FG-Attn: Leveraging Fine-Grained Sparse Attention in Video Diffusion Models

Anonymous authors
Paper under double-blind review

Abstract

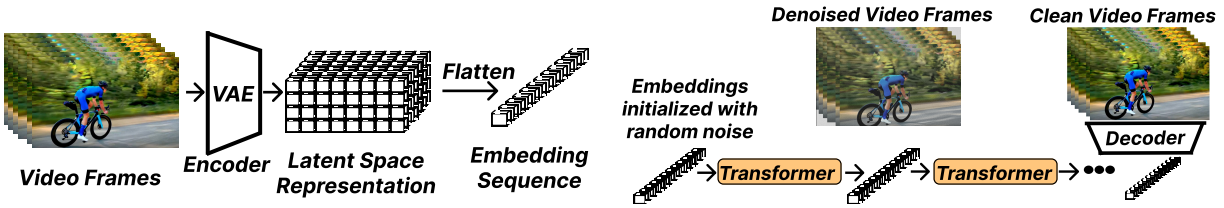
Using diffusion transformers for media generation may require evaluating attention over extremely long sequences, with attention layers accounting for the majority of generation latency. Exploiting sparsity in attention maps offers a promising opportunity to reduce this cost. In this work, we show that attention maps in diffusion transformers exhibit significant fine-grained sparsity in video generation models. Existing sparse attention methods, however, are too coarse-grained, leaving a large fraction of redundant computation unaddressed, or incur high overheads at finer granularity. We propose FG-Attn, a novel, low-overhead fine-grained sparse attention mechanism that skips score computations at the granularity of a $M \times N$ tile, where $N \geq 1$ and $M \geq 16$, and where each block is the result of query-key dot products between M queries and N keys. FG-Attn addresses the key challenge of hardware underutilization in sparse attention kernels on GPUs, without incurring the overheads of irregular memory access and redundant operations. FG-Attn can fully supersede existing sparse attention methods and extend block sparse attention methods to finer granularities on modern GPUs. At 70% sparsity, FG-Attn is up to 2.45× faster than state-of-art FlashInfer, and reduces attention kernel time by 14.7% on average. FG-Attn speeds up end-to-end video generation times by up to 1.40× (1.18× on average) over Flash Attention 3.

1 Introduction

Diffusion transformers (Peebles & Xie, 2023) (DiTs) are widely used for generative modeling because they can capture complex, high-dimensional data distributions over real-world data. This capability makes them particularly effective for media generation tasks spanning videos (Wan et al., 2025; Kong et al., 2024; HaCohen et al., 2026), audio (Kong et al., 2020; HaCohen et al., 2026), 3D models (Xiang et al., 2024; 2025), and images (Labs, 2024; Labs et al., 2025; Esser et al., 2024). DiTs generate synthetic data by iteratively refining a latent-space representation. For example, in video generation, each video is represented as a long sequence of embeddings that are progressively refined by a DiT before being decoded into a video, as illustrated in Fig. 1. A noisy video is first encoded into a set of embedding vectors (the *latent-space* representation) using a variational autoencoder (Fig. 1a). These embedding vectors are flattened into a single sequence. A DiT progressively refines the embeddings to produce clean video-frame embeddings (Fig. 1b).

Even short, low-resolution videos yield extremely long embedding sequences. For example, Wan 2.1 1.3B (Wan et al., 2025) encodes a 5-second 720p video into $\sim$ 75000 embedding vectors, requiring 5 minutes to generate on an H100 GPU; Wan 2.1 14B requires over 15 minutes. The majority of this generation time is incurred by the transformer's attention layers. Since attention scales quadratically with sequence length, higher resolution or longer videos rapidly increase latency: generating a 10-second video requires twice as many embeddings and roughly 4× attention computation. For Wan 2.1 1.3B, attention layers take 61% of the overall time to produce 49 frames and 91% for 81 frames with Flash Attention 2 (Dao, 2024).

It is well known that the attention computation in DiTs contains significant redundancy since many query–key pairs yield negligible attention scores (Xi et al., 2025; Zhang et al., 2025d). Skipping computation of these negligible scores can significantly accelerate video generation. *Sparse attention* mechanisms that enable this require (1) determining the sparse attention *mask*, i.e., which subset of the attention scores can be



(a) A set of noisy video frames is encoded into a sequence of latent space embeddings using a VQVAE.

(b) The diffusion transformer denoises the embeddings to generate clean video frame embeddings.

Figure 1: Latent space representation of video frames, represented as a set sequence of embeddings and encoded using a VQVAE, can be denoised using a DiT to produce embeddings of clean frames.

skipped, and (2) sparse attention kernel that can efficiently skip attention computation without incurring implementation overheads. Prior works propose a range of techniques for (1), e.g., (Xi et al., 2025; Yang et al., 2026). For (2), all existing sparse attention mechanisms use *block-sparse attention* kernels. These block-sparse attention mechanisms (Dong et al., 2024; Guo et al., 2024) divide the attention score matrix into *tiles* of size M×N, for M query tokens and N key tokens–referred to as a *block*. Computing a block of attention scores is skipped only if *all* scores in that block are predicted to be near zero. FlashInfer (Ye et al., 2025) enables *variable* block sparse attention, where different regions of the attention map can use different block sizes. Thus, FlashInfer can skip computation over irregularly sized blocks rather than requiring a single fixed block size.

We observe that attention maps in diffusion transformers exhibit *fine-grained* sparsity: many query–key products are near zero even when others in the same block are not. Exploiting this finer structure has the potential to substantially reduce computation (floating-point operations, or FLOPs). For example, skipping 16×16 blocks of attention can reduce the required FLOPs by up to 70%, compared to only $\sim 15\%$ with 64×64 block sparsity, without noticeable degradation in video quality (Section 4). However, all existing block-sparse attention implementations (except FlashInfer (Ye et al., 2025), which is discussed below) exploit sparsity only at a coarse block granularity. They skip attention computation over large contiguous regions of the attention score matrix, typically blocks of size 128×128. Reducing the block size in these implementations is not directly supported. The reason for this is that high-performance block-sparse attention is implemented using dense *tile* operations: to compute M output vectors, the kernel loads M query vectors (referred to as a *tile* of queries) and iterates through the key/value sequence by loading N key/value vectors (*tiles* of keys/values) at a time, from main memory into the GPU's shared memory. Block-sparse attention either computes all corresponding scores in an $M \times N$ block or skips it. The tile dimensions M and N are typically set to 128 to match hardware requirements, such as tensor core sizes and memory transaction widths. Reducing them below these sizes degrades hardware utilization, making fine-grained block-sparse attention at, say, 16×16 inefficient. A consequence of using coarse-grained block-sparse attention is that prior sparse-video methods rely on token clustering (Yang et al., 2026; Luo et al., 2026) to induce a block-sparse structure before attention is evaluated. This clustering process has to be performed online at inference time, thereby adding significant overhead. Larger clusters may alleviate this overhead, but compromise generation accuracy.

Our goal in this work is to enable highly efficient *fine-grained* block-sparse attention that skips blocks at a granularity of $M \times N$, with $M \geq 16$, $N \geq 1$ (corresponding to M contiguous queries and N contiguous keys) on modern GPUs. To this end, we propose FG-Attn, which comprises a highly efficient fine-grain *sparse attention kernel implementation* and a *sparse mask determination strategy* to identify fine-grain blocks to be skipped. There are two major challenges in avoiding significant efficiency overheads when enabling finer-grained block sizes:

Challenge 1. Naively reducing the N dimension to, say, $N = 16$ or $N = 1$, means that at each iteration over the key/value sequence, the kernel loads only a small number keys to multiply with M query tokens. This is inefficient because high-performance attention kernels use larger tiles, typically $N = 128$, to amortize instruction scheduling, synchronization, and memory-access latencies, while keeping tensor cores well utilized. FlashInfer addresses this by gathering a sparse set of $N = 128$ key/value vectors at each iteration and

packing them into a tile in shared memory. However, this gather step introduces indirect memory accesses, per-token address computation, and scheduling overhead, which can dominate runtime at very fine sparsity granularities, as we demonstrate.

Challenge 2. Reducing the M dimension (which maps to M queries) to smaller sizes than 128 leads to the same challenge where the hardware becomes underutilized. Gathering sparse queries into a larger block size to improve utilization would be significantly inefficient, because now queries would have to be loaded for each tiled multiply. In the baseline implementation, the M queries need only be loaded *once* initially and are then multiplied over all keys iteratively.

FG-Attn: Fine-grain sparse attention kernel. To efficiently gather key/value tokens (**challenge 1**), FG-Attn implements an asynchronous gather-load mechanism to fetch sparse key/value vectors from irregular memory locations and pack them into tiles of size $N = 128$ in shared memory. To hide the high latency of these indirect loads, the gather-load is *parallelized* and *pipelined* with attention computation (Section 5.1). To support small query tile sizes $M \leq 128$ (**challenge 2**), FG-Attn groups together smaller query tiles (for example, eight 16-query tiles into one 128-query tile) before iterating through keys and values. This grouping is based on the sparse mask, so query tiles that attend to similar key/value sets are grouped together to reduce redundant key/value loads while enabling efficient hardware utilization (Section 5.2). After merging, different output tiles, each computed from a query tile of size M, may attend to different numbers of key/value tokens and therefore require different amounts of work. Tiles that iterate over fewer tokens may finish earlier than tiles that iterate over more tokens, leading to hardware underutilization from *load imbalance* across the GPU Streaming Multiprocessors (SMs). FG-Attn introduces a scheduling mechanism in the attention kernel that dynamically partitions work equitably across SMs to reduce idleness (Section 5.2).

FG-Attn: Mask determination. To exploit fine-grained sparsity, FG-Attn must determine which $M \times N$ query–key blocks can be skipped without computing the full attention matrix. FG-Attn uses a training-free strategy for this purpose: For each query tile and key tile, we compute their mean query and mean key vectors. For fine-grained masks, these mean vectors provide effective tile-level summaries for estimating the attention score of the corresponding block. We then retain only the blocks selected by top-p masking (Holtzman et al., 2019) (Section 5.3).

We demonstrate the efficiency of FG-Attn in two ways: First, we use a microbenchmark to demonstrate that our kernel is faster than all existing sparse attention kernels for all sparsity levels. At a sparsity of 70%, FG-Attn speeds up attention computation by up to 2.45× in comparison to the state-of-art FlashInfer. Even without accounting for the planning time overhead, FG-Attn reduces the attention kernel latency by 14.7% on average. Second, for end-to-end video generation using state-of-art generation models such as HunyuanVideo (Kong et al., 2024), Wan 2.1 (Wan et al., 2025), and LTX-2 (HaCohen et al., 2026), FG-Attn provides speedups of up to 1.40× (1.18 on average), compared to Flash Attention 3, with an average PSNR of 25.1 relative to dense attention.

- To our knowledge, FG-Attn is the fastest existing sparse attention kernel implementation on modern GPUs. Thus, FG-Attn's fine-grain sparse kernel implementation can flexibly supersede all existing block sparse attention mechanisms.
- We demonstrate that video diffusion models contain substantial fine-grained sparsity in their attention maps. We show that existing block-sparse attention mechanisms do not fully leverage this sparsity. We identify and address the key challenges towards a practical and efficient fine-grain sparse kernel implementation on GPUs.
- We propose a lightweight strategy for sparse mask generation that operates entirely without retraining. This enables seamless integration with state-of-the-art video DiTs.

2 Background

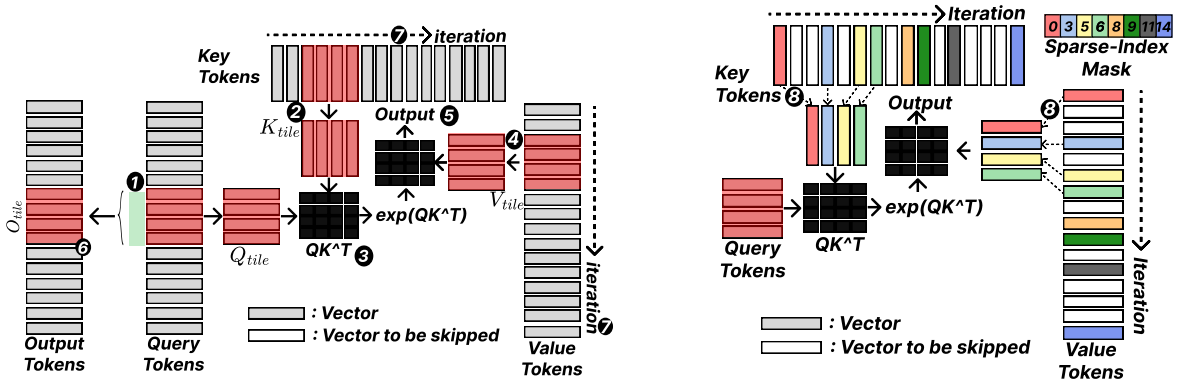
Modern GPU Architecture. A GPU consists of many streaming multiprocessors (SMs), each running thousands of threads organized into thread blocks and 32-thread warps; each SM has a small, fast shared

memory backed by larger but slower high-bandwidth memory (HBM). We refer the reader to Section A of the Appendix for a more detailed description of these components.

Efficient dense attention. Efficient attention implementations (Flash Attention 1, 2, and 3 (Dao et al., 2022; Dao, 2024; Shah et al., 2024)) fuse the attention score computation $\mathbf{P} = \mathbf{softmax}\left(\mathbf{QK}^\top/\sqrt{D}\right)$ and multiplication of attention scores with values $\mathbf{O} = \mathbf{PV}$ for fast execution. Fig. 2a depicts how Flash Attention is implemented in a GPU. The kernel takes queries, keys, and values $(\mathbf{Q}, \mathbf{K}, \mathbf{V})$ matrices as input and produces an output matrix $\mathbf{O}$. The queries, keys, values, and output tokens form a sequence of N tokens (gray bars), as shown in the figure.

GPU threads are organized into thread blocks, where each thread block cooperatively computes a tile of the output for a particular attention head. In the figure, the output tokens labeled $\mathbf{O}_{\text{tile}}$ highlighted in red is computed by the threads of one thread block. This corresponds to a set of query tokens highlighted in green. To compute $\mathbf{O}_{\text{tile}}$, the block first loads a corresponding set of queries $\mathbf{Q}_{\text{tile}}$ into the GPU's scratchpad (shared) memory ❶. Then, the first set of key and value tokens, $\mathbf{K}_{\text{tile}}$ ❷ and $\mathbf{V}_{\text{tile}}$ ❹, is loaded into shared memory. This tile is used to compute $\mathbf{Q}_{\text{tile}}\mathbf{K}_{\text{tile}}^T$ using the tensor core ❸, followed by exponentiation to compute $e^{\mathbf{Q}_{\text{tile}}\mathbf{K}_{\text{tile}}^T}$. This result stored in registers is then multiplied by the corresponding $\mathbf{V}_{\text{tile}}$ using the tensor core ❺. The result of the computation is added to $\mathbf{O}_{reg}$, a slice of output in registers ❻. In the next iteration, the next tile $\mathbf{K}_{\text{tile}}, \mathbf{V}_{\text{tile}}$ in the sequence is loaded into shared memory, and computation is repeated ❼. The thread block also keeps the running sum of exponents, $e^{\mathbf{Q}_{\text{tile}}\mathbf{K}_{\text{tile}}^T}$, required for softmax normalization in registers. The running sum is used to normalize the accumulated partial output before writing the final $\mathbf{O}_{\text{tile}}$.

Efficient block sparse attention. Block-sparse attention follows the same tiled execution pattern as dense Flash Attention, but uses a block-level sparse mask to decide which key/value tiles are processed for each query tile. For a given $\mathbf{Q}_{\text{tile}}$, if the mask selects a key/value block, the corresponding $\mathbf{K}_{\text{tile}}$ ❷ and $\mathbf{V}_{\text{tile}}$ ❹ are loaded into shared memory and used in the tensor core attention computation. Otherwise, that key/value block is skipped entirely: it is not loaded into shared memory, and the corresponding attention scores and values are not computed. This reduces both, expensive memory traffic and computation in proportion to the number of skipped key/value blocks.



(a) Flash Attention: tiles of keys, values are loaded into shared memory. Tensor cores are used to compute a partial sum of the output vector.

(b) FG-Attn: only relevant keys and values corresponding to indices in the sparse index mask are loaded into shared memory.

Figure 2: Implementation overview of Flash Attention (Dao et al., 2022) and FG-Attn.

3 Related Work

Block-sparse attention. Several implementations of block-sparse attention (Guo et al., 2024; Dao et al., 2022; Dong et al., 2024; Wang et al., 2025) propose coarse-grained mechanisms that skip entire tiles of attention scores, typically at 64×64 or 128×128 granularity in half-precision. These methods have been widely

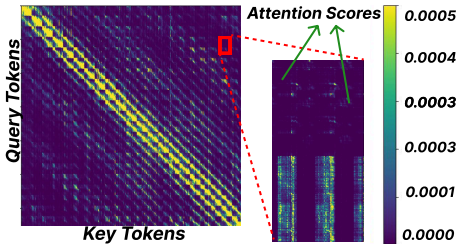
adopted in LLM inference (Jiang et al., 2024; Yuan et al., 2025). However, current block-sparse methods do not support smaller tile sizes: reducing block-size either fails to compile or leads to severe hardware underutilization due to tensor core width constraints (Section 4). FlashInfer (Ye et al., 2025) introduced variable block-sparse attention, enabling finer-grained sparsity beyond what earlier block-sparse implementations support. However, FlashInfer incurs nontrivial overhead in the planning stage when scheduling work to SMs at runtime and pads query-key dot product computations at tiles smaller than 128×1, resulting in wasted compute. We compare FG-Attn against FlashInfer in Section 6.1.

Sparse attention for video diffusion models. Many prior works (Li et al., 2026; Xi et al., 2025; Xu et al., 2025) leverage fixed sparsity patterns based on empirical observations of attention scores. SpargeAttention (Zhang et al., 2025d) predicts a mask based on pooling blocks of queries and keys. These methods use block-sparse attention to compute attention and are orthogonal to our proposed approach. They can leverage FG-Attn for their underlying sparse attention mechanism to exploit fine-grain sparsity. Clustering-based methods (Yang et al., 2026; Luo et al., 2026) reorder similar sets of keys and queries together in order to promote block sparsity in attention score computation. While clustering tokens is a well-known technique to promote sparsity to use block-sparse attention (Vyas et al., 2020), these methods incur high costs in clustering similar query and key tokens. In contrast, FG-Attn directly leverages fine-grained masks without requiring global token clustering or reordering to expose coarse block structure. Training-based methods (Zhang et al., 2026b; 2025f; 2026a; Wu et al., 2026) learn to predict the sparsity mask. All these methods, however, are limited to coarse-grained block skipping. For example, Video Sparse Attention (Zhang et al., 2026b) can also leverage FG-Attn for its sparse attention mechanism, and are thus orthogonal and complementary.

Other techniques to accelerate video diffusion. SageAttention (Zhang et al., 2025c;a;e;b) uses quantization to speed up attention layers in transformers. Since quantization and sparsity are orthogonal, FG-Attn can be applied on top of quantization-based approaches. Caching-based approaches such as DeepCache (Ma et al., 2024), TeaCache (Liu et al., 2025), and TaoCache (Fan et al., 2025) exploit temporal redundancy across denoising steps. These approaches are orthogonal to our work and can be combined with FG-Attn to further accelerate attention computation.

4 Fine-Grained Sparsity in Attention Computation

Fig. 3a shows a heatmap of the attention scores in one attention head of the Wan 2.1 1.3B vDIT model. From the figure, we observe that the vast majority of attention scores are close to zero. Over 85% of the scores are below $0.5/N$ (N is the sequence length) in this example. Attention computation can be significantly accelerated by skipping these score computations without sacrificing output quality. For sequence length N and model dimension D, dense attention costs $O(N^2D)$ floating-point operations. If only a fraction $\rho \in [0, 1]$ of the N^2 query–key pairs are retained (i.e., ρ is the nonzero density, $\rho = 1 - \text{sparsity}$), the cost becomes $O(\rho N^2 D)$.



(a) Sparsity in attention computation: attention scores are highly sparse and irregular.

Block size	Sparsity	TFLOPs
128×128	5.5%	0.519
64×64	22.8%	0.424
32×32	47.7%	0.287
16×16	70.7%	0.161

(b) Sparsity vs. block size: % of $M \times M$ attention map blocks with all scores $\leq$ threshold.

Figure 3: Attention maps are very sparse. Block sparsity increases significantly at finer granularity.

Table 3b reports attention map sparsity at different block sizes, measured as the fraction of $M \times M$ blocks with all scores below a threshold of $0.5/N$, where N is the sequence length. Finer blocks (16×16) yield about

70% sparsity, while coarser blocks (64×64) achieve only 22%. This shows that finer granularity offers much greater opportunity for speedup, yet existing block-sparse implementations cannot exploit blocks smaller than 128×128.

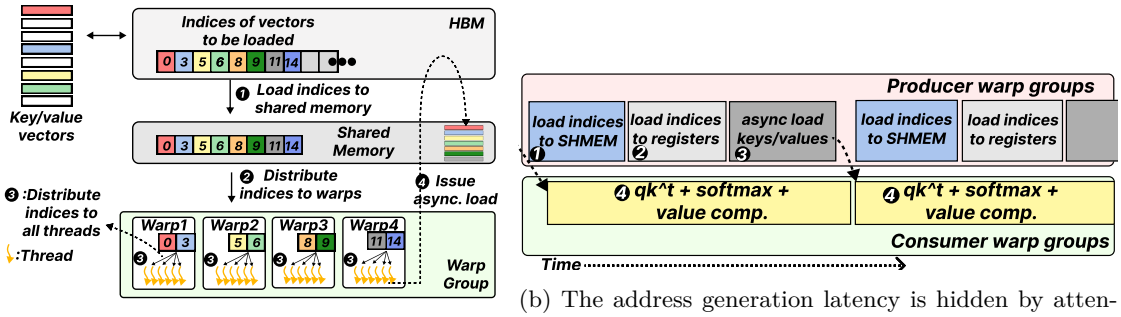
5 Method

5.1 Fetching Keys/Value Tokens Using Gather-Load.

In block-sparse attention (Guo et al., 2024; Dong et al., 2024), a block of M queries $\mathbf{Q}_{tile}$ and N contiguous keys $\mathbf{K}_{tile}$ are loaded into shared memory and multiplied using tensor cores. On GPUs, these loads are accelerated by the Tensor Memory Accelerator (TMA). In FG-Attn, we instead must *gather* M non-contiguous keys, pack them into a tile $\mathbf{K}_{rel}$, to compute $\mathbf{Q}_{tile}\mathbf{K}_{rel}^T$ (❽ in Fig. 2b). To do this efficiently, we introduce a new *gather-load* primitive, which loads sparse key/value vectors from HBM into shared memory (Section A of appendix) based on the attention mask. The corresponding value vectors are loaded with the same indices, and tensor cores compute partial sums across tiles until all relevant slices are processed. Loading sparse key/value vectors requires first computing the addresses of elements specified by the sparse mask before issuing the load. Efficient gather-load requires (i) fast address generation for sparse indices and (ii) hiding this latency. We achieve (i) by parallelizing index-to-address translation across threads in a warp group and (ii) by pipelining (i.e., overlapping) the gather-load with attention computation.

The sparse gather-load primitive. For each group of M queries, only the relevant keys indicated by the sparse mask are loaded (Fig. 4a). Indices are first cooperatively loaded into shared memory ❶ from HBM, then distributed across warps in a warp group ❷. Each warp broadcasts its indices to threads ❸, after which threads compute addresses and issue asynchronous loads to fetch the sparse vectors into shared memory ❹.

Overlapped indirect key/value load latency with attention computation. In modern GPUs, threads in a thread block can be divided into two groups: *producer* threads that load data and *consumer* threads that perform computation. Producers load query, key, and value tiles into shared memory, while consumers compute and accumulate partial sums (Fig. 4b). At each iteration, the producer loads indices for N keys from the sparse mask into shared memory ❶, moves them into registers ❷, and issues asynchronous loads for the corresponding rows from global memory ❸. The consumer computes attention scores, outputs in parallel ❹, fully hiding address generation and load latency.



(a) Detailed diagram showing the implementation of the gather + load operation.

(b) The address generation latency is hidden by attention computation by having the gather-load operation in the producer threads.

Figure 4: FG-Attn parallelizes gather-load by distributing the selected key indices across threads, which then cooperatively compute addresses and issue loads to fetch the key/value tiles.

5.2 Grouping Query Tiles Attending to Similar Sets of Keys/Values

In FG-Attn, however, different query tiles (of size $M \leq 128$) have to be grouped together to form groups of 128 queries for hardware efficiency (see Challenge 1, Section 1). These query tiles iterate over different subsets of keys/values, as determined by the sparse mask. When iterating over keys/values, the kernel

may need to load redundant key/value tokens. For example, in Fig. 5, when $M = 32$, a single thread block contains four query tiles. If the sparse masks of four consecutive query tiles differ substantially (4 consecutive rows in Fig. 5), the merged 128-query tile must load all key/value tokens selected by any of the four tiles. Instead, in FG-Attn, we minimize the number of redundant key/value loads. We collect query tiles that load *similar* sets of key/value tiles into groups of tiles containing 128 queries. For example, in Fig. 5, the first and third sparse mask rows are similar, indicating that the corresponding query tiles attend to a similar set of key/values. These tiles, along with two other similar tiles, can be grouped into a 128-query set into a thread block.

To do this, we first define how two query tiles are *similar*: For an $M \times N$ block sparse mask $\mathbf{B}$, we define the distance between two query tiles as: $d(i,j) = \sum_k b_{ik} \oplus b_{jk}$ where $\oplus$ denotes bitwise XOR, and b_{ik} is a boolean. This counts the number of key blocks selected by exactly one of the two query blocks. A smaller value, therefore, means that the two query tiles attend to more similar sets of KV tokens. For query tile size M, each merged group contains $R = 128/M$ query tiles. Given the sparse mask $\mathbf{B}$, FG-Attn groups query tiles by choosing groups with small distance to a representative tile:

$$\min_{\{G_g, c_g\}} \sum_g \sum_{i \in G_g} d(i, c_g) \quad \text{s.t. } |G_g| = R,\ c_g \in G_g. \tag{1}$$

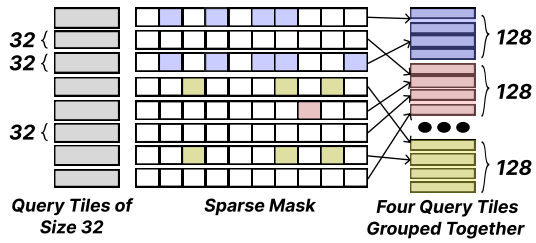


Figure 5: Query tiles (of size 32) attending to the similar sets of keys are grouped together into sets of 128 queries.

Here, G_g is a group of query tiles mapped to one thread block, and c_g is the representative query tile for that group. Minimizing this objective groups query tiles with similar key/value access patterns, reducing redundant key/value loads within each thread block. To find this grouping, we use an approximate, greedy approach: starting from the first ungrouped query tile, we select the remaining $R-1$ ungrouped query tiles with the smallest XOR distance to it, where $R = 128/M$. These R tiles form one merged 128-query group assigned to a thread block. We repeat this process until all query tiles are assigned to a group. This heuristic groups query tiles having small distances, reducing the union of key/value tiles loaded by each thread block.

Load balancing. After query merging, each thread block processes a 128-query tile, but different tiles may attend to different numbers of key/value tokens, as determined by the sparse mask. Since the work for an output tile is proportional to the number of key/value tokens it attends to, different thread blocks can have different runtimes, creating load imbalance across GPU Streaming Multiprocessors (SMs). Prior work addresses this by sorting output tiles on the CPU using a longest-processing-time policy based on their key/value counts, but this CPU-side scheduling creates significant overheads. In contrast, FG-Attn performs work selection directly on the GPU inside the persistent attention kernel, allowing thread blocks to fetch new output tiles without CPU-side scheduling. Details are provided in Section 5.4.

5.3 Determining the Fine-grain Sparse Mask with Top-p Masking.

To exploit fine-grained sparsity, we need a mask that efficiently identifies which query–key slices produce *significant* attention scores. We need to do this without computing the full attention score matrix, as this is prohibitively expensive. We note that in the context of video diffusion models, the query and key tokens forming an $M \times N$ attention block correspond to adjacent pixels in space and time. Such adjacent tokens typically exhibit responses similar to those of their surrounding queries and keys. Motivated by this observation, we propose a simple, lightweight strategy for determining the attention mask. We compute the average of a tile of queries and keys to summarize the corresponding $M \times N$ block of attention scores as follows:

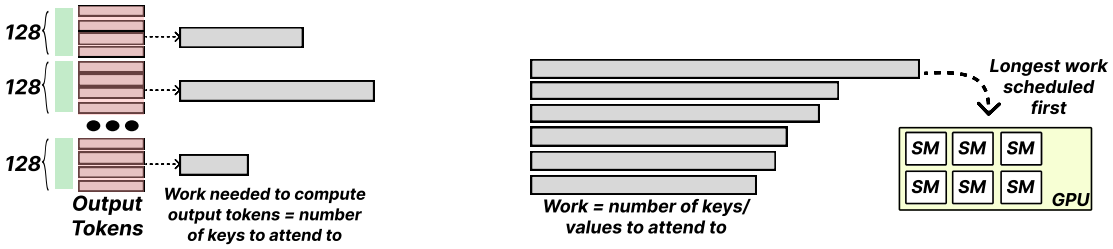
$$\mathbf{q}_{\text{avg}}^{(i)} = \frac{1}{M} \sum_{m=1}^{M} \mathbf{q}_{i,m}, \qquad \mathbf{k}_{\text{avg}}^{(j)} = \frac{1}{N} \sum_{n=1}^{N} \mathbf{k}_{j,n}, \qquad \hat{\mathbf{A}} = \text{softmax}\Big(\mathbf{Q}_{\text{avg}} \mathbf{K}_{\text{avg}}^{\top} / \sqrt{D}\Big),$$

where $\mathbf{q}_{\text{avg}}^{(i)}$ and $\mathbf{k}_{\text{avg}}^{(j)}$ are rows of $\mathbf{Q}_{\text{avg}}$ and $\mathbf{K}_{\text{avg}}$, respectively. The block is included if the dot product $\mathbf{q^{(i)}}_{\text{avg}} \cdot \mathbf{k^{(j)}}_{\text{avg}}$ is significant. We determine significance by computing the approximate attention distribution

between each averaged query block and all averaged key blocks, $\hat{\mathbf{A}}$. containing elements $\hat{a}_{ij}$, $1 \leq i \leq \lceil S/M \rceil, 1 \leq j \leq \lceil S/N \rceil$. $\hat{a}_{ij}$ denotes the approximate importance of key block j for query block i. For each query block i, we sort key blocks by $\hat{a}_{ij}$ and retain the smallest set whose cumulative probability mass is at least p. These retained key blocks define the sparse mask used by FG-Attn. This mean-query/key top-p mask avoids computing the full token-level attention matrix while preserving the dominant attention regions. Since each approximate score represents an $M \times N$ block, mask construction reduces the number of score computations by roughly a factor of MN compared to dense attention.

5.4 Persistent Work Scheduler

In Flash Attention (Dao et al., 2022), each attention head assigns a group of 128 query tokens to a single thread block, which produces the corresponding output tokens. In FG-Attn, however, query groups (of size 128 after merging) attend to a variable number of key–value pairs, which causes load imbalance: some thread blocks finish quickly and sit idle while others grind through much longer key–value lists. To address this, FG-Attn uses a persistent work-stealing scheduler (Fig. 6). Each 128-query block (shown in green in Fig. 6a) is processed by one thread block and attends to a different number of key–value pairs, indicated by the grey bars. Before dispatch, we estimate the cost of each output tile as the length of its associated key–value list (Fig. 6a).



(a) Number of keys/values (KVs) loaded for each query block is first computed.

(b) The query block requiring the longest processing time (proportional to the number of KVs to be processed) is scheduled first.

Figure 6: Scheduling work across output tiles: the cost of each tile is determined by the number of key–value pairs its query block attends to.

The scheduler then applies a longest-processing-time-first (LPT) policy, prioritizing the tiles with the largest key–value lists, since these have the longest expected runtime. Similar LPT-based scheduling is used in FlashInfer (Ye et al., 2025), FA2 (Dao, 2024), and FA3 (Shah et al., 2024); FlashInfer, for instance, builds the schedule on the CPU using a min-heap. We go further by making the scheduler persistent and tailoring it to variable block-sparse attention, which eliminates the kernel-launch and tile-assignment overheads that would otherwise dominate at our granularity. Concretely, each thread block fetches the highest-cost remaining tile directly from a GPU-side work queue, processes it, and then loops back for the next one, so all thread blocks stay busy until the head is complete.

6 Results

We perform two evaluations: First, we micro-benchmark FG-Attn against the only other available fine-grained block-sparse attention mechanism to our knowledge, FlashInfer (Ye et al., 2025) (FI), measuring performance on randomly generated masks across various sparsity levels. We evaluate two mask-merging strategies of FG-Attn: one obtained by grouping query tiles to minimize objective Eq. 1 (FG-Attn +merge-XOR, Section 5.2), and one obtained by merging consecutive query tiles (FG-Attn +merge-consecutive). Second, we evaluate video generation times on three open-source video generation models: (1) LTX-2 (HaCohen et al., 2026), (2) Wan 2.1 (Wan et al., 2025) 14B, and (3) HunyuanVideo (Kong et al., 2024). We report end-to-end video generation times for the following configurations: 1) the baseline dense Flash Attention 3 (FA3) implementation, 2) FlashInfer, using SparseVideoGen2 (Yang et al., 2026) to generate its sparse mask (FI+SVG2), and 3) FG-Attn +merge-XOR with block sizes: 16×16, 32×32, and 64×64 (FG-Attn-16,

FG-Attn-32, FG-Attn-64 respectively). We also present PSNR/SSIM scores for quality comparisons to the baseline video from SVG2, Radial Attention (Li et al., 2026), Sparge Attention (Zhang et al., 2025d), and SparseVideoGen (Xi et al., 2025) (SVG). Prior works, such as SVG, Sparge, and Radial Attention, also exploit sparse attention, but are built on Flash Attention 2. Thus, we are unable to do an apples-to-apples efficiency comparison and only compare the video quality (PSNR/SSIM). We note, however, that FG-Attn is orthogonal to these works: their mask-determination strategies are fully compatible with FG-Attn's sparse attention kernel, enabling finer granularity.

We implement FG-Attn on top of Flash Attention 3 (Shah et al., 2024) in BFloat16 for an H100 GPU. We sample 20 prompts from the Penguin dataset (Kong et al., 2024) for the video, and report video generation times and quality as the arithmetic mean of each metric across all generated videos. Similar to prior work (Yang et al., 2026; Xi et al., 2025; Li et al., 2026), we perform a warmup for the first few denoising iterations during which we compute dense attention for all layers. We set the warmup iterations to 24% of the total for Wan 2.1 and LTX-2, and to 10% for HunyuanVideo. The remaining denoising iterations are run with sparse attention. We set the top-p threshold to 0.9.

6.1 FG-Attn Efficiency Analysis

FG-Attn vs. FI at different sparsities. To evaluate how efficiently FG-Attn computes attention at different levels of sparsity, we generate a random attention mask while controlling the level of sparsity, benchmark the speedup generated, and compare it against FI. Fig. 7 shows the attention computation latency as a breakdown between planning and kernel computation for different sparse mask block sizes at 70% sparsity and 50% sparsity. From these plots, we make the following observations. First, at query tile size 128, FG-Attn achieves speedups of up to 2.60× (2.56× on average) at 70% sparsity, and up to 1.63× (1.59× on average) at 50% sparsity over FA3. At this tile size, there is no additional overhead from merging masks together, since the query tile is already of size 128. Thus, the runtime is dominated by the sparse attention kernel, allowing FG-Attn to directly benefit from reduced key/value computation. Compared to FI, FG-Attn achieves average speedups of 2.45× and 2.38× at 70% and 50% sparsity, respectively. This is because FI incurs significant overhead from planning and constructing the sparse indices on the CPU for a given mask shape, fully avoided by FG-Attn. Second, at query tile size 128, FG-Attn reduces kernel computation time relative to FI by 14.7% and 13.8% on average at 70% and 50% sparsity, respectively. This improvement comes from the parallelized gather-load primitive used by FG-Attn, as opposed to FI's approach, where individual threads load sparse indices from global memory and then issue the corresponding key/value memory loads. Third, at smaller query tile sizes (64× and 32×), FG-Attn +merge-consecutive and FG-Attn +merge-XOR kernels provide smaller speedups: up to 1.57× (1.54× on average) and up to 1.07× (1.06× on average) respectively at 70% sparsity, and a slowdown at 50% sparsity. This is expected because the random masks used in the microbenchmark exhibit fewer correlations in the key blocks attended by different query blocks. As a result, mask merging has fewer opportunities to exploit overlap, and merge-consecutive and merge-XOR provide similar benefits. Fourth, at smaller query block sizes 32 and 64, the planning time of merge-XOR becomes significant, where it accounts for up to 54.2% and 29.1% of kernel time on average. However, this planning time is much lower than FI's planning time. Overall, FG-Attn outperforms FI across a full range of sparsity and block sizes.

Ablation: FG-Attn at different levels of skew and sparsity in sparse mask.

In this experiment, we aim to evaluate the compute throughput of FG-Attn at varying degrees of skew in the number of key–value tokens across query blocks. A greater skew indicates greater variance in the number of KV blocks each query block must iterate over, leading to load imbalance. FG-Attn mitigates using its persistent work scheduler (Section 5.4). To study this, for each sparse mask, we define the skew as: skew $= \frac{L_{\max} - L_{\min}}{L_{\max}}$, where $L_{\max}$ and $L_{\min}$ denote the maximum and minimum number of key/value blocks attended by any query tile, respectively. A skew of 0 corresponds to a balanced mask in which all query tiles attend to the same number of key/value blocks, while larger skew values indicate greater variation in work across query tiles. To generate a mask with a target skew, we first choose $L_{\max}$ and set $L_{\min} = (1 - \text{skew})L_{\max}$. We then assign each query tile a key/value length sampled between $L_{\min}$ and $L_{\max}$, and randomly select that many key/value blocks for the tile. This method lets us control the degree of load imbalance while keeping the sparsity fixed.

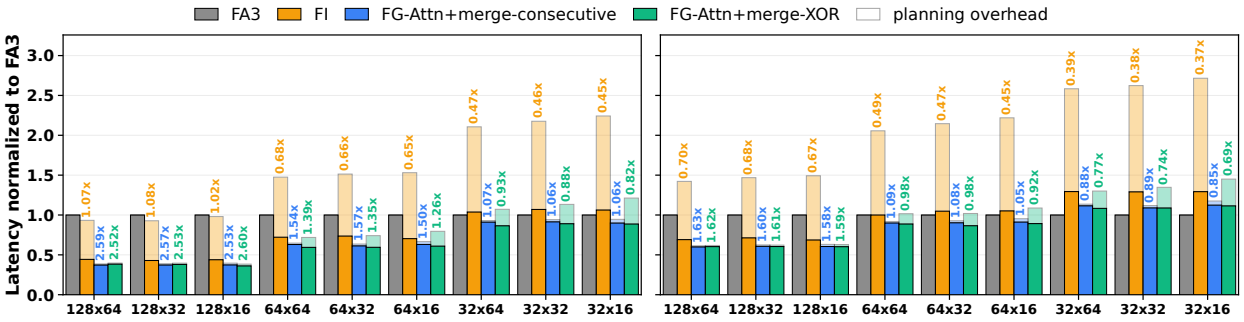


Figure 7: Attention time at two sparsity levels (70%, left, and 50%, right), across block sizes.

Fig. 8 shows the compute throughput (TFLOPs/s) of FA3, FG-Attn with the persistent work scheduler (FG-Attn), and FG-Attn without the scheduler (FG-Attn (no scheduler)), across different sparsity levels and degrees of skew. We chose a block size 128×32 for this experiment. From these experiments, we make three observations. First, FG-Attn achieves throughput within 18% of dense FA3. This is significant because FG-Attn executes fine-grained sparse attention, which requires irregular key/value gathers, sparse-mask processing, and dynamic work scheduling, whereas FA3 operates on dense, contiguous tiles. Despite these additional overheads, FG-Attn preserves most of the compute efficiency of dense attention while skipping masked key/value blocks. This shows that FG-Attn better converts sparsity into high tensor core utilization, rather than incurring performance losses due to scheduler overheads or irregular execution.

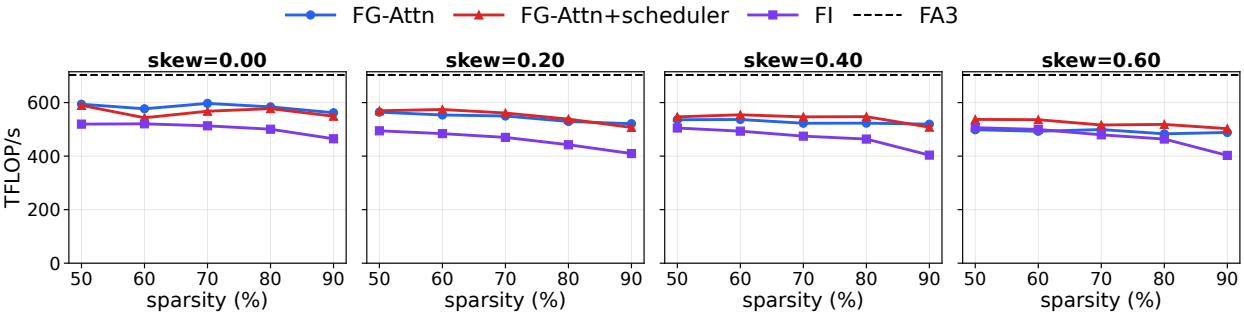


Figure 8: Achieved compute throughput of FG-Attn and FG-Attn (no scheduler) compared to dense flash attention at different sparsities, at varying levels of skew in key/value sequence lengths.

Second, the persistent work scheduler improves utilization by assigning output tiles to SMs dynamically based on their estimated work. As a result, FG-Attn improves throughput by up to 11% over FG-Attn (no scheduler), which statically assigns work to SMs and can leave some SMs idle when different query tiles attend to different numbers of key/value tiles. Third, the throughput gap between FG-Attn and FG-Attn (no scheduler) widens as skew increases. At higher skew, the variation in key/value lengths across query tiles becomes larger, making load imbalance more severe. The persistent work scheduler mitigates this imbalance by prioritizing tiles with larger estimated workloads, reducing tail effects, and improving SM utilization. From these experiments, we conclude that FG-Attn provides a highly efficient kernel for fine-grained sparse attention, that operates efficiently at different levels of skew.

6.2 Evaluating End-to-End Video Generation with Sparse Attention

Table 1 summarizes video generation time, attention FLOPs normalized to dense attention, and video quality relative to dense attention, respectively. We make the following observations.

First, FG-Attn speeds up video generation across all evaluated video models. Compared to the dense FA3 baseline, FG-Attn-16, FG-Attn-32, and FG-Attn-64 achieves speedups of up to 1.26×, 1.40×, and 1.40×,

Method	HunyuanVideo 480p, 129 Frames				HunyuanVideo 720p, 129 Frames			
	Time	Attn. FLOPs	PSNR	SSIM	Time	Attn. FLOPs	PSNR	SSIM
SVG	-	0.3X	20.20	0.79	-	0.3X	21.19	0.76
Sparge	-	-	< 20	0.79	-	-	< 20	< 0.7
Radial	-	0.37X	21.50	0.80	-	0.37X	22.33	0.77
FA3	3:06	1X	inf.	1	17:21	1X	inf.	1
FI+SVG2	3:00	**0.29X**	**23.63**	**0.83**	OOM	OOM	OOM	OOM
FG-Attn-16	2:55	0.31X	23.24	**0.83**	13:49	**0.27X**	**22.40**	**0.81**
FG-Attn-32	2:43	0.32X	22.80	0.83	12:26	0.28X	22.02	0.76
FG-Attn-64	**2:40**	0.34X	22.21	0.81	**12:26**	0.30X	22.21	0.74

Method	Wan 2.1 14B 480p, 81 Frames				Wan 2.1 14B 720p, 81 Frames			
	Time	Attn. FLOPs	PSNR	SSIM	Time	Attn. FLOPs	PSNR	SSIM
SVG	-	0.3X	25.59	0.84	-	0.3X	22.39	0.80
Sparge	-	-	24.11	0.79	-	-	23.51	0.78
Radial	-	0.37X	25.01	0.80	-	0.37X	24.31	0.78
FA3	4:50	1X	inf.	1	18:34	1X	inf.	1
FI+SVG2	5:25	**0.31X**	26.47	0.87	17:31	0.37X	25.36	**0.84**
FG-Attn-16	4:46	0.34X	**26.50**	**0.87**	17:10	**0.34X**	**25.39**	**0.84**
FG-Attn-32	4:26	0.37X	26.22	0.87	16:36	0.40X	24.02	0.80
FG-Attn-64	**4:20**	0.40X	25.82	0.86	**15:37**	0.45X	23.24	0.78

Method	LTX2 512x768, 121 Frames				LTX2 1024x1536, 121 Frames			
	Time	Attn. FLOPs	PSNR	SSIM	Time	Attn. FLOPs	PSNR	SSIM
FA3	0:36	1X	inf.	1.0	3:08	1X	inf.	1.0
FI+SVG2	1:00	**0.44X**	27.03	0.88	4:55	**0.43X**	25.36	0.88
FG-Attn-16	0:33	0.48X	**28.27**	**0.90**	2:51	0.48X	**28.73**	**0.93**
FG-Attn-32	**0:31**	0.52X	27.35	0.90	**2:45**	0.51X	28.29	0.92
FG-Attn-64	**0:31**	0.58X	26.77	0.89	**2:45**	0.59X	27.64	0.91

Table 1: Comparing video generation times, attention FLOPs, and generation quality.

respectively, with average speedups of 1.11×, 1.18×, and 1.19×. These speedups come from reducing the amount of attention computation. Second, fine-grained sparse attention exposes more opportunities to skip unnecessary attention computation. The FG-Attn-16 configuration has the lowest attention FLOPs on average, reducing attention FLOPs to 0.37× of dense attention, compared to 0.40× for FG-Attn-32 and 0.44× for FG-Attn-64. This is because smaller sparse blocks can more precisely remove irrelevant query–key regions, whereas coarser blocks must retain a larger region even when only part of the block is useful. At the same time, finer granularity preserves video quality better: compared to FI+SVG2, FG-Attn-16 improves PSNR by 0.86 on average. Third, compared to FI+SVG2, FG-Attn achieves higher end-to-end speedups while maintaining or improving video quality. FG-Attn-16, FG-Attn-32, and FG-Attn-64 is faster by up to 2.06×, 2.20×, and 2.20×, respectively, with average speedups of 1.42×, 1.48×, and 1.50×. Although FI+SVG2 sometimes reduces attention FLOPs more aggressively, it incurs substantial overhead from sparse-mask processing, query clustering/reordering, and FI planning. These overheads are especially visible for smaller-context models such as LTX-2, where FI+SVG2 is slower than dense FA3 despite reducing attention FLOPs to 0.44× of dense attention. In contrast, FG-Attn more effectively translates attention FLOP reduction into end-to-end generation speedup. Fourth, SpargeAttention, SVG, and Radial Attention rely on coarse block-sparse attention, which can lead to larger quality loss because useful query–key scores may be skipped together with unimportant scores in the same block. In contrast, FG-Attn enables better video quality using fine-grained masks that more precisely retain useful attention regions. Overall, FG-Attn improves video generation efficiency by combining fine-grained sparse attention with low-overhead sparse execution. Finer block sizes expose more sparsity and better preserve quality, while larger block sizes can provide higher end-to-end speedups due to lower mask computation overheads.

Ablation: Attention computation cost vs. block size. Fig. 9 shows the attention and mask computation time breakdown normalized to FA3. We observe that FG-Attn is faster than dense attention in all evaluated settings, with total attention time ranging from 0.37× to 0.94× of dense attention, showing

that FG-Attn can leverage sparsity to speed up attention. The mask computation takes a small amount of time on average (0.06× of FA3), showing that the sparse mask is cheap to compute. However, the mask overhead increases at finer block sizes, from 0.03× at 128×128 to 0.12× at 16×16. Merge-XOR outperforms consecutive merging because it reduces redundant key/value loads. Overall, FG-Attn reduces attention time while keeping mask overhead small.

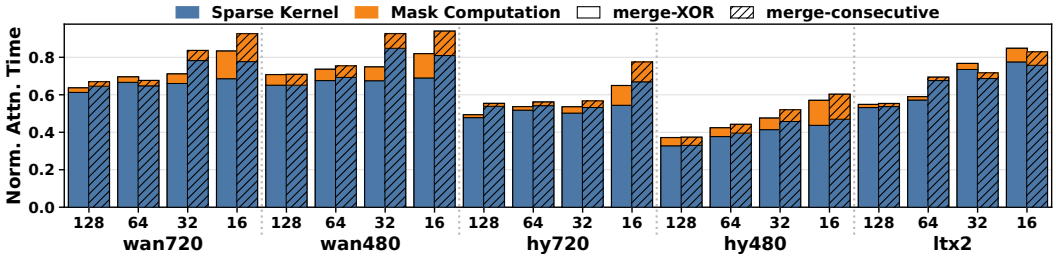


Figure 9: Attention time breakdown at different block sizes.

Ablation: Sparsity on merging query tiles. Fig. 10 shows the block sparsity before merging query tiles, and after merging to 128 queries using minimum XOR distance (FG-Attn +merge-XOR), and merging consecutive elements (FG-Attn +merge-consecutive). First, we observe that smaller blocks expose more sparsity before merging: the average sparsity increases from 0.57 at 128×128 to 0.68 at 16×16. Both merge-XOR and merge-consecutive decrease the effective sparsity. Third, merge-XOR preserves more sparsity than consecutive merging, especially at finer granularities. At 16×16, merge-XOR merging gives an average density of 0.47, compared to 0.55 for merge-consecutive. Overall, finer blocks expose more sparsity, but query merging can reduce the realized benefit. merge-XOR reduces this loss and makes fine-grained sparse attention more efficient.

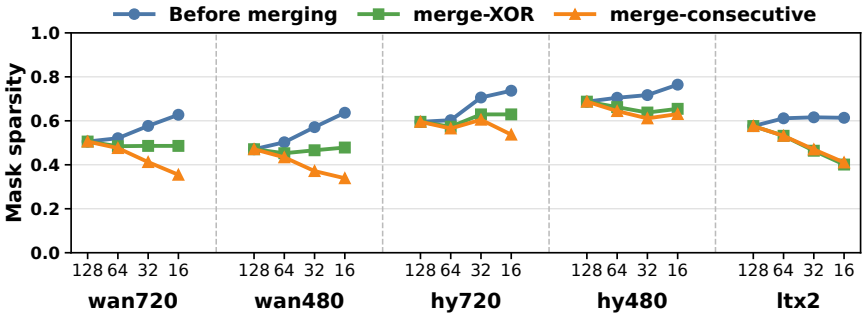


Figure 10: Sparsity at different block sizes.

7 Conclusion

We introduced **FG-Attn**, a fine-grained sparse attention mechanism that skips query–key computations at fine block granularities without incurring the overheads of redundant loads/computation and irregular memory access on modern GPUs. FG-Attn provides a practical training-free substrate to exploit fine-grain sparsity that can be seamlessly integrated into state-of-the-art video and image diffusion transformers. FG-Attn achieves significant end-to-end speedups with negligible quality loss, thus subsuming existing block-sparse attention mechanisms.

References

Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. In *The Twelfth International Conference on Learning Representations*, 2024. URL `https://openreview.net/forum?id=mZn2Xyh9Ec`.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

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*, 2(3):4, 2024.

Patrick Esser, Sumith Kulal, Andreas Blattmann, Rahim Entezari, Jonas Müller, Harry Saini, Yam Levi, Dominik Lorenz, Axel Sauer, Frederic Boesel, et al. Scaling rectified flow transformers for high-resolution image synthesis. In *Forty-first international conference on machine learning*, 2024.

Zhentao Fan, Zongzuo Wang, and Weiwei Zhang. Taocache: Structure-maintained video generation acceleration. *arXiv preprint arXiv:2508.08978*, 2025.

Junxian Guo, Haotian Tang, Shang Yang, Zhekai Zhang, Zhijian Liu, and Song Han. Block Sparse Attention. `https://github.com/mit-han-lab/Block-Sparse-Attention`, 2024.

Yoav HaCohen, Benny Brazowski, Nisan Chiprut, Yaki Bitterman, Andrew Kvochko, Avishai Berkowitz, Daniel Shalem, Daphna Lifschitz, Dudu Moshe, Eitan Porat, et al. Ltx-2: Efficient joint audio-visual foundation model. *arXiv preprint arXiv:2601.03233*, 2026.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. *arXiv preprint arXiv:1904.09751*, 2019.

Huiqiang Jiang, Yucheng Li, Chengruidong Zhang, Qianhui Wu, Xufang Luo, Surin Ahn, Zhenhua Han, Amir H 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.

Weijie Kong, Qi Tian, Zijian Zhang, Rox Min, Zuozhuo Dai, Jin Zhou, Jiangfeng Xiong, Xin Li, Bo Wu, Jianwei Zhang, et al. Hunyuanvideo: A systematic framework for large video generative models. *arXiv preprint arXiv:2412.03603*, 2024.

Zhifeng Kong, Wei Ping, Jiaji Huang, Kexin Zhao, and Bryan Catanzaro. Diffwave: A versatile diffusion model for audio synthesis. *arXiv preprint arXiv:2009.09761*, 2020.

Black Forest Labs. Flux. `https://github.com/black-forest-labs/flux`, 2024.

Black Forest Labs, Stephen Batifol, Andreas Blattmann, Frederic Boesel, Saksham Consul, Cyril Diagne, Tim Dockhorn, Jack English, Zion English, Patrick Esser, Sumith Kulal, Kyle Lacey, Yam Levi, Cheng Li, Dominik Lorenz, Jonas Müller, Dustin Podell, Robin Rombach, Harry Saini, Axel Sauer, and Luke Smith. Flux.1 kontext: Flow matching for in-context image generation and editing in latent space, 2025. URL `https://arxiv.org/abs/2506.15742`.

Xingyang Li, Muyang Li, Tianle Cai, Haocheng Xi, Shuo Yang, Yujun Lin, Lvmin Zhang, Songlin Yang, Jinbo Hu, Kelly Peng, Maneesh Agrawala, Ion Stoica, Kurt Keutzer, and Song Han. Radial attention: $\mathcal{O}(n\log n)$ sparse attention with energy decay for long video generation. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2026. URL `https://openreview.net/forum?id=hYovE4nHTt`.

Feng Liu, Shiwei Zhang, Xiaofeng Wang, Yujie Wei, Haonan Qiu, Yuzhong Zhao, Yingya Zhang, Qixiang Ye, and Fang Wan. Timestep embedding tells: It's time to cache for video diffusion model. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pp. 7353–7363, 2025.

Jiayi Luo, Jiayu Chen, Jiankun Wang, Cong Wang, Hanxin Zhu, Qingyun Sun, Chen Gao, Zhibo Chen, and Jianxin Li. Training-free sparse attention for fast video generation via offline layer-wise sparsity profiling and online bidirectional co-clustering. *arXiv preprint arXiv:2603.18636*, 2026.

Xinyin Ma, Gongfan Fang, and Xinchao Wang. Deepcache: Accelerating diffusion models for free. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 15762–15772, 2024.

William Peebles and Saining Xie. Scalable diffusion models with transformers. In *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 4195–4205, 2023.

Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *Advances in Neural Information Processing Systems*, 37:68658–68685, 2024.

Apoorv Vyas, Angelos Katharopoulos, and François Fleuret. Fast transformers with clustered attention. *Advances in Neural Information Processing Systems*, 33:21665–21674, 2020.

Team Wan, Ang Wang, Baole Ai, Bin Wen, Chaojie Mao, Chen-Wei Xie, Di Chen, Feiwu Yu, Haiming Zhao, Jianxiao Yang, et al. Wan: Open and advanced large-scale video generative models. *arXiv preprint arXiv:2503.20314*, 2025.

Guoxia Wang, Jinle Zeng, Xiyuan Xiao, Siming Wu, Jiabin Yang, Lujing Zheng, Zeyu Chen, Jiang Bian, Dianhai Yu, and Haifeng Wang. Flashmask: Efficient and rich mask extension of flashattention. In *The Thirteenth International Conference on Learning Representations*, 2025. URL `https://openreview.net/forum?id=wUtXB43Chi`.

Jianzong Wu, Liang Hou, Haotian Yang, Ye Tian, Pengfei Wan, Di ZHANG, and Yunhai Tong. VMoBA: Mixture-of-block attention for video diffusion models. In *The Fourteenth International Conference on Learning Representations*, 2026. URL `https://openreview.net/forum?id=oQaRElUdmh`.

Haocheng Xi, Shuo Yang, Yilong Zhao, Chenfeng Xu, Muyang Li, Xiuyu Li, Yujun Lin, Han Cai, Jintao Zhang, Dacheng Li, Jianfei Chen, Ion Stoica, Kurt Keutzer, and Song Han. Sparse video-gen: Accelerating video diffusion transformers with spatial-temporal sparsity. In *Forty-second International Conference on Machine Learning*, 2025. URL `https://openreview.net/forum?id=u8CA3qIS0V`.

Jianfeng Xiang, Zelong Lv, Sicheng Xu, Yu Deng, Ruicheng Wang, Bowen Zhang, Dong Chen, Xin Tong, and Jiaolong Yang. Structured 3d latents for scalable and versatile 3d generation. *arXiv preprint arXiv:2412.01506*, 2024.

Jianfeng Xiang, Xiaoxue Chen, Sicheng Xu, Ruicheng Wang, Zelong Lv, Yu Deng, Hongyuan Zhu, Yue Dong, Hao Zhao, Nicholas Jing Yuan, and Jiaolong Yang. Native and compact structured latents for 3d generation. *Tech report*, 2025.

Ruyi Xu, Guangxuan Xiao, Haofeng Huang, Junxian Guo, and Song Han. Xattention: Block sparse attention with antidiagonal scoring. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, 2025.

Shuo Yang, Haocheng Xi, Yilong Zhao, Muyang Li, Jintao Zhang, Han Cai, Yujun Lin, Xiuyu Li, Chenfeng Xu, Kelly Peng, Jianfei Chen, Song Han, Kurt Keutzer, and Ion Stoica. Sparse videogen2: Accelerate video generation with sparse attention via semantic-aware permutation. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2026. URL `https://openreview.net/forum?id=WPU17d1l7R`.

Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. Flashinfer: Efficient and customizable attention engine for llm inference serving. *arXiv preprint arXiv:2501.01005*, 2025. URL `https://arxiv.org/abs/2501.01005`.

Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, Yuxing Wei, Lean Wang, Zhiping Xiao, et al. Native sparse attention: Hardware-aligned and natively trainable sparse attention. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 23078–23097, 2025.

Jintao Zhang, Haofeng Huang, Pengle Zhang, Jia wei, Jun Zhu, and Jianfei Chen. Sageattention2: Efficient attention with thorough outlier smoothing and per-thread INT4 quantization. In *Forty-second International Conference on Machine Learning*, 2025a. URL https://openreview.net/forum?id=nC8XliUxeg.

Jintao Zhang, Jia Wei, Pengle Zhang, Xiaoming Xu, Haofeng Huang, Haoxu Wang, Kai Jiang, Jianfei Chen, and Jun Zhu. Sageattention3: Microscaling fp4 attention for inference and an exploration of 8-bit training. *arXiv preprint arXiv:2505.11594*, 2025b.

Jintao Zhang, Jia wei, Pengle Zhang, Jun Zhu, and Jianfei Chen. Sageattention: Accurate 8-bit attention for plug-and-play inference acceleration. In *The Thirteenth International Conference on Learning Representations*, 2025c. URL https://openreview.net/forum?id=OL44KtasKc.

Jintao Zhang, Chendong Xiang, Haofeng Huang, Jia Wei, Haocheng Xi, Jun Zhu, and Jianfei Chen. Spargeattn: Accurate sparse attention accelerating any model inference. In *International Conference on Machine Learning (ICML)*, 2025d.

Jintao Zhang, Xiaoming Xu, Jia Wei, Haofeng Huang, Pengle Zhang, Chendong Xiang, Jun Zhu, and Jianfei Chen. Sageattention2++: A more efficient implementation of sageattention2. *arXiv preprint arXiv:2505.21136*, 2025e.

Jintao Zhang, Kai Jiang, Chendong Xiang, Weiqi Feng, Yuezhou Hu, Haocheng Xi, Jianfei Chen, and Jun Zhu. Spargeattention2: Trainable sparse attention via hybrid top-k+ top-p masking and distillation fine-tuning. *arXiv preprint arXiv:2602.13515*, 2026a.

Peiyuan Zhang, Yongqi Chen, Runlong Su, Hangliang Ding, Ion Stoica, Zhengzhong Liu, and Hao Zhang. Fast video generation with sliding tile attention. In *Forty-second International Conference on Machine Learning*, 2025f. URL https://openreview.net/forum?id=U74MOXPEJd.

Peiyuan Zhang, Yongqi Chen, Haofeng Huang, Will Lin, Zhengzhong Liu, Ion Stoica, Eric P. Xing, and Hao Zhang. Faster video diffusion with trainable sparse attention. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2026b. URL https://openreview.net/forum?id=VrYCLQ5inI.

A GPU Architecture Overview

Compute hierarchy. Fig. 11 provides a high-level overview of modern GPU architecture. A GPU consists of many *streaming multiprocessors* (SMs). Each SM contains its own register file, on-chip shared memory (or scratchpad memory), and a set of *tensor cores* for matrix multiplication. Each SM executes many *threads*. Threads are grouped at three levels: 32 threads form a *warp* that executes the same instruction in lockstep. A fixed number of warps form a *thread block* that is scheduled onto a single SM. On Hopper GPUs, four warps (128 threads) form a *warp-group*, the unit at which a tensor core matrix multiplication instruction is issued.

A *kernel* is a function that the programmer launches with a grid of thread blocks; each block runs on one SM. The programmer decides how many threads per block, how much shared memory each block uses, and how work is partitioned across blocks.

Memory hierarchy. GPU memory consists of broadly three types: *High-bandwidth memory* (HBM) is the off-chip RAM (e.g., 80 GB on an H100) that holds the model's weights, activations, and KV cache; it has the highest capacity but also the highest latency, on the order of hundreds of cycles. Each SM has *shared memory* (typically tens to a few hundred KB), an on-chip scratchpad that all threads in a block can access in a few cycles. Each thread additionally has private *registers*, the fastest storage but limited in number. Achieving high performance on a GPU is largely about (a) keeping data resident in shared memory and registers, and (b) overlapping the unavoidable HBM transfers with compute.

Tensor cores. Tensor cores are specialized matrix-multiply units that operate on small tiles (e.g., $64 \times 64 \times 16$ on Hopper) at very high throughput. To use them, a kernel must (i) stage operand tiles into shared memory or registers, (ii) issue a tensor-core instruction (`wgmma` on Hopper), and (iii) wait for the result. A modern attention kernel spends most of its time issuing tensor-core matmuls for $QK^\top$ and the subsequent multiplication by values.

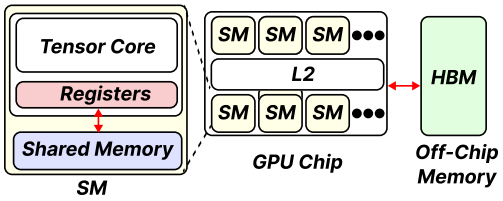


Figure 11: High-level view of modern GPU architecture.

Warp specialization and producer–consumer pipelining. Because HBM latency is so much larger than tensor-core throughput, naively loading a tile and then computing on it leaves the tensor cores idle for most of every iteration. Modern attention kernels hide this latency through *warp specialization*: within a thread block, some threads are designated *producers* and others *consumers*, as shown in Fig. 12. Producer threads issue asynchronous loads (using `cp.async` or, on Hopper, the Tensor Memory Accelerator) that copy tiles of K and V from global to shared memory. Consumer warp-groups wait on a barrier until a tile is ready, multiply them with a group of queries using the tensor core matrix multiply operation. Upon completion, they signal the producer threads to load the next key-value pair into shared memory. With multiple buffer slots in shared memory, the producer can run several iterations ahead of the consumer, so that by the time the consumer finishes one matmul, the next tile is already resident on-chip. This is the software pipeline depicted in Fig. 12. Producer threads issue load operations to move data from HBM into shared memory, while consumer threads schedule computations on the tensor cores using the fetched data. By overlapping these operations, GPU resources remain efficiently utilized.

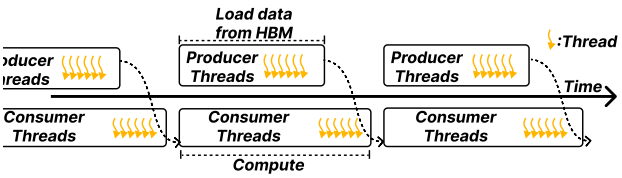


Figure 12: Pipelined execution of producer and consumer threads: Data is prefetched by the producer while the consumer threads are doing computation.