ALISA: Accelerating Large Language Model Inference via Sparsity-Aware KV Caching

Youpeng Zhao, Di Wu, Jun Wang
University of Central Florida
Email: {youpeng.zhao, di.wu, jun.wang}@ucf.edu

***Abstract*—The Transformer architecture has significantly advanced natural language processing (NLP) and has been foundational in developing large language models (LLMs) such as LLaMA and OPT, which have come to dominate a broad range of NLP tasks. Despite their superior accuracy, LLMs present unique challenges in practical inference, concerning the compute and memory-intensive nature. Thanks to the autoregressive characteristic of LLM inference, KV caching for the attention layers in Transformers can effectively accelerate LLM inference by substituting quadratic-complexity computation with linear-complexity memory accesses. Yet, this approach requires increasing memory as demand grows for processing longer sequences. The overhead leads to reduced throughput due to I/O bottlenecks and even out-of-memory errors, particularly on resource-constrained systems like a single commodity GPU.**

In this paper, we propose ALISA, a novel algorithm-system co-design solution to address the challenges imposed by KV caching. On the algorithm level, ALISA prioritizes tokens that are most important in generating a new token via a Sparse Window Attention (SWA) algorithm. SWA introduces high sparsity in attention layers and reduces the memory footprint of KV caching at negligible accuracy loss. On the system level, ALISA employs three-phase token-level dynamical scheduling and optimizes the trade-off between caching and recomputation, thus maximizing the overall performance in resource-constrained systems. In a single GPU-CPU system, we demonstrate that under varying workloads, ALISA improves the throughput of baseline systems such as FlexGen and vLLM by up to $3\times$ and $1.9\times$, respectively.

I. Introduction

Large Language Models (LLMs) stand as a revolutionary breakthrough in the modern era of artificial intelligence (AI). Distinct from previous small language models with only millions of parameters, LLMs often have hundreds of billions or even trillions of parameters. They have exhibited exceptional abilities in solving complex tasks, such as semantic reasoning and creative writing through text generation. GPT-2 XL, one of the earliest LLMs with 1.5 billion parameters, pioneered in showcasing these capabilities [29]. Its successor, GPT-3, showcases even more powerful abilities with 175 billion parameters [6]. To date, the most noteworthy application of LLMs is ChatGPT from OpenAI [26], a tool that allows users to interact with an AI agent in a conversational way to solve tasks ranging from language translation to software engineering, and beyond.

LLMs usually consist of stacked transformer decoder layers, in which the critical component is self-attention (attention for short in this work) [35]. The attention modules empower LLMs to capture contextual information by attending to different positions within the sequences, which however introduces quadratic computation complexity with the sequence length. Such complexity severely bottlenecks the performance and scalability of LLMs and is becoming more pronounced upon the pursuit for longer sequences in existing systems [3, 8, 16, 28]. One viable solution to this problem during LLM inference is KV caching [27]. This idea originates from the fact that LLM inference is *autoregressive*, where LLMs generate new tokens sequentially based on all prior tokens (more details in Figure 2). This characteristic opens up the opportunity of reusing intermediate states, specifically, the key (K) and value (V) tensors, whose sizes increase linearly with the sequence length through caching in attention layers. With KV caching, the quadratic-complexity computation is reduced to linear-complexity computation and memory accesses, therefore substantially speeding up LLM inference.

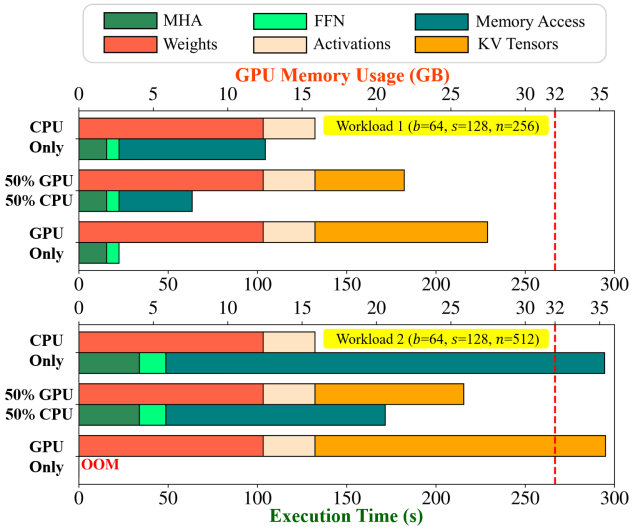


Fig. 1: Breakdown of execution time and memory usage for OPT-6.7B inference on one NVIDIA Tesla V100 GPU with 32 GB memory under different workloads. Weights, activations, and KV tensors (intermediate key and value states) denote the required GPU memory. MHA, FFN, and memory access denote the time for computing multi-headed attention, and feed-forward network (both including the follow-up Addition and LayerNorm operations) and KV caching (moving KV tensors between CPU and GPU if any). 50% means the ratio of the KV tensors allocated to CPU/GPU memory. OOM denotes out-of-memory error, and the red-dot line denotes the GPU memory capacity. The b, s, and n for workloads refer to the batch size, and input and output sequence length. Results are reported using FlexGen [31].

Challenge. Despite KV caching significantly reducing the inference time, *LLM inference with KV caching is predominantly bottlenecked by memory [28], especially in resource-constrained systems*, like a single commodity GPU. First, the most expensive operations in LLMs are matrix multiplication and softmax, which are notoriously memory-bound. Second, the weights and activations in LLMs have already raised an alert on the memory capacity. Third, intermediate KV tensors further exacerbate the requirement on memory capacity, which is determined by the sequence length and model dimension. For a given LLM, as the batch size and sequence length increase, the allocated memory for KV caching continues to grow linearly, and at some point, exceeds the available memory capacity. Ultimately, pursuing longer sequences in larger LLMs ends up with an out-of-memory error and halts the execution, as given by the "GPU only" case on workload 2 in Figure 1. To circumvent the out-of-memory error in a single GPU setting, researchers have developed solutions to offload KV tensors to CPU memory or even secondary storage to free up GPU resources in real-world scenarios [31]. However, frequent offloading and reloading of KV tensors incur significant data transfer overhead, which becomes the new bottleneck towards high throughput and low latency in resource-constrained systems, as seen in Figure 1. To this end, we ask: *how to innovate KV caching for LLMs in resource-constrained systems, to facilitate better scalability and meet the need of longer sequences and larger model sizes.*

Proposal. In this paper, we propose ALISA, an algorithm-system co-design solution to accelerate LLM inference via sparsity-aware KV caching for single GPU-CPU systems. Our key observation is that *during the autoregressive inference process, the attention weight matrix is highly sparse, and larger LLMs exhibit higher attention weight sparsity.* This observation validates the intuition that not all tokens are created equal and only a small number of important tokens contribute to generating a new token. Once these important tokens are identified, we can selectively access the KV tensors corresponding to these important tokens, and skip unimportant ones. To identify which tokens are important, we formulate a Sparse Window Attention (SWA) algorithm, in which both the globally dynamic and locally static sparse patterns are created. A mixture of these sparse patterns can significantly reduce the memory footprint while maintaining model accuracy due to the ability to better capture important tokens in a sequence.

However, as the size of LLMs keeps growing, the above algorithmic optimization is insufficient to guarantee satisfactory performance, i.e., throughput in this work, for resource-constrained systems. We argue that *accelerating LLMs is not only a computation problem but more of a memory problem, in the presence of a gigantic memory footprint.* Three bottlenecks are responsible. Firstly, the size of sparse KV tensors will ultimately exceed the memory capacity with longer sequences, and the long-latency GPU-CPU memory accesses in dense LLMs recur. Secondly, the sparse nature of KV tensors induces unpredictable memory access, which is exacerbated by longer sequences. To address these two challenges, we propose to dynamically schedule the KV tensors at the token level and balance between caching and recomputation for best performance gain. We highlight this token-level scheduling in Table I. Thirdly, high-precision (FP16 in this work) KV tensors still exhibit a large memory footprint, thus high memory access latency. We can compress KV tensors to lower precision (INT8) via quantization and further reduce the overall memory overhead, without sacrificing the accuracy.

TABLE I: Comparison of prior works and our ALISA. Block means a fixed group of tokens. Head means a single attention module.

Design	vLLM [21]	FlexGen [31]	ALISA (Ours)
Sparse Attn.	✗	✗	✓
Caching Granularity	Block-level (Static)	Head-level (Static)	Token-level (Dynamic)
Recomputation	✓	✗	✓
Scenario	Online (Multi-GPU)	Offline (Single-GPU)	Offline (Single-GPU)
Co-Design	✗	✗	✓

In summary, this paper makes the following contributions:

- We identify the challenges in KV caching for LLM inference and propose an algorithm-system co-design solution, ALISA, for efficient LLM inference in resource-constrained systems.
- On the algorithm level, we propose sparse window attention (SWA) that creates a mixture of globally dynamic and locally static sparse patterns in KV tensors to reduce the memory footprint while maintaining high accuracy.
- On the system level, we design a three-phase scheduler to dynamically allocate KV tensors between GPU and CPU memory to reduce data transfer at the token level.
- We evaluate ALISA over a wide range of LLM models, tasks, and workloads. Experiments demonstrate that ALISA can significantly reduce the memory footprint of KV tensors and increase the throughput over previous baselines, with negligible accuracy drop.

The remainder of this paper is organized as follows. Section II recaps LLM-related concepts and works. Then, Section III articulates our perspectives on challenges and corresponding opportunities in accelerating LLMs. Next, Section IV and Section V elaborate the algorithm and system design in ALISA, with evaluation followed in Section VI. Finally, Section VII concludes this work.

II. Background

In this section, we first present some preliminary knowledge of LLMs, including autoregressive inference, the Transformer layer, and KV caching. Afterwards, we discuss related works.

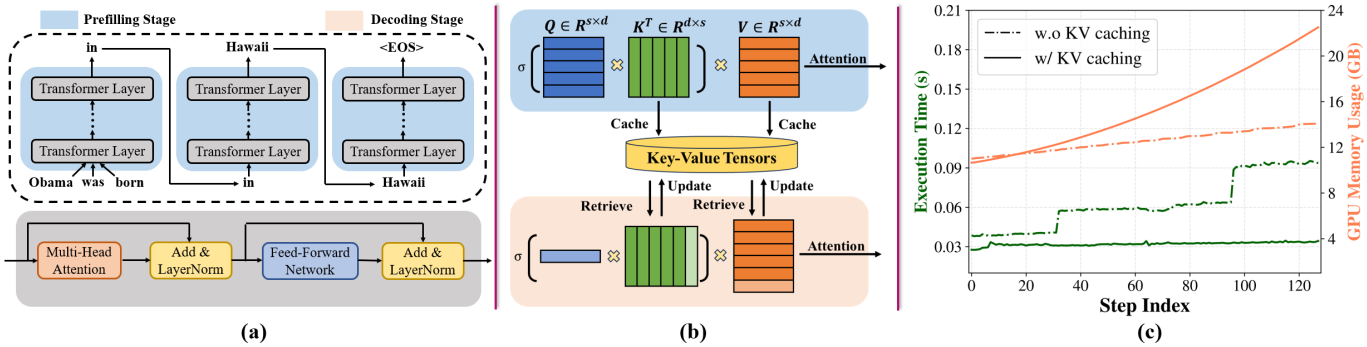


Fig. 2: (a) Top: an example of autoregressive LLM inference. EOS refers to end-of-sentence. Bottom: operation blocks in transformer layers. (b) KV caching: Q, K, V denotes the query, key, and value tensors. At the prefilling stage, all input tokens are processed simultaneously, and the generated intermediate KV tensors are stored, marked by dark colors. s and d represent the input sequence length and the hidden dimension size of KV tensors. At the decoding stage, the stored KV tensors in the dark colors are retrieved. The input Q, K, V tensors are marked by the light colors. The input Q tensor is multiplied with a concatenation of input K and stored K tensors, followed by a softmax of the entire attention weights. The attention weight are further multiplied with a concatenation of input V and stored V tensors to generate new results. Afterward, the input K and V tensors are stored. This process is repeated per token. (c) Execution time and GPU memory usage for OPT-6.7B inference with and without KV caching. The x-axis step index means the output sequence length. Results are reported using HuggingFace Accelerate [39].

A. Large Language Models

Autoregressive Inference. Transformer-based language models function by processing a sequence of input words and generating new, related words as output. Compared with previous small language models in the pre-LLM era, the most distinctive characteristic of LLMs is that LLM inference is *autoregressive*, i.e., output tokens solely depend on past tokens. Figure 2 (a) gives an example of such an autoregressive behavior in LLM inference on the top. The inference process can be divided into two parts, including the prefilling and decoding stages. During the prefilling stage, LLMs process all the input tokens in a single pass. Then, during the decoding stage, a previously generated output token is fed back into the model as an input and generates the next output token. Therefore, the decoding stage unfolds iteratively, processing one token at a time. When the sequence length reaches a maximum threshold (specified by users or service providers) or an "⟨EOS⟩" token is emitted, the decoding process stops. Inside the LLMs, the input words are first tokenized to continuous vectors using an embedding layer (not shown for simplicity) and then go through stacked transformer layers. All transformer layers are identical and include a multi-head attention (MHA) layer and a feed-forward network (FFN) layer, as shown at the bottom in Figure 2 (a). There exist addition and layer normalization layers after MHA and FFN layers. We consider them as part of the MHA and FFN layers in this work during evaluation. Finally, the outputs of transformer layers will go through a linear projection and a softmax layer to generate the corresponding token for the next word (not shown).

Transformer Layer. At the core of transformer layers lies the attention module [35]. The relevant operations are given in Equation 1 and 2. There are three intermediate tensors involved, namely, query Q, key K, and value V. The attention weights $AW(Q, K)$ are calculated by first computing the dot product between Q and K, then scaling the product by the square root of hidden dimension d, and finally going through a softmax operation ($\sigma(\cdot)$). The attention scores $Attn(Q, K, V)$ are calculated by multiplying the attention weights $AW(Q, K)$ to V. The MHA output is obtained by simply concatenating the outputs of all attention heads along the head dimension, with each head being an attention module.

$$AW(Q, K) = \sigma(\frac{QK^T}{\sqrt{d}}) \quad (1)$$

$$Attn(Q, K, V) = AW(Q, K) \cdot V \quad (2)$$

KV Caching. According to Equation 1, the attention operation induces quadratic computation complexity with respect to the sequence length. An example is given at the top of Figure 2 (b). The sequence length quadratically increases the size of the attention weight matrix, therefore quadratically increasing execution time. This overhead is exacerbated when pursuing longer sequences for larger models [3, 8, 16, 28]. To mitigate such a quadratic overhead for LLM inference, KV Caching is proposed to store the intermediate tensors such as key (K) and value (V) tensors in attention layers for computation reuse in future decoding steps [27]. The bottom of Figure 2 (b) showcases how KV caching works. KV caching transforms the original matrix multiplication with quadratic complexity into vector-matrix multiplication and memory accesses with linear complexity, thus significantly improving the performance. Figure 2 (c) draws the execution time and memory usage for LLM inference with and without KV caching. Without KV caching, the execution time increases rapidly, due to calculating the entire attention repeatedly. With KV caching, only the attention weights and scores for the newly generated token are calculated as vector-matrix multiplication, and the execution time stays almost constant across different steps.

However, such runtime reduction is at the expense of GPU memory usage, which increases gradually over time, due to the growing size of KV tensors.

B. Related Work

Algorithmic Optimization for Attention. On the algorithm side, various optimizations have been proposed to address the quadratic complexity of attention modules. In the pre-LLM era, algorithm optimizations largely focus on reducing the attention complexity through approximation methods. For example, Linformer [37] and Reformer [20] approximate the original attention using low-rank matrices and locality-sensitive hashing, respectively, achieving almost linear complexity. However, these approximations are not able to offer competitive accuracy in LLMs. Another line of algorithmic optimization is to create sparse patterns in attention modules [3, 8, 10, 32]. However, most sparsity-driven methods require additional training, which is not scalable for LLMs and cannot guarantee accuracy performance [10, 32]. In the LLM era, Longformer [3] constructs the sparse attention using a fixed-size sliding window on the most recent local tokens. SparseTransformer [8] generates sparse patterns with a fixed stride on all tokens. However, these sparse attention methods are not able to capture important tokens during the autoregressive LLM inference, resulting in accuracy collapse.

Hardware Acceleration for Attention. For small language models, accelerators that utilize algorithm-hardware co-design have been proposed [13, 36]. For example, SpAtten co-designs the algorithm and accelerator architecture to improve the sparsity in attention modules and reduce both the compute and memory overheads in matrix multiplication operations [36]; ViTALiTy approximates the dot-product softmax operation in attention modules using first-order Taylor expansion and linearizes the relevant cost [13]. Though these accelerators are quite effective in the pre-LLM era, their merit fades away in LLM inference, due to their fundamental limitations. First, pre-LLM accelerators simply can not handle the large model size of LLMs. For example, SpAtten balances its design choices among algorithm complexity, computation throughput, and memory capacity for the BERT [15], GPT-2 small and medium model [29]. However, the largest GPT-2 medium model has only 0.36 billion parameters, not even a fraction of that for LLMs, e.g., 175 billion parameters for GPT-3 [6], which engages 652 GB for single-precision model weights. Naively slabbing large memory onto the computing kernels does not offer Pareto efficiency. Second, prior co-designed accelerators are not able to further scale up with longer sequences. For example, SpAtten requires storing the entire attention weights to prune away unwanted tokens. However, the size of the attention weight matrix increases quadratically with sequence length. In the era of LLMs, given limited memory capacity, especially in resource-constrained systems, squeezing memory from KV tensors to attention weights will certainly degrade the efficacy of KV caching and slow down the inference.

KV Caching Optimization. In the LLM era, numerous specialized LLM systems have been developed. We compare some of these systems in Table I. For example, FlashAttention aims to reduce memory accesses between on-chip SRAM and off-chip HBM in GPUs for higher throughput with fine-grained tiling and partitioning at the kernel level [11, 12]. However, it does not optimize the memory accesses between CPUs and GPUs. vLLM proposes storing intermediate KV tensors at the block level, where each block contains a fixed number of tokens and is stored in non-contiguous paged memory to alleviate memory fragmentation for online LLM inference [21]. Identical to this work, FlexGen also targets resource-constrained systems [31]. FlexGen formulates a static offloading strategy for KV tensors throughout the LLM inference and manages them at the head level. H_2O designs a KV caching policy by retaining heavy hitters (H_2) tokens, which are determined by the global attention weight sum [43], rather than the local attention weight sum in ALISA. To summarize, three features differentiate ALISA from prior works. First, ALISA co-designs both the algorithm and system to fully exploit sparse attention for higher throughput, while previous works focus solely on either algorithm improvement (H_2O) or system improvement (vLLM, FlexGen). Second, ALISA performs KV caching at the granularity of one token, allowing flexible KV tensor allocation, which is critical upon sparsity-driven co-design. Third, ALISA adopts an appropriate dynamic scheduler to perform both caching and recomputation, while previous works only employ static KV caching [31, 43].

III. Challenges and Opportunities

A. Challenges

Despite KV caching has significantly improved the end-to-end performance for LLMs by avoiding quadratic-complexity computation, it still introduces a linear-complexity memory footprint. During LLM inference, we have to allocate GPU memory to store intermediate KV tensors. The corresponding memory footprint can vary from hundreds of megabytes to hundreds of gigabytes, depending on batch size, sequence length, and model configuration. As shown in Figure 2 (c), the GPU memory usage with KV caching is about 60% higher than that without KV caching with only 128 tokens in the sequence. In half-precision data format, running OPT-13B with a sequence length of 512 at a batch size of 64 imposes more than 25 GB of memory for KV tensors, which is even larger than the model weight size (about 23 GB). When the sequence length increases, this gap will be further widened.

In resource-constrained systems (e.g., a single GPU with limited memory), KV tensors ought to be offloaded to next-level memory hierarchies, such as CPU memory or even secondary storage, when the size of KV tensors exceeds the capacity of the GPU memory. However, offloading and reloading KV tensors incur significant data transfer overhead (e.g., I/O access between GPU and CPU memory on the PCIe bus), as shown in Figure 1. Storing 50% KV tensors in CPU memory will increase the overall execution time of LLM inference by $3\times$; and this slowdown reaches $5\times$ if storing

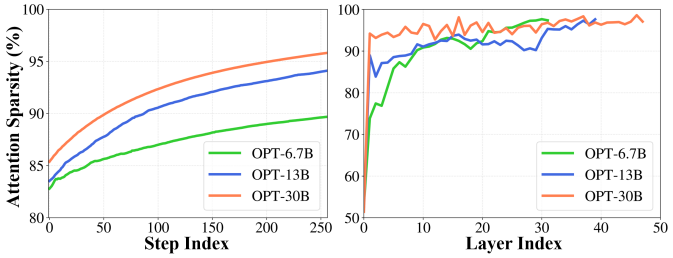


Fig. 3: Attention weight sparsity observed across different steps and layers during OPT model inference using the Wiki-Text-2 dataset [24]. We consider elements as zeros if they fall below 1% of the row-wise maximum value.

all KV tensors in CPU memory. Given this bottleneck in KV caching, we need to find *a solution that orchestrates when, how, and what to offload and reload in resource-constrained systems, so that the overall execution time is minimized.*

B. Opportunities

Let's start with a simple example. Given the question "What is the capital of France?", we humans only need to pay attention to 'capital' and 'France' to respond with the answer "Paris." The intuition is that not all words (tokens) are created equal, and some are more important than others. This intuition has been leveraged in accelerating transformers in the pre-LLM era. Prior works for small language models empirically keep the tokens that lead to large attention weights, and prune away those with smaller weights [36]. In this work, we take one more step to corroborate this intuition in LLMs by profiling the sparsity in attention weights, as shown in Figure 3. We have two key observations. First, the attention weights in LLMs are highly sparse, e.g., the sparsity can vary between 80% and 95% across different inference steps, and reach close to 99% in some layers. Second, larger LLMs exhibit higher sparsity, e.g., the density (i.e., $1 -$ sparsity) of OPT-30B is about $3\times$ less than that of OPT-6.7B. These observations translate to the fact that, from a computation perspective, very few elements in the attention weight matrix contribute to calculating the final attention score and generating new tokens in LLMs. This both motivates and validates our solution to create sparse KV tensors by skipping unimportant tokens in LLM inference.

C. Objective

To make the most of the high sparsity in attention weights, we propose to co-design LLMs in resource-constrained systems from both the algorithm and system sides. Three technical questions need to be answered.

Identifying Important Tokens. In the context of LLMs, individual tokens have varying importance. During the inference process, the attention weights for each token vary from step to step. The nondeterministic nature of language makes it extremely hard to predict which tokens are important. Hence we need a low-cost mechanism to distinguish important tokens without hurting accuracy significantly for LLM inference.

Caching KV Tensors. When KV tensors become too large for GPU memory, we have to store partial KV tensors in CPU memory for future reuse. Theoretically, we could use Belady's Algorithm as the caching policy [2], which evicts the tokens that will not be used for the longest period in the future. However, this oracle algorithm assumes future knowledge and imposes a huge amount of resources, making it impractical in LLM inference. Therefore there is a need to develop a low-cost caching policy to allocate sparse KV tensors and ensure a relatively low miss rate.

Caching vs. Recomputation. As the sequence length grows, the benefit of KV caching diminishes at a certain threshold since the time for accessing CPU memory might outweigh that for recomputing partial KV tensors. Moreover, this sequence length threshold varies across batch sizes and model configurations. Thus, we must design a dynamic scheduling strategy that balances KV caching and recomputation at the token level.

IV. ALISA Algorithm Design

A. Attention Analysis

In the LLM era, existing works mainly aim to create sparsity in attention weights during LLM inference [3, 8]. Longformer [3] adopts a local attention mechanism, which applies a fixed-size sliding window on the KV tensors corresponding to the most recent tokens. The resultant attention weight pattern is shown on the top of Figure 4 (b). SparseTransformer applies a strided mask on the tokens and creates strided attention [8], as shown on the top of Figure 4 (c).

To understand why the previous attention methods fail upon long sequences, we visualize the dense attention weight maps during LLM inference in Figure 5. We observe that attention weights with larger values do not exhibit a specific pattern. Only using the most recent tokens cannot accurately represent the distribution of the entire attention weights, since the tokens with large attention weights (therefore more important) are often far from the current token. A similar problem exists in strided attention, and the stride mask might not always capture large attention weights. Therefore, the attention weight maps of the local attention and the strided attention can not capture a large portion of attention weights. Subsequently, the corresponding attention score distributions significantly drift away from what is expected in dense attention. At the bottom of Figure 4 (a)-(c), we see that dense attention scores follow a near power-law distribution, which is consistent with previous findings [7, 37]. However, the attention score distributions generated by local and strided attention show close to zero correlation to that of dense attention, thus resulting in drastically lower accuracy.

B. Sparse Window Attention (SWA)

To maintain model accuracy, we propose a novel Sparse Window Attention (SWA) method, which produces both *locally static* and *globally dynamic* sparse patterns. We generate static patterns at local tokens by keeping the most recent

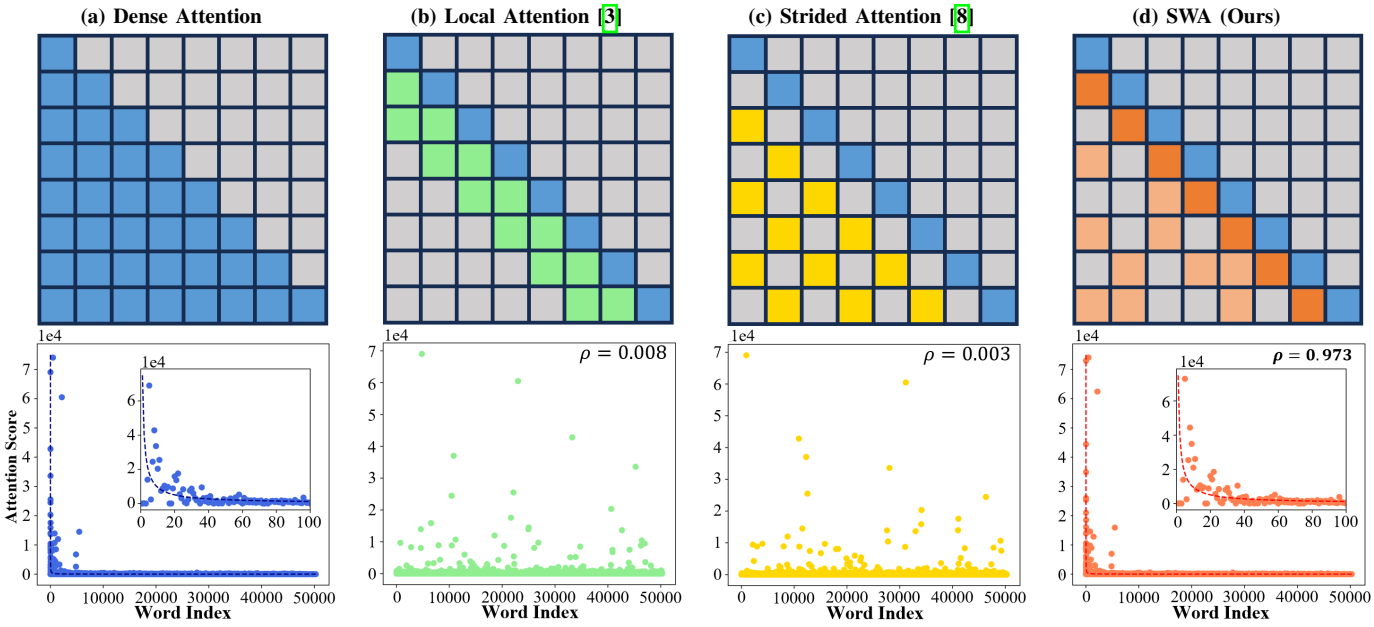


Fig. 4: Comparisons of our proposed Sparse Window Attention (SWA) and existing methods. On the top are illustrative sparse patterns for attention weight matrices generated by each method, where the x-axis the positions in the input sequence that are being attended to, and the y-axis represents the positions in the output sequence. The same notation is used in Figure 5. Grey blocks mean the values are masked with zeros, due to the autoregressive LLM inference. On the bottom are the corresponding average attention score distributions in the Wiki-Text-2 dataset vocabulary for the OPT-6.7B model. ρ is the Spearman correlation score between sparse attention and dense attention (higher is better).

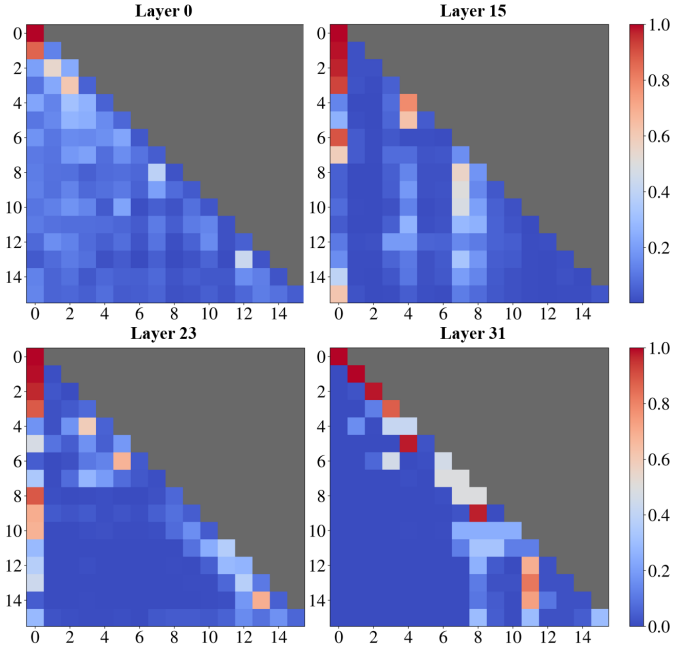


Fig. 5: Average attention weight maps for dense attention in OPT-6.7B on the Wiki-Text-2 dataset [24]. The sequence length is 16. Grey blocks mean the values are masked with zeros, due to the autoregressive LLM inference.

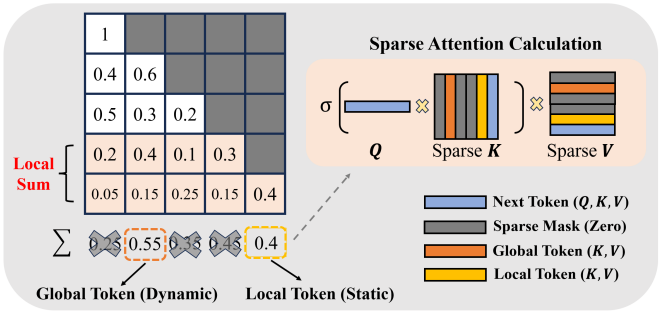


Fig. 6: An illustrative example of our proposed Sparse Window Attention (SWA) algorithm using 40% caching ratio. The left matrix denotes the attention weight map. We calculate the local attention sum using solely the two most recent tokens. The locally static token is kept regardless of the value of the local attention sum. The globally dynamic token is selected as the one with the highest local attention sum.

tokens to preserve language sequential semantics and generate dynamic patterns to capture the dynamically changing semantic importance of prior tokens. The importance of the prior tokens for future token generation is determined by the sum of the local attention weights. Figure 6 draws an example of our SWA algorithm, with the algorithm details formulated as in Algorithm 1.

Our method is based on the hypothesis that multiple preceding steps can provide better hints on which tokens are more important than a single step. The resultant sparse patterns are shown on the top of Figure 4 (d). Note that there do exist prior works that generate sparse patterns based on the entire attention weights [36, 43]. However, the entire attention weights will quadratically increase memory footprint with

Algorithm 1 ALISA's Sparse Window Attention

Input: Previous attention weight AW, query Q, keys and values K, V, caching ratio r, sequence length n, hidden dimension d. Note that this work evenly splits final tokens into k globally dynamic and k local static tokens. The local attention sum is also reduced along the head dimension (not shown for conciseness).

Output: Attention score $Attn$

$k = \lfloor \frac{nr}{2} \rceil$
$S = \sum AW[n-k:n-1]$ ▷ Local attention sum
$I^l = [n-k, ..., n-1]$ ▷ Locally static tokens
$I^g = \text{argmax}_k S$ ▷ Globally dynamic tokens
$I = [I^l, I^g]$ ▷ Sparse tokens
$K_s, V_s = K[I,:], V[I,:]$ ▷ Sparse KV tensors
$AW = \sigma(\frac{QK_s^T}{\sqrt{d}})$ ▷ Attention weight
$Attn = AW \cdot V_s$ ▷ Softmax & attention score
return $Attn$

sequence length, thus not scalable upon long sequences. We plot the distribution of the resultant attention scores at the bottom of Figure 4 (d). Unlike previous fixed sparse patterns, our SWA produces a nearly identical power-law distribution as the dense attention and obtains a Spearman correlation score close to 1. This similarity validates the efficacy of our SWA algorithm. The details of SWA are formulated in Algorithm 1. Two key differences exist between dense attention and SWA. First, the algorithm entails a caching ratio to determine how many tokens to keep at each step for KV sparsity and apply the sparse masks at the token level. While irregular sparsity could exist across tokens, each token is still a dense tensor. Second, we use gather operations to pack sparse KV tensors into a dense one and perform dense matrix operations. Therefore, despite the multi-step attention calculation in SWA, both the computation and memory access for SWA remain regular, if we target a proper granularity.

V. ALISA System Design

Since SWA introduces sparse KV tensors dynamically, designing a system for LLM inference that handles such sparsity effectively is essential. We propose dynamic scheduling to ensure SWA lives up to its potential at the system level. Then, we leverage KV compression to further improve the system-level performance. Our proposed ALISA is a synergetic symbiosis of SWA, dynamic scheduling, and KV compression. Specifically, SWA identifies important KV tensors and generates sparse patterns. Then the dynamic scheduling utilizes important tokens and user-specified caching ratio to balance sparsity-aware caching and recomputation at the token level during LLM inference. The KV compression further reduces the overall memory footprint of KV tensors by quantizing them into INT8 format.

A. Dynamic Scheduling

Three-Phase Scheduling. Since the size of KV tensors gradually increases with longer sequences, it is evident that the

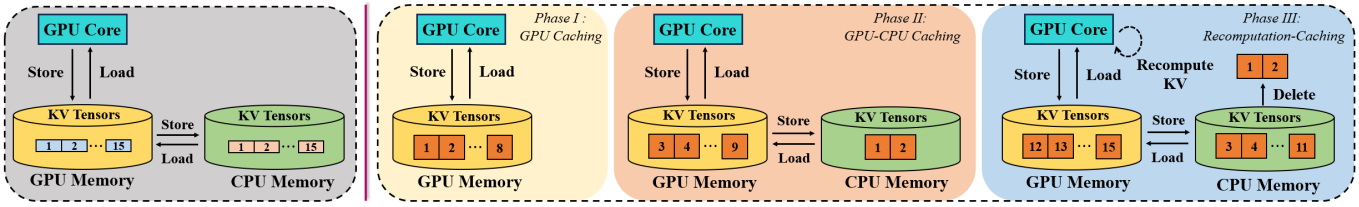
Algorithm 2 ALISA's Dynamic Scheduling

Initialization: GPU core GC, GPU memory GM, CPU memory CM, sequence length n, phase switch step $\{p_1, p_2\}$, offload ratio α and recompute ratio β for KV tensors.
for all $j < n$ **do**
# Load
  if $j \geq p_1$ **then** ▷ Phase II & III
    $CM \rightarrow GM$.load$(*)$
  end if
  $GM \rightarrow GC$.load$(*)$ ▷ Phase I&II&III
Compute
  if $j \geq p_2$ **then** ▷ Phase III
    Recompute$(*)$
  end if
  Update(K_j, V_j) ▷ Attention computation
Store
  $GC \rightarrow GM$.store(K_j, V_j) ▷ Phase I&II&III
  if $j \geq p_1$ **then** ▷ Phase II
    if $j \geq p_2$ **then** ▷ Phase III
      CM.delete(K_j^β, V_j^β)
    end if
    $GM \rightarrow CM$.store(K_j^α, V_j^α)
  end if
end for

engaged memory will increase over time. Due to the high cost of CPU memory I/O accesses, one shall balance the memory access and computation at the token level to maximize the performance. Our scheduling is described as follows.

- Phase I: *GPU Caching*. All KV tensors can fit in GPU memory and are stored in the GPU.
- Phase II: *GPU-CPU Caching*. The total size of all KV tensors exceeds the capacity of GPU memory, and the KV tensors are split at the token level on both GPU and CPU memory and accessed upon need.
- Phase III: *Recomputation-Caching*. After a certain sequence length, partial KV tensors are deleted from the CPU and recomputed in GPU if needed instead of being accessed from CPU memory.

We illustrate our scheduling with an illustrative example in Figure 7 (b) and a formulation in Algorithm 2. Each inference pass contains load, compute, and store parts. Load from GPU memory to GPU core is mandatory for all phases. In Phase II and III, load from CPU to GPU happens before load from GPU memory to GPU core. The new KV tensors will be computed and then stored in GPU memory. In Phase II and III, certain KV tensors in GPU memory will be stored (offloaded) to CPU memory. Since the global sparse patterns vary from step to step, we choose to keep the KV tensors for the locally static tokens in the GPU and store the preceding ones in the CPU. Though there exist caching policies such as Belady's Algorithm [2], such oracle methods could be too computationally expensive to be impractical for efficient LLM inference. Our heuristic-based caching policy can effectively



(a) FlexGen (Static Scheduling) (b) ALISA (Dynamic Scheduling)

Fig. 7: (a) FlexGen's static scheduling. This scheduling splits KV tensors along the head dimension and remains static across different sequence lengths. (b) ALISA's dynamic scheduling. In phase I, the entire KV tensors are small enough to fit in GPU memory, and no CPU memory access exists. In phase II, when GPU capacity is not enough for all KV tensors, the CPU is also used for caching preceding KV tensors. In phase III, recomputing partial KV tensors is faster than retrieving them from CPU memory. These KV tensors are deleted from CPU memory and recomputed in GPU. The phase change is triggered by the sequence length, and the autoregressive inference of different tokens can be in different phases.

TABLE II: Notations.

h, l, b	hidden dimension, layer count, batch size
s, n	input length, output length
r, B	KV caching ratio, CPU-GPU bandwidth
α, β, p_1, p_2	offload/recompute ratio, phase switch step
T^c, T^r	Time for compute and recompute
T^m	Time for KV caching (CPU-GPU)

reduce the potential CPU memory access with small enough compute overheads (compared to the memory bottleneck). In Phase III, we delete the oldest KV tensors in the CPU and recompute these KV tensors in the GPU core when needed.

In contrast, prior works usually pre-defined static scheduling for KV tensors throughout the LLM inference [21, 31, 43], as shown in Figure 7 (a). They fail to leverage the opportunity from dynamic memory capacity changes upon longer sequences, leading to sub-optimal performance.

Sparsity-Aware Caching. The subsequent question is how to determine the phase switch step and offload and recompute ratio of KV tensors. We formulate this question as an optimization problem to minimize the total execution time. We list the relevant notations in Table II. With FP16 format, the size of KV tensors for each token is $4 \cdot b \cdot l \cdot h$ bytes. At a sequence length j, we denote the number of tokens moved from GPU to CPU as $\theta_j^c(\alpha) = \alpha(j+s)$ and the number of tokens moved from CPU to GPU as θ_j^g. The execution time of caching at step j can be estimated as:

$$T_j^m(\alpha) = \frac{4 \cdot b \cdot l \cdot h \cdot (\theta_j^c + \theta_j^g)}{B} \tag{3}$$

$$p_1 \leq j < n, \quad 0 \leq \theta_j^g \leq \lfloor (s+j)r \rceil \tag{4}$$

The optimization of the total execution time is formulated as:

$$\min_{\{\alpha,\beta,p_1,p_2\}} \quad \sum_{j=1}^{p_2} T_j^c + \sum_{j=p_1}^{n} T_j^m(\alpha) + \sum_{j=p_2}^{n} T_j^r(\beta) \tag{5}$$

$$\text{s.t.} \quad 0 \leq p_1 < p_2 \leq n, 0 < \alpha < 1, 0 < \beta < 1 \tag{6}$$

We solve this problem by dividing it into two sub-problems, including a data transfer problem and a computation problem. The data transfer problem (the second term) is solved using hardware and software constraints, including memory capacity, bandwidth, KV tensor size, etc. Conversely, the computation problem (the first and third terms) is solved via profiling. We profile the execution time for compute and recompute with different configurations and create a mapping between input configurations and their execution time. Then, we apply a greedy search method to solve the optimization problem for the best performance. This process is done offline, introducing no overhead during LLM inference.

B. KV Compression

Previous works have utilized quantization to accelerate attention computation by compressing model weights [17, 22]. In this work, we leverage quantization for a different purpose, i.e., compressing KV tensors to reduce memory access. We adopt a fine-grained channel-wise quantization for KV tensors for better inference robustness [9]. More specifically, we use the following formula to quantize KV tensors to b-bit integers in memory and de-quantize them to their original format (FP16 in this work) for computation:

$$x_{\text{quant}} = \text{round}(\frac{1}{\lambda}x + z), \quad x = \lambda(x_{\text{quant}} - z) \tag{7}$$

where the scaling factor $\lambda = \frac{max-min}{2^b-1}$, and zero point $z = \text{round}(\frac{-2^b}{max-min})$. Previous work finds that for OPT model can be compressed up to INT4 while maintaining accuracy [14]. In this work, we choose to quantize KV tensors to INT8 to ensure our KV compression can be generalized to more LLMs.

VI. Evaluation

A. Experimental Setup

Models and datasets. We use three open-sourced families of LLM models: OPT with 6.7B, 13B, and 30B parameters [42], LLaMA with 7B, 13B, and 33B parameters [34], and Pythia with 6.7B and 12B parameters [4]. For algorithm-related evaluations, we use the lm-evaluation-harness library [18] and perform two popular language-related tasks on seven

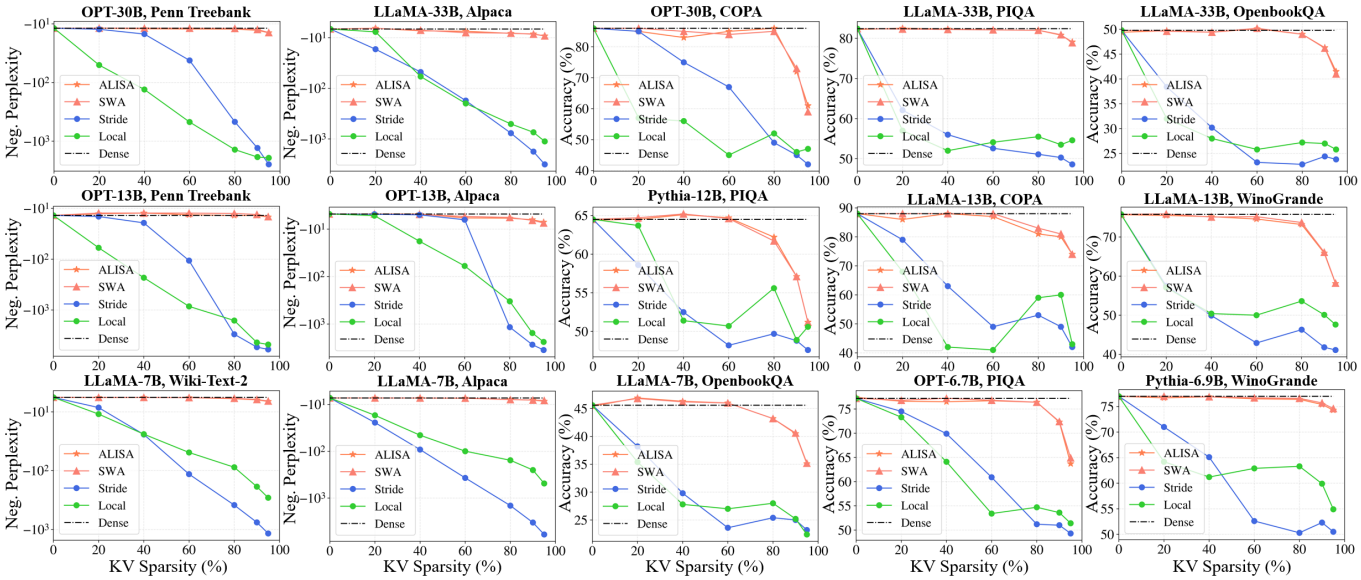


Fig. 8: Accuracy of ALISA (SWA + Compression), SWA, dense attention, local attention [3], and strided attention [8]. Along the y-axis, we arrange the measurements to be higher is better. The input length is set to 2048 for all the datasets to match the maximal context length of each LLM. Negative perplexity and accuracy are utilized to measure the language modeling and question-answering tasks, respectively.

different datasets, namely language modeling for Wiki-Text-2 [24], Penn Treebank [23] and Alpaca [33], and 4-shot question-answering inference for PIQA [5], COPA [41], OpenBookQA [25], and Winogrande [30]. We report task-specific metrics, e.g., perplexity for language modeling and accuracy for question-answering, across different model types and scales. We set the input length as 2048, matching the maximal context length for LLMs, to showcase the algorithmic performance (perplexity and accuracy in this work) when operating at full context.

For the system evaluation, we sample and tokenize inputs from the Alpaca dataset. We use an input sequence length of 128 and an output sequence length of 512 to test our system under varying batch size configurations, ranging from 4 to 64. We aim to evaluate the LLM system performance at all possible model scales, unless the configuration is not available (e.g., Pythia-30B does not exist). The evaluation metric for performance is token throughput, defined as the end-to-end execution time (both prefilling and decoding stages) divided by the total number of generated tokens.

Baselines. To validate the accuracy, we use dense attention, local attention [3], and strided attention [8] as our baselines. For system experiments, we use DeepSpeed-ZeRO [1], HuggingFace Accelerate [39], and FlexGen [31], and vLLM [21]as baselines. DeepSpeed-ZeRO is a deep learning optimization software developed to improve the computation and memory efficiency of training and inference for large models. For LLM inference, DeepSpeed-ZeRO performs offloading weights instead of intermediate KV tensors. HuggingFace Accelerate is another open-sourced library that focuses on promoting easy and reproducible transformer-based research. It supports offloading the whole KV tensors to the CPU memory during LLM inference. FlexGen is a very recent LLM-specific work that focuses on optimizing LLM inference in single GPU-CPU systems. It defines a static scheduling allocation strategy by solving an offline linear programming problem to minimize the total execution time given the memory constraints. vLLM is a dedicated online LLM inference serving system for multi-tenant user requests [21]. It manages the KV tensors at the block level (fixed group of tokens). Each block is stored in non-contiguous paged memory and is swapped between CPU and GPU memory.

Implementation. We conduct our experiments in single GPU-CPU systems. We use two types of GPUs, namely NVIDIA Tesla V100 with 16/32 GB HBM and NVIDIA H100 with 80 GB HBM. Due to GPU memory constraints, we run 30B level models only on H100 GPUs. The CPU is 2.60 GHz Intel Xeon with 128 GB DRAM, and the bandwidth between GPU and CPU is 20 GB/s. ALISA is implemented on top of FlexGen [31] and HuggingFace Transformers [40]. FlexGen allows users to offload model weights, KV tensors, and activations simultaneously. Since we focus on optimizing KV caching, we keep both the model weights and activations always in GPU memory. In terms of memory allocation, we manage the memory space at the token level and schedule KV tensors in a layerwise manner. We use the FP16 format for all variables, except the KV compression.

B. Accuracy

We evaluate the accuracy for different KV sparsity, with results given in Figure 8. We observe that *ALISA has consistent and significant improvements over local and strided attention methods* across different model types, model sizes, and datasets, demonstrating the effectiveness of ALISA. We

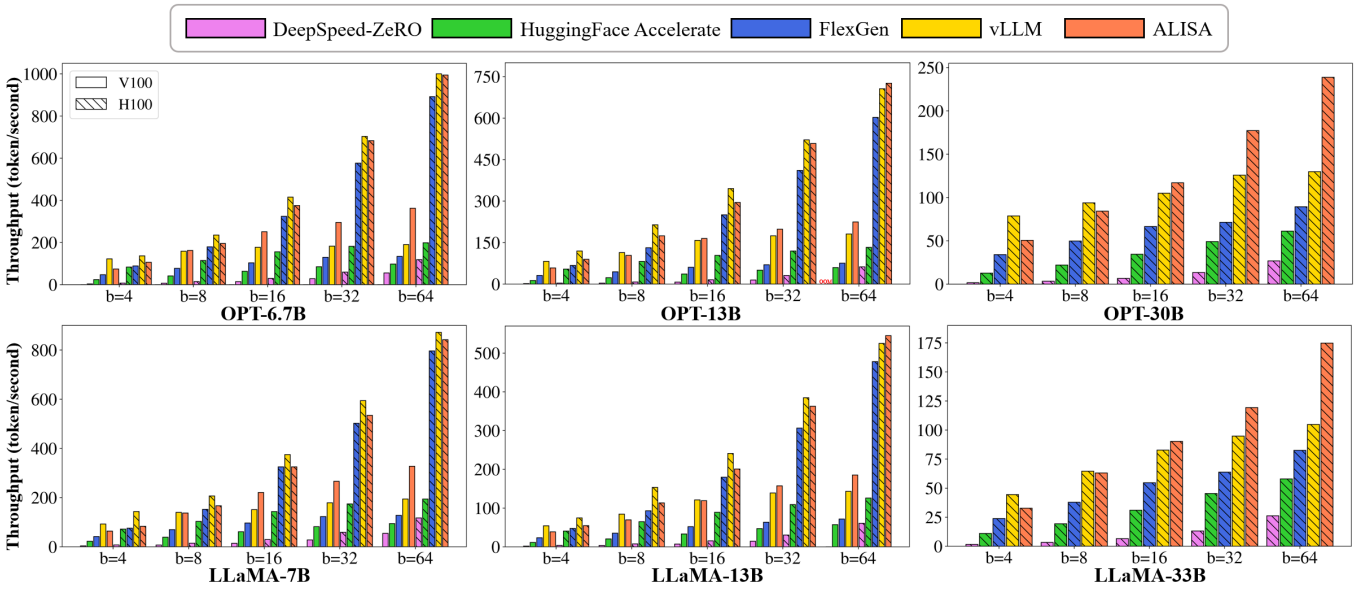


Fig. 9: Throughput of ALISA with 80% KV Sparisty and baselines, including DeepSpeed-ZeRO [1], HuggingFace Accelerate [39], FlexGen [31], and vLLM [21] on the Alpaca dataset [33]. Along the y-axis, higher measurements are better. OOM denotes out-of-memory error. We use an input length of 128 and an output length of 512. No results are given for 30B-level models on V100, as the model weights exceed the GPU memory capacity.

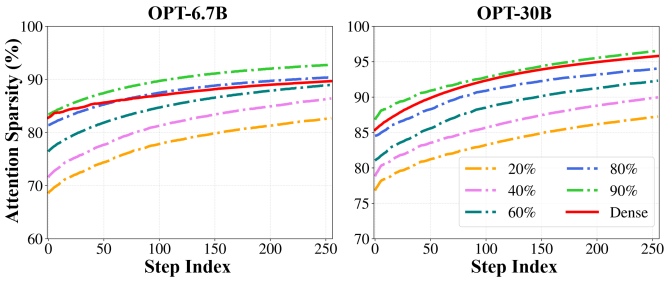


Fig. 10: Attention weight sparsity (averaged across all layers) upon different KV sparsity. We consider elements as zeros if they fall below 1% of the row-wise maximum value.

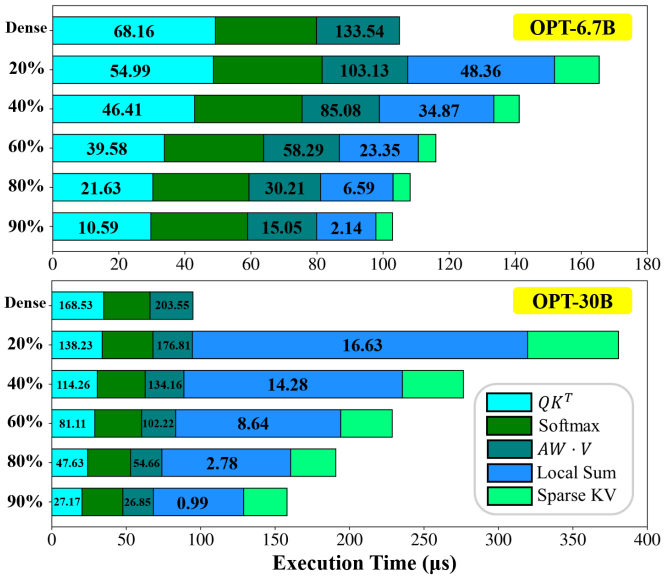


Fig. 11: Execution time of a single attention module. Numbers within the bar indicate the corresponding floating point operations per second (FLOPS), either MAC or ADD. We use a batch size of 64 and a sequence length of 128.

summarize three key insights here. First, ALISA is a much more robust sparse attention method for LLMs. For example, for LLaMA-33B on the top right, the accuracy of local and strided attention collapses instantly when sparse attention is adopted, i.e., the largest accuracy drop occurs from 0% to 20% KV sparsity, while ALISA almost maintains an identical accuracy to that of dense attention up to 80% KV sparsity. Second, ALISA is more robust when LLMs become larger. With ALISA, fewer accuracy collapses occur at 80% KV sparsity when LLM sizes increase from the 7B level to 13B/30B level, regardless of the model families. However, no such trends exist in local and strided attention. Third, KV compression is scalable with almost no accuracy impact for LLMs. In all settings, we see that the accuracy of ALISA almost perfectly tracks that of SWA. In certain settings, ALISA can even outperform dense attention, since well-structured sparsity can often act as regularization to improve accuracy on unseen datasets [19, 38]. Though minor discrepancies exist in OPT-30B and LLaMA-13B on the COPA dataset, we conclude that KV compression is practically scalable with KV sparsity and model size.

C. Performance

End-to-end Throughput. We further evaluate the end-to-end system performance (i.e., throughput) of ALISA. We choose 80% as the evaluated KV sparsity, as it is the maximum

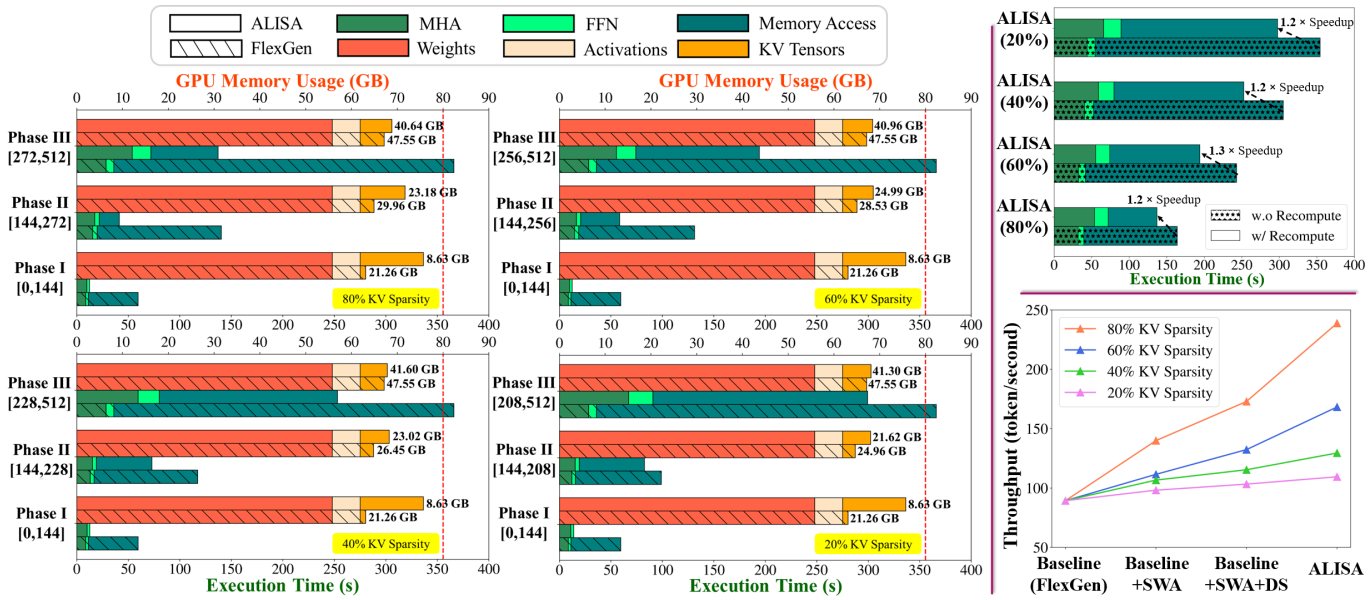


Fig. 12: LLM inference. All experiments are conducted using OPT-30B with a batch size of 64, input length of 128, and output length of 512 on one NVIDIA H100 GPU. All bars correspond to the sequence length at the end of the phase. (a) Left four: execution time and memory usage of FlexGen and ALISA by our proposed scheduling phase for different KV sparsity. Numbers on the right-hand side of the memory bar indicate CPU memory usage and the red-dot line denotes the GPU memory capacity. (b) Top right: impact of recomputation on the execution time for different KV sparsity at the full sequence length. (c) Bottom right: ablation study on the impact of techniques for different KV sparsity. DS denotes dynamical scheduling.

value for ALISA to retain good algorithmic performance, i.e., perplexity and accuracy, less than 5% drop for most tasks shown in Figure 8. Specifically, the drop for the Alpaca dataset is around 3%. Figure 9 shows the performance of OPT and LLaMA models on the Alpaca dataset. *Overall, ALISA offers the highest attainable throughput for LLM inference in resource-constrained systems.* There are three key observations. First, ALISA achieves consistent speedup over all baselines, showing $1.4 \sim 3.0\times$ higher throughput over FlexGen. Prior works like DeepSpeed-ZeRO are not fully optimized for LLM inference by introducing out-of-memory errors upon large batch sizes, since it does not offload KV tensors. Second, ALISA is more scalable than previous works. As the batch size grows, the speedup of ALISA over FlexGen and other methods increases. Third, ALISA can sustain up to $1.9\times$ improvement over vLLM under larger batch sizes and larger model scales, especially when GPU memory capacity is limited. This is due to two reasons: 1) ALISA co-designs the sparsity patterns and KV caching to reduce the memory footprint, while vLLM only optimizes the memory management of KV tensors; 2) the dynamic scheduling strategy in ALISA features recomputation to further alleviate the memory bottleneck upon large KV tensors. Note that when serving smaller models with smaller batch sizes, vLLM outperforms as it is optimized for online serving with fine-grained memory management.

Attainable Sparsity. We further show why ALISA can achieve this speedup. Figure 10 shows the achieved sparsity after SWA. Two key observations exist. First, for both LLMs, allowing more sparse KV tensors will increase the sparsity in attention weights. Second, for larger LLMs, we need a higher KV sparsity to close the gap between the attainable sparsity in our SWA and dense attention. However, the accuracy with higher KV sparsity will likely drop according to Figure 8. Overall, the insight is that ALISA can reasonably take advantage of the opportunities in sparse attention to accelerate LLM inference.

Breakdown of Attention Module. To better understand the impact of SWA, we profile the execution time of key operations in Algorithm 1, with results given in Figure 11. There are two key observations here. First, SWA introduces an execution overhead, which varies across different KV sparsity. Higher KV sparsity in SWA always reduces the execution time. The main sources of reduction are the process of QK^T, local attention sum, and sparse KV tensors (i.e., using sparse token indices to generate dense KV tensors). Larger LLMs incur higher overheads, especially in local attention sum and sparse KV tensors. The reason is that larger LLMs have larger model dimensions. For example, the hidden dimension and head number increase from $[4096, 32]$ in OPT-6.7B to $[7168, 56]$ in OPT-30B. This larger overhead also validates our argument that prior works that generate sparse attention based on the entire attention weight are not scalable [36, 43]. Second, there exists under-utilization in the QK^T computation for SWA. The corresponding execution time does not decrease proportionally as KV sparsity increases, leading to a significant FLOPS drop. The main reason is that a smaller dense tensor gathered from sparse KV tensors can not fully utilize massive parallel GPU cores. The execution time for the local sum scales with KV sparsity. Higher sparsity is induced by a lower caching ratio, which in turn reduces the number of additions in the local

sum. However, the local sum could spend more time than QK^T computation, due to its low data use, i.e., vector vs. matrix operation.

Breakdown of LLM Inference. Figure 12 shows the details of the full LLM inference. There are three key observations in Figure 12 (a). First, ALISA is always faster than FlexGen for all KV sparsity and all phases. With higher KV sparsity, the speedup of ALISA over FlexGen is more significant. Both the time spent on computation and memory access is less when KV sparsity goes higher since fewer KV tensors are involved per step. However, the main contributor to reducing the execution time is that fewer KV tensors need to be moved between CPU and GPU. As we have both statically local and dynamically global sparse patterns, we prefer allocating local tokens in GPU to reduce CPU memory access, since global tokens are less predictable. Second, ALISA always makes better use of the GPU memory than FlexGen in all cases. The total memory requirement of all KV tensors (the total GPU and CPU memory usage) increases with the sequence length. The GPU memory usage is not directly related to KV sparsity, as our dynamic scheduling optimizes the execution time instead of GPU memory usage. Third, the size of KV tensors indeed impacts when a phase starts. Different KV sparsity leads to varying tensor sizes and triggers Phase III at different sequence lengths, and higher KV sparsity enters Phase III later. ALISA in Phase III has a smaller size of KV tensors than FlexGen due to deleting partial KV tensors. Overall, ALISA manages KV caching at the token level and balances the caching and recomputation in a more fine-grained manner than FlexGen. Figure 12 (b) studies the impact of recomputation in Phase III. We observe that recomputation can reduce the total execution time by $1.2 \sim 1.3\times$. Though recomputation induces additional computation overhead, it results in a more substantial reduction in execution time due to decreased memory accesses. Figure 12 (c) shows the ablation study. Across different KV sparsity, we observe that different techniques almost contribute equally, and the gain of each technique increases proportionally with the KV sparsity.

VII. Conclusion

In this work, we present an algorithm-system co-design solution, ALISA, to accelerate LLM inference in resource-constrained systems. On the algorithm level, ALISA adopts a Sparse Window Attention (SWA) algorithm to create a mixture of globally dynamic and locally static sparse patterns and reduces the memory footprint with negligible accuracy degradation. On the system level, ALISA leverages a three-phase scheduler to dynamically allocate KV tensors and achieves optimal throughput by balancing caching and recomputation. Experiments show that, in single GPU-CPU systems, ALISA achieves up to $3\times$ and $1.9\times$ throughput improvement over FlexGen and vLLM, respectively.

VIII. Acknowledgement

This work was sponsored in part by the U.S. National Science Foundation (NSF) under Grants 1907765, 2028481, and 2400014. The authors would like to thank the anonymous ISCA reviewers for their constructive feedback to improve this work.

References

[1] R. Y. Aminabadi, S. Rajbhandari, M. Zhang, A. A. Awan, C. Li, D. Li, E. Zheng, J. Rasley, S. Smith, O. Ruwase, and Y. He, "Deepspeed-inference: Enabling efficient inference of transformer models at unprecedented scale," *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15, 2022.

[2] L. A. Belady, R. A. Nelson, and G. S. Shedler, "An anomaly in space-time characteristics of certain programs running in a paging machine," *Commun. ACM*, vol. 12, pp. 349–353, 1969.

[3] I. Beltagy, M. E. Peters, and A. Cohan, "Longformer: The long-document transformer," *ArXiv*, vol. abs/2004.05150, 2020.

[4] S. R. Biderman, H. Schoelkopf, Q. G. Anthony, H. Bradley, K. O'Brien, E. Hallahan, M. A. Khan, S. Purohit, U. S. Prashanth, E. Raff, A. Skowron, L. Sutawika, and O. van der Wal, "Pythia: A suite for analyzing large language models across training and scaling," *ArXiv*, vol. abs/2304.01373, 2023.

[5] Y. Bisk, R. Zellers, R. L. Bras, J. Gao, and Y. Choi, "Piqa: Reasoning about physical commonsense in natural language," *ArXiv*, vol. abs/1911.11641, 2019.

[6] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, and et al, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[7] B. Chen, Z. Liu, B. Peng, Z. Xu, J. L. Li, T. Dao, Z. Song, A. Shrivastava, and C. Ré, "Mongoose: A learnable lsh framework for efficient neural network training," *International Conference on Learning Representations*, 2021.

[8] R. Child, S. Gray, A. Radford, and I. Sutskever, "Generating long sequences with sparse transformers," *ArXiv*, vol. abs/1904.10509, 2019.

[9] B. Chmiel, R. Banner, G. Shomron, Y. Nahshan, A. Bronstein, U. Weiser *et al.*, "Robust quantization: One model to rule them all," *Advances in neural information processing systems*, vol. 33, pp. 5308–5317, 2020.

[10] S. Dai, H. Genc, R. Venkatesan, and B. Khailany, "Efficient transformer inference with statically structured sparse attention," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2023, pp. 1–6.

[11] T. Dao, "Flashattention-2: Faster attention with better parallelism and work partitioning," *ArXiv*, vol. abs/2307.08691, 2023.

[12] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," *Advances in Neural Information Processing Systems*, vol. 35, pp. 16 344–16 359, 2022.

[13] J. Dass, S. Wu, H. Shi, C. Li, Z. Ye, Z. Wang, and Y. Lin, "Vitality: Unifying low-rank and sparse approximation for vision transformer acceleration with a linear taylor attention," *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 415–428, 2022.

[14] T. Dettmers and L. Zettlemoyer, "The case for 4-bit precision: k-bit inference scaling laws," in *International Conference on Machine Learning*. PMLR, 2023, pp. 7750–7774.

[15] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, vol. 1, 2019, p. 2.

[16] J. Ding, S. Ma, L. Dong, X. Zhang, S. Huang, W. Wang, N. Zheng, and F. Wei, "Longnet: Scaling transformers to 1,000,000,000 tokens," *ArXiv*, vol. abs/22307.02486, 2023.

[17] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, "Gptq: Accurate post-training quantization for generative pre-trained transformers," *International Conference on Learning Representations (ICLR)*, 2023.

[18] L. Gao, J. Tow, S. Biderman, S. Black, A. DiPofi, C. Foster, L. Golding, J. Hsu, K. McDonell, N. Muennighoff, J. Phang, L. Reynolds, E. Tang, A. Thite, B. Wang, K. Wang, and A. Zou, "A framework for few-shot language model evaluation," 2021. [Online]. Available: https://doi.org/10.5281/zenodo.5371628

[19] Y. Guo, A. Yao, and Y. Chen, "Dynamic network surgery for efficient dnns," *Advances in neural information processing systems*, vol. 29, 2016.

[20] N. Kitaev, L. Kaiser, and A. Levskaya, "Reformer: The efficient transformer," *International Conference on Learning Representations (ICLR)*, 2019.

[21] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

[22] J. Lin, J. Tang, H. Tang, S. Yang, X. Dang, and S. Han, "Awq: Activation-aware weight quantization for llm compression and acceleration," *ArXiv*, vol. abs/2306.00978, 2023.

[23] M. P. Marcus, B. Santorini, and M. A. Marcinkiewicz, "Building a large annotated corpus of english: The penn treebank," *Comput. Linguistics*, vol. 19, pp. 313–330, 1993.

[24] S. Merity, C. Xiong, J. Bradbury, and R. Socher, "Pointer sentinel mixture models," *ArXiv*, vol. abs/1609.07843, 2016.

[25] T. Mihaylov, P. Clark, T. Khot, and A. Sabharwal, "Can a suit of armor conduct electricity? a new dataset for open book question answering," *ArXiv*, vol. abs/1809.02789, 2018.

[26] OpenAI, "Introducting chatgpt," 2022. [Online]. Available: https://openai.com/blog/chatgpt

[27] M. Ott, S. Edunov, A. Baevski, A. Fan, S. Gross, N. Ng, D. Grangier, and M. Auli, "fairseq: A fast, extensible toolkit for sequence modeling," in *North American Chapter of the Association for Computational Linguistics*, 2019, pp. 6151–6162.

[28] R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, J. Heek, K. Xiao, S. Agrawal, and J. Dean, "Efficiently scaling transformer inference," *Proceedings of Machine Learning and Systems*, vol. 5, 2023.

[29] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019. [Online]. Available: https://insightcivic.s3.us-east-1.amazonaws.com/language-models.pdf

[30] K. Sakaguchi, R. L. Bras, C. Bhagavatula, and Y. Choi, "Winogrande: An adversarial winograd schema challenge at scale," *Commun. ACM*, vol. 64, pp. 99–106, 2019.

[31] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, D. Y. Fu, Z. Xie, B. Chen, C. W. Barrett, J. Gonzalez, P. Liang, C. Ré, I. C. Stoica, and C. Zhang, "High-throughput generative inference of large language models with a single gpu," in *International Conference on Machine Learning*. PMLR, 2023, pp. 31 094—-31 116.

[32] H. Shi, J. Gao, X. Ren, H. Xu, X. Liang, Z. Li, and J. T.-Y. Kwok, "Sparsebert: Rethinking the importance analysis in self-attention," in *International Conference on Machine Learning*. PMLR, 2021, pp. 9547–9557.

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

[34] H. Touvron, L. Martin, K. R. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. M. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. S. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. M. Kloumann, A. V. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom, "Llama 2: Open foundation and fine-tuned chat models," *ArXiv*, vol. abs/2307.09288, 2023.

[35] A. Vaswani, N. M. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, vol. 30, 2017.

[36] H. Wang, Z. Zhang, and S. Han, "Spatten: Efficient sparse attention architecture with cascade token and head pruning," *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 97–110, 2020.

[37] S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma, "Linformer: Self-attention with linear complexity," *ArXiv*, vol. abs/2006.04768, 2020.

[38] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," *Advances in neural information processing systems*, vol. 29, 2016.

[39] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, and J. Brew, "Huggingface's transformers: State-of-the-art natural language processing," *ArXiv*, vol. abs/1910.03771, 2019.

[40] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, and J. Brew, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*, 2020, pp. 38–45.

[41] J. Yeo, G. Lee, G. Wang, S. Choi, H. Cho, R. K. Amplayo, and S. won Hwang, "Visual choice of plausible alternatives: An evaluation of image-based commonsense causal reasoning," in *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, 2018.

[42] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, and et al, "Opt: Open pre-trained transformer language models," *ArXiv*, vol. abs/2205.01068, 2022.

[43] Z. A. Zhang, Y. Sheng, T. Zhou, T. Chen, L. Zheng, R. Cai, Z. Song, Y. Tian, C. Ré, C. W. Barrett, Z. Wang, and B. Chen, "H2o: Heavy-hitter oracle for efficient generative inference of large language models," *ArXiv*, vol. abs/2306.14048, 2023.