QiMeng-GEMM: Automatically Generating High-Performance Matrix Multiplication Code by Exploiting Large Language Models

Qirui Zhou[1,3], Yuanbo Wen[1], Ruizhi Chen[2], Ke Gao[2], Weiqiang Xiong[2, 3], Ling Li[2, 3*], Qi Guo[1], Yanjun Wu[2], Yunji Chen[1, 3, 4]

[1] SKL of Processors, Institute of Computing Technology, CAS, Beijing, China
[2] Intelligent Software Research Center, Institute of Software, CAS, Beijing, China
[3] University of Chinese Academy of Sciences, Beijing, China
[4] Institute of AI for Industries, CAS, China
zhouqirui22s@ict.ac.cn

Abstract

As a crucial operator in numerous scientific and engineering computing applications, the automatic optimization of General Matrix Multiplication (GEMM) with full utilization of ever-evolving hardware architectures (e.g. GPUs and RISC-V) is of paramount importance. While Large Language Models (LLMs) can generate functionally correct code for simple tasks, they have yet to produce high-performance code. The key challenge resides in deeply understanding diverse hardware architectures and crafting prompts that effectively unleash the potential of LLMs to generate high-performance code.

In this paper, we propose a novel prompt mechanism called QiMeng-GEMM , which enables LLMs to comprehend the architectural characteristics of different hardware platforms and automatically search for the optimization combinations for GEMM. The key of QiMeng-GEMM is a set of informative, adaptive, and iterative meta-prompts. Based on this, a searching strategy for optimal combinations of meta-prompts is used to iteratively generate high-performance code. Extensive experiments conducted on 4 leading LLMs, various paradigmatic hardware platforms, and representative matrix dimensions unequivocally demonstrate QiMeng-GEMM's superior performance in auto-generating optimized GEMM code. Compared to vanilla prompts, our method achieves a performance enhancement of up to 113×. Even when compared to human experts, our method can reach 115% of cuBLAS on NVIDIA GPUs and 211% of OpenBLAS on RISC-V CPUs. Notably, while human experts often take months to optimize GEMM, our approach reduces the development cost by over 240×.

1 Introduction

As one of the most important computations in computer science (Tan et al. 2011), GEMM occupies over 90% of the total computations in deep learning applications, including LLMs. High-performance implementation of GEMM can achieve over 1300× acceleration (Hennessy and Patterson 2019), making it essentially important for the execution efficiency of most scientific and engineering applications (Jia 2014; Zhu et al. 2024b). Thus, designing high-performance GEMM code has received considerable attention from both academia (Sui et al. 2024; Jang et al. 2024; Wei et al. 2024) and industry such as hardware vendors including Intel (Intel 2023), NVIDIA (NVIDIA 2023b), and AMD (Sitaraman 2024), etc.

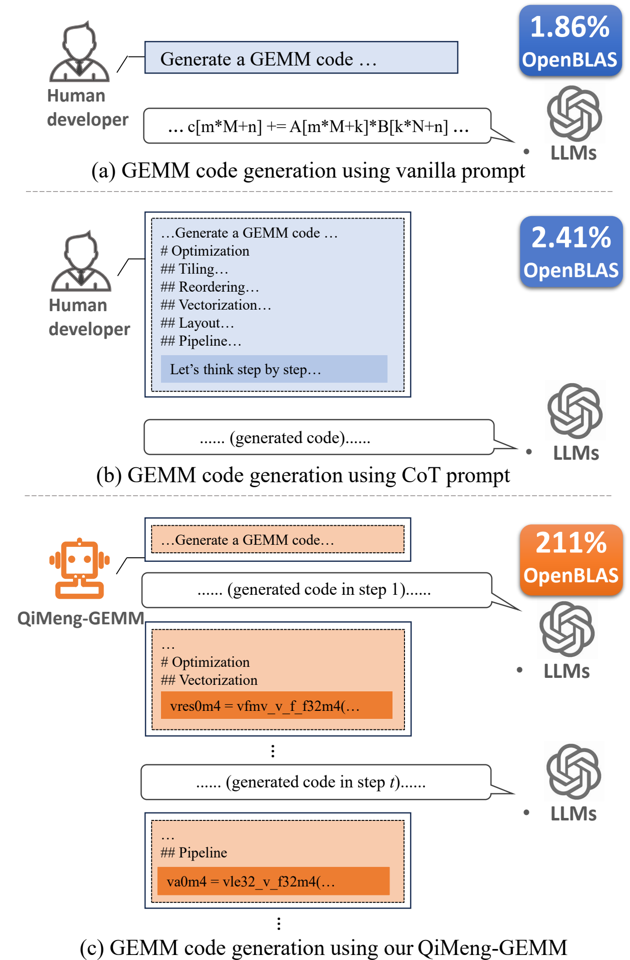

Figure 1: LLMs guided by our QiMeng-GEMM can generate high-performance GEMM code.

*Corresponding author.
Copyright © 2025, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

There are two main paradigms for designing high-performance GEMM, i.e., manual implementation and automatic compilation. The manual implementation approach requires extensive hand-tuned manipulation of hardware components (Rasch et al. 2021), which is time-consuming, error-prone, and not portable across different platforms. Hardware vendors like Intel and NVIDIA provide manually optimized GEMM libraries (e.g., MKL (Krainiuk et al. 2021) and cuBLAS (NVIDIA 2023a)) to fully exploit the computing capability of the underlying hardware. The automatic compilation approach formulates code generation as the exploration in a large program space (Purini et al. 2013; Shin, Nam et al. 2021; Bi et al. 2023; Chen et al. 2024; Shi et al. 2023, 2022), which can significantly reduce human efforts while the attained performance often lags behind manual implementations. As the hardware heterogeneity and complexity (e.g., X86 CPU, RISC-V CPU (Greengard 2020), NIVIDA GPU, Google TPU (Jouppi et al. 2017; Norrie et al. 2021; Jouppi et al. 2023)) continues to grow, automatically generating GEMM code, which delivers comparable or even better performance than manual implementations, is crucial to the success of hardware platforms.

LLMs such as GPT-4 (Achiam et al. 2023) and CodeLlama (Roziere et al. 2023) have demonstrated remarkable capabilities in code generation (Bairi et al. 2024; Zhong et al. 2024; Li et al. 2024; Holt et al. 2024; Wang et al. 2023; Li et al. 2022; Chen et al. 2021; Nijkamp et al. 2022), and thus offer a promising new approach for automatically generating high-performance GEMM code. However, existing LLMs fail to address this problem, because they cannot comprehend the complicated characteristics of hardware platforms (Olausson et al. 2023), let alone manipulate the hardware components in a sophisticated manner. We observe that appropriate prompt design can help LLMs better understand hardware characteristics, thereby significantly enhancing the functional accuracy and performance.

The design of LLM prompts for high-performance GEMM code generation poses three major challenges. The first challenge is to design robust prompts that generate functionally correct code during performance optimization. The second challenge is to provide a searching space for the optimal prompt permutations. The third challenge is to design versatile prompts that are compatible with different hardware platforms.

To mitigate the above challenges, we propose QiMeng-GEMM, a novel prompt mechanism for LLMs to automatically generate high-performance GEMM code across hardware platforms, as illustrated in Figure 1. It defines a set of meta-prompts by summarizing the core optimization heuristics of GEMM conducted by human experts. Then, an automatic searching strategy determines the optimal meta-prompt combinations for code generation. QiMeng-GEMM well addresses the aforementioned challenges: (1) the entire code generation process is formulated as a series of well-defined meta-prompts to decrease the reasoning complexity and enhance the functional accuracy, (2) the automatic tuning enables exhaustive search of all possible combinations of meta-prompts, potentially surpassing manual optimization for better performance, and (3) the meta-prompts contain both *platform-agnostic description* and *platform-specific hints* for achieving portability across diverse platforms.

To our best knowledge, we are *the first to automatically generate high-performance GEMM code by exploiting LLMs*. This paper makes the following contributions:

- We propose a set of meta-prompts that incorporate five common GEMM optimization techniques, enabling LLMs to understand the architectural characteristics of various hardware platforms.
- We introduce auto-tuning, which automatically searches for the optimal permutations of meta-prompts to generate high-performance GEMM code tailored to specific hardware platforms.
- We conduct thorough evaluations across various platforms and GEMM shape configurations. The results show that **the code generated by our approach outperforms vanilla prompt and hand-optimized libraries crafted by human experts**. Compared with the vanilla prompt, our method achieves up to $113.67\times$ and $9.65\times$ performance improvement on a CPU and a GPU respectively. Compared with hand-optimized libraries, our method achieves performance increase of up to 211% of OpenBLAS on a CPU and 115% of cuBLAS on a GPU.
- Last but not least, the proposed QiMeng-GEMM framework significantly reduces the development cost by over $240\times$, compared with human experts.

2 Preliminary

2.1 Optimization Primitives

GEMM optimization techniques can be essentially categorized into several fundamental and executable operations (named as "primitive"), which serve as the building block of GEMM optimization. The illustration of those primitives is shown in Figure 2.

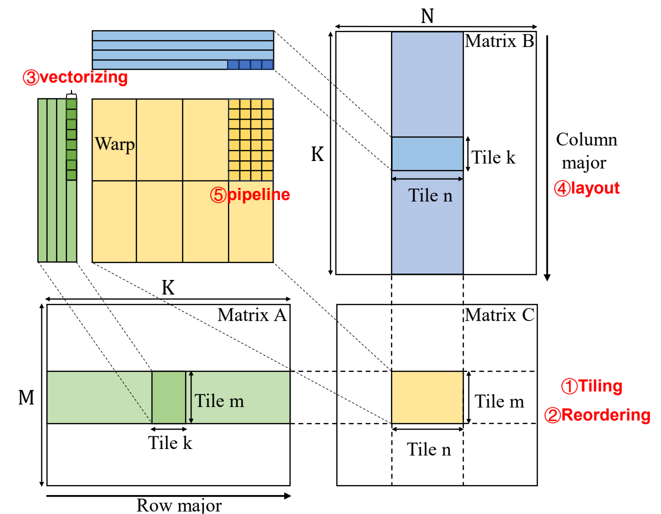

Figure 2: The illustrative example of the five GEMM optimization primitives.

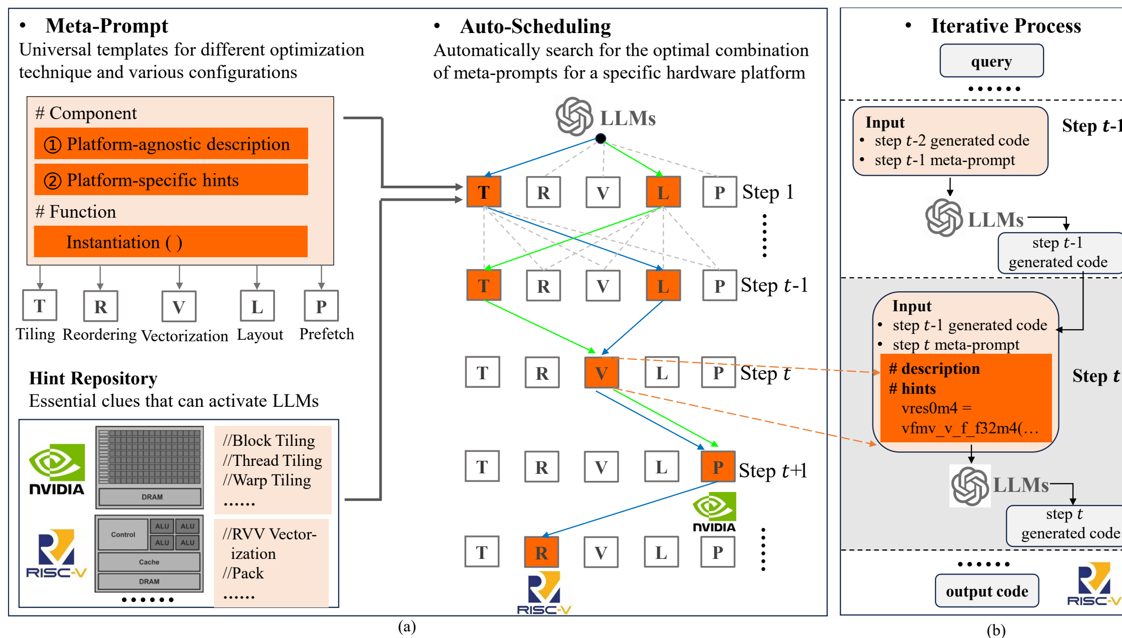

Figure 3: **QiMeng-GEMM overview.** (a) The proposed QiMeng-GEMM framework is comprised of two main components: meta-prompts and auto-tuning. We first introduce a set of well-defined meta-prompts that incorporate five common GEMM optimization techniques (i.e., *tiling, reordering, vectorization, layout, and pipeline*) along with platform-specific hints retrieved from pre-defined hint repositories. These meta-prompts enable LLMs to fully understand the optimization goals by capturing the architectural characteristics of various platforms. The auto-tuning mechanism allows for an exhaustive search of all possible optimization permutations of meta-prompts, facilitating the generation of high-performance GEMM code that can even surpass manual optimization. (b) illustrates the iterative process of using LLMs to generate GEMM code on a RISC-V CPU.

- **Tiling** involves partitioning the matrix into blocks that can fit into the cache, thus enhancing memory access efficiency (Faingnaert et al. 2022).
- **Reordering** optimizes memory access by swapping the order of the for-loops in the code (Anam et al. 2013).
- **Vectorization** packs several numbers from the matrix together for efficient memory accessing and computing (Katel et al. 2021).
- **Layout** changes the arrangement of the matrix to match specific hardware memory access characteristics (Kurzak, Tomov, and Dongarra 2012).
- **Pipeline** improves performance by overlapping different stages of computation and data movement, allowing for continuous processing and reducing the idle time between operations (Tan et al. 2011).

2.2 Hardware Characteristics

The diverse hardware characteristics pose substantial challenges for high-performance GEMM code generation, which mainly stem from their different *memory hierarchy* and *computing model*.

- **Memory Hierarchy** is a critical factor in determining the appropriate tiling strategy. For instance, on GPUs, dataflows move from global memory to shared memory and then to local memory, which requires the use of multi-level tiling. Conversely, on CPUs, only cache locality needs to be considered, making a simple block tiling sufficient. Additionally, the memory access pattern in GEMM code must align with the hardware characteristics, such as swizzle (Phothilimthana et al. 2019) on GPUs and array packing on CPUs (Jian Weng et al. 2023).
- **Computing Model** directly influences how GEMM is computed and its overall efficiency. For example, GPUs and CPUs utilize SIMT (Single Instruction, Multiple Threads) and SIMD (Single Instruction, Multiple Data) models, respectively. As a result, GEMM optimization on GPUs primarily focuses on optimizing task mapping to be executed across more threads. In contrast, on CPUs, the emphasis is on leveraging specialized vector instructions to enhance performance.

3 Method

In this section, we first present the overview of our QiMeng-GEMM framework, which effectively unleashes the potential of LLMs and thus automatically generates high-performance GEMM code across diverse platforms, as shown in Figure 3. The two key components will be detailed in subsequent subsections.

3.1 Meta-Prompt

The meta-prompt offers universal templates for various general optimization techniques and platform-specific optimization details. It is composed of a platform-agnostic description, platform-specific hints, and an instantiation method.

```
# Meta-Prompt for Tiling on RISC-V CPU
# Component 1: platform-agnostic description
Tiling the matrix to small pieces ....

# Component 2: platform-specific hint
## Specific tiling method on RISC-V CPU
Block Tiling: Separate "for" loops to small ...

## Specific tiling code on RISC-V CPU
......
for (int jc = 0; jc < n; jc += NC) {
  for (int pc = 0; pc < k; pc += KC) {
    for (int ic = 0; ic < m; ic += MC){
    // loop within blocks
......
```

RISC-V

```
# Meta-Prompt for Tiling on NIVIDIA GPU
# Component 1: platform-agnostic description
Tiling the matrix to small pieces ....

# Component 2: platform-specific hint
## Specific tiling method on NVIDIA GPU
Warp Tiling: Warp is ... Add warp-level tiling ...

## Specific tiling code on NVIDIA GPU
......
int wid = tid / 32;
...
for(int wm=0; ...)
  for(int wn=0; ...)
......
```

NVIDIA

Figure 4: An illustrative example of instantiation for the tiling meta-prompts on RISC-V CPU and NVIDIA GPU.

Platform-Agnostic Description. A platform-agnostic description is the *natural language description* for the general GEMM optimization techniques for different hardware platforms. As demonstrated in Section Preliminary, we have categorized the general optimization techniques for GEMM into five primitives: *tiling*, *reordering*, *layout*, *vectorization*, and *pipeline*. For each optimization primitive, the corresponding meta-prompts are pre-defined by the human experts to decrease the reasoning complexity and thus enhance the functional accuracy of LLMs.

Platform-Specific Hints. Platform-specific hints include the *natural language description* and *code skeleton* tailored to specific hardware platforms, which provide detailed hardware characteristics to generate optimized GEMM code across diverse hardware platforms. The platform-specific natural language descriptions integrate general GEMM optimization strategies with the hardware characteristics of the target platform. For example, on NVIDIA GPUs, dataflow progresses from global memory to shared memory and finally to local memory, which necessitates the use of multi-level tiling. In contrast, on RISC-V CPUs, only cache locality needs to be considered, so a single level of tiling is sufficient. The platform-specific code skeleton is also crucial for enhancing LLMs to generate the proper optimized GEMM code. It does not desire a complete piece of code, but rather includes only the nested for-loop structures (e.g., for tiling optimization), instruction examples (e.g., for vectorization optimization), or pseudocode representing the computing stages of the code (e.g., for pipeline optimization)

To help LLMs fully understand the architectural characteristics of diverse platforms (such as memory hierarchy and computing model), we have developed hint repositories containing pre-defined natural language descriptions and corresponding code snippets. This allows for the flexible combination of GEMM primitives and platform-specific hints to generate diverse prompts, a process that will be detailed in the next subsection.

Instantiation. The instantiation primarily describes the process from the general meta-prompts to the prompts that are tailored for specific platform. Concretely, before prompting to LLMs for generating high-performance code on a specific platform, QiMeng-GEMM first retrieves the essential clues from the pre-defined hint repository, and integrates the corresponding hints into the meta-prompt templates. An illustrative example of instantiation for the tiling meta-prompts on RISC-V CPU and NVIDIA GPU is shown in Figure 4. Regarding the RISC-V CPU, since only cache locality needs to be considered, the platform-specific hints required for the instantiation of tiling meta-prompts are quite simple, such as breaking down the original GEMM output into smaller blocks. Regarding the NVIDIA GPU, given its unique *warp* programming abstraction, the platform-specific hints include a basic explanation of the *warp* and a fundamental definition of warp-level tiling. Finally, a relevant code skeleton is provided to assist LLMs in understanding the semantics of this GEMM optimization. This instantiation mechanism reduces the need for manual intervention and can adaptively carry out platform-specific GEMM optimization, relying solely on a pre-defined repository of hints that outlines the architectural characteristics of the hardware.

Algorithm 1: Pseudocode of Auto-Tuning in QiMeng-GEMM

Input: A set of meta-prompts p_A, beam width k, the max iteration T
Output: High-performance GEMM code

1: Initialize the naive implementation of GEMM code c^0 as the root of tree $C \leftarrow \{c^0\}$
2: **for** $t < T$ **do** {$\triangleright$ t denotes the t-th iteration}
3:   **for** Each code c^{t-1} in C **do**
4:     LLMs suggest optimization primitive candidates
5:     **for** Each optimization primitive candidate mp **do**
6:       Instance mp wsith platform-specific hint
7:       $c^t = LLM(c^{t-1}, mp)$
8:       Add c^t to C_{new} with its performance
9:     **end for**
10:   **end for**
11:   Prune C_{new} to retain only the top k performance
12:   $C \leftarrow C_{\text{new}}$
13: **end for**
14: **return** The code in C with the highest performance

3.2 Auto-Tuning

To fully explore the large search space of GEMM optimizations and achieve higher performance, we propose the auto-tuning mechanism, which allows the LLMs to automatically search through different permutations of optimization primitives in a tree structure.

Algorithm 1 details the auto-tuning process. Concretely, it starts with the naive GEMM code generated by LLMs as the root node. In the following iterations of GEMM optimization, LLMs analyze the characteristics of current code and recommend multiple candidates from the given five GEMM optimization primitives. For each candidate, relevant platform-specific optimization hints are extracted from the hint repository, and the meta-prompts are instantiated into prompts for LLM interaction to generate a new optimized code. During this process, incorrect GEMM codes are pruned from the entire optimization tree, leaving only the *Top-K* results based on the evaluated performance. The auto-tuning process concludes when there is no further performance improvement in the optimized GEMM code or when the maximum number of iterations is reached, resulting in the final optimized code.

4 Evaluation

4.1 Experiment Setup

To study the effects of the proposed QiMeng-GEMM, experiments are organized around four main factors that influence performance. Due to space limitations, additional experiments are shown in the appendix.

Hardware Platforms. The experiments are conducted on various hardware platforms, including XuanTie C910 RISC-V CPU (Chen et al. 2020), NVIDIA RTX 4070 GPU, and A100 GPU.

LLMs. The proposed framework is validated on four SOTA LLMs, including the closed-source models of GPT-4o (OpenAI 2024) and Claude 3.5 sonnet (Claude3.5 2024), as well as the open-source models of DeepSeek-Coder-V2 (Zhu et al. 2024a) and Codestral-22B (Mistral.AI 2024).

Prompt Variants. The proposed meta-prompt is compared with vanilla prompt and CoT prompt (Wei et al. 2022). The vanilla prompt consists of basic LLM activation phrases, the fundamental concept of GEMM, and the function interfaces required. Based on the vanilla prompt, the CoT prompt adds a phrase to encourage step-by-step thinking in LLMs.

Matrix Dimensions. Experiments are conducted on typical matrix multiplication dimensions, prevalent in deep learning (LeCun et al. 2015), such as Llama2-7b (Touvron et al. 2023), Mistral-7B (Jiang et al. 2023) etc.

4.2 Overall Performance

Table 1 shows the performance of QiMeng-GEMM with several SOTA LLMs on different hardware platforms. Table 2 demonstrates the performance comparison between the QiMeng-GEMM and simple CoT methods. It is clearly evident from the table that our method can enable LLMs to generate code that exhibits both functional correctness and high performance.

QiMeng-GEMM outperforms the vanilla prompts on all four SOTA LLMs. Taking GPT-4o as an example, on C910 CPU, our method performs up to $113.67\times$ better than the vanilla prompts. On RTX 4070 and A100 GPUs, our approach achieves a performance improvement up to $9.65\times$ and $3.92\times$ respectively. More importantly, our method can even achieve better performance than the manually-optimized libraries such as OpenBLAS and cuBLAS. The code generated by QiMeng-GEMM significantly outperforms OpenBLAS up to $211\%(10.23/4.85)$ on C910 CPU and cuBLAS up to $115\%(10.55/9.14)$ on RTX 4070 GPU.

Subsequently, we will present the experimental observations pertaining to the four primary factors influencing performance, and provide insight into the underlying rationales. The primary experimental results are presented in the main text. More details please refer to Appendix.

- **The Impact of Hardware Platforms: Using our delicate meta-prompt, the code automatically generated by LLMs generally achieves better performance on RISC-V CPUs than on NVIDIA GPUs.** The performance improvements on C910 CPU and RTX4070 GPU are up to $113.67\times$ and $9.65\times$ respectively, which highlights the disparity in speedup between the CPU and GPU. Unlike the SIMD parallelization on CPUs, GPUs use SIMT parallelization, which results in a coding pattern for GPUs that differs from the conventional C programming paradigm typically used for CPU code. Unlike the conventional C code style in CPUs, the CUDA C programming paradigm on GPUs achieves functionality by writing kernel functions that are executed massively across each thread. This results in the outer loops of the code being implicit in CUDA C, causing a domain shift from the extensive C-style codes that LLMs are typically trained on.
- **The Impact of LLMs: The more powerful LLMs are, the greater performance improvement can be achieved in generating complex task code with QiMeng-GEMM.** As experiments with different LLMs, two closed-source LLMs demonstrated solid performance optimizations across different hardware platforms up to $9.65\times$ on RTX4070. Among the open-source models, the larger-parameter Deepseek-v2 (with 236B parameters) gains relatively better results of $8.79\times$, while the smaller-parameter Codestral-22B (with 22B parameters) exhibits lower performance on GPU. Solving challenging tasks such as GEMM optimization requires robust capabilities of LLMs. During the pre-training stage, LLMs need to learn enough domain knowledge from the training data to facilitate generalization to new tasks with minimal prompting. During the inference stage, LLMs need to understand over-length context presenting various domain knowledge and examples. Besides, LLMs must be able to follow complex prompt instructions to automatically schedule multi-turn conversations and integrate results.
- **The Impact of Prompt Mechanism: The informative, adaptive and iterative prompt can efficiently activate the potential of LLMs to generate high-performance code.** Our meta-prompt constantly outperforms vanilla prompt and simple CoTs. It not only includes a highly refined and critical description of optimization methods but also incorporates the core code structure and code hints essential for implementing the optimization techniques. For example, in *warp tiling* method in GPU, it is needed to incorporate an additional granularity level in the tiling blocking strategy. We just add the "for-loops" *for(wm;wm<WM;wm++)* to templates and then generate the code optimized by warp tiling. Our prompts can fully unlock the potential of LLMs in a concise and efficient manner, guide them to focus on high-quality solution spaces, and find the correct optimal solution quickly.
- **The Impact of Matrix Dimensions: The larger the matrix dimensions are, the greater performance gains can be achieved by QiMeng-GEMM.** Taking C910 CPU as an example, we achieve $113.67\times$ speedup on the matrix of dimension of 4096, while $34.15\times$ of 512. In the case of GEMM algorithms on CPUs, the vanilla implementation often faces an increased cache miss rate as matrix dimensions grow, leading to a decrease in

Hardware	LLMs	256	512	768	1024	1536	2048	4096
C910 (CPU)	GPT-4o (Achiam et al. 2023)	0.27	0.18	0.19	0.14	0.18	0.10	0.09
	+Our QiMeng-GEMM	**9.22**	**9.40**	**9.93**	**9.91**	**9.81**	**10.08**	**10.23**
		(↑34.15×)	**(↑52.22×)**	**(↑52.26×)**	**(↑70.79×)**	**(↑54.50×)**	**(↑100.80×)**	**(↑113.67×)**
	Claude 3.5 Sonnet (Claude3.5 2024)	2.79	2.62	2.72	2.64	2.75	1.56	0.74
	+Our QiMeng-GEMM	7.42	7.07	6.93	6.83	7.09	6.33	6.35
		(↑2.66×)	(↑2.70×)	(↑2.55×)	(↑2.59×)	(↑2.58×)	(↑4.06×)	(↑8.58×)
	DS-Coder-V2 (Zhu et al. 2024a)	0.76	0.60	0.48	0.61	0.67	0.46	0.10
	+Our QiMeng-GEMM	1.44	1.33	1.30	1.24	1.27	1.17	1.15
		(↑1.89×)	(↑2.22×)	(↑2.71×)	(↑2.03×)	(↑1.90×)	(↑2.54×)	(↑11.50×)
	Codestral-22B (Jiang et al. 2024)	0.16	0.09	0.07	0.05	0.08	0.03	0.02
	+Our QiMeng-GEMM	0.86	0.48	0.63	0.28	0.40	0.30	0.17
		(↑5.38×)	(↑5.33×)	(↑9.00×)	(↑5.60×)	(↑5.00×)	(↑10.00×)	(↑8.50×)
	OpenBLAS 0.3.27 (Zhang et al. 2012)	7.35	5.82	5.49	5.01	4.94	5.11	4.85
Hardware	**LLMs**	**512**	**768**	**1024**	**2048**	**4096**	**8192**	**16384**
RTX 4070 (GPU)	GPT-4o (Achiam et al. 2023)	1.53	1.68	1.77	1.78	1.65	1.48	1.46
	+QiMeng-GEMM	**8.17**	**10.55**	**11.47**	**13.31**	**14.16**	**14.28**	**13.92**
		(↑5.34×)	**(↑6.28×)**	**(↑6.48×)**	**(↑7.48×)**	**(↑8.58×)**	**(↑9.65×)**	**(↑9.53×)**
	Claude 3.5 Sonnet (Claude3.5 2024)	1.49	1.59	1.71	1.79	1.61	1.47	1.42
	+Our QiMeng-GEMM	5.13	10.21	10.94	12.10	14.05	14.08	13.11
		(↑3.44×)	(↑6.42×)	(↑6.40×)	(↑6.76×)	(↑8.73×)	(↑9.58×)	(↑9.23×)
	DS-Coder-V2 (Zhu et al. 2024a)	1.42	1.64	1.63	1.71	1.58	1.42	1.38
	+Our QiMeng-GEMM	4.04	8.44	9.14	10.73	12.65	12.48	10.72
		(↑2.85×)	(↑5.15×)	(↑5.61×)	(↑6.27×)	(↑8.01×)	(↑8.79×)	(↑7.77×)
	Codestral-22B (Jiang et al. 2024)	1.35	1.42	1.62	1.68	1.51	1.35	1.27
	+Our QiMeng-GEMM	1.55	1.40	1.73	1.82	1.46	1.40	1.30
		(1.15×)	(0.99×)	(1.07×)	(1.08×)	(0.97×)	(1.03×)	(1.02×)
	cuBLAS 12.1 (NVIDIA 2023a)	7.81	9.14	10.79	12.77	12.78	14.25	12.74
A100 (GPU)	GPT-4o (Achiam et al. 2023)	3.62	3.98	4.19	4.27	4.71	4.72	4.73
	+Our QiMeng-GEMM	**7.02**	**10.48**	**12.61**	**15.27**	**17.99**	**18.36**	**18.55**
		(↑1.94×)	**(↑2.63×)**	**(↑3.01×)**	**(↑3.57×)**	**(↑3.82×)**	**(↑3.89×)**	**(↑3.92×)**
	Claude 3.5 Sonnet (Claude3.5 2024)	3.57	4.26	4.64	5.33	5.27	5.32	5.34
	+Our QiMeng-GEMM	6.91	10.29	12.04	16.17	**18.27**	**18.83**	**18.89**
		(↑1.91×)	(↑2.42×)	(↑2.59×)	(↑3.03×)	**(↑3.27×)**	**(↑3.54×)**	**(↑3.47×)**
	DS-Coder-V2 (Zhu et al. 2024a)	3.63	3.98	4.20	4.61	4.70	4.71	4.74
	+Our QiMeng-GEMM	6.84	11.14	11.37	13.46	17.65	18.5	18.61
		(↑1.88×)	(↑2.80×)	(↑2.68×)	(↑2.88×)	(↑3.76×)	(↑3.93×)	(↑3.92×)
	Codestral-22B (Jiang et al. 2024)	3.61	3.94	4.20	4.33	4.68	4.72	4.73
	+Our QiMeng-GEMM	3.71	4.18	4.16	4.67	4.77	4.64	4.86
		(1.03×)	(1.06×)	(0.99×)	(1.08×)	(1.02×)	(0.98×)	(1.03×)
	cuBLAS 12.1 (NVIDIA 2023a)	8.60	12.32	16.26	17.20	18.97	19.07	19.22

Table 1: Performance comparison on various hardware, LLMs, and matrix dimensions. The first line of each LLM indicates the vanillina prompt. Performance is measured with GFLOPS and TFLOPS for CPU and GPU, respectively.

GFLOPS as the matrix size increases. More importantly, with the rapid growth of LLM sizes and the widespread use of long contexts in recent years, GEMM dimensions also increase rapidly. Performance optimization for larger matrices has become a key factor affecting overall performance. QiMeng-GEMM gains significantly higher speedup for larger matrices, which perfectly aligns with this trend.

	512	1024	2048
GPT-4o	0.18	0.14	0.10
CoT	0.81 (↑ 4.50×)	0.63 (↑ 4.50×)	0.18 (↑ 1.80×)
+Our QiMeng-GEMM	**9.40 (↑ 52.22×)**	**9.91 (↑ 70.79×)**	**10.08 (↑ 100.80×)**
Claude 3.5	2.62	2.64	1.56
+CoT	3.66 (↑ 1.40×)	2.79 (↑ 1.06×)	2.47 (↑ 1.58×)
+Our QiMeng-GEMM	**7.07 (↑ 2.70×)**	**6.83 (↑ 2.59×)**	**6.33 (↑ 4.06×)**

Table 2: GFLOPS performance comparison of typical prompt methods on C910 CPU.

Template	Instantiation	w/o Hint	w/ Hint
Tiling	Block Tiling	4.74	4.78
	Thread Tiling	4.31	14.59
	Warp Tiling	-	16.41
Vectorization	Float4	-	16.85
Pipeline	Double Buffer	16.72	18.51

Table 3: Platform-specific hints effectively activate the programming capabilities of LLMs for generating correct and high-performance GEMM codes on A100 GPU. Performance is measured with TFLOPS.

	A100 GPU		C910 CPU	
	Time	TFLOPS	Time	GFLOPS
Junior Coder	24 days	11.70	7 days	1.45
Senior Coder	5 days	16.34	3 days	3.08
QiMeng-GEMM	10 mins	15.27	7 mins	9.91

Table 4: Comparison of development cost.

4.3 Ablation Study

The Impact of Platform-Specific Hints. As shown in Table 3, the proposed platform-specific hints assist LLMs in fully comprehending platform architecture characteristics, thereby significantly enhancing both the functional correctness and performance of the code generated by LLMs.

Concretely, for simple optimization techniques such as block tiling, the performance gap with or without our hints is not significant. However, for thread tiling, the proposed hints can boost the performance of the code generated by LLMs from 4.31 TFLOPS to 14.59 TFLOPS. Moreover, without the platform-specific hints, LLMs are unable to generate correctly executable code for the GEMM optimization tailored to specific hardware, such as warp tiling, thread tiling, and float4 vectorizaion.

The Impact of Auto-Tuning. To better illustrate QiMeng-GEMM's ability to generate high-performance GEMM code, we examine the auto-tuning process as shown in Figure 5. It is important to note that we retained all search trails without pruning incorrect or low-performance code, allowing for a more comprehensive evaluation of the search space quality. Results indicate that among the 25 search trails, QiMeng-GEMM consistently generates functionally correct

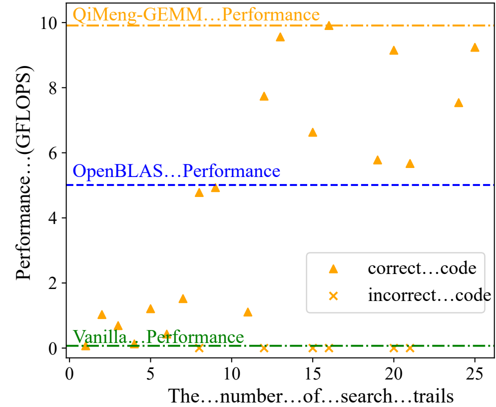

Figure 5: Search exploration to illustrate the possibility of achieving positive results on C910 CPU. The horizontal axis represents the iteration number of the search, and the vertical axis represents the performance of the code searched.

GEMM code (76%), and 9 out of 25 trails (36%) surpass the performance of the manually optimized OpenBLAS library. Notably, the best performance achieved by QiMeng-GEMM even exceeds OpenBLAS by 100%.

The Impact of Development Costs. Table 4 compares the development costs of QiMeng-GEMM for the matrix of dimension 1024 on two representative platforms. Two CS Master's students with three years of coding experience and two software engineers with five years of experience serve as junior and senior coders, respectively. Results show that, on GPU, the proposed method reduces development time from 24 days to 10 minutes while also surpassing 30% of the junior coders' performance. Compared to a senior coder, our approach reduces the development cost by 240× (assuming an eight-hour workday), while maintaining comparable performance. Similarly, on CPU, it reduces development time from 3 days to 7 minutes, exceeding 221% of the senior coders' performance.

5 Conclusion

In this paper, we propose a novel prompt mechanism called QiMeng-GEMM to fully activate LLMs in high-performance GEMM code generation. Through extensive experimental comparison, we find that the key challenge resides in unleashing the potential of LLMs to comprehend various architectural characteristics by meta-prompts and automatically search for the optimal combinations of meta-prompts for optimizing GEMM. The results show that the code generated by our approach could outperform hand-optimized libraries crafted by human experts. Crucially, while human experts often take months to optimize GEMM, our approach reduces the development cost by over 240×.

The progress of QiMeng-GEMM has demonstrated significant potential within rapidly evolving architectures. In the future, we plan to extend our adaptations to more architectures and instruction sets. The novel methodology delineated in this paper will be made accessible as a dedicated tool, to encourage future research in the field.

Acknowledgments

This work is partially supported by the National Key R&D Program of China(under 2022YFB4501603) , the NSF of China(under Grants 92364202, 61925208, U22A2028, 62222214, 62341411, 62102398, 62102399, U20A20227, 62372436, 62302478, 62302482, 62302483, 62302480), 2022 Fundamental Disciplines Top Quality Student Training Program 2.0 Project, Strategic Priority Research Program of the Chinese Academy of Sciences, (Grant No.XDB0660300, XDB0660301, XDB0660302), CAS Project for Young Scientists in Basic Research(YSBR-029), Youth Innovation Promotion Association CAS and Xplore Prize, and Major Program of ISCAS (Grant No. ISCAS-ZD-202402).

References

Achiam, J.; Adler, S.; Agarwal, S.; et al. 2023. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*.

Anam, M. A.; Whatmough, P. N.; Andreopoulos, Y.; et al. 2013. Precision-energy-throughput scaling of generic matrix multiplication and discrete convolution kernels via linear projections. In *The 11th IEEE Symposium on Embedded Systems for Real-time Multimedia*, 21–30.

Bairi, R.; Sonwane, A.; Kanade, A.; et al. 2024. CodePlan: Repository-Level Coding using LLMs and Planning. *Proc. ACM Softw. Eng.*, 1(FSE).

Bi, J.; Guo, Q.; Li, X.; et al. 2023. Heron: Automatically constrained high-performance library generation for deep learning accelerators. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 314–328.

Chen, C.; Xiang, X.; Liu, C.; et al. 2020. Xuantie-910: A commercial multi-core 12-stage pipeline out-of-order 64-bit high performance RISC-V processor with vector extension: Industrial product. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 52–64. IEEE.

Chen, H.; Wen, Y.; Cheng, L.; et al. 2024. AutoOS: Make Your OS More Powerful by Exploiting Large Language Models. In *Forty-first International Conference on Machine Learning*.

Chen, M.; Tworek, J.; Jun, H.; et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Claude3.5. 2024. Claude 3.5 Sonnet. https://www.anthropic.com/news/claude-3-5-sonnet. Accessed: 2024-07-17.

Faingnaert, T.; Besard, T.; De Sutter, B.; et al. 2022. Flexible Performant GEMM Kernels on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 33(9): 2230–2248.

Greengard, S. 2020. Will RISC-V revolutionize computing? *Communications of the ACM*, 63(5): 30–32.

Hennessy, J. L.; and Patterson, D. A. 2019. A new golden age for computer architecture. *Communications of the ACM*, 62(2): 48–60.

Holt, S.; et al. 2024. L2MAC: Large Language Model Automatic Computer for Extensive Code Generation. In *The Twelfth International Conference on Learning Representations*.

Intel. 2023. Using oneMKL for Matrix Multiplication. https://www.intel.com/content/www/us/en/docs/onemkl/tutorial-c/2023-2/overview.html. Accessed: 2024-07-22.

Jang, J.; Kim, Y.; Lee, J.; et al. 2024. FIGNA: Integer Unit-Based Accelerator Design for FP-INT GEMM Preserving Numerical Accuracy. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 760–773. IEEE.

Jia, Y. 2014. *Learning semantic image representations at a large scale*. University of California, Berkeley.

Jian Weng, R. Y.; et al. 2023. How to optimize GEMM on CPU. https://tvm.apache.org/docs/how_to/optimize_operators/opt_gemm.html. Accessed: 2024-07-22.

Jiang, A. Q.; Sablayrolles, A.; Mensch, A.; et al. 2023. Mistral 7B. arXiv:2310.06825.

Jiang, A. Q.; Sablayrolles, A.; Roux, A.; et al. 2024. Mixtral of experts. *arXiv preprint arXiv:2401.04088*.

Jouppi, N. P.; Kurian, G.; Li, S.; et al. 2023. TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings. *Proceedings of the 50th Annual International Symposium on Computer Architecture*.

Jouppi, N. P.; Young, C.; Patil, N.; Patterson, D.; Agrawal, G.; Bajwa, R.; Bates, S.; Bhatia, S.; Boden, N.; Borchers, A.; et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, 1–12.

Katel, N.; Khandelwal, V.; Bondhugula, U.; et al. 2021. High performance gpu code generation for matrix-matrix multiplication using mlir: some early results. *arXiv preprint arXiv:2108.13191*.

Krainiuk, M.; Goli, M.; Pascuzzi, V. R.; et al. 2021. oneAPI Open-Source Math Library Interface. In *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 22–32.

Kurzak, J.; Tomov, S.; and Dongarra, J. 2012. Autotuning GEMM Kernels for the Fermi GPU. *IEEE Transactions on Parallel and Distributed Systems*, 23(11): 2045–2057.

LeCun, Y.; Bengio, Y.; Hinton, G.; et al. 2015. Deep learning. *nature*, 521(7553): 436–444.

Li, R.; He, L.; Liu, Q.; et al. 2024. CONSIDER: Commonalities and Specialties Driven Multilingual Code Retrieval Framework. In *AAAI Conference on Artificial Intelligence*.

Li, Y.; Choi, D.; Chung, J.; et al. 2022. Competition-level code generation with AlphaCode. *Science*, 378(6624): 1092–1097.

Mistral.AI. 2024. Codestral. https://mistral.ai/news/codestral/. Accessed: 2024-07-19.

Nijkamp, E.; Pang, B.; Hayashi, H.; et al. 2022. CodeGen: An Open Large Language Model for Code with Multi-Turn

Program Synthesis. In *International Conference on Learning Representations*.

Norrie, T.; Patil, N.; Yoon, D. H.; Kurian, G.; Li, S.; Laudon, J.; Young, C.; Jouppi, N.; and Patterson, D. 2021. The Design Process for Google's Training Chips: TPUv2 and TPUv3. *IEEE Micro*, 41(2): 56–63.

NVIDIA. 2023a. CUBLAS LIBRARY user guide v12.1. https://docs.nvidia.com/cuda/archive/12.1.0. Accessed: 2024-07-22.

NVIDIA. 2023b. Matrix Multiplication Background. https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html. Accessed: 2024-07-21.

Olausson, T. X.; Inala, J. P.; Wang, C.; et al. 2023. Is Self-Repair a Silver Bullet for Code Generation? In *The Twelfth International Conference on Learning Representations*.

OpenAI. 2024. GPT-4o. https://openai.com/index/hello-gpt-4o/. Accessed: 2024-07-17.

Phothilimthana, P. M.; Elliott, A. S.; Wang, A.; Jangda, A.; Hagedorn, B.; Barthels, H.; Kaufman, S. J.; Grover, V.; Torlak, E.; and Bodik, R. 2019. Swizzle inventor: data movement synthesis for GPU kernels. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 65–78.

Purini, S.; et al. 2013. Finding good optimization sequences covering program space. *ACM Trans. Archit. Code Optim.*, 9(4).

Rasch, A.; Schulze, R.; Steuwer, M.; and Gorlatch, S. 2021. Efficient Auto-Tuning of Parallel Programs with Interdependent Tuning Parameters via Auto-Tuning Framework (ATF). *ACM Trans. Archit. Code Optim.*, 18(1).

Roziere, B.; Gehring, J.; Gloeckle, F.; et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Shi, K.; Dai, H.; Ellis, K.; et al. 2022. CrossBeam: Learning to Search in Bottom-Up Program Synthesis. In *The Tenth International Conference on Learning Representations (ICLR)*.

Shi, K.; Dai, H.; Li, W.-D.; et al. 2023. LambdaBeam: Neural Program Search with Higher-Order Functions and Lambdas. In Oh, A.; Naumann, T.; Globerson, A.; Saenko, K.; Hardt, M.; and Levine, S., eds., *Advances in Neural Information Processing Systems*, volume 36, 51327–51346. Curran Associates, Inc.

Shin, J.; Nam, J.; et al. 2021. A survey of automatic code generation from natural language. *Journal of Information Processing Systems*, 17(3): 537–555.

Sitaraman, G. 2024. AMD Matrix Cores. https://fs.hlrs.de/projects/par/events/2024/GPU-AMD/day4/20.%20AMD_Matrix_Cores.pdf. Accessed: 2024-07-21.

Sui, B.; Shen, J.; Sun, C.; et al. 2024. MACO: Exploring GEMM Acceleration on a Loosely-Coupled Multi-Core Processor. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 1–6. IEEE.

Tan, G.; Li, L.; Triechle, S.; et al. 2011. Fast implementation of DGEMM on Fermi GPU. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 1–11.

Touvron, H.; Martin, L.; Stone, K.; et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Wang, Y.; Le, H.; Gotmare, A. D.; et al. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. arXiv:2305.07922.

Wei, C.; Jia, H.; Zhang, Y.; et al. 2024. IrGEMM: An Input-Aware Tuning Framework for Irregular GEMM on ARM and X86 CPUs. *IEEE Transactions on Parallel and Distributed Systems*, 35(9): 1672–1689.

Wei, J.; Wang, X.; Schuurmans, D.; et al. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems*, volume 35, 24824–24837. Curran Associates, Inc.

Zhang, X.; Wang, Q.; Zhang, Y.; et al. 2012. openblas: a high performance blas library on loongson 3a cpu. *Journal of Software*, 22(zk2): 208–216.

Zhong, L.; et al. 2024. Can LLM Replace Stack Overflow? A Study on Robustness and Reliability of Large Language Model Code Generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, 21841–21849.

Zhu, Q.; Guo, D.; Shao, Z.; et al. 2024a. DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence. *arXiv preprint arXiv:2406.11931*.

Zhu, R.-J.; Zhang, Y.; Sifferman, E.; et al. 2024b. Scalable MatMul-free Language Modeling. *arXiv preprint arXiv:2406.02528*.