SwizzlePerf: Hardware-Aware LLMs for GPU Kernel Performance Optimization

Arya Tschand[1,3]   **Kesavan Ramakrishnan**[2,3]   **Muhammad Awad**[3]   **Ryan Swann**[3]
Jeffrey Ma[1]   **Keith Lowery**[3]   **Ganesh Dasika**[3]   **Vijay Janapa Reddi**[1]

[1]Harvard University   [2]Stanford University   [3]Advanced Micro Devices, Inc.

aryatschand@g.harvard.edu[1]

Abstract

Large language models (LLMs) have shown progress in GPU kernel performance engineering using inefficient search-based methods that optimize around runtime. Any existing approach lacks a key characteristic that human performance engineers rely on for near-optimal utilization – *hardware-awareness*. By leveraging the workload's specific memory access patterns, architecture specifications, filtered profiling logs, and reflections on historical performance, we can make software-level optimizations that are tailored to the underlying hardware. SwizzlePerf automatically generates spatial optimizations for GPU kernels on disaggregated architectures by giving LLMs explicit hardware-awareness.

For a GEMM kernel, SwizzlePerf takes less than 5 minutes to generate the same hardware-specific optimal swizzling pattern that took expert performance engineers 2 weeks to find. On a suite of 10 diverse ML and Science kernels, SwizzlePerf can generate swizzling patterns for 9 of the kernels that achieve up to a $2.06\times$ speedup and 70% improvement in L2 hit rate. This work is the first of many steps toward systematically creating hardware-aware LLM performance engineering agents.

1 Introduction

GPU code performance engineering is a necessary step in enabling efficient machine learning (ML) systems and High-Performance Computing (HPC) applications. Performance engineering requires hardware-software codesign through understanding how the specific workload is executing on the specific underlying hardware. In this work, we automate this process for GPU kernels by imitating this hardware-software codesign process that human performance engineers follow. This is accomplished by giving LLMs hardware-aware context, which unlocks their ability to structure GPU code optimizations around the underlying hardware architecture and scheduling methodology.

SwizzlePerf is our proposed hardware-aware LLM workflow that automatically generates *swizzling* patterns. Swizzling is a transformation that reorders the mapping between data or work and their execution/storage locations to enhance spatial/temporal locality and align with hardware topology. Compilers and runtimes use swizzling for data layouts and for clustering cooperating blocks. In the case study that we evaluate in this paper, swizzling is a programmer-defined remapping of GPU workgroup program IDs (PIDs) that co-locates related tiles on the same accelerator complex die (XCD) to increase per-XCD L2 reuse on GPUs with disaggregated architectures. By default, workgroups are assigned in a round-robin manner across XCDs (Figure 1b and Appendix A.2). For applications with predictable memory-access patterns, statically arranging computation and data to maximize local L2 access yields substantial benefits.

It takes expert GPU performance engineers multiple weeks to make targeted spatial optimizations. SwizzlePerf autonomously finds these optimizations for a wide range of kernels in minutes.

Contributions of this work:

1. We facilitate hardware-aware kernel optimization by intentionally crafting context with necessary hardware and scheduling information from profilers and runtimes.

2. We augment IntelliPerf, an open-source autonomous performance engineering tool, with hardware-awareness and evaluate a case study on generating GPU kernel swizzling patterns.

3. We show performance results on a wide range of ML and scientific GPU kernels of up to a 2.1x speedup and 70% L2 hit rate improvement, indicating that hardware-awareness is necessary to unlock hardware-specific optimizations.
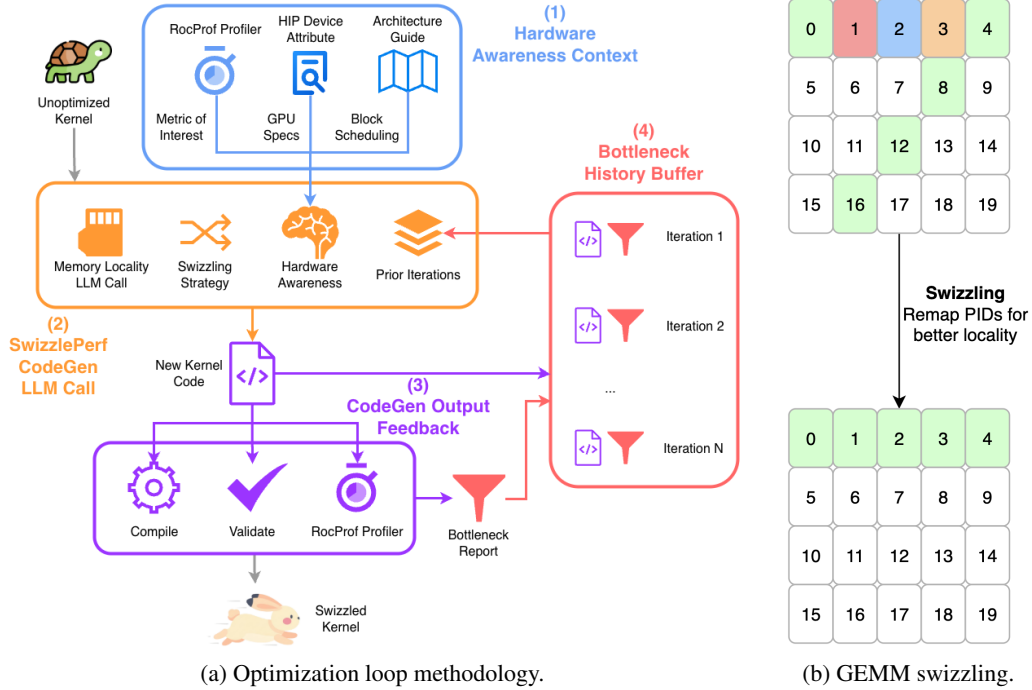

(a) Optimization loop methodology.　　　　(b) GEMM swizzling.

Figure 1: *SwizzlePerf methodology and example swizzling outcome.* **(a)** The optimization loop integrates the Hardware Awareness Context of bottleneck metrics and GPU characteristics into the SwizzlePerf CodeGen LLM Call, that proposes a new swizzling formula, validates correctness, and profiles execution for the CodeGen Output Feedback. The swizzling formulas and compiled bottleneck reports are stored in the Bottleneck History Buffer and reflected upon in future iterations. **(b)** SwizzlePerf generates this pattern for GEMM on a 4-XCD architecture. The swizzling pattern improves locality in the L2 cache by co-locating tiles that reuse rows in A on the same XCD. While the goal is always to improve intra-XCD locality, the exact swizzling formula varies by algorithm.

2　Related Work

Prior work on autonomous kernel optimization largely optimizes for *runtime* as the single objective using three families of methods: (i) heuristic autotuners and analytic cost models that apply rule-based schedules (tiling, unrolling, coalescing) [26, 13, 6] (ii) learned cost models that predict runtime from code or schedule features and drive combinatorial search [9, 27, 1] and (iii) LLM/RL-driven search that performs test-time exploration or multi-turn refinement, ranging from competitive-programming systems [18, 22] to GPU-kernel frameworks [20, 8, 5, 17]. These approaches can work for software-level bottlenecks, but provide minimal machine feedback and abstract away architecture-specific features, limiting its ability to utilize underlying compute capabilities. To converge on *hardware-specific* optimizations, the LLM needs relevant context that enables *hardware-awareness*.

We approach this problem by imitating human performance engineers: profile the workload, isolate the bottleneck, apply a targeted fix, and precisely evaluate the improvement. Practically, this means intentionally supplying the profiling, architectural, and scheduling context that is relevant to the optimization task and guiding feedback with a *bottleneck metric* that provides a stable signal aligned

with the transformation. By elevating this metric to the core optimization objective, we narrow the search to changes that directly target the bottleneck. For the cache efficiency bottleneck case study in this paper, we measure L2 hit rate as a direct, low-noise proxy metric for spatial locality. To our knowledge, SwizzlePerf is the only work that adds rich context from a suite of profilers into the context to directly reflect cache-locality improvements and improve LLM optimization.

Within this framing, we implement *swizzling* patterns for disaggregated GPUs (e.g., AMD Instinct[TM] MI300x [23]) where multiple accelerator complex dies (XCDs) each host a dedicated L2 cache. By default, workgroups are scheduled round-robin across XCDs (Figure 1b, Appendix A.2). However, for applications with predictable access patterns, statically co-locating cooperating tiles on the same XCD with swizzling can substantially improve data reuse in the shared L2 cache. Our contribution adds hardware-awareness and optimizes around a bottleneck metric so the search is focused, the signal is stable, and the resulting transformations align with the underlying hardware.

3 Methodology

SwizzlePerf (Figure 1a) is a hardware-aware, bottleneck-driven optimization loop.

(1) Hardware Aware Context. We construct a structured context from public profilers and documentation. From `rocprofv3`[4] we extract bottleneck metrics, from HIP device attributes[3] we gather GPU specifications, and from architecture guides we derive the default block-scheduling policy. This parsed context exposes the metric of interest (for swizzling, this metric is L2 hit rate) and the spatial constraints the remapping must satisfy.

(2) SwizzlePerf CodeGen LLM Call. We formulate a targeted code-generation request that fixes the swizzling optimization objective and explicitly defines the bottleneck metric of interest. The prompt (Appendix A.1) bundles a short memory-locality summary of the kernel, a compact trace of prior attempts, and architecture details (e.g., number of XCDs, cache sizes, and the block-scheduling policy). We build on the open source IntelliPerf [7] framework to run a a *hardware-aware* GPU kernel swizzling optimization loop.

(3) CodeGen Output Feedback. Using DSPy [15, 16], we specify the output signature so the model must critique past attempts and propose a new swizzling pattern. The LLM returns (i) a reasoning trace that contrasts old and new mappings, and (ii) a swizzling formula implementation. We then compile the new code, validate correctness against the ground truth, and run `rocprofv3` to obtain the updated bottleneck report. We reuse IntelliPerf's compile/validate/profile scaffold, but guide code generation with hardware-aware context and a fixed objective rather than unconstrained search.

(4) Bottleneck History Buffer. Each iteration appends the code diff and bottleneck report to a persistent history buffer. Subsequent calls see this history, reflect on failures (e.g., broken mappings or no L2 change), and propose diversified remappings. We rank candidates by L2 hit rate (primary) and retain the best validated kernel. This buffer closes the hardware–software loop by feeding back a bottleneck-specific signal that accelerates convergence to architecture-aligned swizzling patterns.

4 Results

We compare the hardware-aware *SwizzlePerf* against two baselines: *Hardware-unaware*, the base IntelliPerf loop without hardware or scheduling context, and *Hardware-overload*, which passes in an unfiltered 10k+ token public GPU architecture documentation dump. We evaluate 10 GPU kernels that are relevant to real-world workloads. We collect 6 ML kernels (GEMM, fused elementwise, layer normalization, softmax, naive sparse matrix vector multiplication (SpMV), transpose) and 4 Science workloads (Black-Scholes, finite-difference time-domain (FDTD) 2D, Smith-Waterman, Stencil 2D). We benchmark on medium problem sizes (~5ms) and validate correctness against reference implementations (PyTorch for ML kernels, CPU implementation for Science kernels). In Figure 2, $\times$ denotes cases where swizzling had no effect on L2 hit rate, while $\otimes$ marks broken remappings.

Across the 10 kernels, SwizzlePerf achieves an average speedup of $1.29\times$ and up to a $2.06\times$ on the transpose kernel. This large gain on transpose comes from striding the M×N tile grid across XCDs so that both the original reads and the transposed writes stay within the same XCDs's L2 cache, eliminating cross-XCD thrashing. The softmax kernel has a $1.54\times$ speedup by grouping all row chunks into the same XCD across its two-phase reduction, which keeps row values resident in L2 and reduces conflict misses. These results show that SwizzlePerf-generated patterns consistently improve end-to-end runtimes and overall system efficiency on disaggregated GPUs.
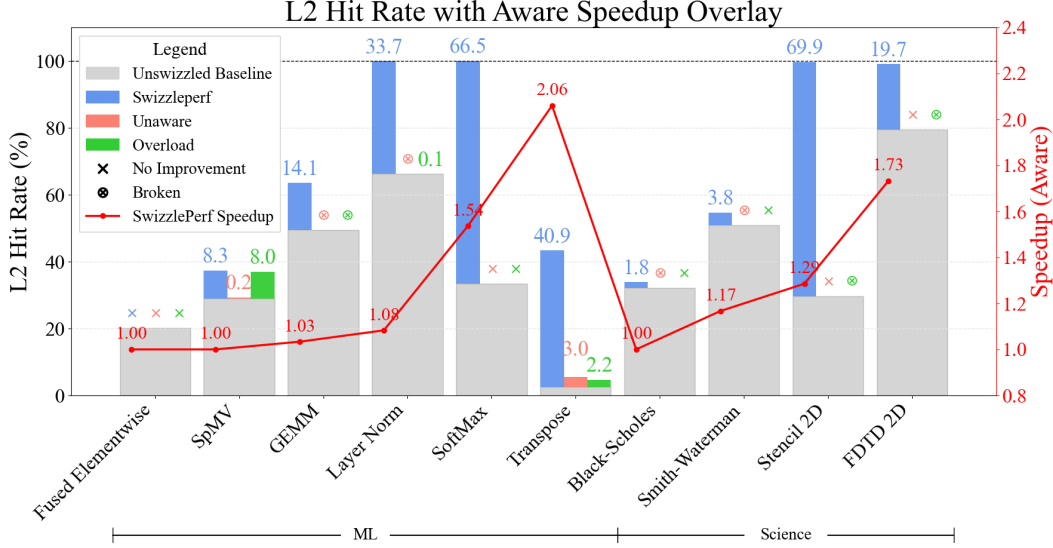
Figure 2: *L2 hit rate improvements and speedups from swizzling patterns on 10 kernels.* The gray bars denote the original L2 hit rate of unswizzled code, blue bars show the L2 hit rate improvement from the SwizzlePerf-generated swizzling patterns, and orange and green bars show baselines. The red line denotes the corresponding SwizzlePerf end-to-end kernel speedup. SwizzlePerf achieves speedups on 9 out of 10 kernels of up to $2.06\times$. These speedups are driven by higher cache efficiency, where SwizzlePerf improves L2 hit rate by an average of 23.9% and up to 70%. 4 of these SwizzlePerf implementations approach the maximum cache efficiency of 100% L2 hit rate.

While our swizzling results show strong end-to-end speedups, we can take advantage of our direct optimization for cache locality. It is more representative to measure success through L2 hit rate rather runtime alone. Runtime can be noisy due to kernel launch overheads, compute saturation, and overlapping bottlenecks, but L2 hit rate directly reflects whether cache-aware remapping is successful. SwizzlePerf finds patterns that improve L2 hit rates by up to 70%, with an average of 23.9%. The swizzled implementation of 4 kernels reach nearly 100% L2 hit rate, showing how SwizzlePerf can help approach the hardware maximum for cache efficiency.

For compute-bound kernels like GEMM, the 14% L2 hit rate improvement translates to a modest $1.03\times$ speedup. Conversely, memory-sensitive kernels like stencil and transpose show that when L2 locality is the bottleneck, hit rate improvements enable significant speedups. Strong gains across the board showcase the importance of *intentionally* providing relevant hardware context regardless of the kernel's specific bottleneck. By contrast, the hardware-unaware and hardware-overload swizzling patterns lead to minimal L2 hit rate improvements and never give a speedup.

Hardware-awareness and specific guiding metrics enable SwizzlePerf to reliably uncover optimizations that both (a) validate the generalizable effectiveness of swizzling (Appendices A.5 and A.6) and (b) translate into significant end-to-end speedups.

5 Discussion and Future Work

SwizzlePerf addresses the missing hardware-awareness in today's autonomous performance engineering frameworks. By demonstrating that an LLM equipped with hardware-aware context can replicate expert reasoning in minutes on a wide range of GPU kernels, we underscore how closing the hardware–software feedback loop unlocks substantial efficiency gains. Looking ahead, we believe the next breakthrough will come from expanding the modalities through which an LLM perceives hardware to improve kernel performance and power [24]. Non-text modalities like visualizations of swizzling patterns are challenging because they can be algorithm-specific. We believe that attention kernels are a great place to start because they are functionally similar to many different implementations [11, 21, 2, 10, 12], each with its own optimal swizzling pattern.

Our future work is guided by the question - **What are the right modalities of hardware-awareness that enable LLMs to think like a human performance engineer on real-world workloads?**

A   Technical Appendices and Supplementary Material

A.1   LLM Input Prompt and Output Structure

Input prompt

The original code is {...} with bottleneck {...}

The memory analysis is {...}

History of previous optimization attempts (do not repeat an implementation): {

    iteration: 1, applied diff: {...}, bottleneck report: {...},

    iteration: N-1, applied diff: {...}, bottleneck report: {...}

}

On the AMD MI300X, there are {A} XCDs, each has {B} CUs and a {C} MB L2 cache

Blocks are scheduled {Round-robin to XCDs}

Your swizzling goal is to {...}, pay special attention to {...}, code should be structured as {...}
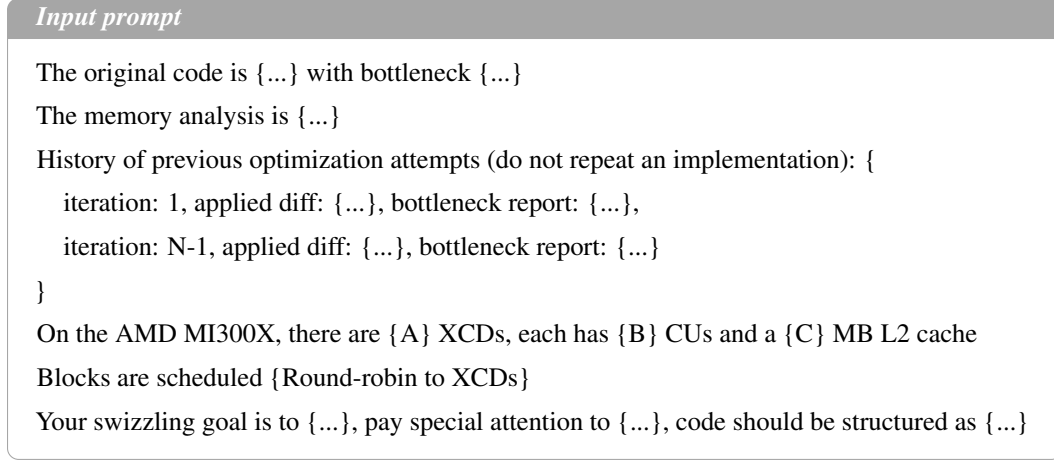

Figure 3: Structure of hardware-awareness input prompt to LLM.

Figure 3 shows the structure of the code optimization input prompt. Note that it takes in the original unoptimized code with its bottleneck report, the memory analysis from the LLM memory analysis call, the history of all prior iterations and bottlenecks, and a description of the hardware architecture with profiled details.

Output Signature

Chain-of-Thought Reasoning

JSON dict of why old implementations were suboptimal: {

    iteration 1: This implementation only marginally increased L2 hit rate because {...},

    iteration N-1: This iteration didn't change L2 hit rate because {...}

}

My new swizzling approach will remap blocks by {...}

My new swizzling approach will be better than any prior implementation because {...}
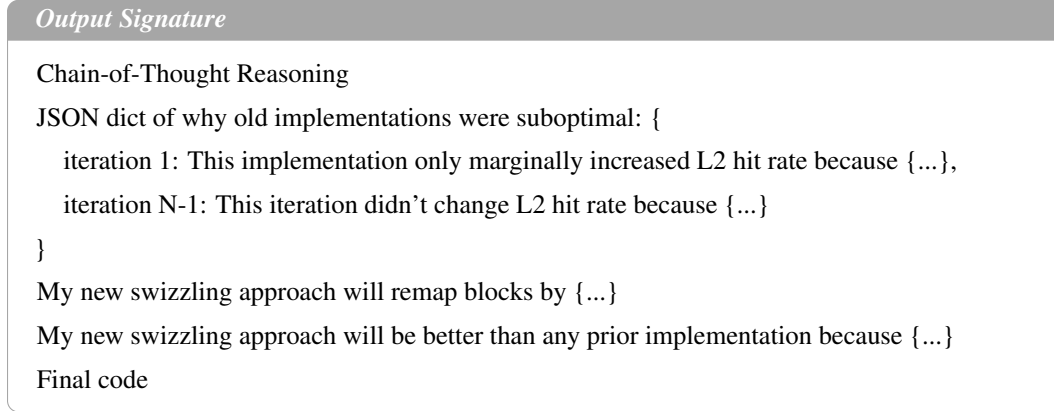
Final code

Figure 4: Structure of DSPy output signature.

Figure 4 shows the structure of the code optimization output signature. The output signature is a parameter into DSPy that structures how we want the LLM to output information. We first output an unstructured chain-of-thought reasoning trace, and then explicitly reason about the shortcomings about each prior iteration. Lastly, we rationale the new implementation and output the code for it.

A.2   Case Study: General Matrix Multiplication (GEMM) Swizzling

In tiled GEMM implementations, tiles closer together in the output matrix C (M×N) share more data from input matrices A (M×K) and B (K×N). Tiles in the same row in C share the rows from A, and tiles in the same column of C share columns from B. Unfortunately, because of the default round-robin block scheduling in the M=4, N=5, #XCD=4 example in Figure 5, adjacent tiles are placed on different XCDs and we therefore observe a hit rate on the L2 cache shared by each XCD below the theoretical maximum. While we cannot explicitly reschedule blocks to new CUs, we can

make each CU process different input data and write to a different output tile by recomputing the PIDs of each block. This essentially trades work between CUs, has minimal overhead to compute, and can improve L2 hit rate in many kernels.

SwizzlePerf automatically generates swizzling patterns for any kernel algorithm, hardware architecture, or default block scheduling scheme. The GEMM swizzling example in Figure 5 took expert performance engineers 2 weeks to design and validate. SwizzlePerf achieved a functionally identical solution with additional edge case catching in < 5 minutes. The ceiling division edge case catch when calculating blocks per XCD is not present in the expert-generated swizzling pattern, and implies that SwizzlePerf is applying new techniques and not just retrieving existing solutions.


```
pid = tl.program_id(0)

num_xcds = 8
num_blocks = NUM_SMS


# Calculate blocks per XCD with ceiling division
b_per_xcd = (num_blocks + num_xcds - 1) // num_xcds


# Swizzle so all contiguous blocks are on same XCD
# before moving to the next XCD
pid = (pid % num_xcds) * b_per_xcd + (pid // num_xcds)
```

(a) GEMM swizzling.

(b) SwizzlePerf-generated swizzling Triton code. Note the additional edge case catch with ceiling division.
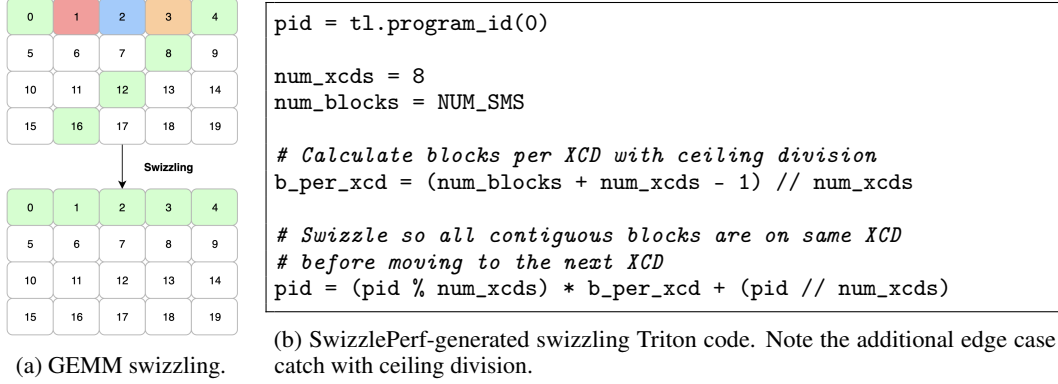
Figure 5: GEMM swizzling strategy and equivalent SwizzlePerf-generated swizzling pattern.

A.3 Case Study: Other Swizzling Pattern Examples and Interpretations

```
pid_m = tl.program_id(0)
num_xcds = 8
num_blocks = M

blocks_per_xcd = (num_blocks + num_xcds - 1) // num_xcds
xcd_id = (pid_m // num_xcds) % num_xcds

local_block_id = pid_m % num_xcds
offset = (local_block_id // num_xcds) * num_xcds + xcd_id

pid = offset * blocks_per_xcd + local_block_id
```

Figure 6: SwizzlePerf-generated swizzling pattern for layer normalization kernel.

In the layernorm swizzling shown in Figure 6, we remap pid_m so that all BLOCK_SIZE_N column-chunks for the same input row are routed to the same XCD. The kernel keeps that row's partial sums (mean and variance) and the loaded weight/bias vectors resident in that XCD's L2 across multiple loads, massively boosting intra-row data reuse. At the same time, spreading different rows evenly over all eight XCDs prevents any single L2 cache from thrashing, which improves hit rate by the observed 34%.

```
num_xcds = 8
num_blocks = tl.num_programs(0)

pid = tl.program_id(0)

blocks_per_xcd = (num_blocks + num_xcds - 1) // num_xcds
xcd_id = (pid // num_xcds) % num_xcds
block_in_xcd = pid % num_xcds
new_pid = xcd_id * blocks_per_xcd + block_in_xcd
pid = new_pid
```

Figure 7: SwizzlePerf-generated swizzling pattern for softmax kernel.

In the softmax swizzling shown in Figure 7, because softmax for each row does a two-phase reduction (first finding the maximum, then exponentiating and summing), grouping all the BLOCK_SIZE chunks of a given row into the same XCD keeps those row values resident in that XCDs's L2 cache across both passes, rather than bouncing them through DRAM. At the same time, by evenly interleaving different rows across all eight XCDs, no single XCD becomes a hotspot, so conflict misses and evictions drop—together driving a 66% lift in L2 hit rate.

```
pid_x = tl.program_id(0)
pid_y = tl.program_id(1)

num_XCD = 8
num_blocks_x = (Nx + BLOCK_SIZE_X - 1) // BLOCK_SIZE_X
num_blocks_y = (Ny + BLOCK_SIZE_Y - 1) // BLOCK_SIZE_Y
num_blocks = num_blocks_x * num_blocks_y

# Calculate new block index using a different swizzling pattern
block_id = pid_y * num_blocks_x + pid_x
xcd_id = (block_id // num_blocks_x) % num_XCD
new_block_id = (block_id % num_blocks_x) * num_XCD + xcd_id

# Map new block id back to 2D grid
pid_x = new_block_id // num_blocks_x
pid_y = new_block_id % num_blocks_x
```

Figure 8: SwizzlePerf-generated swizzling pattern for finite-difference time-domain kernel.

In the finite difference time domain (FDTD) swizzling shown in Figure 8, it maps each 2D FDTD block so that blocks lying on the same vertical stride (i.e., sharing the same x-range but different y-ranges) end up on the same XCD. At the same time, by distributing those strides round-robin across all eight XCDs (via the xcd_id = pid_y % num_XCD step), no gets overloaded with too many neighbor-dependent blocks, so you avoid cache thrashing. Together, this alignment of FDTD's neighboring accesses and balanced loads across XCDs improves L2 hit rate by 20%.

```
pid_m = tl.program_id(0)
pid_n = tl.program_id(1)

num_blocks_m = (M + BLOCK_SIZE_M - 1) // BLOCK_SIZE_M
num_blocks_n = (N + BLOCK_SIZE_N - 1) // BLOCK_SIZE_N
total_blocks = num_blocks_m * num_blocks_n

block_id = pid_m * num_blocks_n + pid_n
xcd_id = (block_id // num_blocks_m) % 8
round_id = block_id // (8 * num_blocks_m)
new_block_id = round_id * 8 + xcd_id

pid_m = new_block_id // num_blocks_n
pid_n = new_block_id % num_blocks_n
```

Figure 9: SwizzlePerf-generated swizzling pattern for stencil 2D kernel.

In the stencil 2D swizzling shown in Figure 9, it remaps the blocks so that blocks which are neighbors in the M or N direction (and thus share data for the dimensions in stencil) end up on the same XCD's L2 cache. By laying out the grid in these diagonal-stride groups, each XCD keeps its center-and-neighbor loads resident across multiple block invocations, while the round-robin scheduling prevents any one XCD from thrashing-together, and improves L2 hit rate by 70%.

7

```
num_xcds = 8
pid_m = tl.program_id(0)
pid_n = tl.program_id(1)


pid = pid_m * (N // BLOCK_SIZE_N) + pid_n
xcd_idx = (pid // num_xcds) % num_xcds
pid = xcd_idx * (N // BLOCK_SIZE_N) * (M // BLOCK_SIZE_M) // num_xcds + pid //
    num_xcds


pid_m = pid // (N // BLOCK_SIZE_N)
pid_n = pid % (N // BLOCK_SIZE_N)
```

Figure 10: SwizzlePerf-generated swizzling pattern for transpose kernel.

In the transpose swizzling shown in Figure 10, it strides the linearized M×N tile grid across the eight XCDs. This takes each block's row-major index modulo num_xcds to pick a XCD and then re-assembles the remaining blocks contiguously. Each XCD ends up owning whole "bands" of input rows and their corresponding output columns. For transpose, that means when you load a BLOCK_SIZE_M×BLOCK_SIZE_N tile and then write its BLOCK_SIZE_N×BLOCK_SIZE_M transpose, both the original reads and the transposed writes hit the same XCD's L2 rather than thrashing across XCDs. At the same time, because the blocks are evenly distributed, no single L2 becomes a hotspot and leads to a 34% improvement in hit rate.

A.4 Case Study: Hardware-Overload Induces Bitwise Swizzling

```
pid_m = tl.program_id(0)
pid_n = tl.program_id(1)


num_xcds = 8
num_blocks_m = (M + BLOCK_SIZE_M - 1) // BLOCK_SIZE_M
num_blocks_n = (N + BLOCK_SIZE_N - 1) // BLOCK_SIZE_N
total_blocks = num_blocks_m * num_blocks_n


original_block_index = pid_m * num_blocks_n + pid_n


xcd_id = original_block_index % num_xcds
round_id = original_block_index // num_xcds
new_block_index = round_id + xcd_id * (total_blocks // num_xcds)


pid_m = new_block_index // num_blocks_n
pid_n = new_block_index % num_blocks_n
```

Figure 11: Hardware-aware SwizzlePerf swizzling code for transpose kernel.

```
pid = tl.program_id(0)


# Swizzle the program IDs to improve cache locality
pid_m = pid // (N // BLOCK_SIZE_N)
pid_n = pid % (N // BLOCK_SIZE_N)
```

Figure 12: Hardware-unaware swizzling code for transpose kernel.

```
pid = tl.program_id(0)


# Swizzle the pid to improve cache locality
pid = ((pid >> 1) & 0x55555555) | ((pid & 0x55555555) << 1)
```

Figure 13: Hardware-overload swizzling code for transpose kernel.

In experiments on specific kernels and problem sizes, the hardware-overload optimization loop achieves strong speedups by applying bitwise SHIFT and ADD swizzling (Figure 13). Hardware-unaware applies a simple size-based pattern (Figure 12), while SwizzlePerf generates more complex,

MI300x-specific mappings (Figure 11). The bitwise approach can achieve up to 70% higher L2 hit rate than SwizzlePerf, but it is overfit to particular sizes and often fails correctness. For example, on the transpose kernel with M=32768 and N=32768, a low-bit swap aligned perfectly to co-locate tiles on the same XCD, but similar patterns failed on other kernels.

These results show that hardware-overload swizzling does not generalize: if the tile count is not aligned with the number of XCDs, if BLOCK_SIZE_M/N changes, or if the GPU has a different XCD count, the hard-coded swap breaks reuse. Many overload patterns also failed bijectivity, sending blocks to non-existent partners when grid shapes changed. While hardware-overload offers an interesting idea that could be integrated into SwizzlePerf, its lack of robustness across problem sizes limits usability. By contrast, SwizzlePerf generalizes correctly across kernels and sizes, even if it does not always achieve the very best result on every configuration.

A.5 Optimization Loop Progression

We evaluate the swizzling patterns generated in subsequent iterations of SwizzlePerf against the two baselines. In Figure 14, the dotted line is the L2 hit rate of the implementation from the numbered iteration, and the solid line is the best prior implementation that would be returned. Each iteration receives context on the weaknesses of earlier swizzling patterns, so improvement in hit rate represents successful reflection on past approaches.

(a) GEMM kernel progression.

(b) Stencil 2D kernel progression.

(c) Naive SpMV kernel progression.

(d) Softmax kernel progression.
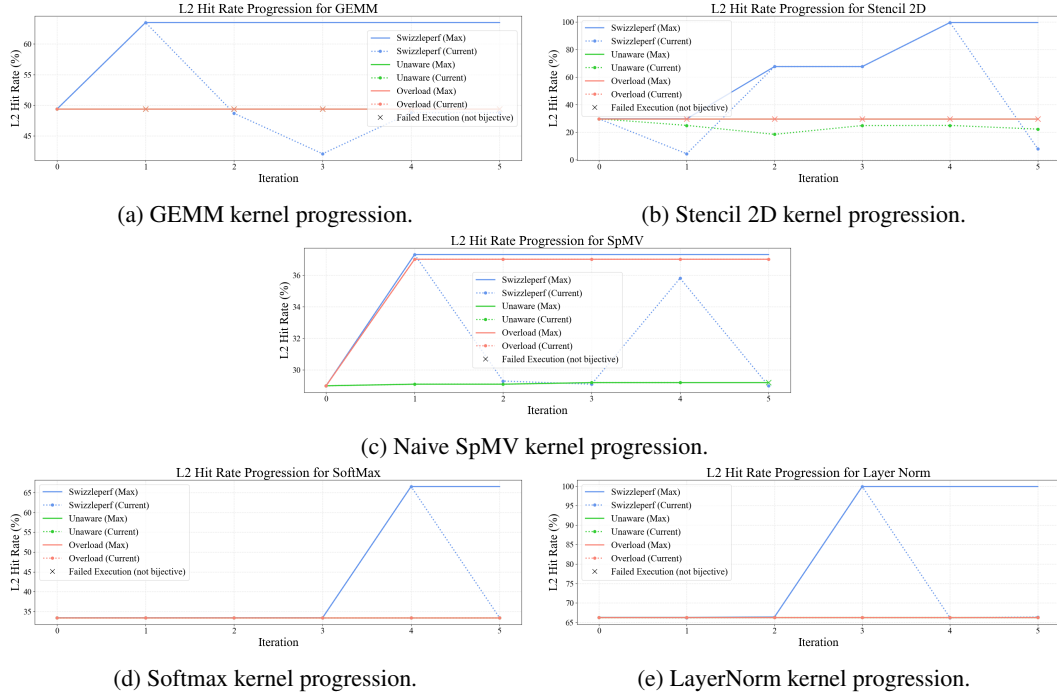
(e) LayerNorm kernel progression.

Figure 14: Progression plots for GEMM, Stencil 2D, SpMV, Softmax, and LayerNorm kernels. SwizzlePerf consistently finds more performant swizzling patterns and explores more diverse solutions than baselines. The dotted line shows the L2 hit rate of the current iteration's implementation, while the solid line tracks the best-so-far implementation.

For GEMM in Figure 14a, SwizzlePerf finds a correct swizzling pattern within one iteration, while both baselines fail across all five iterations. For Stencil 2D in Figure 14b, SwizzlePerf shows steady improvement to nearly 100% L2 hit rate and explores diverse solutions instead of getting stuck in a local minimum. The hardware-unaware approach consistently generates patterns that degrade L2 hit rate, and the hardware-overload approach always fails by generating incorrect swizzling patterns with a bitwise AND/SHIFT strategy (Appendix A.4).

For Naive SpMV in Figure 14c, hardware-overload achieves slightly worse performance than SwizzlePerf and fixates on its initial method instead of exploring alternatives. For Softmax in Figure 14d

and LayerNorm in Figure 14e, SwizzlePerf converges on successful patterns after multiple iterations, while the baselines fail to adapt.

A.6 Ablation Study: Evaluating on Different Problem Sizes

We evaluate the SwizzlePerf generated swizzling patterns across different problem sizes of the target kernel. It is important to ensure that the implementations are generalizable to many inputs and not just overfit to a specific problem size.

The layernorm kernel shown in Figure 15a is evaluated on problem sizes $64 \times 1024 \times 1024$ to $512 \times 8192 \times 1024$ elements. Swizzling lifts the L2 hit rate from 46% to 60%. This delta remains essentially flat as the tensor grows. The consistent gap indicates that SwizzlePerf was able to find a pattern that was generalizable and consistently outperforms the unswizzled for.

The Smith-Waterman kernel shown in Figure 15b is evaluated across five doublings in sequence length (from 512^2 to 8192^2 DP cells), the swizzled kernel raises the L2 hit rate from 50% to 65%, while the unswizzled baseline tops out near 56%. The gap therefore widens from about 3% on the smallest input to 10% on the largest. This equates to about a 15-20% relative gain that becomes more pronounced as the grid grows.

Stencil 2D kernel shown in Figure 15c is evaluated with grid side increasing twenty-fold (512×512 to $10{,}240 \times 10{,}240$ cells). The unswizzled hit rate falls from 65% down to 53%, whereas the swizzled version stays in the 74-82% range. This yields a 20-30% advantage at large problem sizes—over 40% relative—showing that swizzling keeps cache locality almost size-invariant for this Jacobi update.
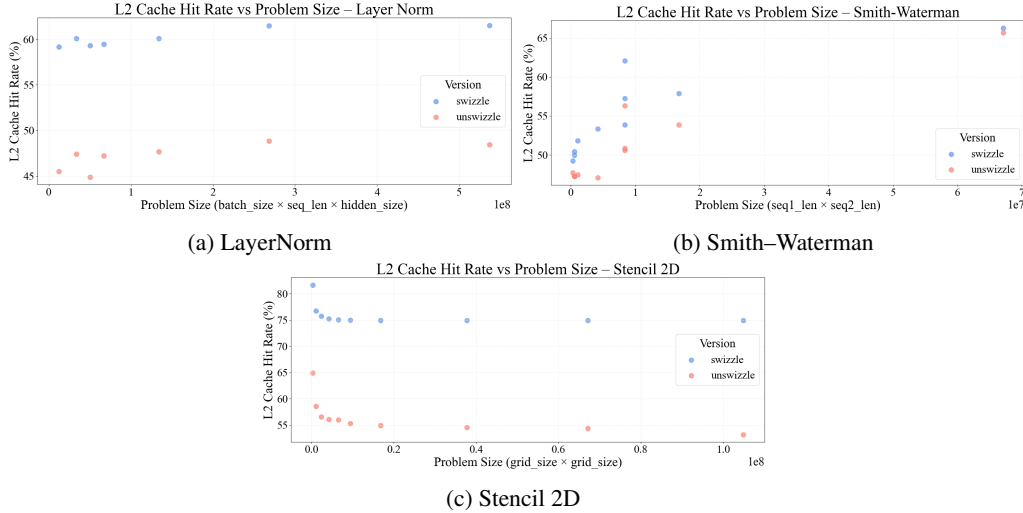

(a) LayerNorm

(b) Smith–Waterman

(c) Stencil 2D

Figure 15: L2 cache hit rate vs. problem size for the three kernels.

A.7 Ablation Study: Evaluating Hardware-Awareness with Different LLMs

In Figure 16, we evaluate how different LLMs impact SwizzlePerf-generated swizzling patterns. We run SwizzlePerf with OpenAI's 4o, 4.1-mini, and o3-mini on GEMM, LayerNorm, FDTD, and Smith-Waterman.

In GEMM, all three models converge to a swizzling pattern equivalent to the expert-generated one, which is expected given GEMM's relatively straightforward locality structure. For LayerNorm and FDTD, only 4o discovers effective solutions. 4.1-mini is likely too small to reason about complex mappings, and o3-mini (despite being a reasoning model and outperforming on most benchmarks) sometimes underperforms. In contrast, on Smith-Waterman, o3-mini generates a more complex swizzling pattern that outperforms both 4o and 4.1-mini.

These results suggest that different models excel on different kernels, and that reasoning-focused training does not always help in hardware-aware optimization where explicit context is already provided. Selecting the best model likely depends on kernel, architecture, and problem size. We
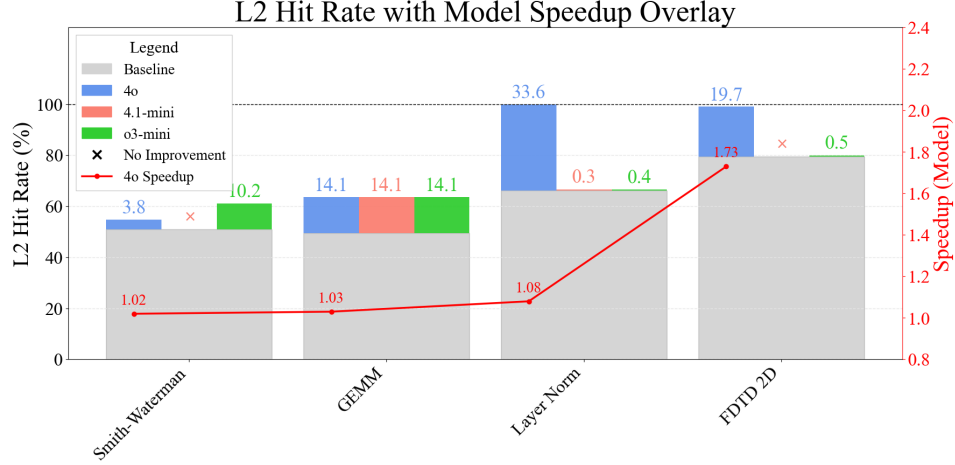
Figure 16: L2 hit rate of SwizzlePerf-generated patterns with 3 different LLMs.

plan to broaden this evaluation to open- and closed-source models across Anthropic, Meta, OpenAI, Google, etc., to better understand which approaches are most effective for hardware-awareness.

A.8 Future Consideration: Swizzling for Power Efficiency

While the primary focus of SwizzlePerf has been performance, we believe that same locality-aware remapping that lifts L2 hit rate will also have pronounced power-efficiency benefits. Each miss that is redirected from device memory to the on-chip L2 avoids the high energy cost of traversing the full memory hierarchy. By clustering cooperative blocks within a single XCD, our generated swizzling patterns cut off-chip traffic and stabilize residency in the disaggregated caches, trimming the average energy per instruction even in kernels whose execution time is dominated by arithmetic throughput. In short, better locality is doubly rewarded on modern chiplet GPUs, once in latency, and again in joules.

These gains persist and may even grow in ostensibly compute-bound workloads because real devices operate under dynamic voltage and frequency scaling (DVFS). More hits in the L2 lowers instantaneous power draw, pushing the kernel farther from the DVFS throttle point and permitting higher sustained clocks or headroom for co-resident kernels. Conversely, when DVFS caps frequency, swizzling can still improve energy-to-solution by ensuring a larger fraction of each watt is spent in compute instead of memory access. We can rigorously evaluate [19, 14, 25] and apply these insights to make energy efficiency-driven swizzling decisions.

References

[1] Andrew Adams, Sung Hee Ma, Luke Anderson, Jonathan Ragan-Kelley, et al. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)*, 38(4):121:1–121:12, 2019. URL `https://halide-lang.org/papers/halide_autoscheduler_2019.pdf`.

[2] Joshua Ainslie, James Lee-Thorp, Michiel De Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245*, 2023.

[3] AMD. Heterogeneous-computing interface for portability (HIP). `https://rocm.docs.amd.com/projects/HIP/en/latest/index.html`, . [Online; accessed 06-August-2025].

[4] AMD. ROCprofiler-SDK: Application profiling, tracing, and performance analysis. `https://github.com/ROCm/rocprofiler-sdk`, . [Online; accessed 06-August-2025].

[5] Martin Andrews and Sam Witteveen. Gpu kernel scientist: An llm-driven framework for iterative kernel optimization. *arXiv preprint arXiv:2506.20807*, 2025.

[6] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2014. doi: 10.1145/2628071.2628092.

[7] Muhammad Awad, Cole Ramos, and Keith Lowery. Intelliperf: LLM-powered autonomous GPU performance engineer, July 2025. URL `https://github.com/AMDResearch/intelliperf`.

[8] Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. Kevin: Multi-turn rl for generating cuda kernels. *arXiv preprint arXiv:2507.11948*, 2025.

[9] Tianqi Chen, Lianmin Zheng, Eddie Q. Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2018. URL `https://arxiv.org/abs/1805.08166`.

[10] Jean-Baptiste Cordonnier, Andreas Loukas, and Martin Jaggi. Multi-head attention: Collaborate instead of concatenate. *arXiv preprint arXiv:2006.16362*, 2020.

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

[12] Juechu Dong, Boyuan Feng, Driss Guessous, Yanbo Liang, and Horace He. Flex attention: A programming model for generating optimized attention kernels. *arXiv preprint arXiv:2412.05496*, 2024.

[13] Matteo Frigo and Steven G. Johnson. The design and implementation of fftw3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[14] Sunpyo Hong and Hyesoon Kim. An integrated gpu power and performance model. In *Proceedings of the 37th annual international symposium on Computer architecture*, pages 280–289, 2010.

[15] Omar Khattab, Keshav Santhanam, Xiang Lisa Li, David Hall, Percy Liang, Christopher Potts, and Matei Zaharia. Demonstrate-search-predict: Composing retrieval and language models for knowledge-intensive NLP. *arXiv preprint arXiv:2212.14024*, 2022. doi: 10.48550/arXiv.2212.14024.

[16] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling declarative language model calls into self-improving pipelines. In *Proceedings of the Twelfth International Conference on Learning Representations*, 2024.

[17] Xiaoya Li, Xiaofei Sun, Albert Wang, Jiwei Li, and Chris Shum. Cuda-l1: Improving cuda optimization via contrastive reinforcement learning. *arXiv preprint arXiv:2507.14111*, 2025.

[18] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

[19] Filip Mazurek, Arya Tschand, Yu Wang, Miroslav Pajic, and Daniel Sorin. Rigorous evaluation of computer processors with statistical model checking. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1242–1254, 2023.

[20] Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels? *arXiv preprint arXiv:2502.10517*, 2025.

[21] Rya Sanovar, Srikant Bharadwaj, Renee St Amant, Victor Rühle, and Saravan Rajmohan. Lean attention: Hardware-aware scalable attention mechanism for the decode-phase of transformers. *arXiv preprint arXiv:2405.10480*, 2024.

[22] Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. Learning performance-improving code edits. *arXiv preprint arXiv:2302.07867*, 2023.

[23] Alan Smith, Eric Chapman, Chintan Patel, Raja Swaminathan, John Wuu, Tyrone Huang, Wonjun Jung, Alexander Kaganov, Hugh McIntyre, and Ramon Mangaser. 11.1 amd instincttm mi300 series modular chiplet package–hpc and ai accelerator for exa-class systems. In *2024 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 67, pages 490–492. IEEE, 2024.

[24] Arya Tschand, Arun Tejusve Raghunath Rajan, Sachin Idgunji, Anirban Ghosh, Jeremy Holleman, Csaba Kiraly, Pawan Ambalkar, Ritika Borkar, Ramesh Chukka, Trevor Cockrell, et al. Mlperf power: Benchmarking the energy efficiency of machine learning systems from microwatts to megawatts for sustainable ai. *arXiv preprint arXiv:2410.12032*, 2024.

[25] Vincent M Weaver, Matt Johnson, Kiran Kasichayanula, James Ralph, Piotr Luszczek, Dan Terpstra, and Shirley Moore. Measuring energy and power with papi. In *2012 41st international conference on parallel processing workshops*, pages 262–268. IEEE, 2012.

[26] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27(1–2):3–35, 2001.

[27] Lianmin Zheng, Chengfan Gao, Eddie Ye, Junru Shao, Ziheng Zhuo, Tianqi Chen, Yida Wang, Zhi Zhou, Jared Roesch, Arvind Krishnamurthy, Luis Ceze, Zhihao Jia, Joseph E. Gonzalez, Ion Stoica, and Koushik Sen. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020. URL https://www.usenix.org/system/files/osdi20-zheng.pdf.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction clearly state the contributions: SwizzlePerf adds hardware-awareness to LLM-based kernel optimization, demonstrates it on swizzling for disaggregated GPUs, and reports speedups and L2 hit rate improvements across diverse kernels. These claims match the results shown in the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Section A.8 and the Discussion acknowledge that SwizzlePerf focuses on cache locality, may not capture other bottlenecks, and that different LLMs succeed differently across kernels. Limitations of generalization across kernels and problem sizes are discussed explicitly.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [NA]

   Justification: The paper is empirical and methodological; it does not present formal theorems or proofs.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: The paper describes the benchmark kernels, problem sizes, profiling tools used (`rocprofv3`), GPU hardware (AMD MI300x), and provides kernel code and swizzling patterns in the appendix. These details are sufficient for reproduction.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: The work uses publicly available GPU kernels (ML kernels from PyTorch and scientific workloads with CPU reference implementations). Swizzling patterns and prompts are included in the appendix. The authors intend to release code and scripts for reproduction.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [Yes]

   Justification: The experimental methodology specifies kernel types, input sizes, profiling setup, correctness validation against references, and evaluation metrics (speedup, L2 hit rate).

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [No]

Justification: Results are reported as average speedups and hit rate improvements across deterministic kernel runs. Error bars are not reported beause of computational costs of running an entire SwizzlePerf optimization loop for each kernel. We observed that GPU kernel executions on fixed problem sizes are stable and exhibit minimal variance.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: The paper specifies that experiments were run on AMD MI300x GPUs and medium problem sizes (5ms per kernel). Resource needs are modest (single GPU with profiling tools), and the methodology makes clear what is required.

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The work evaluates performance optimizations on open kernels with no ethical concerns or sensitive data involved.

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: The Discussion and Future Work sections highlight positive impacts such as more efficient use of GPUs for ML and HPC workloads, reduced runtime, and potential energy savings. Negative impacts are minimal since this is a performance engineering technique, but energy consumption implications are discussed.

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: The paper does not release large-scale models or sensitive datasets; only code and optimization prompts are shared, which carry no significant misuse risk.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: All benchmark kernels (ML workloads from PyTorch, scientific workloads from literature) are credited, and profiling tools and GPU APIs are cited with proper references.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: The new asset is the SwizzlePerf workflow and generated swizzling patterns. Documentation is provided in the methodology, appendix, and figures, which describe the prompt structure, output signature, and example patterns.

14. **Crowdsourcing and research with human subjects**

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [NA]

Justification: The work does not involve crowdsourcing or human subjects.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [NA]

Justification: No human subjects research is conducted.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research?

Answer: [Yes]

Justification: The paper clearly explains that LLMs are used in the optimization loop for code generation, reflection, and swizzling pattern synthesis, making LLM usage a central contribution.