Can Large Language Models Predict Parallel Code Performance?

Gregory Bolet
gbolet@vt.edu
Virginia Tech, USA

Giorgis Georgakoudis
georgakoudis1@llnl.gov
LLNL, USA

Harshitha Menon
harshitha@llnl.gov
LLNL, USA

Konstantinos Parasyris
parasyris1@llnl.gov
LLNL, USA

Niranjan Hasabnis
niranjan@codemetal.ai
Code Metal, USA

Hayden Estes
haydenve@vt.edu
Virginia Tech, USA

Kirk W. Cameron
cameron@vt.edu
Virginia Tech, USA

Gal Oren
galoren@stanford.edu
Stanford University,
Technion, USA

Abstract

Accurate determination of the performance of parallel GPU code typically requires execution-time profiling on target hardware – an increasingly prohibitive step due to limited access to high-end GPUs. This paper explores whether Large Language Models (LLMs) can offer an alternative approach for GPU performance prediction without relying on hardware. We frame the problem as a *roofline classification* task: given the source code of a GPU kernel and the hardware specifications of a target GPU, can an LLM predict whether the GPU kernel is compute-bound or bandwidth-bound?

For this study, we build a balanced dataset of 340 GPU kernels, obtained from HeCBench benchmark and written in CUDA and OpenMP, along with their ground-truth labels obtained via empirical GPU profiling. We evaluate LLMs across four scenarios: (1) with access to profiling data of the kernel source, (2) zero-shot with source code only, (3) few-shot with code and label pairs, and (4) fine-tuned on a small custom dataset. Our results show that state-of-the-art LLMs have a strong understanding of the Roofline model, achieving 100% classification accuracy when provided with explicit profiling data. We also find that reasoning-capable LLMs significantly outperform standard LLMs in zero- and few-shot settings, achieving up to 64% classification accuracy of GPU source codes, without any profiling information. Lastly, we find that model accuracy does not benefit meaningfully from few-shot prompting compared to zero-shot, and that LLM fine-tuning will require much more data than what we currently have available. This work is among the first to use LLMs for source-level roofline performance prediction via classification, and illustrates their potential to guide optimization efforts when runtime profiling is infeasible. Our findings suggest that with better datasets and prompt strategies, LLMs could become practical tools for HPC performance analysis and performance portability. Code and datasets are publicly available at https://github.com/Scientific-Computing-Lab/ParallelCodeEstimation.

CCS Concepts

• **Computing methodologies** → **Natural language generation**; **Machine translation**.

Keywords

Roofline model, LLMs, CUDA, OpenMP, GPU, Performance

ACM Reference Format:

Gregory Bolet, Giorgis Georgakoudis, Harshitha Menon, Konstantinos Parasyris, Niranjan Hasabnis, Hayden Estes, Kirk W. Cameron, and Gal Oren. 2025. Can Large Language Models Predict Parallel Code Performance?. In *Workshop on AI For Systems (AI4Sys '25), July 20, 2025, Notre Dame, IN, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3731545.3743645

HPDC '25, Notre Dame, IN, USA

ACM ISBN 979-8-4007-1869-4/2025/07
https://doi.org/10.1145/3731545.3743645

1 Introduction

High-Performance Computing (HPC) applications increasingly rely on GPUs for acceleration, yet ensuring *performance portability* across diverse GPU architectures has become a pressing challenge [3, 28, 34]. Modern supercomputers feature a mix of GPUs from different vendors, and codes optimized for one platform may not achieve optimal performance on another. Identifying performance bottlenecks, whether a kernel is limited by computation or by memory throughput, is crucial to guide optimizations. Traditionally, this determination requires running detailed profiles on target hardware [19–21], but access to cutting-edge GPUs is often limited and expensive [9, 27]. There is a growing need for *hardware-free performance prediction* methods that can provide insights without extensive benchmarking on physical devices. This work addresses that need by exploring a novel, static approach to classify GPU kernel performance.

A well-established framework for reasoning about code performance is the *Roofline model* [31]. The Roofline model correlates a kernel's *arithmetic intensity (**AI**)* (operations per byte of memory traffic) with hardware peak performance (operations per second) to determine a performance ceiling. Kernels with low AI tend to be *Bandwidth-Bound (**BB**)* (limited by memory bandwidth), while those with high AI are *Compute-Bound (**CB**)* (limited by the processor's peak FLOPs) [33]. By plotting a kernel's AI against achievable performance, one can visualize whether performance is capped by bandwidth or compute limits. This understanding of AI and performance relative to a target hardware roofline can then be used to intuitively guide kernel optimization decisions to reach peak system performance [34, 35]. However, obtaining a kernel's AI and achieved performance usually entails *profiling* – measuring runtime operations and memory transfers on the actual GPU. Similarly, previous GPU performance modeling works also depend on performance counter metrics [2, 23]. This requirement hampers rapid performance analysis, especially when developers lack access to the target hardware. Static analysis works [1, 14–16] have shown potential for GPU power and execution time prediction, however, these still require hardware access for micro-benchmarking.

Large Language Models (LLMs) offer a fresh opportunity to address this problem of requiring hardware access and runtime information to predict performance. Recent advances in code-focused LLMs have demonstrated remarkable capabilities in understanding and generating code. These models can perform tasks such as summarization, translation, bug fixing, and optimization [4, 7, 32]. In HPC contexts, early studies have begun applying LLMs to assist in

automatic parallelization [29], GPU kernel execution time prediction [24], performance slowdown prediction [11], and performance optimization [6, 12]. These developments suggest that LLMs not only understand code syntax but may also capture deeper semantic features and performance-affecting structures. The advantage of the Roofline model is that it can intuitively guide *developers* – and thus, LLMs, towards an optimal program. Our work is the beginning of an effort that seeks to leverage this nature of the Roofline model to guide LLMs towards delivering performant code. However, integration of *modern LLMs* into Roofline-based performance analysis remains unexplored. This work seeks to establish a base understanding of how well LLMs recall and use the Roofline model to predict program performance.

We build upon the trends of: Roofline modeling, static prediction, and LLM-driven code understanding – by framing GPU performance prediction as a binary classification task. We set our sights on a simple sub-problem within Roofline modeling: classifying a program as BB or CB based on its AI. We first assess how well LLMs can classify kernels when given known AI values; expecting the LLMs to do simple calculations and reasoning to arrive at an answer. We then profile CUDA-based and OpenMP-based GPU kernels from the HeCBench benchmark suite [17, 18] and obtain their ground-truth AI classification labels. Given a GPU kernel, we have LLMs predict whether it is BB or CB using only the source code and hardware/execution specs. By comparing LLM predictions to our ground-truth labels, we assess whether LLMs can intuit about AI, and if they can serve as practical tools for source-code-level performance classification. From this point forward, we refer to this task of classifying the AI of a program (to be BB or CB) as the *roofline classification task*.

Research Questions. To study how LLMs can be applied to the roofline classification task, we ask the following questions:

- **RQ1 (Baseline Roofline Classification):** Can LLMs classify codes *well* when given the hardware roofline, and arithmetic intensity values?
- **RQ2 (Zero-Shot Classification):** Can LLMs classify codes *well* when *given their source code, hardware specs, and minimal instructions*?
- **RQ3 (Few-Shot Classification):** Can LLMs classify codes *well* when given their source code, hardware specs, and a few examples of codes with their expected classifications?
- **RQ4 (Fine-Tuned Classification):** Can we fine-tune LLMs for roofline classification?

Contributions. The contributions of this paper are:

(1) To the best of our knowledge, this is the first systematic evaluation of LLMs for the GPU roofline classification task.
(2) We demonstrate that existing state-of-the-art (SoTA) LLMs are highly accurate in roofline classification when presented with the profiled measurements of a kernel.
(3) We find that small, reasoning-capable LLMs can significantly outperform their larger counterparts as well as traditional non-reasoning LLMs in roofline classification.
(4) Our experiments suggest that there exists room for improvement of current LLMs in roofline classification.

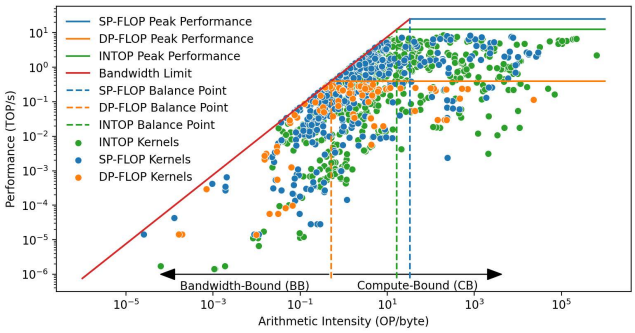

Figure 1: RTX 3080 Roofline Model and profiled HeCBench executable metrics. CB and BB classifications are shown for DP-FLOP balance point.

2 Our Approach

To gauge the efficacy of LLMs for the task of roofline classification, we empirically measure the AI of existing programs to form a ground-truth database. We query LLMs, asking for roofline classifications based on a program's source code and target hardware specs. We focus on GPU codes for two reasons: 1) because GPU workloads have become ubiquitous, thus this work can have a broad impact, and 2) because (in contrast to CPU-based codes) their source codes do not need to be manually instrumented to record runtime performance metrics for the roofline model. In the following, we describe the process of profiling, data collection, and pre-processing for evaluation.

2.1 Sampled Codes & Hardware

We built and profiled 446 CUDA and 303 OpenMP offload GPU codes from the HeCBench suite [17, 18]. The target hardware for profiling was an NVIDIA RTX 3080 with 10GB of RAM; selected due to physical access to the hardware. We profiled the GPU *kernels* within these HeCBench codes and were able to use their empirical measurements to form ground-truth AI classification labels. We capture single-precision (SP-FLOP) float, double-precision (DP-FLOP) float, and integer operations (INTOP) counts alongside execution time and number of global memory read/write operations for all the kernels within each program. To save time on profiling overhead of multiple/repeated kernel invocations, we only profile the first execution of each kernel within a program. From these metrics, we classify each of the kernels as BB or CB, relative to the three arithmetic operation rooflines: SP-FLOP, DP-FLOP, or INTOP corresponding to the major types of operations performed by the kernels. If a kernel is BB in all 3 arithmetic operations, we consider it BB for classification; otherwise if there exists at least 1 operation type where the kernel is CB, we consider it CB.

Figure 1 shows the roofline data of each of the sampled kernels for the target hardware. The rooflines for SP, DP, and INT ops are shown, along with their respective balance points and the profiled kernel samples as points on the plot. For all sampled kernels, the theoretical peak performance is usually unmet, as it is often not attainable [33, 34] without hardware-specific code modifications to take full advantage of the GPU architecture. Lastly, we note that the majority of the SP-FLOP and INT samples are BB on this hardware.

2.2 Dataset Creation & Pruning

Due to the imbalance of BB kernels that could bias our reported results, we take an extra pruning step to create a balanced inference dataset. This pruning step consists of dropping programs with long source code inputs, and dropping excess BB codes.

Source Scraping. For the source code that we feed to the LLMs for prediction, we opted for the easiest approach: to concatenate all the source files for each program into one string that is appended to the end of the LLM query prompt. Initial attempts to extract only the source code of each kernel proved complicated due to many edge cases related to templating, preprocessor defines, and determining critical execution path that were too challenging to handle.

Token Count Pruning. Prompting the LLMs with full source codes leads to token imbalance across prompts. Given that the LLM context can get lost when prompted with long inputs [22], LLMs can provide better responses about shorter programs. To homogenize queried source codes and drop long inputs, we set a cutoff of 8e3 tokens, determined after checking the distribution of tokens and source codes using the `gpt-4o-mini` tokenizer. This cutoff allowed us to keep a little more than 50% of all the profiled kernels, leaving us with 297 CUDA-based and 242 OMP-based sampled HeCBench programs.

Dataset Balancing. Because we feed the entire source code of a program when querying an LLM, we risk an unbalanced database if we choose to query with two different kernels from the same program. To avoid biasing our evaluation metrics, we only query LLMs about the first kernel from the object dump of each program; thus, each program appears only once in the dataset.

The final balancing step was to force the number of samples for combinations of code *language* (CUDA/OMP) and *class* (BB/CB) to be equal to the smallest set of said combinations. The smallest combination totaled 85 samples, for a final dataset of 340 samples. For later fine-tuning in RQ4, we further divide our dataset with an 80/20 training/validation split. This gave us 68 samples for each language/class *training* combo, and similarly 17 samples for *validation* combos.

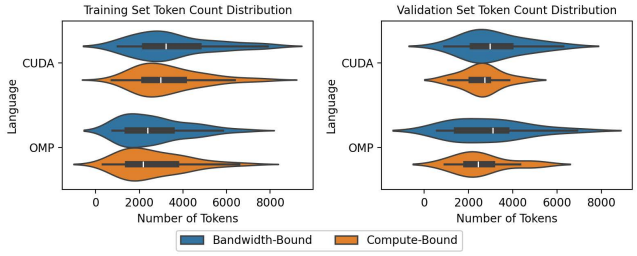

Figure 2: Training/Validation set token count distributions, showing a reasonably balanced dataset.

Figure 2 shows the `gpt-4o-mini` token distribution of the balanced dataset with box-and-whisker plots. The OMP codes, are on average, able to use less tokens than the CUDA codes, while the IQR ranges and medians mostly line up. Given the small size of the validation set, ensuring balance without dropping too many samples was difficult, so we end up with the CB samples having a slightly tighter range than the BB samples.

3 Evaluation

3.1 Evaluation Metrics

Because we are performing a binary classification task with LLMs on a balanced dataset, we use the metrics of ***accuracy*** and ***F1-score*** (macro variant), as they do not depend on defining a true-positive and true-negative class, which could be either of the CB or BB classes. For the same class definition reason, we also use the ***Matthews Correlation Coefficient*** (MCC) metric. For the MCC, the metric values range from +1 (for perfect prediction), -1 (for inverse prediction), and 0 (for an average random prediction). Because all these metric values range from 0 to 1 or -1 to 1, we multiply them by 100 for readability purposes.

3.2 Sampled LLMs

Table 1 lists the various LLMs we sampled for this work. The LLMs fall into one of two categories: reasoning and non-reasoning LLMs. We mainly sample OpenAI LLMs because of their popularity in other work, as well as their ease of access through Microsoft Azure's OpenAI LLM services [10]. As an additional comparison point, we also sample Google's Gemini Flash 2.0 [25]. Our objective is to see if LLMs can even be good roofline performance classifiers, thus we only sample a handful of LLMs that are representative of the current SoTA.

LLM Hyperparameters. LLMs come with various sampling hyperparameters that can affect their generated output. The main hyperparmeters are: **`temperature`** and **`top_p`**, which control the variety of the model responses, and limits the model's output token choices, respectively. We conducted a Chi-Squared test on the LLMs listed in Table 1 and found that a change in these two hyperparameters did not have any statistically significant impact on the predicted outcomes of the LLMs [26]. For all further tests, we set the `temperature` to 0.1 and `top_p` to 0.2, which essentially guarantees less diverse and consistent model responses. The reasoning models in Table 1 do not allow for changes in the sampling hyperparameters, therefore, we query them with their default settings.

3.3 Experimental Design

We conducted one experiment for each research question from section 1; details are provided below.

RQ1: Baseline Roofline Calculations. To understand whether LLMs were capable of roofline model calculations for *arbitrary* hardware, we first tried a simple experiment of querying them with random rooflines accompanied by random AI values for classification. Given a roofline, the LLMs are expected to derive a balance point that separates BB from CB codes on the AI axis, allowing them to easily categorize requested AI values. We randomly generated 240 different rooflines, where we would randomly select a BB and a CB AI for classification.

Figure 3 shows the provided prompt, where the highlighted regions show values that were changed between prompts according to the random roofline used. The prompt was designed to show 2, 4, or 8 examples, with (and without) chain-of-thought (CoT) [30] thought text, demonstrating how the model should structure its output and perform reasoning to arrive at a classification. To avoid erratic responses and allow for easy automation of response

Model Name	Reasoning	Input/Output Cost (1M tokens)	RQ1 Acc.	RQ1 CoT Acc.	RQ2 Acc.	RQ2 F1	RQ2 MCC	RQ3 Acc.	RQ3 F1	RQ3 MCC
o3-mini-high	✓	$1.1 / $4.4	**100**	**100**	**64.12**	**62.33**	31.36	**63.53**	**60.91**	**31.63**
o1	✓	$15 / $60	–	–	64.12	61.67	**32.73**	61.47	58.77	26.70
o3-mini	✓	$1.1 / $4.4	**100**	**100**	62.06	60.80	25.84	62.94	60.88	29.13
gpt-4.5-preview		$75 / $150	–	–	59.71	59.45	19.66	60.88	60.25	22.50
o1-mini-2024-09-12	✓	$1.1 / $4.4	**100**	**100**	59.64	58.91	19.92	56.47	55.98	13.24
gemini-2.0-flash-001		$0.1 / $0.4	91.25	92.50	55.59	55.45	11.25	53.82	48.96	9.72
gpt-4o-2024-11-20		$2.5 / $10	91.25	96.25	52.06	41.04	8.20	53.24	44.17	10.93
gpt-4o-mini		$0.15 / $0.6	90.00	**100**	50.59	50.03	1.20	52.35	50.92	5.01
gpt-4o-mini-2024-07-18		$0.15 / $0.6	90.00	**100**	50.29	49.88	0.60	52.06	50.46	4.41

Table 1: Evaluation results sorted by RQ1 Accuracy. Cost (as of April 2025), reasoning capabilities, and metrics across RQ1–RQ3 are shown. All values are percentages unless otherwise noted.

correctness checking, we purposely include at least *two* question and response examples in the prompts.

Chain-of-Thought (CoT) Prompt Example

CoT example 1 (shown below):

Question: Given a GPU having a global memory with a max bandwidth of 45.9 GB/s and a peak performance of 52.22 GFLOP/s, if a program executed with an Arithmetic Intensity of 0.6 FLOP/Byte and a performance of 19.4 GFLOP/s, does the roofline model consider the program as compute-bound or bandwidth-bound?

Thought: The max bandwidth is 45.9 GB/s, and peak performance is 52.22 GFLOP/s. The balance point is at 52.22 / 45.9= 1.14 FLOP/Byte. The program's Arithmetic Intensity is 0.6 FLOP/Byte. Because 0.6 < 1.14, it is before the balance point, putting the program in the bandwidth-bound region. The roofline model would consider the program as bandwidth-bound.

Answer: Bandwidth

CoT examples 2-8 [redacted]

Question: Given a GPU having a global memory with a max bandwidth of 99.9 GB/s and a peak performance of 73.45 GFLOP/s, if a program executed with an Arithmetic Intensity of 1.55 FLOP/Byte and a performance of 32.8 GFLOP/s, does the roofline model consider the program as compute-bound or bandwidth-bound?

Figure 3: Prompt for RQ1. Highlighted text indicates values changed between invocations.

RQ2: Zero-Shot. For RQ2, we increase the roofline classification task difficulty from RQ1 by querying the LLMs without profiling information, instead we provide hardware information and source code to be classified, with minimal instructions and pseudo-code classification examples.

Figure 4 shows the prompt we used to query the LLMs. We found the pseudo-code examples in the prompt provided enough context for smaller LLMs to produce only the desired output singleton tokens of *Compute* or *Bandwidth*, corresponding to CB and BB classifications, respectively. The highlighted portions of: *language* (CUDA or OMP), GPU hardware information, and source code change between invocations. We query all 340 profiled programs from the dataset defined in subsection 2.2.

RQ3: Few-Shot. Similar to RQ2, this experiment uses the same prompt but replaces the pseudo-code examples with actual code examples which can be found on our repository [5]. These real examples are not part of the dataset from subsection 2.2. In testing, we noticed that supplying more than the two classification examples led to excessively bloated input prompts that easily lost contextual information, thus we only query with two examples, effectively making our few-shot experiments *two-shot*. For RQ3, we again query all 340 profiled programs from subsection 2.2, where we make sure to only supply examples in the language (CUDA/OMP) of the queried source code.

RQ4: Fine-tuned. We wanted to see how well a model performs when fine-tuned with the roofline dataset we created. We used the Microsoft Azure OpenAI services [10] to train `gpt-4o-mini-2024-07-18` with the 80/20 train/test split dataset from subsection 2.2. It is typical to train LLMs on datasets with thousands of samples, unfortunately it is difficult to gather more samples while maintaining a balanced dataset, so we experimented with what we had. We leave it as future work to augment the dataset for more holistic training.

3.4 RQ1: Roofline Calculation Results

Columns 4 and 5 of Table 1 show the best accuracy metrics from the 2, 4, and 8-shot prompting of the LLMs as in Figure 3. All the LLMs achieve a high accuracy between 90-100%, where the COT prompting notably helps the non-reasoning LLMs. Across all the few-shot CoT prompts, the reasoning LLMs score 100% accuracy. We omit the metrics of F1 and MCC from Table 1 because they exhibit the same trends as the accuracy. The results for `o1` and `gpt-4.5-preview` are excluded because their smaller counterparts already perform so well.

Answer to RQ1

Existing SoTA LLMs are capable of making sense of roofline problems and categorizing a program's arithmetic intensity when given all the GPU roofline specifications and program profiling results.

3.5 RQ2: Zero-Shot Results

Columns 6, 7, and 8 of Table 1 show the results for the zero-shot experiments. The best models in terms of *accuracy* are the `o3-mini-high` and `o1`, both correctly predicting the class of 64% of the source code samples. Due to only having accuracy differences of on average 5% when solely predicting CUDA or OMP codes, we omit the metrics for predictions across language, instead we present joint metrics. Note that there is a clear performance difference between the reasoning and non-reasoning models, where the non-reasoning LLMs make predictions akin to a random predictor, given the accuracy

System Prompt for Classifying GPU Kernels (RQ1–RQ3)

You are a **GPU performance analysis expert** that classifies kernels into Arithmetic Intensity Roofline model categories based on their source code characteristics. Your task is to provide one of the following performance boundedness classifications: **Compute** or **Bandwidth**.
A kernel is considered **Compute bound** if its performance is primarily limited by the number of operations it performs, and **Bandwidth bound** if its performance is primarily limited by the rate at which data can be moved between memory and processing units.

Provide **only one word** as your response, chosen from the set: `['Compute', 'Bandwidth']`.

Examples:

Example 1: [changed to real CUDA/OMP code for RQ2]

```
Kernel Source Code (simplified):
for i = 0 to 1000000 {
 a[i] = a[i] + b[i];
}
```

Response: **Compute**

Example 2: [changed to real CUDA/OMP code for RQ2]

```
Kernel Source Code (simplified):
for i = 0 to 10 {
 load_data(large_array);
 process_data(large_array);
 store_data(large_array);
}
```

Response: **Bandwidth**

Now, analyze the following source codes for the requested kernel of the specified hardware.
Classify the [language] kernel called [kernel name] as **Bandwidth** or **Compute** bound. The system it will execute on is a [GPU model] with:

- peak single-precision performance of [X] GFLOP/s
- peak double-precision performance of [X] GFLOP/s
- peak integer performance of [X] GINTOP/s
- max bandwidth of [X] GB/s

The block and grid sizes of the invoked kernel are (X,Y,Z) and (X,Y,Z), respectively. The executable running this kernel is launched with the following command-line arguments: `[arg1 arg2 arg3]`.
Below is the source code of the requested [language] kernel:

[concatenated source code files]

Figure 4: System prompt to query LLMs. Highlighted text changes based on the queried source code and hardware.

values near 50% and MCC values close to 0. The reasoning LLMs are able to achieve about a 10% greater accuracy over the non-reasoning LLMs, indicating that the extra reasoning steps of the models are contributing to better predictions.

Answer to RQ2

LLMs *can* predict a CUDA/OMP program's arithmetic intensity classification, especially when using reasoning-based models. However, there is still room for improvement, with the best model achieving a maximum accuracy of 64%.

3.6 RQ3: Few-Shot Results

Similar to RQ2, columns 9, 10, and 11 of Table 1 show the results of the few-shot experiments. We can notice that again `o3-mini-high` and `o1` end up on top across all three metrics. For the reasoning models, there is not much of a difference in metrics between the few-shot and zero-shot experiments. However, the small (*mini*) non-reasoning LLMs perform marginally better (2% accuracy improvement) given the real examples in the prompt.

Answer to RQ3

Few-shot examples yield marginal improvements on the metrics of non-reasoning models, while hardly impacting reasoning-based LLMs. Our recommendation would be to save money on input token costs by prompting in zero-shot style with reasoning models.

3.7 RQ4: Fine-Tuning Results

For model fine-tuning, the outcome was poor. We fine-tuned `gpt-4o-mini` for two epochs, where at the end of both epochs, the model had devolved and would always predict either CB or BB for the whole validation set. The same behavior would occur when fine-tuned solely on CUDA or OMP programs. The training prompts used were the zero-shot prompts from RQ2, as we wanted to see if we could achieve an accuracy above 50%. We strongly suspect the fine-tuned model did not have enough training samples to guarantee generalization, which caused it to produce incorrect predictions.

Answer to RQ4

Our current results indicate fine-tuning completely biases LLMs to always answer either CB or BB. We speculate that a larger training dataset is necessary to guarantee a more generalizable fine-tuned LLM.

4 Conclusion & Future Work

In this paper, we explore the applicability of LLMs for classifying the arithmetic intensity (AI) of CUDA-based and OpenMP-based GPU source codes. We discover that LLMs indeed can predict whether a program is compute-bound (CB) or bandwidth-bound (BB) without the need of execution/profiling. A distinction between reasoning-based models and non-reasoning models arises, where reasoning LLMs significantly outperform non-reasoning models. We show that with some simple prompting techniques, we are able to get LLMs to predict with at best a 64% accuracy.

Expanding Dataset. Given different GPU hardware, the arithmetic intensity of a program may change from CB to BB. Our experimentation was performed with a single GPU. To have more generalizable results, it would be best to re-profile all our GPU programs on varying hardware to see how LLM prediction accuracy changes. This would expand our dataset size to also allow for more robust fine-tuning using thousands of samples, rather than hundreds. We would be able to make more conclusive statements about whether fine-tuning an LLM to this problem is worth the effort.

Improve Prediction Accuracy. Our evaluation of LLMs using prompting techniques demonstrated promise. However, more recent question-decomposition [8], successive-prompting [13], and least-to-most [36] prompting techniques have shown effectiveness in breaking down and solving complex tasks. In an effort to improve roofline classification metrics, these techniques warrant further investigation.

Acknowledgments

This research was supported by Code Metal Inc. and the Pazy foundation. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

References

[1] Gargi Alavani, Jineet Desai, and Santonu Sarkar. 2020. An Approach to Estimate Power Consumption of a CUDA Kernel. In *2020 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*. IEEE, Exeter, United Kingdom, 984–991. doi:10.1109/ISPA-BDCloud-SocialCom-SustainCom51426.2020.00149

[2] Arunavo Dey, Aakash Dhakal, Tanzima Z. Islam, Jae-Seung Yeom, Tapasya Patki, Daniel Nichols, Alex Movsesyan, and A. Bhatele. 2024. Relative Performance Prediction Using Few-Shot Learning. *Annual International Computer Software and Applications Conference* (2024). doi:10.1109/compsac61105.2024.00278

[3] David Beckingsale, Richard Hornung, Tom Scogland, and Arturo Vargas. 2019. Performance portable C++ programming with RAJA. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 455–456. doi:10.1145/3293883.3302577

[4] Bin Lei, Caiwen Ding, Le Chen, Pei-Hung Lin, and Chunhua Liao. 2023. Creating a Dataset for High-Performance Computing Code Translation Using LLMs: A Bridge Between OpenMP Fortran and C++. *IEEE Conference on High Performance Extreme Computing* (2023). doi:10.1109/hpec58863.2023.10363534

[5] Gregory Bolet. 2025. LLMs4Roofline Performance Prediction. https://github.com/Scientific-Computing-Lab/ParallelCodeEstimation.

[6] Bowen Cui, Tejas Ramesh, Oscar Hernandez, and Keren Zhou. 2025. Do Large Language Models Understand Performance Optimization? (2025). arXiv:2503.13772 [cs.DC] https://arxiv.org/abs/2503.13772

[7] Mark Chen and et al. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]

[8] Ru Chen, Jingwei Shen, and Xiao He. 2024. A Model Is Not Built By A Single Prompt: LLM-Based Domain Modeling With Question Decomposition. arXiv:2410.09854 [cs.SE] https://arxiv.org/abs/2410.09854

[9] Peter Cohan. 2024. *Generative AI Hardware*. Apress, Berkeley, CA, 237–279. doi:10.1007/979-8-8688-0318-5_7

[10] Microsoft Corporation. 2025. Azure OpenAI Service documentation - Quickstarts, Tutorials, API Reference - Azure AI services — learn.microsoft.com. https://learn.microsoft.com/en-us/azure/ai-services/openai/. [Accessed 11-04-2025].

[11] Daniel Nichols, Aniruddha Marathe, Harshitha Menon, T. Gamblin, and A. Bhatele. 2023. Modeling Parallel Programs Using Large Language Models. *Information Security Conference* (2023). doi:10.23919/isc.2024.10528929

[12] Daniel Nichols, Pranav Polasam, Harshitha Menon, Aniruddha Marathe, T. Gamblin, and A. Bhatele. 2024. Performance-Aligned LLMs for Generating Fast Code. *arXiv.org* (2024). doi:10.48550/arxiv.2404.18864

[13] Dheeru Dua, Shivanshu Gupta, Sameer Singh, and Matt Gardner. 2022. Successive Prompting for Decomposing Complex Questions. arXiv:2212.04092 [cs.CL] https://arxiv.org/abs/2212.04092

[14] Kaijie Fan, Biagio Cosenza, and Ben Juurlink. 2019. Predictable GPUs Frequency Scaling for Energy and Performance. In *Proceedings of the 48th International Conference on Parallel Processing*. ACM, Kyoto Japan, 1–10. doi:10.1145/3337821.3337833

[15] Gargi Alavani, Kajal Varma, and Santonu Sarkar. 2018. Predicting Execution Time of CUDA Kernel Using Static Analysis. *2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)* (Dec. 2018), 948–955. doi:10.1109/bdcloud.2018.00139

[16] Joao Guerreiro, Aleksandar Ilic, Nuno Roma, and Pedro Tomas. 2019. GPU Static Modeling Using PTX and Deep Structured Learning. *IEEE Access* 7 (2019), 159150–159161. doi:10.1109/ACCESS.2019.2951218

[17] Zheming Jin. 2024. HeCBench. https://github.com/zjin-lcf/HeCBench.

[18] 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

[19] José Morgado, Leonel Sousa, and A. Ilic. 2024. CARM Tool: Cache-Aware Roofline Model Automatic Benchmarking and Application Analysis. *IEEE International Symposium on Workload Characterization* (2024). doi:10.1109/iiswc63097.2024.00016

[20] Laksono Adhianto, Subarno Banerjee, Michael Fagan, Mark W. Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R. Tallent. 2009. HPCTOOLKIT: Tools for Performance Analysis of Optimized Parallel Programs. *Concurrency and Computation: Practice and Experience* 22, 6 (Jan. 2009), 685–701. doi:10.1002/cpe.1553

[21] Le Chen, Nesreen K. Ahmed, Akashnil Dutta, Arijit Bhattacharjee, Sixing Yu, Quazi Ishtiaque Mahmud, Waqwoya Abebe, Hung Phan, Aishwarya Sarkar, Branden Butler, N. Hasabnis, Gal Oren, Vy A. Vo, J. P. Muñoz, Ted Willke, Tim Mattson, and Ali Jannesari. 2024. The Landscape and Challenges of HPC Research and LLMs. *arXiv.org* (2024). doi:10.48550/arxiv.2402.02018

[22] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023. Lost in the Middle: How Language Models Use Long Contexts. arXiv:2307.03172 [cs.CL] https://arxiv.org/abs/2307.03172

[23] Lorenz Braun, Sotirios Nikas, Chen Song, Vincent Heuveline, and Holger Fröning. 2020. A Simple Model for Portable and Fast Prediction of Execution Time and Power Consumption of GPU Kernels. *ACM Transactions on Architecture and Code Optimization* 18, 1 (2020), 1–25. doi:10.1145/3431731

[24] Minh-Khoi Nguyen-Nhat, Hoang Duy Nguyen Do, Huyen Thao Le, and Thanh Tuan Dao. 2024. LLMPerf: GPU Performance Modeling Meets Large Language Models. *IEEE/ACM International Symposium on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems* (2024). doi:10.1109/mascots64422.2024.10786558

[25] Sundar Pichai, Demis Hassabis, and Koray Kavukcuoglu. 2024. Introducing Gemini 2.0: our new AI model for the agentic era — blog.google. https://blog.google/technology/google-deepmind/google-gemini-ai-update-december-2024/. [Accessed 11-04-2025].

[26] Matthew Renze. 2024. The Effect of Sampling Temperature on Problem Solving in Large Language Models. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). Association for Computational Linguistics, Miami, Florida, USA, 7346–7356. doi:10.18653/v1/2024.findings-emnlp.432

[27] Seonho Lee, Amar Phanishayee, and Divya Mahajan. 2024. Forecasting GPU Performance for Deep Learning Training and Inference. *International Conference on Architectural Support for Programming Languages and Operating Systems* (2024). doi:10.1145/3669940.3707265

[28] Christian R. Trott, Damien Lebrun-Grandié, Daniel Arndt, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S. Hollman, Dan Ibanez, Nevin Liber, Jonathan Madsen, Jeff Miles, David Poliakoff, Amy Powell, Sivasankaran Rajamanickam, Mikael Simberg, Dan Sunderland, Bruno Turcksin, and Jeremiah Wilke. 2022. Kokkos 3: Programming Model Extensions for the Exascale Era. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2022), 805–817. doi:10.1109/TPDS.2021.3097283

[29] Yu Wang and et al. 2023. LLM-OMP: Automated OpenMP Parallelization Using Large Language Models. *arXiv preprint arXiv:2308.02744* (2023).

[30] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., 24824–24837. https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf

[31] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (April 2009), 65–76. doi:10.1145/1498765.1498785

[32] Frank Xu and et al. 2022. Systematic Evaluation of Large Language Models for Code. *arXiv preprint arXiv:2202.13169* (2022).

[33] Charlene Yang, Rahulkumar Gayatri, Thorsten Kurth, Protonu Basu, Zahra Ronaghi, Adedoyin Adetokunbo, Brian Friesen, Brandon Cook, Douglas Doerfler, Leonid Oliker, Jack Deslippe, and Samuel Williams. 2018. An Empirical Roofline Methodology for Quantitatively Assessing Performance Portability. In *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 14–23. doi:10.1109/P3HPC.2018.00005

[34] 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 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.5547 e5547 cpe.5547.

[35] Yu Jung Lo, Samuel Williams, Brian Van Straalen, Terry J. Ligocki, Matthew J. Cordery, Nicholas J. Wright, Mary Hall, and Leonid Oliker. 2014. Roofline Model Toolkit: A Practical Tool for Architectural and Program Analysis. 8966 (Nov. 2014), 129–148. doi:10.1007/978-3-319-17248-4_7

[36] Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, and Ed Chi. 2023. Least-to-Most Prompting Enables Complex Reasoning in Large Language Models. arXiv:2205.10625 [cs.AI] https://arxiv.org/abs/2205.10625

Received 20 February 2007