Opal: A Modular Framework for Optimizing Performance using Analytics and LLMs

Mohammad Zaeed
Texas State universitry
San Marcos, USA
cup7@txstate.edu

Dr. Tanzima Z. Islam
Texas State universitry
San Marcos, USA
tanzima@txstate.edu

Vladimir Inđić
University of Novi Sad
Novi Sad, Serbia
vladaindjic@uns.ac.rs

Abstract

Large Language Models (LLMs) show promise for automated code optimization but struggle without performance context. This work introduces Opal, a modular framework that connects performance analytics insights with the vast body of published literature by guiding LLMs to generate informed, trustworthy optimizations. Unlike traditional performance tools that identify bottlenecks but stop short of actionable suggestions, Opal bridges this long-standing gap by linking dynamic insights—from hardware counters and Roofline analysis to stall events—to optimization decisions. We evaluate Opal across 1640 experiments on real-world GPU kernels and find that in over 98.5% of cases, even a single insight source yields speedups, ranging on average from 19.34% to 52.3%. Our prompt template produced correct code in all but one case, where a vague diagnostic caused an unsafe suggestion. By automatically optimizing GPU kernels using performance analytics and LLMs, Opal marks a leap toward democratizing expert-level performance engineering for all.

CCS Concepts

• **Do Not Use This Code → Generate the Correct Terms for Your Paper**; *Generate the Correct Terms for Your Paper*; Generate the Correct Terms for Your Paper; Generate the Correct Terms for Your Paper.

Keywords

Multi-source analysis, Automation, Performance engineering, Large language models

ACM Reference Format:

Mohammad Zaeed, Dr. Tanzima Z. Islam, and Vladimir Inđić. 2025. Opal: A Modular Framework for Optimizing Performance using Analytics and LLMs. In . ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

As High Performance Computing (HPC) systems grow in complexity, writing high-performance GPU code demands deep architectural insight and expert-level interpretation of performance diagnostics. Profilers and models can pinpoint bottlenecks—but translating those diagnostics into code changes remains a manual, expertise-driven task. This final step creates a steep barrier: few developers have access to vendor engineers or system architects. Most are left staring at myriads of diagnostic messages and visualizations with no clear path to an actionable step.

The rise of Large Language Models (LLMs) such as GPT-4o [18] and LLaMoCo [36] has created new opportunities for code optimization. Several efforts, such as DeepDev-PERF [14] and RapGen [15], fine-tune LLMs for domain-specific tasks. However, most rely on static analysis [13, 33] or prompt tuning [9, 37, 45], and ignore how the code performs on actual hardware. This is a big gap as optimization decisions are sensitive to runtime behavior—e.g., stalls, warp occupancy, memory throughput—and static inputs miss these signals. This gap between runtime diagnostics and targeted optimization motivates the central research question of this work: **Can LLMs, when guided by dynamic performance insights, generate valid and effective code transformations that directly address runtime bottlenecks?** This gap is empirically validated in our experiments: when no performance insight is provided, the LLM fails to generate any optimizations (see Section 6.1), confirming that performance context is essential for meaningful code transformation.

To bridge the gap, we present Opal[1], a modular and extendable framework for automatically translating *runtime diagnostics into optimizations*. Unlike prior work that treats LLMs as black-box generators, Opal aims to leverage their reasoning capability. To achieve that, Opal encodes multiple complementary performance diagnostics in a prompt described in natural language. We hypothesize that the LLM's pretraining on HPC literature enables it to reason about code transformations that directly address runtime bottlenecks. To our knowledge, Opal is the first framework to close the gap between diagnosis and transformation.

Opal integrates three complementary sources of performance diagnostics: Roofline Analysis, Program Counter (PC) Stall Sampling, and Application–hardware interactions, as none of the sources alone is sufficient. For instance, while counters quantify pressure

[1] Opal stands for **O**ptimizing **P**erformance using **A**nalytics and **L**LMs

on architectural components, they lack code-level precision; conversely, PC Sampling pinpoints code lines but cannot explain architectural boundedness. Thus, these sources form a complementary triad: counters explain *why* performance degrades across many configurations, Roofline analysis shows *how* the kernel behaves on the architectural spectrum, and stall events pinpoint *what and where* micro-level behaviors limit efficiency. To leverage Opal, a user only needs to select the code and their choice of the diagnostic source. They can select none, one, two, or all three of the sources. Our extensive ablation study (Section 6.1) demonstrates that Opal can recommend code transformations with just a single source of information, reducing the average execution time by up to 64.93%.

Building Opal is not as simple as wrapping performance data into a prompt and querying an LLM. Dynamic performance profiles are often large—hundreds of megabytes—making it impractical to send raw data. Opal addresses this by extracting only the most salient signals from each source, significantly reducing input size while preserving relevance. Even with compact prompts, model edits are untrustworthy unless the reasoning behind them is clear. Without understanding why a change was made or how it links to diagnostics, users cannot validate or trust the transformation. To make reasoning transparent, Opal uses belief tracing, which extracts token-level log-probabilities and reconstructs human-readable phrases by merging subword tokens based on a reference dictionary and a greedy sliding window.

Finally, Opal presents users with the complete diagnostic-to-edit chain, including the original and modified code, the rationale behind each change, any deferred suggestions, and the specific performance bottleneck that motivated the transformation. When a transformation is deemed unsafe, the LLM does not apply it but still recommends it with justification based on the diagnostics. These suggestions can serve as educational material for users or guide experts. To ensure reproducibility, we use low-temperature decoding (0–0.15). Although this paper focuses on GPU kernels, the methodology applies to CPUs and other accelerators. Opal can be easily extended to accommodate new sources and formats. Finally, while Opal currently uses a general-purpose LLM in the backend, users can substitute local or specialized models of their choice without any modification to the code–making Opal fully extensible and adaptable to evolving capabilities.

In summary, in this paper, we:

- **Introduce** Opal, a multi-source performance analytics-powered automated code optimization framework using Large Language Models (LLMs).
- **Construct** a token-aware prompt that yields responses that link each optimization to its diagnostic source.
- **Design** a method to infer the reasoning of an LLM to identify what influenced each decision.
- **Validate** Opal using extensive experiments across eight benchmarks on both NVIDIA and AMD GPUs, showing correctness and performance improvement, and explaining reasoning.
- **Implement** an end-to-end user interface for practitioners to explore, apply, and trust LLM-based optimizations, which will be made publicly available along with all data collected.

We also demonstrate that Opal's recommended optimizations to two HPC proxy applications—XSBench [43] and SW4lite [47]. Moreover, to demonstrate that Opal is not tied to a specific programming model, we select benchmarks with both CUDA and HIP implementations. Our study includes 8 real-world GPU applications,[2] evaluated across 68 input configurations and over 1640 experiments. These applications span a range of runtime bottlenecks, including memory-bound and compute-bound kernels. Despite these differences, all but one generated version passed benchmark-supplied correctness tests. While the exact transformations may vary between platforms, the diagnostic reasoning pipeline remains unchanged. Each optimization is generated in a few seconds and delivers up to 87.6% reduction in execution time, with a 33.3% average improvement across all trials. These results validate our core claim: Opal provides a generalizable, architecture-agnostic methodology for optimizing performance using analytics and LLMs.

Limitation To keep validation tractable, we synthesize optimizations from a single representative configuration per application. Although performance bottlenecks may vary across inputs, our experiments across 68 configurations show that Opal's recommendations generalize well when performance characteristics remain consistent. This paper builds and validates the foundation for broader support; in future work, we plan to cluster input regimes based on shared bottleneck patterns and apply Opal to a representative from each regime.

The rest of the paper is organized as follows: Section 2 presents high-level information about each of the analytics sources; Section 3 describes our approach including salient information identification, prompt template, and belief tracing in details; Section 4 describes the implementation-specific details along with how users can use Opal; Section 5 describes the benchmarks and experimental setup in details; Section 6 presents thorough evaluation. Finally, Section 7 presents related literature and Section 8 concludes.

2 Background

2.1 Roofline Analysis

To design effective optimization, it is required to know *how* an application behaves: whether it is compute-bound or memory-bound, for instance. By comparing achieved performance against arithmetic intensity, Roofline models can reveal fundamental performance bottlenecks [46]–critical information not easily inferred from isolated counters or stalls. Several prior works have leveraged Roofline analysis to guide manual GPU optimizations. Yang et al. [48] used hierarchical Roofline modeling to identify bandwidth-bound kernels and applied loop tiling for better L2 cache locality. Bauer et al. [8] applied Roofline-based insights to recommend texture memory and warp-specialized data movement. Lee et al. [30] proposed a Roofline-based data migration strategy for hybrid memories and Williams et al. [46] used structure-of-arrays layouts to improve coalescing. On AMD GPUs, Leinhauser et al. [31] used instruction-level Roofline analysis to guide LDS usage and memory coalescing. However, users need to be experts to infer which code transformation to apply given these insights.

[2]Applications part of Babelstream [16], HeCBench[27] and ECP proxy application suite [43, 47].

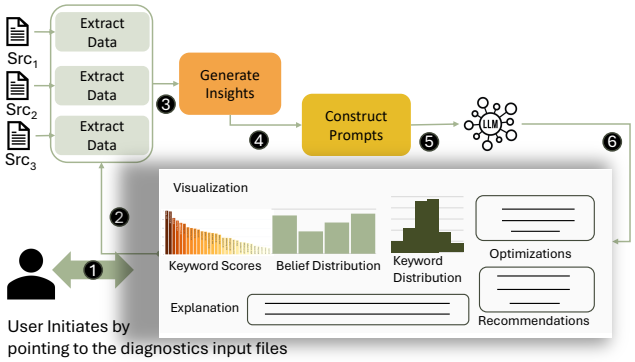

Figure 1: Overview of the Opal framework. Users select performance profiles, and Opal constructs structured prompts compatible with any LLM, including custom models.

2.2 PC Sampling-Based Analysis (PC)

Program Counter (PC) Sampling provides fine-grained insights into *where* in the source code performance stalls occur and *what* microarchitectural reasons cause them. Unlike Roofline models, PC Sampling attributes specific stall types (e.g., `stall_barrier`, `stall_memory_throttle`) to individual assembly instructions and their corresponding source lines. This precision has enabled tools such as GPU Performance Advisor [53] to trace 40% of stalls in a kernel to a single load and recommend targeted fixes. However, existing LLM-based tools such as [11, 50] do not integrate such instruction-level data into their reasoning. Opal fills this gap by injecting these stall names, their meanings and specific code in the identified line into the structured prompt described in Section 3.3 to guide the LLM to reason about micro-level bottlenecks. Still, PC Sampling data has limitations as it may miss rare stalls or input-dependent behaviors. This is why we augment these insights further using input-aware behavioral profiling via hardware counters (Section 3.2.3).

2.3 Application-Hardware Interaction Analysis (IA)

While Roofline analysis highlights whether a kernel is compute- or memory-bound at a high level [46], and PC Sampling pinpoints localized stalls at specific instruction lines [53], neither fully explains how performance is shaped by the interaction between the application and the architecture across varying configurations. This missing layer—application-specific, dynamic characterization—can only be captured by observing which hardware events consistently correlate with runtime degradation across multiple input sizes, thread block shapes, and control parameters. Opal addresses this gap by identifying application-hardware interactions that can explain the degradation. Hardware counters such as warp divergence, L2 cache misses, shared memory bank conflicts are critical for understanding why an application performs poorly. While prior work has shown their utility for isolated diagnosis [6, 17, 26], the challenge lies in automatically identifying which counters matter most, among hundreds exposed by modern architectures.

3 Our Approach

Figure 1 shows the entire methodology of Opal. The tasks can be decomposed into three parts: (1) generating important insights from different sources, (2) constructing prompts, (3) explaining LLM's reasoning using belief tracing.

3.1 Scope

Opal is designed to optimize individual GPU kernels. This scope aligns with prior work showing that kernel-level modeling is effective for isolating bottlenecks using Roofline analysis, hardware counters, and PC Sampling [24, 31, 46, 48, 54]. A study of 31 GPU workloads found that 70% of total execution time is spent in a single kernel [39], and commonly used profilers such as NVIDIA's Nsight Compute [40] and AMD's ROCm profiler [4] provide diagnostics at this level. Focusing on kernels ensures that optimizations remain interpretable and actionable. Opal currently supports performance profiles; extending support to handle time-series performance traces will require modifications to the `Extract Data` and `Generate Insight` modules in Figure 1.

3.2 Extract Data & Generate Insights

Since the size of the raw data generated by the runtime diagnostic tools can be hundreds of megabytes in size, a naive approach of simply sending all data to an LLM is impractical due to the limited token available for communication. This is why, Opal implements source-specific salient information extraction methodology for effectively communicate critical information in token-efficient manner. `Opal` prioritizes those performance problems with associated speedup suggestions or architectural bottlenecks, and integrates them into their respective `<*_summary>` sections of the prompt template in Listing included in 3.3.

3.2.1 Extract Insights from Roofline Data

The `<roofline_analysis_summary>` section is populated differently for each architecture due to variations in vendor tooling. For NVIDIA GPUs, `Opal` leverages the Roofline insights provided by Nsight Compute. These insights include both kernel-level metrics and accompanying diagnostic comments generated by the tool itself. For example, if Nsight Compute identifies that "this kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of this device," it may recommend checking scheduler or warp state statistics. `Opal` summarizes such rule-based diagnostics, prioritizes those with associated speedup suggestions or architectural bottlenecks, and integrates them into the `<roofline_analysis_summary>` section. This makes the prompt contextually rich and actionable without requiring the LLM to re-interpret raw numerical outputs.

In contrast, AMD Roofline reports are minimal and provide only numerical bandwidth and FLOPs data across memory levels (e.g., L1, L2, HBM, LDS) and operation types (e.g., FP32, FP64, MFMA). Lacking interpretive guidance, Opal (1) identifies underutilized components by comparing observed and peak values, and (2) supplements this with insights from AMD optimization guides and modeling studies [2]. This enables the prompt to convey directional performance cues, even without automated analysis.

3.2.2 Extract Insights from Stall Information

To pinpoint where in the application code performance bottlenecks manifest, Opal integrates program counter (PC) Sampling data. While hardware counters explain what architectural inefficiencies

exist and why they arise, PC Sampling identifies where these issues occur in the source code, enabling more actionable and localized optimizations. Opal analyzes assembly instruction-level samples collected via Nsight Systems and maps them to source-level line numbers using debug symbols. For each kernel, it identifies lines where individual stalls exceed a 10% threshold and attributes those stall causes (e.g., memory dependency, execution dependency, pipeline busy). Rather than directly including PC Sampling output, Opal summarizes the root stall occurrences, line numbers, and adds that to the prompt in the designated <PC_sampling_summary> section. For instance, raw PC Sampling output for `Accuracy` (41 lines of code) is 990 MB. After pre-processing, the data size reduces to 10 MB, and Opal's summarization compresses it further to under 80 lines (6 KB). This 99.9% reduction preserves actionable stall insights while being under LLM's token limit.

3.2.3 Extract Insights from Hardware Counters

Modern architectures expose hundreds of hardware performance counters. Many are redundant, correlated, or irrelevant to optimization. Since in this study, our goal is to validate whether important signals exist across any subset, we collect the full set of available counters to avoid missing potentially useful ones. However, in practice, users do not need to collect every counter. If they already know a relevant subset based on domain knowledge or architectural experience, Opal can still operate on that list and systematically reduce it further to just the counters that can explain how performance changes across configurations.

We frame this filtering task as a sparse feature selection problem. Each run of a kernel produces a vector of runtime measurements and a corresponding vector of hardware counter values. Together, they form a matrix $\mathbf{D} \in \mathbb{R}^{N \times C}$, where N is the number of runs and C is the number of counters, and a target vector $\mathbf{t} \in \mathbb{R}^N$ representing the runtime or any performance metric. We solve for a sparse weight vector $\mathbf{a}$ that identifies a small set of counters that best explain $\mathbf{t}$ using the formation in Equation 1. To avoid overfitting and isolate only the most informative counters, we solve for a sparse weight vector $\mathbf{a}$ that minimizes the prediction error between observed performance $\mathbf{t}$ and the counter matrix $\mathbf{D}$. The sparsity level is controlled by κ, which limits the number of non-zero entries in $\mathbf{a}$ (Equation 1). Higher sparsity causes a reduction in the number of insights provided to the LLM.

$$\min_{\mathbf{a}} \|\mathbf{t} - \mathbf{D}\mathbf{a}\|_2^2 \quad \text{s.t. } \|\mathbf{a}\|_0 \leq \kappa \tag{1}$$

Because this problem is NP-hard, we use Ensemble Orthogonal Matching Pursuit (OMP), a randomized greedy algorithm. At each step, instead of picking just the most correlated counter, Ensemble OMP selects the top τ candidates, samples one at random based on their correlation strengths, and repeats. This ensemble approach explores multiple plausible solutions and avoids the local bias of standard greedy methods.

Finally, Opal selects the top-k (this work uses $k = 5$) influential counters and summarizes their meanings in natural language. In our experiments, we observed no significant difference in performance outcomes between using the top-3 versus top-5 counters, so we chose $k = 5$ to balance completeness with brevity. These descriptions, generated once per architecture offline by mining vendor documentation, are stored in a reusable dictionary. The top-k summaries are then embedded into the prompt to provide context, e.g.: "`l1tex__data_bank_conflicts` tracks shared memory bank conflicts, typically caused by misaligned or reduction operations, increasing latency." For this step, we build on the publicly available Dashing repository [21, 23] and will make the dictionaries for NVIDIA and AMD GPUs publicly available.

3.3 Prompt Construction

To ensure consistent and parseable responses, Opal uses a prompt template with programmatically populated placeholders. As shown in Listing included in 3.3, each prompt begins with the unoptimized kernel code annotated with line numbers. This section is followed by structured summaries from each diagnostic source. A final instruction block then directs the LLM to (1) generate optimized code, (2) explain each change by linking it to a specific diagnostic insight, and (3) list additional optimizations that were considered but not applied, along with the corresponding evidence. These deferred suggestions serve as actionable leads for human experts. Opal parses the response and presents the optimizations and suggestions along with their diagnostic rationale to user through an interactive user interface.

Listing 1: The prompt template used by `Opal`

```
code:
  snippet: "<source_code_with_line_numbers>"

roofline_summary: <roofline analysis summary>

stall_analysis_summary: <stall analysis summary>

key_hardware_events: <counter analysis summary>

task_instruction: You are an HPC performance optimization expert. Optimize
   the provided <programming language> code specifically for <architecture>
   on input configuration <input config>. Clearly reference the diagnostic
    data by number or quoted text in each inline comment.

Required Output Format:

1. Provide the complete optimized code wrapped within ```cpp code blocks```.
2. After the code, list applied changes:
   optimizations = [ {'lines': [line_numbers], 'reason': 'justification from
    diagnostic'}, ... ]
3. List suggestions not applied: suggested_but_not_applied =
   [ {'lines': [line_numbers], 'reason': 'diagnostic, but deferred due to
    uncertainty'}, ... ]
4. Do not include any additional explanation beyond the code and the two lists.
```

3.4 Explain LLM's Reasoning using Belief Tracing

To explain why Opal suggests a particular code transformation, users need visibility into the model's internal reasoning. LLMs produce output one token at a time by selecting the most likely next token based on prior context. Internally, the model assigns a probability $P(x)$ to every possible next token x in its vocabulary, and selects the one with the highest likelihood. For instance, when generating an explanation about optimizing memory access in the `Accuracy` kernel, the model may generate a sequence of tokens that together form the phrase `memory coalescing`. This occurs because it has learned from the literature that "memory" and "coalescing" together describe a meaningful concept in GPU optimization. However, models do not explicitly tell us which high-level phrases they

considered important. Instead, they emit subword tokens such as "co", "alesc", and "ing", without revealing how those pieces combine.

Calculate belief scores for `logprobes` tokens To provide explainability to Opal's reasoning, we introduce a mechanism called belief tracing. Specifically, Opal uses the `logprobes` API to extract the log-probability of each generated token. However, raw log-probabilities are difficult to interpret: they are negative, unbounded, and inversely correlated with confidence. Lower log-probability indicates lower confidence. To convert this signal into a more intuitive scale, we define a belief score $B(x)$ for each token in Equation 2:

$$B(x) = \exp(-\alpha \cdot \log P(x)) \tag{2}$$

where α is a positive scaling constant (we use $\alpha = 2$). This transformation maps the log-probabilities to the $[0, 1]$ range, where values closer to 1 indicate higher model confidence. It preserves the relative differences across tokens while inverting the log scale, so that less likely (more surprising) tokens receive lower belief scores. These belief scores form the basis for identifying which keywords the LLM focused on during code generation.

Construct keywords from tokens Since the generated tokens are subwords, next, we construct keywords from these tokens by using a reference dictionary $\mathcal{R}$ created from the input prompt, extracting all n-grams ($n \leq 4$). We then scan the logprobe token sequence using a greedy sliding window, merging the longest contiguous span that appears in $\mathcal{R}$. If a match is found, we assign the belief score of the phrase $w = \{t_1, \ldots, t_n\}$ in Equation 3:

$$B(w) = \prod_{i=1}^{n} B(t_i) \tag{3}$$

This multiplicative aggregation captures compound reasoning–phrases are assigned high belief only if all component tokens are influential. If no match is found, we retain the original single-token belief. This reconstruction step thus calculates importance of domain-specific keywords in $\mathcal{R}$ for explanation.

Filtering Finally, we filter all punctuation, numeric fragments, or partial words and those: (1) with non-alphabetic characters only, (2) with length ≤ 2 unless alphabetic, and (3) not in $\mathcal{R}$. Then, we apply logarithmic scaling to compress large values and amplify small differences by adding a small constant ($\epsilon = 10^{-7}$) to avoid undefined behavior for zero-valued beliefs. and re-normalized to $[0, 1]$ using min-max scaling. Section 6.5 shows an example of how these keywords and their beliefs explain reasoning.

4 The Opal Framework

Figure 1 illustrates the end-to-end workflow of Opal. Users start by selecting the source code and any number of diagnostic inputs via a lightweight Streamlit-based dashboard. Currently supported inputs include PC Sampling logs, hardware counters, and Roofline analysis, all in JSON format (①). Data collection is performed offline using standard profiling tools commonly used in HPC workflows [1, 24, 46]. After file selection, Opal invokes a separate `Extract Data` module for each input source (② in the figure). Each module parses source-specific outputs into structured format, as described in: (1) Section 3.2.2 for PC Sampling logs, (2) Section 3.2.1 for Roofline analysis, and (3) Section 3.2.3 for hardware counter attribution. These structures are passed to the `Generate Insights` module (Step ③), which summarizes and compresses each input into prompts. This insight reduction pipeline, detailed in Section 3.2, ensures the prompt remains actionable while avoiding verbosity.

Next, the Construct Prompts module (Step ④) combines the unoptimized kernel and diagnostic summaries into a structured prompt following a template (Listing included in 3.3). This step, described in Section 3.3, links each performance issue to corresponding parts of the code. The prompt is then sent to an LLM with log probability tracking enabled (temperature=0.15), as shown in Step ⑤. The LLM returns three outputs: (1) an optimized kernel, (2) explanations justifying each transformation, and (3) deferred suggestions that were considered but not applied due to uncertainty.

4.1 User Interaction and Visualization

Opal displays the results through an interactive web dashboard (Step ⑥). Users can examine: (1) the optimized code, with each modification linked to its triggering diagnostic, (2) explanations for each change, including rationale for rejected transformations, and (3) visualizations that expose how the Opal reached its decisions. The visualization panel includes: (1) a flame-style Keyword Score bar chart showing belief scores for the most influential prompt tokens, A Belief Distribution histogram indicating Opal's overall confidence (calculated in Section 3.4), and a Keyword Distribution chart summarizing recurring reasoning patterns. These are derived from belief tracing (Section 3.4), which reconstructs meaningful phrases from token-level logprobs and highlights the reasoning path the model followed. Every decision is traceable—down to the diagnostic message and the specific lines of code affected—allowing users to validate, override, or extend the transformation.

Opal is modular by design. To support new tools such as HPCToolkit [1], users only need to extend the `Extract Data` module. All other components remain unchanged. In future, Opal can be integrated into a multi-agent AI framework, where agents autonomously run measurement tools, process the logs, and constructs the prompts in a collaborative manner. This will unify data collection and analysis.

5 Evaluation Setup

5.1 Hardware and Environment

We evaluate Opal on two GPU platforms: (1) NVIDIA H100 GPUs and (2) AMD MI 210 on an unnamed cluster (hidden for double blind review). Each system provides modern CPUs, high-bandwidth interconnects, and GPU-optimized software stacks. Experiments use CUDA-12.3, Nsight Compute-2025.1.1.0, ROCm-rocm/6.3.2, Omniperf-2.0.1 and GPT-4o via the OpenAI API. Every configuration is run 10 times, and results report the mean with error bars.

5.2 Benchmarks and Configurations

We use four GPU applications comprising eight kernels that stress distinct bottlenecks: memory-bound, compute-bound, and stall-sensitive behavior. Table 1 summarizes each application's LOC (lines of code), parameter space and default configurations. These benchmarks include simple kernels (`Copy`, `Mult`, `Add`, `Triad`, `Dot`),

Table 1: Benchmark applications, their size, and tunable configurations.

App.	LOC	Configuration Description
`Copy`	2	ARRAY_SIZE∈[2000896,4000768,6000640], `num_times`∈[100,300], step=100; Default=(2000896,100).
`Mul`	3	
`Add`	2	
`Triad`	3	ARRAY_SIZE: dataset size;
`Dot`	19	`num_times`: repetition count.
Shmemench [38]	35	`rep`∈[300,1500], step=300; Default=600. `rep`: kernel repetition count.
`Accuracy` [27]	41	(`nrows`, `ndims`, `top_k`, `rep`)=8192 × [5000,10000] ×[10,20] × [100,300]; Default=(8192,5000,10,100). `nrows`: prediction vectors; `ndims`: dimensions/vector; `top_k`: threshold; `rep`: repetition count.
`Sobol` [27]	195	(`n_vectors`, `n_dimensions`, `repeat`)=[10K,1M] × [1K,10K] × 100; Default=(10K,1000,100). `n_vectors`: vectors generated; `n_dimensions`: dimensions/vector; `repeat`: repetition count.
`addsgd4_SM` (SW4Lite [47])	149	(`input_datasets`)= [pointsource, uni, uni_rev, skinny, gaussianHill]; Default=(pointsource).
XSBench [43]	81	(`s`, `G`)=[small, large] × [unionized, nuclide, hash]; Default=(large, unionized). `s`: problem size; `G`: grid search type.

`Shmembench` for shared-memory operations, the compute-intensive `Accuracy` kernel, and the memory-sensitive `Sobol` kernel. These diverse workloads ensure a thorough evaluation of each diagnostic source's impact on LLM-driven optimization. The chosen benchmarks specifically have both CUDA and HIP implementations, enabling a cross-platform validation of our methodology.

5.3 Data Collection Tools

Roofline Analysis For NVIDIA GPUs, we use Nsight Compute (`ncu`) to collect Roofline metrics, including arithmetic intensity, throughput, and diagnostic messages: `ncu --csv --import <output_file> --page details > <roofline_file>.csv`. AMD GPUs provide numeric Roofline data (bandwidth, FLOPs) via rocprof, without interpretive diagnostics. We manually identify bottlenecks by comparing these metrics to theoretical peaks, supplemented by AMD optimization guides [2].

PC Sampling PC Sampling requires compiling the application with debug information enabled using either the `-G` or `-lineinfo` flag. PC Sampling support is currently unavailable for AMD GPUs but can be integrated once vendor support becomes available.

Hardware Performance Counters We use NVIDIA's Nsight Compute tool to collect all available counters using the full preset (`-set full`) flag using the following command: `ncu --target-processes all -set full -o <output_file> ./<app> <args>` This allows users to capture the entire list of counters without selecting specific ones manually. Because this command is backend-agnostic with respect to CUDA versions, it supports forward compatibility. For AMD GPUs, similar counter data is collected using Omniperf [3]. Advanced users may specify subsets to streamline data collection. Opal then applies the eOMP algorithm to identify important counters (Section 3.2.3).

5.4 Baselines

We compare optimizations generated by Opal across seven scenarios against the performance of the unmodified code. The scenarios include: (1) PC Sampling (PC), (2) Importance Analysis (IA), (3) Roofline, (4) PC Sampling with Importance Analysis (PC+IA), (5) PC Sampling with Roofline (PC+Roofline), (6) Importance Analysis with Roofline (IA+Roofline), and (7) PC Sampling with Importance Analysis and Roofline (PC+IA+Roofline). Additionally, to demonstrate the benefit and applicability of Opal to HPC, we select two proxy applications—XSBench [43] and a kernel named `addsgd4_SM` from the SW4Lite benchmark suite [47]. We do not compare with hand tuned baselines as they depend strongly on domain expertise. We also evaluated an additional baseline, simply uploading the code to ChatGPT-4o. However, under our minimal *temperature* setting, the Opal did not receive any optimizations, and thus, we exclude this baseline from our comparison.

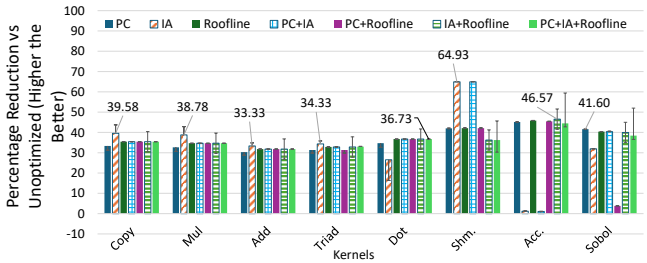

Figure 2: Ablation study of the impact of diagnostic sources on optimization performance. Error bars represent performance variability across repeated runs. IA alone achieves significant single-source gains, notably 64.93% on `Shmembench`. PC+IA consistently delivers stable optimizations across kernels. `Sobol` shows substantial variability, indicating that the type of information can impact the effectiveness of code transformation.

5.5 Metrics Collected

We collect the following metrics per application and for a selected set of configurations to keep the number of experiments tractable. We measure performance improvements primarily using wall-clock execution time. To validate correctness, every optimized kernel is tested against benchmark-supplied validation tests. **Notably, 100% of the LLM-generated code transformations passed these checks using our current prompts**. Additionally, we present detailed validations of Roofline insights, PC Sampling data, and key hardware counter metrics for the optimized versions of a selected set of kernel. We also analyze keyword belief scores to explain the LLM's reasoning processes.

6 Results

The primary goal of our evaluation is to test the hypothesis that integrating performance insights into structured prompts enables LLMs to produce code transformations that directly target observed bottlenecks and improve runtime. To assess this hypothesis, we:

- **Quantify** the contribution of each insight source via ablation.
- **Evaluate** the effectiveness of code transformations across many configurations.
- **Validate** that the bottlenecks actually get fixed.
- **Generalize** Opal's approach to AMD GPUs.
- **Explain** Opal's reasoning.
- **Demonstrate** on HPC proxy applications.

Table 2: Default configuration and corresponding optimization summary for each benchmark on NVIDIA H100 GPUs.

App.	Default Config.	Optimization Summary
Copy	(2000896, 100)	Replaced cudaMemcpy with cudaDeviceSynchronize for managed memory to reduce overhead. Added boundary checking clause.
Mul	(2000896, 100)	Same as Copy
Add	(2000896, 100)	Same as Copy
Triad	(2000896, 100)	Same as Copy
Dot	(2000896, 100)	Same as Copy
Shmemench	600	Simplified vector assignment (line 28) to reduce instruction count; removed __threadfence_block() comment (line 48) to reduce MIO pipeline stalls.
Accuracy	(8192,5000,10,100)	Added __restrict__ to pointers to reduce uncoalesced loads and enable alias-free memory access optimization.
Sobol	(10K,1000,100)	Replaced float conversion with __uint2float_rn at lines 29 and 50 to reduce latency and improve throughput; increased block size to 128 at line 66 to improve occupancy and GPU utilization.
addsgd4_SM	pointsource	Unrolled loop and removed array offset function calls (lines 12702–12722) to reduce register pressure and improve locality.
XSBench	(large, unionized)	Increased block size to 512 to improve occupancy (line 5); added shared memory to reduce global memory access (line 6); reorganized memory access for coalescing (line 7); reduced register usage to boost occupancy (line 8); used warp-level primitives to minimize divergence (line 9); replaced double with single precision for faster computation (line 10); unrolled loops to reduce overhead and increase throughput (line 11).

6.1 Quantify the contribution of each insight source via ablation

The goal of this experiment is to quantify how much each performance insight contributes individually and in combination to guiding LLMs toward producing faster code. Figure 2 shows results for eight GPU kernels on an NVIDIA H100, comparing optimizations generated using single sources, pairwise combinations, and all three sources combined. The X-axis lists applications, and the Y-axis shows the average percentage execution-time reduction relative to the unoptimized baseline, measured over 10 runs. Higher bars represent more effective LLM-guided optimizations. Error bars indicate variability: narrow bars imply consistent improvements, while wide bars highlight cases with significant fluctuations.

We restrict this study to one default configuration per application (defaults marked in Table 1) to keep the number of experiments tractable. Including all analysis sources across multiple configurations of multiple applications would exponentially increase the number of experiments. Section 6.2 investigates if recommended optimizations for the default configuration remains effectively for other configurations. Table 2 summarizes all the optimizations automatically applied to the benchmark applications.

Key observations From Figure 2, we observe that: (1) Each source, when used independently, provides measurable average performance gains across all but one kernels (between 1.2 – 64.93%). (2) Double source improvements range from 1.0 – 64.93%. (3) The full combination of all three sources (`PC + IA + Roofline`) achieves the most consistent gains across all applications. The average range varies from 31.8 – 44.49%. (4) Some of the optimizations cause run to run variability. Further investigation shows that

Not all optimizations are safe, guidance helps fix it All but one generated code variants (out of 1640 experiments) passed benchmark-supplied validation checks for output correctness. In that one case, Opal incorrectly suggested `__int_as_float` to reduce memory stalls, leading to test failure. The root cause was a vague counter description that failed to warn against unsafe casting. After refining the explanation to highlight bandwidth saturation risks and explicitly discourage this intrinsic, Opal consistently chose the correct alternative, `__uint2float_rn`, eliminating correctness issues and runtime variability.

Effectiveness of Insight Types by Bottleneck Class

Not all performance insight sources contribute equally; their usefulness depends on the nature of the application bottleneck. As shown in Figure 2, the `accuracy` kernel benefits significantly from **Roofline analysis and PC Sampling**, each delivering over 45% average execution time reduction. In contrast, IA alone yields negligible gain (1.2%).

This contrast stems from the type of bottlenecks each source reveals. Roofline analysis identified key architectural inefficiencies in the `accuracy` kernel—underutilized compute pipelines and poor DRAM fetch granularity—while PC Sampling exposed persistent stalls such as `stall_wait` and `stall_membar`. Both of these signal memory-level throttling. The hardware counter IA, on the other hand, surfaced only three moderately weighted events, which primarily described shared memory usage and L1TEX-level store pressure—important but not dominant.

This pattern suggests that macro-level models (like Roofline) and line-level stall diagnostics (from PC Sampling) are more effective when memory coalescing, warp scheduling, or pipeline utilization is the core issue. In such cases, these sources guide Opal to concrete, literature-supported transformations such as pointer alias removal or loop restructuring. We validate one such transformation in detail below.

Why IA underperformed for `Accuracy` Figure 2 shows that the benefit of using hardware counter analysis alone was only 1.2% for `Accuracy`. This is because there were only three moderately important hardware events for `Accuracy`, primarily pointing to L1TEX store pressure and shared memory underuse. Addressing these would require invasive changes to data layout or memory hierarchy–transformations that are risky and often workload-specific. Hence, Opal refrained from applying these transformations and instead provided guidance to a user.

6.2 Evaluate the effectiveness of optimizations across many configurations

The objective of this experiment is to assess whether Opal 's code transformation decided based on a single configuration generalize across diverse input sizes and configurations. As listed in Table 2, we evaluate eight kernels across 68 total configurations. Each run compares the unoptimized version to the Opal-optimized version generated using analytics from the default configuration. Figure 3 plots the percentage reduction in execution time: the X-axis shows each configuration; the Y-axis shows the percentage reduction in runtime relative to the unoptimized baseline.

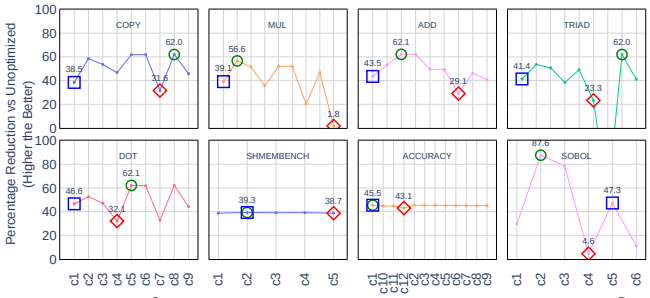

Figure 3: Performance improvements across input configurations. Blue rectangles represent improvements for default configuration (Table 1), green circles indicate maximum improvement, and red diamonds indicate minimum improvement. This figure shows that optimizations generated for one configuration also yield performance improvements across other configurations, however, the margin varies across applications. `Accuracy` achieves the most stable gains (43.1%–45.5%), while that for `Sobol` varies widely (4.6%–87.6%).

Key observations From Figure 3, we can observe that: (1) reduction in execution time persist across all but 1 out of 68 configurations across 8 applications, demonstrating an overall generalizability of the observation that multi-source analytics insights-guided code transformations suggested by Opal can be a great starting point and a lot better than starting from scratch. (2) The only configuration for `Triad` that achieved no benefit was when the kernel ran with an array size of 6000640 and 100 repetitions of kernels. (3) Kernels like `Shmembench` (up to 69.10%) and `Sobol` (up to 87.60%) see the largest gains, likely due to recurring bottlenecks across input ranges that let Opal generalize effectively. However, the improvements for `Sobol` drop for $C4$, respectively, suggesting input-sensitive behavior where bottlenecks shift with configuration. (4) In contrast, `Accuracy` consistently improves (43.1%–45.5%) across all inputs. The applied transformation (`__restrict__`) targets aliasing-related stalls that persist regardless of scale, indicating stable bottlenecks and validating that optimizations from one configuration can transfer when performance characteristics remain consistent.

6.3 Validate whether post-optimization performance aligns with Opal's stated rationale

The objective of this experiment is to assess whether Opal's code transformations mitigate the performance bottlenecks identified in the analysis phase. To keep the validation tractable, we focus on the two largest kernels, `Accuracy` and `Sobol`, using their default configurations. We re-run all analysis on both unoptimized and optimized versions to examine changes in key performance characteristics. These kernels expose different types of bottlenecks, making them ideal for evaluating whether Opal's choices align with known optimization strategies. Validation is considered successful if the problematic behaviors reduce and if the explanations are consistent with published literature.

6.3.1 Validating Bottleneck Elimination in `Accuracy`

Performance profiling revealed poor warp readiness (0.16 eligible warps per cycle), inefficient DRAM fetch (2.7 sectors per 128-byte cache line), and persistent memory stalls, including `stall_barrier` and `stall_membar`. To address these, Opal added the `__restrict__` keyword to `Xdata` and `labelData`. This keyword disambiguates pointer aliasing, which makes the compiler safely reorder memory accesses and apply coalescing optimizations. Published literature confirms that this keyword is used to enable contiguous memory access, which reduces transactional overhead [12, 41, 46]. The post-optimization metrics in Figure 4 show that (1) scheduler efficiency improves from one instruction every 6.4 cycles to 3.5, (2) eligible warp rate increases from 0.16 to 0.40, and (3) DRAM fetch efficiency improves as indicated by a lower L2 miss-to-sector ratio. These improvements confirm that the LLM correctly identified a transformation that aligns with expert practices–validating that Opal's reasoning is both explainable and effective.

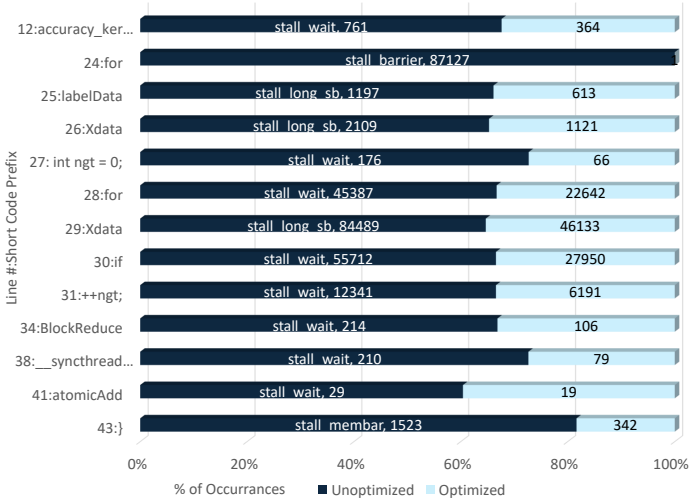

Figure 4: Comparison of stall occurrences between unoptimized (left) and optimized (right) `Accuracy` kernels. The Y-axis lists line numbers and shortened code snippets. Each bar is labeled by the dominant stall type. At line 6, `stall_wait` occurrences decrease from $45,387 \rightarrow 22,642$; similar significant reductions occur at lines $29 \rightarrow 31$, validating the effectiveness of targeted optimizations.

6.3.2 Validating Bottleneck Elimination in `Sobol`

Similarly, we also validate the same for `Sobol`. For `Sobol`, the PC Sampling data identified three issues: low occupancy, MIO stalls linked to shared memory, and expensive float conversion operations marked by high `stall_math_ratio`. In response, Opal replaced `(float)X` with the intrinsic `__uint2float_rn(X)` in two critical lines. This transformation is supported by CUDA literature advocating the use of intrinsics to bypass normalization overhead. Post-optimization metrics (not shown) confirm the intended effect: occupancy increases from 25.6% to 48.4%, scheduler throughput improves from 1.9 to 1.6 cycles/issue, and stall types shift from MIO to fixed-latency dependencies–typical of highly optimized code. These changes validate that the LLM's recommendation was not only guided by real profiling data, but also converged on an established best-practice transformation shown to reduce instruction latency in throughput-bound kernels [19, 41].

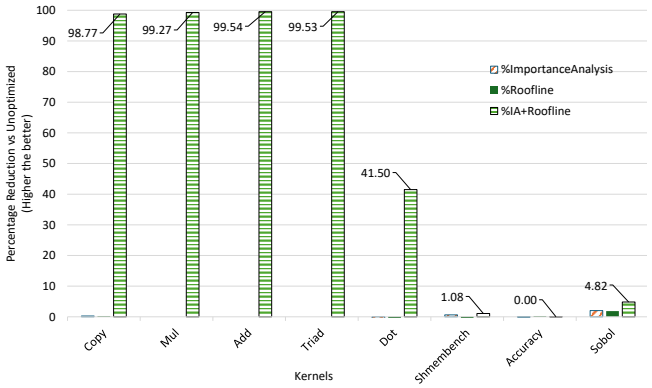

Figure 5: Ablation study on source contributions to optimization performance for HIP applications. We observe massive improvements for Babelstream kernels (41.5 to 99.27%) using IA and Roofline analysis together as performance insights. For all other kernels, there is no improvement (some improvements are shown, but that is purely because of GPU execution uncertainty).

6.4 Generalize Opal's Approach to AMD GPUs

The objective of this experiment is to assess whether Opal generalizes to HIP applications and AMD GPUs. We replicate the ablation study from our CUDA evaluation using HIP versions of the same kernels. Performance data is collected on AMD MI210 GPUs using Omniperf. Due to tool limitations, PC Sampling is currently unsupported, and Roofline analysis lacks diagnostic commentary. Instead, we extract raw Roofline metrics—including **HBM Bandwidth**, **L2 Bandwidth**, **L1 Bandwidth**, **LDS Bandwidth**, and peak compute throughput across **FP32 VALU**, **FP32 MFMA**, **FP16 MFMA**, and **INT8 MFMA**. These values are passed into the prompt using the same structured format as in the CUDA pipeline.

Key observations From Figure 5, we observe: (1) Single-source prompts (e.g., Roofline-only or counter-only) fail to yield performance gains across all HIP kernels. This is because Omniperf outputs only raw numeric values without descriptive bottleneck summaries. In the absence of contextual information—such as whether a kernel is memory- or compute-bound—the LLM struggles to reason about actionable optimizations. Unlike Nsight Compute, which provides direct language-level insights (e.g., "kernel underutilizing SM"), Omniperf requires the LLM to infer everything from sparse numbers, which limits its effectiveness.

(2) In contrast, combining both Roofline and counter-based insights leads to substantial improvements in `Copy` (41.50%), `Mult` (78.43%), `Add` (93.00%), `Triad` (99.54%), and `Dot` (86.31%). These improvements are largely due to a consistent LLM-generated fix: explicitly setting `hipStreamDefault` during kernel launch. This parameter was omitted in the original HIP code, causing unintended synchronous behavior. By suggesting this fix, the LLM indirectly resolved a performance bug—demonstrating that Opal can help detect general inefficiencies even without detailed profiling context. While the fix resembles static correction, the fact that it emerged from performance diagnostics suggests that even limited context can trigger meaningful repairs.

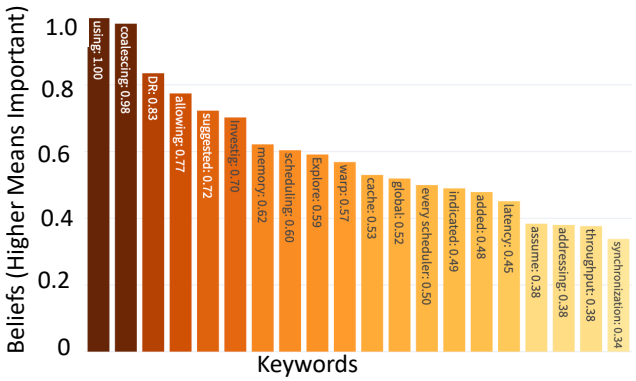

Figure 6: Top 20 keywords and their relative importance. These keywords were deemed important while recommending to use `__restrict__` for the `Accuracy` kernel.

(3) For more complex applications like `Sobol` and `Accuracy`, the LLM suggests optimizations using CUDA-specific syntax (e.g., `__restrict__`, `threadIdx.x`) that are not directly applicable in HIP. As a result, these suggestions either do not compile or fail to yield measurable performance benefits. This reflects a current limitation in the LLM's training data: it lacks exposure to HIP-specific optimization idioms and AMD-specific hardware behaviors. For example, the LLM may recommend shared memory tuning or warp-level primitives that are implemented differently in HIP or require architecture-specific flags. These gaps do not reflect a flaw in Opal itself, but rather in the general-purpose model's hardware-specific knowledge. We believe this limitation can be addressed in future work by integrating more HIP and AMD-specific performance data via retrieval-augmented generation (RAG) [32], thereby improving the LLM's architectural grounding while preserving the prompt-driven interface.

6.5 Explain Opal's Reasoning

The objective of this experiment is to explain the reasoning process behind the LLM's code transformation decisions. We analyze belief scores computed from the logprobabilities returned by the LLM during inference. These scores reflect which keywords in the prompt had the greatest influence on its final recommendation. To understand why the LLM inserted the `__restrict__` qualifier in the `Accuracy` kernel, we examine its belief distribution in Figure 6. High-scoring keywords include *coalescing*, *pointers*, *memory access*, and *alias*—all terms associated with memory layout and bandwidth utilization. These directly relate to the cache line underutilization problem identified by the Roofline analysis. Interestingly, while `__restrict__` itself did not appear in the top-ranked tokens, the LLM inferred aliasing as the root cause, linking it to known solutions from the literature. This multi-step inference supports our central claim that by providing structured performance insights to LLMs, even general-purpose LLMs can reason through the implications and synthesize relevant code optimizations known in the literature.

6.6 Demonstrate on HPC Proxy Applications

The objective of this experiment is to evaluate the applicability of Opal on real-world HPC proxy applications that reflect the computational patterns found in national security and scientific workloads. We selected two widely used proxy applications: `XSBench` and `sw4lite`. `XSBench` models the core loop of Monte Carlo neutron transport, a critical component in nuclear reactor simulation and radiation shielding analysis. `sw4lite` captures stencil-based wave propagation, a common kernel in seismic inversion and earthquake modeling used for geophysical exploration and hazard assessment. These codes represent two distinct but foundational computation patterns—irregular lookups and structured stencils—found in mission-critical applications ranging from nuclear safety to homeland infrastructure resilience.

For this experiment, we used only two performance insight sources: Roofline analysis and IA. While it is technically possible to apply PC Sampling to these codes by restructuring them into single-file kernels, doing so would require significant manual effort and restructuring—an unrealistic burden in practice. Instead, we chose to investigate whether Opal can still generate meaningful optimizations using partial diagnostics, reflecting a more practical deployment scenario for large-scale scientific applications.

Key observations For `XSBench`, Opal increased block size and introduced shared memory usage to reduce global memory pressure and improve data locality. For `sw4lite`, it unrolled a compute-intensive loop and removed offset computations to reduce register pressure and improve instruction throughput. These optimizations were guided by the combination of Roofline analysis and IA, as PC Sampling could not be applied due to tool limitations on multi-file applications.

Although PC Sampling was unavailable, these results show that Opal can still operate effectively using partial diagnostics. The improvements—ranging from 0.00% to 11.58% for `XSBench` and 13.15% to 14.56% for `sw4lite`—suggest that combining architectural modeling with low-level interaction profiling is sufficient to trigger meaningful transformations in large, real-world codebases.

6.7 Discussions

Overhead The overhead of recommending code transformations is a few seconds for all experiments. However, the data collection time can be extensive for PC Sampling source. Since these tools and methods are established in the community [2, 7, 21, 23–25, 28–30, 46, 49, 52, 53], we assume that users interested in optimizing their most time consuming kernel's performance will start with a subset of these tools. Perhaps future research can synthesize these profiles for different regimes from a few runs instead of repetitively running the code.

As discussed in Section 6.1, Opal's explanations made it possible to identify an unsafe transformation and swap it with a safer one. With Opal, users can easily validate, revert, or substitute code transformations without needing to design them from scratch.

One size does not fit all Some GPU kernels (e.g., `accuracy`) show consistent bottlenecks across configurations, while others may vary based on input regimes. In this work, to keep validation tractable, we use Opal to optimize for one configuration per application. Despite this simplification, we observe that these optimizations generalize across configurations. In the future, we will building on Opal to investigate novel approaches for clustering input regimes based on bottlenecks and generating tailored optimizations per group. Without Opal, mapping from bottlenecks to code edits would remain a manual trial-and-error process.

More data is not always better: prompt tuning matters Our experiments show that combining multiple analysis sources helps. However, longer prompts often cause LLMs to focus on less relevant bottlenecks. This issue stems from prompt token limits and is compounded by the fact that hardware counters often lack clear, descriptive explanations. Even so, Opal consistently improves performance across applications. These findings highlight that success depends not just on the data, but on the quality of the insights extracted by Opal. In the future, we will explore adaptive prompting strategies that identify the most relevant diagnostics across sources.

7 Related Work

Performance analysis methodologies such as Roofline modeling [2, 20, 28, 30, 46], PC Sampling [10, 51–53] and hardware performance counters [22, 25, 44] help identify architectural inefficiencies (e.g., memory bottlenecks, TLB misses) but require manual interpretation and expert-guided code tuning. Similarly, PC Sampling tools such as Nsight [40, 53] localize stalls to source lines but leave the optimization step to the user. While these approaches are effective at diagnosis, they do not automate the path to resolution. Opal provides an extendable framework that complements this ecosystem by structurally translating these diagnostics into insights that can guide LLMs to generate actionable code changes, thus closing the loop between profiling and code transformation.

General-purpose models such as Codex [11], StarCoder [34], and CodeLlama [42] are trained for code completion and refactoring but lack awareness of runtime performance bottlenecks. Similarly, Star-Agents [50] learn from version histories but do not incorporate profiling data. As a result, these models tend to propose generic transformations that are disconnected from actual hardware constraints. Opal addresses this gap by capitalizing on the reasoning capability of the LLMs to connect optimization strategies to bottlenecks commonly found in the literature.

Autotuners such as OpenTuner [5] and learning-based tools such as OptFormer [35] automate code-space exploration via heuristic search or policy learning. However, they rely on manual knob definition or optimize for proxy objectives such as loop unrolling depth or binary size. In contrast, Opal derives optimization intent directly from multi-source performance diagnostics and maps them to meaningful code transformations using off-the-shelf LLMs. To our knowledge, Opal is the first framework to integrate diverse analytic sources into an automated code optimization engine.

8 Conclusions

Despite the availability of many HPC performance analysis tools, the final step of translating diagnostic outputs into actionable optimizations remains manual and expert-driven. We introduce Opal

to bridge this gap by transforming multi-source performance insights into structured prompts for LLMs. These natural language prompts enable LLMs to reason about which code transformation to apply based on their knowledge of the literature. Across ~1640 experiments, Opal achieved performance improvements ranging from 1.76% to 87.6%, with only 1 out of 1640 experiments failing to pass the validation tests, greatly highlighting that Opal is able to use the LLMs as a reasoning engine. By combining performance analytics with LLM-guided transformation, Opal takes a significant step toward democratizing expert-level GPU performance engineering. In future work, we aim to extend Opal into a fully integrated multi-agentic system that automates the invocation and execution of performance analysis tools, completing the loop from profiling to synthesis.

References

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. 2010. HPCTOOLKIT: Tools for Performance Analysis of Optimized Parallel Programs Http://Hpctoolkit.Org. *Concurr. Comput. : Pract. Exper.* 22, 6 (April 2010), 685–701. doi:10.1002/cpe.v22:6

[2] Advanced Micro Devices, Inc. 2024. MI200 Performance Counters and Metrics. https://rocm.docs.amd.com/en/docs-6.0.0/conceptual/gpu-arch/mi200-performance-counters.html. Accessed: 2025-04-14.

[3] Advanced Micro Devices, Inc. 2024. Omniperf: GPU Performance Analysis and Profiling Tool. https://github.com/AMDResearch/omniperf. Accessed: 2023-03-11.

[4] AMD. 2023. *ROCm Compute Profiler Documentation.* Technical Report. AMD.

[5] Jason Ansel, Shoaib Kamil Chan, Jonathan Ragan-Kelley Wong, Kalyan Veeramachaneni, Lukasz Ziarek, Saman Amarasinghe, and Martin C O'Reilly. 2014. Opentuner: An extensible framework for program autotuning. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. IEEE, 303–316. doi:10.1145/2628071.2628092

[6] Fabio Banchelli, Marta Garcia-Gasulla, Guillaume Houzeaux, and Filippo Mantovani. 2020. Benchmarking of state-of-the-art HPC Clusters with a Production CFD Code. In *Proceedings of the Platform for Advanced Scientific Computing Conference* (Geneva, Switzerland) *(PASC '20)*. Association for Computing Machinery, New York, NY, USA, Article 3, 11 pages. doi:10.1145/3394277.3401847

[7] Tania Banerjee, Jason Hackl, Mrugesh Shringarpure, Tanzima Islam, S Balachandar, Thomas Jackson, and Sanjay Ranka. 2016. CMT-Bone — A Proxy Application for Compressible Multiphase Turbulent Flows. *IEEE 23rd International Conference on High Performance Computing (HiPC)* (Dec 2016), 173–182. doi:10.1109/HiPC.2016.029 Acceptance rate: 23%.

[8] Michael Bauer, Henry Cook, and Brucek Khailany. 2011. CudaDMA: optimizing GPU memory bandwidth via warp specialization. In *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*. 1–11.

[9] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[10] Milind Chabbi, Karthik Murthy, Michael Fagan, and John Mellor-Crummey. 2013. Effective sampling-driven performance tools for GPU-accelerated supercomputers. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–12.

[11] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] https://arxiv.org/abs/2107.03374

[12] 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.

[13] Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Roziere, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. 2025. LLM Compiler: Foundation Language Models for Compiler Optimization. In *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction*. 141–153.

[14] Spandan Garg, Roshanak Zilouchian Moghaddam, Colin B Clement, Neel Sundaresan, and Chen Wu. 2022. DeepDev-PERF: a deep learning-based approach for improving software performance. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 948–958.

[15] Spandan Garg, Roshanak Zilouchian Moghaddam, and Neel Sundaresan. 2023. Rapgen: An approach for fixing code inefficiencies in zero-shot. *arXiv preprint arXiv:2306.17077* (2023).

[16] Jeff R. Hammond and Timothy G. Mattson. 2019. Evaluating data parallelism in C++ using the Parallel Research Kernels. In *Proceedings of the International Workshop on OpenCL* (Boston, MA, USA) *(IWOCL '19)*. Association for Computing Machinery, New York, NY, USA, Article 14, 6 pages. doi:10.1145/3318170.3318192

[17] Yueming Hao, Nikhil Jain, Rob Van der Wijngaart, Nirmal Saxena, Yuanbo Fan, and Xu Liu. 2023. DrGPU: A Top-Down Profiler for GPU Applications. In *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering*. 43–53.

[18] Alaa Hesham and Abeer Hamdy. 2024. Fine-Tuning GPT-4o-Mini for Programming Questions Generation. In *2024 International Conference on Computer and Applications (ICCA)*. IEEE, 1–6.

[19] Pieter Hijma, Stijn Heldens, Alessio Sclocco, Ben Van Werkhoven, and Henri E Bal. 2023. Optimization techniques for GPU programming. *Comput. Surveys* 55, 11 (2023), 1–81.

[20] Aleksandar Ilic, Frederico Pratas, and Leonel Sousa. 2014. Cache-aware Roofline Model: Upgrading the Loft. *IEEE Comput. Archit. Lett.* 13, 1 (Jan. 2014), 21–24.

[21] Tanzima Islam, Alexis Ayala, Quentin Jensen, and Khaled Ibrahim. 2019. Toward a Programmable Analysis and Visualization Framework for Interactive Performance Analytics. In *2019 IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools)*. IEEE, 70–77.

[22] Tanzima Islam, Alexis Ayala, Quentin Jensen, and Khaled Ibrahim. 2019. Toward a Programmable Analysis and Visualization Framework for Interactive Performance Analytics. In *2019 IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools)*. 70–77. doi:10.1109/ProTools49597.2019.00015

[23] Tanzima Z. Islam, A. Ayala, and Q. Jensen. [n. d.]. Dashing–A Machine Learning Toolkit for HPC Performance Analysis. https://github.com/tzislam/Dashing.

[24] Tanzima Z. Islam, Aniruddha Marathe, Holland Schutte, and Mohammad Zaeed. 2025. Data-Driven Analysis to Understand GPU Hardware Resource Usage of Optimizations. *International Journal of High Performance Computing Applications* (2025). Accepted.

[25] Tanzima Z Islam, Jayaraman J Thiagarajan, Abhinav Bhatele, Martin Schulz, and Todd Gamblin. 2016. A machine learning framework for performance coverage analysis of proxy applications. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 538–549.

[26] Quentin Jensen, Filip Jagodzinski, and Tanzima Z. Islam. 2021. FILCIO: Application Agnostic I/O Aggregation to Scale Scientific Workflows. In *2021 IEEE International Workshop on Big Data Computation, Analysis, and Application (BDCAA)*. IEEE. doi:10.1109/BDCAA54681.2021.00009

[27] Zheming Jin and Jeffrey S. Vetter. 2023. A Benchmark Suite for Improving Performance Portability of the SYCL Programming Model. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 325–327. doi:10.1109/ISPASS57527.2023.00041

[28] Khaled Z. Ibrahim, Samuel Williams, Leonid Oliker. 2018. Roofline Scaling Trajectories: A Method for Parallel Application and Architectural Performance Analysis. In *International Conference on High Performance Computing & Simulation (HPCS)*.

[29] Gary Lawson, Vaibhav Sundriyal, Masha Sosonkina, and Yuzhong Shen. 2015. Experimentation Procedure for Offloaded Mini-Apps Executed on Cluster Architectures with Xeon Phi Accelerators. arXiv:1509.02135 [cs.DC] https://arxiv.org/abs/1509.02135

[30] Jongmin Lee, Kwangho Lee, Mucheol Kim, Geunchul Park, and Chan Yeol Park. 2020. Roofline-based Data Migration Methodology for Hybrid Memories. *Journal of Internet Technology* 21, 3 (2020), 849–859.

[31] Matthew Leinhauser, René Widera, Sergei Bastrakov, Alexander Debus, Michael Bussmann, and Sunita Chandrasekaran. 2022. Metrics and design of an instruction roofline model for amd gpus. *ACM Transactions on Parallel Computing* 9, 1 (2022), 1–14.

[32] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems* 33 (2020), 9459–9474.

[33] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024. Enhancing static analysis for practical bug detection: An llm-integrated approach. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 474–499.

[34] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: May the source be with you! *arXiv preprint arXiv:2305.06161* (2023). https://arxiv.org/abs/2305.06161

[35] Yukuo Luo, Yanqi Zheng, Jiacheng Deng, Tianjun Zhang, Hang Su, Stefano Ermon, Yuke Ye, Tianshi Zhao, and Yuxin Wu. 2023. OptFormer: Schedule Optimization as Program Generation. *arXiv preprint arXiv:2305.10765* (2023). https://arxiv.org/abs/2305.10765

[36] Zeyuan Ma, Hongshu Guo, Jiacheng Chen, Guojun Peng, Zhiguang Cao, Yining Ma, and Yue-Jiao Gong. 2024. LLaMoCo: Instruction Tuning of Large Language Models for Optimization Code Generation. arXiv:2403.01131 [math.OC] https://arxiv.org/abs/2403.01131

[37] Aman Madaan, Alexander Shypula, Uri Alon, Milad Hashemi, Parthasarathy Ranganathan, Yiming Yang, Graham Neubig, and Amir Yazdanbakhsh. 2023. Learning performance-improving code edits. *arXiv preprint arXiv:2302.07867* 8 (2023).

[38] Mitesh R Meswani, Laura Carrington, Allan Snavely, and Stephen Poole. 2012. Tools for benchmarking, tracing, and simulating SHMEM applications. In *Proceedings Cray user group conference (CUG)*.

[39] Mahmood Naderan-Tahan and Lieven Eeckhout. 2021. Cactus: Top-Down GPU-Compute Benchmarking using Real-Life Applications. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*. 176–188. doi:10.1109/IISWC53511.2021.00026

[40] NVIDIA Corporation. 2023. *Kernel Profiling Guide for NVIDIA Nsight Compute*. Technical Report. NVIDIA.

[41] NVIDIA Corporation. 2024. CUDA C++ Best Practices Guide. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/.

[42] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code Llama: Open Foundation Models for Code. *arXiv preprint arXiv:2308.12950* (2023). https://arxiv.org/abs/2308.12950

[43] John Tramm, Andrew Siegel, Tanzima Islam, and Martin Schulz. 2014. XSBench-the Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)* (2014).

[44] Leif Uhsadel, Andy Georges, and Ingrid Verbauwhede. 2008. Exploiting hardware performance counters. In *2008 5th Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, 59–67.

[45] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[46] S Williams, A Waterman, and D Patterson. 2009. Roofline: Aninsightful visual performance model for floating-point programsand multicore architectures. *Communications of theAssociation for Computing Machinery* (2009).

[47] Xingfu Wu, Valerie Taylor, and Zhiling Lan. 2021. Performance and energy improvement of ECP proxy app SW4lite under various workloads. In *2021 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. IEEE, 17–24.

[48] Charlene Yang, Thorsten Kurth, and Samuel Williams. 2020. Hierarchical Roofline analysis for GPUs: Accelerating performance optimization for the NERSC-9 Perlmutter system. *Concurrency and Computation: Practice and Experience* 32, 20 (2020), e5547. doi:10.1002/cpe.5547 Received August 2019; Accepted August 28, 2019.

[49] Mohammad Zaeed, Tanzima Z Islam, and Vladimir Indict. 2024. Characterize and Compare the Performance of Deep Learning Optimizers in Recurrent Neural Network Architectures. In *2024 IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 39–44.

[50] Hang Zhou, Yehui Tang, Haochen Qin, Yujie Yang, Renren Jin, Deyi Xiong, Kai Han, and Yunhe Wang. 2024. Star-Agents: Automatic Data Optimization with LLM Agents for Instruction Tuning. *Advances in Neural Information Processing Systems* 37 (2024), 4575–4597.

[51] Keren Zhou, Laksono Adhianto, Jonathon Anderson, Aaron Cherian, Dejan Grubisic, Mark Krentel, Yumeng Liu, Xiaozhu Meng, and John Mellor-Crummey. 2021. Measurement and analysis of GPU-accelerated applications with HPCToolkit. *Parallel Comput.* 108 (2021), 102837.

[52] Keren Zhou, Mark W Krentel, and John Mellor-Crummey. 2020. Tools for top-down performance analysis of GPU-accelerated applications. In *Proceedings of the 34th ACM International Conference on Supercomputing*. 1–12.

[53] Keren Zhou, Xiaozhu Meng, Ryuichi Sai, and John Mellor-Crummey. 2021. Gpa: A gpu performance advisor based on instruction sampling. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 115–125.

[54] Bob Zigon and Fengguang Song. 2020. Utilizing gpu performance counters to characterize gpu kernels via machine learning. In *Computational Science–ICCS 2020: 20th International Conference, Amsterdam, The Netherlands, June 3–5, 2020, Proceedings, Part I 20*. Springer, 88–101.