Flex Attention: a Programming Model for Generating Optimized Attention Kernels

Juechu Dong [*1 2] **Boyuan Feng** [*1] **Driss Guessous** [*1] **Yanbo Liang** [*1] **Horace He** [*1]

Abstract

Over the past 7 years, attention has become one of the most important primitives in deep learning. The primary approach to optimize attention is FlashAttention, which fuses the operation together, drastically improving both the runtime and the memory consumption. However, the importance of FlashAttention combined with its monolithic nature poses a problem for researchers aiming to try new attention variants — a “software lottery”. This problem is exacerbated by the difficulty of writing efficient fused attention kernels, resisting traditional compiler-based approaches. We introduce *FlexAttention*, a novel compiler-driven programming model that allows implementing the majority of attention variants in a few lines of idiomatic PyTorch code. We demonstrate that many existing attention variants (e.g. Alibi, Document Masking, PagedAttention, etc.) can be implemented via *FlexAttention*, and that we achieve competitive performance compared to these handwritten kernels. Finally, we demonstrate how *FlexAttention* allows for easy *composition* of attention variants, solving the combinatorial explosion of attention variants.

1 Introduction

High-performance scaled dot product attention kernels, at the heart of Transformers (Vaswani et al., 2017), have become one of the most important building blocks in deep learning. FlashAttention (Dao et al., 2022; Dao, 2024), the standard kernel strategy for attention, fuses the tensor operations in self-attention into one kernel, drastically improving the runtime and memory consumption. These optimizations are essential in enabling efficient training and decoding, particularly for long sequences. Unfortunately, this performance comes at the cost of flexibility. In exchange for the performance and memory benefits, users are restricted to only the handful of popular attention variants supported.

However, new attention mechanisms continue to be a focus of significant research. The goal of these changes vary widely, including reductions in computational complexity (Sliding Window Attention (Beltagy et al., 2020b)), improvements to training stability (Softcapping (Team et al., 2024)), ability to work with sequences of differing lengths (Document Masking), better length extrapolation (ALiBI (Press et al., 2022)), applying attention to other domains such as images (Neighborhood Attention (Hassani & Shi, 2022)), modifications for better inference throughput (PagedAttention (Kwon et al., 2023a)), etc. Furthermore, users often want combinations of these (such as Sliding Window Attention combined with ALiBI), leading to a combinatorial explosion of possible attention kernels. These modifications are not merely speculative either – many of the most popular Large Language Models (LLMs) released use these variants, such as Sliding Window Attention in Mistral-7B, Softcapping in Gemma-2, or ALiBI in MPT-7B.

The importance of FlashAttention combined with its limited ability to support all attention variants poses a problem for researchers aiming to try new attention variants — a ”software lottery” of sorts. If a particular attention variant is not supported by the existing kernels, further exploration can be hindered by slow runtime and memory limitations.

This problem is exacerbated by the difficulty of automatically generating fused attention kernels, resisting traditional compiler-based approaches. In addition to all the standard difficulties with generating efficient matrix multiplication-based primitives on accelerators, there also exist several algebraic rewrites that are difficult for compilers to perform. For example, although

[*]Equal contribution [1]Meta Platforms, Menlo Park, California, USA [2]Computer Science and Engineering Department, University of Michigan-Ann Arbor, Ann Arbor, MI, USA (work done while at Meta). Correspondence to: Horace He <horace@thinkingmachines.ai>.

Proceedings of the 8^{th} *MLSys Conference*, Santa Clara, CA, USA, 2025. Copyright 2025 by the author(s).

Mirage (Wu et al., 2024) is able to generate attention forward from primitive operations, it misses key components for practical usage of attention such as safe softmax or the backwards pass. Furthermore, many variants of attention require block sparsity, a further difficulty for traditional compiler-based approaches.

1.1 Our Approach

We present *FlexAttention*, a novel compiler-driven programming model that allows implementing the majority of attention variants in a few lines of idiomatic PyTorch code.

Flexible Programming Model. Instead of a constrained interface designed with specific attention variants in mind, *FlexAttention* is a general programming model for attention that can be used to define many different flavors of scaled dot product attention. We observe that many attention variants can be defined as a score modification applied on the intermediate score matrix before conducting softmax (Eq. 1). Additionally, a good portion of these modifications take the form of masks, which set part of the score matrix to `-inf`.

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}(\frac{\mathbf{QK}^T}{\sqrt{d_k}})\mathbf{V}$$

$$\text{FlexAttention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}(\text{mod}(\frac{\mathbf{QK}^T}{\sqrt{d_k}}))\mathbf{V} \quad (1)$$

With these insights, we build *FlexAttention*, which takes a score modification callable (`score_mod`) and an attention mask callable (`mask_mod`) in addition to tensor inputs. It enables users to implement a new attention variant by defining how to update scores or calculate boolean masks based on positional information. We demonstrate that many existing attention variants (e.g. Alibi, Document Masking, PagedAttention, etc.) can be implemented via *FlexAttention*. In addition, our programming model allows for easy composition of attention variants via nested `score_mods` and boolean operations between `mask_mods`, solving the combinatorial explosion of attention variants.

Handwritten Template-Based Generation During PyTorch compilation, `score_mod` and `mask_mod` are lowered and codegened into the main loop of a handwritten attention kernel. This abstraction allows us to leverage the performance of hand-written kernels behind the scenes, while still providing significant flexibility in the frontend. The `torch.compile` lowering framework is natively compatible with PyTorch and offers strong flexibility to handle diverse `score_mod` and `mask_mod` defined by the users. Additionally it also supports automatic backward pass generation via `torch.autograd`. We build forward and backward graphs for `score_mod` and `mask_mod`, perform operator fusion and produce kernel code blocks to be integrated with attention kernel templates. Compute buffers are pre-allocated for inputs, outputs and saved intermediate values.

Block Sparsity Optimization. In addition to its semantic changes, masking also introduces sparsity. To take advantage of this, *FlexAttention* leverages block sparsity and employs a pre-computed `BlockMask`. `BlockMask` is a small matrix that tracks block-level sparsity on wether a tiled score matrix block is fully masked out. With the help of `torch.vmap`, our `create_block_mask` utility automatically generates `BlockMask` from user defined `mask_mod`. The `BlockMask` unlocks the opportunity to save the work for fully-masked score matrix blocks without loading a large elementwise attention mask. For partially masked score matrix blocks, we can still exploit `mask_mod` to elementwisely mask out score scalars, thus maintaining the semantics. Additionally, `BlockMask` is implemented as a index vector which can also serve as the mapping used in PagedAttention(Kwon et al., 2023a) (subsection 5.1).

We evaluate *FlexAttention* on 7 popular attention variants and compare its performance against today's most commonly used infrastructures: PyTorch SPDA, and handwritten kernels from FlashAttention (FAv2 (Dao, 2024), FAv3 (Shah et al., 2024) and FAKV(Dao et al., 2023)). We show that *FlexAttention* delivers 0.68x-1.43x the performance of FAv2 and 0.93x-1.45x of FAKV for decoding on conventional attention variants supported by FlashAttention. *FlexAttention* works well with existing ML infrastructure and improves end-to-end performance for inference by 2.04x in gpt-fast for 16k context length and training by 2.4x in torchtune. Additionally, we show that *FlexAttention* supports paged attention at negligible overhead.

2 Background

2.1 Attention Variants

The attention mechanism plays a critical role at the core of Transformers. Each attention layer takes three input tensors: a query $\mathbf{Q} \in \mathbb{R}^{B\times H\times Q\_LEN\times D}$, and a key and a value $\mathbf{K}, \mathbf{V} \in \mathbb{R}^{B\times H\times KV\_LEN\times D}$, where B denotes the batch size, H is the number of attention heads, D is the feature dimension, and Q_LEN and KV_LEN represents the query and key-value lengths, respectively. The attention mechanism first computes a score matrix $\mathbf{S} \in \mathbb{R}^{B\times H\times Q\_LEN\times KV\_LEN}$, which encodes context information by attending each query token to every key token.

$$\mathbf{S} = \text{softmax}(\frac{\mathbf{QK}^T}{\sqrt{d_k}}) \tag{2}$$

Next, attention computes the output as $\mathbf{SV} \in \mathbb{R}^{B\times H\times Q\_LEN\times D}$, weighting the value features by the score matrix.

Building on this foundational algorithm, ML researchers are exploring methods to enhance it by modifying the score matrix $\mathbf{S}$ for more effective and efficient context extraction. For example, neighborhood (Hassani & Shi, 2022) and sliding window (Beltagy et al., 2020b) attention attends each query token only to its neighboring key tokens to reduce computational and memory complexity for large images and long sequences. Softcapping (Team et al., 2024) adds a tanh layer to prevent excessive growth of logits. Alibi (Press et al., 2022) embeds an elementwise bias in the score matrix that penalizes distant tokens to allow models trained on short inputs to perform on long prompts. This large design space motivates efficient compiler designs for facilitating the training and inference of attention variants.

2.2 State-of-the-art Attention Implementations

Many kernels have been manually implemented to efficiently support attention. Among these, the IO-aware FlashAttention (Dao, 2024) has become one of the most widely adopted solutions. FlashAttention significantly reduces memory access and achieves substantial speedups. Specifically, it avoids materializing the large score matrix $\mathbf{S}$ and computes it on the fly. Recently, Flash Attention v3 has been released to further accelerate attention by leveraging advanced hardware features and manually tuning the performance. While flash attention delivers state-of-the-art performance, it is specifically tailored towards a limited selection of attention variants (Table 1). One recent work FlashMask (Wang et al., 2024) extends Flash Attention with a column-wise sparse representation to support more mask designs. However, it still lacks of flexibility in terms of score modifications and adds large overhead for complex masks. Given that attention variants exhibit diverse computational characteristics such as sparsity and locality, significant manual effort is required to adapt existing attention implementations to support numerous attention variants. This lack of flexibility and efficient kernels has become a barrier for machine learning researchers seeking to explore novel attention variants.

2.3 Machine Learning Compilers

Many compilers have been built to accelerate machine learning workloads. torch.compile (Ansel et al., 2024) uses TorchDynamo to capture computation graph from an arbitrary code and TorchInductor to optimize the graph. TVM (Chen et al., 2018) allows users to describe the computation of a kernel and generate optimized code. Mirage (Wu et al., 2024) utilizes μgraph to explore operator-level optimization opportunities. However, these machine learning compilers fail to deliver a satisfactory performance on attention variants due to the special computing patterns in attention. First, attention requires fusing two matrix-matrix multiplications (i.e., QK^T and SV) while existing machine learning compilers usually focuses on optimizing one matrix-matrix multiplications. Second, as shown in Flash Attention (Dao, 2024), online softmax significantly reduces memory access and improve performance. Since online softmax is tailored towards attention designs, it is not well supported by existing general machine learning compilers. To the best of our knowledge, we are the first to compile arbitrary attention variants while delivering state-of-the-art performance.

3 Front-end Design and Implementation

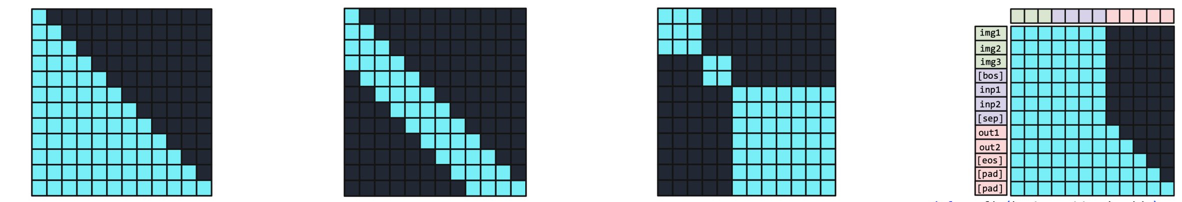
In this section, we propose a unified abstraction of diverse attention variants. This abstraction enables programmers to easily express attention semantics without worrying about implementation details and kernel performance.

3.1 Unified Abstraction

Despite the numerous attention variants designed by machine learning researchers, we unify these variants into two commonly shared patterns. The first pattern masks out features from certain tokens to manage contextual relationships. For example, a causal mask forces a model to predict based on previous tokens and ignore future tokens. A sliding window mask (Beltagy et al., 2020b) forces a model to concentrate on nearby tokens for the local context.

The second pattern further adjusts the attention score among tokens in a fine-grained style. For example, *Alibi* (Press et al., 2022) adjusts the attention score based on the relative position of two tokens, which helps the model to better understand the context and handle long sequences. It also adds bias to attention scores based on head dimension, which specializes heads on different types of information.

With these insights, we design a unified abstraction



```
def causal_mask(b, h, q, kv):
    return q >= kv
```

```
def sliding_window_mask(b, h, q, kv):
    return q <= kv + window and q >= kv
```

```
def document_mask(b, h, q, kv):
    return doc_id[q] == doc_id[kv]
```

```
def prefix(b, h, q_idx, kv_idx):
    return kv_idx <= prefix_length[b]
prefixLM = or_mask(prefix, causal_mask)
```

Figure 1. Attention variants and `mask_mod` examples, including causal mask, sliding window mask, document mask, and prefixLM mask.

including a score modification callable (`score_mod`) and an attention mask callable (`mask_mod`). It enables users to build a new attention variant by defining how to update scores or calculate boolean masks based on positional information in idiomatic PyTorch code.

```
def mask_mod(batch_idx: int, head_idx: int, q_idx
    : int, kv_idx: int) -> bool

def score_mod(score: T, batch_idx: int, head_idx:
     int, q_idx: int, kv_idx: int) -> T
```

Specifically, consider a score matrix $\mathbf{S} \in \mathbb{R}^{B\times H\times M\times N}$. `mask_mod` takes the input position and specifies whether the corresponding score scalar is set to `-inf`. `score_mod` additionally takes the score scalar of arbitrary type T (e.g., `bfloat16` or `float32`) and applies updated to the score based on its position.

While these two patterns generalize many attention variants, it does not yet accommodate fundamentally paradigm changes beyond scaled-dot-product-attention, such as the Differential Transformer (Ye et al., 2025).

Mapping to concrete examples Our unified abstraction captures the two important patterns in attention variants and enables automated compiler optimizations. Figure 1 demonstrates 4 example attention variants, each of which has been widely studied and optimized. However, there is a lack of a unified optimization approach that applies to all of them. Instead, our abstraction provides full flexibility in specifying masking and modification to the score matrix, enabling machine learning researchers to easily explore novel attention variants.

As we discussed earlier, a `causal mask` forces a token's query feature to only attend to key and value features from previous tokens. This leads to a simple expression of $q\_idx \geq kv\_idx$. Since the causal mask is not related to the batch and head dimension, the `mask_mod` does not involve b and h inputs.

A `sliding window mask` requires a token to only attend to nearby tokens within a certain sliding window size. We can easily describe such requirement as $q\_idx - kv\_idx \leq window$.

A `document mask` trains multiple documents simultaneously and requires a token to only attend to other tokens in the same document. We can record 'document_id' as a mapping from the token index to the document index, and keep the score only if $document\_id[q\_idx] == document\_id[kv\_idx]$.

A `alibi bias` modifies each score scalar by adding a bias scalar, which scales with the distance between the query index and the key value index. We can easily describe it as $score + alibi\_bias[h] * (q\_idx - kv\_idx)$. Note that we can easily support other score modifications such as scaling the score scalar, softcapping or even non-linear transformations.

Why `mask_mod`? `mask_mod` is a special case of `score_mod` and any `mask_mod` can be easily converted to a `score_mod` semantically. However, we still observe two fundamental reasons for distinguishing these two APIs. First, `score_mod` applies expensive modification to every score scalar. When converting `mask_mod` to `score_mod`, we multiply each score with 1 or 0, leading to extra overhead. Second, comparing with `score_mod`, `mask_mod` provides extra semantic information that certain score computations can be skipped. We exploit this information to further accelerate attention computation, as detailed in subsection 4.2.

3.2 Logical Fusion for Composability

As many attention variants have been designed, one recent trend is to compose existing attention variants to further extract semantic information. We support the logical fusion to enable the composability of mask designs via `and_mask` and `or_mask`. By taking two `mask_mod` functions, we can automatically compose them by applying elementwise logical operations. The generated `mask_mod` can be further combined with other `mask_mod` which significantly mitigates the programmability burden when designing attention variants.

Figure 1 shows an example of logical fusion. `PrefixLM` (Raffel et al., 2023) performs full bidirectional attention on prefix inputs and causal attention on the rest. Instead of complex conditional branches, we can build a simple `prefix mask` and compose it with a causal mask via `or_mask`.

Additionally, `mask_mod` functions can be nested to construct more complex masking patterns – for example, Neighborhood Attention with Tiled Mapping (NATTEN); see AppendixA.1.

4 Backend Design and Implementation

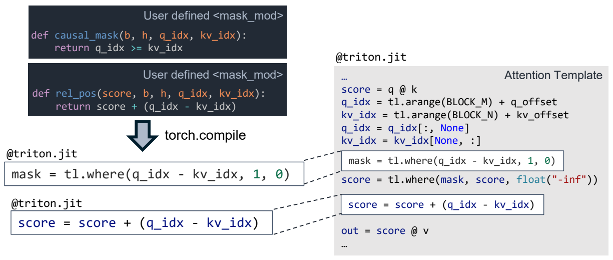
In this section, we propose a backend design that compiles attention variants to efficient kernels.

4.1 Template-based Lowering

We build a template-based lowering system that generates optimized GPU kernels, combining the flexibility with state-of-the-art attention optimizations. The key innovation of our approach lies in recognizing that attention variants primarily differ in their pointwise score modifications, allowing us to templatize the common computational patterns. To this end, we build template-based lowering to capture the common patterns and apply `score_mod` elementwisely on score matrix. We will present BlockSparsity design on exploiting sparsity from `mask_mod` in subsection 4.2.

Template-based lowering first exploits TorchDynamo (Ansel et al., 2024) to capture the computation graph of `score_mod` and `mask_mod`. As shown in Figure 1, these two functions are usually lightweight in terms of computation and memory access, and could be fused with other computation in attention. Then, we build a highly optimized triton template for high performance. The Triton kernel design adopts established optimization techniques for fused attention kernels, including online softmax, careful GPU occupancy management, efficient memory handling via partitioning and broadcasting, and specialized support for grouped query attention (GQA) (Ainslie et al., 2023).

We capture these advanced techniques in three handwritten attention kernel templates (forward, backward, and decoding). The templates are designed to accept custom score modification code blocks, which are generated from captured score_mod and mask_mod operations using torch.compile. TorchInductor translates these subgraphs into Triton code, dynamically injecting both forward and backward score modification operations into the predefined templates at runtime. This approach results in attention kernels that closely match the performance of manually optimized, variant-specific



```
def causal_mask(b, h, q_idx, kv_idx):
    return q_idx >= kv_idx
```

```
def rel_pos(score, b, h, q_idx, kv_idx):
    return score + (q_idx - kv_idx)
```

```
@triton.jit
mask = tl.where(q_idx - kv_idx, 1, 0)
```

```
@triton.jit
score = score + (q_idx - kv_idx)
```

```
@triton.jit
…
score = q @ k
q_idx = tl.arange(BLOCK_M) + q_offset
kv_idx = tl.arange(BLOCK_N) + kv_offset
q_idx = q_idx[:, None]
kv_idx = kv_idx[None, :]
mask = tl.where(q_idx - kv_idx, 1, 0)
score = tl.where(mask, score, float("-inf"))
score = score + (q_idx - kv_idx)

out = score @ v
…
```

Figure 2. Compilation from user-defined attention patterns to optimized Triton kernels. **Top:** User-defined PyTorch functions specify custom masking (`causal_mask`) and score modification (`rel_pos`) operations. **Bottom:** `torch.compile` captures and lowers these operations into Triton primitives. **Right:** The resulting Triton code blocks are integrated into our attention template, which handles core attention computations while preserving the user-defined modifications.

implementations. Figure 2 illustrates our lowering pipeline, showing how user-defined PyTorch operations are translated into optimized Triton code and integrated into our attention templates.

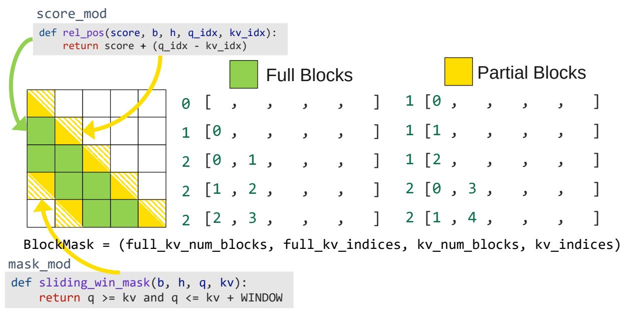
4.2 Block Sparsity

Attention variants usually show high sparsity such as 50% sparsity from causal mask and significantly higher sparsity from sliding window mask. One natural question is,

could our compiler exploit such sparsity for acceleration?

One naive approach is to check during runtime whether a position is masked out and skip the computation. However, this adds a large runtime overhead by iterating through all scalars even if it is masked out. Another approach is to pre-compute a mask tensor of shape $B \times H \times Q\_LEN \times KV\_LEN$ and compute a score scalar according to the mask tensor. However, this mask tensor adds significant memory overhead, which contradicts the principle of flash attention that avoids realizing the score matrix.

BlockMask FlexAttention implements a BlockMask data structure to exploit the sparsity while adding only negligible memory overhead. We pre-compute a block mask during compilation time to remove the runtime overhead. BlockMask first splits the score matrix into blocks along the Q_LEN and KV_LEN dimensions. Then, BlockMask specifies a block as non-computed if all score scalars in it are masked as `-inf`. By recording sparsity at the block level, we do not realize a large sparsity matrix, significantly reducing memory overhead.


score_mod
```
def rel_pos(score, b, h, q_idx, kv_idx):
    return score + (q_idx - kv_idx)
```

```
BlockMask = (full_kv_num_blocks, full_kv_indices, kv_num_blocks, kv_indices)
```
mask_mod
```
def sliding_win_mask(b, h, q, kv):
    return q >= kv and q <= kv + WINDOW
```

Figure 3. BlockMask for Sliding Window Attention. **Left:** The score matrix is divided into fully visible blocks (green), partially masked blocks (yellow), and oblivious blocks (white). **Right:** `BlockMask` encoded as an index matrix plus block count per row. **Top:** `score_mod` is applied to full & partial blocks. **Bottom:** `mask_mod` is applied to partial blocks.

Concretely, BlockMask contains two tensors, a `kv_num_block` of shape $B \times H \times Num\_Row$ and a `kv_indices` of shape $B \times H \times Num\_Row \times Num\_Col$. Here, Num_Row and Num_Col are the number of blocks in each row and column along the query and key value dimension, respectively. `kv_num_block` stores the number of non-zero blocks for each row and `kv_indices` stores the indices of these non-zero blocks. By accessing `kv_num_block`, *FlexAttention* skips masked blocks and achieves proportional performance speedup.

Full Block Optimization We split BlockMask into full blocks and partial blocks to further improve the performance. Our key idea is to minimize runtime overhead caused by applying `mask_mod`, skipping it whenever possible. We identify two types of blocks in terms of sparsity.

1. Partial blocks where some score scalars are masked as `-inf`, necessitating the application of `mask_mod` elementwise at runtime.

2. Full blocks where no score scalar is masked, allowing us to skip `mask_mod` and apply only `score_mod`.

This optimization yields approximately a 15% performance improvement for common patterns such as causal masks. Figure 3 illustrates the BlockMask for sliding window attention. The `sliding_win_mask` is applied only to partial blocks while the `rel_pos` score modification is applied elementwise on both full and partial blocks.

BlockMask Guided Indirect Memory Access Exploiting sparsity in attention shows significant performance speedup. Existing solutions such as FlashAttention iterate through the KV_LEN dimension and manually specifies the start and end index of iteration based on attention variants. This leads to a significant programmability burden. Moreover, it requires extra efforts to ensure that attention scores in the final block are properly masked if the query token index is less than the key token index.

FlexAttention utilize the BlockMask information to automatically exploit the sparsity in attention variants. It first adjusts the workload in each GPU block according to the `kv_num_block`, which specifies the number of score matrix blocks that are not masked out. Then, it uses the KV Indices to map to the next block to be processed. Here, we utilizes an indirect memory access strategy since the indices need not point to contiguous tokens in the sequence. This provides flexibility in compiling various attention patterns, such as sliding window attention, local-global attention, or custom sparse patterns, without modifying kernel.

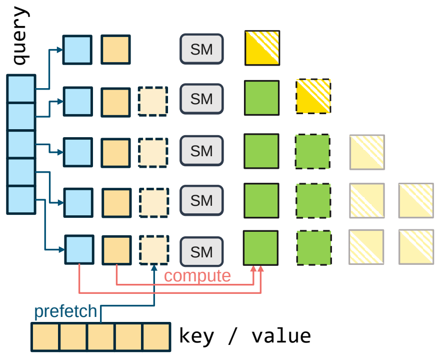


Figure 4. Scheduling full and partial blocks to SM.

Data Prefetching Pipeline *FlexAttention* divides the score matrix into tiles along the Q_LEN dimension to parallelize the computation on multiple Stream Processors(SM). Each SM iterates along the KV_LEN dimension and processes a row of blocks (Figure 4). Special care needs to be taken in pipelining the data fetch and compute for these blocks. When the first score block is being computed from the first kv tile, the second kv tile (shown in dashed borders) is being pre-fetched from HBM to SRAM. Our BlockMask access strategy removes conditional branch on checking whether a score scalar is masked out, thus allowing efficient pipelining and hiding data access latency between iterations.

Overhead Analysis BlockMask representation achieves memory efficiency through a careful balance of granularity and overhead. Although we store auxiliary tensors to encode the sparsity pattern, their memory footprint scales with $O(\lceil Q\_LEN/BS\rceil \times \lceil KV\_LEN/BS\rceil)$, where BS is the block size (=128 by default). This is significantly less than the $O(M \times N)$ required for the full score matrix.

5 Case Study

5.1 Paged Attention Variants

PagedAttention (Kwon et al., 2023a) has been widely deployed for inference on a batch of sentence requests to reduce GPU memory consumption. However, the current PagedAttention design requires manually rewriting the attention kernel to support page table lookup for its irregular memory accesses. Additionally, it is tightly coupled with specific attention mask designs, limiting composability with other attention variants. In this subsection, we explain how *FlexAttention* enables paging support for arbitrary attention variants without manually rewriting the kernels, while still delivering high performance and low memory usage. [1]

Page Table. PagedAttention introduces a page table to efficiently manage the KV cache and reduce its memory usage. As shown in Figure 5(a), a logical KV cache stores the key and value features of each sequence, typically in shape $B \times Max\_len \times D$ where B is the batch size, Max_len is the maximum context length, and D is the feature dimension. Since sequence lengths vary significantly between sentences and could be much shorter than Max_len, a substantial portion of the logical KV cache remains unused. To address this fragmentation issue, PagedAttention instead allocates a physical KV cache of shape $1 \times Max\_token \times D$, where Max_token is the maximum number of tokens across all sentences combined. Storing sentences in the physical KV cache reduces fragmentation and enables memory optimizations such as KV block sharing between sentences.

[1]Due to the scope of this paper, we focus on supporting KV cache in GPU memory and leave the memory swapping to host disk as future work.

PagedAttention maintains a page table as 2D matrix that maps the batch index and logical KV index to the corresponding physical KV index. Within the attention kernel, one `scatter` operation is used efficiently translate indexes using this matrix, while minimizing memory access overhead. During runtime, once a sentence reaches its end token, we clear the corresponding row in the page table and update it with a new request.

Fused indirect memory access. Having attention kernel compute on the physical KV cache instead of the logical KV cache, presents several challenges. First, the page table adds an extra layer of indirection in memory accesses, which existing designs typically handle with manually rewritten CUDA kernels. Second, the page table imposes a significant programming burden on already complex implementations for supporting diverse attention variants.

FlexAttention tackles this problem through the BlockMask conversion, eliminating the need for kernel rewrites. Our key idea is that since BlockMask already incorporates one layer of indirect memory access to avoid unnecessary computations (subsection 4.2), we can merge it with the indirect memory access from the page table to further skip unnecessary memory accesses. Specifically, as shown in Figure 5(b), given a BlockMask of an attention variant, we take the kv_index and map the logical block index to the corresponding physical block index according to the page table. During runtime, *FlexAttention* relies on the converted kv_index to access tokens for a specific sentence from the physical KV cache. Note that we keep the kv_num_blocks unchanged since paged attention does not change the number of unmasked blocks.

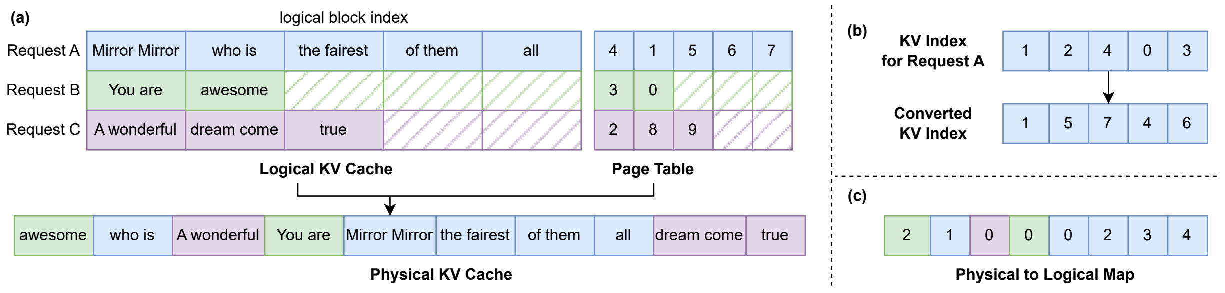


Figure 5. Illustration of *FlexAttention* with paged attention. a) Update physical KV cache with a page table; b) Convert a block mask especially KV Index; c) Physical to logical map for converting `mask_mod` and `score_mod`.

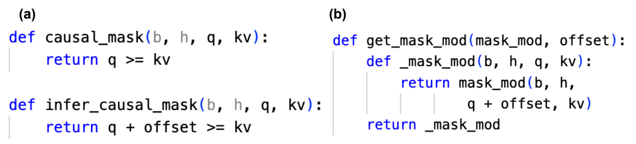
`mask_mod` and `score_mod` Conversion *FlexAttention* supports attention variants with `mask_mod` and `score_mod`, which utilizes position information (e.g., logical KV index) to specify mask and score modifications. This position information is changed when using paged attention and operating on the physical KV cache. One naive approach is to manually rewrite the `mask_mod` and `score_mod` to cater to such changes on position information. However, this adds an large programmability burden, considering numerous attention variants and even logical fusion of these variants (subsection 3.2).

We automatically compile the `mask_mod` and `score_mod` to support paged attention.

```
def converted_mask_mod(batch_idx: int, head_idx:
    int, q_idx: int, physical_kv_idx: int)
    -> bool

def converted_score_mod(score: T, batch_idx: int,
     head_idx: int, q_idx: int, physical_kv_idx:
    int) -> T
```

Specifically, we maintain a vector mapping physical block indices to logical block indices using $O(1)$ overhead when updating the page table. Given the physical KV token index, we can first compute the physical KV block index and offset due to the fixed block size. Then, we can lookup the corresponding logical KV block index and regenerate the logical KV token index. Finally, we call the user-provided `mask_mod` and `score_mod` with the generated logical KV token index.


(a)

```
def causal_mask(b, h, q, kv):
    return q >= kv

def infer_causal_mask(b, h, q, kv):
    return q + offset >= kv
```

(b)

```
def get_mask_mod(mask_mod, offset):
    def _mask_mod(b, h, q, kv):
        return mask_mod(b, h,
                q + offset, kv)
    return _mask_mod
```

Figure 6. `mask_mod` conversion

5.2 Modification Conversion for Inference

In attention variants, the mask and score modifications may change with query indices, which is supported by taking *q_idx* in `mask_mod` and `score_mod`. However, LLM inference usually iteratively takes one query token at a time and requires additional context information in terms of the number of query tokens has been processed, namely offset. Figure 6(a) shows the difference between causal mask during training and inference. To tackle this problem, we automatically convert `mask_mod` and `score_mod` to their inference counterpart, as shown in Figure 6(b). Specifically, we provide a decorator that takes the user-defined `mask_mod` and an offset, and generate a converted `mask_mod` that consumes the offset information.

6 Evaluation

6.1 Experiment Setup

We evaluate the performance of *FlexAttention* on 7 popular attention modifications (Table. 1), including the classic `noop` and `causal`, popular positional embedding `alibi` (Press et al., 2022), local attention `sliding_window` (Beltagy et al., 2020a), fully visible prefix `prefixLM`, additional tanh layer `soft_cap` and `document_mask` for batching input of various lengths. These attention variants are evaluated as Multi-head Attention (MHA) and Grouped Query Attention (GQA).

These attention variants have varying levels of support across our five baselines. **FlashAttention-v2** (`FAv2`) (Dao, 2024) is a state-of-the-art, high-performance attention kernel optimized to reduce HBM read/write through kernel fusion, tiling and recomputation. **FlashAttention-v3** (Shah et al., 2024) (`FAv3`) is an experimental kernel that leverages new hardware features in Hopper GPUs to further enhance performance. FlashAttention also provides a **FlashDecoding** (`FAKV`) kernel optimized for inference with kvcache support. Scale dot-product attention (SDPA) is the native PyTorch functional API for attention, supporting **math**, **memory-efficient attention** (`mem_efficient`) (Rabe & Staats, 2022), FAv2, and **cuDNN** backends. SDPA integrates smoothly into the PyTorch framework, enabling optimizations and features such as `NestedTensor`, `torch.compile`, and CUDA graph support.

We evaluate end-to-end training and inference performance on LLaMa3 and LLaMa3.1 models (Dubey et al., 2024) using `gpt-fast` (gpt-fast maintainers & contributors, 2023) and `torchtune` (torchtune maintainers & contributors, 2024). `gpt-fast` and `torchtune` are native PyTorch libraries that provide accessible finetuning and inference recipes for popular models They rely on `torch.compile` to enable optimizations, including CUDA graphs, kernel fusion and inlining, matmal templates, parameter freezing etc., and use SDPA for attention by default.

Our experiments are conducted on Nvidia H100 GPUs with power capped at 650W and memory bandwidth limited to 2.4TB/s, Nvidia A100 GPUs with power capped at 330W, and Nvidia A6000 GPUs.

6.2 Attention Kernel Performance.

We characterize *FlexAttention* kernel performance by benchmarking across varying sequence lengths, attention variants, and numbers of heads, with kv size fixed at 256 MiB, head dimension at 64, and data type set

	FAv2	FAv3 (c1d146c)	SDPA - cuDNN (v9.1.1)	SDPA - mem_efficient	*FlexAttention*
noop	native	native	native	native	native
causal	native	native	native	native	native
alibi_bias	native	x	itemized mask	itemized mask	native
sliding_win	native	native	itemized mask	itemized mask	native
prefix_lm	x	x	itemized mask	itemized mask	native
soft_cap	native	x	x	x	native
various lengths	native	native	jagged tensors	jagged tensors	native (doc_mask)
neighbor attention	x	x	x	x	native

Table 1. Tested Attention Variants and their Level of Support in FlashAttention, SDPA & *FlexAttention*

to bfloat16.

Training Performance. As shown in Figure 7, assuming a causal mask, *FlexAttention* yields a consistent 1.00x-1.22x speedup in the forward pass and 0.86x-1.05x speedup in the backward pass compared to `FAv2` across different sequence lengths. For the 7 attention variants we evaluated, *FlexAttention* achieves a 0.68x-1.43x speedup relative to `FAv2` when `FAv2` supports the variant. For the variants lacking native support, *FlexAttention* achieves a 5.49x-8.00x speedup compared to SDPA kernels with itemized attention masks by com-

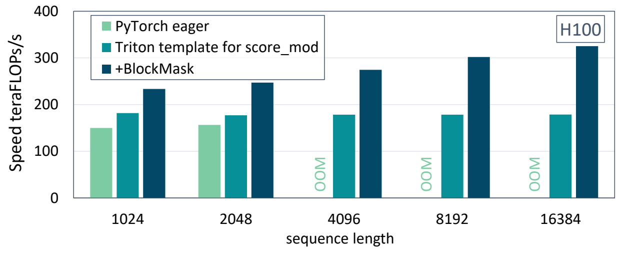
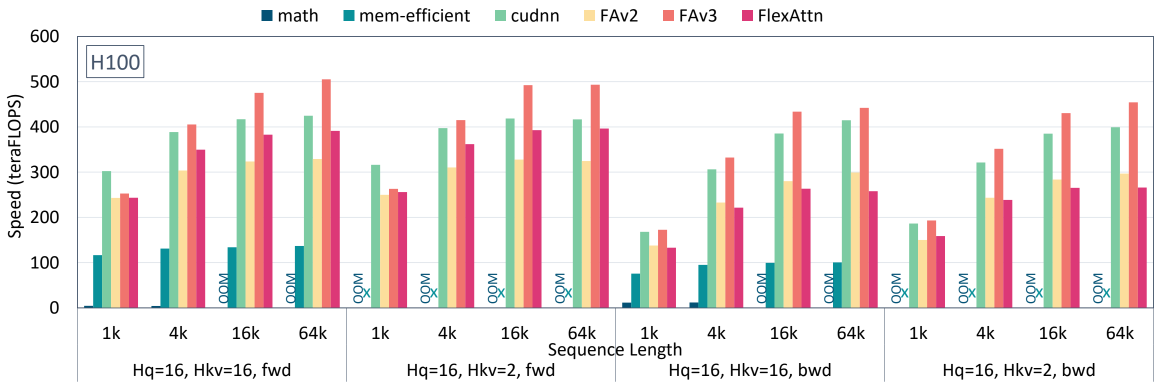


Figure 8. FlexAttention Ablation Study: Forward Kernel Speed for Causal Attention on Eager PyTorch, PyTorch with Triton template, and BlockMask



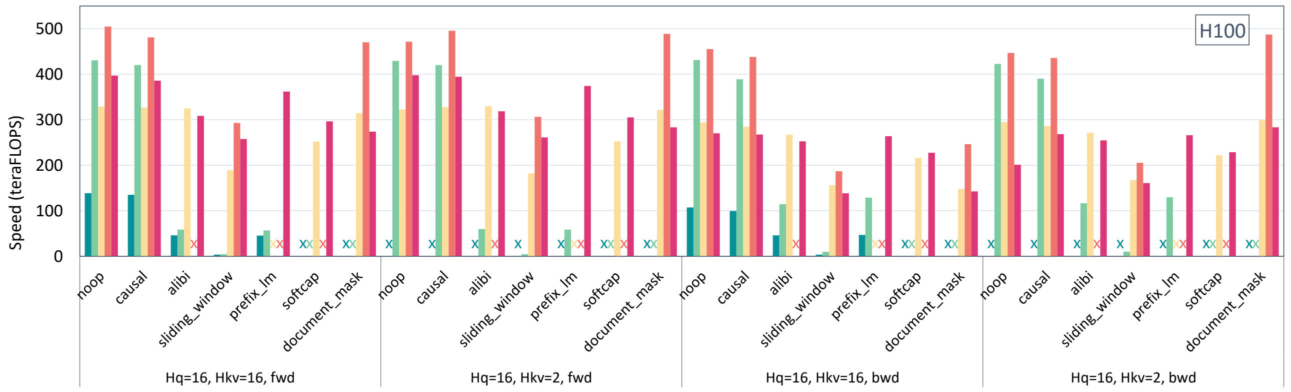
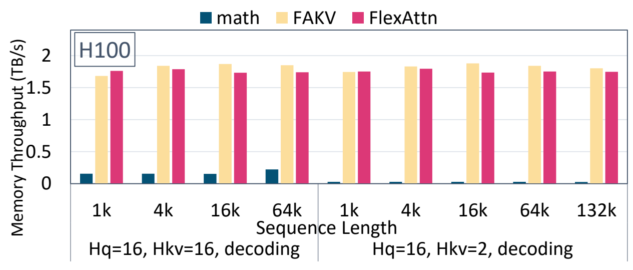


Figure 7. Attention Kernel Speed: Forward and Backward. **Top:** Causal Mask on QKV Length Ranging from 1k to 64k w/wo GQA. **Bottom:** Different Attention Variants on 16k-token-long QKV w/wo GQA.



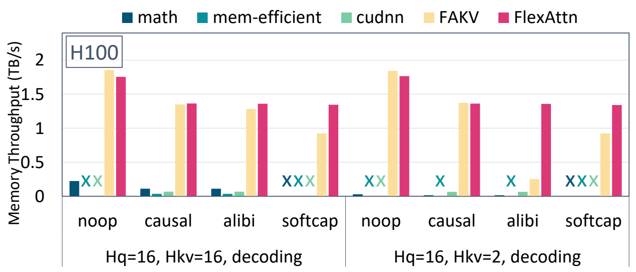


Figure 9. Attention Kernel Speed: Decoding for 1 Query Token. **Left:** Classic Attention on KV Length Ranging from 1k to 132k w/wo GQA. **Right:** Different Attention Variants on 16k-token-long KV cache w/wo GQA.

puting the mask from `mask_mod` at runtime, thereby avoiding the need to realize and load the itemized mask.

In Figure 8, we conduct an ablation study and demonstrate that BlockMask boosts the performance of a causal attention kernel by an average of 1.3x - 1.8x.

Inference Performance. As shown in Figure 9, for a query length of 1, *FlexAttention* delivers decoding performance comparable to FlashDecoding (`FAKV`), achieving a 0.93x-1.45x speedup, except one outlier: ***FlexAttention* is 5.37x faster than `FAKV` when using GQA with `alibi`**. This is an example of the "software lottery", where `FAKV` lacks manual optimization for GQA with `alibi`, and its fallback solution provides only 1/5 of the optimal performance. In contrast, *FlexAttention* maintains consistent performance for this combination without manual tuning. Due to page limit, we leave Neighborhood Attention results in Appendix A.1.

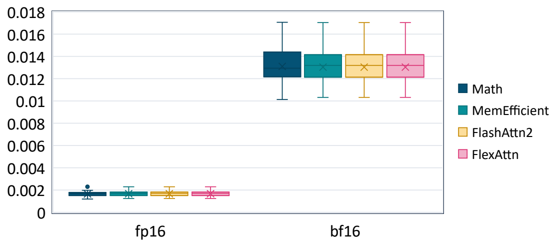


Figure 10. Root-mean-square error (RMSE) of bf16 and fp16 attention output compared to fp64 golden result.

Numeric Accuracy. *FlexAttention* does not introduce additional numeric errors compared to our baselines. (Figure 10)

6.3 End-to-end Performance.

FlexAttention boosts end-to-end training performance by over 2.4x and inference performance by up to 2.04x, and this speedup scales well with sequence length. We replace SDPA with *FlexAttention* in `gpt-fast` and `torchtune` libraries and evaluate performance on LLaMa3 and LLaMa3.1 models. We show that *FlexAttention* integrates well with the PyTorch framework in `gpt-fast` and `torchtune`, enabling optimizations such as CUDA graphs, parameter freezing, and kernel fusion, same as SDPA.

Training Performance of `torchtune`. We set up `torchtune` to fine-tune LLaMa3-8B on the Alpaca (Taori et al., 2023) dataset. To efficiently process the input sequences of different lengths, `torchtune` concatenates them into jagged long sequences of fixed length. This approach requires a document mask that allows each input sequence to attend to itself while ignoring its neighbors. SDPA utilizes a precomputed boolean mask of size $B \times N \times N$, where B is batch size and N is the sequence length. As shown in Figure 11, the cost of accessing this boolean mask grows quadratically, leading to a 25% decrease in the training throughput when the sequence length increases from 2k to 8k. In contrast, *FlexAttention* employs a `BlockMask` and a document ID tensor of size $B \times N$ and scales effectively with sequence length.

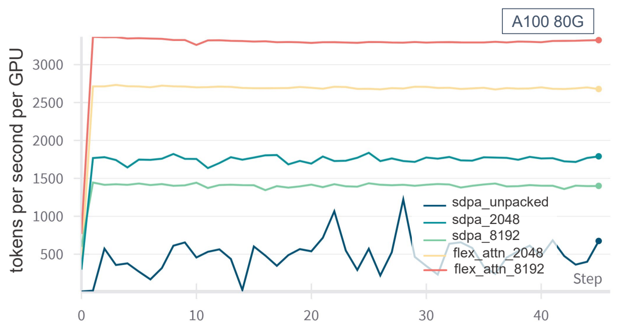


Figure 11. `torchtune` Training Throughput on llama3-8B.

Inference Performance of `gpt-fast`. In Figure 12, we show that *FlexAttention* boosts LLaMa3.1-8B serving performance by 1.22x-2.04x, and LLaMa3.1-70B performance by 0.99x - 1.66x compared to SDPA. The speedup increases as context length grows and the attention kernel increasingly dominates the computation in each iteration.

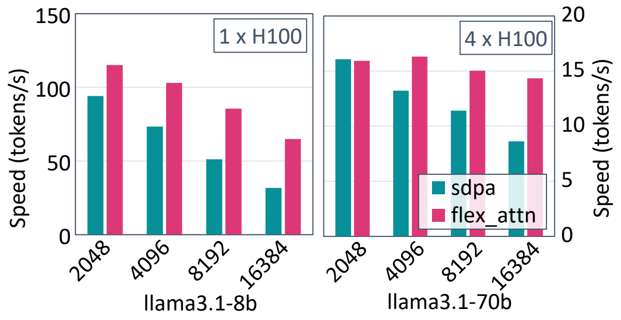


Figure 12. `gpt-fast` Inference Speed on LLaMa3.1-8B and 70B.

6.4 Case Study: Paged Attention

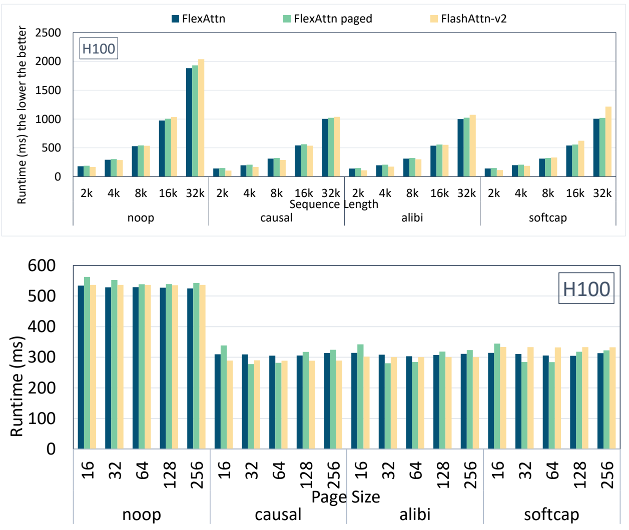


Figure 13. Runtime w/ and w/o paged attention (the lower the better). **Top:** Latency under diverse sequence length. **Bottom:** Latency under diverse page size.

While paged attention stores sentence requests of varying length in a compact physical KV cache, one major question is whether it would introduce high runtime overhead. Figure 13(a) shows runtime latency of flex attention with and without paged attention, as well as FlashAttn-v2. We show runtime latency under changing sequence length, while keeping other dimension the same with batch size of 32, head dimension of 64, and number of heads as 16. Overall, we observe less than 1% runtime overhead on average when using *FlexAttention* with paged attention, which is significantly smaller than the 20–26% higher attention kernel overhead reported in vLLM (Kwon et al., 2023b). The reason is that we do not introduce any kernel changes and rely on a fused indirect memory access to support *FlexAttention* with paged attention. Surprisingly, we even observe that *FlexAttention* with paged attention is faster than FlashAttn-v2 without paged attention on large sequence lengths, which demonstrates the scalability of our design.

Figure 13(b) demonstrates the impact from page size ranging from 16 to 256. Overall, we do not observe significant performance impact from changing page sizes. Note that we manage physical KV cache in GPU global memory and do not swap memory to host disk, which mitigates the disk access overhead.

7 Conclusion

In this paper, we propose *FlexAttention*, a programming model for generating optimized attention kernels. While researchers continue to design new attention variants, they are often limited by the lack of hand-tuned kernels, leading to a large programmability and performance burden. We hope that FlexAttention will allow researchers to explore new attention variants without being held back by what handwritten kernels support.

References

Ainslie, J., Lee-Thorp, J., de Jong, M., Zemlyanskiy, Y., Lebrón, F., and Sanghai, S. Gqa: Training generalized multi-query transformer models from multi-head checkpoints, 2023. URL `https://arxiv.org/abs/2305.13245`.

Ansel, J., Yang, E., He, H., Gimelshein, N., Jain, A., Voznesensky, M., Bao, B., Bell, P., and 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, pp. 929–947, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400703850. doi: 10.1145/3620665.3640366. URL `https://doi.org/10.1145/3620665.3640366`.

Beltagy, I., Peters, M. E., and Cohan, A. Longformer: The long-document transformer, 2020a. URL `https://arxiv.org/abs/2004.05150`.

Beltagy, I., Peters, M. E., and Cohan, A. Longformer: The long-document transformer, 2020b. URL `https://arxiv.org/abs/2004.05150`.

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Cowan, M., Shen, H., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. Tvm: an automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pp. 579–594, USA, 2018. USENIX Association. ISBN 9781931971478.

Dao, T. Flashattention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations (ICLR)*, 2024.

Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and Ré, C. Flashattention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

Dao, T., Haziza, D., Massa, F., and Sizov, G. Flashdecoding for long-context inference, 2023. URL https://crfm.stanford.edu/2023/10/12/flashdecoding.html. Accessed: 2024-09-15.

Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., and Akhil Mathur, e. a. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

gpt-fast maintainers and contributors. Accelerating generative ai with pytorch ii: Gpt, fast, November 2023. URL https://github.com/pytorch-labs/gpt-fast.

Hassani, A. and Shi, H. Dilated neighborhood attention transformer, 2022. URL https://arxiv.org/abs/2209.15001.

Hassani, A., Walton, S., Li, J., Li, S., and Shi, H. Neighborhood attention transformer. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2023.

Hassani, A., Hwu, W.-M., and Shi, H. Faster neighborhood attention: Reducing the $o(n^2)$ cost of self attention at the threadblock level, 2024. URL https://arxiv.org/abs/2403.04690.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023a.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023b.

Press, O., Smith, N. A., and Lewis, M. Train short, test long: Attention with linear biases enables input length extrapolation, 2022. URL https://arxiv.org/abs/2108.12409.

Rabe, M. N. and Staats, C. Self-attention does not need $o(n^2)$ memory, 2022. URL https://arxiv.org/abs/2112.05682.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer, 2023. URL https://arxiv.org/abs/1910.10683.

Shah, J., Bikshandi, G., Zhang, Y., Thakkar, V., Ramani, P., and Dao, T. Flashattention-3: Fast and accurate attention with asynchrony and low-precision, 2024. URL https://arxiv.org/abs/2407.08608.

Taori, R., Gulrajani, I., Zhang, T., Dubois, Y., Li, X., Guestrin, C., Liang, P., and Hashimoto, T. B. Stanford alpaca: An instruction-following llama model. https://github.com/tatsu-lab/stanford_alpaca, 2023.

Team, G., Riviere, M., Pathak, S., Sessa, P. G., Hardin, C., Bhupatiraju, S., Hussenot, L., and et al., T. M. Gemma 2: Improving open language models at a practical size, 2024. URL https://arxiv.org/abs/2408.00118.

torchtune maintainers and contributors. torchtune: Pytorch's finetuning library, April 2024. URL https//github.com/pytorch/torchtune.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. u., and Polosukhin, I. Attention is all you need. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

Wang, G., Zeng, J., Xiao, X., Wu, S., Yang, J., Zheng, L., Chen, Z., Bian, J., Yu, D., and Wang, H. Flashmask: Efficient and rich mask extension of flashattention. *arXiv preprint arXiv:2410.01359*, 2024.

Wu, M., Cheng, X., Padon, O., and Jia, Z. A multi-level superoptimizer for tensor programs, 2024. URL https://arxiv.org/abs/2405.05751.

Ye, T., Dong, L., Xia, Y., Sun, Y., Zhu, Y., Huang, G., and Wei, F. Differential transformer. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=OvoCm1gGhN.

A Appendix

A.1 Neighborhood Attention (NA)

Neighborhood Attention (NA)(Hassani & Shi, 2022) is a local attention pattern used for 2D images, in which each pixel attends to its nearest neighboring pixels. NA mask is very complicated (Figure 14 top left) due to the 1D expansion of a 2D embedding neighborhood and thereby makes NA challenging to compute. Various advanced expansion strategies have been proposed(Hassani et al., 2023; 2024) to improve its block sparsity, but it is challenging manually to implement these optimizations in a high-performance attention kernel. Instead, we showcase that with *FlexAttention*, the Tiled NA and Morton curve NA could be implemented in less than 10 lines of PyTorch code to exploit the sparsity of NA (Figure 14) and enjoy its performance benefits (Figure 15). The idiomatic PyTorch implementation archives comparable performance to NATTEN, a dedicated CUDA implementation.

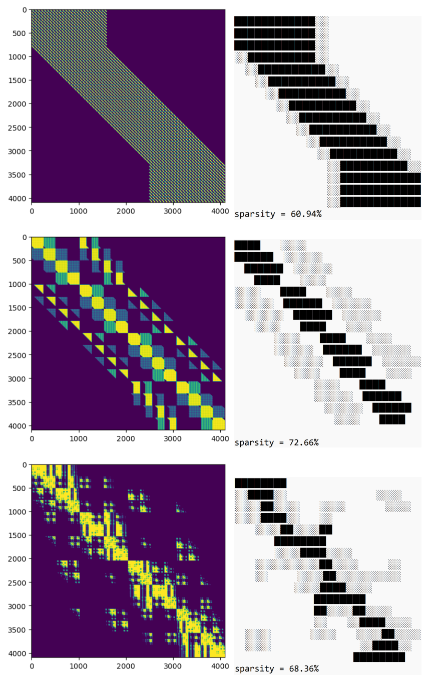
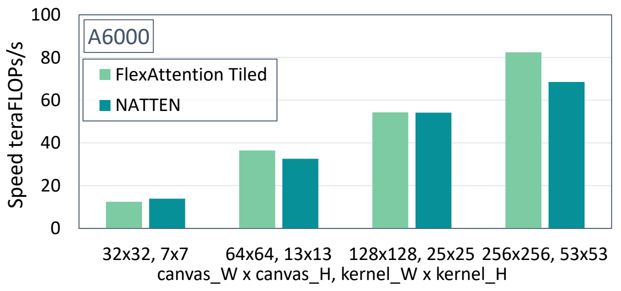


Figure 14. NA Itemized Mask and the Corresponding Block Mask. **Top:** Naive NATTEN Mask. **Middle:** 2D-Tiled NA Mask. **Bottom:** Morton Curve NA Mask.



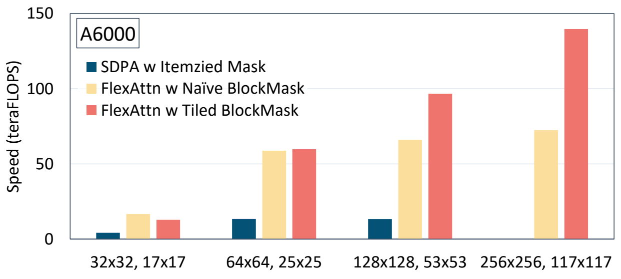


Figure 15. Neighborhood Attention Kernel Speed **Top:** FlexAttention Implementation vs. NATTEN(Hassani et al., 2024). **Bottom:** Different 2D Mapping.

B Artifact Evaluation

B.1 Abstract

The source code for this work is included in PyTorch, available at https://github.com/pytorch/pytorch. The FlexAttention kernel implementation can be found in `torch/_inductor/kernel/flex_attention.py` and `flex_decoding.py`. The kernel benchmark code to reproduce the results in this paper can be found at `benchmark/transformer/score_mod.py`. The FlexAttention implementation of attention variants used in the experiments is included in attention-gym. The source code for conducting end-to-end inference and post-training using FlexAttention is included in gpt-fast and torchtune.

These instructions focus on reproducing the GPU FlexAttention benchmark results (Figure 7 & 9). End-to-end speedup results for inference and training using FlexAttention are obtained from gpt-fast and torchtune benchmarks. Additional instructions are included in Section B.8.

B.2 Artifact check-list (meta-information)

- **Binary:** available at https://pytorch.org/
- **Hardware:** NVIDIA H100 / A100 GPU.
- **Metrics:** FlexAttention speedup compared to PyTorch SDPA and FlashAttention.
- **How much time is needed to prepare workflow (approximately)?:** 1 hour.
- **How much time is needed to complete experiments (approximately)?:** 1 hour.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** BSD-3

B.3 Description

B.3.1 How to access.

Source code and kernel benchmark: PyTorch
https://github.com/pytorch/pytorch

FlexAttetion mods: attention-gym
https://github.com/pytorch-labs/attention-gym

Inference using FlexAttention: gpt-fast
https://github.com/pytorch-labs/gpt-fast

Post-training using FlexAttention: torchtune
https://github.com/pytorch/torchtune

Baseline: FlashAttention
https://github.com/Dao-AILab/flash-attention

B.3.2 Hardware dependencies

- To match the configurations in this paper: NVIDIA H100 GPU.
- Compatible with NVIDIA GPUs (Compute Capability 8.0+) and AMD GPUs (ROCm 6.2+)

B.3.3 Software dependencies

- A recent Linux distribution.
- NVIDIA kernel drivers.
- CUDA version compatible with the chosen version of PyTorch. For PyTorch v2.6, use CUDA 12.4.
- gcc/g++ compatible with the chosen CUDA version.
- Miniconda installed
(https://www.anaconda.com/docs/main)
- pytorch v2.5+
- attention-gym
- Additional Python packages: tabulate tqdm matplotlib
- Baselines: FlashAttention (commit `c1d146c`) , cudnn v9.1.1

B.4 Installation

```
conda create --name=flexattn python=3.11
conda activate flexattn

# install PyTorch using release build (assume cuda 12.4)
pip3 install torch==2.6 torchvision==0.21 torchaudio==2.6 \
--index-url https://download.pytorch.org/whl/cu124

# install dependencies used in benchmark
conda install tabulate tqdm matplotlib

# Install attention-gym
git clone https://github.com/pytorch-labs/attention-gym.git
cd attention-gym
pip install .
cd ..

# get benchmark script
wget https://raw.githubusercontent.com/pytorch/pytorch/
     f1cbf4b1b5a299f999c11e77bfabe39c7f04efdc/benchmarks/
     transformer/score_mod.py
```

B.5 Experiment workflow

To reproduce FlexAttention speedup over PyTorch SDPA and FlashAttention on bfloat16, GPU, training (q_len = kv_len) (Figure 7), run:

```
# sweep sequence length
python -W ignore::UserWarning: score_mod.py -dtype bfloat16 \
-s 1024 4096 16384 65536  --kv-size 256 \
--backend cudnn efficient math fav2 fav3 \
-mods causal -d 64 --calculate-bwd --throughput

# sweep attention variants
python -W ignore::UserWarning: score_mod.py -dtype bfloat16 \
-s 16384 --kv-size 256 \
--backend cudnn efficient math fav2 fav3 \
-mods noop causal alibi sliding_window document_mask \
prefix_lm softcap \
-d 64 --calculate-bwd --throughput
```

To reproduce FlexAttention speedup over PyTorch SDPA and FlashAttention on bfloat16, GPU, inference (q_len = 1) (Figure 9), run:

```
# sweep kv length
python -W ignore::UserWarning: score_mod.py -dtype bfloat16 \
-s 1024 4096 16384 65536 131072  --kv-size 256 \
--backend cudnn efficient math fakv -mods noop \
-d 64 --throughput --decoding

# sweep decoding mods
python -W ignore::UserWarning: score_mod.py -dtype bfloat16 \
-s 65536 --kv-size 256 --backend cudnn efficient math fakv \
-mods noop causal alibi softcap \
-d 64 --decoding --throughput
```

This runs batched attention with FlexAttention, SDPA (with math, mem-efficient, cudnn) and FlashAttention (v2 & v3) backends and computes FlexAttention speedups. If one wants to save the results in a csv file, `--save-path` should be set.

B.6 Evaluation and expected result

The `score_mod.py` benchmark prints runtime (ms), kernel speed (TFLOPS/s), memory throughput (TB/s), and FlexAttention speedup numbers for each setup. It may also print some messages starting with `[SKIP]`, which means that some backends are skipped due to errors such as GPU out-of-memory or the lack of support for certain attention variants. Metrics for the skipped backends are reported as `nan`. The kernel speed should be similar to Figure 7 and Figure 9 reported in the paper.

B.7 Experiment customization

The above command can be customized in many ways, including sequence length, batch size, head dimension, mods, baseline backends, maxautotune etc. Run `python score_mod.py --help` for details.

B.8 Notes

FlexAttention inference speed (Figure 12) is reported by gpt-fast benchmarks. See https://github.com/pytorch-labs/gpt-fast#benchmarks for detailed instructions.

FlexAttention training speed (Figure 11) is reported by torchtune maintainers in https://gist.github.com/RdoubleA/012409f7919973d6ba7e9ca3efd5c237. Details results of this experiment can be found at https://wandb.ai/rafi-personal/torchtune?nw=4slh3i00evh.

Benchmark for FlexAttention implementation of paged kv cache (Figure 13) can be found at https://github.com/pytorch-labs/attention-gym/blob/main/attn_gym/paged_attention/latency.py.