$ count(load/store instructions)
$\mathcal{F}.N_{\text{comp}} \leftarrow$ count(FP32/INT/SFU instructions)
$\mathcal{F}.N_{\text{sync}} \leftarrow$ count(. sync instructions)
$\mathcal{F}.\texttt{warps} \leftarrow \frac{C.\texttt{block\_x} \times C.\texttt{block\_y}}{32}$
$\mathcal{F}.\texttt{blocks\_per\_SM} \leftarrow \min\left(\frac{\texttt{MAX\_WARPS}}{\mathcal{F}.\texttt{warps}}, \frac{\texttt{MAX\_SHARED}}{\mathcal{R}.\texttt{shared\_mem}}\right)$ ▷ Input-dependent occupancy
// **Phase 2: Execution Time Estimation**
$T_{\text{exec}} \leftarrow 0$
$\texttt{MWP} \leftarrow L^{\text{coal}}_{\text{mem}}/\texttt{departure\_delay}$ ▷ Hong-Kim model [21]
$\texttt{CWP} \leftarrow (\texttt{cycles}_{\text{mem}} + \texttt{cycles}_{\text{comp}})/\texttt{cycles}_{\text{comp}}$
$BW_{\text{eff}} \leftarrow BW_{\max} \times \mathcal{F}.\eta_{\text{coal}}$
$T_{\text{mem}} \leftarrow \mathcal{F}.N_{\text{mem}}/(\texttt{MWP} \times BW_{\text{eff}})$
$T_{\text{comp}} \leftarrow \mathcal{F}.N_{\text{comp}}/(\texttt{CWP} \times \texttt{IPC})$
$T_{\text{sync}} \leftarrow \mathcal{F}.N_{\text{sync}} \times t_{\texttt{barrier}}$
$T_{\text{exec}} \leftarrow \alpha T_{\text{mem}} + \beta T_{\text{comp}} + \gamma T_{\text{sync}} + T_{\text{base}}$ ▷ Eq. (2)
// **Phase 3: Dynamic Power Estimation**
$P_{\text{dyn}} \leftarrow 0$
for each each unit $u \in \{\texttt{FP32}, \texttt{INT}, \texttt{SFU}, \texttt{ALU}, \texttt{Mem}\}$ **do**
  $\texttt{AR}_u \leftarrow \mathcal{F}.\texttt{inst\_count}(u) \times \frac{\texttt{warps\_per\_SM}}{\texttt{exec\_cycles/issue\_cycles}}$
  $P_{\text{dyn}} \leftarrow P_{\text{dyn}} + \beta_u \times \texttt{AR}_u$ ▷ β_u from calibration
end for each
$\texttt{CI} \leftarrow \mathcal{F}.N_{\text{comp}}/\mathcal{F}.N_{\text{mem}}$ ▷ Compute intensity
$P_{\text{shape}} \leftarrow P_{\text{base}} \times \left(1 + \frac{\kappa\left|\ln\left(\frac{C.\texttt{block\_x}}{C.\texttt{block\_y}}\right)\right|}{1+\texttt{CI}}\right)$ ▷ Eq. (4)
$P_{\text{mem}} \leftarrow P^{\text{base}}_{\text{mem}} \times (1 + \lambda(1 - \mathcal{F}.\eta_{\texttt{coal}}))$ ▷ Eq. (5)
$P_{\text{dyn}} \leftarrow P_{\text{dyn}} + P_{\text{shape}} + P_{\text{mem}}$
$n \leftarrow$ estimate_active_SMs($C, \mathcal{A}$)
$P_{\text{SM}} \leftarrow \alpha n^{\beta} + \delta$ ▷ Concurrency scaling
$P_{\text{dyn}} \leftarrow P_{\text{dyn}} + P_{\text{SM}}$
if $T_{\text{exec}} < \tau_{\text{short}}$ **then**
  $P_{\text{dyn}} \leftarrow P_{\text{dyn}} \times r$ ▷ Transient correction
end if
// **Phase 4: Energy Prediction**
$f_{\text{adj}} \leftarrow f_{\text{base}} \times \left(\frac{C.P_{\text{cap}}}{P_{\text{TDP}}}\right)^{1/3}$ ▷ DVFS adjustment
$P_{\text{dyn}} \leftarrow P_{\text{dyn}} \times \frac{f_{\text{adj}}}{f_{\text{base}}}$
$E_{\text{pred}} \leftarrow T_{\text{exec}} \times (P_{\text{dyn}} + P_{\texttt{static}}) + E_{\texttt{overhead}}$
return E_{pred}

Algorithm 2 Configuration Space Exploration

Require: PTX code, GPU architecture $\mathcal{A}$, input dimensions $\mathcal{I}$ (seq_len, batch, heads)
Ensure: Pareto-optimal configurations $\mathcal{P}$
1: $\mathcal{P} \leftarrow \emptyset$
2: $\mathcal{R} \leftarrow$ ComputeInputResources($\mathcal{I}$) ▷ shared_mem, grid_dims from input
3: $\mathcal{S} \leftarrow$ GenerateValidConfigs($\mathcal{A}, \mathcal{R}$) ▷ HW + input constraints
4: **for each each** configuration $C \in \mathcal{S}$ **do**
5: $E_{\text{pred}} \leftarrow$ PredictEnergy(PTX, $\mathcal{A}, C$)
6: $T_{\text{pred}} \leftarrow$ EstimateExecutionTime($\cdots$) ▷ From Phase 2
7: **if** $\nexists C' \in \mathcal{P}$ s.t. $E'_{\text{pred}} \leq E_{\text{pred}} \wedge T'_{\text{pred}} \geq \rho T_{\text{peak}}$ **then**
8: $\mathcal{P} \leftarrow \mathcal{P} \cup \{(C, E_{\text{pred}}, T_{\text{pred}})\}$
9: **end if**
10: **end for each**
11: **return** $\mathcal{P}$

The *FlipFlop* framework employs a four-phase pipeline (Figure 2) to predict energy-efficient GPU kernel configurations, addressing three key challenges: predicting thread block geometry and power limit interactions during design, maintaining accuracy without costly runtime measurements, and generalizing optimizations across diverse GPU architectures. We elaborate on the pipeline and its key components in the rest of the section.

The framework integrates input size throughout the pipeline via `ComputeInputResources` (Algorithm 2, line 2), which computes kernel-specific resource requirements that directly constrain configuration validity. Shared memory requirements scale with input dimensions—for instance, our MHA kernel implementation requires

Algorithm 3 Architecture Calibration

Require: GPU $\mathcal{A}$
Ensure: Calibrated parameters
1: // Power characterization via microbenchmarks
2: **for each** each unit $u \in$ {FP32, INT, Mem, ...} **do**
3: $\beta_u \leftarrow \frac{P_{\text{saturated}} - P_{\text{idle}}}{\text{max operations/s}}$ ▷ Unit power coefficient
4: **end for each**
5: // Memory profiling
6: $L_{\text{mem}}^{\text{coal}} \leftarrow$ measure_latency(stride=1) ▷ Coalesced
7: $L_{\text{mem}}^{\text{uncoal}} \leftarrow$ measure_latency(stride=128) ▷ Uncoalesced
8: // Concurrency scaling
9: **for each** $n \leftarrow 1$ to max SMs **do**
10: $P_{\text{SM}}(n) \leftarrow$ measure_power(grid_size = n)
11: $\alpha_c, \beta_c, \delta_c \leftarrow \text{fit}(P_{\text{SM}} = \alpha_c n^{\beta_c} + \delta_c)$
12: **end for each**
13: // Transient effects
14: $P_{\text{sustained}} \leftarrow$ measure($n = 100$)
15: $P_{\text{short}} \leftarrow$ measure(single kernel)
16: $r \leftarrow P_{\text{short}}/P_{\text{sustained}}$
17: // Shape/coalescing coefficients
18: $\kappa, \lambda \leftarrow$ fit via aspect ratio & stride sweeps
return $\{\beta_u, L_{\text{mem}}^{\text{coal}}, \ldots\}$

shared memory proportional to both the head dimension and sequence length—directly limiting blocks-per-SM as inputs grow. Grid dimensions similarly scale with workload geometry, determining total compute demand. The `GenerateValidConfigs` function (line 3) filters configurations against both hardware limits (registers, warp count) and these input-dependent constraints, as configurations valid for short sequences may violate shared memory limits for longer inputs. This input-aware filtering propagates to execution time modeling (Algorithm 1, lines 9, 14), where occupancy modulates warp-level parallelism, capturing input size effects through hardware resource modeling.

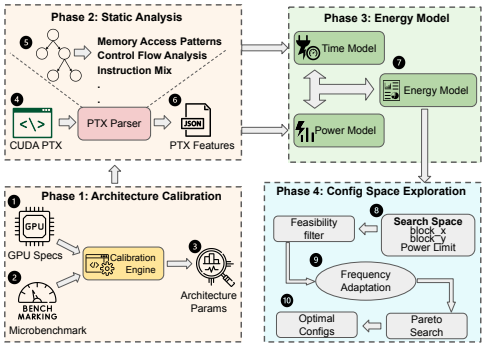

Figure 2: *FlipFlop* Architecture

Architecture-specific calibration. **To ensure accurate energy predictions across diverse** GPU **architectures, FlipFlop calibrates hardware-specific parameters, such as memory access latencies, concurrency limits, and per-instruction power consumption, through a systematic process.** This calibration abstracts low-level hardware complexities, allowing developers to reason about energy impacts without deep GPU expertise. The process involves three stages. First, power characterization dynamically adjusts GPU power limits (*e.g.,* 100W to 250W on an RTX 5000 GPU) and uses targeted microbenchmarks to isolate energy contributions from functional units, such as floating-point (FP32) cores, integer ALUs, and memory controllers. This yields power coefficients (β_u, Algorithm 3, line 3), enabling the model to adapt to hardware-specific behaviors without runtime measurements. Second, memory subsystem profiling measures latencies for coalesced (efficient) and uncoalesced (inefficient) memory accesses ($L_{\text{mem}}^{\text{coal}}$, $L_{\text{mem}}^{\text{uncoal}}$), quantifies shared memory bank conflicts [40], and profiles effective bandwidth via sustained VRAM copy operations. This translates memory access patterns into energy costs, guiding developers to optimize memory usage. Third, concurrency scaling models power consumption as a function of active streaming multiprocessors (SMs) using a power-law relationship ($P_{\text{SM}}(n) = \alpha n^{\beta} + \delta$) [55], capturing diminishing returns due to resource contention and dynamic voltage and frequency scaling (DVFS) limitations (as shown in Figure 3). Additionally, transient power effects for short-duration kernels (*e.g.,* 10μs) are quantified with a scaling factor ($r = P_{\text{short}}/P_{\text{sustained}} = 0.833$), preventing overestimation of power for bursty workloads (Algorithm 1, line 32). This comprehensive calibration ensures a portable and accurate energy model, enabling developers to focus on writing efficient code.

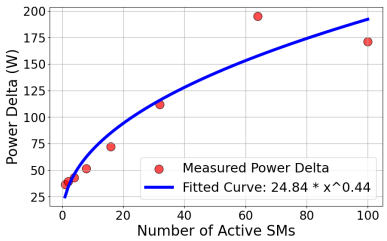

Figure 3: Power scaling vs. active SMs on RTX 5000 Ada.

Static kernel feature extraction. To predict kernel behavior before execution, *FlipFlop* analyzes PTX code. We choose PTX over SASS (hardware-specific assembly) because: (1) PTX enables architecture-agnostic analysis across NVIDIA GPU generations without retraining, whereas SASS requires per-architecture models; and (2) our calibration phase (Section 5) bridges PTX's abstraction gap by directly measuring hardware-specific behaviors (memory latencies, coalescing efficiency, power scaling) on target GPUs, capturing low-level details while maintaining portability.

The analysis proceeds in three stages. First, control flow reconstruction builds a control flow graph (CFG) [63], mapping execution paths by identifying basic blocks, branches, and loops to estimate dynamic instruction counts crucial for understanding performance and computational intensity. Second, memory access characterization classifies memory operations as coalesced or uncoalesced based on access patterns [43], computing a coalescing efficiency metric (η_{coal}, Algorithm 1, line 4) that quantifies how well memory accesses merge into fewer transactions, reducing overheads. Shared and local memory usage is analyzed via instruction pattern matching [46]. Third, computational profile estimation categorizes instructions into floating-point (FMA), integer, special function units (SFU), and ALU operations, identifying compute-intensive regions and resource demands. Loop iteration counts and resource demands (*e.g.,* registers, shared memory) are estimated from compiler outputs. *FlipFlop* determines whether a kernel is memory-bound or compute-bound by integrating these static metrics with runtime-derived Memory Warp Parallelism (MWP) and Compute Warp Parallelism (CWP) [21, 22].

Performance-Time Modeling. The performance-time model predicts kernel execution time by analyzing the interplay between memory and compute operations, helping developers understand how configuration choices, such as thread block size and power cap, impact runtime. Building on the MWP-CWP framework [21, 22], the model accounts for modern GPU features such as partial memory coalescing and concurrency constraints. It integrates static code features extracted from PTX code, a low-level GPU instruction representation (Section 5), with hardware-specific calibration data (Section 5) to compute execution time (T_{exec}) for specific configurations. Implemented in Algorithm 1, lines 14–17, the model calculates T_{exec} as a weighted sum of four components: memory access time (T_{mem}), compute time (T_{comp}), synchronization overhead (T_{sync}), and fixed kernel launch overhead (T_{base}), with weights (α, β, γ) derived from calibration to reflect kernel characteristics. Memory subsystem modeling extracts memory operations, such as load and store instructions, from PTX code [46], estimating access times based on coalescing efficiency (η_{coal}) and VRAM bandwidth, and incorporates calibrated latencies for global, shared, and local memory. Compute pipeline modeling evaluates instruction throughput for floating-point (*e.g.,* FMA), integer, special function unit (SFU), and ALU operations, factoring in instruction-level parallelism using issue cycle metrics from calibration. Concurrency constraints assess how thread block dimensions influence warp residency, balancing register usage, shared memory allocation, and thread counts, while accounting for diminishing returns from parallel streaming multiprocessor (SM) activation, as observed in calibration data (Figure 3) [15, 46]. By modeling these interactions, the framework enables developers to select configurations that optimize performance and minimize energy consumption without requiring extensive runtime testing.

Dynamic Power Modeling. The power model estimates GPU energy consumption by decomposing dynamic power into functional unit activities and static leakage [22, 25, 31] with architectural calibration and shape-aware corrections. Total power (P_{total}) is computed in Algorithm 1, lines 16–35, using calibrated power coefficients (β_u) for functional units ($\mathcal{U}$ = {FP32, INT, SFU, ALU, Mem}).

Refinements include:

- *Aspect ratio correction*: Adjusts for thread block geometry effects on memory coalescing efficiency (Algorithm 1, line 27 (A1.L27 in short)), capturing the well-documented principle that consecutive threads accessing consecutive addresses maximize memory bandwidth [43].
- *Transient power compensation*: Accounts for short-duration kernel effects using microbenchmark-derived ramp factors.
- *Coalescing-aware DRAM power*: Penalizes non-contiguous memory accesses by adjusting base memory power (A1.L28).

By separating out architectural parameters and measuring them through calibration, our model maintains portability while capturing power variance induced by thread block geometry changes.

Energy Consumption Modeling. The energy model integrates execution time and power predictions to estimate total energy consumption, validated against hardware measurements. Total energy (E_{total}) is calculated in A1.L43, as $E_{\text{total}} = T_{\text{exec}} \times (P_{\text{dyn}} + P_{\text{static}}) + E_{\text{overhead}}$, where T_{exec} is the predicted execution time, P_{dyn} is dynamic power, P_{static} is leakage power, and E_{overhead} accounts for kernel launch costs. The workflow involves three phases. Configuration space generation enumerates valid thread block dimensions (*e.g.,* 1×32 to 1024×1024), filtering out infeasible configurations based on hardware constraints like warp alignment and resource constraints (*e.g.,* register and shared memory limits). Predictive analysis computes execution time and power using the respective models, combining them to estimate energy. Empirical validation executes configurations using the Kernel Tuner framework [58], collecting actual energy measurements via NVML hardware counters and comparing them to predictions for model refinement. This process enables developers to identify Pareto-optimal configurations that balance energy efficiency and performance, streamlining the development of sustainable GPU applications.

Configuration Space Exploration. Our methodology explores the high-dimensional configuration space (*e.g.,* block_x, block_y, GPU power cap) to identify operating points that minimize energy while meeting performance constraints. Infeasible configurations are filtered based on hardware limits (*e.g.,* occupancy). Following this constraint filtering, the method adapts the GPU's operating frequency to the active power cap using a simplified voltage-frequency scaling relation. Specifically, we adjust the frequency via the equation: $f_{\text{adj}} = f_{\text{base}} \times \sqrt[K]{\frac{P_{\text{cap}}}{P_{\text{TDP}}}}$ where f_{base} represents the GPU's default clock, P_{TDP} is its thermal design power, P_{cap} is the desired power limit, and K is a fitting integer (typically set to 3) [20, 56]. This formulation is motivated by standard dynamic voltage and frequency scaling (DVFS) principles—in processors, dynamic power is proportional to $f \times V^2$ [20, 56] while the operating frequency f scales roughly linearly with voltage V [29]. Consequently, if we assume an approximate cubic relationship (i.e., $P \propto f^3$), then reducing the power cap to a fraction of P_{TDP} necessitates a corresponding scaling down in frequency. A Pareto set $\mathcal{P}$ is constructed to include configurations that reduce energy while meeting performance targets (A2.L8). This automated exploration eliminates manual tuning, providing developers with optimal configurations for energy-efficient GPU applications across diverse workloads.

In summary, this four-phase pipeline (Fig. 2), systematically predicts how kernel block dimensions and power limits jointly influence execution time and energy consumption. Phase 1 inspects PTX-level instructions to establish memory and compute characteristics. Phase 2 tailors the model to the specific hardware via microbenchmark-derived latencies and power coefficients. Phase 3 merges these inputs into coupled time and power models, capturing concurrency effects. Finally, Phase 4 explores the configuration space—pruning infeasible points and retaining only Pareto-optimal solutions—to discover an energy-optimal GPU kernel configuration.

Results—RQ2. To evaluate *FlipFlop*'s ability to predict and optimize energy consumption, we analyze its performance on the multi-head attention (MHA) kernel across sequence lengths 128–8192, validated on an RTX 5000 GPU and an additional RTX 3070 to ensure architectural portability. We also extend validation to diverse CUDA kernels (convolution, laplace3d, matMul, reduction, scalarProd, transpose, vecAdd) to confirm generalization across kernel types. As no existing approach in Table 1 provides pre-execution static energy modeling for kernel configurations, we establish a rigorous baseline using Kernel Tuner [54, 58] for exhaustive runtime

profiling and actual hardware measurements. Figure 4 illustrates predicted energy per token for the MHA kernel across block configurations, closely mirroring measured trends from RQ1 (Figure 1). For sequence length 8192, Figure 5 shows normalized predicted and actual energy values, revealing consistent local minima and maxima with an accuracy of 83% in identifying Pareto-optimal configurations (maxima and minima) across all tested kernels.

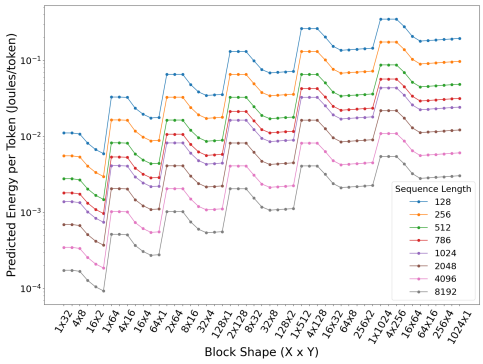

Figure 4: Predicted energy per token as a function of block configuration. Each line corresponds to a different seq. length

FlipFlop's static PTX analysis, combined with a brief calibration phase, effectively identifies inefficient block shapes, reducing energy consumption up to 20% on average compared to unconstrained/default Power, with less than 5% performance loss. Across architectures, the model maintains prediction accuracy on the RTX 3070, consistent with RTX 5000 results, confirming portability.

Adaptive power capping, refined by real-time feedback, reduces worst-case prediction errors from 10–12% to below 5% by adjusting coalescing and latency parameters when power deviates by more than 10 W. The power limit is set as `power_limit` = $\min(\text{TDP}, \alpha\hat{P} + \beta\Delta P_{\text{history}})$, where $\hat{P}$ is the predicted power from the static model and $\Delta P_{\text{history}}$ is the cumulative deviation between predicted and actual power over recent iterations. This approach ensures robust energy savings under realistic constraints, with no single block configuration optimal across all workloads, underscoring the need for multi-objective optimization.

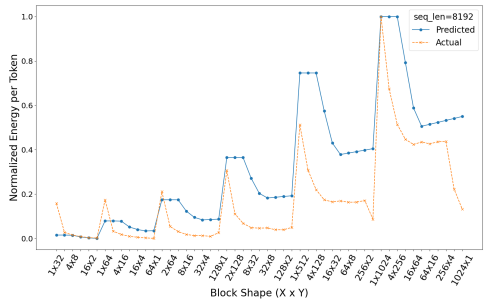

Figure 5: Trends between Actual and Predicted Energy

Statistical validation of prediction accuracy. To rigorously assess *FlipFlop*'s predictive reliability, we conduct comprehensive rank correlation analysis using Spearman's ρ (robust to non-linear relationships) and Kendall's τ (robust to outliers), following best practices for non-normally distributed data. Energy prediction achieves strong correlation across sequence lengths, *e.g.*, $\rho = 0.846$ ($p = 5.53 \times 10^{-15}$) at seq_len=128, $\rho = 0.875$ ($p = 4.69 \times 10^{-17}$) at seq_len=512, $\rho = 0.834$ ($p = 3.17 \times 10^{-14}$) at seq_len=1024, with aggregate mean $\rho = 0.857 \pm 0.019$ and Kendall $\tau = 0.653 \pm 0.028$. Power modeling exhibits consistent accuracy ($\rho = 0.840 \pm 0.031$, all $p < 10^{-11}$) and execution time prediction achieves $\rho = 0.66 \pm 0.067$ (all $p < 10^{-6}$), validating architectural generalizability. Paired t-tests confirm statistically significant agreement ($t = 12.52$, $df = 50$, $p < 10^{-15}$). Translating correlation metrics into practical performance, *FlipFlop*'s top-20 predictions capture 94% of actual top-20 energy-efficient configurations, demonstrating reliable optimization guidance without exhaustive profiling. *Comparing against Bayesian optimization*: Bayesian optimization evaluated 800 configurations yet missed optimum by 9.3%, while *FlipFlop* reaches optima after only 4.4 evaluations, demonstrating that static PTX analysis provides fundamental efficiency advantages over iterative search methods.

Ablation study validating aspect ratio correction. To validate the aspect ratio correction (Algorithm 1, line 27), we compare model performance with and without this refinement. The correction penalizes configurations deviating from memory-efficient access patterns, capturing documented GPU behavior where memory bandwidth is maximized when consecutive threads access consecutive addresses [43]. For MHA kernels, disabling the correction degrades energy prediction correlation by 83% (ρ reduction from 0.852 to 0.142), as the model loses ability to distinguish configurations with poor energy efficiency due to bank conflicts and suboptimal warp scheduling.

*Answer to RQ2. A hybrid static-and-dynamic model that combines PTX-level feature extraction can predict per-kernel energy usage to within 5–10% of measured values and enable **adaptive power capping** strategies. This yields significant energy savings (20% on average) while maintaining throughput above 90–95% of peak.*

6 RQ3: Practical Impact of Optimization

Methods—RQ3. We evaluate *FlipFlop*'s effectiveness in reducing developer effort and computational resources by comparing its static optimization approach against exhaustive profiling of the MHA kernel across sequence lengths 128–8192, with a batch size of four, and 16 heads at 256 dimensions per head. We define 66 thread block configurations per sequence length, where `block_x` and `block_y` are powers of two (1–1024), constrained by total threads (32–1024, divisible by 32) per NVIDIA warp-alignment rules, each profiled five times for statistical reliability.

Our proposed approach, *FlipFlop*, uses static analysis to recommend only Pareto-optimal configurations, minimizing profiling by recommending only high-performing setups, that balance energy and execution time, incorporating power limits (100–250W, 25W increments) to expand the search space to 528 configurations. *FlipFlop*, significantly reduces the number of configurations profiled while preserving energy-efficient performance, saving substantial time, energy, and computational resources.

We employ following metrics to quantify savings.

(1) **Configuration Reduction Ratio (CRR)**: Fraction of configurations avoided, computed as $\text{CRR} = 1 - \frac{\#\ \textit{FlipFlop}\text{-recommended configs}}{\text{Total valid configs}}$
(2) **Energy and Carbon Savings**: Avoided profiling energy and associated emissions computed as $\Delta E = \sum(E_{\text{run}} \times N_{\text{runs}})$ and $\Delta\text{CO}_2 = \Delta E \times \text{CI}_{\text{regional}}$ where E_{run} is measured energy per kernel execution and CI is carbon intensity (0.384 kg CO_2/kWh for US [24]. We choose US as it houses nearly half of the global datacenters [34]).

Results—RQ3. *Efficiency Gains. FlipFlop* significantly reduces the optimization search space by 93.4%, recommending an average of 4.4 configurations per sequence length out of the total 66 configurations, as detailed in Table 3. The reduction effectiveness varies by sequence length: at a sequence length of 128, it achieves a 98.5% reduction with only 1 configuration, while at 4096, it offers a 95.5% reduction with 3 configurations, and at 8192, a 97.0% reduction with 2 configurations. This demonstrates the approach's ability to adaptively minimize the search space across different workload sizes.

Table 3: Config Reduction Ratio (CRR) per sequence length

Seq. Length	128	256	512	786	1024	2048	4096	8192	Avg.
Total Configs	66	66	66	66	66	66	66	66	66
Recommended	1	5	6	6	6	6	3	2	**4.4**
CRR	0.985	0.924	0.909	0.909	0.909	0.909	0.955	0.970	**0.934**

Resource Savings. The configuration reduction translates into substantial resource conservation, saving 14.3 minutes of computation time, which is 172,986× faster than dynamic tuning. This efficiency also yields energy savings of 188,680 J, equivalent to 52.4 watt-hours, sufficient to charge three smartphones [12]. For context, using US coal-fired power generation as a reference point (the calculation basis for EPA equivalency metrics [12]), this energy savings corresponds to approximately 19.4g CO_2e [12] avoided, though actual emissions vary substantially by regional power mix and generation source.

Production Context for Profiling Costs. While FlipFlop's configuration space reduction saves 14.3 minutes and 188,680 J per optimization session, the true value emerges in production deployment contexts. This one-time optimization cost amortizes after approximately 150 inference runs (at 1,257 J average per run for seq_len=512). More critically, AI is projected to use 165–326 terawatt-hours annually by 2028, enough to power 22% of US households [26]. Even 1% energy inefficiency from suboptimal configurations—running continuously in production—generates ongoing waste far exceeding any profiling overhead. Unlike iterative profiling methods that risk convergence failure or require re-tuning when configuration spaces change, FlipFlop's static analysis provides deterministic near-optimal configurations without heavy kernel execution. This eliminates both the initial profiling cost and the compounding production inefficiency that runtime methods may introduce. Furthermore, when compared against non-exhaustive alternatives like Bayesian optimization (which evaluated 785 configurations yet missed the optimum by 9.3%, Section 5), FlipFlop's 392× efficiency advantage stems not from avoiding exhaustive search but from eliminating runtime profiling altogether—a fundamental difference from tools like KLARAPTOR [7] and Kernel Tuner [58] that require kernel execution even with smart search strategies.

Developer Impact. For AI developers, static optimization with *FlipFlop* fundamentally transforms kernel configuration workflows by eliminating manual tuning effort, reducing configuration evaluation by 93.4% across sequence lengths. This cuts experimentation time by 15.1×, shrinking the process from hours to minutes, and reduces optimization overhead by 95.6% in energy consumption, enabling more sustainable development practices. Additionally, it scales efficiently, maintaining consistent speedup for complex models, allowing a 100-kernel optimization to be completed in under 12 minutes.

Scaling Analysis. The resource savings demonstrated by static optimization compound substantially as problem complexity increases. Experimental projections show that optimizing a 100-kernel model would require approx. 3 hours with dynamic runtime profiling versus just minutes with *FlipFlop*. This scaling efficiency stems from *FlipFlop*'s fundamental advantage: its static analysis methodology remains computationally efficient regardless of problem scale. As context windows expand to 1M+ tokens requiring optimization across hundreds of sequence lengths, or when tuning entire models containing dozens of diverse kernels, the relative time and energy savings per kernel grow linearly with the number of configurations evaluated. For example, extending our approach to 100 sequence lengths would yield 150× time savings. This scalability ensures that the environmental benefits and developer time savings become increasingly significant when applied to complex modern architectures with massive context windows and heterogeneous kernel collections, making *FlipFlop* particularly valuable for next-generation LLMs where iterative tuning would otherwise incur prohibitive computational costs.

Answer to RQ3. *FlipFlop reduces developer effort by 93.4%, accelerates optimization by 172,986×, and saves* ***188,680J*** *(19.4g CO_2e) per workload. This transforms kernel tuning from resource-intensive profiling to efficient static analysis—enabling practical iterative optimization while reducing environmental impact.*

7 Case Study: Energy Optimization in LLM Inference

Scenario and Motivation. This case study explores how a software engineer implements CodeLlama 7B, an open-source code generation LLM, using low-level CUDA programming rather than high-level frameworks such as PyTorch. The engineer's primary motivation is maximizing performance and energy efficiency for edge devices with limited computational resources. By implementing core operations directly in CUDA, the engineer achieves fine-grained control over memory access patterns, precise hardware resource management, elimination of framework overhead, and optimal energy-per-inference metrics.

Kernel Optimization Challenge. The software engineer focuses on optimizing the MHA kernel for incrementally growing context windows, as it represents the most critical optimization target due to its substantial compute share in transformer models. Traditional tuning approaches—brute-force parameter search or runtime heuristics—prove problematic due to exponential configuration space growth, hardware-specific performance characteristics,

repeated tuning requirements for new architectures, and prohibitive energy costs during exploration. These limitations motivate the engineer adopting FlipFlop.

Methodology. The developer begins with static analysis using FlipFlop to identify promising thread configurations from the vast combinatorial space, narrowing candidates to a tractable subset for empirical validation. Using HumanEval's 164 programming problems as realistic workloads, six representative thread block configurations are selected from FlipFlop's recommendations: 1024×1 (1D arrangement), 32×32 (wide configuration), 256×4 (2D narrow pattern), 512×1 (reduced 1D layout), 2048×1 (expanded 1D structure), and 64×16 (balanced geometry), systematically exploring thread parallelism, memory access patterns, and resource utilization tradeoffs. FlipFlop's automated workflow generates specialized CUDA binaries for each configuration, creating temporary source files with configuration-specific constants, compiling distinct executables using *nvcc*, organizing outputs into timestamped directories for reproducibility, and validating functionality through prompt-based smoke tests. During benchmark execution, the system samples GPU power at 50ms intervals, records per-prompt latency, calculates energy through numerical integration, and logs latency (seconds), average power (watts), and total energy (joules). The experimental protocol implements best practices: sequential HumanEval executions avoid GPU resource contention, comprehensive process cleanup eliminates residual state, user-specific management ensures isolation, and timeout mechanisms handle failures. Each configuration processes all 164 problems, generating 26,256 individual measurements across three metrics per prompt per configuration.

Results & Discussion. The custom configuration (64×16) recommended by FlipFlop demonstrated superior energy efficiency, consuming 15.7% less energy (5510.7J vs 6536.0J) while achieving 14.0% lower latency (27.97s vs 32.51s) compared to the default configuration. This improvement stems from the 2D thread arrangement better aligning with the kernel's memory access patterns, reducing shared memory bank conflicts while improving coalesced global memory accesses. The energy-performance tradeoffs reveal important architectural insights. While power draw shows moderate variation (195.6-202.1W), latency differences drive significant energy variance due to $E = P \times t$. The 64×16 configuration's 14% latency reduction directly enabled proportional energy savings, confirming execution time optimization as critical for efficiency. Notably, the 2048×1 configuration consumed 17.6% more energy than our recommendation despite similar latency to default, demonstrating how improper thread sizing triggers suboptimal power scaling through resource contention.

This case study demonstrates that energy-aware kernel optimization delivers concrete engineering advantages. Three system-level implications emerge: (1) Static analysis reduces optimization effort by orders of magnitude, narrowing thousands of configurations to six viable candidates while identifying global optima; (2) Significant energy savings are achievable solely through thread reshaping without algorithmic modifications; (3) Future GPUs with tightening thermal limits may prioritize rapid completion over momentary power reduction, validating FlipFlop's energy modeling approach.

8 Implications

The *FlipFlop* framework enhances energy-efficient computing by using static analysis to optimize GPU kernel configurations, embedding energy efficiency as a core design principle. **Software Developers:** *FlipFlop* empowers developers to create energy-efficient GPU applications without requiring deep hardware expertise, embedding energy optimization directly into the development process. By analyzing PTX code, *FlipFlop* enables developers to identify optimal kernel configurations early. As demonstrated in Section 7, *FlipFlop* accelerates discovery of energy-optimal configurations, ensuring developers—regardless of hardware knowledge—can build efficient software pipelines that scale effectively in deployment with minimal performance overhead. **System Architects:** *FlipFlop*'s ability to predict optimal configurations without runtime profiling reduces complexity in designing portable, adaptive systems. By offering explainable guidance on resource allocation, *FlipFlop* enables architects to balance performance and energy efficiency, fostering resilient systems that adapt to evolving hardware and workload demands. **Hardware Vendors:** *FlipFlop*'s insights into memory access patterns and power dynamics inform optimizations in GPU architectures and tools such as CUDA [64] and ROCm [67]. Integrating its predictive models into development kits enhances access to energy-efficient features. **Sustainability Researchers:** By enabling proactive energy optimization without runtime costs, *FlipFlop* supports development of eco-friendly systems for AI, scientific simulations, and beyond. Its approach aligns with global efforts to minimize computing's carbon footprint, providing a replicable model for studying energy-performance trade-offs and fostering sustainable design principles across industries.

9 Threats to Validity

Internal validity risks arise from model assumptions (e.g., memory coalescing, thread block impacts) that may not hold across all GPUs or workloads, potentially skewing predictions. We mitigate this with a hybrid static-dynamic model, reducing prediction errors to below 5% via corrections. Calibration on the RTX 5000 Ada, with additional validation on an RTX 3070, ensures hardware-specific accuracy. Energy measurement inaccuracies are minimized by using NVIDIA's NVML [45] and CUPTI [44] tools and following best practices from prior work [50]. **External validity** is challenged by testing primarily on NVIDIA GPUs, which may limit generalizability to other architectures. We address this with an architecture-agnostic model validated on both RTX 5000 and RTX 3070 GPUs. Similarly, focusing on multi-head attention (MHA) kernels may not extend to all workloads, so we evaluate additional CUDA kernels (e.g., convolution, matrix multiplication, reduction) to confirm 83% accuracy in identifying Pareto-optimal configurations across diverse kernel types (Section 5).

Conclusions and Future Work

We introduce FlipFlop, a framework providing optimized GPU kernel configurations calibrated to underlying hardware through PTX-level feature extraction via static analysis and performance-power modeling. By systematically modeling memory access patterns, arithmetic intensity, and concurrency limits, our framework reveals how moderate thread block configurations can better balance

utilization and power draw than occupancy-focused heuristics. Our evaluations on LLM workloads involving multi-head attention confirm that jointly tuning kernel shapes and GPU power caps yields meaningful energy reductions without sacrificing throughput or model quality.

Looking ahead, we plan to expand the framework's applicability to broader AI and scientific computing kernels and investigate more granular power-capping strategies based on dynamic workload phases. Incorporating dynamic voltage-frequency scaling (DVFS) within our static modeling will further refine energy-saving strategies under rapidly changing load conditions typical in real-world large-scale LLM inference pipelines. Our controlled evaluation uses single-kernel execution to isolate configuration effects, reflecting common deployment patterns (dedicated GPUs, containerization) that minimize contention for predictable latency. While our calibration models resource-sharing dynamics through power-law SM scaling and bandwidth saturation, comprehensive validation under heavy multi-stream contention represents valuable future work. We also plan to develop libraries abstracting low-level optimizations—thread block tuning and power capping—while preserving developer control for fine-grained customization. These advancements will further streamline sustainable GPU software development, supporting efficient, large-scale LLM inference with convenience across diverse environments.

References

[1] Sarah Abdulsalam, Ziliang Zong, Qijun Gu, and Meikang Qiu. 2015. Using the Greenup, Powerup, and Speedup metrics to evaluate software energy efficiency. In *2015 Sixth International Green and Sustainable Computing Conference (IGSC)*. IEEE, 1–8.

[2] Gargi Alavani, Jineet Desai, Snehanshu Saha, and Santonu Sarkar. 2023. Program Analysis and Machine Learning–based Approach to Predict Power Consumption of CUDA Kernel. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems* 8 (2023), 1 – 24. https://api.semanticscholar.org/CorpusID:259065373

[3] Gargi Alavani and Santonu Sarkar. 2023. Prediction of Performance and Power Consumption of GPGPU Applications. http://arxiv.org/abs/2305.01886 arXiv:2305.01886 [cs].

[4] Anonymous Anonymous. 2025. *FlipFlop*. doi:10.5281/zenodo.16147188

[5] Abdul Ali Bangash, Kalvin Eng, Qasim Jamal, Karim Ali, and Abram Hindle. 2023. Energy consumption estimation of API-usage in smartphone apps via static analysis. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, 272–283.

[6] Dorra Boughzala, Laurent Lefèvre, and Anne-Cécile Orgerie. 2020. Predicting the Energy Consumption of CUDA Kernels using SimGrid. In *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 191–198. doi:10.1109/SBAC-PAD49847.2020.00035 ISSN: 2643-3001.

[7] Alexander Brandt, Davood Mohajerani, Marc Moreno Maza, Jeeva Paudel, and Linxiao Wang. 2019. KLARAPTOR: A Tool for Dynamically Finding Optimal Kernel Launch Parameters Targeting CUDA Programs. http://arxiv.org/abs/1911.02373 arXiv:1911.02373.

[8] Andrew A Chien, Liuzixuan Lin, Hai Nguyen, Varsha Rao, Tristan Sharma, and Rajini Wijayawardana. 2023. Reducing the Carbon Impact of Generative AI Inference (today and in 2035). In *Proceedings of the 2nd workshop on sustainable computer systems*. 1–7.

[9] Neal C Crago, Mark Stephenson, and Stephen W Keckler. 2018. Exposing memory access patterns to improve instruction and memory efficiency in GPUs. *ACM Transactions on Architecture and Code Optimization (TACO)* 15, 4 (2018), 1–23.

[10] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems* 35 (2022), 16344–16359.

[11] Alex de Vries. 2023. The growing energy footprint of artificial intelligence. *Joule* 7, 10 (2023), 2191–2194.

[12] EPA. 2025. Greenhouse Gas Equivalencies Calculator | US EPA — epa.gov. https://www.epa.gov/energy/greenhouse-gas-equivalencies-calculator. [Accessed 03-07-2025].

[13] Charles Frye and Matthew Nappo. 2025. GPU Glossary — modal.com. https://modal.com/gpu-glossary. [Accessed 15-07-2025].

[14] Charles Frye and Matthew Nappo. 2025. What is a Streaming Multiprocessor? | GPU Glossary — modal.com. https://modal.com/gpu-glossary/device-hardware/streaming-multiprocessor. [Accessed 15-07-2025].

[15] Lan Gao, Jing Wang, and Weigong Zhang. 2022. Adaptive Contention Management for Fine-Grained Synchronization on Commodity GPUs. *ACM Trans. Archit. Code Optim.* 19, 4, Article 58 (Sept. 2022), 21 pages. doi:10.1145/3547301

[16] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[17] Neville Grech, Kyriakos Georgiou, James Pallister, Steve Kerrison, Jeremy Morse, and Kerstin Eder. 2015. Static analysis of energy consumption for LLVM IR programs. In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*. 12–21.

[18] João Guerreiro, Aleksandar Ilic, Nuno Roma, and Pedro Tomás. 2019. Modeling and decoupling the GPU power consumption for cross-domain DVFS. *IEEE Transactions on Parallel and Distributed Systems* 30, 11 (2019), 2494–2506.

[19] Gianluca Guidi, Francesca Dominici, Jonathan Gilmour, Kevin Butler, Eric Bell, Scott Delaney, and Falco J. Bargagli-Stoffi. 2024. Environmental Burden of United States Data Centers in the Artificial Intelligence Era. arXiv:2411.09786 [cs.CY] https://arxiv.org/abs/2411.09786

[20] Yunchu Han, Zhaojun Nan, Sheng Zhou, and Zhisheng Niu. 2025. DVFS-Aware DNN Inference on GPUs: Latency Modeling and Performance Analysis. arXiv:2502.06295 [cs.LG] https://arxiv.org/abs/2502.06295

[21] Sunpyo Hong and Hyesoon Kim. 2009. 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*. 152–163.

[22] Sunpyo Hong and Hyesoon Kim. 2010. An integrated GPU power and performance model. In *Proceedings of the 37th annual international symposium on Computer architecture*. ACM, Saint-Malo France, 280–289. doi:10.1145/1815961.1815998

[23] IEA. 2023. Data Centres and Data Transmission Networks. https://www.iea.org/energy-system/buildings/data-centres-and-data-transmission-networks. [Accessed 17-07-2025].

[24] Ember ; Energy Institute. 2024. Carbon intensity of electricity generation — ourworldindata.org. https://ourworldindata.org/grapher/carbon-intensity-electricity?mapSelect=~USA. [Accessed 04-07-2025].

[25] C. Isci and M. Martonosi. 2003. Runtime power monitoring in high-end processors: methodology and empirical data. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36*. 93–104. doi:10.1109/MICRO.2003.1253186

[26] O'Donnell James and Crownhart Casey. 2025. We did the math on AI's energy footprint. Here's the story you haven't heard. *MIT Technology Review* (May 2025). https://www.technologyreview.com/2025/05/20/1116327/ai-energy-usage-climate-footprint-big-tech/ Accessed: 2025-11-05.

[27] Vijay Kandiah, Scott Peverelle, Mahmoud Khairy, Junrui Pan, Amogh Manjunath, Timothy G Rogers, Tor M Aamodt, and Nikos Hardavellas. 2021. AccelWattch: A power modeling framework for modern GPUs. In *MICRO-54: 54th Annual IEEE/ACM International symposium on microarchitecture*. 738–753.

[28] Andrej Karpathy. [n. d.]. GitHub - karpathy/llama2.c: Inference Llama 2 in one file of pure C — github.com. https://github.com/karpathy/llama2.c. [Accessed 09-04-2025].

[29] Ben Keller, Borivoje Nikolic, and Krste Asanović. 2015. Opportunities for fine-grained adaptive voltage scaling to improve system-level energy efficiency. *EECS Department, University of California* (2015).

[30] KernelTuner. 2025. Observers—Kernel Tuner 1.1.3 documentation. https://kerneltuner.github.io/kernel_tuner/stable/observers.html. [Accessed 17-07-2025].

[31] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: enabling energy optimizations in GPGPUs. *ACM SIGARCH Computer Architecture News* 41, 3 (June 2013), 487–498. doi:10.1145/2508148.2485964

[32] Robert Lim, Boyana Norris, and Allen Malony. 2017. Autotuning GPU kernels via static and predictive analysis. In *2017 46th international conference on parallel processing (icpp)*. IEEE, 523–532.

[33] Mark Lou and Stefan K Muller. 2024. Automatic Static Analysis-Guided Optimization of CUDA Kernels. In *Proceedings of the 15th International Workshop on Programming Models and Applications for Multicores and Manycores*. 11–21.

[34] Marcus Lu. 2025. Ranked: The Top 25 Countries With the Most Data Centers — visualcapitalist.com. https://www.visualcapitalist.com/ranked-the-top-25-countries-with-the-most-data-centers/. [Accessed 04-07-2025].

[35] Alexandra Sasha Luccioni, Emma Strubell, and Kate Crawford. 2025. From Efficiency Gains to Rebound Effects: The Problem of Jevons' Paradox in AI's Polarized Environmental Debate. In *Proceedings of the 2025 ACM Conference on Fairness, Accountability, and Transparency (FAccT '25)*. Association for Computing Machinery, New York, NY, USA, 76–88. doi:10.1145/3715275.3732007

[36] Sasha Luccioni, Yacine Jernite, and Emma Strubell. 2024. Power hungry processing: Watts driving the cost of ai deployment?. In *Proceedings of the 2024 ACM*

conference on fairness, accountability, and transparency. 85–99.

[37] Charalampos Marantos, Konstantinos Salapas, Lazaros Papadopoulos, and Dimitrios Soudris. 2021. A flexible tool for estimating applications performance and energy consumption through static analysis. *SN Computer Science* 2, 1 (2021), 21.

[38] Cedric Nugteren and Valeriu Codreanu. 2015. CLTune: A generic auto-tuner for OpenCL kernels. In *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*. IEEE, 195–202.

[39] NVIDIA. [n. d.]. NVIDIA. *NVIDIA*. ([n. d.]). https://github.com/NVIDIA/TensorRT Accessed: 2025-11-05.

[40] NVIDIA. 2013. Using Shared Memory in CUDA C/C++ | NVIDIA Technical Blog — developer.nvidia.com. https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/. [Accessed 15-04-2025].

[41] NVIDIA. 2025. 3. Nsight Compute; NsightCompute 12.8 documentation — docs.nvidia.com. https://docs.nvidia.com/nsight-compute/NsightCompute/index.html#occupancy-calculator. [Accessed 01-04-2025].

[42] NVIDIA. 2025. CUDA C++ Best Ppractice. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide. [Accessed 27-10-2025].

[43] NVIDIA. 2025. CUDA C++ Programming Guide — docs.nvidia.com. https://docs.nvidia.com/cuda/cuda-c-programming-guide/. [Accessed 12-04-2025].

[44] NVIDIA. 2025. CUPTI 2014; Cupti 12.8 documentation — docs.nvidia.com. https://docs.nvidia.com/cupti/index.html. [Accessed 06-04-2025].

[45] NVIDIA. 2025. NVIDIA Management Library (NVML) — developer.nvidia.com. https://developer.nvidia.com/management-library-nvml. [Accessed 01-04-2025].

[46] NVIDIA. 2025. PTX ISA 8.7 documentation — docs.nvidia.com. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html?highlight=sync#parallel-synchronization-and-communication-instructions. [Accessed 12-04-2025].

[47] NVIDIA. 2025. PTX ISA 8.8 documentation — docs.nvidia.com. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html. [Accessed 19-07-2025].

[48] Sang Park. 2024. llama3.cuda: pure C/CUDA implementation for Llama 3 model. https://github.com/likejazz/llama3.cuda. llama3.cuda, MIT License.

[49] PyTorch. [n. d.]. PyTorch Inductor. *PyTorch* ([n. d.]). https://docs.pytorch.org/docs/stable/torch.compiler.html Accessed: 2025-11-05.

[50] Saurabhsingh Rajput, Tim Widmayer, Ziyuan Shang, Maria Kechagia, Federica Sarro, and Tushar Sharma. 2024. Enhancing Energy-Awareness in Deep Learning through Fine-Grained Energy Measurement. *ACM Trans. Softw. Eng. Methodol.* 33, 8, Article 211 (Dec. 2024), 34 pages. doi:10.1145/3680470

[51] Roger Allen. 2024. llama2.cuda: pure C/CUDA implementation for Llama 2 model. https://github.com/rogerallen/llama2.cu. llama2.cuda, MIT License.

[52] Mohammad Sadrosadati, Seyed Borna Ehsani, Hajar Falahati, Rachata Ausavarungnirun, Arash Tavakkol, Mojtaba Abaee, Lois Orosa, Yaohua Wang, Hamid Sarbazi-Azad, and Onur Mutlu. 2019. ITAP: Idle-time-aware power management for GPU execution units. *ACM Transactions on Architecture and Code Optimization (TACO)* 16, 1 (2019), 1–26.

[53] Siddharth Samsi, Dan Zhao, Joseph McDonald, Baolin Li, Adam Michaleas, Michael Jones, William Bergeron, Jeremy Kepner, Devesh Tiwari, and Vijay Gadepally. 2023. From Words to Watts: Benchmarking the Energy Costs of Large Language Model Inference. arXiv:2310.03003 [cs.CL] https://arxiv.org/abs/2310.03003

[54] Richard Schoonhoven, Bram Veenboer, Ben van Werkhoven, and Kees Joost Batenburg. 2022. Going green: optimizing GPUs for energy efficiency through model-steered auto-tuning. doi:10.48550/arXiv.2211.07260 arXiv:2211.07260 [cs].

[55] Hossein SeyyedAghaei, Mahmood Naderan-Tahan, and Lieven Eeckhout. 2024. GPU scale-model simulation. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 1125–1140.

[56] Amit Kumar Singh, Alok Prakash, Karunakar Reddy Basireddy, Geoff V. Merrett, and Bashir M. Al-Hashimi. 2017. Energy-Efficient Run-Time Mapping and Thread Partitioning of Concurrent OpenCL Applications on CPU-GPU MPSoCs. *ACM Trans. Embed. Comput. Syst.* 16, 5s, Article 147 (Sept. 2017), 22 pages. doi:10.1145/3126548

[57] Tensorflow. 2025. TensorFlow — tensorflow.org. https://www.tensorflow.org. [Accessed 19-07-2025].

[58] Ben van Werkhoven. 2019. Kernel Tuner: A search-optimizing GPU code auto-tuner. *Future Generation Computer Systems* 90 (2019), 347–358. doi:10.1016/j.future.2018.08.004

[59] Tina Vartziotis, Ippolyti Dellatolas, George Dasoulas, Maximilian Schmidt, Florian Schneider, Tim Hoffmann, Sotirios Kotsopoulos, and Michael Keckeisen. 2024. Learn to Code Sustainably: An Empirical Study on LLM-based Green Code Generation. arXiv:2403.03344 [cs.SE] https://arxiv.org/abs/2403.03344

[60] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[61] Vasily Volkov. 2010. Better performance at lower occupancy. In *Proceedings of the GPU technology conference, GTC*, Vol. 10. San Jose, CA, 16.

[62] Wikipedia contributors. 2025. Compute kernel — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Compute_kernel&oldid=1289526415. [Online; accessed 15-July-2025].

[63] Wikipedia contributors. 2025. Control-flow graph — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Control-flow_graph&oldid=1272774587. [Online; accessed 12-April-2025].

[64] Wikipedia contributors. 2025. CUDA — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=CUDA&oldid=1298142890. [Online; accessed 30-June-2025].

[65] Wikipedia contributors. 2025. Model collapse — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Model_collapse&oldid=1295800934. [Online; accessed 17-July-2025].

[66] Wikipedia contributors. 2025. Parallel Thread Execution — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Parallel_Thread_Execution&oldid=1281527860. [Online; accessed 1-April-2025].

[67] Wikipedia contributors. 2025. ROCm — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=ROCm&oldid=1297555474. [Online; accessed 30-June-2025].

[68] Wikipedia contributors. 2025. SYCL — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=SYCL&oldid=1295237063. [Online; accessed 30-June-2025].

[69] Jie You, Jae-Won Chung, and Mosharaf Chowdhury. 2023. Zeus: Understanding and optimizing {GPU} energy consumption of {DNN} training. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 119–139.

[70] Hadi Zamani, Devashree Tripathy, Laxmi Bhuyan, and Zizhong Chen. 2020. SAOU: safe adaptive overclocking and undervolting for energy-efficient GPU computing. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*. ACM, Boston Massachusetts, 205–210. doi:10.1145/3370748.3406553

[71] Xunyu Zhu, Jian Li, Yong Liu, Can Ma, and Weiping Wang. 2024. A survey on model compression for large language models. *Transactions of the Association for Computational Linguistics* 12 (2024), 1556–1577.

[72] ZJin. 2025. GitHub - zjin-lcf/HeCBench — github.com. https://github.com/zjin-lcf/HeCBench. [Accessed 14-04-2025].