QiMeng-TensorOp: One-Line Prompt is Enough for High-Performance Tensor Operator Generation with Hardware Primitives

Xuzhi Zhang[1,3] , **Shaohui Peng**[1] , **Qirui Zhou**[2,3] , **Yuanbo Wen**[2] , **Qi Guo**[2,3] ,
Ruizhi Chen[1] , **Xinguo Zhu**[1,3] , **Weiqiang Xiong**[1,3] , **Haixin Chen**[2,3] , **Congying Ma**[1,4] ,
Ke Gao[1] , **Chen Zhao**[1] , **Yanjun Wu**[1,3] , **Yunji Chen**[2,3] and **Ling Li**[1,3]*
[1]Institute of Software Chinese Academy of Sciences
[2]Institute of Computing Technology, Chinese Academy of Sciences
[3]University of Chinese Academy of Sciences
[4]Peking University
{zhangxuzhi2023, pengshaohui, ruizhi, gaoke, zhaochen, yanjun, liling}@iscas.ac.cn, {zhuxinguo23, xiongweiqiang20, chenhaixin24}@mails.ucas.ac.cn, {zhouqirui22s, guoqi, cyj}@ict.ac.cn, wenyb@mail.ustc.edu.cn, 2100013182@stu.pku.edu.cn

Abstract

Computation-intensive tensor operators constitute over 90% of the computations in Large Language Models (LLMs) and Deep Neural Networks. Automatically and efficiently generating high-performance tensor operators with hardware primitives is crucial for diverse and ever-evolving hardware architectures like RISC-V, ARM, and GPUs, as manually optimized implementation takes at least months and lacks portability. LLMs excel at generating high-level language codes, but they struggle to fully comprehend hardware characteristics and produce high-performance tensor operators.

We introduce a tensor-operator auto-generation framework with a one-line user prompt (QiMeng-TensorOp), which enables LLMs to automatically exploit hardware characteristics to generate tensor operators with hardware primitives, and tune parameters for optimal performance across diverse hardware. Experimental results on various hardware platforms, SOTA LLMs, and typical tensor operators demonstrate that QiMeng-TensorOp effectively unleashes the computing capability of various hardware platforms, and automatically generates tensor operators of superior performance. Compared with vanilla LLMs, QiMeng-TensorOp achieves up to 1291× performance improvement. Even compared with human experts, QiMeng-TensorOp could reach 251% of OpenBLAS on RISC-V CPUs, and 124% of cuBLAS on NVIDIA GPUs. Additionally, QiMeng-TensorOp also significantly reduces development costs by 200× compared with human experts.

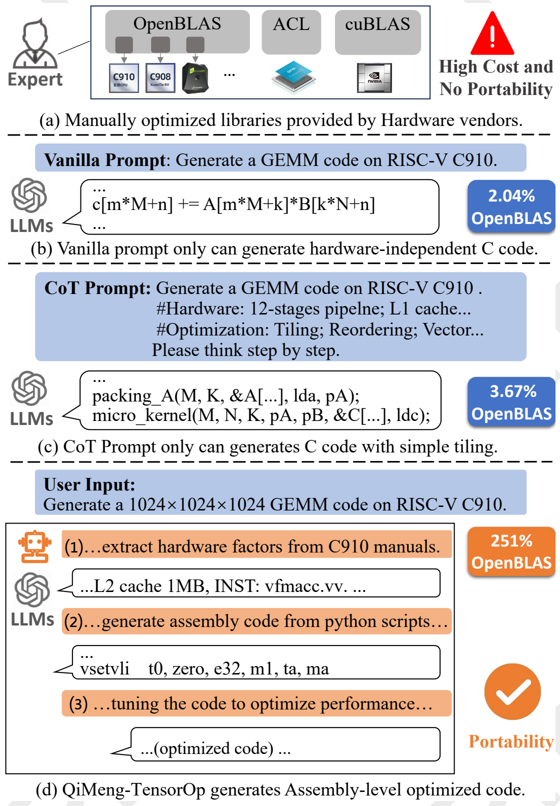
1 Introduction

Tensor operators, like General Matrix Multiplication (GEMM) and Convolution (Conv) [Chen *et al.*, 2018b], are critical in various mathematical and computational fields, especially in deep learning, as they constitute over 90% of the computations in LLMs and Deep Neural Networks(DNNs) [Sze *et al.*, 2017; Kim *et al.*, 2024]. The significance and unique computational requirements of tensor operators have driven heterogeneity and complexity in hardware design, such as Tensor Core in NVIDIA GPU [Markidis *et al.*, 2018], RISC-V vector extension (RVV), and deep learning accelerators like Google TPU [Jouppi *et al.*, 2017] and Cambricon [Liu *et al.*, 2016].

Implementing tensor operators with hardware primitives (such as assembly instructions for CPU and hardware intrinsic for CUDA Tensor core) is the only way to fully maximize hardware performance [Xiao *et al.*, 2021; Liu *et al.*, 2022; Igual *et al.*, 2023; Guillermo *et al.*, 2024; Castelló *et al.*, 2024; Wu *et al.*, 2024]. Hardware primitives provide programmers with precise control over hardware resources (including computing units, registers, memory, etc.), thereby enabling exceptional performance [Zhai *et al.*, 2024]. For instance, an assembly implementation of GEMM can yield over 62,000× performance improvement compared to the vanilla Python implementation [Hennessy and Patterson, 2019].

However, designing high-performance tensor operators with hardware primitives is challenging, as it requires a deep comprehension of hardware architectures. Besides, it is intricate and often results in low efficiency and high error rates. Existing tensor operators are mainly implemented with two paradigms, manually optimized libraries and auto-compilers. Manually optimized libraries provided by hardware vendors, such as MKL(Intel Math Kernel Library) [Krainiuk *et al.*, 2021] and ACL(Arm Compute Library) for CPUs, cuBLAS and cuDNN for GPUs, are developed by human experts using hardware primitives and specific optimizations for various platforms. This process is time-consuming, often taking

*Corresponding author



(a) Manually optimized libraries provided by Hardware vendors.

Vanilla Prompt: Generate a GEMM code on RISC-V C910.

```
...
c[m*M+n] += A[m*M+k]*B[k*N+n]
...
```

(b) Vanilla prompt only can generate hardware-independent C code.

CoT Prompt: Generate a GEMM code on RISC-V C910 .
#Hardware: 12-stages pipelne; L1 cache...
#Optimization: Tiling; Reordering; Vector...
Please think step by step.

```
...
packing_A(M, K, &A[...], lda, pA);
micro_kernel(M, N, K, pA, pB, &C[...], ldc);
```

(c) CoT Prompt only can generates C code with simple tiling.

User Input:
Generate a 1024×1024×1024 GEMM code on RISC-V C910.

(1)...extract hardware factors from C910 manuals.

...L2 cache 1MB, INST: vfmacc.vv. ...

(2)...generate assembly code from python scripts...

```
...
vsetvli   t0, zero, e32, m1, ta, ma
```

(3) ...tuning the code to optimize performance...

...(optimized code) ...

(d) QiMeng-TensorOp generates Assembly-level optimized code.

Figure 1: Comparison of tensor operator optimization paradigms

months to optimize tensor operators, and these libraries also lack portability across different platforms. Auto-compilers, such as Halide [Ragan *et al.*, 2013], TVM [Chen *et al.*, 2018a], and Ansor [Zheng *et al.*, 2020; Shao *et al.*, 2022; Bi *et al.*, 2023; Zhai *et al.*, 2023; Zhai *et al.*, 2024], explore a vast program space to generate efficient tensor operators. However, they still require human experts to manually define hardware-specific rules and backend implementations to optimize and deploy tensor operations. Despite alleviating certain manual labor, they necessitate significant expertise and involve high development costs, as exemplified by the arduous challenge of deploying compilers such as TVM(Tensor Virtual Machine) on RISC-V CPUs [Chen *et al.*, 2020b].Consequently, existing paradigms struggle with development cost and portability, making them insufficient to keep up with rapid hardware advancements.

LLMs (GPT-4o [OpenAI, 2025], DeepSeek-V3 [DeepSeek-AI *et al.*, 2024]) have achieved remarkable progress in code generations, making natural language to code (NL2Code) one of the most popular paradigms [Austin *et al.*, 2021; Zan *et al.*, 2022; Athiwaratkun *et al.*, 2022] However, existing LLM-based code generation researches [Lu *et al.*, 2022; Li *et al.*, 2023; Rozière *et al.*, 2023; Gunasekar *et al.*, 2023] mainly focus on high-level language code generation, and are incapable of hardware-primitive-level code generation. LLMs yet struggle to fully comprehend hardware characteristics and correctly manipulate hardware resources to implement and optimize assembly code, let alone high-performance tensor operator generation with simple prompts.

In summary, we observe two major challenges for LLMs in automatically generating hardware-primitive-level tensor operators. The first challenge is enabling LLMs to comprehend hardware architectures and accurately utilize hardware primitives to implement tensor operators. The second challenge is optimizing the performance of the generated tensor operators through hardware intrinsic optimization techniques.

To address the challenges, we propose the QiMeng-TensorOp framework that automatically generates high-performance tensor operators with hardware primitives on various platforms. QiMeng-TensorOp only needs a one-sentence prompt from user, which describes the target tensor operator and hardware, as shown in Figure 1. QiMeng-TensorOp uses general hints to trigger LLMs to comprehend hardware optimization and auto-extract target hardware factors. Then it leverages LLMs and extracted factors to generate sketch and kernel codes with hardware primitives for target operators. Finally, it exploits LLMs' in-context learning via MCTS to uncover the optimization opportunities for generated codes. In short, QiMeng-TensorOp effectively harnesses LLMs' knowledge and capabilities to comprehend and apply optimization techniques on various hardware platforms, enabling efficient auto-generation and tuning of tensor operators with hardware primitives.

To our knowledge, we are *the first to automatically generate high-performance tensor operators with hardware primitives by exploiting LLMs*. Our key contributions are:

- We propose a framework for automatically generating tensor operators at the hardware primitive level across various platforms, requiring only a single sentence from users to describe the target operator and hardware.
- We develop general hardware intrinsic optimization hints and workflow to help LLMs comprehend hardware and optimization techniques, allowing to automatically extract information from manuals to generate tensor operators with hardware primitives.
- We design an LLM-assisted MCTS algorithm that effectively enhances the efficiency and performance of tuning primitive-level tensor operators on specific hardware.
- Extensive evaluations across diverse hardware platforms and tensor operators (GEMM and Conv) of various dimensions demonstrate QiMeng-TensorOp significant performance promotion (up to 1291× of vanilla prompt and up to 251% than manually optimized libraries) and development cost reduction (200× than senior coder).

2 Preliminary

2.1 Tensor Operator

GEMM and Conv, the most important tensor operators, are computation-intensive.

GEMM refers to the multiplication of two dense matrices, $A \in R^{m\times k}$ and $B \in R^{k\times n}$, as $(AB)_{ij} = \sum_{q=1}^{k} A_{iq} \cdot B_{qj}$.

Conv slides a filter (K) on an input (X), and calculates element-wise dot product, $Y(i,j) = \sum_{m=0}^{M-1}\sum_{n=0}^{N-1} X(i +$

$m, j+n) \cdot K(m,n)$. Conv is commonly implemented by converting the input and filter tensors into 2D matrices with Image-to-Column and then calling GEMM in BLAS libraries.

2.2 Optimization Techniques

The optimization techniques for tensor operators can be essentially classified into several fundamental operations across hardware architectures.

Tiling (T) decomposes a matrix into smaller blocks to improve memory access [Faingnaert *et al.*, 2021].

Reordering (R) exchanges a for-loops to boost memory access efficiency [Anam *et al.*, 2013].

Vectorization (V) packs matrix data to use vector instructions for computation and memory access [Katel *et al.*, 2021].

Layout (L) rearranges matrix data to better fit the hardware's memory access patterns [Kurzak *et al.*, 2012].

Pipeline (P) overlaps the computation and memory access to minimize the memory access latency [Tan *et al.*, 2011].

The five optimization techniques vary in implementation challenge [Feng *et al.*, 2023]. Vectorization, Layout, and Pipeline need hardware primitives for specialized data movement and computation, while Tiling and Reordering involve designing high-level task and data allocation strategies to fit hardware characteristics.

2.3 Hardware Factors

We summarize four key hardware factors essentially impacting the implementation and optimization of tensor operators.

Memory Hierarchy (MH) refers to how hardware organizes and manages different levels of memory, such as L1/L2/L3 cache structures of CPUs [Wu *et al.*, 2021] and the global/shared memory of NVIDIA GPUs [Dally *et al.*, 2021]. Memory Hierarchy is crucial for efficient memory access and optimizations like Tiling and Reordering.

Instructions (INST) are the basic operation units of computation and data movement, such as the RVV instructions in the RISC-V architecture, NEON(ARM's SIMD) instructions in the ARM architecture, and CUDA Templates (CuTe) of NVIDIA GPU Tensor Cores. Instructions determine the hardware primitives for tensor operator implementation [Xiao *et al.*, 2021], and optimization techniques like Vectorization.

Vector/Scalar Registers of CPUs ((V)R) refers to the number and width of tensor/scalar registers. They are crucial for data movement and computation efficiency in tensor operators, affecting the generation of vector instructions and the granularity of Pipeline optimizations.

Streaming Processor Information of GPUs (SMs) includes the number of SMs, and the number of CUDA Cores and Tensor Cores within each SM [Choquette *et al.*, 2021]. It determines the grid and block dimension task allocation and data tiling when generating CUDA kernels.

3 Method

In this section, we introduce the QiMeng-TensorOp framework. It enables LLMs to comprehend hardware architectures, auto-generate high-performance tensor operators with hardware primitives, and optimize the generated code via MCTS. As depicted in Figure 2, it requires just a one-line user prompt and comprises three key components (a detaled).

3.1 Hardware Architecture Comprehending

The Hardware Architecture Comprehending consists of two parts. The Hardware Intrinsic Optimization Hints activate LLMs' comprehension of hardware and optimization, guiding subsequent generation, while the Hardware Factor Extraction leverages target hardware characteristics to further implement cross-platform optimization automation.

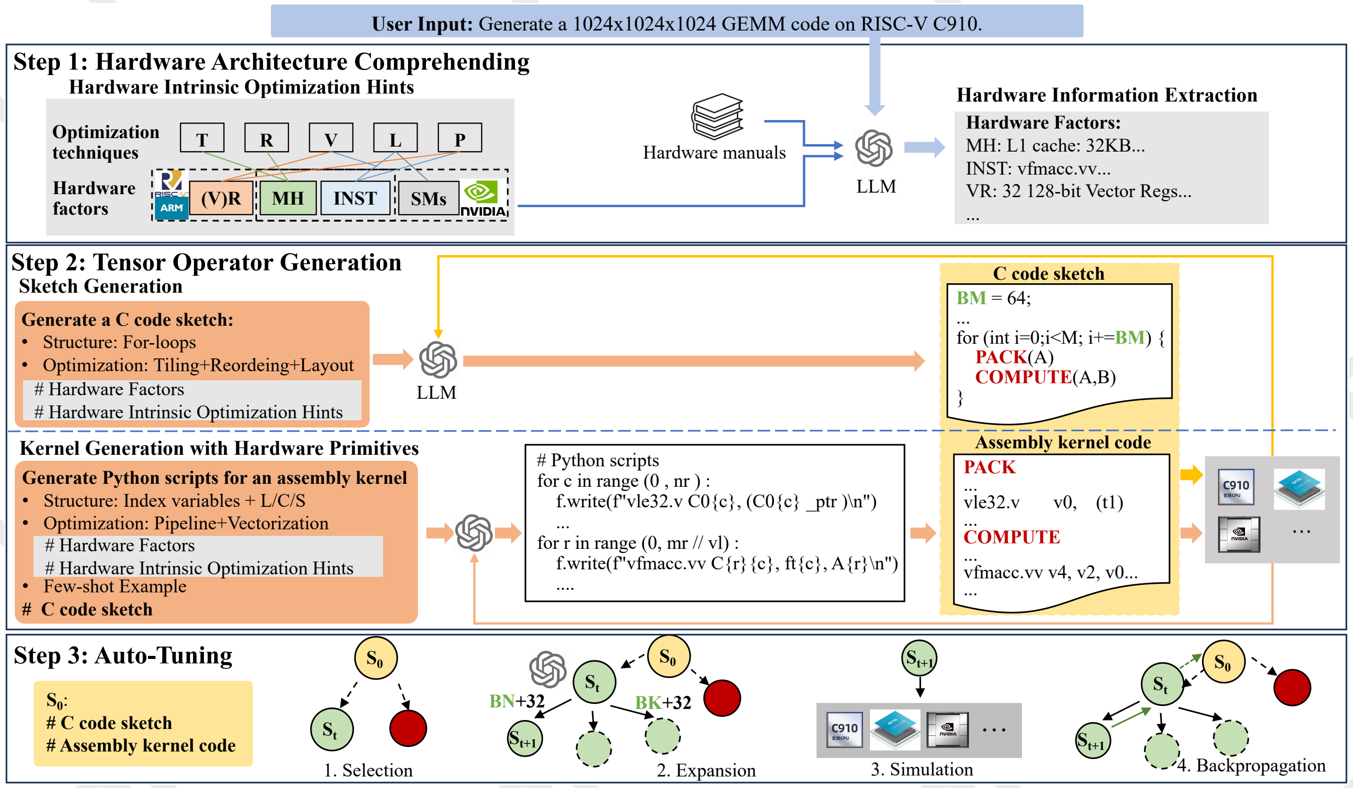
Hardware Intrinsic Optimization Hints. These hints serve as the background knowledge for LLMs to automatically conduct hardware factor extraction and tensor operator generation. As in Figure 2, we summarize the description of five general tensor operator optimization primitives and their relationship with four key hardware factors (as described in Section 2.3). For example, the cache hierarchy of CPUs determines the tiling size of the matrix dimensions to ensure the locality of input data, thereby improving memory access efficiency. For GPUs, the information of SMs determines the size of grid-dimension and block-dimension for the matrix computation to be allocated, thus enhancing the computational efficiency of the CUDA kernel. The optimization hints are described with nature language, and thus developers can add knowledge to enable more hardware characteristics and optimization techniques conveniently.

Hardware Factor Extraction. Hardware Factor Extraction takes users' prompt of tensor operator type and the target hardware name as input, and then efficiently retrieve information about the hardware factors of target hardware from the pre-collected set of manuals or those provided by the user (optional). As depicted in Figure 2, hardware factors of RISC-V C910 include the cache levels and size, the type and usage of vector computation instructions, and so on. Based on retrieved hardware factors, the subsequent process can generate targeted tensor operator implementation.

3.2 Tensor Operator Generation

This component generates hardware-primitive-level tensor operators with proper optimization techniques. Considering the different implementation challenges of optimization techniques, Vectorization, Pipeline require hardware primitives for optimized data movement and computation, while Tiling, Reordering and Layout involve task and data allocation strategies to match target hardware. Thus, Tensor Operator Generation consists of Sketch Generation and Kernel Generation with Hardware Primitives as shown in Figure 2.

Sketch Generation. Sketch Generation leverages LLMs to generate the main function (C for CPU, CUDA C++ for GPU) of tensor operators with task and data allocation optimization for better memory access efficiency. It reserves indivisible computation or data movement as kernel calls (e.g., *PACK* and *COMPUTE* for CPUs) to subsequent hardware-primitive-level generation. The prompt delineates basic sketch structures and optimization techniques, including Tiling, Reordering, and Layout. For CPUs, the basic structure is a simple three-level for-loop. Specifically for the C910 CPU, LLMs utilize the prompt in conjunction with hardware factors and optimization hints to generate the main function featuring multi-level for-loops. This implementation enables scheduling optimization tailored to the C910's characteristics while preserving PACK and COMPUTE kernels for subsequent


User Input: Generate a 1024x1024x1024 GEMM code on RISC-V C910.
Step 1: Hardware Architecture Comprehending
Hardware Intrinsic Optimization Hints
Optimization techniques
T
R
V
L
P
Hardware factors
ARM
(V)R
MH
INST
SMs
NVIDIA
Hardware manuals
LLM
Hardware Information Extraction
Hardware Factors:
MH: L1 cache: 32KB...
INST: vfmacc.vv...
VR: 32 128-bit Vector Regs...
...
Step 2: Tensor Operator Generation
Sketch Generation
Generate a C code sketch:
• Structure: For-loops
• Optimization: Tiling+Reordeing+Layout
# Hardware Factors
# Hardware Intrinsic Optimization Hints
LLM
C code sketch
Kernel Generation with Hardware Primitives
Generate Python scripts for an assembly kernel
• Structure: Index variables + L/C/S
• Optimization: Pipeline+Vectorization
# Hardware Factors
# Hardware Intrinsic Optimization Hints
• Few-shot Example
# C code sketch
Assembly kernel code
C910
NVIDIA
...
Step 3: Auto-Tuning
S0:
# C code sketch
# Assembly kernel code
S0
St
1. Selection
BN+32
BK+32
St+1
2. Expansion
C910
...
3. Simulation
4. Backpropagation

```
BM = 64;
...
for (int i=0;i<M; i+=BM) {
  PACK(A)
  COMPUTE(A,B)
}
```

```
# Python scripts
for c in range (0 , nr ) :
    f.write(f"vle32.v C0{c}, (C0{c} _ptr )\n")
...
for r in range (0, mr // vl) :
    f.write(f"vfmacc.vv C{r}{c}, ft{c}, A{r}\n")
....
```

```
PACK
...
vle32.v      v0,    (t1)
...
COMPUTE
...
vfmacc.vv v4, v2, v0...
...
```

Figure 2: **QiMeng-TensorOp overview.** The proposed framework takes a user's one-sentence description as input and generates high-performance tensor operators using hardware primitives via three automatic steps. Step 1 activates LLMs' comprehension of tensor operator optimization and extracts target hardware factors for subsequent generation. In Step 2, sketches and kernels are generated to form tensor operators. Sketch generation uses optimizations such as tiling, and hardware-primitive-level kernels like *PACK* and *COMPUTE* employ instruction-level optimization. In Step 3, Auto-Tuning uncovers subtle optimization opportunities and further enhances performance. (A detailed showcase please refer to Appendix C.)

hardware-primitive-level optimization. For GPUs, LLMs produce CUDA C++ code that defines grid and block dimensions to efficiently allocate computational resources and execute CUDA kernels.

Kernel Generation with Hardware Primitives. LLMs perform well in generating Python code, but they face challenges in generate functionally correct codes with hardware primitives (e.g., CPU assembly instructions or GPU PTX(Parallel Thread Execution)). Thus, we prompt LLMs to generate Python scripts as bridge for kernels. The prompt comprises sketch code, the description of kernel structure and optimizations, few-shot examples, along with optimization-related hints and target hardware information acquired in Step 1. For instance, when generating a computation kernel with Pipeline optimization for CPUs, the structure section delineates vector register definitions and the layout of Load, Compute, and Store instruction blocks. Along with the target C910 CPU information (the type and usage of assembly INST, number and width of (V)R, etc.) in prompt, LLMs write a Python script to print the assembly-level kernel implementation. Conversely, for GPUs, LLMs write CuTe code to implement CUDA kernels with PTX-level optimization, aiming to fully exploit the Tensor Core's peak performance. To improve sketch parameter initialization for the subsequent Auto-Tuning step and the correctness of assembly kernels, they are jointly compiled and tested on the target hardware to obtain feedback for refining sketches and Python scripts of kernel generation.

3.3 Auto-Tuning

Auto-Tuning aims to optimize the sketch parameters and instruction orders to uncover subtle optimization opportunities that are even hard to be identified by human experts. It integrates MCTS with LLMs to tackle the shortage of semantic priors for individual tuning actions and leverages search history to guide exploration for efficiency and performance.

The MCTS nodes represent the current implementation of kernels and sketches. The search space encompasses sketch parameters and the instruction order within kernels. Consequently, there are two types of actions: 1) adjusting parameters in the sketch, such as matrix tiling sizes, through addition or subtraction; 2) reordering memory access or computation independent within a single instruction block.

As shown in Figure 2, the process iterates 4 steps: 1). **Selection** uses the UCB algorithm to identify a node for expansion; 2). **Expansion** dynamically grows by utilizing LLMs based on search history ; 3). **Simulation** tests the tensor operator of the newly expanded node for initial performance; 4).

Backpropagation updates node values and visit counts along the path. The pseudo-code is provided in the Appendix B.

The LLMs-driven expansion comprises **path history-based tuning action selection** and **global history-based tuning space generation**. When reaching a node, take the history tuning actions and performance feedback along the path as context. LLMs are prompted to output the most valuable tuning action for expansion. For instance, LLMs tend to select tuning parameters that have a greater impact on performance in the path history, such as the tiling size of BN. After expansion, LLMs take the tuning constraint description and the performance feedback of all expanded actions in the tree as context and output legal and potential candidate tuning actions for the new node. For example, if most nodes achieve stable performance improvements when the tiling size granularity increases by 32, LLMs tend to generate a similar granularity action space, $\{\mathrm{BN}+32, \mathrm{BM}+32, \mathrm{BK}+32, ...\}$. In summary, fine-tuning is challenging due to the lack of expert priors. However, QiMeng-TensorOp enables LLMs to utilize history trajectories for automatic reasoning and dynamic adjustment of search, thereby effectively enhancing efficiency and performance.

4 Evaluation

4.1 Experiment Setup

To validate the performance of the tensor operators generated by QiMeng-TensorOp and assess the method's generality and efficiency, we conduct comprehensive evaluations across three different hardware platforms, four distinct LLMs, and two representative tensor operators with varied dimensions. (Due to page limits, more results are shown in Appendix A.)

Hardware Platforms. QiMeng-TensorOp is tested on various hardware, including different CPUs with diverse architectures and capabilities (C906, C908, C910[Chen *et al.*, 2020a] and K1 of RISC-V, A76 and A72 of ARM, and NVIDIA GPU RTX4060 with CUDA Core and A100 with Tensor Core).

LLMs. The overall performance of QiMeng-TensorOp is validated with two SOTA LLMs: GPT-4o[OpenAI, 2025] (proprietary) and DeepSeek-V3 (DS-V3 for short) [DeepSeek-AI *et al.*, 2024] (open-source). Two additional LLMs including Claude 3.5 Sonnet-20241022[Claude3.5, 2024] and Llama-3.1-405B[Llama3.1, 2024], are used in ablation experiments.

Benchmarks. Experiments are conducted on representative tensor operators (GEMM and Conv). Typical regular and irregular GEMM dimensions in Llama7b[Touvron *et al.*, 2023] and Llama3 70b[Grattafiori *et al.*, 2024] are included. Typical Conv dimensions in classical CNNs, such as ResNet-50[He *et al.*, 2015], VGG-16[Simonyan and Zisserman, 2015] and U-Net[Ronneberger *et al.*, 2015], are included.

Comparison Baselines. QiMeng-TensorOp is compared with vanilla prompt and CoT prompt [Wei *et al.*, 2022]. Besides, manually optimized libraries (OpenBLAS[Xianyi *et al.*, 2012] for RISC-V CPUs, ACL / OpenBLAS for ARM CPUs, cuBLAS[NVIDIA, 2023] / cuDNN for NVIDIA GPUs), and commonly used auto-compilers (TVM [Chen *et al.*, 2018a] for ARM CPUs and NVIDIA GPUs [1]) are also compared.

[1]TVM lacks support for RISC-V CPUs.

4.2 Overall Performance

Table 1 show the performance of the GEMM operator generated by QiMeng-TensorOp on various hardware platforms. Due to page limits, the results of the convolution operator are shown in Appendix A. The performance is compared with vanilla prompt, manually optimized libraries, and auto-compiler. The results clearly show that QiMeng-TensorOp enables LLMs to generate high-performance assembly-level tensor operators across diverse platforms.

QiMeng-TensorOp outperforms LLMs with vanilla prompt by several orders of magnitude (up to $1291\times$) on GPT-4o and DeepSeek-V3 across various hardware platforms. On CPUs, QiMeng-TensorOp surpasses most manually optimized libraries. For instance, on the RISC-V CPUs, QiMeng-TensorOp achieves up to $2.51\times$ performance enhancement over OpenBLAS, while on the ARM CPUs, it achieves at most $1.21\times$ the performance of ACL. On GPUs, our method significantly outperforms TVM (up to $1.38\times$ for GEMM and $2.43\times$ for Conv), and achieves comparable or higher performance of cuBLAS and cuDNN in most dimensions (up to $1.24\times$ for GEMM and $1.61\times$ for Conv). We will present the observations of the experiments in the following aspects: LLMs, hardware platforms, and operator dimensions.

The Impact of LLMs: QiMeng-TensorOp consistently yields high performance, whereas the vanilla prompt performs poorly. QiMeng-TensorOp activates LLMs to automatically generate sketch and hardware-primitive-level kernels with high performance across LLMs. For example, on C910 CPU, the performance disparity of QiMeng-TensorOp using GPT-4o and DeepSeek-V3 is within 10%. In contrast, the vanilla prompt can only enable LLMs to generate tensor operators with high-level language (C code), but cannot manipulate hardware resources, thus yielding much lower performance. Moreover, LLMs of vanilla prompt have obvious performance disparities. For instance, on C910 CPU, the tensor operator generated by GPT-4o incorporates tiling optimization, while DeepSeek-V3 does not, resulting in a $3\times$ performance disparity.

The Impact of Hardware Platforms: QiMeng-TensorOp achieves a more significant performance boost on RISC-V CPUs than on ARM CPUs and GPUs. As an open-source instruction set architecture, RISC-V CPUs feature a highly flexible and diverse set of instructions and architectures. Thus, it poses significant challenges for manually optimized libraries. Besides, the official version of TVM even lacks native support for RISC-V CPUs. For instance, on C908, C910, and K1 CPUs, we obtain performance enhancements of $1.10\times$, $2.32\times$, and $2.51\times$ respectively in GEMM operation compared with OpenBLAS. As commercial processors, ARM and Nvidia hardware vendors have employed human experts to manually optimize the BLAS libraries. Our approach still yields competitive performance compared ACL on ARM CPUs ($1.02\times$ to $1.21\times$), and cuBLAS on NVIDIA GPUs ($0.98\times$ to $1.24\times$). Compared with manually optimized libraries, our approach can rapidly adapt to diverse architectures, achieving high performance across all platforms.

The Impact of Operator Dimensions: QiMeng-TensorOp outperforms auto-compilers and manually optimized libraries, especially on GEMM with larger or

Hardware	Method	$(m = k = n)$				(m, k, n)			
		512	1024	2048	4096	(512,4096,4096)	(768,4096,4096)	(1024,4096,4096)	(2048,4096,4096)
C908 (RISC-V)	GPT-4o	0.04	0.02	0.02	0.02	0.02	0.03	0.02	0.02
	+Ours	**8.70(↑ 218×)**	**9.16(↑ 458×)**	8.78(↑ 439×)	**9.23(↑ 462×)**	8.33(↑ 417×)	7.83(↑ 261×)	**9.40(↑ 470×)**	**9.45(↑ 473×)**
	DS-V3	0.03	0.01	0.01	0.01	0.03	0.03	0.03	0.02
	+Ours	7.97(↑ 266×)	7.5(↑ 750×)	8.62(↑ 862×)	8.97(↑ 897×)	**8.57(↑ 286×)**	**9.83(↑ 328×)**	8.35(↑ 278×)	9.25(↑ 463×)
	OBLAS	7.98	8.34	**8.85**	9.00	8.28	8.70	8.51	9.00
C910 (RISC-V)	GPT-4o	0.18	0.14	0.1	0.09	0.15	0.16	0.16	0.16
	+Ours	**11.21(↑ 62×)**	11.21(↑ 80×)	**10.94(↑ 109×)**	9.21(↑ 102×)	**11.48(↑ 77×)**	**11.74(↑ 73×)**	11.56(↑ 72×)	**11.05(↑ 69×)**
	DS-V3	0.09	0.05	0.03	0.03	0.81	0.82	0.82	0.82
	+Ours	10.58(↑ 118×)	**11.48(↑ 230×)**	10.84(↑ 361×)	**9.47(↑ 316×)**	11.36(↑ 14×)	11.67(↑ 14×)	**11.68(↑ 14×)**	10.89(↑ 13×)
	OBLAS	5.91	5.85	4.9	4.88	4.57	4.90	5.37	5.41
K1 (RISC-V)	GPT-4o	0.32	0.28	0.28	0.19	0.50	0.52	0.48	0.47
	+Ours	9.97(↑ 31×)	9.18(↑ 33×)	9.53(↑ 34×)	9.9(↑ 52×)	**11.43(↑ 23×)**	10.74(↑ 21×)	**10.78(↑ 22×)**	**10.7(↑ 23×)**
	DS-V3	0.36	0.33	0.31	0.23	0.45	0.47	0.46	0.45
	+Ours	**10.34(↑ 29×)**	**9.74(↑ 30×)**	**10.29(↑ 33×)**	**11.74(↑ 51×)**	10.48(↑ 23×)	**10.87(↑ 23×)**	10.14(↑ 22×)	10.41(↑ 23×)
	OBLAS	4.12	4.19	4.46	4.76	4.57	4.90	5.37	5.41
A76 (ARM)	GPT-4o	0.35	0.33	0.27	0.17	0.34	0.39	0.32	0.27
	+Ours	**34.46(↑ 98×)**	34.81(↑ 105×)	**36.77(↑ 136×)**	**37.31(↑ 219×)**	**33.84(↑ 100×)**	**33.77(↑ 87×)**	**35.72(↑ 112×)**	35.32(↑ 131×)
	DS-V3	0.22	0.04	0.04	0.04	0.29	0.29	0.29	0.28
	+Ours	33.91(↑ 154×)	**35.70(↑ 893×)**	36.05(↑ 901×)	36.67(↑ 917×)	32.72(↑ 113×)	33.48(↑ 115×)	35.66(↑ 123×)	**36.43(↑ 130×)**
	TVM	33.79	33.57	32.99	Failed	29.56	28.92	27.17	Failed
	OBLAS	27.97	31.25	33.48	34.27	30.01	31.25	32.37	33.95
	ACL	34.08	32.44	32.43	30.82	32.20	31.86	31.64	31.10

Hardware	Method	$(m = k = n)$				(m, k, n)			
		2048	4096	8192	16384	(16384,8192,1280)	(16384,1024,8192)	(16384,8192,7168)	(16384,3584,8192)
RTX 4060 (with CUDA core)	GPT-4o	1.2	1.1	1.1	1.1	1.12	1.18	1.00	1.10
	+Ours	7.84(↑ 7×)	**8.08(↑ 7×)**	**7.84(↑ 7×)**	**7.8(↑ 7×)**	8.14(↑ 7×)	7.63(↑ 6×)	7.93(↑ 8×)	**8.19(↑ 7×)**
	DS-V3	1.21	1.08	1.09	1.1	1.13	1.20	1.00	1.07
	+Ours	**8.1(↑ 7×)**	7.99(↑ 7×)	7.13(↑ 7×)	7.75(↑ 7×)	**8.28(↑ 7×)**	**7.71(↑ 6×)**	**8(↑ 8×)**	7.87(↑ 7×)
	TVM	7.87	6.55	6.93	Failed	7.63	6.7	6.25	7.1
	cuBLAS	7.37	7.34	7.23	7.19	7.06	6.21	7.21	7.24
A100 (with tensor core)	GPT-4o	0.81	0.86	0.88	0.74	0.75	0.85	0.80	0.74
	+Ours	183.73(↑ 227×)	260.59(↑ 303×)	289.13(↑ 329×)	293.44(↑ 397×)	**278.44(↑ 371×)**	237.25(↑ 279×)	283.87(↑ 355×)	278.04(↑ 376×)
	DS-V3	14.13	17.74	17.31	18.76	18.98	18.85	18.88	18.96
	+Ours	183.19(↑ 13×)	**262.05(↑ 15×)**	290.86(↑ 17×)	292.36(↑ 16×)	274.88(↑ 14×)	**238.26(↑ 13×)**	283.66(↑ 15×)	**278.48(↑ 15×)**
	TVM	159.66	195.59	212.79	212.48	220.33	199.8	221.02	217.76
	cuBLAS	**185.96**	246.1	**292.2**	**298.44**	268.58	237.19	**285.18**	253.72

Table 1: GEMM performance comparison on various hardware, LLMs, and matrix dimensions. The first line of each LLM indicates using the vanilla prompt. Performance is measured with GFLOPS and TFLOPS for CPUs and GPUs, respectively. (OBLAS means OpenBLAS, and Failed signifies that the target platform lacks adequate memory for TVM compiling.)

irregular dimensions. GEMM of larger or irregular dimensions would pose challenges of elaborately manipulating computation and memory resources, otherwise it may lead to inefficient memory access and lower performance, as we see in the manually optimized library of ACL and auto-compiler of TVM. In contrast, QiMeng-TensorOp performs more precise and efficient tensor operator generation and optimization at the hardware-primitive-level to fully leverage hardware characteristics, thereby achieving higher performance. For instance, when the matrix dimension is $1024 \times 4096 \times 4096$ on A76 CPU, our method performance is $1.31\times$ and $1.29\times$ better than TVM and ACL, respectively.

4.3 Ablation Study

To analyze the effectiveness and efficiency of QiMeng-TensorOp components, we conduct such ablation studies.

Ablation of Prompts: With just a one-line user prompt, QiMeng-TensorOp demonstrates exceptional performance, while the enhancement from CoT is slight. As shown in Table 2, our approach ranges from $26.49\times$ to $47.00\times$ compared with vanilla prompt, while the CoT prompt ranges from $1.29\times$ to $2.09\times$. The results indicate that QiMeng-TensorOp is applicable to different LLMs (even to slightly weaker LLMs) and effectively activates LLMs to generate high-performance tensor operator, showing exceptional robustness and scalability.

	512	1024	2048
GPT-4o	0.32	0.28	0.28
+CoT	0.67(↑ 2.09×)	0.45(↑ 1.61×)	0.44(↑ 1.57×)
+QiMeng-TensorOp	**9.97(↑31.16×)**	**9.18(↑32.79×)**	**9.53(↑34.04×)**
Claude 3.5 Sonnet	0.37	0.25	0.24
+CoT	0.49(↑ 1.32×)	0.38(↑ 1.52×)	0.32(↑ 1.33×)
+QiMeng-TensorOp	**9.80 (↑ 26.49×)**	**7.87 (↑ 31.48×)**	**8.95 (↑ 37.29×)**
DeepSeek-V3	0.36	0.33	0.31
+CoT	0.56(↑ 1.56×)	0.53(↑ 1.61×)	0.40 (↑ 1.29×)
+QiMeng-TensorOp	**10.34 (↑ 28.72×)**	**9.74 (↑ 29.52×)**	**10.29 (↑ 33.19×)**
Llama-3.1-405B	0.31	0.26	0.23
+CoT	0.49(↑ 1.58×)	0.35(↑ 1.35×)	0.33 (↑ 1.43×)
+QiMeng-TensorOp	**9.72 (↑ 31.35×)**	**9.57 (↑ 36.81×)**	**10.81 (↑ 47.00×)**

Table 2: GEMM performance (GFLOPS) comparison of vanilla prompt, CoT, and QiMeng-TensorOp on K1 CPU.

Ablation of QiMeng-TensorOp Components: Figure 3(a) compares the GEMM performance on K1 CPU among five conditions, including QiMeng-TensorOp with both Tensor Operator Generation (TOG) and Auto-Tuning (AT), QiMeng-TensorOp with only TOG, a GPT-4o generated C code with AT, and OpenBLAS with and without AT. **(1) Tensor Operator Generation effectively raises performance upper bound.** QiMeng-TensorOp (blue curve) obviously exceeds LLM Code + AT (green curve) and OpenBLAS Code + AT (red curve). As LLMs can only generate C code, and so does the general implementation in OpenBLAS, this indicates that the assembly kernels generated by TOG can better exploit the computing capability of hardware compared with C code. **(2) Auto-Tuning is vital for superior performance of tensor operates.** Auto-Tuning improves the performance of QiMeng-TensorOp with only TOG (yellow curve), as well as OpenBLAS code (black curve), respectively. This indicates that Auto-Tuning can effectively identify subtle optimization opportunities to achieve superior performance. Moreover, it has a certain degree of generalization ability for tensor operators generated by different methods.

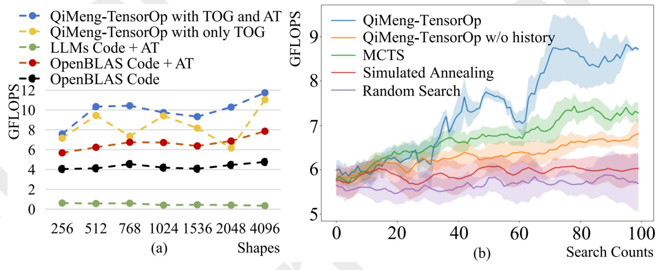


Figure 3: (a) Ablation results of QiMeng-TensorOp components for GEMM on K1 CPU. (b) Performance comparison of tuning methods on GEMM of size 1024 on K1 CPU.

Ablation of Search Methods: The historical node expansion records is crucial for Auto-Tuning in QiMeng-TensorOp. As shown in Figure 3(b), when search histories are available, QiMeng-TensorOp (blue curve) can leverage them to do dynamic expansion and search, thus exceeding naive MCTS (green curve) in both efficiency and final performance. In contrast, QiMeng-TensorOp w/o history (orange curve) degrades to worse than MCTS with fixed expansion.

C910 CPU	Time	GFLOPS
Senior Coder	3 days	3.08
TVM	Failed	Failed
ours	20 mins	11.48
A76 CPU	**Time**	**GFLOPS**
Senior Coder	3 days	1.35
TVM	2 hours	33.57
ours	15 mins	35.70
A100 GPU	**Time**	**TFLOPS**
Senior Coder	5 days	45.17
TVM	1.5 hours	77.68
ours	12 mins	105.49

Table 3: Development costs for Senior Coder, TVM and QiMeng-TensorOp across different platforms.

Ablation of Development Cost: (1) QiMeng-TensorOp reduces manually development costs. On the three platforms shown in Table 3, our method reduces development costs from several days to no more than twenty minutes compared to software engineers with 5 years of experience by up to 200× (on A100 GPU, 12 mins vs. 5 × 8 hours), while performance is improved by up to 26× (on A76 CPU). **(2) QiMeng-TensorOp achieves higher performance than TVM with fewer search counts.** As shown in Figure 4, increasing the number of TVM searches improves code performance, but it has an upper limit.

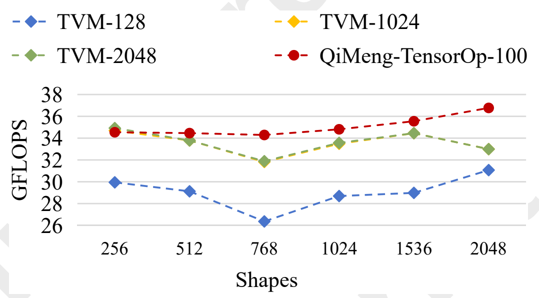


Figure 4: GEMM performance comparison between QiMeng-TensorOp and TVM with different search counts on A76 CPU. TVM-1024 and TVM-2048 overlap as TVM-1024 reaches the upper limit.

5 Conclusion

In this paper, we present QiMeng-TensorOp to automatically generate high-performance tensor operators at the hardware primitive level across various platforms, requiring only one user-provided sentence to describe the target operator and hardware. QiMeng-TensorOp leverages hardware hints and workflow to aid LLMs' comprehension of hardware and optimization, enabling automatic information extraction from manuals for tensor operator generation. It incorporates an LLM-assisted MCTS algorithm, enhancing the tuning efficiency and performance. Extensive evaluations on diverse platforms and tensor operators show significant performance and development cost benefits. QiMeng-TensorOp has the potential for continuously evolving hardware architectures. In the future, we will extend the proposed framework to more operators .

Acknowledgments

This work is partially supported by the NSF of China (under Grant 92364202), and Major Program of ISCAS (Grant No. ISCAS-ZD-202402).

References

[Anam *et al.*, 2013] Ashraful Anam, Paul Whatmough, and Yiannis Andreopoulos. 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*, pages 21–30. IEEE, 2013.

[Athiwaratkun *et al.*, 2022] Ben Athiwaratkun, Sanjay Gouda, and Zijian Wang et al. Multi-lingual Evaluation of Code Generation Models. *ArXiv abs/2210.14868*, 2022.

[Austin *et al.*, 2021] Jacob Austin, Augustus Odena, and Maxwell Nyen et al. Program Synthesis with Large Language Models. *ArXiv*, abs/2108.07732, 2021.

[Bi *et al.*, 2023] Jun Bi, Qi Guo, and Xiaqing Li et al. 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*, pages 314–328, 2023.

[Castelló *et al.*, 2024] Adrián Castelló, Julian Bellavita, and Grace Dinh et al. Tackling the Matrix Multiplication Micro-Kernel Generation with Exo. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 182–193. IEEE, 2024.

[Chen *et al.*, 2018a] Tianqi Chen, Thierry Moreau, and Ziheng Jiang et al. TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[Chen *et al.*, 2018b] Tianqi Chen, Lianmin Zheng, and Eddie Yan et al. Learning to Optimize Tensor Programs. In *Neural Information Processing Systems*, 2018.

[Chen *et al.*, 2020a] Chen Chen, Xiaoyan Xiang, and Chang Liu et al. Xuantie-910: A Commercial Multi-Core 12-Stage Pipeline Out-of-Order 64-bit High Performance RISC-V Processor with Vector Extension. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 52–64. IEEE, 2020.

[Chen *et al.*, 2020b] Yi-Ru Chen, Hui-Hsin Liao, and Chia-Hsuan Chang et al. Experiments and optimizations for TVM on RISC-V Architectures with P Extension. In *2020 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, pages 1–4, 2020.

[Choquette *et al.*, 2021] Jack Choquette, Wishwesh Gandhi, and Olivier et al. Giroux. Nvidia A100 Tensor Core GPU: Performance and innovation. *IEEE Micro*, 41(2):29–35, 2021.

[Claude3.5, 2024] Claude3.5. Claude 3.5 Sonnet. https://www.anthropic.com/news/claude-3-5-sonnet, 2024. Accessed: 2024-12-27.

[Dally *et al.*, 2021] William J Dally, Stephen W Keckler, and David B Kirk. Evolution of the graphics processing unit (GPU). *IEEE Micro*, 41(6):42–51, 2021.

[DeepSeek-AI *et al.*, 2024] DeepSeek-AI, Aixin Liu, and Bing Xue et al. DeepSeek-V3 Technical Report. 2024.

[Faingnaert *et al.*, 2021] Thomas Faingnaert, Tim Besard, and Bjorn De Sutter. Flexible Performant GEMM Kernels on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 33(9):2230–2248, 2021.

[Feng *et al.*, 2023] Siyuan Feng, Bohan Hou, and Hongyi Jin et al. Tensorir: An abstraction for automatic tensorized program optimization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 804–817, 2023.

[Grattafiori *et al.*, 2024] Aaron Grattafiori, Abhima Dubey, and Abhinav Jauhri et al. The Llama 3 Herd of Models. *ArXiv abs/2407.21783*, 2024.

[Guillermo *et al.*, 2024] Alae Guillermo, Martínez Héctor, and Castelló Adrián et al. Automatic generation of ARM NEON micro-kernels for matrix multiplication. *The Journal of Supercomputing*, pages 1–27, 2024.

[Gunasekar *et al.*, 2023] Suriya Gunasekar, Yi Zhang, and Jyoti Aneja et al. Textbooks Are All You Need. *ArXiv abs/2306.11644*, 2023.

[He *et al.*, 2015] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *ArXiv abs/1512.03385*, 2015.

[Hennessy and Patterson, 2019] John Hennessy and David Patterson. A new golden age for computer architecture. *Communications of the ACM*, 62(2):48–60, 2019.

[Igual *et al.*, 2023] Francisco Igual, Luis Piñuel, and Sandra Catalán et al. Automatic Generation of Micro-kernels for Performance Portability of Matrix Multiplication on RISC-V Vector Processors. In *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, pages 1523–1532, 2023.

[Jouppi *et al.*, 2017] Norman P Jouppi, Cliff Young, and Nishant Patil et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

[Katel *et al.*, 2021] Navdeep Katel, Vivek Khandelwal, and Uday Bondhugula. High performance GPU code generation for matrix-matrix multiplication using mlir: some early results. *arXiv preprint arXiv:2108.13191*, 2021.

[Kim *et al.*, 2024] Hyungyo Kim, Gaohan Ye, and Nachuan et al. Wang. Exploiting Intel® Advanced Matrix Extensions (AMX) for Large Language Model Inference. *IEEE Computer Architecture Letters*, 2024.

[Krainiuk *et al.*, 2021] Mariia Krainiuk, Mehdi Goli, and Vincent R Pascuzzi. oneAPI Open-Source Math Library

Interface. In *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 22–32. IEEE, 2021.

[Kurzak *et al.*, 2012] Jakub Kurzak, Stanimire Tomov, and Jack Dongarra. Autotuning GEMM kernels for the Fermi GPU. *IEEE Transactions on Parallel and Distributed Systems*, 23(11):2045–2057, 2012.

[Li *et al.*, 2023] Raymond Li, Loubna Ben, and Yangtian Zi et al. StarCoder: may the source be with you! *ArXiv*, abs/2305.06161, 2023.

[Liu *et al.*, 2016] Shaoli Liu, Zidong Du, and Jinhua Tao et al. Cambricon: An instruction set architecture for neural networks. *ACM SIGARCH Computer Architecture News*, 44(3):393–405, 2016.

[Liu *et al.*, 2022] Xiao-Yang Liu, Zeliang Zhang, and Zhiyuan Wang et al. High-Performance Tensor Learning Primitives Using GPU Tensor Cores. *IEEE Transactions on Computers*, 72(6):1733–1746, 2022.

[Llama3.1, 2024] Llama3.1. Introducing Llama 3.1: Our most capable models to date. https://ai.meta.com/blog/meta-llama-3-1/, 2024. Accessed: 2024-12-27.

[Lu *et al.*, 2022] Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung won Hwang, and Alexey Svyatkovskiy. ReACC: A Retrieval-Augmented Code Completion Framework. *ArXiv abs/2203.07722*, 2022.

[Markidis *et al.*, 2018] Stefano Markidis, Der Chien, and Steven Wei et al. NVIDIA Tensor Core Programmability, Performance & Precision. In *2018 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, pages 522–531. IEEE, 2018.

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

[OpenAI, 2025] OpenAI. GPT-4o. https://openai.com/index/hello-gpt-4o/, 2025. Accessed: 2024-12-27.

[Ragan *et al.*, 2013] Jonathan Ragan, Connelly Barnes, and Andrew Adams et al. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.

[Ronneberger *et al.*, 2015] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation, 2015.

[Rozière *et al.*, 2023] Baptiste Rozière, Jonas Gehring, and Fabian Gloeckle et al. Code Llama: Open Foundation Models for Code. *ArXiv*, abs/2308.12950, 2023.

[Shao *et al.*, 2022] Junru Shao, Xi Zhou, and Siyuan Feng et al. Tensor program optimization with probabilistic programs. *Advances in Neural Information Processing Systems*, 35:35783–35796, 2022.

[Simonyan and Zisserman, 2015] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *ArXiv abs/1409.1556*, 2015.

[Sze *et al.*, 2017] Vivienne Sze, Yu-Hsin Chen, and Tien-Ju Yang et al. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

[Tan *et al.*, 2011] Guangming Tan, Linchuan Li, and Sean Triechle et al. Fast implementation of DGEMM on Fermi GPU. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2011.

[Touvron *et al.*, 2023] Hugo Touvron, Thibaut Lavril, and Gautier Izacard et al. LLaMA: Open and Efficient Foundation Language Models. *ArXiv abs/2302.13971*, 2023.

[Wei *et al.*, 2022] Jason Wei, Xuezhi Wang, and Dale Schuurmans et al. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[Wu *et al.*, 2021] Kan Wu, Zhihan Guo, and Guanzhou Hu et al. The storage hierarchy is not a hierarchy: Optimizing caching on modern storage devices with orthus. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 307–323, 2021.

[Wu *et al.*, 2024] Du Wu, Jintao Meng, and Wenxi Zhu et al. autoGEMM: Pushing the Limits of Irregular Matrix Multiplication on Arm Architectures. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2024.

[Xianyi *et al.*, 2012] Zhang Xianyi, Wang Qian, and Zhang Yunquan. Model-driven level 3 BLAS performance optimization on Loongson 3A processor. In *2012 IEEE 18th international conference on parallel and distributed systems*, pages 684–691. IEEE, 2012.

[Xiao *et al.*, 2021] Qingcheng Xiao, Size Zheng, and Bingzhe Wu et al. Hasco: Towards agile hardware and software co-design for tensor computation. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture*, pages 1055–1068. IEEE, 2021.

[Zan *et al.*, 2022] Daoguang Zan, Bin Chen, and Fengji Zhang et al. Large Language Models Meet NL2Code: A Survey. In *Annual Meeting of the Association for Computational Linguistics*, 2022.

[Zhai *et al.*, 2023] Yi Zhai, Yu Zhang, and Shuo Liu et al. TLP: A Deep Learning-Based Cost Model for Tensor Program Tuning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 833–845, 2023.

[Zhai *et al.*, 2024] Yi Zhai, Sijia Yang, and Keyu Pan et al. Enabling Tensor Language Model to Assist in Generating High-Performance Tensor Programs for Deep Learning. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 289–305, 2024.

[Zheng *et al.*, 2020] Lianmin Zheng, Chengfan Jia, and Minmin Sun et al. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pages 863–879, 2020.