QiMeng-Attention: SOTA Attention Operator is generated by SOTA Attention Algorithm

Qirui Zhou[1,3], Shaohui Peng[2], Weiqiang Xiong[2,3], Haixin Chen[1,3], Yuanbo Wen[1], Haochen Li[2], Ling Li[2,3]*, Qi Guo [1], Yongwei Zhao[1], Ke Gao[2], Ruizhi Chen[2], Yanjun Wu[2], Chen Zhao[2], Yunji Chen[1,3]

[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

zhouqirui22s@ict.ac.cn,{pengshaohui,liling}@iscas.ac.cn

Abstract

The attention operator remains a critical performance bottleneck in large language models (LLMs), particularly for long-context scenarios. While FlashAttention is the most widely used and effective GPU-aware acceleration algorithm, it must require time-consuming and hardware-specific manual implementation, limiting adaptability across GPU architectures. Existing LLMs have shown a lot of promise in code generation tasks, but struggle to generate high-performance attention code. The key challenge is it cannot comprehend the complex data flow and computation process of the attention operator and utilize low-level primitive to exploit GPU performance.

To address the above challenge, we propose an LLM-friendly Thinking Language (LLM-TL) to help LLMs decouple the generation of high-level optimization logic and low-level implementation on GPU, and enhance LLMs' understanding of attention operator. Along with a 2-stage reasoning workflow, TL-Code generation and translation, the LLMs can automatically generate FlashAttention implementation on diverse GPUs, establishing a self-optimizing paradigm for generating high-performance attention operators in attention-centric algorithms. Verified on A100, RTX8000, and T4 GPUs, the performance of our methods significantly outshines that of vanilla LLMs, achieving a speed-up of up to 35.16×. Besides, our method not only surpasses human-optimized libraries (cuDNN and official library) in most scenarios but also extends support to unsupported hardware and data types, reducing development time from months to minutes compared with human experts.

1 Introduction

The attention mechanism is the cornerstone of modern Large Language Models (LLMs). Its time and memory complexity grows quadratically in sequence length, and attention has emerged as the bottleneck for the runtime and memory resource requirements for LLMs, particularly in long-context scenarios (Dao et al., 2022). Consequently, numerous acceleration algorithms have been proposed to accelerate attention operators, with FlashAttention being the most effective and widely used (Dao et al., 2022; Dao, 2024; Hong et al., 2024). With the rise of cost-efficient, high-performance LLMs like DeepSeek (DeepSeek-AI et al., 2024a,b, 2025), deploying LLMs across diverse GPU specifications (including legacy generations) has become increasingly prevalent.

However, due to the complexity of attention operators and disparities in the architectures and instruction sets across different GPUs, the efficient implementation and migration of FlashAttention algorithms becomes a pressing challenge. Exist-

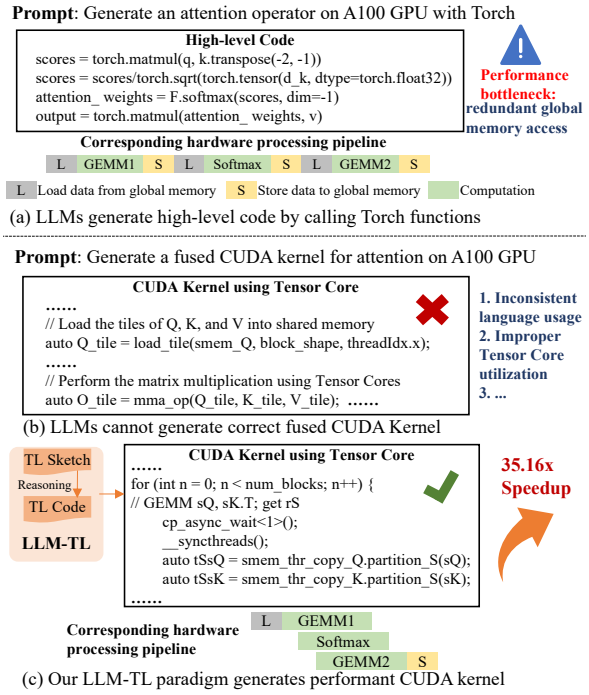

Figure 1: Our LLM-TL enables LLMs to generate high-performance Tensor Core kernel for attention.

*Corresponding Author.

Findings of the Association for Computational Linguistics: ACL 2025, pages 8491–8505
July 27 - August 1, 2025

ing methods for implementing high-performance attention operators can be divided into two categories: i.e., manual implementation and deep learning compilers. Human experts optimize attention operators through fused computation based on their expertise in GPU architecture and algorithm knowledge, thereby effectively enhancing computation efficiency. Consequently, these handcrafted implementations are time-consuming and non-portable, failing to support some common GPUs and datatypes (e.g., RTX8000 GPU and FP8 datatype). In addition, existing mainstream deep learning compilers (e.g., TVM (Chen et al., 2018), Ansor (Zheng et al., 2020)) fail to automatically implement FlashAttention due to their inability to perform optimizations involving fused sequences of multiple complex operations (e.g. GEMM). Overall, current methods struggle to automatically analyze the attention operator characteristics and GPU architecture to efficiently generate high-performance FlashAttention implementation on different GPU platforms, which limits the adaptability of LLMs.

Motivated by strong code-generation abilities of LLMs (Bairi et al., 2024; Zhong et al., 2024; Holt et al., 2024; Li et al., 2024; Wang et al., 2023; Zhou et al., 2025; Zhang et al., 2025), it is a promising direction to tackle these issues by utilizing LLMs automatically implement FlashAttention on GPUs. Although LLMs can create attention implementations by high-level APIs like PyTorch (Paszke and et al., 2019), they have trouble generating high-performance FlashAttention with low-level primitives (e.g., CUDA or CuTe (NVIDIA/CUTLASS, 2023)) with crucial optimization techniques like data blocking and fused computation. This significantly hinders the reduction of memory access overheads and the enhancement of computational efficiency, as shown in Figure 1(a).The primary challenges preventing LLMs from generating high-performance FlashAttention implementations in one pass are twofold: (1) LLMs cannot comprehend the complex data flow and multi-step computation of attention, thus failing to generate effective optimization logic; (2) LLMs are unable to produce optimized implementations based on low-level primitives to fully exploit GPU characteristics.

Inspired by human experts' decoupling of optimization and implementation, we propose an LLM-friendly Thinking Language (LLM-TL) to address the above challenge by helping LLMs decouple the generation of abstract optimization logic and low-level code implementation on GPU. Specifically, LLM-TL encompasses two types of statements and their requisite parameters: memory access and computation, enabling LLMs to comprehend the data flow and computation processes of the attention operator on GPU architecture from the vantage point of high-level abstract semantics, as shown in Figure 1(c). Based on LLM-TL, we further propose an automated workflow, leveraging the reasoning capabilities of LLMs to implement the high-performance FlashAttention operator on GPUs with low-level primitives.Formally, the workflow includes: 1) TL Code generation: LLMs use LLM-TL statements to create sketch code to represent the execution flow on GPU, then reasoning and filling the parameters needed for each statement; 2) TL Code translation: LLMs convert the TL-code into low-level CUDA code based on the target GPU architecture and instruction set. Through LLM-TL and the workflow, LLMs can efficiently and automatically implement the high-performance FlashAttention on different GPUs.

The contributions of this paper are as follows:

(1) We propose LLM-TL, an LLM-friendly abstract language, enabling LLMs to deeply comprehend and optimize complex attention operators on GPUs, thereby establishing a self-optimizing paradigm for generating high-performance attention operators in attention-centric algorithms.

(2) We present an automated workflow that enables LLMs to leverage LLM-TL to implement high-performance FlashAttention on diverse GPUs, which extends the capabilities of LLMs.

(3) Verified on A100, RTX8000, and T4 GPUs, the performance of FlashAttention implementation generated by our methods significantly outshines that of vanilla large language models (LLMs), achieving a speed-up of up to $35.16\times$, and also outperforms with cuDNN or official implementation in most cases. Moreover, the development time also be reduced from months to minutes compared with human experts.

2 Preliminary

2.1 GPU Characteristics

The rapid development of AI is closely tied to the advancement of GPU's high parallel computing power. GPUs achieve high parallelism by stacking a large number of simple computation units. They use the SIMT (Single Instruction, Multiple

Threads) design, which allows for the execution of many threads running the same code. The introduction to the GPU architecture is illustrated in Figure 2.

2.1.1 Memory Hierarchy

The memory hierarchy of GPUs is divided into global memory, shared memory, and registers. Global memory is shared by all threads, shared memory is shared by threads within the same thread block, and each thread has its own private registers. Due to this hierarchical memory structure, explicit management of data movement between different memory levels is required during programming. Data stored in registers can be directly accessed for fast computations, such as General Matrix Multiplication (GEMM).

2.1.2 Computation Unit

To get better performance on GPU, we use Tensor Core (Markidis et al., 2018), which is a specialized processing unit optimized for matrix operations. Unlike standard CUDA Cores, Tensor Cores use the warp (a cluster of 32 threads) as the basic computing unit, enabling efficient register data utilization and coordination among threads within a warp. Due to the design characteristics of Tensor Cores, operators can more fully utilize register resources within threads, which provides better hardware support for operator fusion optimization.

The efficient utilization of Tensor Core typically requires direct manipulation of the CUDA PTX instruction set, a low-level programming approach closer to assembly language, involving numerous complex index calculations. To simplify such process, NVIDIA introduced CuTe (NVIDIA/CUTLASS, 2023), an advanced template librariy which encapsulate the underlying PTX instructions through a high-level abstraction layer and automate index calculations. Compared to PTX, CuTe provides a more user-friendly and efficient programming support for automatic generation.

Based on these characteristics of GPUs, we abstract the execution flow of operators into two basic types of operations: data movement statement (describing the transfer of data between different memory levels) and computation statement (describing computing operations), thereby clearly delineating the execution process of operators on GPUs.

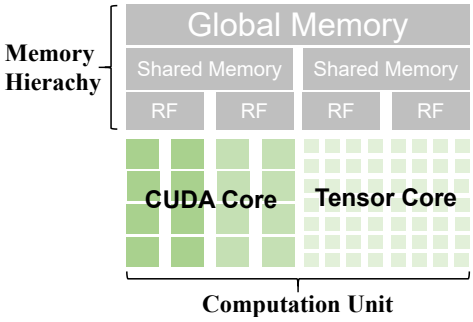

Figure 2: Demonstration of GPU architecture.

2.2 Attention Operators and acceleration

The Attention mechanism computes relevance weights among input elements to focus on key information for effective feature extraction and integration. The input consists of sequences Q, K, V of dimension d and number of tokens N. The attention mechanism computes the output using the formula $\text{Attention}(Q, K, V) = \text{Softmax}(\text{Mask}(\frac{QK^T}{\sqrt{d}}))V$. In addition to the early Multi-Head Attention (MHA) (Vaswani et al., 2017), recent advancements have introduced Multi-Query Attention (MQA) (Shazeer, 2019) and Group-Query Attention (GQA) (Ainslie et al., 2023). Furthermore, DeepSeek recently proposed a new attention variant called Multi-Head Latent Attention (MLA) (DeepSeek-AI et al., 2024a). These attention variants use fewer key/value heads or employ low-rank joint compression for keys and values. As a result, they significantly reduce the KV cache size, and are therefore widely adopted in contemporary LLMs.

To enhance the efficiency of attention operators, numerous optimization methods have emerged. FlashAttention (Dao et al., 2022; Dao, 2024) is a prominent example of optimizing the attention operator on GPUs. However, the implementation of high-performance attention operators like FlashAttention is often highly complex, requiring extensive low-level optimizations, precise computational scheduling, and facing significant challenges in operator fusion. Additionally, the programming model for GPU hardware, particularly Tensor Cores, is inherently complex, demanding developers to have a deep understanding of its architectural characteristics. As a result, the development and application of attention operators still encounter substantial technical challenges.

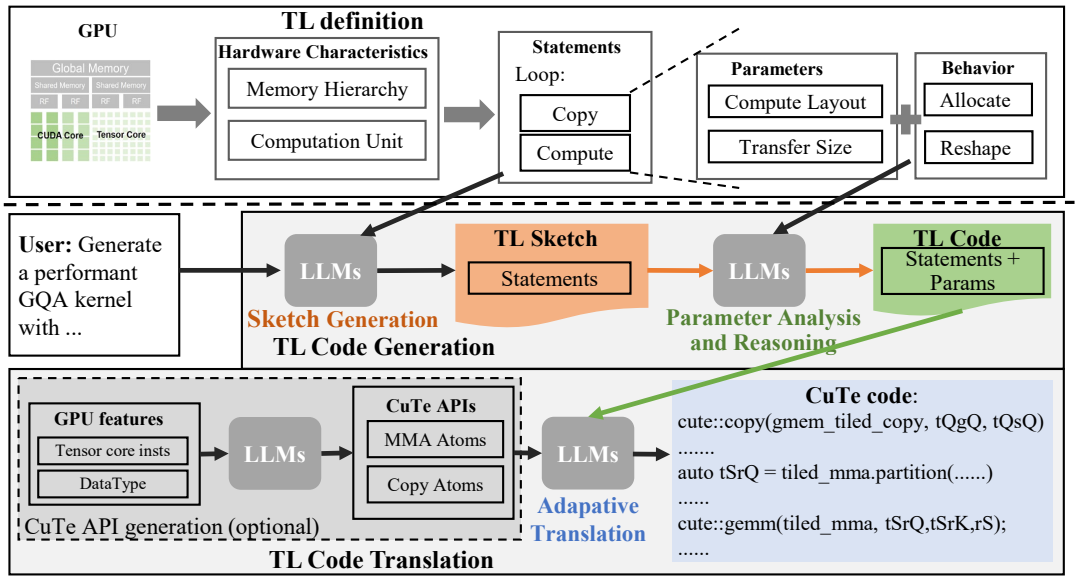

Figure 3: **LLM-TL overview.** We design the Thinking Language (TL) to help LLMs describe attention execution workflows and parameters on GPUs. Our approach consists of two stages, TL Code Generation and Translation. The TL Sketch, representing abstract semantic execution flow, is initially generated from user requirements. Subsequently, the LLM infers the parameter details of the statements within the Sketch. Finally, TL code will be translated to the CuTe implementation by LLMs based on GPU specification.

3 Method

LLM-TL solely incorporates statements that delineate the processing flow of attention and the essential parameters required, thereby enabling the reduction of hundreds of lines of low-level CUDA code to a mere dozen lines of TL code. This simplification empowers Large Language Models (LLMs) to comprehend and implement high-performance optimizations of attention operators on GPUs, encompassing fusion and computational tensorization. We introduce the workflow of TL code generation and adaptive translation from TL code to lower-level CuTe code in the following subsections, as shown in Figure 3.

3.1 LLM-friendly Thinking Language (LLM-TL) Definition

To assist LLMs in generating code more effectively, we abstract the execution process on GPUs into two types of statements: *Copy* for memory access and *Compute* for various types of computations. These statements are used to represent the execution flow and parameter details of attention operators on GPUs. By summarizing GPU characteristics into semantic-level descriptions and hiding unnecessary details, TL can be better leveraged through the generalization and semantic understanding capabilities of LLMs.

3.2 TL Code Generation

Specifically, the generation of TL Code is divided into two steps: first, describe the execution flow of attention into a TL Sketch containing computation and memory access statements, and then analysis and reason to supplement detailed statement parameters, ultimately producing a complete TL Code.

3.2.1 Sketch Generation

To facilitate the gradual understanding of GPU hardware architecture and attention operator characteristics by LLMs, we first drive LLMs to generate the execution flow of the attention operator into a semantically structured representation, TL Sketch, which consists of two fundamental statements: *Compute* and *Copy* hierarchies according to the algorithm's execution flow. Due to the various attention operator variants and GPUs, it is necessary to abstract high-level semantics of the attention execution on general GPU architecture to make the LLMs generate optimization logic. Thus, inspired by concepts from computer architecture, we design efficient statements, copy and compute, for two purposes: comprehensively capture the semantic-level optimization logic (multi-level memory data movement and GPU computation units utilization) and effectively hide low-level implementation details.

Specifically, for *Copy* statements, LLMs use

Copy to describe the movement of tensors between different memory hierarchies (global memory, shared memory, and registers). For example, the clause *Copy Q from global to shared* indicates loading Q from global memory into shared memory. For *Compute* statements, LLMs can represent necessary internal computations (such as GEMM, arithmetic operations, etc.) on the corresponding tensors. Based on TL statements, LLM can incorporate GPU hardware features to optimize attention execution at a semantical level, like representing the fusion optimization by continuously arranging multiple *Compute* at the same memory hierarchy without any *Copy*.

3.2.2 Parameter Analysis and Reasoning

TL Sketch can describe the execution flow of the attention operator on GPUs at a semantic level, but it lacks the details information needed to generate executable code. For *Copy* statement, it requires address allocation behavior and parameters like block dim to specify the transfer size; while for *compute*, it needs parameters to specify the computation layout and sometimes conduct reshape operations to connect adjacent computations to achieve fusion.

Based on the above summary, we leverage the reasoning capabilities of LLMs to supplement the parameter details required for *Copy* and *Compute* statements in the TL Sketch, such as block sizes and shape transformation information, to obtain the final TL-code. Taking the FlashAttention operator as an example, each thread block loads a fixed batch and head of *Q* with dimensions *(BM,HeadDim)*, where *BM* represents the block size of the query sequence length, and *HeadDim* denotes the dimensionality of a single head in multi-head attention (MHA). At this point, the *Copy* statement for Q is expanded to *Copy Q (BM, HeadDim) in coor [L = block_idx] from global to shared memory*, where *block_idx* denotes the block index in CUDA programme, adding the location and size information of Q in global memory. To fuse two consecutive GEMM operations at the register level, a reshape statement is required to perform shape inference for both GEMM operations. Then a reshape statement is introduced, specifically *reshape rS from mma_C to mma_A*[1], to facilitate the transformation and alignment of data structures between the matrix multiplication operations.

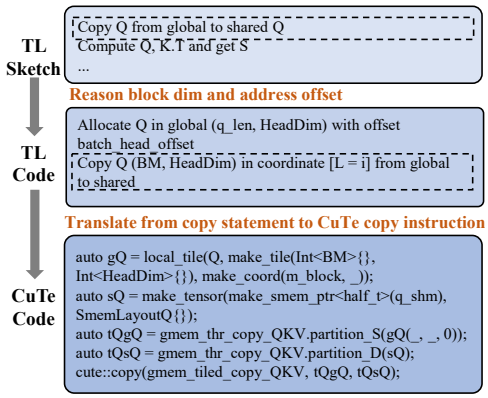

Figure 4: The content within the dashed box illustrates the generation process from a *Copy* statement in TL Sketch to CuTe Code.

3.3 TL Code Translation

3.3.1 CuTe Framework

We employ the CuTe framework to implement the final GPU code, aiming to simultaneously balance the high performance of GPU computing and the reliability of code generation by LLMs. The CuTe framework significantly enhances the semantic expressiveness of the code by providing high-level abstractions over fundamental Tensor Core operations (such as the matrix multiply-accumulate instruction *mma* and the matrix load instruction *ldmatrix*). In CuTe, assembly-level PTX *mma* instructions are encapsulated into corresponding classes, which include parameters such as computation scale and data types, while also reducing the need for index calculations. This strategy endows CuTe with a richer semantic hierarchy compared to native Tensor Core instruction programming and can fully leverage the strengths of LLMs in semantic understanding and logical reasoning. For some of the newer architectures, CuTe may not include the corresponding MMA APIs. However, due to their relatively fixed structure, we can quite easily prompt the LLM to generate the corresponding MMA through few-shot learning.

3.3.2 Adaptive Translation

Based on the TL Code, which contains comprehensive attention operator execution details, each statement can be accurately translated into the corresponding CuTe code.Thanks to the decoupling feature of our TL design, each statement can be fully and precisely translated into executable CuTe code, ensuring a smooth transition from high-level specifications to low-level implementations. Tak-

[1]Tensor Cores use *mma* (matrix multiply-accumulate) instructions to achieve , where matrices A, B, and C need to follow hardware-defined layouts. Here the *mma_A* and *mma_C* represent the corresponding layout of each tensor tile.

ing the TL Code as input, we provide the necessary execution information, such as CuTe MMA Atom and *Copy* Atom, for the specific hardware architecture in the prompt. By combining these two parts, we can achieve the translation of TL Code across multiple platforms. For instance, an allocate operation before *Copy* in the TL Code can be implemented using CuTe's APIs for tensor definitions across various memory hierarchies (such as global memory, shared memory, and registers), in conjunction with the parameter information provided in the TL Code. Similarly, a *Copy* statement can be realized through CuTe's definitions of source and destination memory (including caches or registers), along with the *cute::copy* API and other related code blocks. Figure 4 shows some details of how to generate the final CuTe code for a *Copy* statement.

4 Evaluation

4.1 Experiment Setup

To validate the performance of the attention operator implementation generated by LLMs through LLM-TL, we conduct comprehensive evaluation across different GPU platforms, distinct LLMs, and attention variants. Additional experiments are shown in Appendix A due to space limitations.

GPU platforms. LLM-TL is test on GPUs with different architectures, including NVIDIA Ampere (A100) and Turing (RTX8000, T4) architecture.

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-V3 (DeepSeek-AI et al., 2024b) and DeepSeek-R1 (DeepSeek-AI et al., 2025).

Attention variants. Experiments are conducted on attention mechanisms commonly used by current LLMs, including MHA (GPT-style models), GQA (Llama 3.1 (Llama3.1, 2024), Qwen2.5 (Yang et al., 2025)), MQA (Falcon (Almazrouei et al., 2023), StarCoder (Li et al., 2023)), and MLA (DeepSeek-V2, DeepSeek-V3).

Comparison Baselines. Implementation generated by our approach is compared with FlashAttention official library (v2.7.3 on A100 and v1.0.9 on RTX8000, as FlashAttention v2 is not available on Turing architecture), NVIDIA official deep learning library cuDNN (Chetlur et al., 2014), FlexAttention official library (Dong et al., 2024),

Benchmark setting. We followed the settings in the FlashAttention(Dao et al., 2022; Dao, 2024), the sequence length varies from 512, 1k, ..., 16k, and the batch size is adjusted such that the total number of tokens remains 16k. We set the hidden dimension to 2048, and the head dimension to be either 64 or 128 (i.e., 32 heads or 16 heads). The FLOPs are calculated as: $4 * \text{seqlen}^2 * \text{head dimension} * \text{number of heads}$. For MLA, we utilize the dimensions specified in DeepSeek-V3, which consist of an embedding dimension of 128 and a RoPE dimension of 64. We also test attention variants with and without causal masks.

4.2 Overall Performance

Table 1 demonstrates LLM-TL's performance across GPU architectures and common attention operators, while Table 2 specifically evaluates the novel MLA operator. The results demonstrate LLM-TL's capability to empower DeepSeek-V3 in generating high-performance implementations for diverse attention operators across GPU platforms.

Consistent superiority compared with vanilla LLM. LLM-TL achieves consistent superiority over vanilla LLM implementations across all tested scenarios (GPUs, operators, dimensions, and causal masking configurations), with peak speedups reaching $35.16\times$ in common attention operators and $9.41\times$ in MLA. Notably, vanilla LLM-generated implementations even trigger out-of-memory (OOM) errors at large sequence lengths. The results show our method can implement fused attention operators with less memory resource requirement and higher computation efficiency.

Competitive or superior performance compared with SOTA handcrafted libraries. LLM-TL, which requires substantially lower development effort, demonstrates superior performance on the novel MLA operator with $2.15\times$ speedup (175.9 vs 81.7 TFLOPS) against SOTA handcrafted library, cuDNN, while achieving comparable or better performance even on established attention operators (like MHA, GQA and MQA) well-supported by existing optimized libraries. Besides, our method maintains performance advantages over handcrafted libraries under real-world LLM-scale attention workloads (Appendix C), demonstrating strong generalizability and practical applicability in production environments.

Generality across multiple generations of GPUs. LLM-TL demonstrates broad generality, being applicable to a wide range of GPUs, from

	Head Dimension		64						128					
	Sequence Length		512	1k	2k	4k	8k	16k	512	1k	2k	4k	8k	16k
A100	MHA w/ casual mask	cuDNN	95.3	124.4	143.7	152.4	162.8	172.5	106.1	135.4	153.3	165.5	177.8	186.3
		FlexAttention	84.4	107.4	123.7	134.7	145.8	153.3	80.5	105.3	124.7	137.4	150.7	160.3
		flash-attn v2	101.2	127.3	146.5	158.5	172.4	180.8	115.3	143.6	163.8	**176.9**	183.3	**195.1**
		DeepSeek-V3	7.6	7.7	5.5	6.7	7.5	7.7	14.3	14.9	10.7	12.9	14.5	14.9
		DeepSeek-V3 + Ours	**107.4**	**134.6**	**154.7**	**163.4**	**177.6**	**184.3**	**132.2**	**155.6**	**168.7**	176.2	**184.9**	194.7
			↑14.13×	**↑17.48×**	**↑28.13×**	**↑24.39×**	**↑23.68×**	**↑34.94×**	**↑9.24×**	**↑10.44×**	**↑15.77×**	↑13.66×	**↑12.75×**	↑13.07×
	GQA w/ casual mask	cuDNN	95.5	125.6	142.4	152.3	164.8	172.1	107.4	136.6	154.2	165.3	177.6	186.9
		FlexAttention	85.6	108.3	124.1	133.7	146.4	155.2	81.8	106.6	125.4	138.4	153.3	161.1
		flash-attn v2	102.2	128.6	147.7	159.9	172.1	182.4	116.6	144.8	164.1	**176.5**	**186.7**	**197.2**
		DeepSeek-V3	5.4	5.6	4.4	5.1	5.5	6.8	10.5	10.8	8.5	9.9	10.9	11.4
		DeepSeek-V3 + Ours	**107.2**	**133.5**	**154.7**	**163.7**	**179.8**	**185.9**	**131.1**	**154.2**	**167.5**	175.5	184.4	195.6
			↑19.85×	**↑23.84×**	**↑35.16×**	**↑32.10×**	**↑32.69×**	**↑27.34×**	**↑12.49×**	**↑14.28×**	**↑19.71×**	↑17.73×	↑16.92×	↑17.16×
	MQA w/ casual mask	cuDNN	95.1	125.4	143.6	152.8	163.4	172.7	108.8	136.2	154.3	165.6	178.7	185.5
		FlexAttention	85.9	108.4	124.4	133.9	145.6	154.7	82.7	106.9	125.7	137.9	151.6	161.8
		flash-attn v2	103.4	129.6	147.1	159.9	172.5	183.7	116.1	144.5	164.4	**176.9**	**185.1**	**196.4**
		DeepSeek-V3	7.7	8.1	5.8	6.9	7.5	7.7	14.7	15.1	10.9	13.2	14.7	15.1
		DeepSeek-V3 + Ours	**107.6**	**134.8**	**155.4**	**163.7**	**179.2**	**186.9**	**131.1**	**153.6**	**167.7**	175.8	183.9	193.4
			↑13.97×	**↑16.64×**	**↑26.79×**	**↑23.72×**	**↑23.89×**	**↑24.27×**	**↑8.92×**	**↑10.17×**	**↑15.39×**	↑13.32×	↑12.51×	↑12.81×
RTX8000	MHA w/ casual mask	cuDNN	21.4	25.7	28.7	31.2	32.7	33.5	20.9	25.0	28.5	31.1	32.1	32.3
		FlexAttention	**30.4**	**34.5**	**39.7**	**43.9**	46.6	47.7	20.9	24.5	28.7	31.6	33.7	34.1
		flash-attn v1	18.1	17.9	24.3	26.8	31.1	33.7	11.7	16.8	18.4	19.1	21.5	21.9
		DeepSeek-V3	2.6	2.5	1.9	2.4	2.6	OOM	5.1	5.2	3.9	4.7	5.1	4.8
		DeepSeek-V3 + Ours	21.6	29.6	37.9	43.5	**47.8**	**49.9**	**32.3**	**33.1**	**39.1**	**41.2**	**45.0**	**45.6**
			↑8.31×	↑11.84×	↑19.95×	↑18.13×	**↑18.38×**	-	**↑6.33×**	**↑6.37×**	**↑10.03×**	**↑8.77×**	**↑8.82×**	**↑9.50×**
	GQA w/ casual mask	cuDNN	21.5	26.5	28.9	31.2	32.5	32.9	20.4	23.3	28.1	30.4	32.3	32.3
		FlexAttention	**30.6**	**34.3**	**39.6**	**44.2**	46.9	47.8	20.9	24.3	28.7	31.8	33.6	34.2
		flash-attn v1	18.1	18.1	24.4	28.4	31.4	33.7	11.9	16.8	18.8	19.1	21.0	21.9
		DeepSeek-V3	2.1	2.2	1.7	1.99	2.2	OOM	4.2	4.3	3.4	3.9	4.2	4.0
		DeepSeek-V3 + Ours	21.6	28.6	39.5	42.4	**47.9**	**50.4**	**25.9**	**31.5**	**36.4**	**40.7**	**44.6**	**45.5**
			↑10.28×	↑13.00×	↑23.24×	↑21.31×	**↑21.77×**	-	**↑6.17×**	**↑7.33×**	**↑10.71×**	**↑10.44×**	**↑10.62×**	**↑11.38×**
	MQA w/ casual mask	cuDNN	21.8	26.3	30.2	31.7	32.7	33.1	20.5	25.4	29.3	30.9	31.6	31.9
		FlexAttention	**30.7**	**34.5**	**39.8**	**44.5**	46.8	47.5	20.2	23.9	28.5	32.4	33.8	34.4
		flash-attn v1	18.1	18.3	25.4	30.1	31.2	34.2	11.9	16.8	18.8	19.1	21.2	22.0
		DeepSeek-V3	2.6	2.6	1.9	2.5	2.7	OOM	5.1	5.3	4.1	4.8	5.3	4.9
		DeepSeek-V3 + Ours	22.1	29.8	39.1	43.3	**47.8**	**50.6**	**26.1**	**32.5**	**36.4**	**40.7**	**43.7**	**45.1**
			↑8.50×	↑11.46×	↑20.58×	↑17.32×	**↑17.70×**	-	**↑5.12×**	**↑6.13×**	**↑8.88×**	**↑8.48×**	**↑8.25×**	**↑9.2×**
A100	MHA w/o casual mask	cuDNN	153.0	158.8	172.4	175.5	184.7	186.2	172.4	184.2	190.0	196.5	206.9	212.0
		FlexAttention	145.8	155.9	162.5	168.4	177.2	179.9	119.7	134.8	143.2	152.4	158.6	163.7
		flash-attn v2	147.5	161.6	171.1	176.8	185.8	190.6	**208.1**	**208.1**	**208.2**	**210.5**	**221.4**	**223.6**
		DeepSeek-V3	28.9	29.6	28.2	28.5	28.5	29.6	51.6	54.5	52.4	52.6	54.7	56.6
		DeepSeek-V3 + Ours	**164.0**	**175.6**	**181.8**	**191.0**	**200.6**	**201.8**	176.2	194.1	201.1	205.6	206.6	207.2
			↑5.67×	**↑5.93×**	**↑6.45×**	**↑6.70×**	**↑7.03×**	**↑6.82×**	↑3.41×	↑3.56×	↑3.84×	↑3.91×	↑3.81×	↑3.73×
	GQA w/o casual mask	cuDNN	148.8	159.1	164.3	172.3	180.5	186.7	172.3	183.5	189.6	193.6	206.9	211.1
		FlexAttention	146.2	156.3	161.6	172.8	176.9	180.5	121.1	135.1	143.5	155.2	160.5	165.5
		flash-attn v2	148.4	161.2	168.5	175.7	186.8	190.1	**176.9**	**192.5**	**200.0**	**210.4**	**218.4**	**223.8**
		DeepSeek-V3	14.3	14.8	11.7	13.4	14.9	19.8	27.5	28.7	23.1	26.3	29.1	30.8
		DeepSeek-V3 + Ours	**156.9**	**168.4**	**177.6**	**186.9**	**195.1**	**202.7**	169.6	180.4	186.2	196.7	204.8	206.9
			↑10.97×	**↑11.38×**	**↑15.18×**	**↑13.95×**	**↑13.09×**	**↑10.24×**	↑6.17×	↑6.29×	↑8.06×	↑7.48×	↑7.04×	↑6.72×
	MQA w/o casual mask	cuDNN	149.1	159.6	164.7	173.5	180.4	185.5	172.5	184.8	189.9	197.5	208.1	211.4
		FlexAttention	145.8	155.8	161.5	173.4	176.3	180.4	119.5	135.6	143.5	152.5	158.4	164.9
		flash-attn v2	149.6	162.1	168.2	175.8	185.9	190.1	**178.2**	**193.2**	**200.7**	**208.8**	**219.1**	**225.8**
		DeepSeek-V3	29.2	30.7	19.9	25.4	29.1	30.2	54.1	56.2	38.4	48.9	55.6	58.9
		DeepSeek-V3 + Ours	**156.5**	**168.4**	**176.8**	**186.7**	**196.7**	**201.2**	169.2	180.5	187.6	197.9	204.6	207.8
			↑5.36×	**↑5.49×**	**↑8.88×**	**↑7.35×**	**↑6.76×**	**↑6.66×**	↑3.13×	↑3.21×	↑4.89×	↑4.05×	↑3.68×	↑3.53×
RTX8000	MHA w/o casual mask	cuDNN	29.2	32.2	33.4	33.6	33.7	33.4	26.5	30.9	32.2	32.6	32.8	32.6
		FlexAttention	**40.5**	**44.7**	**47.5**	**49.4**	**50.1**	**50.4**	29.4	32.5	33.2	34.7	35.3	35.5
		flash-attn v2	28.5	25.9	34.3	35.1	35.4	36.2	17.2	20.4	21.2	21.0	22.0	22.4
		DeepSeek-V3	8.3	8.5	6.6	8.4	9.3	8.9	15.7	16.6	13.4	16.5	17.9	15.7
		DeepSeek-V3 + Ours	34.4	40.5	44.4	45.9	45.5	45.1	**38.5**	**42.9**	**44.9**	**45.8**	**44.4**	**43.9**
			↑4.14×	↑4.76×	↑6.73×	↑5.46×	↑4.89×	↑5.07×	**↑2.45×**	**↑2.58×**	**↑3.35×**	**↑2.78×**	**↑2.48×**	**↑2.80×**
	GQA w/o casual mask	cuDNN	26.8	31.2	32.1	32.9	33.7	33.8	26.3	29.7	32.1	32.6	33.1	32.8
		FlexAttention	**39.2**	**43.9**	**47.3**	**49.4**	**49.8**	**50.2**	29.5	32.2	33.4	34.9	35.2	35.1
		flash-attn v1	28.9	26.5	33.7	35.7	35.8	37.1	17.4	20.5	21.1	21.1	22.1	22.5
		DeepSeek-V3	5.1	5.1	4.4	5.1	5.5	OOM	10.1	10.3	8.8	10.1	10.8	9.9
		DeepSeek-V3 + Ours	30.3	39.3	43.9	45.4	45.8	46.1	**37.4**	**41.9**	**43.3**	**45.1**	**45.1**	**45.2**
			↑5.94×	↑7.71×	↑9.98×	↑8.90×	↑8.33×	-	**↑3.70×**	**↑4.07×**	**↑4.92×**	**↑4.47×**	**↑4.18×**	**↑4.57×**
	MQA w/o casual mask	cuDNN	27.2	31.1	32.6	33.4	33.8	34.1	25.1	29.3	31.2	32.4	32.9	32.5
		FlexAttention	**39.4**	**44.1**	**46.7**	**49.3**	**49.6**	**50.2**	29.4	32.7	33.5	34.6	34.9	35.3
		flash-attn v1	28.8	26.2	34.5	35.7	35.7	37.2	17.4	20.5	21.3	21.1	22.1	22.4
		DeepSeek-V3	8.6	8.6	6.6	8.4	9.3	OOM	16.7	17.2	13.2	16.2	17.8	15.8
		DeepSeek-V3 + Ours	29.8	39.1	44.4	45.3	45.8	45.9	**36.5**	**41.6**	**43.4**	**44.9**	**44.8**	**45.1**
			↑3.47×	↑4.55×	↑6.73×	↑5.39×	↑4.92×	-	**↑2.19×**	**↑2.42×**	**↑3.29×**	**↑2.77×**	**↑2.52×**	**↑2.85×**

Table 1: Performance (TFLOPS) comparison across different sequence length, attention operators, GPUs, and the presence or absence of causal masks. OOM means out of memory.

the newer Ampere architecture to the older Turing architecture, the latter of which is not supported by official FlashAttention v2. This highlights LLM-TL's generality to reduce development overhead for operators across different GPU generations.

Outstanding performance in long-context scenarios with causal mask. LLM-TL performs exceptionally well in long-context scenarios, particularly when using a causal mask that is commonly employed in modern decoder-only LLMs ($34.94\times$ speedup on 16k tokens on A100). Our method's efficient long-context sequences processing with causal mask offers a critical advantage in modern LLMs development.

Sequence Length	512	1k	2k	4k	8k	16k
torch	22.9	28.7	21.7	26.7	32.9	35.1
cuDNN	35.5	48.6	61.1	70.3	77.3	81.7
DeepSeek-V3	17.7	18.5	13.5	16.1	18.2	18.7
DeepSeek-V3 + Ours	**50.6**	**78.6**	**108.2**	**138.6**	**164.3**	**175.9**
	↑2.86×	**↑4.25×**	**↑8.01×**	**↑8.61×**	**↑9.03×**	**↑9.41×**

Table 2: Performance (TFLOPS) comparison of MLA with causal mask and head dimension 128 on A100 GPU. The MLA configuration parameters and "torch" implementation are extracted from the DeepSeek-V3 open-source code.

4.3 Ablation

Impact of Different LLMs. To demonstrate the robustness of LLM-TL, we conduct ablation experiments on various mainstream LLMs, including GPT-4o, Claude 3.5, DeepSeek-V3 and DeepSeek-R1, as shown in Table 3. After applying LLM-TL, nearly all LLMs can generate high-performance attention operators, which shows the strong versatility of our method. Notably, DeepSeek-R1 achieves the highest performance, while GPT-4o struggles to translate correct CuTe code, potentially due to limitations in its training corpus (GPT-4o is the earliest one within these LLMs). However, it can still generate the TL Code, while the backend translation is handled by DeepSeek-V3.

Impact of Different Prompts. To demonstrate the efficacy of LLM-TL, we compare its performance with other LLMs using CoT for CUDA-based attention operator generation (notably, vanilla DeepSeek-V3 in Table 1 utilizes Torch). Results in Table 5 demonstrate that while LLMs with CoT can only generate basic CUDA implementation, LLM-TL produces high-performance implementations based on CuTe. Performance evaluation reveals that LLM-TL achieves up to $895\times$ speedup compared to CoT.

LLM-TL	Seq = 4k	Seq = 8k	Seq = 16k
w/ GPT-4o	-	-	-
w/ GPT-4o + DeepSeek-V3	165.5	171.9	178.5
w/ Claude 3.5	175.2	179.4	181.3
w/ DeepSeek-V3	175.5	179.3	185.5
w/ DeepSeek-R1	176.2	184.9	194.7

Table 3: Performance (TFLOPS) comparison of MHA with causal mask and head dimension 128 generated by our methods with different LLMs on A100 GPU.

	Time	TFLOPS
Human Expert	$\sim$ months	162.7
LLM-TL	10 mins	175.6

Table 4: Comparison of MHA development cost with human expert on A100 GPU under the configuration of head dimension 64, sequence length 1024.

Impact on Development Cost. Table 4 compares the development costs and performance between LLM-TL and human Expert implementing attention operators on A100 GPU, which both utilize CuTe to achieve better performance. Results demonstrate that LLM-TL not only reduces development time from months to minutes (a thousand-fold reduction) but also achieves modest performance improvements.

Impact on TL Designation. To validate the necessity of hierarchical generation in LLM-TL design (first TL sketch then TL code), we design an ablation experiment to make LLMs directly output TL code. The results demonstrate that none of the existing LLMs is capable of generating entirely correct TL code in a single stage, and representative errors are like Reshape omission when generating fused computation and GEMM layout error. For more details refer to Appendix B.

4.4 Case Study

To demonstrate the generality of LLM-TL, we validate our approach under two extra scenarios, 1) FP8-compatible MHA operator (unsupported by other libraries), and 2) T4 GPU-compatible attention operators. Results in Table 6 and Table 7 in

Sequence Length	512	1k	2k
DeepSeek-V3	0.02	0.004	-
+ CoT	0.12	0.27	0.52
+ LLM-TL	107.4	134.6	154.7

Table 5: CUDA implementation performance (TFLOPS) comparison of MHA with causal mask and head dimension 64 on A100 GPUs between CoT and our method.

Sequence Length	512	1k	2k	4k	8k	16k
Performance	224.8	241.1	248.3	254.6	255.1	257.9

Table 6: Performance (TFLOPS) of MHA with causal mask on L40S GPU using LLM-TL under the configuration of head dimension 128, FP8 datatype.

Appendix show that LLMs leveraging LLM-TL successfully generated high-performance attention operator implementations.

5 Conclusion

In this paper, we propose a novel framework called LLM-TL to systematically unlocks LLMs' potential for high-performance attention operator generation. Extensive experiments identify that LLM-TL enables LLMs to fully understand algorithm-hardware execution flows via TL-code and adapt across GPU architectures through TL-code translation. The results show that the code generated through LLM-TL could outperform hand-optimized libraries crafted by human experts while reducing months development time to minutes. In summary, LLM-TL not only extends the capacities of LLMs on complex code optimization, but also achieves a self-optimizing paradigm for generating high-performance attention operators in attention-centric algorithms.

Limitations

Our proposed LLM-TL and its corresponding workflow enable LLMs to comprehend and generate high-performance attention operator implementations across various GPUs. Compared to vanilla LLMs, this approach demonstrates significant performance improvements, achieving results that are comparable to or surpass those of state-of-the-art handcrafted libraries. However, our method does have some limitations.

Due to resource constraints, although we conducted extensive experiments across multiple generations of GPUs, testing on the latest H100 was not performed. Additionally, while LLM-TL and its workflow are designed to be generalizable to complex operators on GPUs, we did not further evaluate its applicability to other types of operators, given that FlashAttention has been a highly complex and widely used operator already.

To address these limitations, our future work will focus on extending LLM-TL to GPUs with newer architectures and applying it to a broader range of complex operators.

6 Ackowledgement

This work is partially supported by the Strategic Priority Research Program of the Chinese Academy of Sciences (Grants No.XDB0660300, XDB0660301, XDB0660302), the NSF of China (Grants No.62525203, 92364202, U22A2028, 62302483), Major Program of lSCAS (Grant No. ISCAS.ZD-202402), CAS Project for Young Scientists in Basic Research (YSBR-029) and Youth Innovation Promotion Association CAS.

Ethics Statement

This research on generating high-performance attention operators using LLMs considers ethical implications, including bias, misuse, and societal impact. We emphasize transparency, responsible data practices, and encourage ethical deployment of AI in software engineering.

References

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebron, and Sumit Sanghai. 2023. GQA: Training generalized multi-query transformer models from multi-head checkpoints. In *The 2023 Conference on Empirical Methods in Natural Language Processing*.

Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Mérouane Debbah, Étienne Goffinet, Daniel Hesslow, Julien Launay, Quentin Malartic, Daniele Mazzotta, Badreddine Noune, Baptiste Pannier, and Guilherme Penedo. 2023. The falcon series of open language models. *Preprint*, arXiv:2311.16867.

Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, et al. 2024. Codeplan: Repository-level coding using llms and planning. *Proc. ACM Softw. Eng.*, 1(FSE).

Tianqi Chen, Thierry Moreau, Ziheng Jiang, et al. 2018. 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.

Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. *Preprint*, arXiv:1410.0759.

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

Tri Dao. 2024. Flashattention-2: Faster attention with better parallelism and work partitioning. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. In *Advances in Neural Information Processing Systems*, volume 35, pages 16344–16359. Curran Associates, Inc.

DeepSeek-AI, Daya Guo, Dejian Yang, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *Preprint*, arXiv:2501.12948.

DeepSeek-AI, Aixin Liu, Bei Feng, et al. 2024a. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *Preprint*, arXiv:2405.04434.

DeepSeek-AI, Aixin Liu, Bei Feng, et al. 2024b. Deepseek-v3 technical report. *Preprint*, arXiv:2412.19437.

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

Samuel Holt et al. 2024. L2mac: Large language model automatic computer for extensive code generation. In *The Twelfth International Conference on Learning Representations*.

Ke Hong, Guohao Dai, Jiaming Xu, Qiuli Mao, Xiuhong Li, Jun Liu, Kangdi Chen, Yuhan Dong, and Yu Wang. 2024. Flashdecoding++: Faster large language model inference on gpus. *Preprint*, arXiv:2311.01282.

Raymond Li, Loubna Ben Allal, Yangtian Zi, et al. 2023. Starcoder: may the source be with you! *Preprint*, arXiv:2305.06161.

Rui Li, Liyang He, Qi Liu, et al. 2024. Consider: Commonalities and specialties driven multilingual code retrieval framework. In *AAAI Conference on Artificial Intelligence*.

Llama3.1. 2024. Introducing llama 3.1: Our most capable models to date. https://ai.meta.com/blog/meta-llama-3-1/.

Stefano Markidis, Der Chien, and Steven Wei et al. 2018. Nvidia tensor core programmability, performance & precision. In *2018 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, pages 522–531. IEEE.

NVIDIA/CUTLASS. 2023. Getting started with cute. https://github.com/NVIDIA/cutlass/blob/main/media/docs/cute/00_quickstart.md.

OpenAI. 2024. Gpt-4o. https://openai.com/index/hello-gpt-4o/.

Adam Paszke and Gross et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32.

Noam Shazeer. 2019. Fast transformer decoding: One write-head is all you need. *Preprint*, arXiv:1911.02150.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010. Curran Associates Inc.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, et al. 2023. Codet5+: Open code large language models for code understanding and generation. *Preprint*, arXiv:2305.07922.

An Yang, Baosong Yang, Beichen Zhang, et al. 2025. Qwen2.5 technical report. *Preprint*, arXiv:2412.15115.

Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, YX Wei, Lean Wang, Zhiping Xiao, et al. 2025. Native sparse attention: Hardware-aligned and natively trainable sparse attention. *arXiv preprint arXiv:2502.11089*.

Xuzhi Zhang, Shaohui Peng, Qirui Zhou, Yuanbo Wen, Qi Guo, Ruizhi Chen, Xinguo Zhu, Weiqiang Xiong, Haixin Chen, Congying Ma, Ke Gao, Chen Zhao, Yanjun Wu, Yunji Chen, and Ling Li. 2025. Qimeng-tensorop: Automatically generating high-performance tensor operators with hardware primitives. *Preprint*, arXiv:2505.06302.

Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: Generating High-Performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 863–879. USENIX Association.

Li Zhong 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, pages 21841–21849.

Qirui Zhou, Yuanbo Wen, Ruizhi Chen, Ke Gao, Weiqiang Xiong, Ling Li, Qi Guo, Yanjun Wu, and Yunji Chen. 2025. Qimeng-gemm: Automatically generating high-performance matrix multiplication code by exploiting large language models. *Proceedings of the AAAI Conference on Artificial Intelligence*, 39(21):22982–22990.

A Additional Evaluations

We further conduct extensive experiments across various NVIDIA hardware architectures to validate the robustness and scalability of our proposed method. Comparing our automatically generated code with three implementations on T4 GPU. The experimental results, as presented in Table 7, demonstrate that the code generated by our method consistently achieves superior performance metrics compared to all four implementations. This performance advantage is maintained across all evaluated dimensions, indicating the robustness and effectiveness of our approach on the NVIDIA T4 GPU architecture.

Moreover, we extend our experiments to assess the applicability of our method on alternative attention mechanisms, NSA(Yuan et al., 2025). The results, summarized in Table 9, show that our approach maintains its performance advantage when applied to these attention variants. For NSA, the automatically generated code consistently outperforms manually optimized baselines in terms of latency and throughput, reaffirming the versatility and adaptability of our method across diverse attention architectures.

B Ablation on TL Code Generation stage

We conduct an ablation study to validate the necessity of the hierarchical generation in TL Code Generation Stage (i.e., first generate TL sketch then TL code). Results demonstrate that existing LLMs fails to directly generate TL code. Listing 1 and Listing 2 provide the failure cases.

Reshape omission. LLMs occasionally fail to perform the necessary layout reshape operation on the result matrix generated by the first GEMM operator before proceeding to the second GEMM operator. Although the output matrix of the first GEMM operator and one of the input matrices of the second GEMM operator are mathematically consistent, their memory layouts differ significantly due to the use of MMA instructions. Therefore, a layout reshape operation must be introduced to reorganize the matrix layout to meet the requirements of MMA instructions. The lack of TL Sketch prevents LLMs from recognizing the critical layout transformation, causing reshape omissions and computation errors.

GEMM error. LLMs fail to rigorously distinguish between TL symbolic representations and their corresponding physical memory layout mappings. Specifically, although the first GEMM operator in the TL layer is defined as Q multiplied by K^T, the matrix K physically retains its original layout in memory due to the characteristics of the MMA instruction set. Nevertheless, the TL abstraction layer must maintain formal transpose notation to guide subsequent translation processes properly. Due to the absence of TL Sketch intermediate representation generation, LLMs fail to effectively bridge the semantic gap between algorithmic abstraction and hardware implementation layers. This results in the conflation of formal transpose notation in TL with actual physical memory layouts (without explicit transposition), leading to the loss of critical semantic constraints during TL-to-CuTe translation and ultimately causing implementation-level mismatches.

```
Compute GEMM Q_shared, K_shared.T and
    get S
if i < (kv_len/BN) - 1
  Copy K (BN, HeadDim) in coordinate [L
    = i+1] from global to shared
end
Compute Softmax S
// No reshape!
Compute GEMM S, V_shared and accumulate
    O_register
```

Listing 1: A Case of Reshape Omission.

```
Compute GEMM Q_shared, K_shared and get
    S //K_shared should be transposed
Compute Softmax S with Smax and Ssum
Reshape S from (MMA_C, MMA_M, MMA_N) to
    (MMA_A, MMA_M, MMA_N_new)
Compute GEMM S, V_shared and accumulate
    O_reg
```

Listing 2: A Case of GEMM Error.

C Real-World LLM Configuration

Currently, LLMs predominantly employ MHA and GQA mechanisms in their transformer backbone. While these models uniformly adopt a head dimension of 128 across different architectures, significant variations exist in their configuration of query heads versus key and value heads. To systematically evaluate LLM-TL on these architectural distinctions, we select three representative models spanning diverse scales and configurations: Llama2 7B (32 query heads, key and value heads), Qwen2.5 72B (64 query heads / 8 key and value heads), and Llama3.1 405B (128 query heads / 8 key and value heads).The experimental results, as presented in Table 8, demonstrate that the code generated by LLM-TL consistently achieves superior performance metrics compared to all four

Head Dimension		64						128					
Sequence Length		512	1k	2k	4k	8k	16k	512	1k	2k	4k	8k	16k
Masked MHA	cuDNN	8.11	10.84	12.13	13.22	13.69	13.83	7.73	10.41	12.14	13.03	13.76	13.01
	FlexAttention	**10.82**	13.45	16.31	18.52	19.84	20.47	7.81	10.53	12.62	14.26	15.02	14.91
	flash-attn v1	8.68	9.85	12.81	12.81	13.83	13.25	7.8	8.77	9.88	10.68	10.68	10.53
	DeepSeek-V3	1.33	1.35	0.99	1.21	OOM	OOM	2.46	2.64	1.92	2.26	2.38	OOM
	DeepSeek-V3 + Ours	9.83	**13.48**	**16.62**	**19.11**	**20.72**	**21.43**	**9.92**	**13.67**	**16.82**	**18.61**	**19.62**	**19.07**
		↑7.39×	↑**9.99×**	↑**16.79×**	↑**15.80×**	-	-	↑**4.03×**	↑**5.18×**	↑**8.76×**	↑**8.23×**	↑**8.24×**	-
Masked GQA	cuDNN	8.21	10.42	12.13	13.03	13.5	13.85	7.74	10.41	11.63	12.84	13.49	12.93
	FlexAttention	**10.51**	13.45	16.17	18.54	20.13	20.58	7.92	10.24	12.43	14.48	15.13	14.89
	flash-attn v1	6.85	7.93	12.79	13.22	13.56	13.46	5.3	9.37	9.85	10.62	10.86	10.51
	DeepSeek-V3	1.05	1.06	0.82	0.97	OOM	OOM	1.97	2.05	1.58	1.81	1.93	OOM
	DeepSeek-V3 + Ours	9.76	**13.68**	**16.45**	**19.13**	**20.81**	**21.58**	**10.73**	**14.08**	**16.32**	**18.03**	**19.15**	**18.7**
		↑9.30×	↑**12.91×**	↑**20.06×**	↑**19.72×**	-	-	↑**5.45×**	↑**6.87×**	↑**10.33×**	↑**9.96×**	↑**9.92×**	-
Masked MQA	cuDNN	8.41	10.45	11.77	12.95	13.43	13.78	8.39	10.32	11.62	13.03	13.6	12.82
	FlexAttention	**10.67**	13.42	16.44	18.39	19.91	20.27	7.89	10.42	12.64	14.12	14.98	14.95
	flash-attn v1	5.73	11.53	12.65	13.74	13.59	13.38	5.27	9.42	9.95	10.71	10.32	10.47
	DeepSeek-V3	1.35	1.36	0.99	1.21	OOM	OOM	2.52	2.61	1.91	2.26	2.37	OOM
	DeepSeek-V3 + Ours	9.65	**13.61**	**17.11**	**18.84**	**20.59**	**21.27**	**10.19**	**14.01**	**16.7**	**17.92**	**19.17**	**18.79**
		↑7.15×	↑**10.00×**	↑**17.28×**	↑**15.57×**	-	-	↑**4.04×**	↑**5.37×**	↑**8.74×**	↑**7.93×**	↑**8.09×**	-
MHA	cuDNN	10.47	12.51	13.01	13.61	13.95	13.76	10.81	12.35	12.95	13.51	13.53	13.58
	FlexAttention	**14.42**	**16.91**	**18.77**	**19.85**	**20.13**	**20.25**	11.34	13.50	14.83	15.67	15.74	15.42
	flash-attn v1	12.25	15.27	14.28	14.25	14.25	14.46	8.78	11.03	10.95	10.61	10.38	10.78
	DeepSeek-V3	4.23	4.54	3.18	4.07	OOM	OOM	7.87	8.23	6.11	7.16	7.48	OOM
	DeepSeek-V3 + Ours	13.35	15.68	18.35	19.36	19.59	19.67	**15.8**	**17.79**	**18.59**	**18.68**	**18.86**	**19.13**
		↑3.16×	↑3.45×	↑5.77×	↑4.76×	-	-	↑**2.01×**	↑**2.16×**	↑**3.04×**	↑**2.61×**	↑**2.52×**	-
GQA	cuDNN	10.63	12.35	13.14	13.39	13.67	13.52	10.97	12.28	13.02	13.52	13.64	13.47
	FlexAttention	**14.43**	16.72	**18.87**	**20.12**	**20.34**	**20.24**	11.43	13.74	14.95	15.52	15.64	15.51
	flash-attn v1	8.89	15.43	14.41	14.48	14.35	14.52	10.98	11.01	10.95	11.01	10.33	10.8
	DeepSeek-V3	2.47	2.49	2.08	OOM	OOM	OOM	4.56	4.77	3.97	4.48	OOM	OOM
	DeepSeek-V3 + Ours	13.82	**16.79**	18.47	19.68	19.51	19.47	**15.81**	**17.59**	**18.82**	**19.6**	**19.53**	**19.58**
		↑5.60×	↑**6.74×**	↑8.88×	-	-	-	↑**3.47×**	↑**3.69×**	↑**4.74×**	↑**4.38×**	-	-
MQA	cuDNN	10.79	12.17	13.23	13.62	13.85	13.69	10.45	12.39	13.03	13.25	13.48	13.38
	FlexAttention	**14.61**	**16.94**	18.57	**19.93**	**20.31**	**20.14**	11.42	13.85	14.64	15.28	15.76	15.56
	flash-attn v1	11.23	15.32	14.59	14.12	14.14	14.51	11.19	11.06	11.01	11.02	10.72	10.88
	DeepSeek-V3	4.33	4.41	3.15	3.97	OOM	OOM	7.91	8.31	5.99	7.11	7.32	OOM
	DeepSeek-V3 + Ours	13.58	16.34	**18.77**	19.67	19.79	19.62	**14.94**	**16.39**	**18.14**	**19.4**	**19.08**	**19.46**
		↑3.14×	↑3.71×	↑**5.96×**	↑4.95×	-	-	↑**1.89×**	↑**1.97×**	↑**3.03×**	↑**2.73×**	↑**2.61×**	-

Table 7: Performance (TFLOPS) comparison across different dimensions, attention operators and the presence or absence of causal masks on T4 GPU.

implementations. This performance advantage is maintained across all evaluated configuration, indicating the robustness and effectiveness of LLM-TL on different heads.

D Prompts

Some prompts for guiding TL Code generating and reasoning are shown in Listing 3 and Listing 4, respectively.

For TL Code generating, take *Copy* statement as an example, we first instruct LLMs about the memory hierarchies of GPUs, to establish a foundational understanding of data movement across these hierarchies. Additionally, we introduce the architecture and operational principles of Tensor Core, emphasizing their role in MMA operations through warp-level parallelism and efficient register utilization. Following this, we provide the precise syntax and structure of TL Code, enabling LLMs to generate hardware-aware implementations correctly and effectively.

For TL Code reasoning, we instruct LLMs to derive specific dimensional information to refine the execution process of attention operator. Take *Copy* statement again as an example, we first introduce parameter-related variable allocation mechanisms. Then, we let LLMs perform different operations based on the different cases of the *Copy* statements, while the specific details such as dimensional information are generated by the LLMs themselves.

```
Use the following basic TL statements to
     describe the provided algorithm
    flow. Focus solely on describing the
     hardware execution process of the
    algorithm on the GPU, without adding
     excessive complex information.
Here are two basic statements of the TL,
     `Copy` and `Compute`:
### Copy
The term "Copy" is used to denote the
    transfer of data between different
    levels of storage hierarchy in
    hardware. In GPUs, there are three
```

Sequence Length	512	1k	2k	4k	8k	16k
Llama2 7B (32 Q-heads/32 KV-heads/128 Head-dimension)						
cuDNN	112.4	142.6	164.1	176.8	197.2	201.7
FlexAttention	82.8	108.6	128.6	150.2	158.4	167.1
flash-attn v2	122.5	152.5	173.4	186.3	201.5	207.3
DeepSeek-V3	14.6	15.1	10.9	13.3	14.6	15.1
DeepSeek-V3 + Ours	**137.1**	**160.6**	**180.3**	**186.7**	198.3	202.7
	↑**9.39×**	↑**10.64×**	↑**16.54×**	↑**14.04×**	↑13.58×	↑13.42×
Qwen2.5 72B (64 Q-heads/8 KV-heads/128 Head-dimension)						
cuDNN	116.3	148.3	176.1	191.8	202.1	207.9
FlexAttention	86.1	117.4	136.7	154.7	168.1	172.4
flash-attn v2	127.1	157.8	181.5	**201.8**	**216.0**	**222.5**
DeepSeek-V3	10.9	11.1	8.8	10.2	11.2	OOM
DeepSeek-V3 + Ours	**140.2**	**165.1**	**186.4**	192.9	201.8	205.1
	↑**12.86×**	↑**14.87×**	↑**21.18×**	↑18.91×	↑18.02×	-
Llama3.1 405B (128 Q-heads/8 KV-heads/128 Head-dimension)						
cuDNN	121.9	157.2	183.4	196.7	206.4	211.2
FlexAttention	93.2	117.2	144.4	157.7	169.7	175.3
flash-attn v2	129.1	164.1	**192.2**	**209.1**	**221.1**	**225.3**
DeepSeek-V3	11.1	11.2	8.9	10.2	OOM	OOM
DeepSeek-V3 + Ours	**146.5**	**168.4**	187.5	200.3	204.4	206.9
	↑**13.20×**	↑**15.03×**	↑21.07×	↑19.64×	-	-

Table 8: Performance (TFLOPS) compared with handcraft libraries in different configuration on A100 GPU. The head dimension for all three LLMs is consistently set to 128 and all of attention operators are applied with causal mask.

Sequence Length	512	1k	2k	4k	8k	16k
Naive NSA	0.84	1.68	3.35	6.61	13.34	26.29
ours	**0.67**	**1.26**	**2.59**	**5.25**	**10.59**	**21.27**
	↑**1.25×**	↑**1.33×**	↑**1.29×**	↑**1.26×**	↑**1.26×**	↑**1.24×**

Table 9: The NSA evaluations of latency(s) on A100 with head dimension of 128. We compared with naive pyTorch implementation of NSA.

```
    primary storage levels:
- **global memory:** Global memory is
    high-capacity, high-latency video
    memory used for storing data shared
    by all threads, typically for
    holding complete matrices or large-
    scale datasets.
- **shared memory:** Shared memory is
    high-speed, low-latency on-chip
    memory used for data sharing within
    a thread block. It is typically used
     to store a small portion of data
    required for collaborative
    computation by the current thread
    block, loaded from global memory.
- **register:** Registers are the
    fastest private storage units, used
    for storing thread-local variables
    and temporary data. In CUDA Tensor
    Core operations, registers are
    directly involved in matrix multiply
    -accumulate (MMA) computations,
    where each thread fetches its
    assigned data from shared memory and
     stores it in registers.
```

```
In this context, a "Copy" operation
    requires specifying the variable
    name as well as the source and
    destination addresses of the
    variable.  Here is the usage of `
    Copy` statement:
```
```
Copy A from global to shared
```
```
The clause means load a block of the
    matrix `A` to the corresponding
    shared memory storage.
### Compute
The term "compute" is used to represent
    computations performed on hardware.
    Computational descriptions include
    various types of operations,
    primarily arithmetic operations (
    addition, subtraction,
    multiplication, division), matrix
    multiplication (GEMM), accumulation,
     and others. Here are some typical
    computations:
- **GEMM:** Use *GEMM* (General Matrix
    Multiplication) to represent the
    multiplication operation of two
    matrices at the register storage
    level, leveraging the fast access
    characteristics of registers to
    achieve high-performance matrix
    computations. Use GEMM. This
    primitive can be used in the
    following manner:
    ```
    Compute GEMM A, B and get S
    Compute GEMM A, B and accumulate S
```
```

````
    ```
    Here we compute the GEMM result of A
     and B, and store the result to the
    variable S.
- **Regular computation:** We need some
    regular computation like **the four
    basic arithmetic operations**, and
    we can use these basic operations
    like this:
    ```
    Compute Multiply A, x and get new A
    Compute Multiply A, x and get B
    ```
    Here we use these clauses to define
    the **multiplication** operation,
    and the first means store the result
     back to A while the other means
    store to a new variable B.
- **Other operators:** Sometimes the
    users will define some other custom
    operators like softmax, and we can
    use it like:
    ```
    Compute Softmax A
    ```
In this context, "compute" first
    requires specifying the exact type
    of computation, along with the
    variables involved in the
    computation and the variable name
    for the result of the computation.
### Loop
When describing the execution flow of an
     algorithm, it is often necessary to
     use **for loops** to represent
    iterative operations of operators.
    In such cases, a **For statement**
    is used to describe these loops. The
     syntax `for i = 0:N ... end` is
    employed to indicate a loop that
    iterates **N times**, and **
    indentation** is used to control
    which code blocks are executed
    within the loop.
```
for i=0:N
  ...
end
```
Now, based on the algorithm workflow
    provided by the user, analyze the
    memory access and computational
    behavior of the algorithm on the GPU
    , and use the aforementioned
    statements to represent the
    semantically enriched execution flow
     of the algorithm.
````

Listing 3: Prompt for guiding LLMs to generate TL Code.

````
Several fundamental statement types have
     been defined to represent the
    algorithm execution flow. Building
    upon these primitive constructs, we
    need to derive specific dimensional
    information to refine the algorithm
    execution process. Besides, to
    facilitate the generation of
    functionally correct final code,
    this step introduces parameter-
    related variable allocation
    mechanisms. We achieve this process
    by incorporating allocate statements
    , thereby ensuring proper allocation
     and management of memory resources.
     Specifically, the dimensional
    information to be derived includes:
### Copy
For the `Copy` statement, it is
    necessary to complete its parameter
    configuration based on the execution
     characteristics of algorithm on the
     GPU architecture. The optimization
    primarily focuses on access
    locations within global memory.
- Global to shared
Prior to performing the `Copy` statement
     from or to global memory, it is
    necessary to fully characterize the
    matrix information in global memory.
     The Allocate statement can be used
    to represent the storage layout and
    attributes of the entire matrix in
    global memory like the follow:
```
Allocate A in global (M, K) with offset
    batch_offset
Copy A from global to shared
...
```
For  matrix `A` with shape `(batch, M, N
    )`, here we allocate tensor `A`
    whose shape is (M, K). Assuming that
     each block will be responsible for
    a batch, the global that should be
    loaded will have an offset of
    batch_offset. It can be used in the
    final implementation code. This
    Allocate statement needs to be
    applied to every `Copy` statement
    involving global memory, unless the
    corresponding memory allocation has
    already been explicitly declared in
    the preceeding context.
For `Copy` statement itself, for
    instance, this clause:
```
Copy A from global to shared
```
If it is required to load a data block
    of size (BM, BK) from matrix `A`
    located at position L = i, the
    parameter configuration can be added
     in the following manner:
```
Copy A (BM, BK) in coordinate [L = i]
    from global to shared
```
The first clause represents the original
     Sketch description. By
    incorporating the aforementioned
    parameter information, the following
     complete implementation can be
    derived the second one.
Here you should note that: L = i
    represents the i -th block after
    tiling. For example, if A is a
    matrix with shape (M, BK),  `Copy` A
     (BM, BK) in coordinate [L = i]
````
````

````
    selects the i-th block, which is
    with the shape (BM, BK) . And there
    will be M / BM blocks in total.
### Compute
The dimensionality of the compute
    statement is directly determined by
    the dimensions of its input and
    output registers, thus requiring no
    additional dimensional information.
    However, there are still two aspects
     related to the compute statement
    that require further supplementation
    :
- **Declare intermediate variables:**
    ****During the computation process,
    certain intermediate register
    variables serving as outputs require
     additional definition of their
    shape and storage attributes. Here
    is an example of GEMM:
    ```
    ...
    for i in 0:K:
      Compute GEMM A, B and get C
      ...
    ```
    where it didn't declare what C is,
    so you should add a allocate clause
    before the for loop to represent the
     shape information of S in register.
    ```
    ...
    Allocate C in register (BM, BN)
    for i in 0:K:
      Compute GEMM A, B and get C
      ...
    ```
    Here we allocate the C in register
    with the information of whole
    dimension of block memory, and the
    real register shape can be inferred
    from the whole size.
- **Fuse two GEMM(Add reshape):** In the
     scenario of fusing two consecutive
    GEMM operations, where the output of
     the first operation directly serves
     as the input to the second, it is
    necessary to transform the output of
     the first operation according to
    the specific computational scale to
    meet the input requirements of the
    second GEMM operation.
    Here is some prior knowledge: The
    layout of Tensor Core can be
    represented as (MMA, MMA_M, MMA_N).
    Here, MMA denotes the computational
    scale required for matrices A, B,
    and C in the GEMM operation, while
    MMA_M and MMA_N represent the
    repetition counts of the computation
     along the M and N dimensions,
    respectively.
    Here is an example of fuse:
    ```
    Compute GEMM E, F and get G
    ... //There might be other
    operations, such as `Compute Softmax
     G`
    Compute GEMM G, H and accumulate I
    ```
    Here `G` is the output of first GEMM
     and then used in second GEMM. There
     might be other operations between
    two GEMMs, such as softmax.
    HOWEVER, whenever there is an
    association between two GEMMs, a
    reshape statement must be added. So
    you have to add a reshape statement
    according to layout G. Add the
    statement before GEMM-II, just like
    the example below:
    ```
    Compute GEMM E, F and get G
    ... //There might be other
    operations, such as `Compute Softmax
     G`
    Reshape G from (MMA_C, MMA_M, MMA_N)
     to (MMA_A, MMA_M, MMA_N_new)
    Compute GEMM G, H and accumulate I
    ```
    In this example, the `G` of GEMM-I
    is one of the input of GEMM-II.
    Since C is the `A` matrix of tensor
    core, the MMA shape of G should be
    changed from `MMA_C` to `MMA_A` ,
    and change the `MMA_N` to adapt this
     difference.
````

Listing 4: Prompt for guiding LLMs to analyse parameters in the TL Code and reason the TL Code.