# SpInfer: Leveraging Low-Level Sparsity for Efficient Large Language Model Inference on GPUs

Ruibo Fan[1], Xiangrui Yu[1], Peijie Dong[1], Zeyu Li[1], Gu Gong[1], Qiang Wang[2], Wei Wang[3], Xiaowen Chu[1,3]
[1]The Hong Kong University of Science and Technology (Guangzhou), China
[2]Harbin Institute of Technology, Shenzhen, China
[3]The Hong Kong University of Science and Technology, Hong Kong SAR
{ruibo.fan,xyu868,pdong212,zli755,ggong504}@connect.hkust-gz.edu.cn
qiang.wang@hit.edu.cn,weiwa@cse.ust.hk,xwchu@hkust-gz.edu.cn

## Abstract

Large Language Models (LLMs) have demonstrated remarkable capabilities, but their immense scale poses significant challenges in terms of both memory and computational costs. While unstructured pruning offers promising solutions by introducing sparsity to reduce resource requirements, realizing its benefits in LLM inference remains elusive. This is primarily due to the storage overhead of indexing non-zero elements and the inefficiency of sparse matrix multiplication (SpMM) kernels at low sparsity levels (around 50%). In this paper, we present SpInfer, a high-performance framework tailored for sparsified LLM inference on GPUs. SpInfer introduces Tensor-Core-Aware Bitmap Encoding (TCA-BME), a novel sparse format that minimizes indexing overhead by leveraging efficient bitmap-based indexing, optimized for GPU Tensor Core architectures. Furthermore, SpInfer integrates an optimized SpMM kernel with Shared Memory Bitmap Decoding (SMBD) and asynchronous pipeline design to enhance computational efficiency. Experimental results show that SpInfer significantly outperforms state-of-the-art SpMM implementations (up to 2.14× and 2.27× over Flash-LLM and SparTA, respectively) across a range of sparsity levels (30% to 70%), with substantial improvements in both memory efficiency and end-to-end inference speed (up to 1.58×). SpInfer outperforms highly optimized cuBLAS at sparsity levels as low as 30%, marking the first effective translation of unstructured pruning's theoretical advantages into practical performance gains for LLM inference.

***CCS Concepts:*** • **Computing methodologies → Shared memory algorithms**.

***Keywords:*** Unstructured Pruning, SpMM, Sparse, LLM Inference, GPU, Tensor Core


*EuroSys '25, March 30-April 3, 2025, Rotterdam, Netherlands*

ACM ISBN 979-8-4007-1196-1/2025/03
https://doi.org/10.1145/3689031.3717481

**ACM Reference Format:**
Ruibo Fan[1], Xiangrui Yu[1], Peijie Dong[1], Zeyu Li[1], Gu Gong[1], Qiang Wang[2], Wei Wang[3], Xiaowen Chu[1,3]. 2025. SpInfer: Leveraging Low-Level Sparsity for Efficient Large Language Model Inference on GPUs. In *Twentieth European Conference on Computer Systems (EuroSys '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3689031.3717481

## 1 Introduction

Large Language Models (LLMs) [1, 7, 65, 80] have revolutionized AI applications, demonstrating exceptional capabilities across diverse domains including summarization, instruction following, and question answering. However, these models' immense scale, often comprising billions of parameters, presents significant challenges. The extensive memory requirements and associated computational costs of LLMs render their deployment and inference highly resource-intensive, substantially impeding their widespread implementation on contemporary hardware platforms [94, 102].

In response to these challenges, model compression techniques have gained significant attention, with **weight pruning** (or sparsification) emerging as a promising method for reducing both memory consumption and computational burden [20, 41, 52, 76, 97]. Weight pruning eliminates less salient connections in neural networks, introducing **sparsity** into the model. Pruning methods are categorized into structured [5], semi-structured [20], and unstructured [20, 97]. Structured pruning removes entire components but typically needs costly post-training. Semi-structured, like N:M pruning, balances flexibility and efficiency by controlling sparsity. Unstructured pruning, which removes individual weights freely, offers the most flexibility and typically yields better post-training performance [15, 20, 77], generally surpassing structured methods in accuracy [5, 52].

However, leveraging unstructured sparsity for performance gains and memory savings in LLM inference remains particularly challenging. Unlike smaller models, where higher sparsity ratios can be achieved, LLMs exhibit a far lower tolerance for sparsity. For instance, models like Vision Transformer (LPViT) [88] and ResNet-50 [27] can achieve **70%**

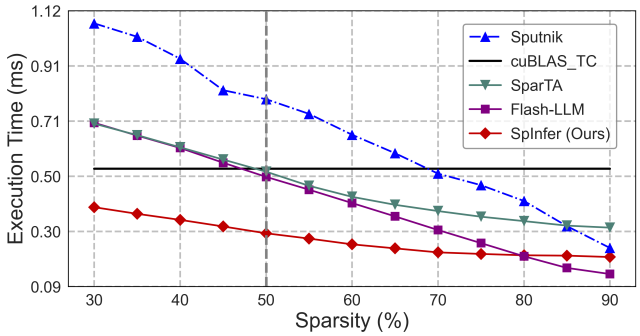


**Figure 1.** Execution time comparison of unstructured SpMM implementations against cuBLAS on Nvidia RTX4090 (M/K/N=28K/8K/16, typical in LLM inference).

to **95%** sparsity without significant accuracy loss, thanks to the feasibility of extensive post-training [24]. However, for extremely large language models, such post-training is often prohibitively expensive due to the high computational and time costs associated with retraining. From an algorithmic perspective, achieving similarly high sparsity levels in LLMs without severe performance degradation is impractical. Current state-of-the-art pruning techniques, such as SparseGPT [20], Wanda [77], GBLM-Pruner [13] and Pruner-Zero [15], typically attain around **50%** sparsity before the adverse effects on model accuracy become unacceptable.

This low sparsity level poses **two key challenges** for realizing the benefits of unstructured pruning in LLM inference. First, **reducing weight storage** at this level of sparsity becomes difficult due to the indexing overhead (i.e., storing indices for non-zero elements). Sparse formats like traditional CSR and Tiled-CSL from Flash-LLM [86], state-of-the-art sparse LLM inference framework, result in increased memory usage at around 50% sparsity, as the need to store both non-zero values and their indices can negate the memory savings from pruning. Second, **achieving practical acceleration** remains a significant hurdle, especially on GPUs, which dominate LLM deployment. While CPU-based sparse acceleration solutions like Neural Magic's DeepSparse [38, 39] have shown promise, GPU acceleration faces unique challenges due to their SIMT execution model and complex memory hierarchy [57]. State-of-the-art sparse matrix multiplication (SpMM) kernels, which provide the foundational support for pruning, struggle to outperform their dense counterparts (cuBLAS [59]). Figure 1 illustrates this performance gap. Despite being specifically designed for LLM pruning, even Flash-LLM struggles to achieve speedup at sparsities of 50% or lower. As a result, LLM sparsification has yet to fully achieve its theoretical potential in real-world systems, leaving a significant gap between theoretical acceleration and practical speedup in LLM inference.

To bridge these gaps, we propose **SpInfer**, a high-performance framework specifically designed to accelerate LLM inference by leveraging low-level unstructured sparsity on GPUs. At the core of SpInfer is the Tensor-Core-Aware Bitmap Encoding (TCA-BME), a novel sparse matrix storage format that minimizes indexing overhead by employing efficient bitmap-based indexing. TCA-BME is carefully designed to align with GPU Tensor Core architecture, ensuring that SpMM operations can fully exploit the computational power of these cores, even in the presence of unstructured sparsity. By reducing the memory footprint of sparse matrices and optimizing data access patterns, TCA-BME enables SpInfer to achieve substantial improvements in both memory efficiency and computational throughput.

Building upon the TCA-BME format, SpInfer integrates a highly optimized SpMM kernel that further enhances performance. The kernel implements a well-optimized data movement path and introduces Shared Memory Bitmap Decoding (SMBD), which enables sparse matrices to be decoded directly within shared memory, significantly reducing the decoding overhead. Additionally, the kernel features an asynchronous pipeline design that overlaps memory transfers with computations, enhancing the utilization of GPU resources.

We evaluate the performance of SpInfer from both the kernel level and the end-to-end framework level. At the kernel level, SpInfer is compared with state-of-the-art SpMM implementations, including both Tensor-Core-based Flash-LLM [86], SparTA [100], SMaT [64] and CUDA-core-based Sputnik [24] and cuSPARSE [60]. SpInfer achieves significant speedups over these methods across different sparsity levels, ranging from low (30%) to moderate (70%). At the framework level, SpInfer is compared with Flash-LLM [86], FasterTransformer [56], and DeepSpeed [4], achieving substantial improvements in generation latency and reductions in memory usage during inference, demonstrating its effectiveness for deployment in resource-constrained environments.

The main contributions of our paper include:

- We conduct detailed analysis and identify indexing overhead as the key bottleneck in realizing benefits from unstructured pruning, highlighting the need to address it for both memory efficiency and computational acceleration.
- We present SpInfer, a high-performance sparse LLM inference framework. At its core, we introduce Tensor-Core-Aware Bitmap Encoding format, which mitigates indexing overhead and efficiently compresses sparse matrices. We also devise a specialized SpMM kernel with tailored optimizations, allowing SpInfer to significantly accelerate sparse matrix computations.
- We demonstrate that SpInfer delivers substantial improvements in both inference speed and memory efficiency at the kernel and framework levels, outperforming previous state-of-the-art solutions across a

wide range of sparsity levels, from low (30%) to moderate (70%). To the best of our knowledge, SpInfer is the first to successfully translate Sparse LLM theoretical speedups into real-world performance benefits.

## 2 Background and Related Work

### 2.1 LLM Architecture and Inference Process

LLMs are built on the transformer architecture [81], which uses stacked layers of self-attention and feed-forward networks (FFNs). The self-attention mechanism enables LLMs to model relationships between all tokens in a sequence. Input tokens are transformed into Query (Q), Key (K), and Value (V) vectors through linear projections. The attention process involves multiplying the Q and K matrices, producing attention scores that weight the V matrix. Additionally, each transformer layer includes an FFN that refines token embeddings through two linear transformations with a non-linear activation in between. LLM inference consists of two phases: the prefill and decode phases. In the prefill phase, the entire input prompt is processed in parallel, while the decode phase generates tokens sequentially in an autoregressive manner, processing one token at a time.

The efficiency of LLM inference relies on matrix multiplications, particularly in self-attention and FFNs. We denote the weight matrix as $W \in \mathbb{R}^{M\times K}$ and the token embeddings as $X \in \mathbb{R}^{K\times N}$, where $M$ is the output dimension, $K$ is the hidden dimension, and $N$ is the number of tokens. The matrix multiplication $W \times X$ yields the transformed token representations. In the prefill phase, $N$ is the sequence length multiplied by the batch size (seq_len $\times$ $BS$). During the decode phase, $N = BS \times 1$, as tokens are processed one at a time in an autoregressive manner.

### 2.2 NVIDIA GPU and Tensor Core

NVIDIA GPUs feature multiple streaming multiprocessors (SMs) with CUDA cores, Tensor Cores (TCs), and a hierarchical memory structure. Thread blocks are scheduled onto SMs, with 32 threads in a warp executing instructions simultaneously in SIMT mode. The memory hierarchy includes high-latency global memory accessible by all threads, faster shared memory within each SM for thread block access, and fast but limited registers private to each thread. The caching system includes an L1 cache per SM, configurable with shared memory, and a unified L2 cache that optimizes bandwidth and latency between processing cores and global memory [57]. TCs are specialized units for accelerating dense matrix multiplication [51, 54, 78]. TCs perform the computation $D_{frag} = A_{frag} \times B_{frag} + C_{frag}$, where $A_{frag} \in \mathbb{R}^{m\times k}$ and $B_{frag} \in \mathbb{R}^{k\times n}$ are inputs, $C_{frag}$ is the accumulator, and $D_{frag}$ is the output. We denote the matrix shape as $m\times k\times n$. For our implementation, we utilize the low-level ***mma*** instructions at the PTX level [58], which offers greater flexibility in managing registers. With FP16 precision, the *mma* instructions require matrix shapes of $16 \times 16 \times 8$ or $16 \times 8 \times 8$. Listing 1 provides an example of an FP16 *mma* instruction. While TCs excel at accelerating dense matrix multiplication, leveraging them to accelerate unstructured SpMM remains challenging.

**Listing 1.** FP16 Tensor Core *mma* instructions: This instruction performs a 16×16×8 matrix multiplication using FP16 inputs stored in .f16x2 registers (Ra, Rb) and accumulates the result into FP32 registers (Rd, with Rc as the accumulator).

```
.reg .f16x2 %Ra<4>, %Rb<2>;
.reg .f32 %Rc<4>, %Rd<4>;
mma.sync.aligned.m16n8k16.row.col.f32.f16.f16.f32
  {%Rd0, %Rd1, %Rd2, %Rd3},
  {%Ra0, %Ra1, %Ra2, %Ra3},
  {%Rb0, %Rb1},
  {%Rc0, %Rc1, %Rc2, %Rc3};
```

### 2.3 Related work

**Quantization and Sparsification.** Quantization and sparsification are key model compression techniques for reducing the computational and memory demands of LLMs. **Quantization** leverages low-precision representations, with numerous works improving algorithms, such as post-training quantization (PTQ) [6, 14, 21, 45, 49, 87, 98] and quantization-aware training (QAT) [16, 48], and system-level support, such as MARLIN [22], LADDER [82], and Qserve [46], making it widely applicable in practice. **Sparsification**, on the other hand, reduces the number of non-zero weights through various pruning strategies, including structured pruning [5, 52] and unstructured pruning [13, 15, 20, 77, 89, 97], typically targeting around 50% sparsity while maintaining accuracy. Although unstructured pruning achieves better precision, its reliance on sparse matrix kernels limits its efficiency on current hardware. Our SpInfer provides practical system-level support for low-level sparsity pruning, while also complementing these quantization techniques. Recent works have also explored dynamic **activation sparsity** to enhance efficiency, such as Deja Vu [50], PIT [99], and PowerInfer [75]. These methods leverage the sparsity induced by ReLU activation functions rather than weight sparsity. However, they require models to either use sparse activation functions like ReLU or undergo retraining. Our approach targets weight sparsity, eliminating the need for retraining and operating in a different scope from these methods.

**Sparse Matrix-Matrix Multiplication.** SpMM computes $O_{M\times N} = Ws_{M\times K} \times X_{K\times N}$, where $Ws$, $X$, and $O$ are the sparse weight matrix, input embedding, and output matrix, respectively. We denote $NNZ$ as the number of non-zero elements in $Ws$. Many works have aimed to accelerate SpMM for highly sparse scientific and GNN workloads [10, 18, 19, 30, 31, 33, 34, 53, 64, 66, 74, 83, 91]. While less effective for low-sparsity LLM inference, their designs provide valuable insights. For DL workloads, works have focused on structured sparsity at various granularities, including

block sparsity [26], vector sparsity [8, 9, 43], and N:M semi-structured pruning [44]. However, these methods' reliance on structured pruning limits their applicability to unstructured sparsity. Recent research has focused on the more challenging issue of unstructured pruning at low sparsity levels. Sputnik [24] applies one-dimensional tiling and reverse offset memory alignment to efficiently utilize CUDA cores. SparTA [100] partitions matrices into 2:4 structured sparse and unstructured sparse components, leveraging both sparse Tensor Cores and CUDA cores. Flash-LLM [86] employs the Load-as-Sparse-Compute-as-Dense approach to reduce memory footprint. While effective at higher sparsity levels (70%-90%), our analysis (Section 3) reveals that the above approaches overlook the indexing overhead, a key bottleneck in low-sparsity scenarios. Consequently, they struggle to reduce storage or enhance performance at sparsity levels of below 50%.

**System-level Optimization.** System-level optimizations involve enhancing both the inference engine and online serving systems. Inference engines primarily aim to accelerate the forward pass through graph and kernel optimizations and offloading techniques. These optimizations focus on refining attention mechanisms (e.g., FlashAttention [11, 12, 71]), restructuring computation graphs (e.g., ByteTransformer [95] and DeepSpeed [4, 29]), and optimizing linear operations (e.g., TensorRT-LLM, MegaBlocks [23], and FlashDecoding++ [32]). Offloading methods, such as those implemented by FlexGen [73] and llama.cpp [25], optimize memory usage by distributing model components across various hardware resources. Our SpInfer, targeting weight pruning, can be combined with these methods to further enhance performance. Moreover, many works focus on optimizing online serving systems to efficiently handle requests from multiple users. Key areas of improvement include memory management [40], continuous batching [29, 40, 93], scheduling strategies [72, 85], and distributed serving [63, 101]. Our work is orthogonal to these serving systems and can complement and improve their performance.

# 3 Gaps and Opportunities

## 3.1 Bottleneck of LLM Inference

LLM inference faces significant computational and storage challenges. Figure 2 shows the runtime and memory breakdown for OPT-13B on 2 RTX4090 GPUs using FasterTransformer, with a batch size of 16 and an output length of 256. Model weight storage occupies 87.6% of memory, and associated matrix multiplication operations (GEMM) consume 61.6% of execution time, constituting the primary bottlenecks. Although weight pruning can potentially reduce both memory and computation by removing less important weights, the low sparsity in LLM pruning limits the practical effectiveness of current pruning methods on modern GPUs. This challenge is further discussed in Section 3.2.

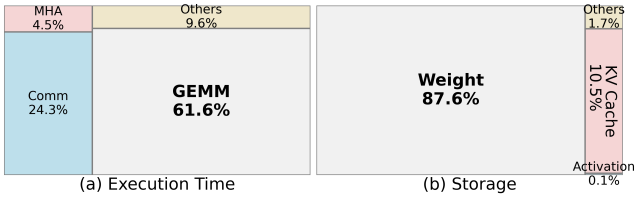


**Figure 2.** Breakdown of OPT-13B on 2 RTX4090 GPUs.

## 3.2 Overlooked Indexing Overheads

Existing sparse LLM inference techniques leverage sparse computation, but introduce significant storage overheads at low sparsity levels due to the need to store indexing information for non-zero elements. Prior works like Flash-LLM, SparTA, and Sputnik commonly overlook these costs. Specifically, indexing overheads not only hinder storage efficiency but also compromise computational performance. The space needed for indices can offset pruning's storage gains, while accessing indices during matrix multiplication can reduce computational efficiency, especially on GPUs, where memory bandwidth is a bottleneck.

### 3.2.1 Gaps in storage complexity.

To quantify the impact of indexing overheads on storage complexity, we define a compression ratio (CR) metric representing the storage efficiency of a sparse matrix format:

$$CR = \frac{2B \times M \times K}{Stor_{Format}}, \tag{1}$$

where $2B \times M \times K$ represents the size of the original dense matrix, and $\text{Stor}_{\text{Format}}$ refers to the compressed storage size of the sparse format. We conduct a comparative analysis of several widely-used sparse matrix formats: Tiled-CSL (used in Flash-LLM [86]), CSR (used by Sputnik [24] and other CUDA-core SpMM implementations), and SparTA [100].

Tiled-CSL stores non-zero elements in tiles using two arrays: *NonZeros*, which holds 32-bit (16-bit × 2) values (weight and location), and *TileOffsets*, which tracks the starting offset of each tile. The storage overhead for Tiled-CSL can be calculated as:

$$Stor_{Tiled-CSL} = 4B \times NT + 4B \times NNZ, \tag{2}$$

where $NT$ is the number of tiles. With each non-zero element requiring a 16-bit index, the indexing overhead is comparable to the data size itself.

The CSR format is a traditional sparse representation that stores non-zero elements along with their column indices. The storage overhead for CSR is:

$$Stor_{CSR} = (2B + 4B) \times NNZ + 4B \times (M + 1). \tag{3}$$

In CSR, the 32-bit indices used for column storage can result in significant overhead.

SparTA uses a composable format, dividing the matrix into two parts: one following the 2:4 sparsity pattern and another

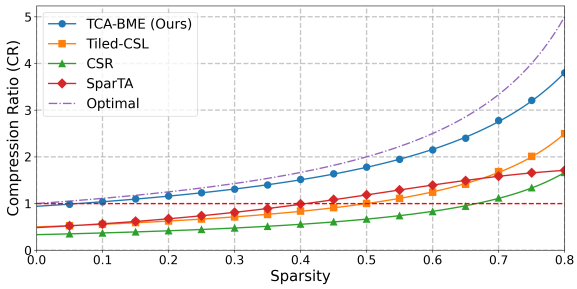


**Figure 3.** Compression Ratio (CR) across varying sparsity levels for different sparse matrix formats.

using a CSR-like format. It reduces overhead by storing non-zero elements with 2-bit indices within 2:4 blocks. For blocks containing more than two non-zero elements, SparTA utilizes a CSR-like storage format for the excess non-zero elements. As a result, the actual storage overhead of SparTA depends on the distribution of non-zero elements within the matrix. Assuming a uniform distribution of non-zero elements, the expected number of non-zero elements in a block requiring CSR storage can be expressed as:

$$E_{CSR\_nnz} = \left(\frac{M \times K}{4}\right) \times (4 \times (1-s)^3 \times s + 2 \times (1-s)^4). \quad (4)$$

The storage overhead for SparTA can thus be written as:

$$Stor_{SparTA} = (2B + \frac{B}{4}) \times \frac{M \times K}{2} + Stor_{CSR}(E_{CSR\_nnz}). \quad (5)$$

Figure 3 shows the CR trends across various sparsity levels, using a representative scale of $M = K = 4096$ for LLM model weights. CSR and Tiled-CSL have a CR below 1 at sparsities under 50%, meaning their indexing overhead outweighs the pruning gains. SparTA performs better, with a CR slightly above 1 at 50%, but still falls short of the theoretical optimal (indicated by the dotted line) due to reliance on CSR-like indexing. In contrast, our TCA-BME format (the blue line) consistently achieves a CR above 1, even at lower sparsity levels. This is due to its advanced bitmap-based indexing technique, which significantly reduces the overhead of storing non-zero element positions.The details of TCA-BME are discussed in Section 4.2.

#### 3.2.2 Gaps in Computation Efficiency.

To analyze the impact of indexing overhead on computational efficiency, we employ the Roofline model [84] and focus on the Compute Intensity (CI) of both dense matrix multiplication (GEMM) and sparse matrix multiplication (SpMM).

**Compute Intensity (CI).** CI is a critical metric for understanding the balance between computational and memory-bound operations in matrix calculations. It is defined as the ratio of floating-point operations (FLOPs) to memory accesses. For GEMM, CI is calculated as:

$$CI_{\text{GEMM}} = \frac{M \times N}{M + N}. \quad (6)$$

For SpMM, CI is affected by the compression ratio (CR), reflecting the reduction in storage due to sparsity. Additionally, indexing overhead can further reduce the effective CI. To account for this, we define CI for SpMM as:

$$CI_{\text{SpMM}} = \frac{M \times N}{\frac{M}{\text{CR}} + N}. \quad (7)$$

To measure the performance gap introduced by indexing overhead, we compare the actual compute intensities of these sparse formats with an optimal CI, which assumes negligible indexing overhead. The optimal CI for SpMM can be defined as:

$$CI_{\text{Optimal}} = \frac{M \times N}{M \times (1-s) + N}, \quad (8)$$

where $s$ denotes the sparsity level. The optimal CI represents the theoretical upper bound on performance, reflecting the maximum compute intensity that could be achieved if the overhead of indexing non-zero elements were negligible.

**Roofline Model Analysis.** Figure 4 illustrates that both GEMM and SpMM operations predominantly reside in the memory-bound region of the Roofline model across varying sparsity levels and matrix sizes. In this region, performance scales linearly with CI. Theoretically, due to the reduction in global memory access and the enhancement in CI brought by sparsity, SpMM can achieve linear speedup compared to GEMM, as indicated by the star (∗) in Figure 4. However, the actual CI for SpMM is influenced by CR, reflecting the reduction in memory access cost due to sparsity.

As CR increases, the global memory access cost decreases, leading to a higher CI, which translates into improved performance. Therefore, formats with higher CR values theoretically achieve better performance compared to formats with lower CR. This relationship is clearly visualized in the Roofline model, where our TCA-BME moves closer to the compute-bound region due to its efficient bitmap-based indexing, which increases the CR and, consequently, the CI.

In contrast, formats like CSR and Tiled-CSL, which have lower CR values due to their traditional indexing schemes, suffer from higher memory access costs. The indexing overhead reduces their effective CI, which results in a significant performance gap when compared to the optimal CI, as shown by the star (∗) in Figure 4.

Our analysis identifies indexing overhead as a major factor limiting the storage and performance benefits of pruning in practical applications. By addressing this overhead, storage requirements can be greatly reduced, bringing performance closer to theoretical gains. This challenge serves as the key motivation behind SpInfer's design.

## 4 Design of SpInfer

### 4.1 Design Overview

SpInfer is a high-performance framework designed to accelerate LLM inference on GPUs by leveraging sparse matrix

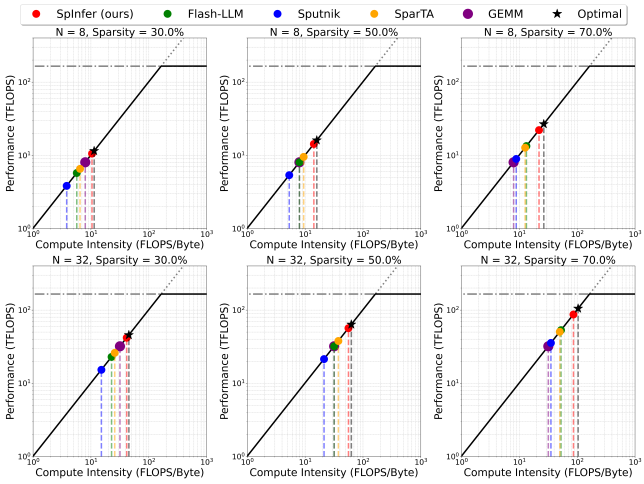


**Figure 4.** Roofline comparison of various SpMM implementations against GEMM at varying sparsities and batch sizes.

multiplication. Figure 5 illustrates the system overview of SpInfer. With advanced unstructured pruning algorithms, SpInfer reduces model size without compromising accuracy. The framework's foundation lies in its Tensor-Core-Aware Bitmap Encoding (TCA-BME) scheme, which efficiently compresses sparse weight matrices with minimal indexing overhead. At its core, SpInfer features a highly optimized SpMM kernel that improves efficiency through a combination of efficient data movement, Shared Memory Bitmap Decoding (SMDB), and a fine-grained asynchronous pipeline. These designs enable SpInfer to significantly reduce latency and memory consumption in large-scale LLM inference while maintaining model accuracy, without the need for additional fine-tuning.

## 4.2 Tensor-Core-Aware Bitmap Encoding

As the foundation of SpInfer, we develop a novel Tensor-Core-Aware Bitmap Encoding (TCA-BME) scheme to efficiently store sparse weight matrices with low sparsity. This format is designed to minimize memory footprint (increasing compression ratio) while maintaining computational efficiency, laying the groundwork for subsequent high-performance sparse matrix multiplication on Tensor Cores.

**4.2.1 Tiling Design.** TCA-BME employs a multi-level tiling design, partitioning the weight matrix into tiles of varying granularity to align with different levels of GPU hardware. As shown in Figure 6, this design encompasses three key abstraction levels: *BitmapTile (BT)*, *TCTile (TT)*, and *GroupTile (GT)*, each corresponding to distinct computational units within the GPU hardware.

The innermost abstraction is the *BitmapTile*, which serves as the smallest granular unit in the TCA-BME format. With dimensions of $BT_H \times BT_W$ set to $8 \times 8$, this design targets the minimum computational unit of Tensor Cores, namely an $8 \times 8$ matrix block. An additional advantage of aligning the *BitmapTile* dimensions with this unit is the ability to utilize CUDA's natively supported *uint64_t* data type as a 64-bit bitmap, indicating the positions of non-zero elements within the *BitmapTile*. Each bit in the bitmap corresponds to a specific element in the *BitmapTile*, with set bits representing non-zero values.

The intermediate level consists of *TCTiles*, with dimensions $TT_H \times TT_W$, comprising $2 \times 2$ *BitmapTiles* for a total size of 16×16. This *TCTiles* abstraction corresponds to the matrix shape of Tensor Core `mma` instructions at the PTX level. For FP16 precision, two relevant PTX-level instructions are available: `mma.m16n8k8` and `mma.m16n8k16`. Micro-benchmark results indicate that `mma` instructions with larger shapes offer higher throughput, leading us to opt for the `mma.m16n8k16` instruction and align the *TCTiles* dimensions with its m×k completely. Within a *TCTiles*, the 2×2 *BitmapTile* are arranged in column-major format, ensuring consistency with the order of the four `Ra` registers in the `mma` instruction. Specifically, the top-left *BitmapTile* corresponds to `Ra0`, bottom-left to `Ra1`, top-right to `Ra2`, and bottom-right to `Ra3`. This column-major storage approach facilitates subsequent decoding processes without complex coordinate transformations, reducing online overhead.

The outermost level is the *GroupTile*, with dimensions $GT_H \times GT_W$, encompassing multiple *TCTiles* and corresponding to the thread block level. *TCTiles* within *GroupTiles* are also stored in column-major order. Thread blocks are responsible for loading and processing *GroupTiles*, while warps within the thread block handle computations for *TCTiles* within the *GroupTile*. The *GroupTiles* themselves are stored in row-major order.

**4.2.2 Storage.** The TCA-BME format employs three arrays to represent sparse weight matrices efficiently. The *GTileOffset* array records the starting offset positions of each *GroupTile* within the sparse matrix, enabling rapid localization and parallel processing of different *GroupTiles*. The *Values* array stores all non-zero elements, arranged in a nested order of *GroupTile*, *TCTiles*, and *BitmapTile*. The *Bitmap* array contains bitmap value for all *BitmapTiles*, with each *BitmapTile* represented by a 64-bit integer, where each bit indicates whether the corresponding element is non-zero. Specifically, we define $NGT = (M/GT_H) \times (K/GT_W)$ as the number of *GroupTiles*, $NBT = (M/BT_H) \times (K/BT_W)$ as the number of *BitmapTiles*, and $NNZ = M \times K \times (1 - s)$ as the number of non-zero elements, where s denotes the matrix sparsity. The *GTileOffset* array utilizes 32-bit integers (4B) to represent offsets, with a size of $4B \times (NGT + 1)$, including an additional element to mark the end of the last *GroupTile*. The Value array employs half-precision floating-point numbers (FP16) to store non-zero elements, occupying 2B per element, with a total size of $2B \times NNZ$. In the Bitmap array, each *BitmapTile* corresponds to a 64-bit integer (8B), resulting in a total size

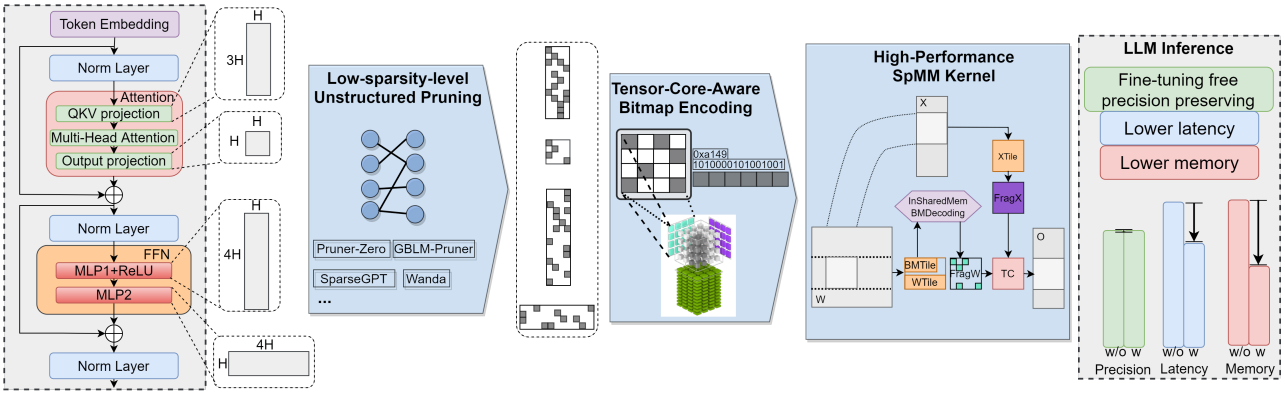


**Figure 5.** System overview of SpInfer.

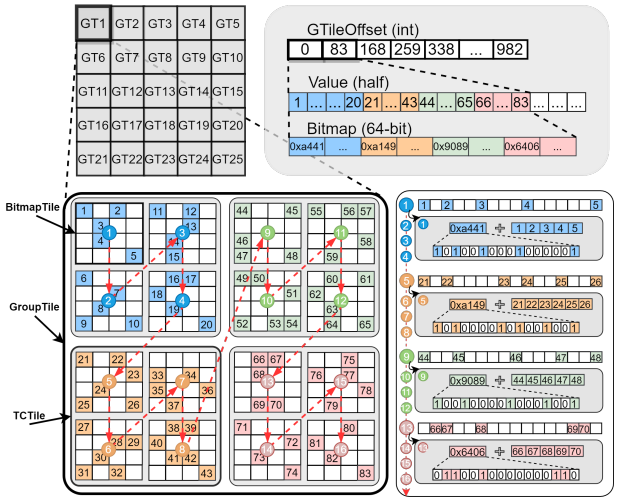


**Figure 6.** Tensor-Core-Aware Bitmap Encoding. BitmapTile is actually 8×8, shown as 4×4 for illustration.

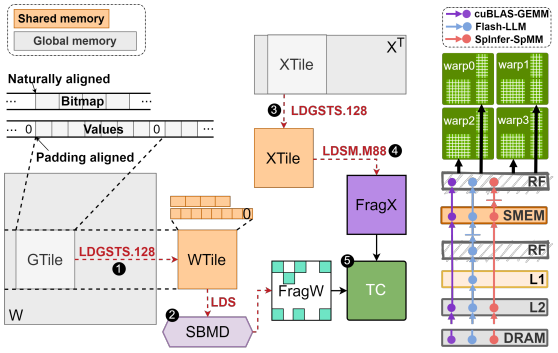


**Figure 7.** Data movement and instruction pipeline.

of $8B \times NBT$. Consequently, the total storage overhead for the TCA-BME format can be calculated as:

$$Stor_{TCA-BME} = 4B \times (NGT+1) + 8B \times NBT + 2B \times NNZ. \quad (9)$$

TCA-BME maintains an effective compression ratio (CR > 1) even at low sparsity levels and shows rapid CR growth as sparsity increases (Figure 3). This superior performance stems from its efficient bitmap-based indexing scheme, which significantly reduces indexing overhead, especially at low to moderate sparsities (30%-70%).

### 4.3 High-Performance SpInfer-SpMM Kernel Design

**4.3.1 Workflow.** The workflow of SpInfer-SpMM kernel is illustrated in Figure 7. A detailed pseudocode representation is provided in Algorithm 1. Our kernel adopts similar tiling-based strategies to CUTLASS GEMM with splitK parallelism [79] to efficiently distribute computation across thread blocks, where each block processes a portion of the K-dimension independently. During each iteration, a thread block executes five key operations. ❶ *GTile* Loading. Threads within a block collaboratively load a *GTile* (*GroupTile*) from global memory into a *WTile* in shared memory. ❷ *WTile* Decoding. The *WTile* is decoded from shared memory into registers through a crucial technique named Shared Memory Bitmap Decoding (SBMD). This step translates the compact bitmap representation of the sparse matrix into a correct distribution in register file that's ready for Tensor Core computation, all within the high-speed register file. ❸ *XTile* Loading. The corresponding *XTile* from the dense input matrix $X^T$ is loaded from global memory into shared memory. ❹ *XTile* Register Transfer. The *XTile* data is then transferred from shared memory to registers and be arranged for the TC computations. ❺ Tensor Core Computation (TCC). The Tensor Cores then perform the matrix multiplication between the decoded sparse *WTile* and the dense *XTile* , both now residing in registers.

**4.3.2 Efficient Data Movement.** In steps ❶ and ❸, we employ the `LDGSTS.128` asynchronous vectorized memory access instruction to improve global memory bandwidth utilization. Introduced from the Ampere architecture [55], `LDGSTS` eliminates the need for intermediate staging of data

**Algorithm 1** SpInfer-SpMM kernel pseudo code

**Input:** SparseMatrix $W$ (TCA-BME format), Matrix $X$, Split_K
**Output:** Matrix $Y$ in $ReductionWorkspace$

$BatchID = blockIdx.y/(M/TILE\_M)$
$TileY = blockIdx.y\%(M/TILE\_M), TileX = blockIdx.x$
$NumIter = CalculateIterations(BatchID, Split\_K)$ ▷ K-dim iterations
__shared__ *ValueBuffer[max_nnz_per_tile]* ▷ Sparse buffer
__shared__ *XTileBuffer[2][TILE_K][TILE_N]* ▷ Double buffer
__shared__ *BitmapBuffer[2][TILE_M][TILE_K]* ▷ Double buffer
// Pre-loop initialization
LoadBitmapAndSparse(*BitmapBuffer, ValueBuffer, W*)
LoadDenseToShared(*XTileBuffer*, $X + BatchID * TILE\_K$)
cp.async.commit() ▷ Commit for dense
cp.async.wait_group(0)
$W_{frag}$ = SharedMemoryBitmapDecoding(*ValueBuffer, BitmapBuffer*)
// Main computation loop
**for** $k \leftarrow 0$ **to** $NumIter - 2$ **step** 1 **do**
  // Start prefetch for next iteration
  LoadBitmapAndSparse(*BitmapBuffer, ValueBuffer, W + offset*)
  cp.async.commit() ▷ Commit for bitmap/sparse
  LoadDenseToShared(*XTileBuffer, X + offset*)
  cp.async.commit() ▷ Commit for dense
  // Current iteration computation
  $X_{frag}$ = LoadDenseToRegisters(*XTileBuffer*)
  $Y_{frag}$ = TensorCoreCompute($W_{frag}, X_{frag}, Y_{frag}$)
  // Prepare sparse data for next iteration
  cp.async.wait_group(1) ▷ Wait for bitmap/sparse
  $W_{frag}$ = SharedMemoryBitmapDecoding(*ValueBuffer, BitmapBuffer*)
  cp.async.wait_group(0) ▷ Wait for dense
  __syncthreads()
**end for**
// Epilogue: process final iteration
$X_{frag}$ = LoadDenseToRegisters(*XTileBuffer*)
$Y_{frag}$ = TensorCoreCompute($W_{frag}, X_{frag}, Y_{frag}$)
StoreResults($ReductionWorkspace, Y_{frag}$)

through the L1 cache and register file, thereby reducing register file bandwidth consumption. The `128` indicates that each thread reads 128 bits of data from global memory (e.g., 8 half-precision operands). To enable 128-bit vectorization in step ❶, the `Value` array within each *GTile* is preprocessed with padding, ensuring that the starting address of each *GTile* is aligned to an 8-byte boundary. In step ❹, we utilize the `LDSM.M88` instruction (corresponding to `ldmatrix.x4` at the PTX level) to load the *XTile* from shared memory. This instruction allows a warp to load a 16x16 matrix tile from shared memory and automatically arranges the data in registers according to the layout required for ❺ TC computations. For step ❷, we use generic `LDS` instructions to decode the *WTile* from shared memory into registers.

Figure 7 illustrates the data movement path during the fetching of the weight matrix (`W`) in cuBLAS-GEMM, Flash-LLM, and SpInfer-SpMM. cuBLAS represents the ideal case, where the use of `LDGSTS` allows data to bypass both the L1 cache and the register file, directly storing it in shared memory. In contrast, Flash-LLM first loads the Tiled-CSL format's *NonZeros* Array into the register file using `LDG.128`, and subsequently unpacks it into shared memory. SpInfer-SpMM directly loads the *GTile* into shared memory via `LDGSTS.128`, achieving a data movement path that closely approximates the ideal cuBLAS case. This approach also conserves SM internal bandwidth by avoiding the roundtrip through the register file, which incurs additional overhead in Flash-LLM.

#### 4.3.3 Shared Memory Bitmap Decoding (SMBD).

The SMBD mechanism is an essential optimization of the SpInfer-SpMM kernel, designed to efficiently decompress the bitmap-compressed *WTile* into the register file, ensuring proper layout for subsequent Tensor Core computations. This technique leverages bitmaps to represent the sparsity pattern of the matrix, while the non-zero values are stored in a compressed format, allowing for both efficient memory usage and high-performance matrix operations.

**Register Distribution.** In warp-level Tensor Core operations, a warp (32 threads) collectively processes fragments of operand matrices. Each thread in the warp holds part of the operand matrix, and the distribution of these fragments across the threads must be done carefully to ensure correct execution of the `mma` instructions. For half-precision computations, we employ the `mma.m16n8k16` instruction, which operates on $16 \times 16$ matrix fragments. Figure 8(a) illustrates the matrix fragment distribution, where each thread holds two half-precision values per 32-bit register (.f16x2). Four such registers (`Ra0`, `Ra1`, `Ra2`, and `Ra3`) are needed to store the entire fragment in each thread. These registers are populated via bitmap decoding, which extracts non-zero values from the compressed format.

**Two-Phase Decoding Process.** As described in Section 4.2, the *TCTile* consists of four *BitmapTiles*, each corresponding to one register (`Ra0`, `Ra1`, `Ra2`, and `Ra3`). A *BitmapTile* is a 64-bit value that encodes the sparsity pattern of an $8 \times 8$ matrix fragment, with each bit indicating whether a non-zero value exists at the corresponding location.

A challenge arises from the compressed storage of the non-zero values, which means that the exact offset for each thread to load its values is not explicitly stored. To calculate the correct offset, we rely on two key operations. ❶ **PopCount**, which is implemented using Nvidia GPU's integer intrinsic `__popcll`, counts the number of `1` bits in a 64-bit bitmap. This count represents the number of non-zero values in the corresponding *BitmapTile*. By accumulating the result of PopCount across *BitmapTile*, the correct starting offset in the compressed `Values` array is determined dynamically for each tile. This allows the warp to efficiently load non-zero values without storing explicit offsets in global memory. ❷ **MaskedPopCount.** In addition to calculating the offset for the entire *BitmapTile*, each thread needs to determine how many non-zero values precede its lane within the bitmap. The MaskedPopCount operation counts the number of `1` bits before the current thread's lane ID, as depicted in Figure 8(b). This operation is crucial for calculating the correct offset within the compressed `Values` array for each thread to load its non-zero values. The detailed implementation is presented in Algorithm 2, which demonstrates this efficient bit-counting process.

The bitmap decoding process is performed in two phases, as shown in Figure 8 (c). ❶ **Phase I (Decoding `a0`)**. In the

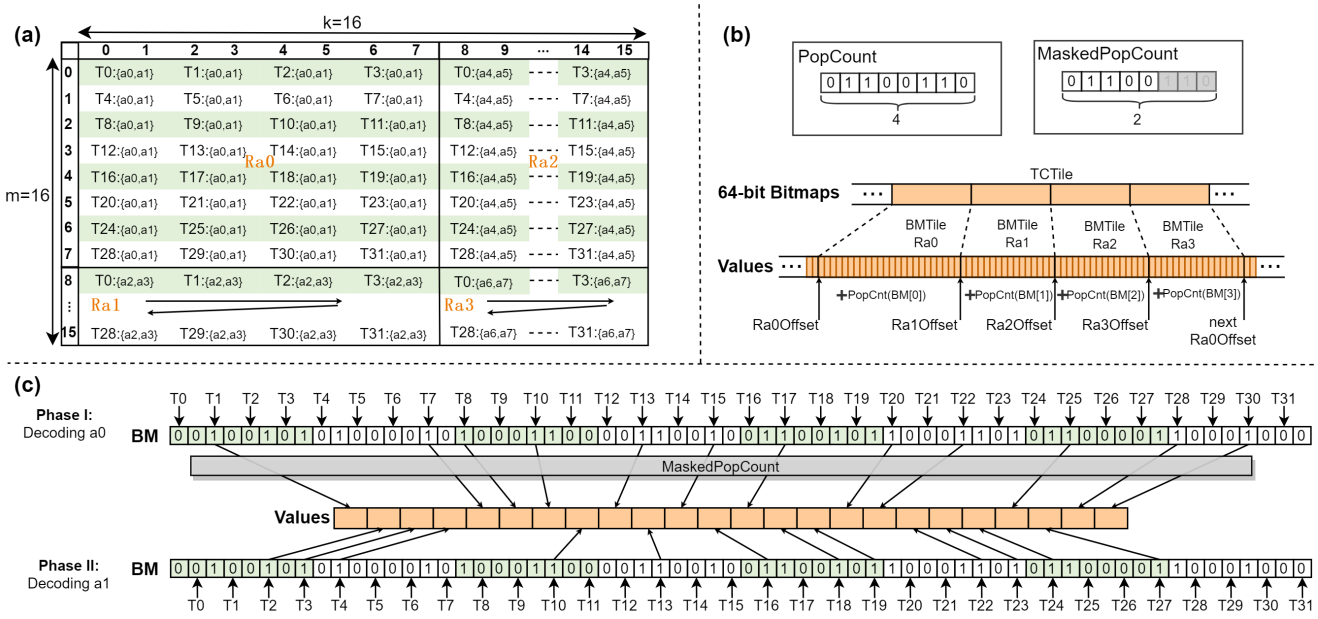


**Figure 8.** Shared Memory Bitmap Decoding. (a) Register distribution of Tensor Core *mma* instruction. (b) PopCount and online offset calculation. (c) The two-phase bitmap decoding process.

**Algorithm 2** MaskedPopCount pseudo code

**Input:** Bitmap $b$, Thread Lane ID $l$
**Output:** Count of preceding ones *count*
1: $offset = l \times 2$ ▷ Calculate base offset
2: $mask = (1 \ll offset) - 1$ ▷ Generate preceding mask
3: $count = PopCount(b \& mask)$ ▷ Count ones before offset
4: **return** *count*

first phase, each thread decodes the first half-precision value (`a0`) in its 32-bit register. The thread with ID `i` examines the `(2i)`-th bit of the bitmap. If this bit is set to 1, the thread uses the MaskedPopCount to calculate how many non-zero values exist before its position and loads the corresponding value from the compressed `Values` array. If the bit is `0`, the thread loads a zero value into its register. **❷ Phase II (Decoding a1)**. In the second phase, each thread decodes the second half-precision value (`a1`) from the same 32-bit register. The thread with ID `i` examines the `(2i+1)`-th bit of the bitmap to determine whether a non-zero value exists at that position. However, no additional MaskedPopCount is required in Phase II. The result from Phase I is reused. Specifically, if the first value (`a0`) was non-zero, the offset is incremented by one to load the second value (`a1`). This reuse of the MaskedPopCount result from Phase I minimizes the number of pop count operations, enhancing performance.

By using the intrinsic PopCount and MaskedPopCount operations, we efficiently decode the compressed matrix fragment in parallel across all threads, ensuring that each thread accesses the correct non-zero values without the need for explicit storage of offsets.

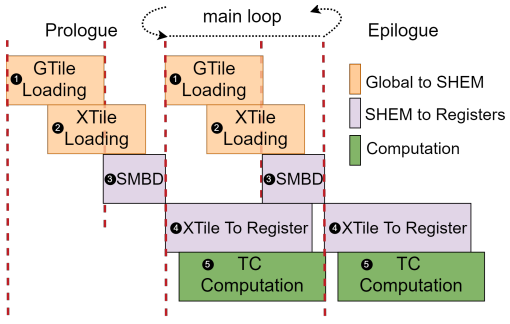


**Figure 9.** Schematic representation of the asynchronous pipeline design. The pipeline depth is 2.

#### 4.3.4 Asynchronous Pipeline Design.

We develop a fine-grained asynchronous pipeline to further optimize the performance of the SpInfer-SpMM kernel. As illustrated in Figure 9, this pipeline enhances TC utilization by maximizing the overlap between memory transfers and TC computations.

**Double Buffering Mechanism.** Double buffering forms the cornerstone of the pipeline design. We implement two separate shared memory buffers for *GTiles* and *XTiles*. This architecture enables prefetching of data for the next iteration into shared memory while computing with the current iteration's data, thereby hiding memory load latency and improving overall throughput. Specifically, in each iteration, the current *GTile* and *XTile* data reside in one shared memory buffer, while the next set of data is asynchronously prefetched (using `cp.async`) into the alternate buffer. As mentioned in Section 4.3.1, our workflow design allows for

the use of LDGSTS asynchronous instructions for both W and X matrices, enabling us to implement a double buffering mechanism similar to that used in cuBLAS.

**Fine-grained Asynchronous Group Management.** To further enhance efficiency, we employ two separate **cp.async groups** to manage the loading of *GTiles* and *XTiles* independently. This fine-grained control enables greater concurrency across different pipeline stages. Our design incorporates two key overlapping strategies. Once the *GTile* loading is complete, the Shared Memory Bitmap Decoding (SMBD) process begins immediately, running concurrently with the ongoing *XTile* loading. Since *XTile* loading and SMBD are independent operations, their parallel execution can effectively hide the latency of SMBD, preventing it from becoming a performance bottleneck. Furthermore, after issuing Tensor Core computation instructions for the current tile, the SMBD process for the next tile begins immediately. The bit manipulation and counting operations in SMBD, which run on CUDA cores, are independent of Tensor Core instructions. This interleaving of SMBD with Tensor Core computations increases Instruction Level Parallelism (ILP) and optimizes hardware resource utilization by keeping both CUDA cores and Tensor Cores active, reducing pipeline stalls and improving throughput.

## 5 Performance Evaluation

We assess the performance of SpInfer at two levels: the SpMM kernel level and the end-to-end framework level. The experiments are run on two platforms. ❶ Intel Xeon Platinum 8352V CPU (2.10GHz) with 4 NVIDIA RTX4090 GPUs (Ada Lovelace, Compute Capability 8.9, 24 GB memory per GPU), connected via PCIe with a bandwidth of 30.5 GB/s. ❷ Intel Xeon Gold 6133 CPU (2.50GHz) with 4 NVIDIA A6000 GPUs (Ampere, Compute Capability 8.6, 48 GB memory per GPU), connected via pairwise NVLink. The code is compiled using GCC 9.4.5 and NVCC 12.1. For kernel-level evaluation, Nsight Compute [61] is used to measure the precise execution times. For end-to-end evaluation, the inference process is run 100 times, and the average wall-clock time is recorded.

### 5.1 Kernel Performance Comparison

**Datasets.** We evaluate SpInfer-SpMM using a diverse set of weight matrix sizes derived from prominent LLM models. These include the OPT-Series (13B, 30B, 66B, and 175B) [96], the LLaMA2-Series (7B, 13B, and 70B) [80], the LLaMA3-Series (8B and 70B) [17], Qwen2 (7B and 72B) [90], and the Mixtral-8×7B MoE model [35].

**Baselines.** SpInfer-SpMM is compared against several key baselines, including: ❶ cuSPARSE v12.1, the widely used vendor-provided SpMM library [53]; ❷ Sputnik [24], a state-of-the-art CUDA-core-based SpMM optimized for sparsity in deep learning; ❸ SparTA [100], the first approach to leverage sparse Tensor Cores for unstructured SpMM; ❹ Flash-LLM [92], a state-of-the-art Tensor-Core-based SpMM designed for sparse LLM inference; and ❺ Tensor-Core-based cuBLAS, the counterpart used in dense LLM inference. Additionally, we compare SpInfer with advanced Tensor-Core-based SpMM for scientific workloads, including ❻ SMaT [64]. The evaluation is conducted at sparsity levels between 40% and 70%, which represent the optimal sparsity range targeted by cutting-edge LLM pruning techniques.

**Results.** Figure 10 depicts the measured performance on RTX4090 and A6000. The speedup values are normalized to Tensor-Core-Based cuBLAS (cuBLAS_TC), represented by the red dashed line. SpInfer consistently delivers superior speedups against both dense and sparse implementations. On the RTX4090, SpInfer achieves an average speedup of 1.79× over cuBLAS, with average speedups of 18.14×, 2.55×, 1.67×, and 1.56× against cuSPARSE, Sputnik, SparTA, and Flash-LLM, respectively. On the A6000, similar trends are observed, with SpInfer achieving an average speedup of 1.51× over cuBLAS and outperforming cuSPARSE by up to 24.80×.

At lower sparsity levels (40%), SpInfer is the only method capable of consistently outperforming cuBLAS, achieving a 1.46× average speedup and surpassing cuBLAS on 94.44% of matrices. At the critical 50% sparsity level, SpInfer maintains its lead with an average speedup of 1.66× over cuBLAS, outperforming all other kernels on 96.30% of test cases. Competing methods like SparTA and Flash-LLM offer only marginal improvements over cuBLAS, with 1.01× and 1.00× speedups, respectively.

As sparsity increases to 70%, where SpMM typically becomes more advantageous, SpInfer's performance further excels, achieving a 1.90× speedup over cuBLAS and outperforming it in 100% of test cases. In comparison, SparTA and Flash-LLM achieve more modest gains (1.16× and 1.22× speedups). These results reflect SpInfer's ability to handle low to moderate unstructured sparsity, which often challenges traditional sparse kernels.

Figure 11 shows the performance comparison between SpInfer and SMaT. At 50% sparsity, SpInfer outperforms SMaT with a 2.12× speedup. SMaT only surpasses SpInfer at extreme sparsity levels above 99.7%, where its design optimizes performance by skipping zero blocks in highly sparse scientific matrices. However, in the low to moderate sparsity ranges typical of LLM inference, there are few blocks that can be skipped, limiting SMaT's advantage.

**Micro-Analysis.** To further explain the performance gains of SpInfer, we conduct a detailed micro-level analysis of the SpMM kernels. Key indicators examined include register allocation, DRAM bytes read, bandwidth utilization, bank conflicts, and Tensor Core utilization. We collect these indicators through Nsight Compute [61]. The results are shown in Figure 12. SpInfer consumes the fewest registers compared to

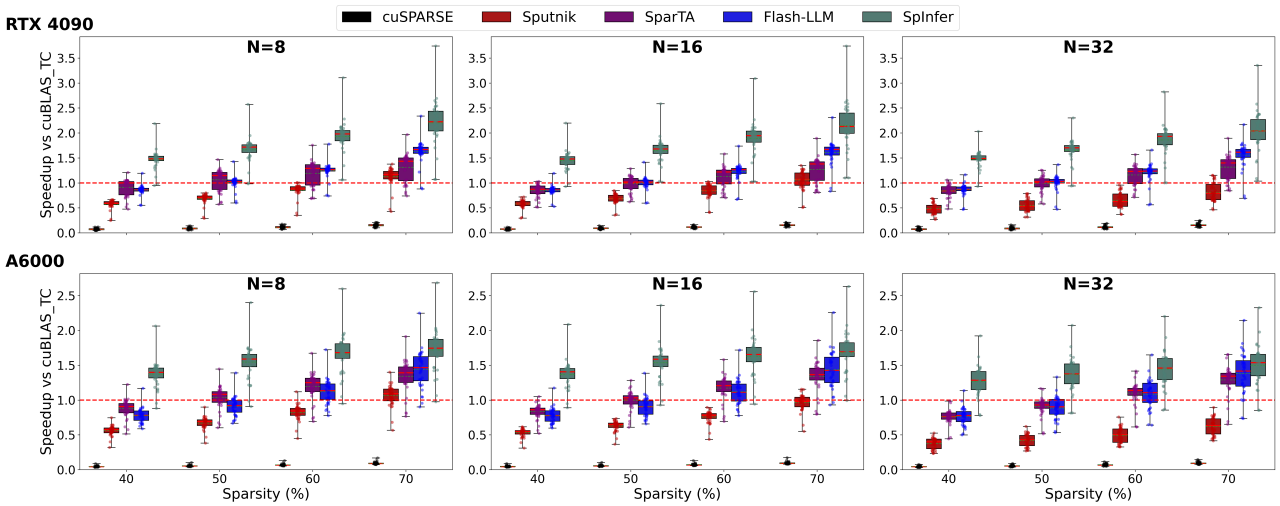


**Figure 10.** SpMM kernel performance comparison on RTX4090 and A6000 GPUs. $N$ denotes the batch size.

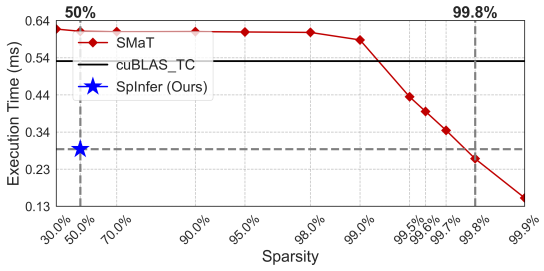


**Figure 11.** Comparison with SpMM for scientific workloads.

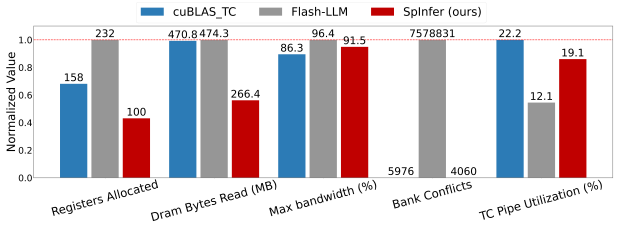


**Figure 12.** Micro-level comparison of SpInfer against cuBLAS_TC and Flash-LLM across key metrics.

other methods. This efficiency is achieved by directly decoding sparse data in shared memory, avoiding the need for additional registers to store sparse data. The lower register usage allows for higher GPU occupancy, enabling more threads to run concurrently and improving overall computational efficiency. Additionally, SpInfer significantly reduces time-consuming DRAM access, minimizing the volume of data transferred between global memory and compute units. This reduction is primarily due to the efficiency of the TCA-BME format, which optimizes data storage and access patterns.

Furthermore, SpInfer excels in minimizing shared memory bank conflicts. In contrast, Flash-LLM requires consecutive threads to write sparse data to specific locations in shared memory. Due to the inherent randomness of sparse data, this often leads to unavoidable shared memory bank conflicts during write operations. SpInfer's design avoids such conflicts.

Finally, SpInfer achieves higher Tensor Core pipeline utilization than Flash-LLM, due to the efficient SMBD and asynchronous pipeline design. This optimized pipeline ensures that data transfer and computation are overlapped effectively, allowing Tensor Cores to be better leveraged.

| Optimizations | | Duration↓ | Max | Issue Slot | Warp Cycles | TC Pipe |
|---|---|---|---|---|---|---|
| SMBD | AsyncPipe | unit:ms | BW(%)↑ | Busy(%)↑ | Per Inst↓ | UTIL(%)↑ |
| ✓ | ✓ | 303.1 | 91.5% | 37.6% | 9.1 | 19.1% |
| | ✓ | 333.5 | 28.6% | 9.1% | 9.6 | 4.1% |
| ✓ | | 309.2 | 89.3% | 35.9% | 9.5 | 18.7% |

**Table 1.** Kernel-level ablation study. BW: Bandwidth.

**Ablation Study.** To quantify the impact of key optimizations in SpInfer, we conduct an ablation study by selectively removing the SMBD and asynchronous pipeline (AsyncPipe) optimizations, and analyze their effects on performance. The results are shown in Table 1. Without SMBD, kernel execution time increases by 10.03%, with a 68.78% drop in bandwidth utilization and a 75.77% reduction in issue slot activity. Furthermore, Tensor Core utilization decreases by 78.41%, showing that SMBD is crucial for optimizing memory access and ensuring efficient hardware usage. When AsyncPipe is removed, execution time increases by 1.98% and Tensor Core utilization drops by 2.00%, indicating that this optimization plays a key role in overlapping memory transfers with computation, improving overall efficiency.

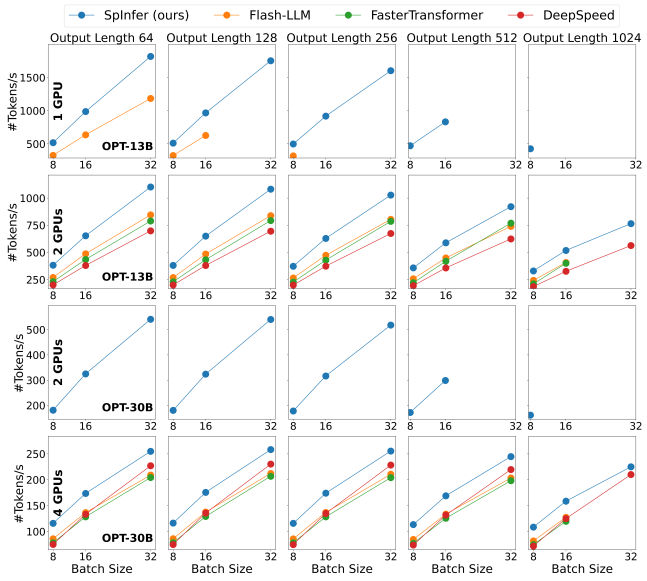


**Figure 13.** End-to-end inference performance of OPT-13B and OPT-30B on RTX4090 GPUs.

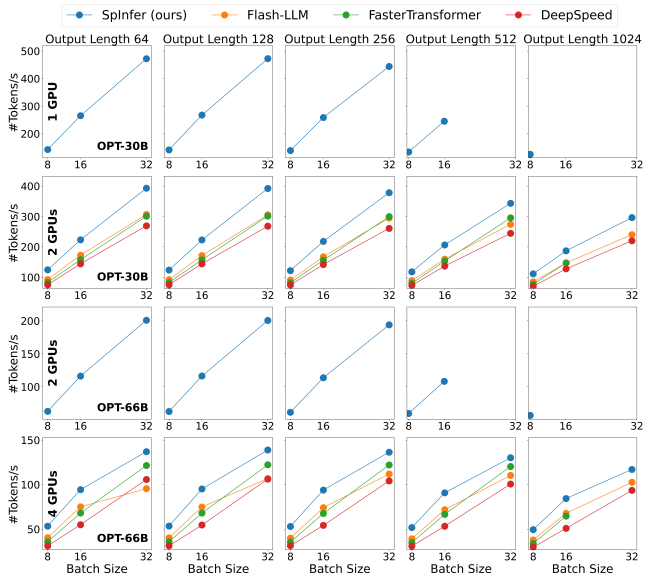


**Figure 14.** End-to-end inference performance of OPT-30B and OPT-66B on A6000 GPUs.

### 5.2 End-to-end LLM Inference

**Baselines and settings.** We compare SpInfer against state-of-the-art frameworks, including Flash-LLM (FL), DeepSpeed (DS) [69], and FasterTransformer (FT) [56] for sparse and dense LLM inference. Models used include OPT-13B, OPT-30B, and OPT-66B, providing a wide range of sizes. With the advanced Wanda algorithm [77], the model sparsity is set at 60%, allowing OPT-13B to maintain a perplexity of 15.9 on the WikiText dataset. The precision of SpInfer relies on and is guaranteed by current LLM pruning algorithms. Experiments are conducted with batch sizes of 8, 16, and 32 on 1, 2, and 4 GPU configurations to assess scalability and efficiency across different parallel processing scenarios. Output lengths are set to 64, 128, 256, 512, and 1024 tokens, allowing performance analysis under varying computational loads during inference.

**Results.** The end-to-end inference results for the OPT models on RTX4090 and A6000 GPUs are presented in Figures 13 and 14, respectively. SpInfer consistently outperforms the baseline frameworks, showcasing significant improvements in both latency and memory efficiency.

Regarding **latency**, SpInfer exhibits notable speedups over the baseline frameworks. On RTX4090, SpInfer achieves average speedups of 1.35×, 1.42×, and 1.49× compared to Flash-LLM, FT, and DS, respectively. Similar trends are observed on A6000, where the corresponding speedups are 1.29×, 1.36×, and 1.55×. The maximum speedup of 1.58× over Flash-LLM on RTX4090 occurs in the 1-GPU configuration with a batch size of 32, where SpInfer processes over 1817.02 tokens/second, compared to Flash-LLM's 1183.58 tokens/second. In the 2-GPU configuration with OPT-13B, SpInfer achieves an average speedup of 1.34× over Flash-LLM, while in the 4-GPU OPT-30B setup, the speedup slightly decreases to 1.28×. Although the relative speedups tend to diminish as the number of GPUs and model size increase—primarily due to the rising communication overhead associated with model parallelism—SpInfer continues to be the most efficient solution.

In terms of **memory efficiency**, SpInfer outperforms other frameworks, particularly in scenarios where they encounter out-of-memory (OOM) issues. Leveraging the TCA-BME format, SpInfer achieves sparsity-aligned memory reduction in model weights, thereby fundamentally improving storage efficiency. For instance, when performing OPT-13B inference with a batch size of 16 and a sequence length of 256, SpInfer's 60%-sparsity model consumes merely 14.4 GB memory, achieving a 47.5% reduction compared to the dense baseline's 27.4 GB requirement. This memory compression becomes particularly crucial for larger batch sizes and longer output sequences where competing frameworks exhibit limitations. With OPT-13B on a single RTX4090 GPU and a batch size of 8, SpInfer can support up to 1024 output tokens, whereas Flash-LLM is limited to a maximum of 256 tokens. Similarly, with OPT-30B on 2 RTX4090 GPUs, Flash-LLM encounters OOM errors across all batch sizes and output lengths, while SpInfer can handle up to 512 tokens with a batch size of 16, and up to 1024 tokens with a batch size of 8. This trend is also evident when inferring the OPT-66B model on 2 A6000 GPUs, where SpInfer demonstrates superior memory management compared to other frameworks.

The reason behind these advantages lies in SpInfer's superior SpMM performance and the high compression ratio

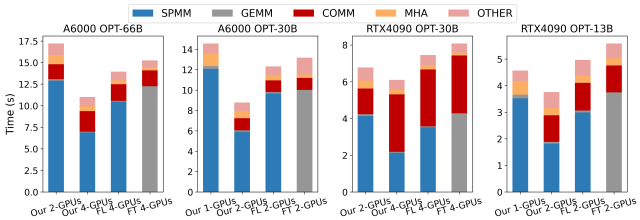


**Figure 15.** Breakdown of end-to-end inference time. FL: Flash-LLM. MHA: Multi-Head Attention. COMM: Inter-GPU Communication.

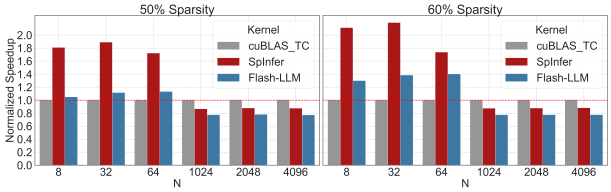


**Figure 16.** Performance comparison under small and large N settings. $M = 28672$ and $K = 8192$.

of its TCA-BME format, which effectively reduces memory requirements almost linearly with sparsity. This combination makes SpInfer more versatile and scalable for real-world LLM inference scenarios.

**Breakdown Analysis.** To further explain the performance gains of SpInfer, we use Nsight Systems [62] to break down the execution time, as shown in Figure 15. The primary time consumption for both SpInfer and Flash-LLM is spent on SpMM operations, while for FasterTransformer it's GEMM. Under equivalent configurations, SpInfer's SpMM takes significantly less time compared to Flash-LLM's SpMM and FasterTransformer's GEMM.

Furthermore, due to SpInfer's superior memory efficiency, it can support the same configurations with fewer GPUs - typically half the number required by Flash-LLM and FasterTransformer. This not only reduces hardware requirements but also brings additional performance benefits. For instance, with the OPT-13B model, SpInfer only needs 1 RTX4090 GPU, eliminating the inter-GPU communication time required by FasterTransformer and Flash-LLM when using 2 GPUs. This advantage is also evident in A6000 GPU clusters. However, this benefit is particularly pronounced in RTX4090 GPU clusters, where only PCIe with relatively low bandwidth is available, and NVLink cannot be used.

## 6 Limitation and Discussion

Although SpInfer shows notable improvements in performance and memory efficiency, it faces limitations during the **prefill phase** when batch size and sequence length ($N = BS \times Seq\_len$) are large. In these cases, SpInfer can be up to 11.8% slower than cuBLAS_TC because the operation becomes more *compute-bound*, thus reducing the benefits of our memory-access optimizations (shown in Figure 16). The bitmap decoding overhead contributes to this performance gap, especially in dense matrix operations where cuBLAS leverages Tensor Cores more effectively. However, this impact is mitigated by several factors. Even in the prefill phase, SpInfer achieves substantial memory savings due to its high-compression TCA-BME format, which is crucial for managing long sequences and large models on limited hardware. Additionally, as inference systems increasingly adopt a decoupled prefill and decode phase architecture [63, 67, 68, 101], SpInfer's optimization for the decode phase makes it well-suited for scalable deployment. Addressing this limitation requires hardware-level support, such as **Sparse Tensor Cores** or specialized sparse GEMM accelerators, which represent promising directions for future optimization.

Beyond weight sparsity, SpInfer does not currently support **dynamic activation sparsity**, where sparsity patterns vary at runtime based on input-dependent activations [42, 47, 50, 75, 99]. Extending SpInfer to accommodate such runtime sparsity would require adaptive sparse encoding techniques to maintain computational efficiency. Furthermore, at extreme sparsity levels (>90%), the efficiency of bitmap indexing declines as excessive bits are used to represent zeros, resulting in a lower compression ratio than CSR formats [28]. In such scenarios, alternative approaches like DTC-SpMM [19] and SMaT [64] are more effective.

Although SpInfer is optimized for NVIDIA Tensor Cores, its core techniques are generalizable to other hardware architectures. The TCA-BME tiling strategy can be tailored to different matrix multiplication units, such as Google TPU [36], AMD Matrix Cores [70], and Intel AMX [37], by aligning the tile configurations with their respective specifications. Similarly, SMBD relies on basic bitwise operations, which are available across modern architectures [2, 3]. Future research includes developing compiler optimizations to automate SpInfer's adaptation for diverse hardware architectures, enhancing its cross-platform efficiency.

## 7 Conclusion

In this paper, we have presented SpInfer, an efficient framework designed to accelerate LLM inference on GPUs by leveraging unstructured pruning and sparse matrix multiplication. At the core of SpInfer is a novel Tensor-Core-Aware Bitmap Encoding (TCA-BME) format, which addresses the critical issue of indexing overhead, enabling substantial improvements in both memory efficiency and computational performance, even at low sparsity levels. We have also introduced a highly optimized SpInfer-SpMM kernel, which incorporates techniques including Shared Memory Bitmap Decoding (SMBD) and an asynchronous pipeline to maximize GPU resource utilization. Our evaluation reveals that

SpInfer consistently surpasses state-of-the-art SpMM kernels and inference frameworks across various sparsity levels, delivering substantial speedups alongside reduced memory usage. SpInfer is demonstrated to be the first framework that can effectively accelerate LLM inference at low sparsity levels (below 50%) while maintaining both computational efficiency and memory savings, addressing a critical gap in current sparse inference techniques.

## Acknowledgments

We extend our thanks to the anonymous EuroSys reviewers and our shepherd, Fan Yang, for their valuable feedback. This work was partially supported by National Natural Science Foundation of China under Grant No. 62272122, the Guangzhou Municipal Joint Funding Project with Universities and Enterprises under Grant No. 2024A03J0616, Guangzhou Municipality Big Data Intelligence Key Lab (2023 A03J0012), Hong Kong CRF grants under Grant No. C7004-22G and C6015-23G, the NSFC/RGC Collaborative Research Scheme under the contract of CRS_HKUST601/24, and National Natural Science Foundation of China under Grant No. 62302126. Qiang Wang and Xiaowen Chu are the corresponding authors.

## References

[1] Marah Abdin, Sam Ade Jacobs, Ammar Ahmad Awan, Jyoti Aneja, Ahmed Awadallah, Hany Awadalla, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Harkirat Behl, et al. Phi-3 technical report: A highly capable language model locally on your phone. *arXiv preprint arXiv:2404.14219*, 2024.

[2] Advanced Micro Devices. *AMD Instinct MI300 CDNA3 Instruction Set Architecture Reference Guide*. Advanced Micro Devices, Inc., 2024.

[3] Advanced Micro Devices. *AMD RDNA3 Instruction Set Architecture*. Advanced Micro Devices, Inc., 2024.

[4] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, and Yuxiong He. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '22. IEEE Press, 2022.

[5] Saleh Ashkboos, Maximilian L. Croci, Marcelo Gennari do Nascimento, Torsten Hoefler, and James Hensman. SliceGPT: Compress large language models by deleting rows and columns. In *ICLR*, 2024.

[6] Saleh Ashkboos, Amirkeivan Mohtashami, Maximilian L Croci, Bo Li, Martin Jaggi, Dan Alistarh, Torsten Hoefler, and James Hensman. Quarot: Outlier-free 4-bit inference in rotated llms. *arXiv preprint arXiv:2404.00456*, 2024.

[7] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc.

[8] Roberto L Castro, Diego Andrade, and Basilio B Fraguela. Probing the efficacy of hardware-aware weight pruning to optimize the spmm routine on ampere gpus. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 135–147, 2022.

[9] Zhaodong Chen, Zheng Qu, Liu Liu, Yufei Ding, and Yuan Xie. Efficient tensor core-based gpu kernels for structured sparsity under reduced precision. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[10] Guohao Dai, Guyue Huang, Shang Yang, Zhongming Yu, Hengrui Zhang, Yufei Ding, Yuan Xie, Huazhong Yang, and Yu Wang. Heuristic adaptability to input dynamics for spmm on gpus. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 595–600, 2022.

[11] Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations (ICLR)*, 2024.

[12] 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.

[13] Rocktim Jyoti Das, Liqun Ma, and Zhiqiang Shen. Beyond size: How gradients shape pruning decisions in large language models. *arXiv preprint arXiv:2311.04902*, 2023.

[14] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale. *Advances in Neural Information Processing Systems*, 35:30318–30332, 2022.

[15] Peijie Dong, Lujun Li, Zhenheng Tang, Xiang Liu, Xinglin Pan, Qiang Wang, and Xiaowen Chu. Pruner-zero: Evolving symbolic pruning metric from scratch for large language models. In *Proceedings of the 41st International Conference on Machine Learning*. PMLR, 2024. [arXiv: 2406.02924].

[16] Dayou Du, Yijia Zhang, Shijie Cao, Jiaqi Guo, Ting Cao, Xiaowen Chu, and Ningyi Xu. Bitdistiller: Unleashing the potential of sub-4-bit llms via self-distillation. *arXiv preprint arXiv:2402.10631*, 2024.

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

[18] Ruibo Fan, Wei Wang, and Xiaowen Chu. Fast sparse gpu kernels for accelerated training of graph neural networks. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 501–511. IEEE, 2023.

[19] Ruibo Fan, Wei Wang, and Xiaowen Chu. Dtc-spmm: Bridging the gap in accelerating general sparse matrix multiplication with tensor cores. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 253–267, 2024.

[20] Elias Frantar and Dan Alistarh. Sparsegpt: Massive language models can be accurately pruned in one-shot. In *ICML*, 2023.

[21] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.

[22] Elias Frantar, Roberto L Castro, Jiale Chen, Torsten Hoefler, and Dan Alistarh. Marlin: Mixed-precision auto-regressive parallel inference on large language models. *arXiv preprint arXiv:2408.11743*, 2024.

[23] Trevor Gale, Deepak Narayanan, Cliff Young, and Matei Zaharia. Megablocks: Efficient sparse training with mixture-of-experts, 2022.

[24] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. Sparse gpu kernels for deep learning. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14. IEEE, 2020.

[25] Georgi Gerganov. llama.cpp. https://github.com/ggerganov/llama.cpp, 2023.
[26] Scott Gray, Alec Radford, and Diederik P Kingma. Block-sparse gpu kernels, 2017.
[27] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, pages 770–778, 2016.
[28] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *The Journal of Machine Learning Research*, 22(1):10882–11005, 2021.
[29] Connor Holmes, Masahiro Tanaka, Michael Wyatt, Ammar Ahmad Awan, Jeff Rasley, Samyam Rajbhandari, Reza Yazdani Aminabadi, Heyang Qin, Arash Bakhtiari, Lev Kurilenko, and Yuxiong He. Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference, 2024.
[30] Changwan Hong, Aravind Sukumaran-Rajam, Bortik Bandyopadhyay, Jinsung Kim, Süreyya Emre Kurt, Israt Nisa, Shivani Sabhlok, Ümit V Çatalyürek, Srinivasan Parthasarathy, and P Sadayappan. Efficient sparse-matrix multi-vector product on gpus. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, pages 66–79, 2018.
[31] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P Sadayappan. Adaptive sparse tiling for sparse matrix multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 300–314, 2019.
[32] Ke Hong, Guohao Dai, Jiaming Xu, Qiuli Mao, Xiuhong Li, Jun Liu, Kangdi Chen, Yuhan Dong, and Yu Wang. Flashdecoding++: Faster large language model inference on gpus, 2024.
[33] Yuwei Hu, Zihao Ye, Minjie Wang, Jiali Yu, Da Zheng, Mu Li, Zheng Zhang, Zhiru Zhang, and Yida Wang. Featgraph: A flexible and efficient backend for graph neural network systems. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13. IEEE, 2020.
[34] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. Gespmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2020.
[35] Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. Mixtral of experts. *arXiv preprint arXiv:2401.04088*, 2024.
[36] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.
[37] Hyungyo Kim, Gaohan Ye, Nachuan Wang, Amir Yazdanbakhsh, and Nam Sung Kim. Exploiting intel® advanced matrix extensions (amx) for large language model inference. *IEEE Computer Architecture Letters*, 2024.
[38] Eldar Kurtic, Denis Kuznedelev, Elias Frantar, Michael Goin, and Dan Alistarh. Sparse fine-tuning for inference acceleration of large language models, 2023.
[39] Mark Kurtz, Justin Kopinsky, Rati Gelashvili, Alexander Matveev, John Carr, Michael Goin, William Leiserson, Sage Moore, Bill Nell, Nir Shavit, and Dan Alistarh. Inducing and exploiting activation sparsity for fast inference on deep neural networks. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 5533–5543, Virtual, 13–18 Jul 2020. PMLR.
[40] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.
[41] Yann LeCun, John Denker, and Sara Solla. Optimal brain damage. In *NeurIPS*, volume 2, 1989.
[42] Je-Yong Lee, Donghyun Lee, Genghan Zhang, Mo Tiwari, and Azalia Mirhoseini. Cats: Contextually-aware thresholding for sparsity in large language models. *arXiv preprint arXiv:2404.08763*, 2024.
[43] Shigang Li, Kazuki Osawa, and Torsten Hoefler. Efficient quantized sparse matrix operations on tensor cores. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2022.
[44] Bin Lin, Ningxin Zheng, Lei Wang, Shijie Cao, Lingxiao Ma, Quanlu Zhang, Yi Zhu, Ting Cao, Jilong Xue, Yuqing Yang, and Fan Yang. Efficient gpu kernels for n:m-sparse weights in deep learning. In D. Song, M. Carbin, and T. Chen, editors, *Proceedings of Machine Learning and Systems*, volume 5, pages 513–525. Curan, 2023.
[45] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. *Proceedings of Machine Learning and Systems*, 6:87–100, 2024.
[46] Yujun Lin, Haotian Tang, Shang Yang, Zhekai Zhang, Guangxuan Xiao, Chuang Gan, and Song Han. Qserve: W4a8kv4 quantization and system co-design for efficient llm serving. *arXiv preprint arXiv:2405.04532*, 2024.
[47] James Liu, Pragaash Ponnusamy, Tianle Cai, Han Guo, Yoon Kim, and Ben Athiwaratkun. Training-free activation sparsity in large language models, 2024.
[48] Zechun Liu, Barlas Oguz, Changsheng Zhao, Ernie Chang, Pierre Stock, Yashar Mehdad, Yangyang Shi, Raghuraman Krishnamoorthi, and Vikas Chandra. Llm-qat: Data-free quantization aware training for large language models. *arXiv preprint arXiv:2305.17888*, 2023.
[49] Zechun Liu, Changsheng Zhao, Igor Fedorov, Bilge Soran, Dhruv Choudhary, Raghuraman Krishnamoorthi, Vikas Chandra, Yuandong Tian, and Tijmen Blankevoort. Spinquant–llm quantization with learned rotations. *arXiv preprint arXiv:2405.16406*, 2024.
[50] Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Re, et al. Deja vu: Contextual sparsity for efficient llms at inference time. In *International Conference on Machine Learning*, pages 22137–22176. PMLR, 2023.
[51] Weile Luo, Ruibo Fan, Zeyu Li, Dayou Du, Qiang Wang, and Xiaowen Chu. Benchmarking and dissecting the nvidia hopper gpu architecture. *arXiv preprint arXiv:2402.13499*, 2024.
[52] Xinyin Ma, Gongfan Fang, and Xinchao Wang. Llm-pruner: On the structural pruning of large language models. *Advances in neural information processing systems*, 36:21702–21720, 2023.
[53] Maxim Naumov, L Chien, Philippe Vandermersch, and Ujval Kapasi. Cusparse library. In *GPU Technology Conference*, 2010.
[54] NVIDIA. NVIDIA volta gpu architecture whitepaper. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2017.
[55] NVIDIA. Nvidia a100 tensor core gpu architecture, 2020.
[56] NVIDIA. Fastertransformer. https://github.com/NVIDIA/FasterTransformer, 2023.
[57] NVIDIA. NVIDIA CUDA C Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2023.
[58] NVIDIA. PTX ISA: CUDA Toolkit documentation. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html, 2023.
[59] NVIDIA. cublas docs. https://docs.nvidia.com/cuda/cublas/index.html, 2024.
[60] NVIDIA. cusparse library. https://docs.nvidia.com/cuda/cusparse/index.html, 2024.

[61] NVIDIA. Nsight compute. https://developer.nvidia.com/nsight-compute, 2024.

[62] NVIDIA. Nsight systems. https://developer.nvidia.com/nsight-systems, 2024.

[63] Hyungjun Oh, Kihong Kim, Jaemin Kim, Sungkyun Kim, Junyeol Lee, Du-seong Chang, and Jiwon Seo. Exegpt: Constraint-aware resource scheduling for llm inference. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 369–384, New York, NY, USA, 2024. Association for Computing Machinery.

[64] Patrik Okanovic, Grzegorz Kwasniewski, Paolo Sylos Labini, Maciej Besta, Flavio Vella, and Torsten Hoefler. High performance unstructured spmm computation using tensor cores. *arXiv preprint arXiv:2408.11551*, 2024.

[65] OpenAI. Gpt-4 technical report, 2023.

[66] Meng Pang, Xiang Fei, Peng Qu, Youhui Zhang, and Zhaolin Li. A row decomposition-based approach for sparse matrix multiplication on gpus. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 377–389, 2024.

[67] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 118–132. IEEE, 2024.

[68] Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Kimi's kvcache-centric architecture for llm serving. *arXiv preprint arXiv:2407.00079*, 2024.

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

[70] Gabin Schieffer, Daniel Araújo De Medeiros, Jennifer Faj, Aniruddha Marathe, and Ivy Peng. On the rise of amd matrix cores: Performance, power efficiency, and programmability. In *2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 132–143. IEEE, 2024.

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

[72] Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E. Gonzalez, and Ion Stoica. Fairness in serving large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 965–988, Santa Clara, CA, July 2024. USENIX Association.

[73] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y. Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E. Gonzalez, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu, 2023.

[74] Jinliang Shi, Shigang Li, Youxuan Xu, Rongtian Fu, Xueying Wang, and Tong Wu. Flashsparse: Minimizing computation redundancy for fast sparse matrix multiplications on tensor cores. *arXiv preprint arXiv:2412.11007*, 2024.

[75] Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. Powerinfer: Fast large language model serving with a consumer-grade gpu. *arXiv preprint arXiv:2312.12456*, 2023.

[76] Mingjie Sun, Zhuang Liu, Anna Bair, and J Zico Kolter. A simple and effective pruning approach for large language models. In *Workshop on Efficient Systems for Foundation Models @ ICML2023*, 2023.

[77] Mingjie Sun, Zhuang Liu, Anna Bair, and J. Zico Kolter. A simple and effective pruning approach for large language models. In *ICLR*, 2024.

[78] Wei Sun, Ang Li, Tong Geng, Sander Stuijk, and Henk Corporaal. Dissecting tensor cores via microbenchmarks: Latency, throughput and numeric behaviors. *IEEE Transactions on Parallel and Distributed Systems*, 34(1):246–261, 2022.

[79] Vijay Thakkar, Pradeep Ramani, Cris Cecka, Aniket Shivam, Honghao Lu, Ethan Yan, Jack Kosaian, Mark Hoemmen, Haicheng Wu, Andrew Kerr, Matt Nicely, Duane Merrill, Dustyn Blasig, Fengqi Qiao, Piotr Majcher, Paul Springer, Markus Hohnerbach, Jin Wang, and Manish Gupta. CUTLASS, January 2023.

[80] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

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

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

[83] Yuke Wang, Boyuan Feng, Zheng Wang, Guyue Huang, and Yufei Ding. TC-GNN: Bridging sparse GNN computation and dense tensor cores on GPUs. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 149–164, 2023.

[84] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[85] Bingyang Wu, Yinmin Zhong, Zili Zhang, Shengyu Liu, Fangyue Liu, Yuanhang Sun, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models, 2024.

[86] Haojun Xia, Zhen Zheng, Yuchao Li, Donglin Zhuang, Zhongzhu Zhou, Xiafei Qiu, Yong Li, Wei Lin, and Shuaiwen Leon Song. Flash-llm: Enabling cost-effective and highly-efficient large generative model inference with unstructured sparsity. *Proc. VLDB Endow.*, 17(2):211–224, October 2023.

[87] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning*, pages 38087–38099. PMLR, 2023.

[88] Kaixin Xu, Zhe Wang, Chunyun Chen, Xue Geng, Jie Lin, Xulei Yang, Min Wu, Xiaoli Li, and Weisi Lin. Lpvit: Low-power semi-structured pruning for vision transformers. *arXiv preprint arXiv:2407.02068*, 2024.

[89] Peng Xu, Wenqi Shao, Mengzhao Chen, Shitao Tang, Kaipeng Zhang, Peng Gao, Fengwei An, Yu Qiao, and Ping Luo. BESA: Pruning large language models with blockwise parameter-efficient sparsity allocation. In *ICLR*, 2024.

[90] An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, et al. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*, 2024.

[91] Carl Yang, Aydın Buluç, and John D Owens. Design principles for sparse matrix multiplication on the gpu. In *European Conference on Parallel Processing*, pages 672–687. Springer, 2018.

[92] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. Sparsetir: Composable abstractions for sparse compilation in deep learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 660–678, New York, NY, USA, 2023. Association for Computing Machinery.

[93] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages

521–538, Carlsbad, CA, July 2022. USENIX Association.
[94] Zhihang Yuan, Yuzhang Shang, Yang Zhou, Zhen Dong, Chenhao Xue, Bingzhe Wu, Zhikai Li, Qingyi Gu, Yong Jae Lee, Yan Yan, et al. Llm inference unveiled: Survey and roofline model insights. *arXiv preprint arXiv:2402.16363*, 2024.
[95] Yujia Zhai, Chengquan Jiang, Leyuan Wang, Xiaoying Jia, Shang Zhang, Zizhong Chen, Xin Liu, and Yibo Zhu. Bytetransformer: A high-performance transformer boosted for variable-length inputs. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 344–355, 2023.
[96] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.
[97] Yingtao Zhang, Haoli Bai, Haokun Lin, Jialin Zhao, Lu Hou, and Carlo Vittorio Cannistraci. Plug-and-play: An efficient post-training pruning method for large language models. In *The Twelfth International Conference on Learning Representations*, 2024.
[98] Yilong Zhao, Chien-Yu Lin, Kan Zhu, Zihao Ye, Lequn Chen, Size Zheng, Luis Ceze, Arvind Krishnamurthy, Tianqi Chen, and Baris Kasikci. Atom: Low-bit quantization for efficient and accurate llm serving. *Proceedings of Machine Learning and Systems*, 6:196–209, 2024.
[99] Ningxin Zheng, Huiqiang Jiang, Quanlu Zhang, Zhenhua Han, Lingxiao Ma, Yuqing Yang, Fan Yang, Chengruidong Zhang, Lili Qiu, Mao Yang, et al. Pit: Optimization of dynamic sparse deep learning models via permutation invariant transformation. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 331–347, 2023.
[100] Ningxin Zheng, Bin Lin, Quanlu Zhang, Lingxiao Ma, Yuqing Yang, Fan Yang, Yang Wang, Mao Yang, and Lidong Zhou. SparTA: Deep-Learning model sparsity via Tensor-with-Sparsity-Attribute. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 213–232, 2022.
[101] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210, Santa Clara, CA, July 2024. USENIX Association.
[102] Zixuan Zhou, Xuefei Ning, Ke Hong, Tianyu Fu, Jiaming Xu, Shiyao Li, Yuming Lou, Luning Wang, Zhihang Yuan, Xiuhong Li, Shengen Yan, Guohao Dai, Xiao-Ping Zhang, Yuhan Dong, and Yu Wang. A survey on efficient inference for large language models. *arXiv preprint arXiv:2404.14294*, 2024.

# A Artifact Appendix

SpInfer is a high-performance sparse LLM inference framework on GPUs. At the kernel level, SpInfer introduces the Tensor-Core-Aware Bitmap Encoding (TCA-BME) format for sparse matrix storage and implements a specialized SpMM kernel. At the framework level, SpInfer integrates with the popular FasterTransformer framework to reduce learning overhead and enhance user productivity and code portability.

## A.1 Artifact Check-list

- **Program**: SpInfer-kernel, SpInfer-integrated FasterTransformer
- **Compilation**: gcc (≥7.3), cmake (≥3.30.3), CUDA (≥12.2), nvcc (≥12.0)
- **Hardware**: NVIDIA RTX4090 or A6000 GPUs
- **Execution time**: ~9 hours for kernel benchmarks, model-dependent for end-to-end evaluation
- **Publicly available**: Yes, via GitHub and Zenodo

## A.2 Description

### A.2.1 How to Access.

- GitHub repository: https://github.com/HPMLL/SpInfer_EuroSys25.git
- Zenodo artifact: https://doi.org/10.5281/zenodo.14946485

**Listing 2.** Repository setup commands

```
git clone https://github.com/HPMLL/SpInfer_EuroSys25.git
cd SpInfer
git submodule update --init --recursive
source Init_SpInfer.sh
```

### A.2.2 Hardware Dependencies.

- NVIDIA RTX4090 or A6000 GPUs
- System memory ≥ 128GB (for model loading)

### A.2.3 Software Dependencies.

- Operating System: Ubuntu 18.04 or higher
- Compiler: gcc ≥ 7.3
- CUDA Toolkit: CUDA ≥ 12.2, nvcc ≥ 12.0
- Python Environment Manager: Miniconda/Anaconda

## A.3 Installation

**A.3.1 Environment Setup.** After installing Miniconda on the system, create the required environment:

**Listing 3.** Environment setup commands

```
cd $SpInfer_HOME
conda env create -f spinfer.yml
conda activate spinfer
```

**A.3.2 Building SpInfer.** Compile the core library:

**Listing 4.** Build command

```
cd $SpInfer_HOME/build && make -j
```

## A.4 Kernel-level Benchmarking (Figure 10)

**A.4.1 Build dependencies:** Execute the following instruction to build Sputnik and SparTA.

**Listing 5.** Dependency build commands

```
cd $SpInfer_HOME/third_party/
source build_sputnik.sh
source prepare_cusparselt.sh
```

**A.4.2 Execute benchmarks:** Run kernel-level benchmarks and check the output Figure10.png.

**Listing 6.** Benchmark execution commands

```
cd $SpInfer_HOME/kernel_benchmark
source test_env
make -j
source benchmark.sh
```

### A.5 End-to-End Model Evaluation

Follow the detailed `SpInfer/docs/LLMInferenceExample` for:

- Building Faster-Transformer with SpInfer, Flash-llm or Standard integration
- Downloading & Converting OPT models

**Configuration Note**: Model_dir differs for SpInfer, Flash-llm and Faster-Transformer.

**SpInfer Inference:**

**Listing 7.** SpInfer evaluation commands

```
# Single-GPU evaluation
cd $SpInfer_HOME/third_party/
bash run_1gpu_loop.sh
# Results in $SpInfer_HOME/third_party/
    FasterTransformer/Result_13B/1-gpu/

# Multi-GPU evaluation
bash run_2gpu_loop.sh  # For tensor_para_size=2
bash run_4gpu_loop.sh  # For tensor_para_size=4
```

**Flash-llm Inference:**

**Listing 8.** Flash-llm evaluation commands

```
# You need to download Flash-llm and write
    run_gpu_loop like SpInfer
cd /mnt/flash-llm/
source Init_FlashLLM.sh
cd $FlashLLM_HOME/third_party/
bash run_1gpu_loop.sh  # For tensor_para_size=1
# Results in $FlashLLM_HOME/third_party/
    FasterTransformer/Result_13B/1-gpu/

# Multi-GPU evaluation
bash run_2gpu_loop.sh  # For tensor_para_size=2
bash run_4gpu_loop.sh  # For tensor_para_size=4
```

**Faster-transformer Inference:**

**Listing 9.** Faster-transformer evaluation commands

```
# You need to download FT
cd /mnt/faster-transformer/
export FT_HOME=$pwd
cd $FT_HOME/third_party/
bash run_2gpu_loop.sh # For tensor_para_size=2
# Results in $FT_HOME/FasterTransformer/Result_13B/2-
    gpu/
bash run_4gpu_loop.sh  # For tensor_para_size=4
```

**DeepSpeed Inference:**

**Listing 10.** DeepSpeed evaluation commands

```
cd $SpInfer_HOME/end2end_inference/ds_scripts
pip install -r requirements.txt
bash run_ds_loop.sh
# Results in $SpInfer_HOME/end2end_inference/
    ds_scripts/ds_result/
```

**Drawing plot:**

**Listing 11.** Draw plot commands

```
cd $SpInfer_HOME/end2end_inference
python draw_plot.py
```

### A.6 Notes

- All experiments should be conducted with CUDA 12.2 or higher to ensure reproducibility
- Memory requirements may vary based on the selected model size and batch configuration
- Ensure experiments are conducted on NVIDIA RTX 4090 or NVIDIA A6000 GPUs to precisely replicate the reported results