AttentionEngine: A Versatile Framework for Efficient Attention Mechanisms on Diverse Hardware Platforms

Feiyang Chen[1,3], Yu Cheng[2,3], Lei Wang[2,3], Yuqing Xia[3], Ziming Miao[3], Lingxiao Ma[3], Fan Yang[3], Jilong Xue[3], Zhi Yang[2], Mao Yang[3], and Haibo Chen[1]

[1]Shanghai Jiao Tong University, [2]Peking University, [3]Microsoft Research

Abstract

Transformers and large language models (LLMs) have revolutionized machine learning, with attention mechanisms at the core of their success. As the landscape of attention variants expands, so too do the challenges of optimizing their performance, particularly across different hardware platforms. Current optimization strategies are often narrowly focused, requiring extensive manual intervention to accommodate changes in model configurations or hardware environments.

In this paper, we introduce AttentionEngine, a comprehensive framework designed to streamline the optimization of attention mechanisms across heterogeneous hardware backends. By decomposing attention computation into modular operations with customizable components, AttentionEngine enables flexible adaptation to diverse algorithmic requirements. The framework further automates kernel optimization through a combination of programmable templates and a robust cross-platform scheduling strategy. Empirical results reveal performance gains of up to 10× on configurations beyond the reach of existing methods. AttentionEngine offers a scalable, efficient foundation for developing and deploying attention mechanisms with minimal manual tuning. Our code has been open-sourced and is available at https://github.com/microsoft/AttentionEngine.

1 Introduction

Attention is a fundamental mechanism in modern large langauge models (LLMs), enabling groundbreaking advancements in natural language understanding and related domains. By dynamically weighting interactions across input tokens, attention allows models to capture sophisticated contextual relationships, making it an indispensable component of modern deep learning systems.

Attention mechanisms dominate the computational workload in LLMs, and their proportion continuously increases with the growing sequence length. This trend underscores the critical importance of optimizing attention for end-to-end model training and inference. For instance, as illustrated in Table 1, attention accounts for 55% of the computational time in LLAMA-3B when the sequence length is 2048. This proportion further escalates to 82% as the sequence length extends to 8192. Such a significant computational burden highlights the necessity of efficient attention mechanisms to ensure optimal performance and scalability of LLMs across various applications and hardware platforms.

However, attention optimization is nontrivial due to high computation and memory demands and often relies on handcrafted kernels. For example, FlashAttention [11] employs online softmax, memory-efficient pipelining, and kernel fusion to improve canonical attention; while Mamba2 [12], a linear version of attention, utilizes Triton-based [23] kernels with selective gating and chunk-based processing for performance improvement. These handcrafted optimizations are labor-intensive, hardware-specific, and constrained to fixed configurations, thus limiting the adaptability to diverse attention designs and configurations.

The diversity of attention variants continues to expand, driven by task-specific requirements and innovations. For instance, sigmoid attention [18] replaces softmax with sigmoid activation for improved efficiency, and linear attention mechanisms, such as Mamba [12], reformulate computation with selective gating for enhanced efficiency. Other variants, like DeepSeek V2 [13] and RetNet [21], deviate further by requiring non-standard tensor dimensions, introducing additional computational challenges.

Adapting to this growing diversity requires significant expert efforts for kernel customization. Furthermore, differences in Attention input configurations and hardware platforms, such as NVIDIA A100, H100, and AMD MI300X GPUs, complicate the landscape. Hardware differences in tile sizes, memory hierarchies, and pipelining strategies necessitate new implementations, significantly increasing development overheads and limiting scalability. For example, FlashAttention v2 reached 70% of the peak computation throughput on the NVIDIA A100, but only achieved 30% on the NVIDIA H100. Complex techniques such as register-level pipelining and ping-pong kernel design must be used to achieve peak

performance [19].

To address these challenges, we propose AttentionEngine, a unified framework for designing, optimizing, and executing diverse attention mechanisms across hardware platforms. At its core, AttentionEngine abstracts attention mechanisms into two fundamental operations: relevance scoring and aggregation. These operations capture the essence of attention mechanisms, ensuring a consistent yet flexible foundation for diverse designs.

Building on this abstraction, AttentionEngine introduces customizable attention templates that fix the core operations of relevance scoring and aggregation while exposing `customizable functions` for user-defined extensions. These functions allow users to design their attention variants by applying transformations like masking, scaling, or row-wise normalization, enabling seamless adaptation to task-specific requirements.

One challenge is how to retain high performance customization despite abstraction. AttentionEngine enables automated optimization through a cross-backend scheduling and execution framework that dynamically adapts to input configurations and hardware constraints. By abstracting kernel generation and optimization complexities, AttentionEngine supports a wide range of attention variants and hardware platforms while delivering exceptional performance.

We implemented AttentionEngine with 7.3k lines of C++ and Python code and have open-sourced the system to foster further innovations. Evaluation results demonstrate that AttentionEngine achieves performance comparable to handcrafted expert-optimized kernels, delivering up to 10.4× speedup for configurations unsupported by existing implementations. Moreover, AttentionEngine provides unparalleled flexibility for designing and optimizing custom attention mechanisms, marking a significant step toward scalable and generalizable attention computation.

2 Background

2.1 Attention Mechanisms

Large Language Models (LLMs) have transformed natural language processing (NLP), enabling breakthroughs in tasks such as text understanding and generation. At the core of this success is the attention mechanism, which allows models to selectively focus on relevant parts of an input sequence, significantly enhancing sequence-to-sequence tasks like translation [6]. Attention computes pairwise relevance, or *attention scores*, between input tokens, which are then used to weight and aggregate token representations, guiding the generation of output tokens.

The introduction of Queries (Q), Keys (K), and Values (V) in the Transformer architecture [24] formalized and generalized attention computation. Queries represent what the model seeks, Keys encode the input attributes, and Values carry the associated content. Modern attention computation, as summarized in Figure 1, follows five key stages:

- **Input Tokens:** Raw input sequences serve as the foundation for computation.
- **Embedding:** Input tokens are mapped to continuous vector representations through projections of Q, K, and V matrices, encapsulating semantic information.
- **Interaction:** Pairwise relevance scores are computed using the dot product of Q and K, optionally scaled, to quantify token relationships.
- **Normalization:** Relevance scores are transformed into normalized weights using functions like softmax, ensuring interpretability and row-wise consistency.
- **Composition:** Weighted scores are combined with V representations to generate context vectors, integrating information from relevant tokens into a single output for each token.

The Q, K, V framework has established itself as the foundation of modern attention mechanisms, offering scalability, flexibility, and computational efficiency. This structured approach underpins the success of neural architectures in addressing a wide range of NLP tasks.

2.2 Diversity in Attention Mechanisms

Building on the foundational design of attention mechanisms, researchers have introduced numerous variants aimed at improving performance, addressing task-specific requirements, and enhancing computational efficiency. As illustrated on the right side of Figure 1, these innovations can be categorized into algorithmic and efficiency advancements, each targeting specific stages of the attention mechanism.

Algorithmic Innovations focus on enhancing robustness, accuracy, and task-specific capabilities in attention:

- Task-Specific Modifications: Causal attention [24] modifies the interaction stage by restricting interactions to prior tokens. This design supports autoregressive decoding, a critical feature for applications like text generation and speech synthesis.
- Improved Robustness and Accuracy: DiffTransformer [31] refines both the interaction and normalization stages for higher accuracy and reduced noise in attention scores.
- Non-Conventional Tensor Dimensions: Models like DeepSeek V2 [13] and RetNet [21] enhance the embedding stage by employing higher hidden dimensions, enabling richer semantic representation.

Efficiency Innovations aim to reduce computational overhead while maintaining the effectiveness of attention:

- Compact Representations: Linear attention, such as Mamba [12] and the recurrent form of RetNet [21], transforms the interaction, normalization and composition stages by compressing past information into compact KV representations. Sliding Window Attention [7] modifies the interaction stage

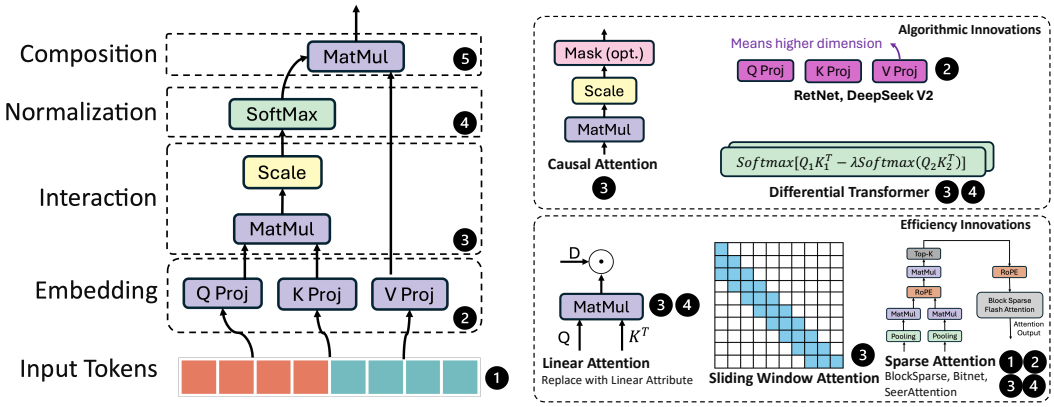

Figure 1: The foundational attention mechanism and its variants: Attention mechanisms is divided into stages such as embedding, interaction, normalization, and composition(left). Attention variants make various changes to these stages(right). For example, Causal Attention modified the interaction stage to apply a mask, which makes the computation flow different.

Seqlen	2K	4K	6K	8K
LLAMA-3B	55%	70%	78%	82%

Table 1: Attention proportion in LLAMA-3B inference

by limiting the attention scope to a fixed local window, optimizing memory usage and computational focus.

- Sparse Attention: Sparse attention mechanisms, such as BigBird [33], SeerAttention [16], and BitNet [26], introduce sparsity across multiple stages, including input tokens, embedding, interaction, and normalization. These methods leverage structured patterns or treat low-bit precision as sparse regions to reduce computational and memory demands without sacrificing effectiveness.

2.3 Efficient Implementation of Attention

The attention mechanism takes large proportion in LLM computation. Table 1 shows the attention proportion in LLAMA-3B inference. Efficient implementations of various attention mechanisms hinge on reducing memory access and maximizing the utilization of compute units. Many libraries with handcrafted kernels achieve this by fusing memory-intensive operations, including element-wise calculations and reductions.

FlashAttention [19] exemplifies this approach by integrating softmax computation, memory-efficient pipelining, and kernel fusion, thereby reducing computational overhead and improving performance. However, these libraries impose strict constraints on the attention patterns they support. Even minor deviations, such as the atypical input dimensions used in DeepSeek V2 and RetNet, can invalidate these optimizations. Figure 2 illustrates the performance disparity across different attention variants. For standard Softmax-Attention, the handcrafted library FlashAttention3 [19] significantly outperforms the native PyTorch implementation, achieving over 60% FLOPS utilization. In contrast, for less common variants like Gated-RetNet and ReLU-Attention, these libraries exhibit

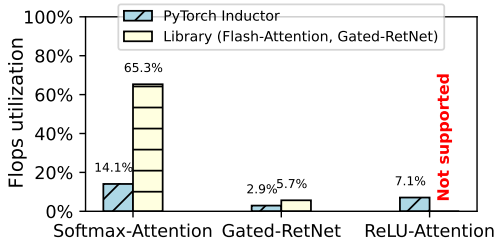

Figure 2: The performance of attention implementations.

poor performance or provide no support at all.

Additionally, due to limited development resources, these libraries predominantly target top-tier hardware, such as NVIDIA's H100 and A100 GPUs, and are not easily transferable to alternative platforms like AMD GPUs. Adapting these implementations to different hardware ecosystems remains challenging and demands significant expertise.

To simplify kernel development, automated compilers [1, 4, 5, 8, 20, 34] have emerged. While these tools reduce development effort, they struggle to match the performance of handcrafted kernels for attention variants. This limitation arises from their inability to fully understand the semantics of attention computation, as they often treat it as a sequence of discrete and opaque operations. Advanced optimizations, such as transforming softmax into an online softmax, are beyond the scope of current compiler capabilities, resulting in suboptimal performance.

To balance performance and development efficiency, some approaches adopt trade-offs between flexibility and optimization. For instance, FlexAttention [14] utilizes a template-based methodology in which the majority of the computation is predefined, while exposing a limited set of customizable functions to users. This design enables the optimization of the entire attention operation while providing some flexibility for specific variants. However, these templates are derived from the computational flow of a particular variant, making it difficult to generalize to a wider range of attention variants, such as linear attention.

3 A Unified Attention Abstraction

Attention mechanisms exhibit significant diversity at the implementation level. For example, standard attention utilizes matrix multiplication to compute attention scores between Q and K, followed by a weighted aggregation of V to produce the output representation. In contrast, linear attention compresses K and V using a recurrent loop before applying Q to compute the output. Despite these implementation differences, these variants adhere to the same underlying principles of attention semantics.

By examining the native implementation of attention as a loop-based operation, we identify two fundamental components common to all attention mechanisms:

- *Relevance Scoring:* This operation forms the core of attention mechanisms, capturing pairwise similarities or interactions between input tokens. It is typically realized through inner products or other similarity measures to determine token relationships.
- *Aggregation:* Using the relevance scores, this operation consolidates contextual information into a representation for each token.

Building on these two fundamental operations, we propose a unified template that encapsulates the diverse spectrum of attention variants. This template abstracts the core semantics of relevance scoring and aggregation while offering customizable components, striking a balance between broad applicability and development flexibility. By providing a consistent framework, this approach streamlines the design and implementation of new attention mechanisms while enabling efficient adaptation to evolving computational demands. The next section introduces AttentionEngine, a unified framework that brings this abstraction to life, facilitating efficient and scalable attention mechanism design across diverse hardware platforms.

4 Design

Expanding on our attention abstraction, we introduce AttentionEngine, a unified framework designed to streamline the design, optimization, and execution of diverse attention mechanisms across hardware platforms. As shown in Figure 3, AttentionEngine begins with attention templates in the Programming Interface. These templates retain the core abstractions of attention—relevance scoring and aggregation—outlined in §3 while providing customizable functions that allow users to design their own attention variants. By preserving the essential principles of attention and offering flexibility for user-defined extensions, AttentionEngine facilitates the creation of a wide range of attention mechanisms and simplifies backend optimization.

Once customized, the attention mechanisms are lowered to `Kernel Templates`, which formalizes computation and memory operations. These templates, combined with two key components—`IntermediateTensor`, representing transient computational data, and `DeviceConfig`, capturing hardware constraints—define the scheduling space. AttentionEngine employs a two-layer scheduling policy within this space to determine the optimal execution plan, balancing performance and resource utilization. The finalized plan is then mapped to hardware backends, ensuring scalability and efficiency across various configurations.

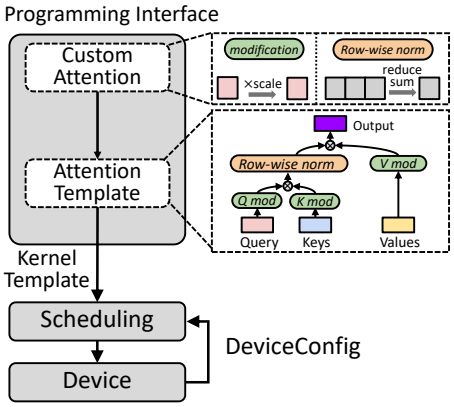

Figure 3: System overview: AttentionEngine begins with attention templates in the Programming Interface to define Custom Attention. Then they are lowered to kernel templates and automatically scheduled to generate the best execution plan on the device.

The following sections delve into the components of this framework, demonstrating how AttentionEngine integrates abstraction, optimization, and execution to unify and extend the implementation of attention mechanisms.

4.1 Programming Interface

Attention Patterns and Templates Building on our abstraction of attention operations—relevance scoring and aggregation—we design a unified attention template that serves as a versatile foundation for implementing diverse attention mechanisms. As depicted in Figure 4, the template takes Q, K, and V after projection as inputs, retaining two fixed computations: $Q@K$ for relevance scoring and $S@V$ for aggregation. These computations capture the essence of attention mechanisms while offering flexibility through customizable functions.

The template includes two key customizable functions, `modification` and `row-wise normalization`, which can be inserted at designated points to enable users to define attention variants tailored to specific needs. These functions allow for operations such as applying masking, implementing custom normalization schemes, or adapting to unique computational goals.

To facilitate optimization, this unified template is instantiated in two computational patterns—parallel and recurrent:

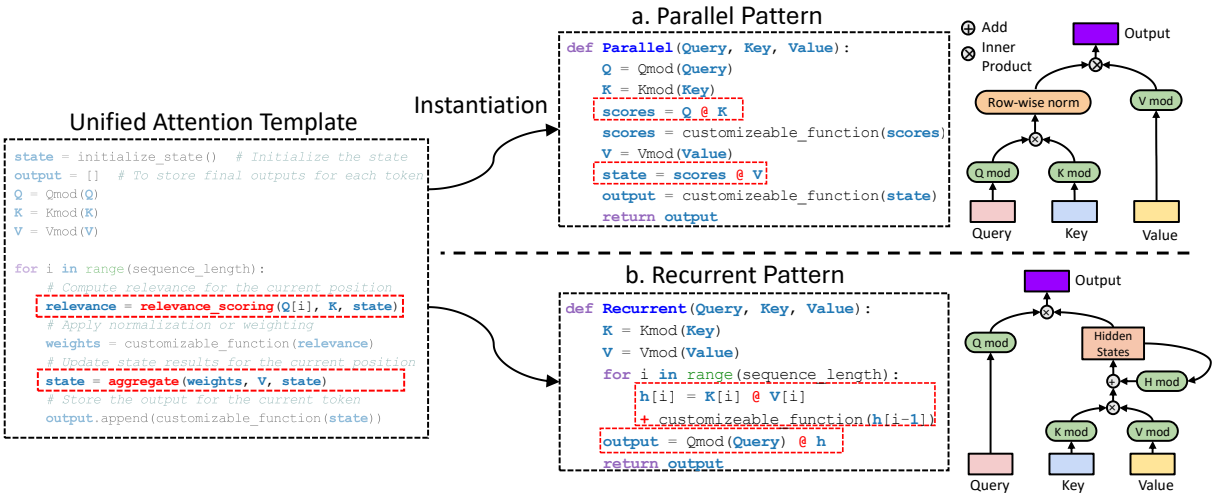

Figure 4: On the left is AttentionEngine's unified attention template. By instantiating this template, two distinct patterns are produced (Parallel Pattern and Recurrent Pattern). The red box highlights the operations corresponding to the core components of the attention mechanism in the unified attention template: `relevance_scoring` and `aggregate`. Both the `customizable_function` and the `mod` function are user-defined. The `customizable_function` encompasses both `modification` function and `row-wise normalization` function, whereas the `mod` function is restricted to `modification` function only.

```
class modification:
    Func mod: Tensor->Tensor;
    Bool ismask;
class row-wiseNormalization:
    Func online_prologue: Tensor->Tensor;
    Func online_fwd: Tensor->Tensor;
    Func online_epilogue: Tensor->Tensor;
```

Figure 5: Customizable functions in programming interface

- Parallel Pattern: Relevance scoring and aggregation are implemented as matrix multiplications, with $Q@K$ representing the scoring and $S@V$ representing the aggregation. Customizable functions are applied to the relevance scores to compute weights and to the state to produce the final output. Since most existing parallel attention variants do not innovate on state transformations, the customizable function for the state often defaults to an identity operation. This pattern is well-suited for mechanisms requiring global context and high parallelism.
- Recurrent Pattern: Relevance scoring and aggregation are sequentially computed, with $K@V$ and $Q@h$ together capturing the relevance scoring and aggregation, iteratively maintaining compressed states. In this pattern, the customizable functions on weights and states are reformulated as customizable function on the hidden state h. This makes the recurrent pattern ideal for memory-efficient designs and tasks with sequence dependencies.

By integrating the two instantiated patterns, this unified attention template empowers users to design high-level attention mechanisms while AttentionEngine seamlessly handles low-level implementation and hardware-specific optimization, ensuring both efficiency and scalability.

Customizable functions and flexibility. As shown in Figure 5, customizable functions in AttentionEngine include the `modification function` and the `row-wise normalization function`, which serve as user-defined components within the attention templates.

The `modification function` supports fine-grained elementwise transformations and masking, allowing users to customize operations applied to individual tensor elements. For example, scaling the query tensor by $1/\sqrt{d_k}$ in standard softmax attention can be achieved using this function. Masking operations, such as applying a causal mask, can also be implemented by annotating this function for masking purposes.

The `row-wise normalization function` provides a placeholder for normalizing or weighting. It enables global adjustments across tensor rows, accommodating a combination of elementwise and row-reduce computations. Examples include applying a row-wise softmax for normalizing attention scores or implementing numerical stabilization techniques. To enhance performance, the `row-wise normalization function` can be defined as an online function, where computations are processed sequentially in blocks along the rows. This approach significantly reduces memory overhead and ensures efficient execution.

The `online row-norm interface` facilitates the implementation of online row-wise normalization, inspired by FlashAttention [11]. As shown in Figure 6, this interface includes three main components:

- `online_prologue`, which initializes the state variables before entering the online loop.
- `online_fwd`, which defines computations within each block of rows, updating state variables like row maxima or sums.

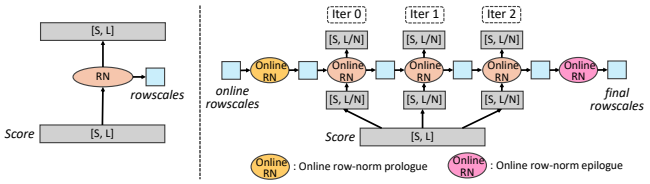

Figure 6: Illustration of the online row-norm interface. The left panel shows the standard row-wise normalization function, while the right panel demonstrates how AttentionEngine enables users to implement the same functionality as an online function using the online row-norm interface.

```
def online_prologue():
    row_max = -inf
    row_sum = 0
def online_fwd(row_max_prev,row_sum_prev):
    row_max_cur = scores.reduceMax()
    row_max = max(row_max_cur, row_max_prev)

    scores = exp(scores-row_max)
    row_sum_cur = scores.reduceSum()
    row_sum = row_max_prev/row_max * row_sum_prev +
              row_sum_cur

    return row_max, row_sum
def online_epilogue():
    scores = scores / row_sum
```

Figure 7: Example of online row-wise normalization function: softmax attention.

- `online_epilogue`, which finalizes the computation after the loop.

Users can leverage this interface to construct both forward and backward computation graphs via the `forward` and `backward` methods, enabling seamless integration with automatic differentiation and backend optimization. Key variables, such as `online_rowscales` (state variables passed between blocks) and `final_rowscales` (storing the final reduction results), provide the flexibility to define new online functions beyond softmax, significantly broadening the scope of attention mechanisms supported by AttentionEngine.

Computing primitives. To simplify user-defined operations, we introduce a set of `computing primitives`, which abstract hardware-specific details. These primitives are categorized as:

- Elementwise Operations: Operations like *add(), sub(), tanh()*, and others allow fine-grained transformations of individual tensor elements.
- Row Reduce Operations: Aggregation functions such as *reduceSum(), reduceMax()*, and *reduceAbssum()* enable efficient row-wise reduce computations.

These primitives provide a robust foundation for defining both modification and row-wise normalization functions, ensuring compatibility with diverse hardware platforms.

Examples of Attention variants. Our interfaces support a wide range of attention mechanisms, demonstrating their flexibility and generality.

The Softmax Attention mechanism involves scaling the query tensor by $1/\sqrt{d_k}$ for normalization and applying a numerically stable softmax function to the scores. Specifically, the modification interface for q is defined as $q\_mod = lambda\ \ q : q/\sqrt{d_k}$, while the row-wise normalization interface on scores is expressed as: $score\_rownorm = lambda\ \ scores : (\_scores = \exp(scores - scores.\text{reduceMax}()); \_scores = \_scores/\_scores.\text{reduceSum}())$. To enhance performance, we implement the row-wise normalization function in an online form using our online row-norm interface (Figure 7). Specifically, `online_prologue` initializes the state, `online_fwd` performs intermediate computations on scores and state, and `online_epilogue` finalizes the computation.

ReluAttention replaces the softmax function with a row-wise normalization function that contains only an elementwise operation. We use our modification interface on scores as $score\_mod = lambda\ \ scores : \max(scores, 0)$. In this case, no additional normalization is applied.

Similarly, in RetNet parallel attention, a retention mask is applied to the scores. This is represented by the modification function $score\_mod = lambda : \ \ scores = scores \times mask$, and a row-wise normalization function ensures numerical stability, defined as $score\_rownorm = lambda\ \ scores : (scores/scores.\text{abs}().\text{reduceSum}().\text{clamp}(\text{min} = 1))$.

Mamba2, a representative linear attention mechanism, incorporates a selective gating mechanism to modulate the key and hidden states, allowing selective attention to past information. Our interface represents this as a modification function on the key $k\_mod = lambda\ k : k \times gate$ and a modification function on the hidden states $h\_mod = lambda\ h : h \times decay \times gate$.

4.2 Scheduling Space

The scheduling space in AttentionEngine is inherently shaped by the kernel templates, which encapsulate the computation flow of attention mechanisms. These templates, derived from our attention pattern abstractions, constraining the range of scheduling options while enabling efficient and adaptable execution. Together with `IntermediateTensor` and `DeviceConfig` components, the kernel templates form the foundation for determining optimal execution strategies.

Kernel template. Kernel templates play a pivotal role in structuring the scheduling space by formalizing the computation and memory operations of attention mechanisms. These templates provide a consistent structure for implementing diverse attention mechanisms while allowing flexibility for hardware-specific optimizations. For example, templates designed for parallel patterns incorporate online techniques to efficiently manage row-wise normalization, while those for recurrent patterns utilize chunk parallelism to maximize tensor core utilization and computational efficiency.

Additionally, AttentionEngine supports multiple kernel

```
class IntermediateTensor{
    TileShape tile;
    MemoryLocation mem;
    int pipelineStage;
};
```

Figure 8: IntermediateTensor component

```
class DeviceConfig{
    BaseTileShape basetile;
    List<MemoryCapacity> memoryInfo;
};
```

Figure 9: DeviceConfig component

templates tailored to different hardware backends, such as those implemented in Triton [23], CUTE [9], and TileLang [3]. Leveraging a common lowering method based on attention pattern abstractions, AttentionEngine ensures that customized attention variants can be seamlessly lowered to these templates. This flexibility allows AttentionEngine to dynamically select the optimal kernel template based on the input data and hardware platform, achieving consistent high performance across configurations.

IntermediateTensor. At the heart of the scheduling space lies the `IntermediateTensor` component, which encapsulates the transient data generated during computation. By focusing on intermediate tensors, AttentionEngine can systematically deduce the tiling, memory allocation, and pipeline requirements for attention mechanisms.

Key attributes of `IntermediateTensor` include:

- Tensor tile shape (`tile`): By dividing tensors into smaller tiles, we can perform operations tile-by-tile and allocate buffers efficiently. Using the computation graph, we propagate the tiling scheme across all operations to infer the tile shapes of Q, K, V and other tensors, ensuring an optimal balance between computation and memory.
- Tensor location (`mem`): Intermediate tensors can be stored in various levels of memory, such as global memory, shared memory, or registers. Each location offers a trade-off between latency, bandwidth, and resource availability.
- Pipeline stage (`pipelineStage`): Operations involving intermediate tensors are divided into multiple pipeline stages, such as memory copy and computation. The number of stages determines the buffer requirements and scheduling flexibility, enabling overlapping operations to maximize throughput and minimize resource contention.

This component ensures that all elements of the attention mechanism, including inputs, outputs, and intermediate results, are unified under a consistent scheduling strategy.

DeviceConfig. The `DeviceConfig` component provides hardware-specific constraints that refine the scheduling space defined by kernel templates and intermediate tensors. It encapsulates attributes such as:

- Base tile shape (`basetile`): Specifies the optimal tile shape for computations on the target hardware, ensuring alignment with hardware-specific constraints, such as alignment with GEMM computing instruction and memory transaction.
- Memory hierarchy (`memoryInfo`): Provides details about the available memory tiers (e.g., registers, shared memory, global memory) and their respective capacities, enabling efficient allocation and minimizing contention.

`DeviceConfig` plays a pivotal role in determining the feasible tiling and memory strategies during scheduling. For instance, the base tile shape ensures hardware-aligned tiling configurations, while memory capacity constraints prevent resource overcommitment.

By combining kernel templates, `IntermediateTensor`, and `DeviceConfig`, AttentionEngine constructs a unified scheduling space that supports diverse attention mechanisms and hardware platforms. Kernel templates anchor the computation flow, `IntermediateTensor` defines the key computational attributes, and `DeviceConfig` introduces hardware constraints, together forming a robust and scalable scheduling framework.

4.3 Scheduling policy

As illustrated in Figure 10, AttentionEngine employs a two-layer scheduling policy to minimize latency and optimize execution. This policy operates at two levels: `tile config scheduling` and `tile resource scheduling`. At the tile config scheduling level, the policy traverses the entire space of possible tile configurations, leveraging the constrained nature of the scheduling space to perform exhaustive exploration. At the tile resource scheduling level, the policy determines the optimal memory placement and execution strategy within each tile configuration, ensuring efficient hardware resource utilization while adhering to hardware constraints.

Tile config scheduling. The tile config scheduling layer takes as input a computation graph (`Graph`) composed of `IntermediateTensor` objects and hardware configuration details (`DeviceConfig`). This layer begins by invoking the `InferPossibleTileConfigs` function (line 2) to identify all potential tile configurations for the computation graph, propagating from the output tensors. Due to the complexity of attention mechanisms, including their intricate computation stages, hardware alignment requirements, and memory limitations, the tile configuration space is constrained. This enables an exhaustive traversal of all possible tile configurations.

For each tile configuration (line 4 - 5), the policy generates a set of execution plans using the tile resource scheduling layer and evaluates their performance through profiling (line 6 -7). Profiling involves calculating the latency of each plan to determine its efficiency. Finally, the tile configuration corresponding to the plan with the lowest latency is selected as the optimal configuration.

Tile resource scheduling. The tile resource scheduling layer

```
1  Func TileConfigScheduling(g: Graph,
     D:DeviceConfig)
2      tensor_tile_configs = InferPossibleTileConfigs(g,
         D.basetile);
3      plans = []
4      for tile_config in tensor_tile_configs do
5          plans += TileResourceScheduling(tile_config,
             g.IntermediateTensors , D);
6      for plan in plans do
7          if Profile(plan) < best_latency
8              best_latency = Profile(plan); best_plan =
                 plan;
9      return best_plan;
10 Func TileResourceScheduling(tile_config: TileConfig,
     t: IntermediateTensors, D:DeviceConfig)
11     InitMemLocation(t.memLoc, REGISTER);
12     t = sortByTensorSizeDec(t);
13     for tensor_i in t do
14         plans = GeneratePlans(t);
15         for plan in plans do
16             if not
                 ComputeMemoryConstraint(tile_config, t,
                 plan, D.memoryInfo)
17                 plans.remove(plan);
18         if not plans.isEmpty()
19             return plans;
20         LowerMemLocation(tensor_i.memLoc)
21     return EmptySet();
```

Figure 10: Scheduling algorithm

optimizes the execution plan for a specific tile configuration. The process starts by initializing all intermediate tensors to the highest memory tier available (e.g., registers) to reduce memory I/O overhead (line 11). The intermediate tensors are sorted in descending order of size, prioritizing larger tensors for memory allocation to maximize efficiency (line 12).

For each tensor, the policy iteratively generates execution plans (line 13-21) and checks their feasibility against hardware constraints, such as memory capacity and alignment requirements (line 16). If no valid plan is found, the policy progressively demotes tensors to lower memory tiers (e.g., shared or global memory) and reattempts plan generation (line 18 -20). This iterative adjustment continues until a feasible plan is identified or all options are exhausted. If no valid plan can be generated, the function returns an empty set (line 21).

By combining the two layers, the scheduling policy systematically explores the design space to produce efficient, hardware-aware execution plans for attention computations. This hierarchical approach enables AttentionEngine to balance performance and resource utilization, supporting diverse attention variants across multiple hardware platforms.

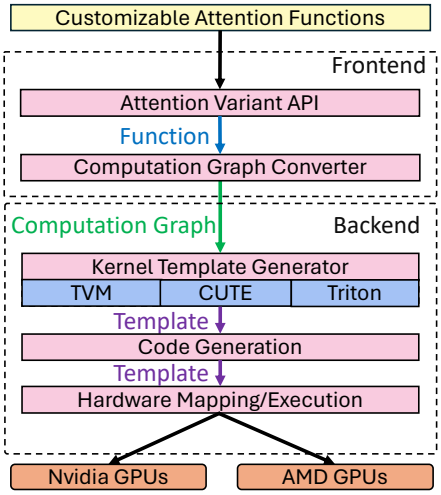

Figure 11: Implementation of AttentionEngine

5 Implementation

In this section, we present the implementation of the AttentionEngine frontend and backend. Figure 11 provides an overview of the AttentionEngine workflow. We intergrate AttentionEngine into pytorch [2] as a module. The frontend accepts user-defined functions as input, constructs the computation graph of intermediate tensors by graph tracer, and passes it to the backend, which generates optimized device kernels for efficient execution.

5.1 Frontend

The frontend of AttentionEngine provides the foundation for defining and representing user-defined attention mechanisms. It introduces a set of computing primitives, facilitates the use of customizable functions such as modification and row-wise normalization. Furthermore, AttentionEngine traces computation graphs by encoding tensor attributes, enabling automatic differentiation and efficient backend integration.

Computing primitives. The frontend includes a rich set of computing primitives, categorized into elementwise and row-reduce operations. Elementwise operations, such as *add(), sub(), tanh()*, are computed in a SIMT style on GPUs and are fused with matrix multiplications at the register or shared memory level to minimize memory access overhead. Row-reduce operations, such as *reduceSum()* and *reduceMax()*, leverage GPU warp-level reduction, where each row-reduce operation is computed by the same thread block and warp.

Modifaction function. The modification function exclusively supports elementwise operations. These operations are fused by AttentionEngine into a single computation unit and lowered to the backend kernel template for efficient execution. The modification function also supports masking operations, allowing users to implement attention variants that require masking logic.

Row-wise normalization. The row-wise normalization function supports both elementwise and row-reduce operations or a combination of the two. Similar to the modification function, all operations within the row-wise normalization function are fused and lowered to backend kernel templates.

Tracing user-defined computation graphs. AttentionEngine traces user-defined computation graphs by building a directed acyclic graph of Tensor. Each node contains the computing primitive(such as *add*(), *reduceSum*()), the coresponding output Tensor attributes(such as shape), and a list of pointers pointing to its preceded node. This enables the system to dynamically trace the dependency between tensors. We also define a *grad* field on each node, which is a pointer to another node containing the gradient of current tensor. By iteratively traverse between nodes, we encodes the gradients informantion of each node into the *grad* field to achieve automatic differentiation. The forward and backward computing graph AttentionEngine constructed ensures seamless integration with the backend for efficient kernel generation.

5.2 Backend

The backend of AttentionEngine transforms user-defined algorithms into kernel templates and optimizes these templates into high-performance kernels.

Kernel template. We design kernel templates to systematically implement attention lowering. Since most custom attention mechanisms preserve the overall kernel computation flow, a template-based approach is particularly effective. Using the computation graphs generated from the `modification` and `row-wise normalization` functions in §5.1, we produce essential components such as intermediate tensor definitions, initialization routines, memory operations, and computation steps. These components are seamlessly fused into the kernel templates, ensuring computational and memory efficiency.

To achieve extreme performance, we leverage TileLang [3] and CUTE [9] to implement optimized kernel templates. For parallel attention, our templates employ advanced online techniques to handle row-wise normalization efficiently, ensuring adaptability across a wide range of configurations. For recurrent attention, we utilize chunk parallelism to fully exploit tensor cores, balancing computational throughput and efficiency. These strategies allow AttentionEngine to accommodate the unique characteristics of different attention mechanisms while maximizing hardware utilization.

Handcrafted kernels, such as FlashAttention, are often limited to specific configurations, typically requiring d_{qk} to equal d_v. This highlights a key shortcoming of ad hoc approaches: they fail to address diverse input configurations without extensive manual tuning of schedules, such as tile sizes, pipelining, and fusion strategies—efforts that demand significant expertise. In contrast, AttentionEngine automates this process, enabling support for various input configurations without manual intervention. Our kernel templates are designed to handle diverse configurations of d_{qk} and d_v, eliminating the need for padding when these dimensions differ, as seen in models like DeepSeek V2 (d_{qk}=192, d_v=128) [13] and DiffTransformer (d_{qk}=128, d_v=256) [31]. By reducing padding overhead and computation costs, AttentionEngine not only extends support to a broader range of attention designs but also enhances performance while maintaining flexibility across hardware platforms.

Lowering computation graphs to kernel templates. The lowering process translates user-defined computation graphs into kernel templates. This process is divided into two stages: expression generation and code generation. The split design enhances extensibility, with expression generation being kernel-template-agnostic and code generation adapting to specific kernel templates.

During expression generation, AttentionEngine inputs a user-defined computation graph and performs a topological sort to convert it into a linear sequence of computation expressions, preserving the computation order. Additionally, as the graph is traversed, the use-define chain for each node is analyzed, enabling optimizations such as variable reuse. In the subsequent code generation phase, these computation expressions are used to produce kernel code tailored to the selected kernel template through string matching. The resulting kernel code includes variable initialization, memory copying, and computation steps, seamlessly integrating user-defined operations into efficient kernel templates.

Map to hardware backend. We map the kernel templates to both NVIDIA GPUs and AMD GPUs, optimizing performance across diverse hardware platforms.

For NVIDIA GPUs, AttentionEngine supports two backends: TileLang [3] and CUTE [9]. Using the Triton-like compiler, we map elementwise operations and reduce operations by utilizing APIs such as `ParallelFor` for thread-level execution and `reduce_sum`/`reduce_max` for block-level row-reduction. With CUTE, we employ `cute::Tensor` and `cute::layout` to define thread-level data layouts and map reduce operations to efficient micro-kernel templates, ensuring high performance for compute-intensive tasks.

For AMD GPUs, AttentionEngine supports the MI250, AMD's high-performance GPU architecture, equipped with Matrix Cores for matrix multiplication, Arithmetic Logic Units (ALUs), and asynchronous copy units for efficient memory transfer. Leveraging TileLang [3]'s capabilities, we generate highly optimized kernels tailored to the MI250, fully utilizing its advanced hardware features for efficient execution.

6 Evaluation

In this section, we evaluate AttentionEngine on both attention microbenchmarks and end-to-end models by comparing them

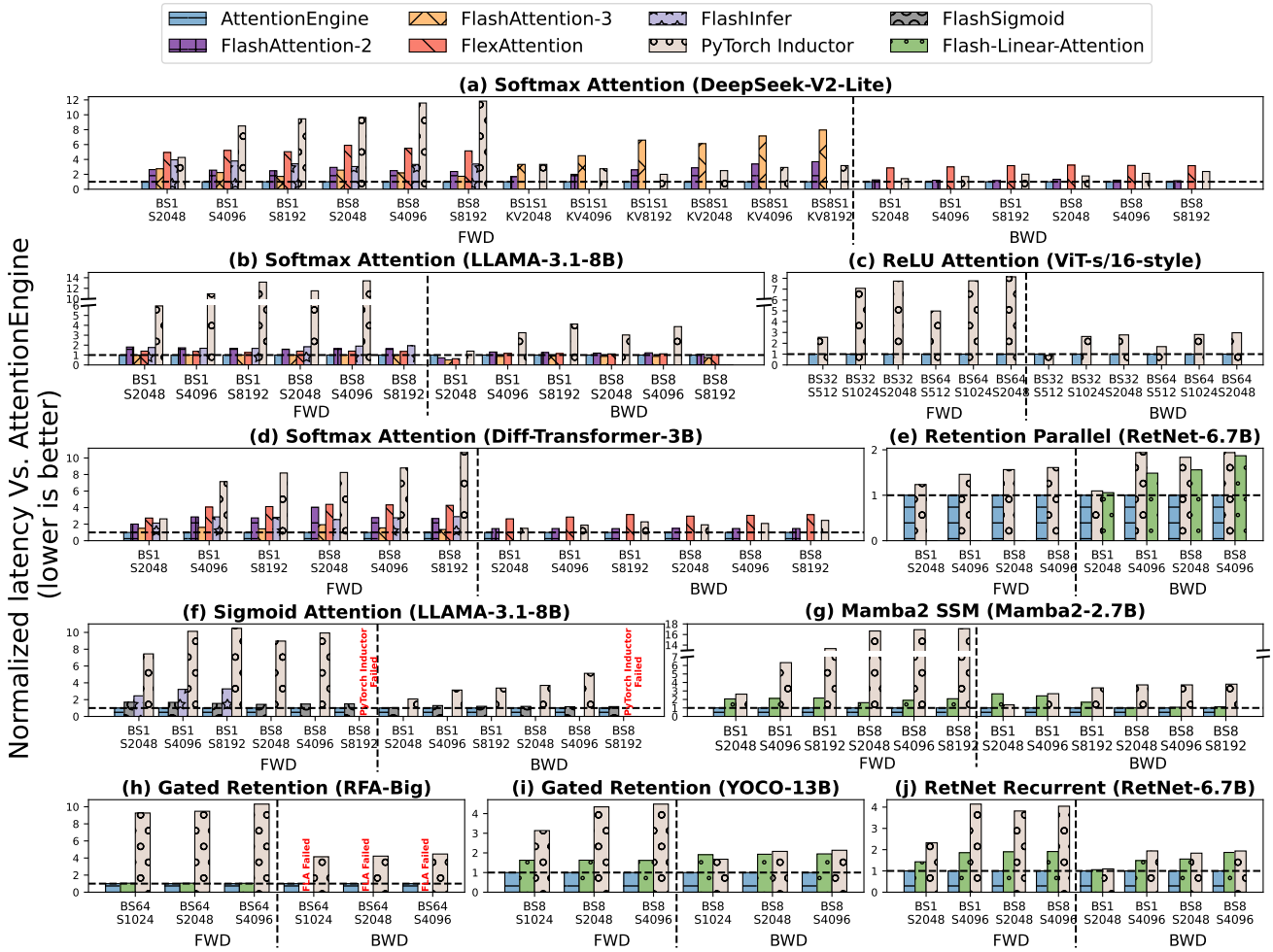

Figure 12: Attention operator performance on H100 GPUs.

Operator	Configuration	Model
Softmax Attention	head=32, dimqk=128, dimv=128	LLAMA3.1-8B
Softmax Attention	head=16, dimqk=192, dimv=128	Deepseek-V2-lite
Softmax Attention	head=12, dimqk=128, dimv=256	DiffTransformer-3B
Sigmoid Attention	head=32, dimqk=128, dimv=128	LLAMA3-8B-style
Relu Attention	head=6, dimqk=64, dimv=64	ViT-s/16-style
Retention Parallel	head=32, dimqk=256,dimv=512	RetNet-6.7B
Mamba2 SSM	headv=80, dimqk=128, dimv=64	Mamba2-2.7B
Retention Recurrent	head=32, dimqk=256,dimv=512	RetNet-6.7B
Gated Retention	head=40, dimqk=256, dimv=256	YOCO-13B
Gated Retention	head=16, dimqk=64, dimv=64	RFA-Big
Softmax Attention Decoding	Seqlen=1, head=16, dimqk=192, dimv=128	Deepseek-V2-lite

Table 2: A subset of attention in our microbenchmark.

with the state-of-the-art libraries and the compiler-based method to demonstrate the effectiveness of AttentionEngine. We summarize our findings: (1) AttentionEngine can optimize standard transformer attention, achieving comparable performance with hand-crafted libraries. (2) AttentionEngine can generate custom attention kernels, achieving speedup up to 10.4×. (3) AttentionEngine support multi-backends, including NVIDIA and AMD GPUs.

6.1 Experimental Setup

Hardware platforms. We evaluate AttentionEngine on both NVIDIA and AMD GPUs, as they are currently the most popular hardware platforms. Our evaluation includes two high-performance GPUs: the NVIDIA H100 and the AMD Instinct MI250 GPU. We use CUDA version 12.4, Triton version 2.3.1 with the H100 GPU, and the ROCm version 6.2.4, Triton version 3.1.0 with the MI250 GPU. Both GPUs are evaluated on the operating system Ubuntu 20.04.

Attention workload. We evaluate eight Attention algorithm, including four parallel pattern attention (Softmax Attention [24], Sigmoid Attention [18], ReLU Attention [29] and parallel form of multi-scale retention [21]) and four recurrent pattern attention (mamba2 [12], random feature attention [17], retention rucurrent [21], gated retention [22]) For softmax attention, we perform the tests using configuration of LLAMA3.1-8B [15], Deepseek-V2-lite [13] and DiffTransformers-3B [31]. We select the batch size as 1 and 8 and sequence length as 2k, 4k and 8k for attention in large

language models, which are common configurations for these models. Table 2 lists a representative subset of operators as well as their configurations.

Baselines. We compare AttentionEngine with manually implemented attention libraries, such as FlashAttention-v2 [10] and FlashAttention-v3 [19] for Softmax attention, FlashSigmoid [18] for Sigmoid attention, Mamba2 chunk kernel [12] for Mamba2 SSM and Flash-Linear-Attention triton library [30] for gated retention. We also compare with state-of-the-art programming model-based approaches, such as FlexAttention [14] and FlashInfer [32] for transformer attention. We use PyTorch [2] as a default baseline for attention that does not have a manually-implemented library, such as Retention Parallel [21] and ReLUAttention [29].

6.2 Attention Performance on NVIDIA H100

Figure 12 shows the performance of attention performance on NVIDIA H100. The x-axis represents different configs of attention operators, and the y-axis indicates the normalized latency relative to AttentionEngine.

Softmax attention. Figure 12 (a)(b)(d) shows the performance of AttentionEngine and other baselines on Softmax attention from Deepseek-V2-Lite, LLAMA3.1-8B, and Diff-Transformer-3B. Compared with highly optimized libraries, AttentionEngine still obtain significant speedup because of more flexible kernel templates. Compared with highly-optimized FlashAttention, AttentionEngine achieves an average speedup of 1.88× for forward and 1.52× for backward on DeepSeek-V2-Lite and Diff-Transformers-3B, and achieves comparable performance on LLAMA3.1-8B. This improvement stems from AttentionEngine's flexible kernel template to natively support different `headdim_qk` and `headdim_v`, instead of padding them to the same dimension. AttentionEngine also outperforms other programming-model-based approaches such as FlexAttention and FlashInfer, due to our scheduling over different shapes.

Customized transformer attention. Figure 12 (c)(e)(f) shows the performance of AttentionEngine and other baselines on customized transformer attention (Sigmoid attention, ReLU attention and retention parallel). Current expert-optimized libraries lack support for these custom attentions. For example, no fused attention kernel is implemented for ReLU attention and fused Sigmoid attention kernel is not optimized for the latest hardware like NVIDIA H100. AttentionEngine can obtain significant speedup on these customized attentions, achieving 3.6× (1.1× ∼ 10.4×) over FlashSigmoid, PyTorch ReLU attention and PyTorch retention parallel. In addition, compared with programming-model-based approaches, AttentionEngine can support all three customized attention, which demonstrates AttentionEngine's expressive ability and scalability.

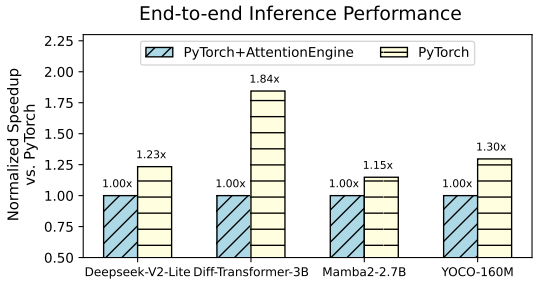

Figure 13: End-to-end inference performance on H100.

Mamba2. Figure 12 (g) represents the linear attention operation of the Mamba2 model: State Space Module. We compare AttentionEngine with the official Mamba2 implementation using Triton. AttentionEngine achieves average speedups of 1.99× and 1.65× over Triton for Mamba2 forward and Mamba2 backward, respectively. This demonstrates the complexity of manually optimizing the attention kernel and the necessity of AttentionEngine.

Retention and gated retention. Figure 12 (h)(i)(j) represents the linear attention operation of RetNet-6.7B, YOCO-13B and RFA-Big.We compare AttentionEngine with Flash-Linear-Attention, which is an expert-optimized linear attention library. The result show that AttentionEngine achieves average speedups of 1.33× and 1.93× for forward and backward, respectively.

6.3 End-to-end Inference on NVIDIA H100

We evaluate the inference latency of large language models like DeepSeek-V2-Lite and Mamba2-2.7B. We show AttentionEngine's applicability to end-to-end inference.

Inference setup. We evaluate end-to-end inference on one NVIDIA H100 GPU. We use Transformers [28] for end-to-end inference, which is the most popular machine learning framework and is backed by PyTorch. We test two models with parallel pattern attention (Deepseek-V2-Lite and Diff-Transformer-3B) and two models with recurrent pattern attention (Mamba2-2.7B and YOCO-160M). We replace the attention operator in these models with AttentionEngine.

Inference performance. As shown in Figure 13, AttentionEngine acheive an average speedup of 1.4× on these models with FP16 precision. These speedup came from our more efficient attention operator. For example, In DeepSpeed-V2-Lite, attention accounts for 85% of the total inference time. We improved the attention operator's speed to 2.2× by supporting different head dimensions for `q`, `k`, and `v`, thereby enhancing the end-to-end performance to 1.85×.

6.4 End-to-end Training on NVIDIA H100

We also evaluate end-to-end training of attention-based model and linear attention-based model to demonstrate AttentionEngine's ability in both forward and backward.

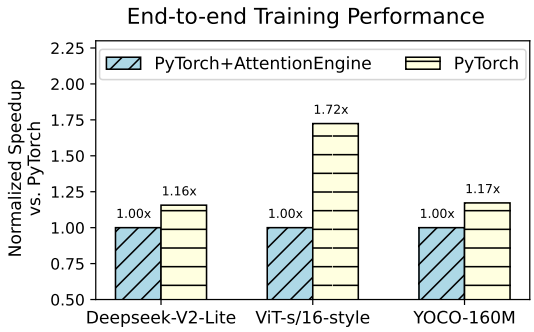

Figure 14: End-to-end training performance on H100.

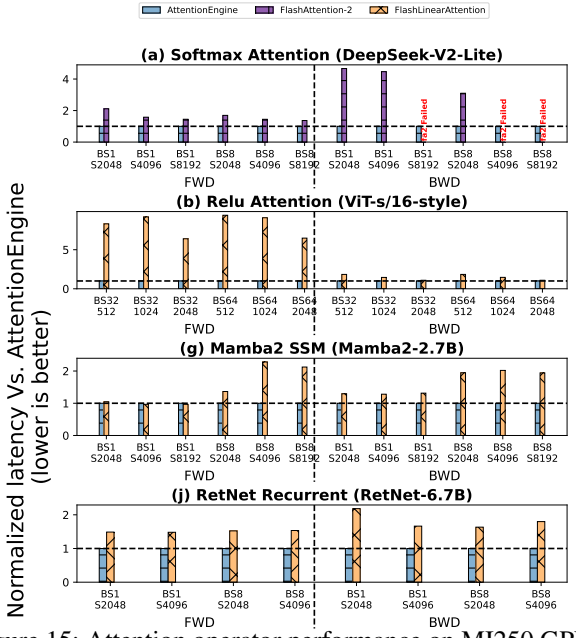

Figure 15: Attention operator performance on MI250 GPUs.

Training setup. We use TRL [25] for training, which is a full stack library based on transformers that provides a set of tools to train transformer language models. Our workloads are Diff-Transformer-3B, YOCO-160M and ViT-S/16 with ReLU attention.

Training performance. As shown in Figure 14, we achieve an average speedup of 1.4× on these models. For ViT-S/16 with ReLU attention, we achieve 1.7× speedup due to the lack of existing libraries for ReLU attention.

6.5 Evaluation on AMD ROCm GPUs

We benchmark the AMD MI250 GPU using a subset of operators selected from the microbenchmark suite originally designed for the NVIDIA H100 GPU, including Softmax Attention, ReLU Attention, Mamba2 and RetNet Recurrent.

Figure 15 shows that AttentionEngine outperforms an average of 3.3× for forward and 2.0× for backward over other baselines across different attention operators. This demonstrated AttentionEngine's ability to support multi-backend.

7 Related Work

Handcrafted attention. High-performance attention mechanisms frequently rely on handcrafted kernel implementations optimized for specific patterns. FlashAttention [11] provides a highly optimized kernel for standard transformer attention, utilizing techniques such as online softmax, memory-efficient fusion, and pipelining. It is implemented using CUTE [9] on NVIDIA GPUs and ComposableKernel on AMD GPUs for low-level optimization. Mamba2 [12], with official kernels developed in Triton [23], focuses on tensor core utilization to enhance efficiency. Flash-Linear-Attention [30], a third-party repository, extends beyond individual methods like Mamba2 and Gated Linear Attention (GLA), offering kernels for a wide variety of linear attention variants.

FlexAttention [14] and FlashInfer [32] aim to simplify the development of attention mechanisms by offering high-level abstractions. However, these approaches primarily focus on elementwise transformations within transformer attention and are exclusively targeted at NVIDIA GPUs. While effective in their domain, their scope is limited, excluding support for linear attention and more advanced optimization strategies. Additionally, their lack of compatibility with AMD GPUs highlights a significant gap in addressing multi-backend requirements.

While these implementations achieve excellent performance, they are restricted to specific attention designs and require substantial manual effort to adapt for new variants. This reliance on handcrafted kernels limits scalability and slows innovation, particularly for emerging attention designs. In contrast, AttentionEngine abstracts the complexity of kernel development, enabling users to define and optimize diverse attention mechanisms without the need for manual implementation. By leveraging a unified programming model and automated optimization pipeline, AttentionEngine supports a broader range of configurations while maintaining competitive performance.

Compiler optimization. Existing DNN compilers, such as TVM [8], Ansor [34], XLA [4], Welder [20], Ladder [27], and TensorRT [1], widely adopt techniques like operator fusion to reduce memory overhead and improve computational efficiency. However, these approaches primarily focus on spatial tiling for regular operators, neglecting the unique challenges and opportunities presented by attention mechanisms. AttentionEngine incorporates common compiler optimization methods, such as fusion and tiling, while extending them to support the irregular computations inherent in attention mechanisms.

AttentionEngine overcomes these limitations by supporting both transformer and linear attention within a single framework. It incorporates advanced scheduling techniques and targets multiple backends, including NVIDIA and AMD GPUs, ensuring high performance and scalability. By unifying diverse attention mechanisms under a comprehensive

programming model, AttentionEngine facilitates the efficient development and deployment of a wide range of attention designs across heterogeneous hardware architectures.

8 Conclusion

Attention mechanisms are central to transformers and large language models (LLMs), driving advancements in natural language processing by capturing contextual relationships. However, their computational demands and growing design diversity pose challenges for scalability and optimization. AttentionEngine addresses these issues by abstracting attention into two core operations, i.e., relevance scoring and aggregation, and introducing customizable templates that combine flexibility with efficiency. With a cross-backend scheduling framework, AttentionEngine automates kernel optimizations, achieving up to 10.4$\times$ speedups for unsupported configurations and providing a foundation for diverse attention designs.

References

[1] NVIDIA TensorRT. https://developer.nvidia.com/tensorrt.

[2] PyTorch. https://pytorch.org/.

[3] Tilelang. https://github.com/tile-ai/tilelang.

[4] XLA. https://www.tensorflow.org/xla.

[5] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, DeVito, et al. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 929–947, New York, NY, USA, 2024. Association for Computing Machinery.

[6] Dzmitry Bahdanau. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[7] Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer, 2020.

[8] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 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, Carlsbad, CA, 2018. USENIX Association.

[9] NVIDIA Corporation. Cutlass: Cuda templates for linear algebra subroutines. https://github.com/NVIDIA/cutlass, 2024.

[10] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

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

[12] Tri Dao and Albert Gu. Transformers are ssms: Generalized models and efficient algorithms through structured state space duality. *arXiv preprint arXiv:2405.21060*, 2024.

[13] DeepSeek-AI, Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, et al. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model, 2024.

[14] Juechu Dong, Boyuan Feng, Driss Guessous, Yanbo Liang, and Horace He. Flex attention: A programming model for generating optimized attention kernels, 2024.

[15] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[16] Yizhao Gao, Zhichen Zeng, Dayou Du, Shijie Cao, Hayden Kwok-Hay So, Ting Cao, Fan Yang, and Mao Yang. Seerattention: Learning intrinsic sparse attention in your llms, 2024.

[17] Hao Peng, Nikolaos Pappas, Dani Yogatama, Roy Schwartz, Noah A. Smith, and Lingpeng Kong. Random feature attention, 2021.

[18] Jason Ramapuram, Federico Danieli, Eeshan Dhekane, Floris Weers, Dan Busbridge, Pierre Ablin, Tatiana Likhomanenko, Jagrit Digani, Zijin Gu, Amitis Shidani, and Russ Webb. Theory, analysis, and best practices for sigmoid self-attention, 2024.

[19] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *arXiv preprint arXiv:2407.08608*, 2024.

[20] Yining Shi, Zhi Yang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Ziming Miao, Yuxiao Guo, Fan Yang, and Lidong Zhou. Welder: Scheduling deep learning memory access via tile-graph. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 701–718, 2023.

[21] Yutao Sun, Li Dong, Shaohan Huang, Shuming Ma, Yuqing Xia, Jilong Xue, Jianyong Wang, and Furu Wei. Retentive network: A successor to transformer for large language models. *arXiv preprint arXiv:2307.08621*, 2023.

[22] Yutao Sun, Li Dong, Yi Zhu, Shaohan Huang, Wenhui Wang, Shuming Ma, Quanlu Zhang, Jianyong Wang, and Furu Wei. You only cache once: Decoder-decoder architectures for language models, 2024.

[23] Philippe Tillet, H. T. Kung, and David Cox. *Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations*, page 10–19. Association for Computing Machinery, New York, NY, USA, 2019.

[24] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[25] Leandro von Werra, Younes Belkada, Lewis Tunstall, Edward Beeching, Tristan Thrush, Nathan Lambert, Shengyi Huang, Kashif Rasul, and Quentin Gallouédec. Trl: Transformer reinforcement learning. https://github.com/huggingface/trl, 2020.

[26] Hongyu Wang, Shuming Ma, Li Dong, Shaohan Huang, Huaijie Wang, Lingxiao Ma, Fan Yang, Ruiping Wang, Yi Wu, and Furu Wei. Bitnet: Scaling 1-bit transformers for large language models, 2023.

[27] Lei Wang, Lingxiao Ma, Shijie Cao, Quanlu Zhang, Jilong Xue, Yining Shi, Ningxin Zheng, Ziming Miao, Fan Yang, Ting Cao, et al. Ladder: Enabling efficient {Low-Precision} deep learning computing through hardware-aware tensor transformation. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 307–323, 2024.

[28] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics.

[29] Mitchell Wortsman, Jaehoon Lee, Justin Gilmer, and Simon Kornblith. Replacing softmax with relu in vision transformers, 2023.

[30] Songlin Yang and Yu Zhang. Fla: A triton-based library for hardware-efficient implementations of linear attention mechanism, January 2024.

[31] Tianzhu Ye, Li Dong, Yuqing Xia, Yutao Sun, Yi Zhu, Gao Huang, and Furu Wei. Differential transformer, 2024.

[32] Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. Flashinfer: Efficient and customizable attention engine for llm inference serving, 2025.

[33] Manzil Zaheer, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. Big bird: Transformers for longer sequences, 2021.

[34] 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. 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, November 2020.