SAS: Sparse Attention Synthesizer for Efficient Language Model Inference

Yuan Zhou, Shaojie Xiang, Lingfan Yu, Zhenyu Song, Charith Mendis, Yida Wang
{yazhom,shaojiex,lingfany,zhenyus,cmendis,wangyida}@amazon.com
Amazon Web Services
Santa Clara, California, USA

Abstract

Modern large language models rely on attention mechanisms that attend to all tokens in a sequence, resulting in quadratic computational complexity that limits scalability. While sparse attention reduces compute and memory requirements by attending to only important tokens, implementing these techniques presents significant challenges due to the complexity of combining static and dynamic sparse patterns and optimizing key-value (KV) cache management.

To address these challenges, we present SAS, a sparse attention synthesizer that automatically generates performant sparse attention kernels for large language model inference. SAS introduces a set of primitives that effectively encapsulate both static and dynamic sparse attention mechanisms, enabling users to compose complex attention patterns through logic operators and declarative functions. The system employs a geometric-based pattern analyzer to optimize for KV caching by determining minimal cache sizes and automatically generating cache management functions. Supporting both Nvidia GPU and AWS Trainium backends, SAS demonstrates significant performance improvements: 1.10-1.22× speedup for context encoding and 2.68-2.80× speedup for token generation over FlexAttention, a state-of-the-art flexible attention kernel synthesis tool, on GPUs, and 1.41-6.49× speedup for context encoding and 1.39-10.87× speedup for token generation over optimized dense attention on Trainium.

CCS Concepts: • **Computing methodologies** → **Machine learning**; • **Software and its engineering** → **Compilers**; **Domain specific languages**.

Keywords: Sparse Attention, Kernel Synthesis, Domain Specific Language, LLM Inference

ACM Reference Format:
Yuan Zhou, Shaojie Xiang, Lingfan Yu, Zhenyu Song, Charith Mendis, Yida Wang. 2026. SAS: Sparse Attention Synthesizer for Efficient Language Model Inference. In *European Conference on Computer Systems (EUROSYS '26), April 27–30, 2026, Edinburgh, Scotland Uk.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3767295.3769364

EUROSYS '26, April 27–30, 2026, Edinburgh, Scotland Uk

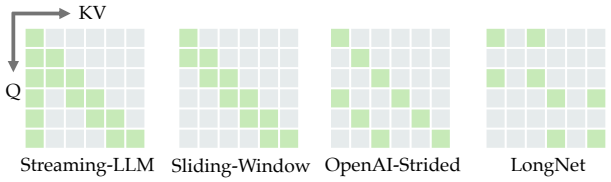
ACM ISBN 979-8-4007-2212-7/26/04.
https://doi.org/10.1145/3767295.3769364

1 Introduction

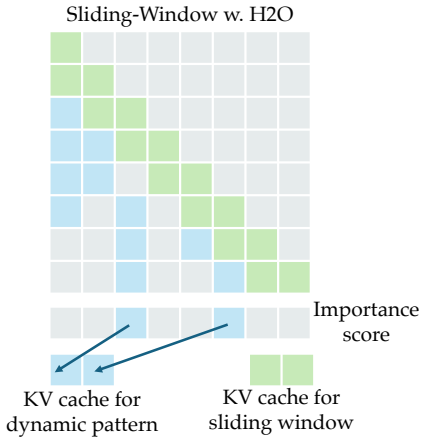
There has been an increasing adoption of transformer-based large language models (LLMs) in recent years. These models are powerful thanks to the self-attention mechanism [34], which allows the model to selectively focus on important parts of previous input tokens when producing every output token. However, vanilla self-attention is expensive because its time and space complexity grows quadratically with the input sequence length. Under the autoregressive generation scenario, key-value (KV) cache is introduced to avoid repetitively computing tokens that are already processed, reducing the time complexity of computing each output token to be linear with the number of processed tokens. However, in terms of space complexity, the memory required by the KV cache grows linearly with the number of processed tokens. For larger models, even this linear scaling is prohibitive for realistic model inference with long context.

Sparse attention is a promising approach to reducing both time and space complexity of LLM inference while maintaining model quality. The intuition behind sparse attention is that for each generated output token, the model only needs to attend to a small number of important prior tokens to get the correct result. From the compute perspective, the self-attention computation can be greatly reduced because the attention with "non-important" tokens does not need to be performed. From the memory perspective, the KV cache only needs to hold a much smaller number of tokens that are useful in the future. In addition, sparse attention masks can be used for addressing novel training scenarios where every token doesn't need to attend every other token [20]. Recent sparse attention techniques differ in the criteria of selecting important or relevant prior tokens. Some techniques employ static sparse attention patterns and choose tokens using relative position information [6, 9, 12, 21, 35, 38], while others leverage token embeddings to dynamically manage the KV cache [8, 25, 29, 32, 39]. State-of-the-art sparse attention techniques like Native Sparse Attention [37] use a combination of static and dynamic sparse attention patterns to achieve better model quality.

While sparse attention effectively reduces compute and memory costs, significant implementation gaps remain for



(a) Common static sparse attention patterns.



(b) Sliding window attention with H2O [39].

Figure 1. Examples of sparse attention patterns — (a) Static sparse attention patterns, each query token only attends to the key/value tokens indicated by the green blocks. (b) Sliding window attention with H2O that combines static and dynamic sparse attention. For each query token, green blocks indicate key/value tokens within the sliding window, while blue blocks show the key/value tokens in H2O's dynamic KV cache. An importance score for each token is maintained at run time to decide which tokens are kept in the KV cache.

deploying sparse attention in practical LLM inference scenarios. Below, we identify two major gaps and their associated technical challenges:

Gap 1: Programming abstraction – Figure 1 illustrates the attention patterns of several sparse attention techniques, each exhibiting distinct attention patterns which might be static (Figure 1a) or dynamic (Figure 1b). We use H2O [39] as a representative example of eviction-based dynamic sparse attention techniques. Under a fixed KV cache budget constraint, H2O maintains an importance score for each cached token. When processing a new token, the algorithm evaluates whether this token should replace the lowest-scoring cached token: replacement occurs if the new token's importance score exceeds that of the least important cached token, otherwise the new token is discarded. Figure 1b illustrates H2O's integration with sliding window attention, a widely adopted strategy for preserving local contextual information. In this hybrid approach, the KV cache contains two components: a static sliding window component that maintains recent tokens, and a dynamic component managed by importance-based eviction. New tokens are first inserted into the static component of the KV cache. As tokens age out of the sliding window due to its fixed size constraint, they become candidates for insertion into the dynamic cache component, where they compete based on their computed importance scores.

In the development cycle, users need to repeatedly experiment with different techniques to optimize the trade-off between model speed and accuracy, or even create a custom sparse attention pattern for their specific use cases. Most existing sparse attention implementations are hard-coded for a few techniques. Some exceptions, like FlexAttention [14] and SPLAT [18], provides limited support to dynamic sparse attention techniques. Given that dynamic sparse attention demonstrates superior model quality compared to static approaches, the limited support for dynamic patterns significantly restricts developers' ability to meaningfully explore the full spectrum of sparse attention techniques.

The technical challenge behind this gap lies in creating **a concise yet generic programming abstraction that accommodates both static and dynamic sparse attention patterns**. Such an abstraction should enable users to create custom attention patterns with minimal coding overhead and intuitive reasoning, while providing a flexible mechanism for composing multiple static and dynamic patterns. Existing programming abstractions for sparse attention have notable limitations: FlexAttention requires users to write PyTorch code and reason about token indexing for attention masking, which is error-prone for complex patterns; SPLAT employs an affine-expression-based representation that imposes restrictive constraints on sparse patterns. Furthermore, neither of them supports dynamic sparse patterns.

Gap 2: KV caching optimizations — Many sparse attention techniques offer the potential to substantially reduce KV cache size and conserve device memory through their inherent sparse patterns. Unfortunately, existing sparse attention libraries like xFormers [24], FlexAttention, and SPLAT lack the capability of performing this optimization, resulting in kernels that fail to achieve memory savings over dense self-attention during token generation. As a result, optimizing KV cache management to achieve the ideal memory savings for sparse attention remains a labor-intensive and error-prone manual effort. For dynamic sparse attention techniques, users needs to implement the criteria for selecting important tokens and optimize performance for these additional computations, further extending development time.

The core technical challenge involves **automated and efficient KV cache management**. To achieve this goal, an attention synthesis tool need to automatically derive the following from the sparse attention pattern: (1) the minimum KV cache size, (2) an attention masking function to mask out invalid tokens in the KV cache, and (3) a KV cache

indexing function to correctly insert new tokens into the compressed KV cache. With sparse attention patterns becoming more complex, automating these steps becomes increasingly difficult. Many static sparse attention techniques utilize attention masks that cannot be represented as affine expressions amenable to static analysis. The incorporation of dynamic patterns further complicates the problem, as dynamic components must be seamlessly integrated into KV cache management utilities.

In response to the aforementioned challenges, we propose Sparse Attention Synthesizer (SAS), a comprehensive framework that automates the deployment of sparse attention for LLM inference. SAS addresses both technical challenges through innovative approaches to attention pattern description and analysis. For the programming abstraction challenge, SAS provides a set of composable primitives that enables developers to describe both static and dynamic sparse attention patterns with ease. To resolve the KV cache optimization challenge, SAS converts the static part of sparse attention patterns into geometric-based intermediate representations and uses an event-based simulator to derive functions for KV cache management and attention masking. For dynamic sparse attention, users can construct a chain of declarative primitives to describe the computation. SAS analyzes this description to generate optimized kernels for efficient KV cache management during inference. SAS's user interface and analysis passes are completely hardware-agnostic. To demonstrate this flexibility, we have implemented hardware backends targeting Nvidia GPUs and AWS Trainium accelerators. For each backend, SAS leverages highly optimized templates to generate performant sparse attention kernels. Our contributions are summarized as follows:

- To the best of our knowledge, SAS is the first automated tool that can generate optimized sparse attention kernels for LLM inference while supporting both static and dynamic sparse patterns.
- We propose a generic programming abstraction that allows users to compose sparse attention patterns from primitives using intuitive boolean logic operators and declarative functions.
- We present an analyzer that extracts regularity from attention patterns and automatically generates functions that handle KV cache management and compute attention masks at run time.
- We implement SAS on two distinct machine learning accelerators: NVIDIA GPUs and AWS Trainium, making it the first attention kernel synthesizer to support multiple hardware backends. SAS significantly reduces development complexity by generating performant kernels from concise pattern descriptions, eliminating the need for developers to write extensive low-level kernel code that typically spans hundreds to thousands of lines.
- Evaluation results show that on Nvidia GPU, compared with a state-of-the-art attention kernel synthesis tool, FlexAttention [14], SAS achieves 1.10-1.22× speedup for context encoding and 2.68-2.80× speedup for token generation. SAS is also able to reduce the KV cache size according to sparse patterns, while FlexAttention only skips redundant computation without compacting the KV cache. On AWS Trainium, we achieve an average speedup of 1.41-6.49× for context encoding and 1.39-10.87× for token generation, over a highly optimized dense attention baseline.

2 LLM Inference with Sparse Attention

In this section we provide some background information on how LLM inference works with sparse attention. We focus on modern decoder-only LLM architectures like GPT [1] and LLAMA [15].

2.1 Standard LLM Inference

The basis of current LLMs is multi-head self-attention [34]. More recently, grouped-query attention (GQA) [3]was proposed to reduce the compute and memory overhead of LLMs, but the computation for each query head remains the same. Without sparse attention, the computation of each query head can be written as follows:

$$\text{Attn}(Q, K, V) = \text{softmax}\left(causal\_mask(\frac{QK^T}{\sqrt{d}})\right)V \quad (1)$$

where Q, K, and V are each head's query, key and value embeddings projected from the input of the transformer layer, respectively. d is the size of each attention head. For decoder-only models, a causal mask is applied onto the attention scores before softmax to guarantee each token can only attend to itself and previous tokens.

An important optimization for the inference stage of decoder-only models is KV caching, where the key and value embeddings of processed tokens are stored and reused for future iterations. With KV caching, autoregressive model inference can be naturally divided into two stages. The first stage is **context encoding**, where the model takes a multi-token input prompt, generates one output token, and populates the KV cache with the keys and values of the input prompt tokens. The second stage is **token generation**, where the model takes the output token from the previous iteration, computes attention using this token and the content of the KV cache, updates the KV cache using the key and value embeddings of this token, and generates another new output token. The token generation stage repeats until the maximum sequence length is reached or an end-of-sequence token is generated. These two stages are also referred to as **prefill** and **decode** in other literature.

2.2 Inference with Sparse Attention

Sparse attention reduces the compute and memory cost of LLM inference by selectively attending to a small set of important tokens. Conceptually, using sparse attention is equivalent to applying a special sparse mask to the attention score:

$$\text{Attn}(Q,K,V) = \text{softmax}\left(sparse\_mask(\frac{QK^T}{\sqrt{d}})\right)V \quad (2)$$

Existing sparse attention techniques differ on their methods of generating this sparse mask. For techniques with static attention patterns, the sparse masks are independent to the token embeddings and can be generated using a fixed rule during model execution. For dynamic sparse attention techniques, such as H2O [39] and its variants, the mask is often derived from an importance score which is a function of the input Q, K, and V embeddings.

Effectively leveraging sparsity patterns to achieve the desired performance improvements in LLM inference presents several non-trivial implementation challenges. For context encoding, an efficient attention kernel should skip most of the computation masked out by the sparse pattern, including computing the attention score and the weighted sum with the value tensor. For token generation, the kernel must use the minimum KV cache size to save memory, and carefully manage the KV cache by generating correct KV cache index and attention mask for each input token. If the attention pattern is dynamic, the overhead of dynamically generating the attention mask must also be minimized. Manually implementing a kernel and optimizing for these goals requires substantial engineering effort. With SAS, developers just need to compose the desired sparse attention pattern using SAS's generic programming interface. SAS will automatically analyze the pattern and generate optimized sparse attention kernels.

3 System Overview

Figure 2 presents an overview of SAS's workflow. The only task for the user is to describe the desired sparse attention patterns using SAS's programming interface, which enables flexible composition of attention patterns by combining primitive components. As a fully automated tool, SAS handles attention pattern analysis and code generation without any user intervention beyond the initial pattern specification. For static attention patterns, SAS converts the description of the sparse attention pattern to a geometric-based polygon representation, analyzes the attention pattern to determine the optimal KV cache size, then executes an event-based simulation to derive functions for computing attention masks and KV cache indices. These functions are represented in Python Abstract Syntax Tree (AST) to facilitate code generation. For the dynamic components, SAS generates specialized functions for computing importance scores based on the user-defined criteria. SAS's programming interface and attention pattern analyzer are completely hardware-agnostic and can be reused for different hardware targets.

The last step is template-based code generation, which is the only hardware-dependent step of the workflow. We choose to use templates because a lot of components in the attention kernel can be shared across various sparse attention techniques. The intermediate representations generated by the pattern analysis step are lowered into formats compatible with the target hardware and inserted into the templates. In this work we target Nvidia GPU and AWS Trainium [30] as hardware backends. Our GPU backend leverages Triton [33] and CUDA [10] implementations of FlashAttention-2 [11], while for Trainium we implement optimized kernels using Neuron Kernel Interface (NKI) [5].

4 SAS Programming Abstraction

SAS introduces an innovative programming abstraction that enables developers to describe complex sparse attention patterns by composing basic components in a productive manner. Unlike previous approaches that require mathematical formulations of attention masking and KV cache indexing using token indices, SAS's programming abstraction is constructed upon intuitive geometric primitives. In addition, developers can easily compose the basic components to create custom static and dynamic sparse patterns. To the best of our knowledge, this is the first time such a composable and versatile abstraction is proposed for sparse attention development.

4.1 Basic Sparse Attention Components

The most intuitive way of defining a sparse attention pattern is to use the attention mask during context encoding. Through a comprehensive investigation of existing sparse attention works, we identified three common basic components used to construct attention masks for sparse attention: (1) rectangle component, (2) diagonal component, and (3) dynamic component, as shown in Figure 3. Compared with the basic patterns used in MInference [21] and SPLAT [18], the basic components in SAS are more primitive and finer-grained. SAS allows users to freely compose these components, thus being able to represent a wider range of sparse patterns than prior works. Below we introduce these three components in more details.

Both the rectangle and diagonal components cover areas where the attention mask is True. A **Rectangle** component defines a rectangular region that can be positioned anywhere within the attention matrix. As shown in Figure 3a, it can be customized with a stride to control the spacing between valid blocks and a stride offset to adjust the initial block position within the defined area. A **Diagonal** component covers an area in the diagonal direction of the attention matrix. Figure 3b shows several examples of using the Diagonal component to describe sliding window or causal masking.

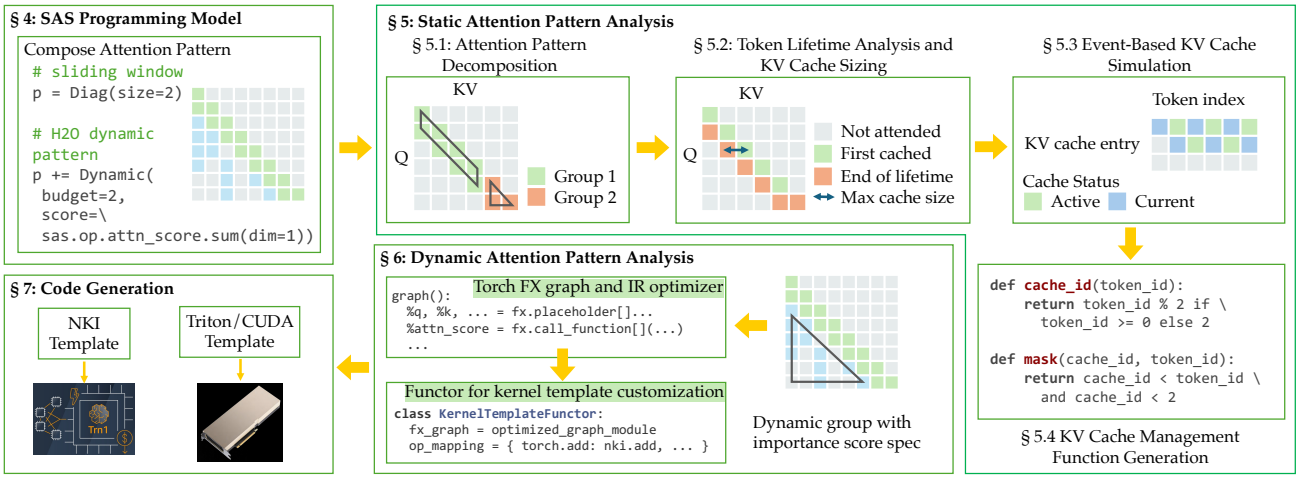
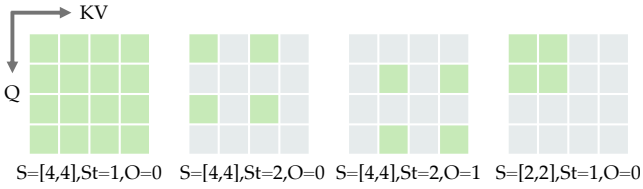


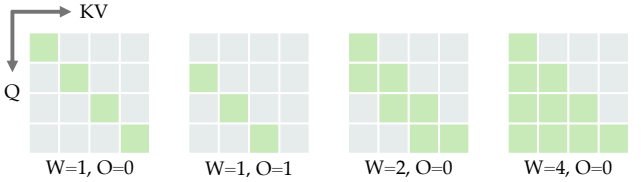
Figure 2. SAS Overall Flow.

A **Dynamic** component covers an area where attention mask is dynamically determined by query and key embeddings at run time. For this component, SAS currently supports token-wise importance score as described in H2O[39] for attention masking and KV cache eviction, where in every iteration of token generation the least important token is evicted from the cache. When creating a dynamic component, users need to specify a maximum cache budget and describe the scheme for computing importance score. Figure 3c shows a few examples of using the dynamic component. Users can tweak the refresh mode (R in the figure) to determine how KV cache is updated when a new token is generated. When $R = True$, the KV cache preserves all past KV tokens but only loads a subset from it for each iteration, where the number of tokens loaded is determined by the cache budget B. If $R = False$, the KV cache size is strictly equal to budget B. In this case, tokens evicted from the cache are permanently erased and will not be attended to in later iterations.

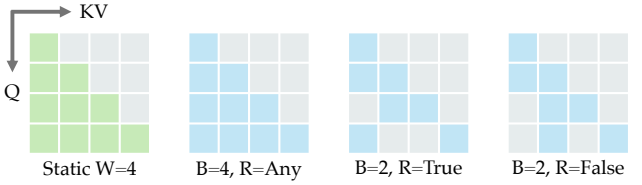
For dynamic sparse attention, SAS provides a set of predefined operators to compute importance scores for each KV token pair, such as `.attn_score()` to compute attention score from raw QK token sequence, `.reduce()` for tensor reduction at specified dimensions, or `.pooling()` to aggregate local information from previous scores. Each operator takes the output from previous one if any, update the scores and return a new score. SAS also allows users to define custom operators to compute token-wise importance scores. Figure 4 shows an example of defining an operator to compute importance scores of tokens using their query token embedding and a pre-trained weight tensor provided by users. The custom op can be used along other predefined ops in SAS to calculate importance of tokens in inference. The custom operator interface enables users to define their own operators using PyTorch's native APIs, providing a generalized approach to support novel dynamic sparse patterns beyond



(a) Rectangle pattern — A rectangle pattern is defined by size S, stride St, and stride offset O.



(b) Diagonal pattern — A diagonal pattern is defined by window size W and offset O.



(c) Dynamic pattern (blue) attached to static pattern (green) — The dynamic pattern takes effect on top of the KV cache of the attached static pattern. Developers use cache budget B and refresh mode R to control the behavior of the dynamic pattern. When R is set to True, all prior tokens are kept in the cache but only tokens with the top-B importance score will be attended to. When R is False, tokens evicted from the cache are permanently erased.

Figure 3. Basic attention patterns in SAS.

those covered in this paper. When integrated with the aforementioned static components, it creates a comprehensive

```python
@sas.ops.register
def mm_score():
    # Symbolic inputs as torch tensors
    q = sas.sym.active_Q
    w = sas.sym.custom_weight(name="wgt")
    return torch.matmul(q, w)

# Use custom mm_score() to compute token importance
p = Dynamic(budget=8, score=\
        sas.ops.mm_score().sum(dim=1))
```

Figure 4. Building custom operator for dynamic sparse attention in SAS

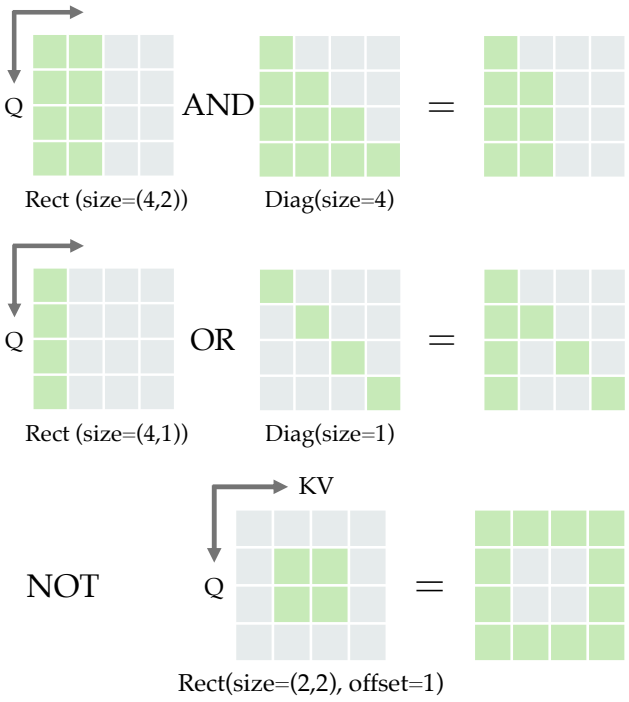


Figure 5. Examples of composing SAS's basic components with boolean logic operators.

design space that allows researchers to systematically explore and implement various sparse attention techniques within their models.

4.2 Composing Sparse Attention

Programmers can perform common boolean logic operations on the basic patterns, including AND (*), OR (+), and NOT (!), to create new sparse patterns. As shown in Figure 5, these boolean operations work in the same way as if we are directly using them on the boolean attention masks. We also introduce a spread operator (») for representing variants of block-sparse attention. The spread operator creates a new pattern from a rectangle pattern and another pattern by treating the rectangle as a unit and expanding the other pattern accordingly. Figure 6 shows an example of creating

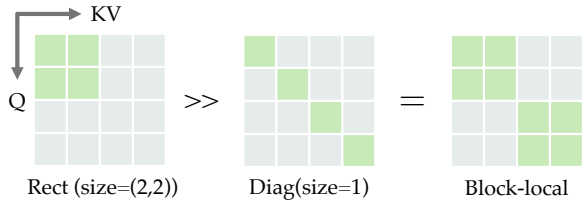


Figure 6. Block-sparse attention using spread operator.

a block-diagonal pattern, where we first define a 2×2 block pattern and then spread it across a diagonal pattern.

Figure 7 shows how we define the sliding window + H2O pattern shown in Figure 1b and the strided block-local pattern from LongNet [12]. We can define both patterns with a few lines of code by composing basic attention patterns. In the supplementary material, we further illustrate what the user needs to write to implement the strided block-local pattern without SAS. Compared with FlexAttention, SAS provides an alternative and more intuitive way of describing the attention patterns. Without any automation, user would need to carefully reason about attention masking and KV cache indexing, then write hundreds of lines of code to implement the kernel in low-level languages like NKI [5].

4.3 Top-Level Interface

Once the desired sparse attention pattern is defined, it can be used to instantiate an attention kernel as shown in Figure 8. If the attention kernel takes other tensors as input, user can pass the tensors into the attention kernel by specifying a mapping from tensor names to their values. When being invoked for the first time, the kernel class analyzes the input sparse patterns to generate optimized compute schedule and kernels for hardware execution. More details for these steps will be covered in Section 5 and 6. Users can directly integrate this class into existing workload without explicitly managing the KV cache or implementing sparse attention specific kernel optimizations.

5 Static Attention Pattern Analysis

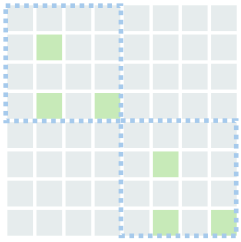
SAS's programming abstraction describes sparse attention patterns efficiently, but it does not directly address the problem of efficient KV cache management during inference. In this section, we discuss how SAS tackles this challenge for static sparse attention. Effective KV cache management for static sparse attention depends on two functions: (1) cache index function, which determines where new token embeddings are stored in KV cache, and (2) attention mask function, which specifies the KV cache entries to use when computing attention scores. Below we explain how SAS automatically analyzes the attention pattern and generates these functions for model inference.

```python
# Sliding window pattern
p0 = Diag(offset=0, size=2)
# Area covered by dynamic component
p1 = Diag(offset=2, size=6)
# Add dynamic component
p2 = p1 * Dynamic(budget=2, score=\
     ops.attn_score().sum(dim=1), \
     refresh=False)
# Compose
window_w_h2o_pattern = p0 + p2
```


(a) Creating a sliding window + H2O pattern in SAS.

```python
# First create a diagonal pattern
p1 = Diag(offset=0, size=1)
# Then craete a rectangular block
p2 = Rect(size=[4, 4])

# Create the block local pattern
block_local = p2 >> p1
# Add stride
strided_block_local = block_local * \
  Rect(size=[8,8], stride=2, offset=1)
```



(b) Creating a strided block-local pattern in SAS.

Figure 7. Examples of composing custom sparse attention patterns using SAS's programming interface — SAS allows users to create realistic sparse attention patterns, potentially with dynamism, using only a few lines of code by composing primitive geometric components. Causal masking is omitted from the code for simplicity. **(a)** H2O is applied to the lower triangular area not covered by the sliding window pattern. **(b)** The `spread` operator (» in line 6) allows easy creation of block-sparse patterns. A global stride is applied on top of the block-local pattern using the logical AND operator (⋆ in line 8).

```python
# Class signature
class SASAttnKernel(self,
    # Attention pattern
    pattern,
    # Additional input tensors
    other_inputs = {}
)
...
# Defining the sliding window + H2O kernel
window_w_h2o_kernel = SASAttnKernel(
  window_w_h2o_pattern,
  other_inputs={})
# Calling it
out = window_w_h2o_kernel(q, k, v, past_k, past_v)
```

Figure 8. Building a sparse attention kernel in SAS.

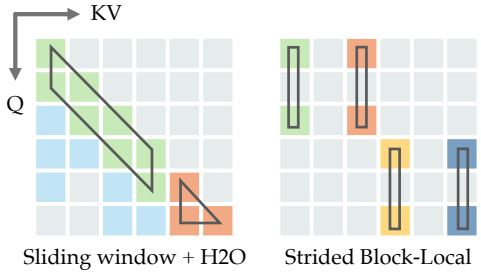


Figure 9. Decomposing static sparse attention into groups: static attention areas in same color belong to the same group, as outlined in the bounding boxes.

5.1 Attention Pattern Decomposition

The composed sparse attention pattern can be complex and hard to analyze. Instead of analyzing the whole pattern in one shot, SAS breaks down the sparse attention pattern into smaller, more manageable groups, making it easier to reveal regularities. Internally, SAS represents attention patterns as polygon objects. Our pattern decomposition starts by analyzing the borders of the polygon and identifying all the corner points. We decompose attention patterns by making vertical cuts at the corner points and categorizing the part between every two cuts into the same group. Figure 9 shows how we decompose the attention patterns for the static part of the sliding window + H2O pattern and the strided block-local pattern. The diagonal pattern in sliding window H2O (static) gets split into a parallelogram and a tailing triangle, while the strided block-local pattern is decomposed to four vertical blocks with stride 2.

5.2 Token Lifetime Analysis and KV Cache Sizing

Sparse attention patterns can be viewed as geometric objects, and their geometric features provide key insights into the KV cache status for each token in the sequence. In Figure 10, we present the attention matrices for the static part of sliding window + H2O and strided block-local attention, highlighting the implications for KV cache status. In these attention matrices, the color of blocks in a column indicates the lifespan of the corresponding KV embeddings. For sliding window + H2O, each column shows that a specific KV token is only attended to by current and next query token before reaching the end of its lifetime. For strided block-local attention, the first token remains in the KV cache until the attention score for the third token is calculated. Since it is not attended to by the second token, the first token is marked as inactive when the query token's index is 2.

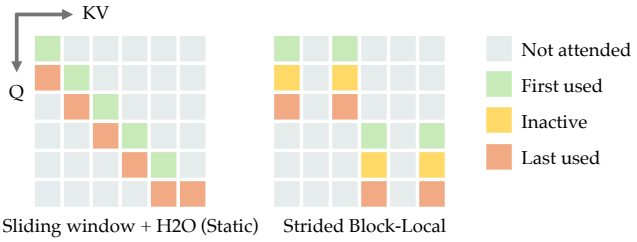


Figure 10. Token lifetime from attention geometry

The geometric features of sparse attention also reveal regularities that indicate repeating KV cache behaviors. For instance, in the sliding window + H2O (static) pattern, when

a new KV token is cached, the KV token from two positions back (if any) is at the end of its lifespan and can be evicted. These memory reuse patterns can be observed from borders of sparse attention and can help simplify KV cache management. We detail how we leverage this information to generate KV cache management functions in Section 5.3.

For a sparse attention pattern, its optimal KV cache size is determined by the maximum number of live KV tokens when processing each query token in the sequence. Deriving the optimal KV cache size requires making a horizontal cut for each Q token in the sequence and find the max length of intersection with KV token's lifespan. With decomposed attention groups, we can narrow down the search space to a smaller range by only examining the projections of all corner points of the decomposed groups onto the Q axis. Since the total number of corner points is significantly smaller than the sequence length, leveraging the decomposed groups greatly improves the scalability of our analysis.

5.3 Event-based KV Cache Simulation

Dividing each attention pattern into groups of regularly shaped polygons simplifies the analysis. However, to track KV cache status during inference, we need to consider the interactions between multiple groups. For example, the KV cache allocation for subsequent groups depends on the allocation and status of the current and previous groups. To capture this, we develop an event-based KV cache simulator to track key events that represent state changes in KV cache:

- `CacheStore`: Store embeddings for KV tokens from certain range into the cache.
- `CacheInactive`: Mark certain KV cache entries as inactive for attention calculation of some Q tokens.
- `CacheActive`: Mark certain KV cache entries as active for attention calculation of some Q tokens.
- `CacheEvict`: Evict certain tokens from the KV cache when progressing through Q tokens beyond a range.

These cache events are defined by the borders of each attention group; the upper and lower borders correspond to `CacheStore` and `CacheEvitc`, while inner borders represent `CacheActive` and `CacheInactive`. These events are then added to a priority queue based on their associated token ranges.

KV Cache Simulation with Event Batching — The simulator processes these events in batches. Events with overlapping token ranges are grouped together, and each batch is processed sequentially, updating the KV cache traces. This approach, detailed in Algorithm 1, allows for efficient updates to KV cache traces by processing related events together. It's particularly necessary for patterns like sliding windows, where overlapping store and evict events allow new embedding to take cache entry that was just evicted, reducing unnecessary cache operations.

It is worth noting that simulation time correlates with sparse pattern complexity rather than sequence length. When sparse patterns have less geometric regularity to exploit, they generate more KV cache events, increasing simulation complexity and analysis time. For example, a sliding window sparse pattern has high geometric regularity and translates to just three cache events in SAS: one `CacheStore` event for storing new tokens into KV cache as inference runs, one `CacheEvict` event for gradually removing oldest entries, and another `CacheEvict` event for clearing cache at inference end. As sequence length increases, the number of events remains constant; only the range within each event changes. Since simulation time depends solely on the number of cache events, the analysis time remains the same no matter how long the sequence length is.

Algorithm 1 Event-based KV Cache Simulator

Require: $E = \{e_1, \ldots, e_n\}$: events with $e_i.range = [x_i, y_i]$ and $e_i.type \in \{\text{store, evict, active, inactive}\}$
Ensure: Processed events and generated KV cache traces

```
procedure KVCacheSimulator(E)
    PQ ← PriorityQueue(E, key = (e) => e.range[0])
    cache ← new KVCache()
    while PQ is not empty do
        event ← PQ.pop()
        if not event.range overlaps lastRange then
            cache.ProcessEvents()
            cache.UpdateState()
        end if
        cache.RegisterEvent(e.type, e.range[0])
    end while
    return cache.GenerateTrace()
end procedure
```

Fast Tracing via Index Mapping: Algorithm 1 provides a general method for collecting KV cache traces from the geometric representation of sparse attention. However, it may encounter scalability issues with sparse attention patterns that include many non-adjacent geometric objects like the strided block-local pattern. In these cases, processing non-adjacent objects in separate batches sequentially leads to inefficiencies. To alleviate this limitation, we apply Algorithm 1 to the original mask, prior to spreading it to blocks or intersecting with striding patterns. The KV cache trace from this original mask can be represented as a simpler linear piecewise function of the token's index. We then use index mapping to derive the actual KV cache trace from the linear piecewise function that has not yet had block spreading or stride intersections applied to it. For instance, the trace of a block-sparse pattern with a block size of B can be obtained from the non-blocked version using the following mapping:

$$\text{trace}_{b=B}(id) = \text{trace}_{b=1}\left(\left\lfloor \frac{id}{B} \right\rfloor\right) \times B + id \bmod B \qquad (3)$$

In Figure 11, we present the derived KV cache traces from simulator for sliding window + H2O (static) and strided block-local attention. These traces show the storage locations of each token embedding in the KV cache and indicate which KV cache embeddings are used for attention calculations for each token in the sequence.

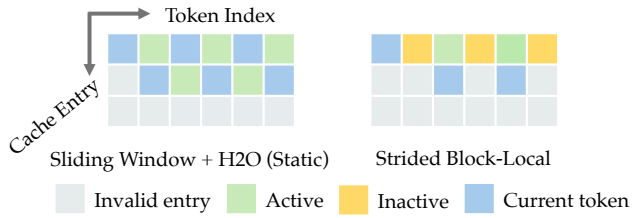


Figure 11. KV cache traces from simulation.

Active cache entries indicate that the cache holds valid KV embeddings, which are used to compute attention scores for the corresponding Q token. Inactive entries mean the KV embeddings are not used for the current Q token's attention computation but will be used later, so they are not evicted. The "current token" entry stores the computed embedding for the current token index. In the next section, we discuss how to convert these traces into AST expressions to enable efficient KV cache management at run time.

5.4 KV Cache Management Function Generation

SAS analyzes KV cache traces and generates simplified functions represented in Python AST for KV cache management during inference. Two key functions are derived: (1) KV cache index function that tells which KV cache entry a specific KV token should be stored to, and (2) attention mask function that tells what KV cache entries should be included when calculating attention score for a specific Q token.

For static sparse attention, SAS generates expressions to capture the states of KV cache trace shown in Figure 11. Examples of generated expressions for sliding window + H2O (static) and strided block-local attention are shown in Figure 12. These functions distill critical information from static sparse attention, and can be directly integrated into an attention kernel for efficient KV cache management.

```
# Sliding window H2O. KV_CACHE_SIZE = 2
def cache_id_func(token_id):
    return token_id % 2 if token_id >= 0 else -1
def mask_func(cache_id, token_id):
    return cache_id <= token_id and cache_id < 2

# Strided block-local attention. KV_CACHE_SIZE = 2
def cache_id_func(token_id):
    return token_id // 2 if token_id % 2 == 0 else -1
def mask_func(cache_id, token_id):
    return cache_id <= (token_id // 2) % 2  \
        if token_id % 2 == 0 else False
```

Figure 12. Functions for KV cache management.

6 Dynamic Attention Pattern Analysis

SAS adopts a compiler-driven approach to analyze the dynamic components in user-specified sparse attention patterns. This approach is necessary because these patterns exhibit runtime dynamism and are and not geometrically static, posing more challenges to predict its behavior.


```
graph():
  %active_q, %active_k, ..., \
    %custom_wgt = fx.placeholder[] ...
  %attn_score = fx.call_function[
    target=attn_score](..., size=...)
  %sum = fx.call_function[
    target=torch.sum](%attn_score, dim)
  ...
  return
```

Figure 13. SAS tracing user-specified operator chain of dynamic sparse attention into torch FX graph.

Importance Score Specification Analysis — SAS traces the chain of operations specified by users into a torch FX graph, a high-level intermediate representation in PyTorch to capture and transform computational graphs. Each operator invoked in the chain is encapsulated into an individual `call_function` node in FX graph. SAS then traverses the FX graph to locate patterns that are feasible for opportunistic operator fusion, e.g., fusing vector operations with activations into a single op to avoid allocating unnecessary on-chip memory.

Inter-kernel Memoization — Dynamic sparse attention involves both standard attention computations and additional calculations of importance scores needed for dynamic KV cache management. In SAS, these computations are organized into two separate kernels. One of common practice is to use attention scores as the base to derive KV importance scores [7, 23, 39]. In this case, the intermediate results, i.e., the row max and sum of exponentials from the FlashAttention kernel, can be memoized to avoid unnecessary recomputation. When the user-specified importance score spec starts with the attention score op, SAS leverages this characteristic and generates low-level kernel to enable inter-kernel memoization between flash attention kernel and importance score kernel.

Functor Generation for Template Customization — In the last step, SAS processes the operators in the graph and assigns the mapping from these operators to native operators for the target hardware backend. For example, when targeting AWS Trainium, SAS provides a mapping that converts operations in the optimized FX graph to library functions or low-level ISA instructions in NKI. The optimized FX graph and operator mapping are wrapped inside a functor, which is intended to be invoked from inside kernel template. Once being invoked, these operators are traced and lazily converted into native operator representation recognizable by the target hardware's compilation flow.

Dynamic KV Cache Management — SAS generates cache IDs and masking functions to manage KV cache indices and identify active tokens. Unlike static patterns where cache indices remain fixed across inference iterations, dynamic sparse attention requires real-time index calculation based on user-specified importance scores. Initially, when the dynamic KV cache budget has available capacity, cache IDs increment sequentially to accommodate new tokens while simultaneously maintaining and updating importance scores for each stored token at every iteration. Once the cache reaches its full budget, SAS transitions from this simple sequential allocation to a more sophisticated eviction-based strategy, where the token-wise importance scores are leveraged to determine which existing tokens should be removed from the cache, thereby creating space for incoming tokens while preserving the most contextually relevant information based on the user-defined scoring criteria.

7 Optimization and Code Generation

SAS's user interface and attention pattern analysis are completely hardware-agnostic. We prototype SAS on Nvidia GPU and AWS Trainium [30] to demonstrate the versatility of SAS on different hardware backends. Our code generation for both backends take a template-based approach, where we lower SAS's intermediate representations to formats compatible with the target hardware and insert the generated functions into the template. Below we introduce our code generation process for Nvidia GPU and AWS Trainium in more details.

7.1 Nvidia GPU

We modify the Triton FlashAttention-2 implementation [27] to construct our template for context encoding. To leverage FlashAttention's tiled computation, our template implements block-sparse attention and takes the indices of valid blocks as inputs. These indices are retrieved by analyzing the geometric representation: if a tile overlaps with the area covered by valid tokens, then the tile needs to be computed. We wrote a custom lowering pass to lower SAS's geometric representation with logic operators into Triton operators for attention masking. The indices of valid tiles are stored in the compressed-sparse-row (CSR) format to minimize memory overhead. Following the original Triton kernel implementation, we split the indices of full tiles and partial tiles into two inputs so that we only apply masking to the partial tiles.

For token generation, the KV cache size is minimized by SAS for static attention patterns and specified by the user for dynamic sparse attention. As a result, the amount of redundant computation is minimized, thus the benefit of skipping redundant compute may not cover the overhead of the additional tile selection logic. Therefore, we choose to leverage the official CUDA implementation of dense FlashAttention-2 for token generation. Custom attention masking function can be generated by converting the Python AST functions generated by SAS to CUDA C++ code and inserted into the CUDA kernel.

7.2 AWS Trainium

AWS Trainium is a machine learning accelerator designed by Amazon for high-performance machine learning model training and inference [16, 17], available in Amazon EC2 instances. In contrast to SIMT architecture of GPUs, AWS Trainium employs a more heterogeneous architecture, which consists of a tensor engine with 128×128 systolic array for efficient matrix multiplication, and a group of hardware engines dedicated for vector, scalar, and DMA operations [30]. To achieve high-performance on Trainium, one critical aspect is to effectively orchestrate the workloads across these engines to improve the overall hardware utilization.

We develop a block-sparse flash attention kernel template for sparse attention computation, and another kernel template for token-importance score computation when dynamic sparsity is used. These templates are developed using NKI [5], which provides developers with direct access to the instruction set of Trainium hardware. Our NKI FlashAttention kernel template takes a pre-computed schedule generated by the SAS analyzer to skip unattended blocks in attention computation. When dynamic sparse attention pattern is based on attention scores, the kernel memoizes the intermediate tensors from FlashAttention into global device memory and reuses them in the importance score computation. When each kernel template is invoked, its operators are traced by the Neuron compiler [4] and compiled to a binary that can be consumed by Trainium.

To maximize performance of these kernel templates on Trainium, we apply a set of optimizations designed to saturate off-chip memory bandwidth and improve overall model FLOPS utilization. This includes: (1) Memory layout customization, where the tensor dimensions are carefully reordered to create more contiguous memory access patterns, in order to improve DMA access efficiency, and (2) Compute engine workload balancing, where operations are strategically assigned to appropriate hardware engines. For example, tensor reduction operations can be handled by both vector engines and tensor engines. We need to carefully select which engine to execute this operators to prevent overloading any single engine, which would cause pipeline stalls. These optimizations ensure efficient execution across various attention operations by balancing memory throughput and computational efficiency.

8 Evaluation

We evaluate the effectiveness of SAS using a set of realistic attention patterns shown in Table 1. The baselines and kernels generated by SAS only contain the self-attention part of an LLM, i.e., the computation defined by Equation 2,

Table 1. Attention patterns used in evaluation and their configurations.

Pattern	Description
Streaming-LLM [35]	Sink size 32, local window size 1024.
Block Local	3 local blocks with block size 128.
Sliding Window [19]	Local window of size 1024.
OpenAI Strided [9]	Local window size 512, stride 512.
OpenAI Fixed [9]	Block size 256, stride 256.
Strided Block-Local [12]	Block size 256, stride 4.
H2O [39]	Using accumulative attention score to indicate KV token importance. We enhance each static sparse attention pattern above with H2O as suggested in [39]. The KV cache budget is 512.

together with any dynamic components defined by the attention pattern (e.g., compute importance scores). For each technique, we evaluate the attention kernel's performance from sequence length 2K to 16K using 64 attention heads with a head size of 128, for both context encoding and token generation.

8.1 GPU Results

We run GPU experiments on an Amazon EC2 `p4d.24xlarge` instance with eight Nvidia A100 GPUs, 96 CPU cores, and 1152 GB memory. In terms of software packages, we use CUDA 12.4, PyTorch 2.6.0, and FlashAttention 2.7.4.post1. We compare our results with FlexAttention [14] for both context encoding and token generation, and also compare with FlashAttention-2 [11] dense attention. Without loss of generality, the baselines and SAS generated kernels are evaluated on one GPU, but the kernels generated by SAS can be directly applied to distributed inference scenarios with tensor parallelism [31].

Table 2 and Table 3 show our results for context encoding and token generation, respectively[1]. For context encoding, on average we achieve 1.10-1.22× speedup over FlexAttention and 1.56-6.69× speedup over dense FlashAttention-2. Breakdown to individual cases, for all but the OpenAI fixed sparse attention pattern, SAS generated kernels are comparable or slightly faster than FlexAttention. For the OpenAI fixed sparse pattern, SAS is faster than FlexAttention by up to 1.87× because FlexAttention uses a larger sparse block size for their kernel implementation[2]. As a result, the FlexAttention kernel spends more time on redundant computation that is eventually masked out. We expect this to be a performance bug that can be fixed by FlexAttention developers with minimal effort. For token generation, on average we achieve 2.68-2.80× speedup over FlexAttention. The main reason is that SAS leverages the CUDA FlashAttention decoding kernel, which has much stronger performance than FlexAttention's Triton-based kernel under our evaluation settings. Compared with dense FlashAttention, SAS achieves 1.93-5.32× speedup thanks to the reduced KV cache size. Table 4 shows the reduced KV cache size derived by SAS. SAS is able to calculate the KV cache size according to the sparse pattern and significantly reduce KV cache memory when possible, while FlexAttention can skip computation but does not provide any memory savings.

[1]For token generation we report the performance of dense FlashAttention-2 decoding kernel with the reduced KV cache size computed by SAS. We expect the additional masking overhead to be minimal.

[2]FlexAttention will raise an error if we configure it to use the same sparse block size as SAS.

8.2 AWS Trainium Results

We run AWS Trainium experiments on an Amazon EC2 `trn1.32xlarge` instance with 16 AWS Trainium chips, 128 vCPU cores, and 512 GB memory. We use AWS Neuron SDK version 2.22 for kernel development and compilation. The experiments are conducted on one Trainium chip using an individual NeuronCore [30]. We compare the performance of SAS-generated NKI kernels against manually optimized FlashAttention NKI kernels released in AWS Neuron SDK. This comparison evaluates both static and dynamic sparse attention techniques, as detailed in Table 1, for context encoding and token generation.

Table 5 and Table 6 show the experiment results of context encoding and token generation for static and dynamic sparse attention patterns. SAS-generated NKI kernel achieves 1.41× to 6.49× speedup in static sparse attention patterns compared with optimized NKI flash-attention kernel. For dynamic sparse attention scenarios, the inter-kernel memoization and importance score computation introduce some overhead, but after accounting for these costs, SAS designs achieve speedups of 1.12× to 1.50×. For token generation, SAS designs achieve 1.97× to 10.87× speedup over optimized NKI flash attention design for static sparse patterns. When combining these patterns with H2O dynamic sparse attention to the area which was not covered by static mask, SAS-generated designs achieve speedups ranging from 1.39× to 7.68×.

8.3 Pattern Analysis Time

Table 7 shows how SAS's processing time scales with sequence length for two representative patterns. Most existing sparse patterns in attention have regular geometric shapes because they follow predictable structures (e.g., sliding windows, block diagonals). For these patterns, SAS analysis time scales primarily with pattern complexity rather than context length using the fast tracing and event batching optimizations introduced in Section 5.3. For example, the processing time of sliding window attention remains around 0.75 seconds even at 128K sequence length.

Sparse attention patterns that lack regular geometric structures and are incompatible with these optimizations require

Table 2. GPU context encoding latency — The *FA2* column in the table shows the latency of FlashAttention-2 CUDA kernel. Numbers in parentheses show speedup over this implementation. Time unit is millisecond.

	Seq = 2K			Seq = 8K			Seq = 16K		
	FA2	Flex Attn	SAS	FA2	Flex Attn	SAS	FA2	Flex Attn	SAS
Streaming-LLM	0.425	0.459 (0.93×)	0.416 (1.02×)	5.293	2.127 (2.49×)	1.84 (2.88×)	20.42	4.389 (5.36×)	3.812 (5.36×)
Block Local		0.228 (1.86×)	0.202 (2.10×)		0.852 (6.21×)	0.742 (7.13×)		1.694 (12.1×)	1.472 (13.9×)
Sliding Window		0.424 (1.00×)	0.412 (1.03×)		1.898 (2.79×)	1.830 (2.89×)		3.926 (5.20×)	3.763 (5.43×)
OpenAI Strided		0.362 (1.17×)	0.373 (1.14×)		2.844 (1.86×)	2.955 (1.79×)		9.438 (2.16×)	9.859 (2.07×)
OpenAI Fixed		0.263 (1.62×)	0.196 (2.17×)		2.31 (2.29×)	1.324 (4.00×)		8.02 (2.55×)	4.287 (4.76×)
Strided Block-Local		0.170 (2.50×)	0.160 (2.66×)		0.603 (8.78×)	0.467 (11.3×)		1.181 (17.3×)	0.905 (22.6×)
Avg. Speedup vs. FA2		1.42×	1.56×		3.43×	4.11×		5.50×	6.69×

Table 3. GPU token generation latency — The *FA2* column in the table shows the latency of FlashAttention-2 decoding CUDA kernel. Numbers in parentheses show speedup over this implementation. Time unit is millisecond.

	Seq = 2K			Seq = 8K			Seq = 16K		
	FA2	Flex Attn	SAS	FA2	Flex Attn	SAS	FA2	Flex Attn	SAS
Streaming-LLM	0.085	0.162 (0.52×)	0.054 (1.57×)	0.227	0.174 (1.30×)	0.053 (4.28×)	0.413	0.178 (2.32×)	0.054 (7.65×)
Block Local		0.076 (1.12×)	0.031 (2.74×)		0.076 (2.99×)	0.031 (7.32×)		0.076 (5.43×)	0.031 (13.3×)
Sliding Window		0.159 (0.53×)	0.053 (1.60×)		0.156 (1.46×)	0.053 (4.28×)		0.159 (2.60×)	0.053 (7.79×)
OpenAI Strided		0.143 (0.59×)	0.070 (1.21×)		0.337 (0.67×)	0.247 (0.92×)		0.613 (0.67×)	0.404 (1.02×)
OpenAI Fixed		0.162 (0.52×)	0.079 (1.08×)		0.548 (0.41×)	0.221 (1.03×)		1.099 (0.38×)	0.409 (1.01×)
Strided Block-Local		0.060 (1.42×)	0.015 (5.67×)		0.081 (2.80×)	0.015 (15.1×)		0.073 (5.66×)	0.015 (27.5×)
Avg. Speedup vs. FA2		0.719×	1.93×		1.28×	3.53×		1.90×	5.32×

Table 4. KV cache size reduction achieved by SAS at 16K context length.

Pattern	KV cache size reduction
Streaming-LLM	16K →1056
Block Local	16K→384
Sliding Window	16K→1K
OpenAI Strided	16K→15.5K
OpenAI Fixed	16K→15.8K
Strided Block-Local	16K→64

longer processing but still complete within minutes. As shown in Table 7, the OpenAI fixed pattern takes approximately 12 minutes at 128K context due to inconsistent block sizes that prevent index re-mapping optimization. This irregularity causes the geometric pattern to fragment into thousands of KV cache events, whereas the rest of the patterns in the paper are efficiently translated into fewer than 10 events in KV cache simulation and complete processing in seconds. Considering that LLM training and serving experiments often takes hours to complete, the processing time of SAS is still tolerable in this case.

9 Related Work

Sparse Attention Techniques Static sparse attention techniques use pre-defined input-agnostic attention patterns. Within this category, OpenAI's Sparse Transformers [9] proposes "strided" and "fixed" attention patterns for processing images. LongFormer [6] combines dilated sliding window attention and global attention for encoding long sequences. BigBird [38] features hardware-friendly block-sparse attention with a global+local+random pattern. LongNet [12] combines dilated attention with different dilation rates to scale to 1B tokens. StreamingLLM [35] proposes to always attend to the first few tokens and use sliding window attention for other input tokens. Dynamic sparse attention uses input-dependent attention patterns. Reformer [22] and Routing Transformer [29] cluster similar tokens based on token embeddings, where each token only attends to tokens within the same cluster. QKSparse [25] dynamically drops queries and keys of non-important tokens to reduce the size of the attention matrix. H2O [39] combines learned sparsity with KV caching optimizations for efficient inference, where the model dynamically select the least important token in the KV cache and replace it with new input tokens. MInference [21] uses an offline search to find the best sparse attention pattern for each head from a predefined search space. LESS [13] uses a low-rank cache to maintain information of tokens evicted from the main KV cache, thus improving

Table 5. AWS Trainium context encoding latency for static and dynamic sparse attention — SAS-Static/Dyn are SAS-generated NKI designs of static or dynamic sparse attention respectively. The SAS-Dyn results represent the combined execution time of the block-sparse flash-attention kernel (which utilizes memoization) and the importance score kernel. Time unit is millisecond.

	Seq = 2K			Seq = 8K			Seq = 16K		
	NKI-FA	SAS-Static	SAS-Dyn	NKI-FA	SAS-Static	SAS-Dyn	NKI-FA	SAS-Static	SAS-Dyn
Streaming-LLM	11.36	8.56 (1.32×)	8.98+1.14 (1.12×)	164.27	55.84 (2.94×)	102.84+12.36 (1.43×)	697.15	129.67 (5.38×)	417.2+46.7 (1.50×)
Block Local		8.57 (1.33×)			41.57 (3.95×)			89.32 (7.81×)	
Sliding Window		8.56 (1.32×)			41.57 (3.95×)			86.45 (8.06×)	
OpenAI Strided		8.56 (1.32×)			94.16 (1.74×)			391.82 (1.78×)	
OpenAI Fixed		8.56 (1.32×)			94.16 (1.74×)			391.9 (1.78×)	
Strided Block-Local		6.15 (1.85×)			22.79 (7.21×)			49.12 (14.12×)	
Avg. Speedup		1.41×			3.59×			6.49×	

Table 6. AWS Trainium token generation latency for static and dynamic sparse attention — Theoretical max is calculated based on fine-grained token-level sparsity patterns in the attention matrix. For dynamic sparsity, H2O technique is applied to each static sparse attention pattern to attend to regions otherwise left unattended. Numbers in parentheses show speedup over this implementation. Time unit is milliseconds.

	NKI-FA			SAS-Static	Theoretical-Max	SAS-Dyn
Static Sparsity with H2O	Seq = 2K	Seq = 8K	Seq = 16K	Seq = 2K / 8K / 16K	Seq = 2K / 8K / 16K	Seq = 2K / 8K / 16K
Streaming-LLM	0.80	2.38	4.39	0.80 (1.00× / 2.98× / 5.51×)	1.94× / 7.75× / 15.52×	0.92 (0.87× / 2.59× / 4.77×)
Block Local				0.32 (2.50× / 7.44× / 13.72×)	5.33× / 21.33× / 42.67×	0.49 (1.63× / 4.86× / 8.96×)
Sliding Window				0.49 (1.63× / 4.86× / 8.96×)	2× / 8× / 16×	0.80 (1.00× / 2.98× / 5.49×)
OpenAI Strided				0.49 (1.63× / 4.86× / 8.96×)	3.97× /15.54× / 30.17×	0.49 (1.63× / 4.86× / 8.96×)
OpenAI Fixed				0.32 (2.50× / 7.44× / 13.72×)	7.79× / 28.54× / 51.36×	0.49 (1.63× / 4.86× / 8.96×)
Strided Block-Local				0.32 (2.50× / 7.44× / 13.72×)	32× / 128× / 256×	0.49 (1.63× / 4.86× / 8.96×)
Avg. Speedup vs. NKI-FA				1.97× / 5.78× / 10.87×		1.39× / 4.17× / 7.68×

Table 7. Sparse Attention Pattern Analysis Time — Processing time for different sparse attention patterns across varying sequence lengths.

Pattern	16K	64K	128K
Sliding Window (window size=1K)	0.76	0.75	0.78
OpenAI Fixed (block size=64)	13.1	185.2	718.9

the model's long context capability. Native Sparse Attention [37] combines sliding window attention, compressed KV attention, and block-sparse attention to let the model collect global, local and fine-grained information while saving compute. Quest [32] divides the KV cache into pages and selectively attend to important pages for each query. MagicPig [8] levarages locality sensitive hashing to select an important set of KV tokens for each query token.

Accelerating Sparse Attention Existing efforts on accelerating sparse attention focus on either providing a small selection of sparse attention implementations or providing a programming abstraction. Examples of the first category are xFormers [24] and DeepSpeed [28], where sparse attention kernels are implemented in a hard-coded manner and gives limited flexibility for users to create their own patterns. In the second category, SPLAT [18] proposes to use affine-compressed-sparse-row format to represent a larger selection of sparse attention patterns and automatically generates performant sparse attention kernels for GPU training. SparseTIR [36] proposes a composable programming abstraction for sparse tensor compilers, but their abstraction focuses on generic, unstructured sparsity and is not optimal for sparse attention. FlexAttention [14] is a PyTorch [26] extension that allows users to manually define sparse attention mask, but it has limited support for dynamic sparse attention techniques. Among these efforts, SAS is the very first work that can generate optimized inference kernels for both static and dynamic sparse attention. In addition, to the best of our knowledge, SAS is also the first work that provides systematic support to sparse attention on multiple hardware backends.

10 Discussions

Generalizability to emerging sparse attention techniques While SAS's geometric representation offers high flexibility for supporting diverse sparse attention techniques, some extensions might be needed when implementing new sparse attention techniques. Here we discuss the extensions necessary for supporting three representative dynamic sparse attention techniques: NSA [37], Quest [32], and MagicPig [8]. SAS can support NSA and Quest with additional dynamic operators, requiring minimal effort under the current kernel synthesis flow. For MagicPig, in addition to new dynamic

operators for hash functions, SAS needs to be extended with an optional attention score modifier, which can be shared with other useful features like positional embedding. This requires modest changes to our attention kernel templates. In general the changes are light-weight and can be implemented with low effort.

Matching the performance of fused kernels We have demonstrated that SAS can generate highly performant kernels for a selection of sparse attention techniques on GPU and AWS Trainium. However, for emerging dynamic sparse attention techniques, performance of kernels generated by SAS may fall short compared with highly optimized fused kernels. At its current state, SAS is a generic kernel synthesis tool focusing on generality over performance. To achieve comparable performance with fused kernels, an optimizing compiler is required to find the best strategy for fusing dynamic components (maintaining attention score, block selection, etc.) with the main attention computation, which is beyond the scope of our work. We leave further performance optimization of sparse attention kernels for future work.

Compatibility with paged attention SAS integrates seamlessly with paged attention when the cache index ranges for each page is available. The mask function accepts KV cache location indices as input, enabling correct attention mask generation for any given page once its index range is known. Similarly, the cache ID function computes token indices assuming contiguous KV cache layout, which can be straightforwardly mapped to appropriate pages using the provided index ranges.

11 Conclusion and Future Work

In this paper, we introduced SAS, an automated sparse attention synthesizer for efficient LLM inference. SAS provides a unified generic interface for creating static and dynamic sparse attention patterns, and uses a geometric-based pattern analyzer to derive minimal KV cache sizes and generate optimized cache management functions. Evaluation shows that on Nvidia GPU, SAS achieved 1.10-1.22× speedup for context encoding and 2.68-2.80× speedup for token generation over FlexAttention; while on AWS Trainium, SAS delivered 1.41-6.49× speedup for context encoding and 1.39-10.87× speedup for token generation over a highly optimized dense attention baseline. Beyond performance gains, SAS dramatically reduces development complexity from hundreds of lines of manual code to just 5-10 lines using our programming abstraction.

For future work, we plan to extend SAS to support more runtime-dependent attention patterns, such as compression-based methods [13] and more recent techniques such as Native Sparse Attention [37] and Quest [32]. We also plan to extend SAS to support advanced batching mechanisms like chunked prefill and decode-maximal batching [2]. We will continue to improve the performance of SAS-generated sparse kernels by exploring better fusing strategies and integrating latest advances of attention kernels into our template.

Acknowledgments

We thank the anonymous reviewers and the shepherd Pengfei Zuo for their valuable insights and feedback.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. arXiv preprint arXiv:2303.08774, 2023.

[2] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, and Ramachandran Ramjee. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. arXiv preprint arXiv:2308.16369, 2023.

[3] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. arXiv preprint arXiv:2305.13245, 2023.

[4] Amazon Web Services. Neuron compiler. https://awsdocs-neuron.readthedocs-hosted.com/en/latest/compiler/index.html, 2024.

[5] Amazon Web Services. Neuron Kernel Interface (NKI) - Beta. https://awsdocs-neuron.readthedocs-hosted.com/en/latest/general/nki/index.html, 2025.

[6] Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. arXiv preprint arXiv:2004.05150, 2020.

[7] Zefan Cai, Yichi Zhang, Bofei Gao, Yuliang Liu, Tianyu Liu, Keming Lu, Wayne Xiong, Yue Dong, Baobao Chang, Junjie Hu, et al. Pyramidkv: Dynamic kv cache compression based on pyramidal information funneling. arXiv preprint arXiv:2406.02069, 2024.

[8] Zhuoming Chen, Ranajoy Sadhukhan, Zihao Ye, Yang Zhou, Jianyu Zhang, Niklas Nolte, Yuandong Tian, Matthijs Douze, Leon Bottou, Zhihao Jia, et al. Magicpig: Lsh sampling for efficient llm generation. arXiv preprint arXiv:2410.16179, 2024.

[9] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. arxiv 2019. arXiv preprint arXiv:1904.10509, 1904.

[10] Nvidia Corporation. Cuda c++ programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2025.

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

[12] Jiayu Ding, Shuming Ma, Li Dong, Xingxing Zhang, Shaohan Huang, Wenhui Wang, Nanning Zheng, and Furu Wei. Longnet: Scaling transformers to 1,000,000,000 tokens. arXiv preprint arXiv:2307.02486, 2023.

[13] Harry Dong, Xinyu Yang, Zhenyu Zhang, Zhangyang Wang, Yuejie Chi, and Beidi Chen. Get more with less: Synthesizing recurrence with kv cache compression for efficient llm inference. arXiv preprint arXiv:2402.09398, 2024.

[14] Juechu Dong, Boyuan Feng, Driss Guessous, Yanbo Liang, and Horace He. Flex attention: A programming model for generating optimized attention kernels. arXiv preprint arXiv:2412.05496, 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] Haozheng Fan, Hao Zhou, Guangtai Huang, Parameswaran Raman, Xinwei Fu, Gaurav Gupta, Dhananjay Ram, Yida Wang, and Jun Huan. Hlat: High-quality large language model pre-trained on aws trainium. In 2024 IEEE International Conference on Big Data (BigData), pages 2100–2109. IEEE, 2024.

[17] Xinwei Fu, Zhen Zhang, Haozheng Fan, Guangtai Huang, Mohammad El-Shabani, Randy Huang, Rahul Solanki, Fei Wu, Ron Diamant, and Yida Wang. Distributed training of large language models on aws trainium. In Proceedings of the 2024 ACM Symposium on Cloud Computing, pages 961–976, 2024.

[18] Ahan Gupta, Yueming Yuan, Devansh Jain, Yuhao Ge, David Aponte, Yanqi Zhou, and Charith Mendis. Splat: A framework for optimised gpu code-generation for sparse regular attention. arXiv preprint arXiv:2407.16847, 2024.

[19] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. Mistral 7b. arXiv preprint arXiv:2310.06825, 2023.

[20] Chenyu Jiang, Zhenkun Cai, Ye Tian, Zhen Jia, Yida Wang, and Chuan Wu. Dcp: Addressing input dynamism in long-context training via dynamic context parallelism. In Proceedings of the 31th Symposium on Operating Systems Principles, 2025.

[21] Huiqiang Jiang, Yucheng Li, Chengruidong Zhang, Qianhui Wu, Xufang Luo, Surin Ahn, Zhenhua Han, Amir Abdi, Dongsheng Li, Chin-Yew Lin, et al. Minference 1.0: Accelerating pre-filling for long-context llms via dynamic sparse attention. Advances in Neural Information Processing Systems, 37:52481–52515, 2024.

[22] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. arXiv preprint arXiv:2001.04451, 2020.

[23] Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. Snapkv: Llm knows what you are looking for before generation. Advances in Neural Information Processing Systems, 37:22947–22970, 2024.

[24] Meta. Hackable and optimized transformers building blocks, supporting a composable construction. https://github.com/facebookresearch/xformers, 2024.

[25] Matteo Pagliardini, Daniele Paliotta, Martin Jaggi, and François Fleuret. Faster causal attention over large sequences through sparse flash attention. arXiv preprint arXiv:2306.01160, 2023.

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

[27] Philippe Tillet. Fused Attention. https://triton-lang.org/main/getting-started/tutorials/06-fused-attention.html, 2025.

[28] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pages 3505–3506, 2020.

[29] Aurko Roy, Mohammad Saffar, Ashish Vaswani, and David Grangier. Efficient content-based sparse attention with routing transformers. Transactions of the Association for Computational Linguistics, 9:53–68, 2021.

[30] Amazon Web Services. Aws trainium: Get high performance for deep learning and generative ai training while lowering costs. https://aws.amazon.com/machine-learning/trainium/, 2024.

[31] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. arXiv preprint arXiv:1909.08053, 2019.

[32] Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. Quest: Query-aware sparsity for efficient long-context llm inference. arXiv preprint arXiv:2406.10774, 2024.

[33] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, pages 10–19, 2019.

[34] A Vaswani. Attention is all you need. Advances in Neural Information Processing Systems, 2017.

[35] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. arXiv preprint arXiv:2309.17453, 2023.

[36] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. Sparsetir: Composable abstractions for sparse compilation in deep learning. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, pages 660–678, 2023.

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

[38] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. Big bird: Transformers for longer sequences. Advances in neural information processing systems, 33:17283–17297, 2020.

[39] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. H2o: Heavy-hitter oracle for efficient generative inference of large language models. Advances in Neural Information Processing Systems, 36, 2024.