CuBridge: An LLM-Based Framework for Understanding and Reconstructing High-Performance Attention Kernels

Xing Ma[1,2*], **Yangjie Zhou**[3*†], **Wu Sun**[1], **Zihan Liu**[1,2], **Jingwen Leng**[1,2], **Yun Lin**[1†], **Shixuan Sun**[1], **Minyi Guo**[1], **Jin Song Dong**[3]
[1]Shanghai Jiao Tong University, [2]Shanghai Qi Zhi Institute, [3]National University of Singapore
{maxing1207, by5892q, altair.liu, lin_yun, sunshixuan}@sjtu.edu.cn
{leng-jw, guo-my}@cs.sjtu.edu.cn, {yj_zhou, dcsdjs}@nus.edu.sg

Abstract

Efficient CUDA implementations of attention mechanisms are critical to modern deep learning systems, yet supporting diverse and evolving attention variants remains challenging. Existing frameworks and compilers trade performance for flexibility, while expert-written kernels achieve high efficiency but are difficult to adapt. Recent work explores large language models (LLMs) for GPU kernel generation, but prior studies report unstable correctness and significant performance gaps for complex operators such as attention.

We present *CuBridge*, an LLM-based framework that adapts expert-written attention kernels through a structured lift–transfer–lower workflow. *CuBridge* starts from expert-written CUDA attention kernels and lifts them into an executable intermediate representation that makes execution orchestration explicit while abstracting low-level CUDA syntax. Given a user-provided PyTorch specification, *CuBridge* generates and verifies a target IR program, then reconstructs optimized CUDA code via reference-guided lowering. Across diverse attention variants and GPU platforms, *CuBridge* consistently produces correct kernels and substantially outperforms general frameworks, compiler-based approaches, and prior LLM-based methods.

1 Introduction

Attention kernels are a central performance component in modern deep learning systems (Paszke et al., 2019; Abadi et al., 2016; Chen et al., 2018). As model architectures evolve, beyond standard softmax attention, attention increasingly appears in customized forms, including new masking behaviors, modified score computations, alternative normalization rules, and fused pipelines. Efficiently supporting these variants on GPUs is therefore not a one-off engineering task, but an ongoing systems challenge.

Existing approaches expose a fundamental trade-off between generality and performance. High-level frameworks (Paszke et al., 2019) and compilers (Guessous et al., 2024) provide flexible expression of new attention semantics but rely on generic and inefficient implementations, while expert libraries (Dao, 2024; Shah et al., 2024) achieve strong efficiency through carefully engineered kernels that are difficult to adapt. As a result, extending high-performance attention kernels to new variants often requires substantial manual effort.

Large language models (LLMs) offer a promising direction for reducing the manual effort involved in GPU kernel development, and recent benchmarks have explored their use for kernel generation (Ouyang et al., 2025; Li et al., 2025a). However, for complex operators such as attention, prior studies report unstable correctness and substantial performance gaps relative to optimized baselines (Ouyang et al., 2025), indicating that directly applying LLMs to attention kernels remains unreliable in practice.

A key observation underlying this work is that expert-written CUDA kernels already encode correct and efficient execution behavior. Rather than generating attention kernels from scratch, we treat these expert implementations as performance references and focus on enabling controlled semantic adaptation while preserving their execution structure. This perspective motivates a workflow that separates semantic reasoning from low-level code manipulation, allowing modifications to be expressed at an appropriate level of abstraction.

Guided by this insight, we present *CuBridge*, an LLM-based framework that adapts expert-written attention kernels through a structured lift-transfer-lower workflow. *CuBridge* first lifts source CUDA code into a structured intermediate representation CuIR, which makes execution orchestration ex-

[*]Equal contribution.
[†]Corresponding authors.

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 10929–10946

plicit while retaining performance-critical information. It then transfers the lifted IR program to match user-specified attention semantics, and finally lowers the transformed representation back into optimized CUDA code via reference-guided reconstruction. Crucially, the intermediate representation is executable, enabling verification at each stage and ensuring semantic correctness.

We implement and evaluate *CuBridge* on NVIDIA A100 and H100 across a range of attention variants and real-model configurations. Our results show that *CuBridge* consistently produces correct kernels and achieves, on average, speedups of $16.03\times$ over general frameworks, $1.39\times$ over template-based compilers, and $3.33\times$ over prior LLM-based approaches. These results demonstrate the effectiveness of our approach for supporting evolving attention mechanisms with both correctness and high performance.

Our contributions are summarized as follows:

- We propose a new LLM-assisted paradigm for adapting expert-written attention kernels that avoids end-to-end code generation, and instead performs semantic adaptation through a structured lift–transfer–lower workflow.
- We design CuIR, an executable intermediate representation that makes performance-critical execution orchestration explicit while abstracting low-level CUDA syntax, and build an end-to-end system *CuBridge* on top of it to enable reliable semantic adaptation, intermediate verification, and reference-guided kernel reconstruction.
- We evaluate *CuBridge* across a wide range of attention variants and GPU platforms, showing that it consistently produces correct kernels and outperforms general frameworks, template-based compilers, and prior LLM-based approaches.

2 Preliminaries

The goal of this work is to efficiently support high-performance CUDA (NVIDIA, 2025b) kernels for diverse attention variants. Achieving high performance on GPUs relies on complex and carefully engineered coordination across multiple levels of parallelism and memory hierarchies (NVIDIA, 2021, 2023). As attention mechanisms continue to evolve, supporting new semantics with high-performance CUDA kernels becomes increasingly challenging.

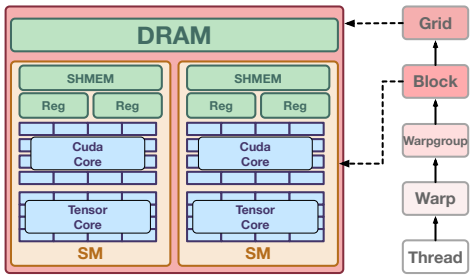

Figure 1: Hardware architecture and execution model of modern GPUs.

2.1 GPU Architecture and Execution Model

Modern GPUs expose a hierarchical organization of both execution units and memory resources, as shown in Figure 1. CUDA kernels run under the SIMT model (Luebke, 2008), where threads are grouped into warps and scheduled in thread blocks, while data resides in a memory hierarchy including global memory, shared memory, and registers. High-performance CUDA kernels depend on carefully coordinating computation and data movement across these execution units and memory levels. We refer to this coordination as *execution orchestration*, which determines how work is assigned to parallel units, how operations are ordered with synchronization, and how data is moved across the memory hierarchy, all of which directly impact performance.

Modern hardware features such as tensor cores (NVIDIA, 2017), asynchronous memory copies (NVIDIA, 2021), and pipelined execution (NVIDIA, 2023) further increase the complexity of execution orchestration. For example, warp-specialized (NVIDIA, 2025a; Bauer et al., 2014) pipelines assign different roles to different warps within a thread block and rely on explicit synchronization to coordinate data movement and computation. As a result, execution orchestration plays a central role in determining the performance characteristics of CUDA kernels.

2.2 Attention Variants

Given query, key, and value matrices $Q, K, V \in \mathbb{R}^{L\times d}$, the standard scaled dot-product attention (Vaswani et al., 2017) is defined as:

$$O = \mathrm{Softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V \tag{1}$$

A wide range of attention variants have been proposed to support different modeling requirements. At a semantic level, many of these variants can be

```
__device__ void flash_gemm(…) {
    ...
    asm volatile("wgmma.fence.sync.…");
    cute::gemm(…); // use wgmma ptx
    asm volatile("wgmma.commit_group…");
}
…
// async gemm to compute S = Q * K
flash_gemm(…);
// other operations overlap with compute S
…
Mask(S); // mistake point
wait_group<1>( ); // wait for async gemm
Mask(S); // correct point
softmax( );
```

Figure 2: Failure case of directly applying LLM to modify FlashAttention-3 CUDA code for the PrefixLM mask variant with GPT-5.

understood as modifying different semantic components of the attention computation. In particular, attention can be viewed as a sequence of score computation, masking, and normalization:

$$O = \textbf{Norm}\left(\textbf{Mask}\left(\textbf{Score}(\frac{QK^T}{\sqrt{d}})\right)\right)V \quad (2)$$

Under this view, variants commonly modify the score function (Shaw et al., 2018), the masking rules (Wang et al., 2025; Pal et al., 2023; Zaheer et al., 2020; Ouyang et al., 2022; Bertsch et al., 2025), or the normalization behavior (Dao et al., 2022; Ramapuram et al., 2025).

While these semantic changes appear localized at the algorithmic level, their realization in GPU kernels often requires conditional execution, boundary handling, or changes in loop structure. Such control-flow and data-dependence changes directly affect how computation, synchronization, and data movement are orchestrated on the GPU, making correct and efficient kernel implementation significantly more challenging.

2.3 Current Support for Attention Variants

Existing systems for supporting attention variants exhibit a fundamental trade-off between generality and performance. General frameworks (e.g., PyTorch (Paszke et al., 2019)) allow new attention semantics to be expressed easily at a high level, but do not provide expert-level CUDA kernels for most variants. Template-based compilation approaches (e.g., FlexAttention (Guessous et al., 2024)) allow limited customization within fixed templates, while expert libraries (e.g., FlashAttention (Dao, 2024; Shah et al., 2024)) achieve high efficiency but require substantial manual effort to extend. As a result, non-standard attention variants either suffer from noticeable performance degradation or demand significant manual engineering.

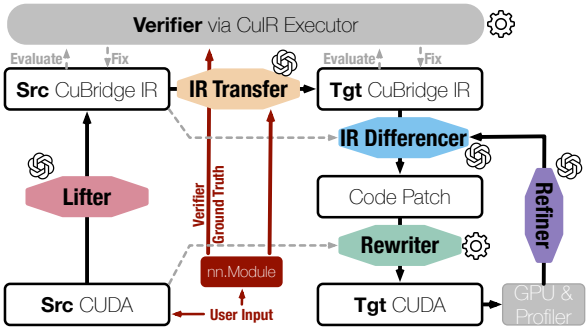

Figure 3: Overview of *CuBridge*.

Recent work has explored the use of large language models (LLMs) to automate GPU kernel generation. Benchmark studies (Ouyang et al., 2025) show that while LLM-generated CUDA kernels can perform competitively on simple operators, their correctness becomes unstable and performance degrades significantly for complex kernels such as attention, with slowdowns of up to 34.9× relative to optimized baselines. Several recent efforts explore LLM-based code generation and optimization, including multi-agent collaboration (Chen et al., 2025b; Huang et al., 2024; Zhang et al., 2024), profiler-driven feedback (Dong et al., 2025; Islam et al., 2024; Chen et al., 2024), and natural language-based planning (Zhou et al., 2025a). Despite their promise, these approaches do not yet provide a reliable way to realize expert-level execution orchestration for complex kernels.

A natural attempt to improve flexibility is to directly modify expert-written CUDA kernels using LLMs. However, as shown in Figure 2, expert kernels often rely on complex syntax and implicit asynchronous execution. Even when the intended semantic modification is clear, locating the correct code region to update can be error-prone, making direct code-level adaptation unreliable.

3 Method

3.1 Overview

We present *CuBridge*, an LLM-powered framework for adapting expert-written CUDA attention kernels to new algorithmic variants while preserving high-performance execution orchestration. Instead of generating kernels from scratch, *CuBridge* follows a lift–transfer–lower workflow centered on an executable intermediate representation, CuIR, which supports both understanding and reconstruction of

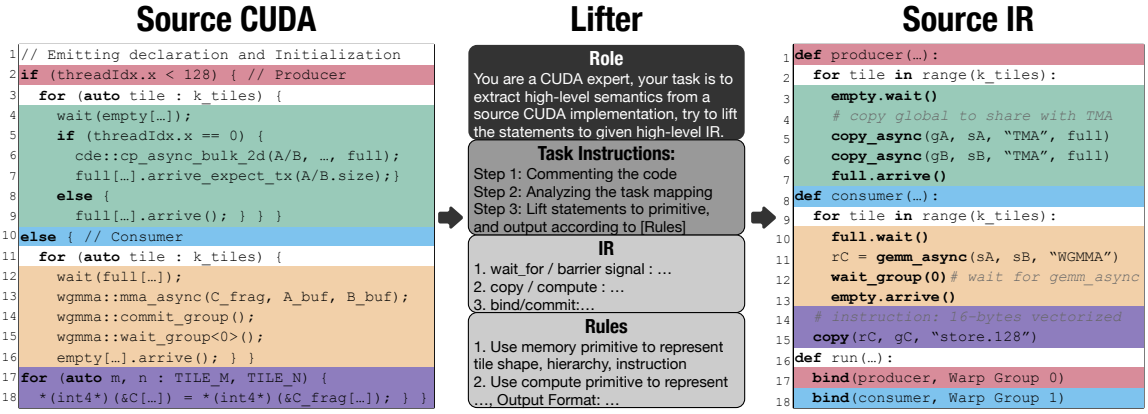

Figure 4: **The Semantic Lifting Process.** The left panel shows the complex Source CUDA, and the right panel displays the lifted Source IR, where code blocks of the same color represent mapping relationships. The center shows the modularized prompt, consisting of the CuIR documentation and the Chain-of-Thought reasoning process.

Category	Primitives	Exposed Information
Memory	`alloc`, `copy`, `copy_async`	Tile Shape, Memory Hierarchy
Compute	`gemm_async`, `gemm`	Instruction Selection, Operand
Sync.	`barrier.{wait, arrive}`	Data Dependency
Control	`bind`, `commit`	Execution Granularity

Table 1: CuIR Primitives and Exposed Hardware Semantics.

expert CUDA kernels.

As shown in Figure 3, given a high-performance source CUDA kernel and a user-provided PyTorch reference specifying target semantics, *CuBridge* proceeds in three stages. (1) **Lifting:** *CuBridge* lifts the Source kernel into CuIR, which abstracts away low-level syntactic details while retaining performance-critical execution orchestration. (2) **Transform:** guided by Target semantics, *CuBridge* generates the Target CuIR from the lifted Source CuIR, jointly reasoning about semantic correctness and performance optimization. (3) **Lowering:** *CuBridge* lowers the Target CuIR back to CUDA by differencing the Source and Target CuIR programs and generating minimal patches that update the corresponding code regions in the Source kernel to reconstruct the target kernel.

3.2 CuIR Design

To enable effective LLM-based understanding and reconstruction of expert CUDA kernels, we design CuIR, a Pythonic and executable intermediate representation that explicitly captures execution orchestration. Structurally, CuIR is expressed in Python syntax and built on a set of custom primitives that operate on data tiles as PyTorch tensors.

Design Rationale. High-performance CUDA kernels derive their efficiency from carefully structured execution orchestration, rather than from individual instructions in isolation. Accordingly, CuIR is designed to explicitly preserve three classes of information: the operations invoked by the kernel, the ordering and data dependencies among these operations, and the execution granularity and synchronization scope at which they are performed. As summarized in Table 1, Memory and Compute primitives make operations explicit, including tile-level data movement and instruction-level choices, while Synchronization and Control primitives encode dependencies, execution ordering, and collective execution granularity. These preserved elements are directly linked to correctness and performance, as GPU efficiency hinges on latency hiding and data locality. Conversely, CuIR abstracts away low-level implementation details such as fragmented thread-level indexing and verbose CUDA syntax, which are not essential to reasoning about execution structure.

Key Capabilities. CuIR provides two capabilities that are essential to *CuBridge*.

1) Kernel Understanding and Reconstructing. By making execution orchestration explicit through structured primitives, CuIR exposes the execution logic of a CUDA kernel as a composition of well-defined operations and dependencies, rather than low-level syntax. This structured representation allows an LLM to understand and reason kernel behavior, enabling semantic changes to be captured in the generated Target CuIR and localized via IR differencing for Target CUDA reconstruction.

2) Intermediate Verification. CuIR is designed to be executable and verifiable through a dedicated

IR backend executor, enabling validation of algorithmic logic and data dependencies at the IR level. To ensure semantic alignment with CUDA execution, the executor explicitly models the kernel's parallel structure and synchronization behavior. Independent parallel tasks are executed sequentially without affecting correctness, while dependent tasks are ordered to respect synchronization and data-dependence constraints. This execution model allows IR-level verification to faithfully reflect CUDA kernel behavior.

3.3 Semantic Lifting

The first stage of *CuBridge* lifts an expert-written CUDA kernel into CuIR, extracting its high-level execution orchestration while abstracting away low-level implementation complexity. As illustrated in Figure 4, the *Lifter* performs a single LLM invocation with structured chain-of-thought (CoT) reasoning, organized into three sub-tasks.

1) Syntax Annotation. This step resolves low-level syntactic ambiguity by annotating CUDA operations with explicit semantic comments. Expert CUDA kernels often rely on low-level PTX intrinsics or library-specific APIs (e.g., CuTe), whose semantics are implicit in the source code and not directly exposed. The *Lifter* augments such operations with concise semantic descriptions grounded in official CUDA and CuTe documentation, producing an annotated CUDA representation that makes the intent of low-level instructions explicit.

2) Code-to-Worker Mapping Analysis. This step identifies which code regions in the source CUDA kernel are executed by which parallel execution units, making implicit execution relationships explicit. Here, we use the term *worker* to denote a cooperative CUDA execution unit, such as a thread block, warp group, or warp. Specifically, the *Lifter* analyzes control-flow predicates over thread indices and attributes guarded code regions to the corresponding workers. For example, as shown in Figure 4 (Source CUDA, Line 2), the branch `if (threadIdx.x < 128)` partitions the kernel into two worker-aligned regions, which are later bound to different warp groups in the generated IR.

3) Primitive Lifting. In the final step, the *Lifter* translates each worker-aligned code region into a sequence of CuIR primitives and constructs the corresponding Source IR. This process recovers primitive parameters such as tile shapes, memory placement, instruction variants, and synchronization scope by analyzing index expressions, memory access patterns, and surrounding control structure. The generated CuIR program explicitly encodes the execution orchestration of the original kernel and serves as the input to subsequent transformation and lowering stages.

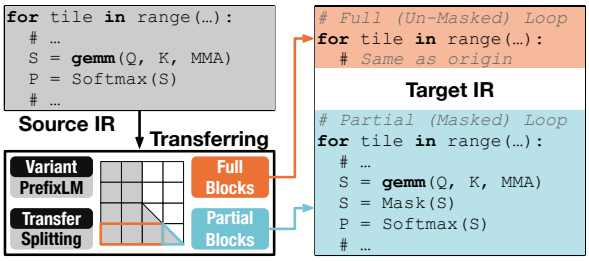

Figure 5: Performance-Aware Transformation for PrefixLM. The IR Transfer performing Loop Splitting to confine mask checks exclusively to boundary tiles.

Post-Lifting Verification & Refinement. The generated CuIR program is executed by the backend executor and its output is compared against the source CUDA within a numerical tolerance (e.g., 10^{-2} for fp16), following CUTLASS (NVIDIA, 2025a). On mismatch, the executor log is used to diagnose errors such as incorrect tile parameters or missing synchronization, and guides the regeneration of the IR. This closed-loop refinement ensures that the final Source IR faithfully preserves the source kernel semantics.

3.4 Semantic Transformation

In this stage, the *IR Transfer* transforms the lifted Source IR into a Target IR that implements the target operator semantics specified by a user-provided PyTorch reference, while enabling the performance-critical execution orchestration.

1) Semantic Alignment. The *IR Transfer* first analyzes the PyTorch reference to identify the semantic difference between source IR and target operator. Guided by this gap, it maps the required semantics to CuIR primitives and determines where the source orchestration must be extended or modified. In this way, the generated Target IR program matches the target computation while remaining structurally aligned with the Source IR.

2) Performance-Aware Transformation. A semantics-correct Target IR can still be inefficient. To maintain efficiency, the *IR Transfer* further analyzes the performance implications of the required semantic changes during Target IR generation. For PrefixLM (Pal et al., 2023), the validity of the attention mask correlates with the nested loop iteration space: element-wise checks are required only for partial blocks, whereas other blocks are fully valid.

As shown in Figure 5, the *IR Transfer* applies loop splitting to construct a check-free *Full Loop* for full blocks and a *Partial Loop* that applies masking for partial blocks, thereby avoiding unnecessary element-wise checking in the full-loop region.

Post-Transformation Verification. The generated Target IR is executed by the CuIR backend executor and validated against the user-provided PyTorch reference implementation, using the same verification procedure as in Sec. 3.3. Any mismatch triggers iterative refinement.

3.5 Kernel Reconstruction

The final stage lowers the transformed Target IR into high-performance Target CUDA code. *CuBridge* reconstructs the target implementation by applying localized, IR-guided code patches within a structured ReAct workflow (Yao et al., 2023).

1) Differential Analysis. Reconstruction begins by comparing the Source IR and the Target IR to identify semantic differences introduced during transformation. Leveraging the region-level correspondence established in the lifting stage, these IR-level changes are traced back to their corresponding locations in the Source CUDA code, precisely identifying the code regions that require modification.

2) Reference-Guided Lowering. To translate the Target CuIR program into CUDA code, *CuBridge* performs reference-guided lowering by using the *Source CUDA* kernel as an implementation reference for the Source IR. This reference specifies how abstract IR primitives should be concretely realized in CUDA, ensuring consistency with the original kernel's implementation style. The lowering process focuses on two tasks:

- **Primitive Lowering:** Lowering abstract IR primitives (e.g., `copy_async`) to concrete hardware intrinsics (e.g., `cp.async.ca`) used in the source kernel.
- **Index Mapping:** Expanding tile-level operations into thread-level indexing that follows the data layout and indexing patterns in the source kernel.

3) Iterative Patching. Finally, *the Rewriter* reconstructs the target CUDA kernel by applying localized and minimal code patches to the source implementation. These updates are realized through a line-level editing action (`Edit_Line`) that supports insertion, deletion, and modification of code lines within identified regions. By restricting changes to affected code segments, this patch-based reconstruction ensures faithful alignment between the Target IR and CUDA code, while avoiding the context-length limitations of full-file rewriting.

4 Experiment

4.1 Experiment Setup

We evaluate *CuBridge* along three aspects: (1) end-to-end kernel correctness and performance across representative model configurations, (2) robustness across diverse attention semantics, including previously unsupported variants, and (3) the impact of key design choices via ablation studies.

GPU platforms. All experiments are conducted on NVIDIA GPUs from two widely deployed architectures: A100 (Ampere) (NVIDIA, 2021) and H100 (Hopper) (NVIDIA, 2023).

LLM Backends. We evaluate *CuBridge* using diverse SOTA LLMs, including the closed-source models of GPT-5 (OpenAI, 2025), Claude-3.5-Sonnet (Anthropic, 2025), as well as open-source models of DeepSeek-V3 (DeepSeek-AI, 2024), Qwen-3-235B (Team, 2025), Qwen-3-32B (Team, 2025).

Attention Variants. We focus our evaluation on attention mechanisms that are currently not supported in the standard FlashAttention (Dao et al., 2022; Dao, 2024; Shah et al., 2024) library. Our test suite encompasses a diverse set of variants, including PrefixLM (Pal et al., 2023), Global Sliding Window (Zaheer et al., 2020), Share Question Mask (Ouyang et al., 2022), Causal Blockwise Mask (Bertsch et al., 2025), Relative Position Embeddings (Shaw et al., 2018), ReLU Attention (Dao et al., 2022), and Sigmoid Attention (Ramapuram et al., 2025). To assess robustness under unseen compositions, we additionally evaluate a composite variant that combines PrefixLM, Softcap, and Sigmoid attention. All hyper-parameters follow the configurations reported in the original works.

Comparison Baselines. *CuBridge* adopts FlashAttention v2.8.0 (Dao, 2024) as the source expert kernel reference. We compare CuBridge with three representative state-of-the-art baselines representing different paradigms currently explored in the literature: (1) PyTorch (Paszke et al., 2019) attention implementations built from standard operators; (2) FlexAttention (Guessous et al., 2024), a compiler-based framework for attention generation; and (3) Qimeng-Attention (Zhou et al., 2025a), an LLM-based attention kernel generation approach. Additionally, we compare *CuBridge* against FlashInfer (Ye et al., 2025), a high-performance expert-hand-tuned library.

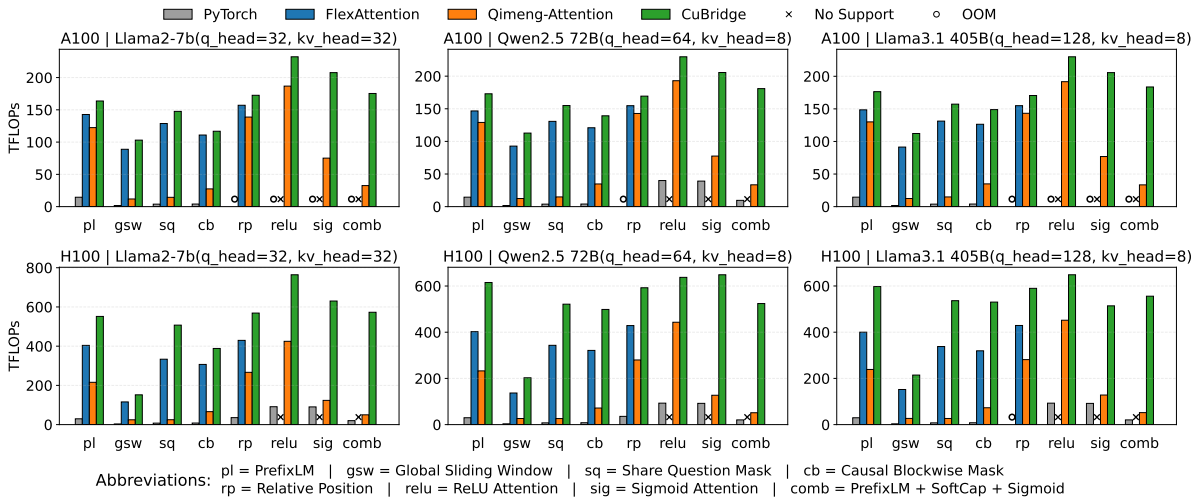

Figure 6: End-to-end performance comparison across attention variants and GPU platforms.

Benchmark Configurations. We benchmark attention configurations derived from three representative real-world LLMs: Llama2-7B (MHA, 32/32/128) (Touvron et al., 2023), Qwen2.5-72B (GQA, 64/8/128) (Yang et al., 2025), and Llama3.1-405B (GQA, 128/8/128) (Team, 2024), where the tuple denotes (query heads / key-value heads / head dimension). Sequence length varies over $\{1k, 2k, 4k, 8k\}$, with batch size adjusted to maintain a constant total of 16k tokens per batch (Dao et al., 2022). For all LLM-driven approaches, we set the generation temperature to zero (Ouyang et al., 2025) and adopt a best-of-k strategy with $k = 10$ (Stiennon et al., 2020), reporting the best achieved performance.

4.2 Overall Performance

Figure 6 compares the performance of *CuBridge* against all baselines on NVIDIA A100 and H100 GPUs. We report results at sequence length $N = 8k$ across diverse attention variants and model configurations. Due to space constraints, the full results are provided in Appendix B. Both *CuBridge* and the Qimeng-Attention baseline use GPT-5 as the underlying LLM backend.

Across all evaluated tasks, *CuBridge* supports all tested attention variants, achieves 100% correctness, and consistently outperforms all state-of-the-art baselines. On A100, *CuBridge* achieves average speedups of $12.69\times$, $1.18\times$ and $2.54\times$ over PyTorch, FlexAttention, and Qimeng-Attention, respectively; on H100, these speedups further increase to $19.82\times$, $1.62\times$, $4.35\times$. Beyond these generative baselines, we also benchmark *CuBridge* against the hand-optimized FlashInfer library. *CuBridge* achieves comparable performance to FlashInfer on its natively supported variants (average $1.07\times$) and substantial speedups (average $3.49\times$) on variants beyond FlashInfer's native support. Detailed results and per-variant comparisons are provided in Appendix C.

Compared to PyTorch, *CuBridge* achieves significantly higher performance. PyTorch realizes attention variants by composing multiple fine-grained operators into separate CUDA kernels, leading to frequent kernel launches, redundant global memory accesses, and occasional out-of-memory failures. By generating a single fused kernel that integrates the full attention computation, *CuBridge* avoids these overheads and enables efficient data reuse.

Compared to FlexAttention, *CuBridge* demonstrates superior performance while supporting a broader range of attention variants. As shown in Figure 6, *CuBridge* outperforms FlexAttention across all evaluated model configurations, with speedups of $1.05\times$–$1.22\times$ on A100 and $1.26\times$–$1.66\times$ on H100. Notably, the performance gap widens on H100, indicating that *CuBridge* better adapts to newer GPU architectures. This advantage stems from preserving expert-optimized, hardware-specific execution orchestration. Beyond performance, *CuBridge* supports a broader set of attention variants that are not covered by FlexAttention, including ReLU Attention, Sigmoid Attention, and complex composite patterns, while maintaining high performance.

Compared to Qimeng-Attention, *CuBridge* consistently achieves higher performance across attention variants of varying complexity. On A100, *CuBridge* attains speedups ranging from $1.18\times$ to

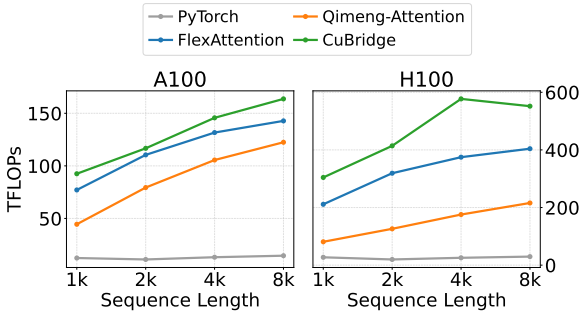

Figure 7: Performance comparison for the PrefixLM variant (Llama2-7B config) across different sequence lengths on A100 (left) and H100 (right) GPUs.

10.56×, while on H100 the speedups further increase to 1.43×–20.3×. As shown in Figure 6, on simpler variants such as relative position attention, *CuBridge* achieves performance comparable to Qimeng-Attention on A100 (average 1.24× speedup). In contrast, for variants with complex masking or irregular execution patterns (e.g., `share_question_mask` and `comb`), *CuBridge* delivers substantially larger gains, reaching 5.37× on A100. For the same `comb` variants, speedups further increase to 11.47× on H100, indicating that *CuBridge* better exploits the execution capabilities of newer GPU architectures. By examining the generated kernels, we observe that while Qimeng-Attention introduces limited performance optimization, it does not construct explicit warp-specialized and tensor/cuda-core overlap execution orchestration. These results suggest that relying solely on the LLM's internal knowledge makes it challenging to perform efficient execution orchestration for complex tasks, leading to the generated kernels only functional but inefficient.

Figure 7 reports performance across varying sequence lengths for the PrefixLM variant under the Llama2-7B configuration. Across all evaluated sequence lengths, *CuBridge* consistently achieves higher throughput than SOTA baselines, indicating robust scalability with respect to sequence length.

4.3 Ablation Study on System Components

To validate the effectiveness of our system components, we conduct an ablation study on the task of rewriting expert-written FlashAttention kernels into target attention variants, guided by a user-provided PyTorch reference. Experiments are conducted on H100 across 96 test cases (8 variants × 12 sequence lengths). We compare three methods: (1) Vanilla GPT-5, which performs single-shot code rewriting of the source kernel based solely on the PyTorch reference; (2) GPT-5 + ReAct, which applies iterative reasoning and code editing without CuIR; and (3) *CuBridge*. We report Pass@k (whether at least one of k samples passes correctness checking) and the geometric mean speedup of successful cases, normalized to Vanilla GPT-5.

Table 2 presents the results. Vanilla GPT-5 exhibits low success rates, which saturates quickly even with increased sampling (Pass@5 reaches only 0.38). Adding ReAct reasoning improves correctness (Pass@1: 0.21 → 0.41), but the gain plateaus as k increases and yields only limited speedup (1.23×). In contrast, *CuBridge* achieves perfect correctness (Pass@5 = 1.00), and substantially higher performance, with a 4.19× speedup.

This confirms that the structured IR provides essential support for reliable code modification. By lifting the source CUDA to the CuIR level, *CuBridge* makes execution orchestration explicit and enables structured transformation and verification before lowering, leading to more stable correctness and higher performance.

Method	Pass@1	Pass@3	Pass@5	Norm. Speedup
GPT-5	0.21	0.33	0.38	1.00×
GPT-5 + ReAct	0.41	0.54	0.58	1.23×
GPT-5 + CuBridge	**0.70**	**0.85**	**1.00**	**4.19×**

Table 2: Comparison of correctness and performance for different llm-based kernel modification methods across 96 test cases on H100. Performance is reported as the geometric mean of speedup normalized to the Vanilla GPT-5 baseline.

4.4 Impact of Different LLM Backends

To evaluate the generality of *CuBridge*, we benchmark it using five representative LLM backends: GPT-5, Claude, DeepSeek-V3, Qwen-3-235B and Qwen-3-32B. Table 3 reports the achieved throughput (TFLOPS) on the H100 GPU for the PrefixLM task under the Llama2-7B attention configuration.

The results demonstrate that *CuBridge* achieves stable, expert-level performance once the backend reaches a baseline CUDA reasoning capability. Specifically, substituting GPT-5 with Claude, DeepSeek-V3, or Qwen-3-235B results in less than 5% performance variation across all sequence lengths. This indicates that our throughput gains primarily stem from the structured lift–transfer–lower workflow and CuIR alignment, rather than reliance on a specific model.

However, we observe a distinct capability threshold for code reconstruction. While the larger models succeed, Qwen-3-32B fails to produce valid ker-

nels. This suggests that reliable low-level CUDA generation requires a baseline level of reasoning ability. While different frameworks may vary in robustness, insufficient model capacity can become a limiting factor for complex kernel generation tasks.

LLM Backends	Seq=1k	Seq=2k	Seq=4k	Seq=8k
GPT-5	**304.35**	426.82	**577.03**	551.73
Claude	292.87	**428.64**	562.91	**569.02**
DeepSeek-V3	294.12	424.05	557.03	549.73
Qwen-3-235B	295.04	421.63	558.74	542.61
Qwen-3-32B	N/A	N/A	N/A	N/A

Table 3: Performance comparison (TFLOPS) across different LLM backends on H100 (PrefixLM, Llama2-7b attention config).

5 Conclusion

We present *CuBridge*, an LLM-based framework for efficiently adapting expert-written attention kernels to new semantic variants. *CuBridge* is built on CuIR, an executable intermediate representation that makes execution orchestration explicit while abstracting low-level CUDA details, and uses a structured lift–transfer–lower workflow for reliable kernel adaptation. Across diverse real-world attention variants on A100 and H100 GPUs, *CuBridge* consistently produces correct kernels and significantly outperforms general frameworks, template-based compilers, and prior LLM-based approaches.

Limitations

Our work has two main limitations. First, *CuBridge* relies on the availability of high-quality expert kernels as source references. While such optimized implementations are widely available in the NVIDIA platform, they are often lacking on non-mainstream customized hardware (e.g., FPGAs). This may limit the effectiveness of our framework in those environments. Second, our current evaluation focuses primarily on attention variants, as they highlight the complexity of execution orchestration. We have not yet extended our validation to other high-performance computing tasks, such as scientific computing, which remains a promising direction for future work.

Acknowledgements

This work was supported by the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (No. JYB2025XDXM113), the National Natural Science Foundation of China (Nos. 62532006, 62572300), the Shanghai Qi Zhi Institute Innovation Program (SQZ202316), the Ministry of Education, Singapore (Nos. MOE-T2EP20124-0017, MOET32020-0004), and the CyberSG R&D Cyber Research Programme Office (A-8002767-00-00). Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

References

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, and 3 others. 2016. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 265–283. USENIX Association.

Anthropic. 2025. Claude 3.7 sonnet and claude code.

Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. 2025. Kevin: Multi-turn RL for generating CUDA kernels. *CoRR*, abs/2507.11948.

Michael Bauer, Sean Treichler, and Alex Aiken. 2014. Singe: leveraging warp specialization for high performance on gpus. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*, pages 119–130. ACM.

Amanda Bertsch, Maor Ivgi, Emily Xiao, Uri Alon, Jonathan Berant, Matthew R. Gormley, and Graham Neubig. 2025. In-context learning with long-context models: An in-depth exploration. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL 2025 - Volume 1: Long Papers, Albuquerque, New Mexico, USA, April 29 - May 4, 2025*, pages 12119–12149. Association for Computational Linguistics.

Feiyang Chen, Yu Cheng, Lei Wang, Yuqing Xia, Ziming Miao, Lingxiao Ma, Fan Yang, Jilong Xue, Zhi Yang, Mao Yang, and Haibo Chen. 2025a. Attentionengine: A versatile framework for efficient attention mechanisms on diverse hardware platforms. *CoRR*, abs/2502.15349.

Terry Chen, Bing Xu, and Kirthi Devleker. Automating GPU kernel generation with DeepSeek-R1 and inference-time scaling.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan,

Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: an automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 578–594. USENIX Association.

Wentao Chen, Jiace Zhu, Qi Fan, Yehan Ma, and An Zou. 2025b. CUDA-LLM: llms can write efficient CUDA kernels. *CoRR*, abs/2506.09092.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024. Teaching large language models to self-debug. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

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, Daniel Y. 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 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

DeepSeek-AI. 2024. Deepseek-v3 technical report. *CoRR*, abs/2412.19437.

Juncheng Dong, Yang Yang, Tao Liu, Yang Wang, Feng Qi, Vahid Tarokh, Kaushik Rangadurai, and Shuang Yang. 2025. STARK: strategic team of agents for refining kernels. *CoRR*, abs/2510.16996.

Driss Guessous, Yanbo Liang, Joy Dong, and Horace He. 2024. Flexattention: The flexibility of pytorch with the performance of flashattention. https://pytorch.org/blog/flexattention/. PyTorch Blog.

Cong Guo, Yangjie Zhou, Jingwen Leng, Yuhao Zhu, Zidong Du, Quan Chen, Chao Li, Bin Yao, and Minyi Guo. 2020. Balancing efficiency and flexibility for dnn acceleration via temporal gpu-systolic array integration. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE.

Weiming Hu, Zihan Zhang, Haoyan Zhang, Chen Zhang, Cong Guo, Yu Feng, Tianchi Hu, Guanglin Li, Guipeng Hu, Junsong Wang, and 1 others. 2026. M2xfp: A metadata-augmented microscaling data format for efficient low-bit quantization. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 1151–1167.

Dong Huang, Jie M. Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. 2024. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *Preprint*, arXiv:2312.13010.

Ziyu Huang, Yangjie Zhou, Zihan Liu, Xinhao Luo, Yijia Diao, Minyi Guo, Jidong Zhai, Yu Feng, Chen Zhang, Anbang Wu, and Jingwen Leng. 2026. Flashfuser: Expanding the scale of kernel fusion for compute-intensive operators via inter-core connection. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2026, Sydney, Australia, January 31 - Feb. 4, 2026*, pages 1–14. IEEE.

Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. MapCoder: Multi-agent code generation for competitive problem solving. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4912–4944, Bangkok, Thailand. Association for Computational Linguistics.

Changxin Ke, Rui Zhang, Shuo Wang, Li Ding, Guangli Li, Yuanbo Wen, Shuoming Zhang, Ruiyuan Xu, Jin Qin, Jiaming Guo, Chenxi Wang, Ling Li, Qi Guo, and Yunji Chen. 2025. Mutual-supervised learning for sequential-to-parallel code translation. *CoRR*, abs/2506.11153.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Yacine Jernite, Margaret Mitchell, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. 2023. The stack: 3 TB of permissively licensed source code. *Trans. Mach. Learn. Res.*, 2023.

Jianling Li, Shangzhan Li, Zhenye Gao, Qi Shi, Yuxuan Li, Zefan Wang, Jiacheng Huang, WangHaojie WangHaojie, Jianrong Wang, Xu Han, Zhiyuan Liu, and Maosong Sun. 2025a. Tritonbench: Benchmarking large language model capabilities for generating triton operators. In *Findings of the Association for Computational Linguistics, ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pages 23053–23066. Association for Computational Linguistics.

Xiaoya Li, Xiaofei Sun, Albert Wang, Jiwei Li, and Chris Shum. 2025b. CUDA-L1: improving CUDA optimization via contrastive reinforcement learning. *CoRR*, abs/2507.14111.

Zihan Liu, Xinhao Luo, Junxian Guo, Wentao Ni, Yangjie Zhou, Yue Guan, Cong Guo, Weihao Cui, Yu Feng, Minyi Guo, Yuhao Zhu, Minjia Zhang, Chen Jin, and Jingwen Leng. 2025. VQ-LLM: high-performance code generation for vector quantization augmented LLM inference. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2025, Las Vegas, NV, USA, March 1-5, 2025*, pages 1496–1509. IEEE.

David P. Luebke. 2008. CUDA: scalable parallel programming for high-performance scientific computing. In *Proceedings of the 2008 IEEE International Symposium on Biomedical Imaging: From Nano to Macro, Paris, France, May 14-17, 2008*, pages 836–838. IEEE.

Xinhao Luo, Zihan Liu, Yangjie Zhou, Shihan Fang, Ziyu Huang, Yu Feng, Chen Zhang, Shixuan Sun,

Zhenzhe Zheng, Jingwen Leng, and Minyi Guo. 2025. Clusterfusion: Expanding operator fusion scope for LLM inference via cluster-level collective primitive. *CoRR*, abs/2508.18850.

Meta PyTorch. 2025. KernelAgent: Autonomous GPU kernel generation via deep agents. https://github.com/meta-pytorch/KernelAgent.

NVIDIA. 2021. Nvidia a100 tensor core gpu architecture. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf.

NVIDIA. 2023. Nvidia h100 tensor core gpu architecture. Technical report.

NVIDIA. 2025a. CUDA Templates for Linear Algebra Subroutines. https://github.com/NVIDIA/cutlass.

NVIDIA. 2025b. NVIDIA CUDA Compiler Driver NVCC. https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/.

Tesla NVIDIA. 2017. Nvidia tesla v100 gpu architecture.

OpenAI. 2025. Introducing gpt-5.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. 2025. Kernelbench: Can llms write efficient GPU kernels? In *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*. OpenReview.net.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Kuntal Kumar Pal, Kazuaki Kashihara, Ujjwala Anantheswaran, Kirby C. Kuznia, Siddhesh Jagtap, and Chitta Baral. 2023. Exploring the limits of transfer learning with unified model in the cybersecurity domain. *CoRR*, abs/2302.10346.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, and 1 others. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32.

Jason Ramapuram, Federico Danieli, Eeshan Gunesh Dhekane, Floris Weers, Dan Busbridge, Pierre Ablin, Tatiana Likhomanenko, Jagrit Digani, Zijin Gu, Amitis Shidani, and Russell Webb. 2025. Theory, analysis, and best practices for sigmoid self-attention. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Morgane Rivière, Shreya Pathak, Pier Giuseppe Sessa, and 1 others. 2024. Gemma 2: Improving open language models at a practical size. *arXiv preprint arXiv:2408.00118*.

Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. 2024. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*.

Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. 2018. Self-attention with relative position representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 2 (Short Papers)*, pages 464–468. Association for Computational Linguistics.

Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel M. Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F. Christiano. 2020. Learning to summarize with human feedback. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Llama Team. 2024. The llama 3 herd of models. *CoRR*, abs/2407.21783.

Qwen Team. 2025. Qwen3 technical report. *CoRR*, abs/2505.09388.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, and 49 others. 2023. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008.

Guoxia Wang, Jinle Zeng, Xiyuan Xiao, Siming Wu, Jiabin Yang, Lujing Zheng, Zeyu Chen, Jiang Bian,

Dianhai Yu, and Haifeng Wang. 2025. Flashmask: Efficient and rich mask extension of flashattention. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Anjiang Wei, Tianran Sun, Yogesh Seenichamy, Hang Song, Anne Ouyang, Azalia Mirhoseini, Ke Wang, and Alex Aiken. 2025. Astra: A multi-agent system for GPU kernel performance optimization. *CoRR*, abs/2509.07506.

An Yang, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoyan Huang, Jiandong Jiang, Jianhong Tu, Jianwei Zhang, Jingren Zhou, Junyang Lin, Kai Dang, Kexin Yang, Le Yu, Mei Li, Minmin Sun, Qin Zhu, Rui Men, Tao He, and 9 others. 2025. Qwen2.5-1m technical report. *CoRR*, abs/2501.15383.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. 2025. Flashinfer: Efficient and customizable attention engine for LLM inference serving. *CoRR*, abs/2501.01005.

Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontañón, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. 2020. Big bird: Transformers for longer sequences. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Huan Zhang, Wei Cheng, Yuhan Wu, and Wei Hu. 2024. A pair programming framework for code generation via multi-plan exploration and feedback-driven refinement. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, page 1319–1331, New York, NY, USA. Association for Computing Machinery.

Zijian Zhang, Rong Wang, Shiyang Li, Yuebo Luo, Mingyi Hong, and Caiwen Ding. 2025. Cudaforge: An agent framework with hardware feedback for CUDA kernel optimization. *CoRR*, abs/2511.01884.

Qirui Zhou, Shaohui Peng, Weiqiang Xiong, Haixin Chen, Yuanbo Wen, Haochen Li, Ling Li, Qi Guo, Yongwei Zhao, Ke Gao, Ruizhi Chen, Yanjun Wu, Zhao Chen, and Yunji Chen. 2025a. Qimeng-attention: SOTA attention operator is generated by SOTA attention algorithm. In *Findings of the Association for Computational Linguistics, ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pages 8491–8505. Association for Computational Linguistics.

Yangjie Zhou, Mengtian Yang, Cong Guo, Jingwen Leng, Yun Liang, Quan Chen, Minyi Guo, and Yuhao Zhu. 2021. Characterizing and demystifying the implicit convolution algorithm on commercial matrix-multiplication accelerators. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*, pages 214–225. IEEE.

Yangjie Zhou, Honglin Zhu, Qian Qiu, Weihao Cui, Zihan Liu, Peng Chen, Mohamed Wahib, Cong Guo, Siyuan Feng, Jintao Meng, and 1 others. 2025b. A sample-free compilation framework for efficient dynamic tensor computation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 167–184.

Xinguo Zhu, Shaohui Peng, Jiaming Guo, Yunji Chen, Qi Guo, Yuanbo Wen, Hang Qin, Ruizhi Chen, Qirui Zhou, Ke Gao, and 1 others. 2025. Qimeng-kernel: Macro-thinking micro-coding paradigm for llm-based high-performance gpu kernel generation. *arXiv preprint arXiv:2511.20100*.

A Definitions of Attention Variants

In this section, we provide the mathematical definitions and configurations for the attention variants used in our evaluation. To encompass various modifications to the scoring logic and normalization, we formulate the general attention computation as:

$$O = \textbf{Norm}\left(\textbf{Mask}\left(\textbf{Score}\left(\frac{QK^T}{\sqrt{d}}\right)\right)\right)V \quad (3)$$

where $Q, K, V \in \mathbb{R}^{L\times d}$. The components are defined as follows:

- **Score**$(\cdot)$: Adjustments on raw attention scores (e.g., relative bias, softcapping).
- **Mask**$(\cdot)$: Visibility constraints (e.g., causal, sliding window).
- **Norm**$(\cdot)$: Normalization functions (e.g., Softmax, ReLU).

Unless otherwise specified, for the mask M, $M_{ij} = 0$ indicates visible, and $M_{ij} = -\infty$ indicates masked.

A.1 Score Modifications

Adjustments applied before masking. In the formulas below, $S_{ij} = q_i k_j^T/\sqrt{d}$ represents the raw attention score.

Relative Position Embeddings (Shaw et al., 2018) Injects positional information by directly adding the relative index difference between the query (i) and key (j) to the score.

$$\textbf{Score}(S_{ij}) = S_{ij} + (i - j)$$

Softcap Attention (Rivière et al., 2024) Applies a 'tanh' cap with a scaling factor C to stabilize logits.

$$\textbf{Score}(S_{ij}) = C \cdot \tanh\left(\frac{S_{ij}}{C}\right)$$

A.2 Mask Variants

These variants define the visibility pattern M_{ij} between query i and key j.

PrefixLM (Pal et al., 2023) Combines bidirectional attention for the prefix (length L_p) and causal attention for generated tokens.

$$M_{ij} = \begin{cases} 0 & \text{if } j \le L_p \\ & \text{(Prefix visibility)} \\ 0 & \text{if } L_p < j \le i \\ & \text{(Causal generation)} \\ -\infty & \text{otherwise} \end{cases}$$

Global Sliding Window (Zaheer et al., 2020) Tokens attend to neighbors within a window w and specific global tokens $(\mathcal{G})$.

$$M_{ij} = \begin{cases} 0 & \text{if } |i - j| \le w \\ 0 & \text{if } i \in \mathcal{G} \text{ or } j \in \mathcal{G} \\ -\infty & \text{otherwise} \end{cases}$$

Share Question Mask (Ouyang et al., 2022) Commonly used in Reward Models and DPO, this mask allows multiple candidate responses (sets A_k) to attend to a single shared prompt (Q) to eliminate redundant computation.

$$M_{ij} = \begin{cases} 0 & \text{if } i, j \in Q \text{ and } j \le i \\ & \text{(Prompt self-attn)} \\ 0 & \text{if } i \in A_k, j \in Q \cup A_k \text{ and } j \le i \\ & \text{(Response} \to \text{Prompt + Self)} \\ -\infty & \text{otherwise} \end{cases}$$

Causal Blockwise Mask (Bertsch et al., 2025) Isolates demonstration examples based on document IDs (D) while granting the test query (starting at L_q) visibility over the full context.

$$M_{ij} = \begin{cases} 0 & \text{if } j \le i \text{ and } D_i = D_j \\ & \text{(Demonstration isolation)} \\ 0 & \text{if } j \le i \text{ and } i \ge L_q \\ & \text{(Test query visibility)} \\ -\infty & \text{otherwise} \end{cases}$$

A.3 Normalization Variants

Replacements for the standard Softmax function. In these variants, normalization logic often uses sequence length L for scaling.

ReLU Attention (Dao et al., 2022) Applies ReLU activation and scales by sequence length.

$$\textbf{Norm}(S) = \frac{1}{L}\,\text{ReLU}(S)$$

Sigmoid Attention (Ramapuram et al., 2025) Applies Sigmoid activation and scales by sequence length.

$$\textbf{Norm}(S) = \frac{1}{L}\,\text{Sigmoid}(S)$$

Task Variant	Method	Llama2-7b ($q=32, k=32$) 1k	2k	4k	8k	Qwen2.5 72B ($q=64, k=8$) 1k	2k	4k	8k	Llama3.1 405B ($q=128, k=8$) 1k	2k	4k	8k
PrefixLM	Torch	12.36	10.99	13.12	14.56	14.58	10.88	13.08	14.51	14.64	10.93	13.03	14.53
	FlexAttention	77.09	110.44	131.67	142.77	99.82	121.96	137.52	146.68	107.89	128.63	140.49	148.55
	Qimeng Attn	44.46	79.37	105.55	122.44	72.37	99.92	120.37	129.07	76.01	101.05	118.83	130.03
	CuBridge	**92.50**	**116.69**	**145.69**	**163.70**	**103.61**	**138.70**	**159.97**	**172.98**	**122.22**	**149.36**	**166.34**	**176.39**
Global Sliding Window	Torch	10.61	5.04	3.15	1.72	12.74	5.14	3.11	1.72	12.82	5.16	3.14	1.71
	FlexAttention	56.96	**85.24**	87.67	88.87	92.52	92.09	92.32	92.78	96.40	94.23	92.47	91.40
	Qimeng Attn	29.82	28.11	19.88	11.86	47.55	34.92	22.14	12.43	49.95	35.44	21.81	12.46
	CuBridge	**67.36**	80.77	**98.56**	**103.05**	**94.80**	**102.50**	**110.08**	**112.75**	**110.95**	**111.42**	**113.23**	**112.23**
Share Question Mask	Torch	5.37	3.72	3.52	3.84	5.67	3.69	3.80	3.82	6.19	3.71	3.81	3.83
	FlexAttention	29.86	84.32	108.10	128.78	64.44	92.66	112.75	130.71	76.01	95.16	116.07	131.24
	Qimeng Attn	12.27	14.36	14.54	14.43	15.77	16.61	15.60	14.88	18.54	16.90	15.65	14.88
	CuBridge	**39.81**	**88.62**	**122.31**	**147.61**	**73.96**	**107.88**	**135.07**	**154.98**	**90.18**	**116.09**	**138.86**	**157.26**
Causal Blockwise Mask	Torch	4.85	3.67	3.89	4.00	6.31	3.72	3.86	3.97	6.33	3.75	3.89	3.98
	FlexAttention	31.77	**62.96**	**87.36**	110.96	**59.41**	79.85	100.77	120.91	70.30	87.45	108.47	126.37
	Qimeng Attn	16.91	25.43	30.93	27.49	26.73	33.81	34.98	34.75	32.58	34.43	34.75	34.90
	CuBridge	**35.12**	54.27	86.21	**116.82**	57.67	**85.10**	**114.39**	**139.24**	**76.49**	**102.34**	**127.40**	**148.91**
Relative Pos.	Torch	14.70	13.72	13.82	OOM	16.27	13.70	15.90	OOM	16.34	13.61	15.82	OOM
	FlexAttention	**114.85**	144.68	149.75	157.08	131.96	142.19	152.02	154.73	134.26	147.91	152.28	154.91
	Qimeng Attn	71.05	112.10	130.56	138.79	105.91	126.59	139.74	142.81	109.04	127.91	135.62	143.22
	CuBridge	105.80	**151.28**	**162.47**	**172.53**	**137.92**	**154.96**	**166.21**	**169.39**	**146.46**	**161.65**	**167.97**	**170.50**
ReLU Attn	Torch	31.16	36.59	33.63	OOM	32.35	36.44	38.28	39.98	33.79	37.38	39.73	OOM
	FlexAttention	no support											
	Qimeng Attn	88.56	142.67	140.83	186.76	134.64	168.17	186.99	193.01	137.73	168.23	182.67	191.71
	CuBridge	**150.88**	**203.30**	**219.46**	**232.19**	**181.02**	**204.08**	**225.00**	**229.70**	**192.73**	**216.88**	**225.96**	**229.92**
Sigmoid Attn	Torch	30.48	35.99	31.95	OOM	30.94	35.80	37.46	39.14	33.37	36.70	38.91	OOM
	FlexAttention	no support											
	Qimeng Attn	45.05	62.92	66.33	75.16	54.36	71.15	74.58	77.51	64.54	71.05	74.98	76.87
	CuBridge	**134.56**	**183.33**	**196.13**	**207.63**	**165.47**	**185.36**	**201.14**	**205.59**	**173.81**	**195.33**	**201.92**	**205.62**
Combo (PrefixLM+ Softcap+ Sigmoid)	Torch	8.33	9.41	9.46	OOM	9.11	9.38	9.45	9.62	9.18	9.50	9.63	OOM
	FlexAttention	no support											
	Qimeng Attn	17.78	26.12	29.39	32.60	25.46	29.37	31.83	33.44	26.90	29.98	32.13	33.39
	CuBridge	**80.03**	**120.02**	**155.65**	**175.34**	**106.20**	**144.75**	**167.16**	**180.89**	**124.73**	**154.32**	**172.60**	**183.61**

Table 4: Performance benchmark (TFLOPS) of attention variants across varying sequence lengths and real-world model attention configurations on NVIDIA A100. (For each backbone the best result are marked in bold, and the second best result are underlined.)

B Detailed Experimental Results

This section provides the complete numerical data for the performance benchmarks discussed in Section 4. Table 4 and Table 5 detail the TFLOPS achieved by *CuBridge* and baselines across all evaluated attention variants, sequence lengths, and model attention configurations on NVIDIA A100 and H100 GPUs, respectively.

C Comparison With Expert Libraries

We compare *CuBridge* with FlashInfer(v0.6.0) (Ye et al., 2025) to evaluate its performance relative to expert-level manual optimization. FlashInfer is widely recognized as a performance benchmark due to its expert-optimized, hand-tuned kernels.

C.1 Experiment Setup

FlashInfer provides three distinct interfaces for implementing attention variants: (1) Native APIs: Hand-optimized kernels for four built-in variants (*Causal, Sliding Window, ALiBi, and Softcap*). (2) Block-Sparse (BSR) Wrapper: A block-sparse mask API (`BlockSparseAttentionWrapper`) that enables sparse mask but does not support score modification or non-Softmax normalization. (3) JIT Templating: A Python JIT compilation API (`gen_customize_batch_prefill_module`) that allows users to inject custom C++ attention logic while keeping the execution schedule fixed.

All experiments were conducted on an NVIDIA H100 GPU using the Llama 3.1-405B attention configuration (Q/KV heads: 128/8, Head dim: 128). We evaluate sequence lengths ranging from 1k to 8k, with the batch size dynamically adjusted to maintain a constant total of 16k tokens per batch.

For each evaluated variant, we benchmark all applicable FlashInfer interfaces and report the best FlashInfer performance (the maximum TFLOPS achieved among the available implementations) to

Task Variant	Method	Llama2-7b ($q=32, k=32$) 1k	2k	4k	8k	Qwen2.5 72B ($q=64, k=8$) 1k	2k	4k	8k	Llama3.1 405B ($q=128, k=8$) 1k	2k	4k	8k
PrefixLM	Torch	27.22	19.72	25.48	29.61	28.40	19.99	25.67	29.77	29.03	20.06	25.67	29.66
	FlexAttention	211.17	319.03	374.71	404.09	248.79	333.70	382.13	402.38	280.66	335.76	377.96	400.05
	Qimeng Attn	81.07	126.08	175.60	215.75	113.24	163.79	207.47	232.61	124.68	173.70	214.14	238.76
	CuBridge	**304.35**	**414.05**	**577.03**	**551.73**	**304.35**	**486.19**	**599.14**	**615.26**	**383.51**	**548.49**	**532.16**	**597.61**
Global Sliding Window	Torch	23.65	9.25	6.00	3.40	24.70	9.36	6.05	3.41	25.23	9.40	6.05	3.41
	FlexAttention	109.37	111.62	113.39	115.70	125.13	136.57	136.69	136.88	152.07	153.13	152.25	152.01
	Qimeng Attn	65.43	53.43	38.77	24.62	86.25	66.80	44.85	26.45	92.97	70.25	46.23	26.85
	CuBridge	**137.80**	**176.88**	**182.44**	**151.90**	**197.82**	**214.90**	**211.71**	**202.74**	**239.29**	**249.16**	**199.39**	**214.46**
Share Question Mask	Torch	11.28	6.66	7.32	7.61	11.76	6.74	7.37	7.62	12.02	6.76	7.38	7.63
	FlexAttention	118.28	188.58	278.01	333.53	143.24	224.59	293.34	343.28	174.97	272.52	287.31	337.92
	Qimeng Attn	20.93	23.83	24.43	24.93	28.74	27.41	26.75	26.22	30.88	28.84	27.45	26.58
	CuBridge	**205.51**	**298.34**	**398.49**	**407.32**	**176.67**	**292.36**	**437.08**	**501.42**	**286.09**	**402.16**	**422.20**	**536.37**
Causal Blockwise Mask	Torch	11.31	6.77	7.56	8.10	12.10	6.84	7.62	8.14	12.46	6.88	7.63	8.10
	FlexAttention	115.00	**179.56**	245.89	306.93	136.82	202.30	262.61	321.27	164.82	213.50	270.69	319.44
	Qimeng Attn	37.26	47.03	57.88	66.07	53.20	62.07	69.33	71.82	58.53	66.18	71.22	72.95
	CuBridge	**157.27**	177.58	**340.00**	**388.39**	**161.75**	**277.11**	**441.31**	**498.62**	**240.40**	**366.02**	**418.17**	**530.28**
Relative Pos.	Torch	31.85	26.07	32.06	35.61	32.75	26.27	32.13	35.70	33.45	26.50	32.52	–
	FlexAttention	301.57	401.78	420.14	429.61	328.40	403.19	416.66	428.74	334.10	380.00	407.89	429.10
	Qimeng Attn	143.41	199.26	242.67	266.63	184.73	231.65	264.98	279.89	193.11	238.75	265.39	281.31
	CuBridge	**458.23**	**527.63**	**659.35**	**568.84**	**393.74**	**519.15**	**622.47**	**592.46**	**471.97**	**595.68**	**540.23**	**589.83**
ReLU Attn	Torch	69.58	81.21	87.45	91.27	72.52	82.30	88.87	92.94	74.31	83.03	88.86	92.87
	FlexAttention	no support											
	Qimeng Attn	182.51	280.51	364.83	424.78	253.49	346.74	415.00	443.19	268.86	357.82	418.74	451.79
	CuBridge	**509.02**	**605.51**	**639.66**	**764.12**	**521.33**	**632.60**	**622.52**	**637.52**	**593.97**	**615.01**	**604.02**	**648.54**
Sigmoid Attn	Torch	69.14	80.58	86.69	90.35	71.92	81.76	88.04	92.04	73.71	82.32	88.00	91.84
	FlexAttention	no support											
	Qimeng Attn	87.16	105.95	117.36	123.61	101.46	115.04	122.90	127.11	104.72	117.21	124.36	128.04
	CuBridge	**432.58**	**584.86**	**626.22**	**630.25**	**474.39**	**626.13**	**647.87**	**648.71**	**532.27**	**635.80**	**629.88**	**514.04**
Combo (PrefixLM+ Softcap+ Sigmoid)	Torch	18.19	19.58	19.87	20.25	18.60	19.76	20.03	20.42	18.94	19.84	20.01	20.40
	FlexAttention	no support											
	Qimeng Attn	34.80	40.97	45.98	49.91	39.30	44.81	48.55	51.39	41.15	45.94	49.30	51.90
	CuBridge	**273.45**	**411.90**	**533.12**	**572.94**	**321.80**	**490.90**	**546.66**	**524.12**	**380.78**	**532.47**	**536.65**	**555.81**

Table 5: Performance benchmark (TFLOPS) of attention variants across varying sequence lengths and real-world model attention configurations on NVIDIA H100. (For each backbone the best result are marked in bold, and the second best result are underlined.)

ensure a rigorous comparison.

C.2 Results and Discussion

Table 6 presents the average throughput across sequence lengths from 1k to 8k. The results indicate that *CuBridge* matches the performance of native expert kernels ($0.99\times$–$1.22\times$) and significantly outperforms FlashInfer's extension mechanisms on non-native variants ($2.73\times$–$5.59\times$).

For variants beyond FlashInfer's native support, CuBridge achieves higher TFLOPS than the best FlashInfer configuration at every evaluated sequence length. The gap arises from how new semantics are realized. FlashInfer extensions attach additional semantics onto a fixed execution template without restructuring the schedule for the modified computation. Specifically, the BSR mechanism introduces extra global memory mask reads. Moreover, while BSR enables block-level skipping, it does not support score transforms or non-softmax activations, and JIT allows semantic modification but fails to enable block-level sparse optimization, resulting in unnecessary computation. And neither BSR nor JIT currently supports H100-specific optimizations like warpgroup specialization, WGMMA, and TMA. In contrast, CuBridge reconstructs kernels from a semantic specification via CuIR and re-derives the execution structure while preserving expert orchestration, enabling schedule-level adaptation for each variant.

D Case Study: GEMM-based Operators

To demonstrate the generality of *CuBridge* beyond attention mechanisms, we conducted an additional case study on GEMM-based operators. While attention was chosen as the primary focus due to its complexity, this study validates the broader applicability of our reference-driven paradigm.

Specifically, we use the open-source CUTLASS GEMM kernel as the source expert ref-

Attention Variant	FI (Native)	FI (BSR)	FI (JIT)	*CuBridge*	vs. Best FI
Causal Mask	**551.62**	110.95	176.54	546.18	0.99×
Sliding Window	276.12	78.92	38.27	**282.60**	1.02×
Softcap	383.90	N/A	91.20	**466.85**	1.22×
ALiBi	412.12	N/A	85.37	**442.71**	1.07×
PrefixLM	N/A	111.10	171.56	**515.44**	3.00×
Global Sliding Window	N/A	82.61	68.28	**225.25**	2.73×
Share Question Mask	N/A	99.28	11.89	**411.71**	4.15×
Causal Blockwise Mask	N/A	125.00	35.67	**388.72**	3.11×
Relative Position	N/A	N/A	163.61	**549.43**	3.36×
ReLU Attention	N/A	N/A	197.06	**615.39**	3.12×
Sigmoid Attention	N/A	N/A	158.25	**578.00**	3.65×
Combo	N/A	N/A	89.76	**501.43**	5.59×

Table 6: Performance comparison (Avg TFLOPS) between *CuBridge* and FlashInfer (FI) on H100 GPU (Llama 3.1-405B configuration).

Implementation	Metric	(4k,4k,4k)	(1k,1k,1k)	(4k,4k,1k)	(2k,4k,1k)	(4k,2k,1k)
PyTorch	Latency (ms)	0.2495	0.0319	0.0802	0.0505	0.0498
	TFLOPS	550.86	67.32	428.43	340.20	344.98
***CuBridge* (Ours)**	**Latency (ms)**	**0.2425**	**0.0245**	**0.0580**	**0.0445**	**0.0384**
	TFLOPS	**566.76**	**87.65**	**592.41**	**386.06**	**447.39**

Table 7: Performance comparison between PyTorch and *CuBridge* for the GEMM+ReduceSum operator on H100 GPU. Shape are denoted as (M, N, K).

erence and apply *CuBridge* to generate a fused GEMM+ReduceSum variant. This transformation involves cross-thread accumulation and non-trivial epilogue modification.

Table 7 summarizes the results on H100 GPU. Across multiple common shapes, the generated GEMM+ReduceSum kernel consistently outperforms PyTorch, achieving 1.03×–1.30× speedups. The performance improvement arises from kernel fusion: while PyTorch executes GEMM and Reduce-Sum as separate kernels, requiring intermediate results to be written to and read from global memory. In contrast, *CuBridge* directly modifies the expert reference kernel, enabling execution within a single fused kernel and eliminating intermediate global memory round-trips. This case study demonstrates that *CuBridge*'s reference-driven transformation mechanism and CuIR abstraction generalize to other core operator families.

E Related Work

E.1 Current support for attention variants

The surge in large-scale models has driven extensive GPU research, ranging from high-performance algorithms (Liu et al., 2025; Luo et al., 2025; Zhou et al., 2021) and software stacks (Huang et al., 2026; Zhou et al., 2025b) to innovative hardware utilization (Guo et al., 2020; Hu et al., 2026). Within this landscape, the support for diverse attention variants has emerged as a particularly significant challenge, as it requires balancing algorithmic flexibility with the need for hardware-specific optimization. Current systems for attention variants mainly fall into two categories.

The first category is Expert Libraries. Libraries like FlashAttention (Dao et al., 2022) achieve state-of-the-art performance through heavy manual tuning. However, they lack flexibility and only support a few variants (e.g., ALiBi, sliding window, causal mask, softcap). Similarly, FlashInfer (Ye et al., 2025) applies these techniques to speed up LLM inference. FlashMask (Wang et al., 2025) introduces a structural sparse mask representation, allowing it to support a wider range of mask variants.

The second category is Compiler-based Approaches (e.g., FlexAttention (Guessous et al., 2024), AttentionEngine (Chen et al., 2025a)). These methods improve programmability through template-based design. Specifically, FlexAttention allows users to write simple PyTorch functions for score and mask computations. It then compiles these functions and injects them into fixed slots within a pre-written attention template. While this offers some flexibility, it suffers from structural rigidity. Users cannot change logic outside these designated slots (e.g., they cannot replace

the Softmax operator), which limits the types of algorithms it can support. Moreover, the fixed execution pipeline prevents adaptation to hardware-specific opportunities like overlapping Tensor Core and CUDA Core workloads. As a result, these systems often still lag behind hand-tuned expert libraries in performance.

E.2 LLM for CUDA Code Generation

Recently, with the continuous improvement of LLMs, there is growing interest in using them to automatically generate GPU kernels. However, SOTA models perform poorly on benchmarks like KernelBench (Ouyang et al., 2025) and TritonBench (Li et al., 2025a). This stems from the complexity of parallel programming models (e.g., managing hierarchical execution units and memory system) combined with the extreme rarity of high-quality GPU code in pre-training corpora (less than 0.1% in The Stack (Kocetkov et al., 2023)). Existing research generally falls into two categories.

Training-based Methods. These approaches try to improve the model's mastery of CUDA code through additional training. Methods like Kevin (Baronio et al., 2025) and CUDA-L1 (Li et al., 2025b) apply reinforcement learning to this domain, using generated samples for training to help with the lack of data. Additionally, QiMeng-MuPa (Ke et al., 2025) uses a mutual supervision framework where a code generator and a test case generator improve together. However, these methods require high training costs to modify the underlying model. Our framework is orthogonal to training-based approaches and can be used in conjunction with them.

Workflow-based Methods. This category treats the LLM as a modular component within a system and focuses on the design of systemic workflows to complete complex tasks. (Wei et al., 2025; Chen et al.). Some use execution-guided feedback pipelines (Chen et al., 2025b), while others employ multi-agent frameworks (such as Astra (Wei et al., 2025), CUDA-LLM (Chen et al., 2025b), and CudaForge (Zhang et al., 2025)) that separate planning, generation, and debugging into distinct roles. Similar pipelines have also been explored for Triton kernel generation (Meta PyTorch, 2025; Zhu et al., 2025). However, these frameworks typically treat code as plain text, lacking insight into the complex execution orchestration required for performance. In contrast, *CuBridge* uses a structured IR to make execution orchestration explicit while abstracting low-level CUDA details and use lift–transfer–lower workflow. This enables the model to understand and modify the kernel's logical structure and data flow, ensuring that complex orchestration patterns are correctly handled.

F Lifter Agent System Prompt

The complete system prompt of Lifter Module is presented below.

Lifter Agent System Prompt

 # Role
You are an expert CUDA System Engineer specializing in **Semantic Lifting**. Your goal is to transform low-level, expert-optimized CUDA kernels into **CuIR**, a Pythonic IR designed to capture **Execution Orchestration** while abstracting away implementation complexity.

 # Task: The Semantic Lifting Process
You must process the CUDA kernel **step-by-step** to ensure semantic reliability:

1. **Step 1: Low-level Annotation.** Begin by commenting all `ptx` usages and `CuTe` APIs. You must refer to the technical document to explain the exact hardware behavior of each instruction.
2. **Step 2: Worker Identification.** Analyze the kernel's parallel hierarchy to identify distinct execution units (e.g., warp, warpgroup, threadblock) and their code snippet.
3. **Step 3: CuIR construction.** Before writing any CuIR, reason about how these workers are synchronized. Identify the specific barriers and arrival counts that manage the execution pipeline. Based on the analysis above, map the logic into CuIR primitives. You must ensure that the lifted representation strictly preserves the original asynchronous dataflow and synchronization logic.

 # CuIR Specification
1. Runtime System
The runtime defines the virtual execution environment for the IR.

`class KernelBase` Represents the context of a Single Thread Block.

- `__init__(self, *args)`: Initialize primitives and allocate Shared Memory here.
  Critical: You MUST calculate `self.gridDim` (e.g., `M // BM`) based on input shapes to configure the simulator.
- `self.blockIdx`: A `Dim3` object mimicking CUDA's `blockIdx`.
- `bind(self, func, role_name)`: Control Primitive. Maps a Python method to a virtual hardware worker.
- `commit(self)`: Control Primitive. Triggers the parallel execution of bound workers.

2. CuIR Primitives
Memory & Data Movement

- `alloc(tile_shape, hierarchy, dtype)`:
  Allocates a memory tile on the specified `hierarchy` (e.g., "shared", "register").
  Returns: A Tensor-like object that supports direct indexing, slicing, and standard PyTorch math operations.
- `copy_async(dst, src, barrier, instruction)`:
  Performs an **asynchronous copy** from `src` to `dst`.
  - The `barrier` will be updated when transaction completion.
  - `instruction`: (String) Annotation tag (e.g., "cp.async", "tma"). Purely for Semantic Lifting info; does not affect simulation logic.
- `copy(dst, src, instruction)`:
  Performs a **synchronous copy** from `src` to `dst`.
  - `instruction`: (String) Annotation tag (e.g., "ldmatrix", "stsw"). Purely for Semantic Lifting info.

Compute

- `gemm_async(A, B, instruction)`:
  Performs an **asynchronous matrix multiplication** (`A @ B`).
  - Synchronization: Execution logic assumes results are not immediately available; completion MUST be managed via `wait_group` primitives.
  - `instruction`: (String) Annotation tag (e.g., "wgmma.mma_async"). Purely for Semantic Lifting info.
- `gemm(A, B, instruction)`:
  Performs a **synchronous matrix multiplication** (`A @ B`).

- `instruction`: (String) Annotation tag (e.g., "mma.sync"). Purely for Semantic Lifting info.

- CUDA Core Operations:
  For standard ALU/SFU operations (Add, Mul, Exp, etc.), use native PyTorch operators directly on the Tensors (e.g., `C = A + B`, `torch.exp(x)`).

Synchronization
Models the `mbarrier` hardware primitive. It tracks two distinct goals: **Thread Arrivals** (persistent) and **Transaction Bytes** (transient).

- `barrier.init(expected_threads)`:
  Sets the persistent `base_goal` (number of participating threads).
  - Call this once in `__init__` or before the main loop.
- `barrier.arrive_expect_tx(bytes_count)`:
  1. Signals thread arrival (increments `current_threads`).
  2. Increases the transaction goal by `bytes_count` (increments `tx_goal`).

  Use this immediately before issuing a `copy_async` (TMA) operation.
- `barrier.arrive()`:
  - Signals thread arrival (increments `current_threads`) without modifying transaction expectations.
- `barrier.wait(parity)`:
  - Blocks the thread until the barrier completes the phase associated with `parity`.

 # Lifting Rule
1. Worker-Logic Disentanglement

- Role Identification: Analyze control flow (e.g., `if warp_group_idx == 0`) to distinguish roles.
- Logic Merging: If multiple hardware units perform the same logical task, merge them into a single logic method.

2. Logical Tile Allocation (Memory)

- Shared Memory: Allocate in `__init__` using `alloc(scope="shared")`.
- Register File: When allocating accumulators, derive the shape from the Logical Tile View (e.g., `[BM, BN]`) rather than the fragmented thread-level view.

3. Preservation of Execution Orchestration (Crucial)

- Instruction Scheduling: You MUST preserve the exact relative order of Memory, Compute, and Barriers.
- Latency Hiding: If the CUDA source interleaves computation with pipeline stages, you must replicate this sequence 1:1. Do not reorder instructions.

 # Example
...

 # Output Format (Strict)
Your output MUST be a single executable Python code block.
1. Define one Python class inheriting from `KernelBase`.
2. Implement `__init__`, `run`, and logic methods.
3. Do NOT include explanations or markdown text outside the code block.

G IR Transfer Implementation Details

IR Transfer is implemented as an iterative transformation-and-validation loop. Given an Origin CuIR, an Origin PyTorch specification, and a Target PyTorch specification, *CuBridge* generates a Target CuIR under explicit transformation constraints, validates it against the Target PyTorch reference, and iteratively refines it if needed. The core logic is defined in the following procedure:

The generation stage is LLM-driven and explicitly rule-constrained. Similar to the lifting stage, IR Transfer uses a structured chain-of-thought prompt with ordered reasoning steps. The LLM is instructed to: (1) identify the semantic delta between the Origin and Target PyTorch specifications, (2) localize the affected region in the Origin CuIR, and (3) apply predefined transformation rules to construct the Target CuIR.

The explicit rule taxonomy, as detailed in Table 8, serves two purposes: it constrains modifications to prevent unintended disruption of expert scheduling structure, and it makes the transfer procedure compositional and reproducible rather than free-form code synthesis. After generation, verification is performed via the CuIR runtime before CUDA reconstruction. Failed cases trigger structured diagnostics and bounded refinement.

Algorithm: IR Transfer Loop

```
target_cuir = GenerateTargetCuIR(origin_cuir,
    origin_pytorch, target_pytorch)
is_valid, diagnostics = VerifyAgainstReference(
    target_cuir, reference=target_pytorch)

while not is_valid and iteration <
    max_iterations:
    target_cuir = RefineCuIR(target_cuir,
        diagnostics)
    is_valid, diagnostics =
        VerifyAgainstReference(target_cuir,
        reference=target_pytorch)
    iteration += 1
```

Rule	Transformation Category	Core Action
R1	Iteration domain transformation	Modify loop bounds/iteration space while preserving tiling.
R2	Execution structure preservation	Retain async pipeline and scheduling patterns.
R3	Computation injection or modification	Insert or adapt intermediate computations.
R4	Reduction or normalization replacement	Modify accumulation or normalization strategy.
R5	Data dependency extension	Introduce additional inputs or memory staging.
R6	Auxiliary structure adaptation	Adjust code logic to match control-flow changes.
R7	Unaffected region preservation	Preserve IR regions not implicated by semantic delta.

Table 8: Taxonomy of IR Transformation Rules. These rules constrain the LLM to preserve the expert execution structure while adapting to new attention semantics.