SparQuant: Accelerating Large Language Models via Integrated Quantization and Unstructured Sparsity

Abstract—**The widespread adoption of large language models (LLMs) across diverse applications is accompanied by substantial computational and memory demands. Consequently, model compression techniques, such as quantization and unstructured pruning, have emerged as effective strategies to mitigate these resource requirements. Despite their individual effectiveness, the synergistic application of these methods to accelerate GPU inference remains a critical, unaddressed challenge that limits performance. To bridge this gap, we introduce SparQuant, a novel framework designed to accelerate LLM inference by integrating 8-bit quantization with unstructured sparsity. Our primary contribution is the Tiled-Bitmap compression format, which enables the efficient storage of sparse, quantized matrices and facilitating a highly optimized sparse matrix-matrix multiplication (SpMM) kernel for LLM inference acceleration on GPU. Experiments on various LLMs demonstrate the effectiveness of SparQuant, yielding a mean speedup of 1.57× in kernel-level SpMM operations and a 1.33-1.83× speedup in the end-to-end model evaluation, compared to the baseline.**

Index Terms—**Large language model, Quantization, Unstructured Sparsity, Tensor Core**

I. Introduction

Large Language Models (LLMs) [1]–[4] have been widely adopted across diverse applications, from text generation and machine translation to multimodal image synthesis. However, the continuous growth in the number of model parameters presents a significant challenge to their deployment on edge devices, which are inherently constrained by limited computational and memory resources. The bottleneck of LLMs lies in the skinny matrices, which are bounded by high bandwidth memory (HBM) bandwidth. Consequently, various research efforts have focused on reducing the model size to enhance performance.

Among the principal strategies for model compression, quantization [5], [6] and weight pruning [7], [8] are the most prominent. Quantization aims to reduce the model's memory and computational footprint by lowering the numerical precision, while weight pruning seeks to achieve similar goals by removing redundant connections.

In the domain of quantization, various precision formats are employed. A widely adopted standard is 8-bit integer (INT8) quantization [9], which represents both model weights and activations to an 8-bit integer representation. For even greater compression, more aggressive 4-bit and 2-bit formats [10]–[12] are typically applied to weights only, while keeping activations in higher precision. However, these low-bit methods often require sophisticated techniques for numerical outliers or expensive retraining to avoid significant accuracy degradation.

Weight pruning algorithms are broadly categorized into three paradigms: structured [13], semi-structured [7], [14], and unstructured [7], [15]. Structured pruning operates at a coarse granularity by eliminating entire components like channels or attention heads. This approach is hardware-friendly, maintaining dense data structures for acceleration, but can cause significant accuracy degradation and require extensive retraining. Offering a middle ground, semi-structured pruning, such as the N:M fine-grained sparsity on modern GPUs, provides a balanced trade-off between performance and accuracy. At the finest granularity, unstructured pruning removes individual weights, typically preserving the most accuracy for a given sparsity level. While specialized hardware like NVIDIA Tensor Cores does not directly accelerate unstructured sparsity, studies [16], [17] show that storing weights in a sparse format and computing in the dense format can still outperform the dense counterpart greatly, motivating this paper's focus on the unstructured approach.

However, effectively integrating quantization with unstructured sparsity presents a non-trivial challenge, as the two techniques impose conflicting constraints on data layout and hardware execution. For instance, high-performance quantized computations on Tensor Cores mandate rigid data shapes and alignments, whereas efficient unstructured compression formats impose their own distinct structural requirements. The bitmap compression used in SpInfer [17], for example, is constrained by the `__popcll` (population count) instruction to tile sizes of 64 elements. To our knowledge, a method that can unify these conflicting requirements to simultaneously optimize compression and computation for LLMs is still lacking in the literature.

To address this issue, we propose SparQuant, a novel framework to accelerate LLM inference via the integrated application of quantization and unstructured sparsity. The primary contributions of this work are as follows:

- We introduce SparQuant, a framework that synergistically combines 8-bit quantization with unstructured sparsity, and we design a highly optimized sparse matrix-matrix multiplication (SpMM) kernel to accelerate LLM inference.
- We propose a novel Tiled-Bitmap compression format with a 4-level tile hierarchy, specifically designed to reconcile the conflicting tile shape constraints of quantization and unstructured sparsity, enabling simultaneous performance gains from both.
- Our extensive evaluations on an NVIDIA RTX 4090 GPU show that SparQuant achieves a mean speedup of 1.57× in kernel-level SpMM operations compared to the state-of-the-art baseline. Furthermore, it delivers end-to-end model speedups ranging from 1.33× to 1.83×.

II. Background

A. Bottleneck of LLM Inference

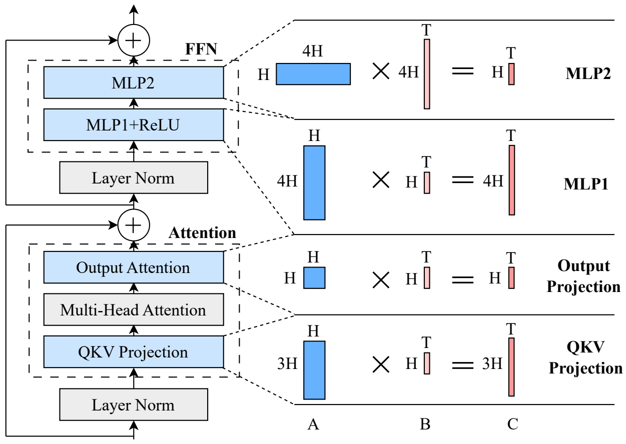


Fig. 1. The bottleneck within the decoder layer architecture.

LLMs [1], [2], [18] are composed of multiple decoder layers, with an architecture as depicted in Fig. 1. Each decoder layer has two main sub-layers: a multi-head attention module and a position-wise feed-forward network (FFN). Residual connections and layer normalization are applied to each of these two sub-layers. The attention module consists of query-key-value (QKV) projections, a multi-head attention, and an output projection. The FFN consists of two multilayer perceptrons (MLPs) with a non-linear activation layer in between.

The inference process of LLMs can be divided into two phases: prefill and decode. During the prefill phase, all prompts are processed in parallel, whereas in the decode phase, tokens are generated sequentially in an autoregressive manner. Previous research [16], [17] has demonstrated that the skinny matrix multiplication (MatMul) operation in the decode phase is the primary bottleneck of LLM inference, accounting for 60% to 80% of the total inference time. In the MatMul operation, a matrix A of size $M \times K$ is multiplied by a matrix B of size $K \times N$ to generate a matrix C of size $M \times N$. The skinny MatMul is characterized by the size of N being significantly smaller than both M and K. For instance, in the right part of Fig. 1, H represents the hidden size and T stands for the batch size, with T being much smaller than H.

B. Quantization and Pruning

Quantization and pruning are two prominent techniques employed to accelerate LLM inference. SmoothQuant [9] quantizes both weights and activations to 8 bits with negligible accuracy loss, while AWQ [5] quantizes weights to 4 bits but maintains activations at 16 bits. Atom [11] quantizes both activations and weights to 4 bits but requires higher precision to handle outliers. Regarding pruning methods, SparseGPT [7] can prune LLMs into N:M sparsity and unstructured sparsity, with unstructured sparsity achieving better accuracy than N:M sparsity. Wanda [14] simplifies the pruning process of SparseGPT and obtains similar results in a significantly shorter time. With unstructured pruning, the bottleneck of LLM inference becomes the skinny SpMM, where matrix A is sparse and matrix B remains dense. These pruning methods are effective for sparsity levels approximately 50%. When the sparsity becomes too high, they still suffer from significant accuracy loss.

C. Compression Formats

To optimize the SpMM operation, various compression formats are employed [16], [17], [19]. Flash-LLM [16] utilizes the Tile_CSL compression format, which stores the non-zero values and the index of each value. Sputnik [20] uses the compressed sparse row (CSR) format, which stores the non-zero elements along with their column indices and a much shorter row array. SpInfer [17] employs the bitmap format, which stores the bitmask array of the original matrix and needs to recover the indices during the computation process.

To compare different compression formats in 8-bit and 16-bit precision, we introduce the compression ratio (CR) metric:

$$\text{CR} = \frac{\text{Stor}_{\text{orig}}}{\text{Stor}_{\text{comp}}} \tag{1}$$

$$\text{Stor}_{\text{orig}} = M \times K \times W_{\text{data}} \tag{2}$$

As defined in Eq. 1, the CR is the ratio of the original storage space ($\text{Stor}_{\text{orig}}$) to the compressed storage space ($\text{Stor}_{\text{comp}}$). A higher CR value signifies more effective compression. The original storage space for a matrix with size $M \times K$ is calculated according to Eq. 2, where W_{data} represents the bit-width of the data (i.e., 8 or 16 bits).

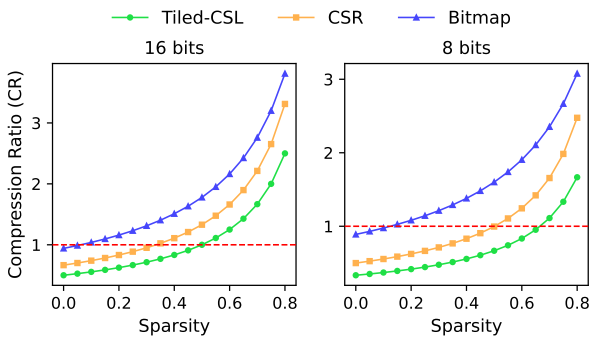


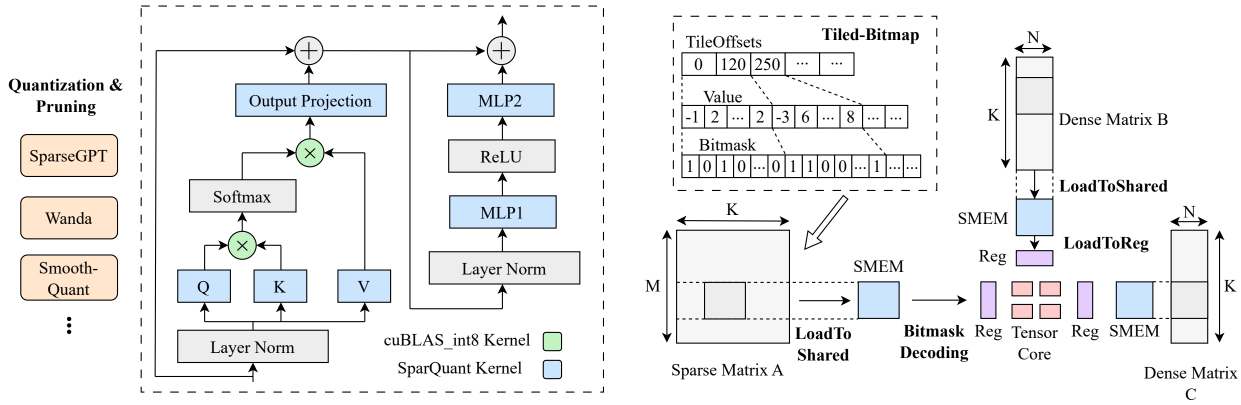
Fig. 2. Comparison of compression ratios for different formats at various sparsity levels for both 8-bit and 16-bit precision.

We evaluated three compression formats across a range of sparsity levels for both 8-bit and 16-bit precision. The results, presented in Fig. 2, show that the bitmap format is superior for matrices with up to 80% sparsity. This advantage is attributable to the memory savings from the bitmask array over direct index storage. However, converting this higher compression ratio into a throughput speedup requires carefully designed tile structures and kernels tailored for each precision.

III. SparQuant

A. System Overview

SparQuant is a framework that integrates 8-bit quantization and unstructured sparsity to optimize the throughput of LLM inference. A system-level depiction of the SparQuant is provided in Fig. 1. The core innovation of this framework



(a) Decoder layer architecture

(b) SparQuant kernel

Fig. 3. System overview of SparQuant, which features a novel decoder layer architecture with the SparQuant kernel for SpMM.

is a novel decoder layer architecture, which incorporates the SparQuant kernel for efficient SpMM. SparQuant applies both quantization and pruning algorithms to each decoder layer, compressing the model's weights into the Tiled-Bitmap format. Since the quantization, pruning and compressing processes are performed offline, they do not introduce any computational overhead during inference.

The decoder layer architecture, illustrated in Fig. 3 (a), utilizes two distinct kernels for matrix multiplication. Dense matrix multiplications, specifically for the multi-head attention, are processed by the cubBLAS_INT8 kernel. Conversely, all sparse matrix multiplications, encompassing the QKV projections, output projections, and the two MLPs in the feed-forward network, are managed by the SparQuant kernel. To ensure high precision, operations such as layer normalization, softmax, and residual calculations are performed using 16-bit floating-point (FP16) arithmetic.

B. SparQuant Kernel

The SparQuant kernel is employed to execute the SpMM operation. This operation involves the multiplication of a sparse matrix, denoted as A, with dimensions $M \times K$, by a dense matrix B, of size $K \times N$, resulting in a dense matrix C, with dimensions $K \times N$. Fig. 3 (b) illustrates an example of the SparQuant kernel. The sparse matrix A is encoded using the **Tiled-Bitmap** compression format, which consists of three arrays: a `TileOffsets` array, a `Value` array, and a `Bitmask` array. The `Value` array stores the non-zero elements, while the `Bitmask` array is a binary representation indicating the positions of non-zero elements, where "1" signifies a non-zero value and "0" indicates a zero value. The `Value` and `Bitmask` arrays are organized into four-level tiles, detailed in Section III-B.

The computational process within the SparQuant kernel begins with loading a tile of the sparse matrix A from the global buffer into shared memory (SMEM) in its compressed form. Subsequently, the bitmap is decoded to reconstruct the dense representation of the data within the registers (Reg). Concurrently, the corresponding tile of the dense matrix B is loaded from the global buffer into shared memory and then

Algorithm 1 SparQuant SpMM kernel pseudo code

Input: sparse matrix A (tiled-bitmap format), dense matrix B, matrix dimensions M, K and N.
Output: dense matrix C.
Hyperparameters:
TILE_M, TILE_N, TILE_K: The dimensions of the tiles used to partition the matrices.

```
// Shared memory allocation for double buffering.
__shared__ uint64_t smem_A_bitmap[2][TILE_M * TILE_K / 64];
__shared__ int8_t smem_A_values[2][TILE_M * TILE_K];
__shared__ int8_t smem_B[2][TILE_K * TILE_N];
// Initial data prefetch (fill first buffer).
LoadToSharedA(A, smem_A_bitmap[0], smem_A_values[0]);
LoadToSharedB(B, smem_B[0]);
cp.async.commit_group;        ▷ Commit async load for the initial stage.
cp.async.wait_all;            ▷ Wait for initial data to become resident.
__syncthreads();
// Main Loop: overlap computation with data loading.
for id from 0 to K / TILE_K do
    // Prefetch the next tile.
    int next = id % 2 + 1;
    LoadToSharedA(A, smem_A_bitmap[next], smem_A_values[next]);
    LoadToSharedB(B, smem_B[next]);
    cp.async.commit_group;  ▷ Commit async prefetch for the next stage.
    // Perform computation on the current tile.
    int curr = id % 2;
    Reg_A = BitmaskDecoding(smem_A_bitmap[curr],
            smem_A_values[curr]);
    Reg_B = LoadToFragB(smem_B[curr]);
    Reg_C = TensorCoreComputation(Reg_A, Reg_B);
    cp.async.wait_all;      ▷ Wait for prefetched data to become resident.
    __syncthreads();
end for
// Store results.
StoreToC(C, Reg_C);
```

into registers. Once the data is resident in the registers, the core computation is executed. Upon completion of the computation, the resulting matrix C is written back to global memory.

The implementation of this computational workflow is formalized in Algorithm 1, which outlines the SparQuant SpMM kernel. The algorithm is configured by tile-size hyperparameters (`TILE_M`, `TILE_N`, `TILE_K`), which partition the matrices for block-based processing. Leveraging software pipelining-based optjmizations with double buffering techniques, the kernel overlaps data transfer and computation, maximizing GPU bandwidth utilization.

The process begins with an initialization phase where the first tiles of matrices A and B are asynchronously prefetched into one of the two allocated shared memory buffers (lines 6-8). The kernel then enters its main computational loop, which iterates through the K dimension of the input matrices (line 12). Within each iteration, the algorithm initiates an asynchronous load of the next required tiles into the alternate shared memory buffer (lines 14-17). Concurrently, it performs computations on the current tiles that are already resident in shared memory. This computational stage involves decoding the sparse matrix A from the Tiled-Bitmap format, loading fragments of the dense matrix B into registers, and finally executing the matrix multiplication using the Tensor Cores (lines 19-22). Subsequently, a synchronization step ensures that the data prefetch for the next stage is complete before the subsequent iteration begins (line 23-24). Once the loop finishes, the accumulated results are written to the output matrix C in global memory (line 27).

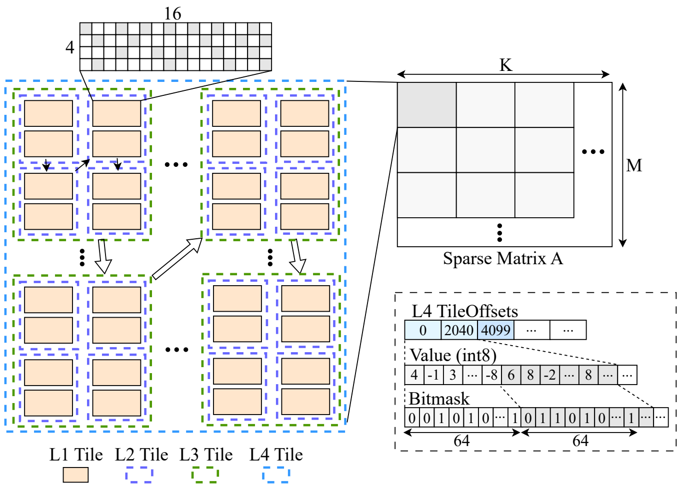


Fig. 4. Layout of the Tiled-Bitmap compression format, characterized by a 4-level tile structure.

C. Tiled-Bitmap

The Tiled-Bitmap compression format employs a four-level tiling scheme to dictate the arrangement of the Value and Bitmask arrays, as depicted in Fig. 4.

Level 1 (L1) Tile: The L1 tile has dimensions of 4×16, encompassing 64 elements. This size is specifically chosen for bitmap compression to align with the 64-bit operand width of the `__popcll` instruction.

Level 2 (L2) Tile: The L2 tile is constructed as 8×16. This configuration is designed for efficient data loading by a single warp from shared memory to registers.

Level 3 (L3) Tile: The L3 tile, with dimensions of 16×32, represents the basic data unit for Tensor Core computation in INT8 (`m16n8k32`). L2 tiles in an L3 tile follow a *column-major* order (see Fig. 5 (b)) to match the `m16n8k32` layout, with **each L3 Tile containing 4 L2 Tiles**.

Level 4 (L4) Tile: The tile size is set to 64×128, resulting in **1 L4 Tile being composed of 16 L3 Tiles**. These dimensions correspond to the `TILE_M` and `TILE_N` parameters in Algorithm 1 and determine the number of data elements held in each shared memory block.

The comprehensive storage scheme is illustrated in the bottom right of Fig. 4. Despite the presence of four-level tiles, only the L4 tile offsets need to be stored. The elements in the `Value` and `Bitmask` arrays should be organized according to the four-level tiling structure to facilitate further decompression and computation.

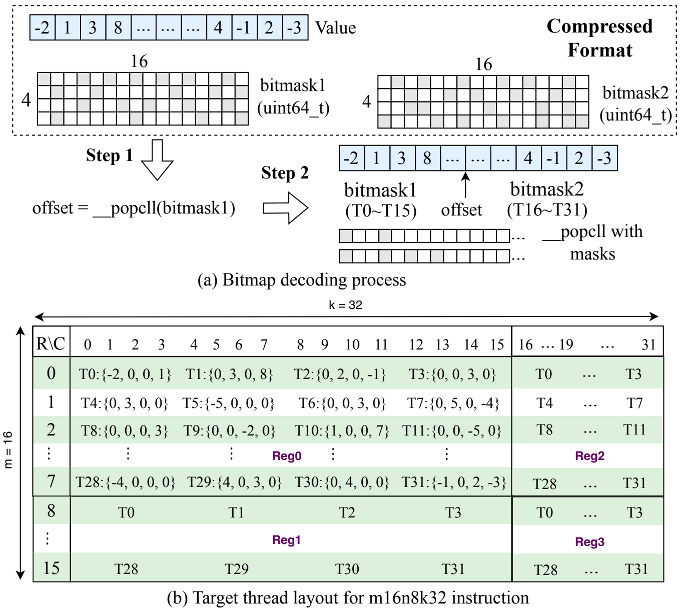


Fig. 5. Bitmap decoding from the Tiled-Bitmap compressed format (input) to the dense format (output adapted to `MMA.m16n8k32` fragment layout for matrix A with `.u8` type.

D. Thread-Data Mapping and Bitmap Decoding

In our design, thread blocks map to L4 Tiles, each warp processes an L3 Tile (`m16n8k32` fragment), and each L2 Tile corresponds to a partition of a `m16n8k32` fragment (see Fig. 5 (b)). Half of a warp map to each L1 Tile for data loading.

The bitmap decoding process is detailed in Fig. 5 (a), which uses a single warp managing two L1 tiles as an example. The compressed data comprises an array of INT8 values (the `Value` array) and two uint64_t variables, `bitmask1` and `bitmask2` (the `Bitmask` array). As depicted in the figure, gray and white boxes within the bitmasks represent "1" and "0", respectively.

The decoding procedure works in two steps. First, the population count of `bitmask1` is calculated using the `__popcll` instruction to generate an offset. This offset signifies the number of non-zero elements within the first L1 tile. Second, this offset is used to partition the `Value` array. The threads in the warp are divided to process the data in parallel: threads 0∼15 are assigned to the first partition, which corresponds to `bitmask1`, while threads 16∼31 handle the second partition, corresponding to `bitmask2`. Through the collaborative execution of `__popcll` instructions with masks, the entire warp reconstructs the data, loading it into registers in the required layout for `MMA.m16n8k32` instruction.

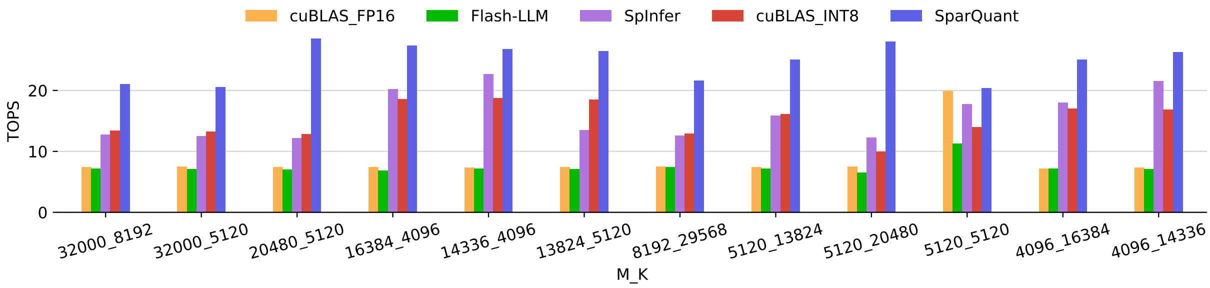


Fig. 6. Kernel performance with 50% sparsity and a batch size of 8.

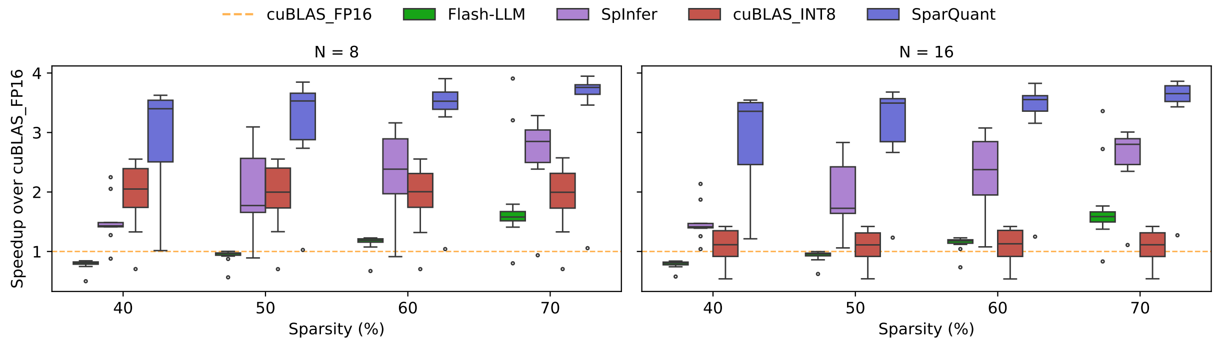


Fig. 7. Distribution of kernel performance across various sparsity levels for batch sizes N=8 and N=16.

IV. EXPERIMENT

We evaluate SparQuant on an NVIDIA RTX 4090 GPU with 24 GB of GDDR6X memory[1]. The evaluation includes both kernel-level benchmarks and end-to-end inference tests on the Llama 2 [4] and OPT model [21] series. All reported performance and energy metrics are the average of 1000 runs.

TABLE I
COMPARISON OF EVALUATED SCHEMES.

Schemes	GEMM	SpMM	Unstructured Sparsity	Quantization
cuBLAS_FP16	✓			FP16
cuBLAS_INT8	✓			INT8
Flash-LLM		✓	✓	FP16
SpInfer		✓	✓	FP16
SparQuant		✓	✓	INT8

A. Kernel Performance

Experimental Setup and Evaluated Schemes. This study evaluates a selection of LLMs, specifically the OPT-Series (7B, 13B, 30B) [21] and LLaMA-2-series (7B, 13B) [4]. We investigate the impact of weight pruning by inducing random sparsity levels ranging from 40% to 70%. Considering the resource constraints of edge devices, we evaluate the performance with batch sizes (N) of 8 and 16. We select cuBLAS_FP16 [22], cuBLAS_INT8 [22], SpInfer [17], and Flash-LLM [16] as baseline methods for comparison. The differences among the comparative works and SparQuant are shown in Table I. cuBLAS_FP16 and cuBLAS_INT8 are designed for dense matrix multiplication, while SpInfer and Flash-LLM are designed for sparse matrix multiplication using different compression formats: TCA-BME and Tiled-CSL, respectively.

[1]https://anonymous.4open.science/r/spar_quant-open-66F7

Performance Results. The kernel performance results for a batch size 8 with 50% sparsity is presented in Fig. 6. The x-axis represents the matrix dimensions (M and K), while the y-axis indicates the throughput in tera operations per second (TOPS). SparQuant achieves significant throughput speedups of 3.21×, 1.57×, and 1.64× over cuBLAS_FP16, SpInfer, and cuBLAS_INT8, respectively. This performance gain is attributable to the effective combination of 8-bit quantization and unstructured sparsity within SparQuant, which substantially reduces memory access.

The distribution of kernel performance across various sparsity levels for batch sizes 8 and 16 is illustrated in Fig. 7. On average, SparQuant achieves a speedup of 2.94-3.50× over cuBLAS_FP16, 1.31-2.01× over SpInfer, and 1.54-3.17× over cuBLAS_INT8. These results demonstrate that SparQuant can achieve superior performance over a wider range of sparsity levels for unstructured pruning compared to the fixed 50% sparsity used in N:M pruning. The suboptimal performance of cuBLAS_INT8 for batch size 16 can be attributed to the lack of optimization for this specific configuration.

B. Kernel Analysis

For a more detailed analysis of the performance results, we employ Nsight Compute [23] to profile several micro-architectural metrics, with the normalized results presented in Fig. 8. These metrics include DRAM read bytes (DRAM Bytes), number of registers (Registers), DRAM bandwidth utilization (DRAM BW), L1 cache throughput utilization (L1 Tput), and streaming multiprocessor (SM) throughput utilization (SM Tput). The analysis reveals that SparQuant substantially reduces DRAM read bytes—by 68.4%, 43.6%, and 36.7%

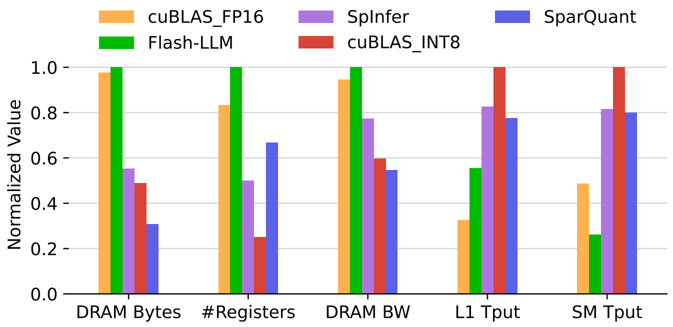


Fig. 8. Performance characterization of the SparQuant Kernel.

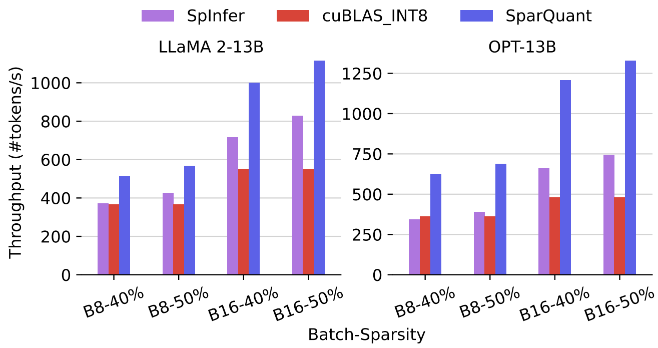


Fig. 9. End-to-end inference throughput for LLaMA 2-13B and OPT-13B.

against the baselines. This reduction in data movement from DRAM is the primary driver behind the throughput gains. Despite this significant improvement, our analysis reveals that the DRAM bandwidth is not yet fully saturated, which suggests that there is still scope for further optimization.

C. End-to-End Inference Performance Comparison

Setup. For the end-to-end evaluation, we select OPT-13B [21] and LLaMA-2-13B [4] models to maximize DRAM utilization on the RTX 4090 GPU. SparseGPT [7] and Smoothquant [9] are employed for unstructured pruning and 8-bit quantization, respectively. For a comparative analysis, SpInfer [17] and cuBLAS_INT8 [22] are chosen as the baseline due to their superior performance among the available baselines. These kernels are integrated into the Punica framework, which is built on Pytorch, to enable efficient inference. To maintain a fair comparison, all tests are performed with a fixed sequence length of 2048, using Wikitext-2 [24] and C4 [25] datasets.

TABLE II
PERPLEXITY OF THE LLAMA-2-13B AND OPT-13B MODELS AT VARIOUS SPARSITIES FOR BOTH INT8 AND FP16 PRECISION.

Sparsity	Dense	0.4	0.5	0.6
LLaMA-2-13B_INT8	5.80	6.36	7.13	9.60
LLaMA-2-13B_FP16	5.80	6.35	7.12	9.55
OPT-13B_INT8	11.09	11.37	12.09	13.70
OPT-13B_FP16	11.09	11.35	12.08	13.66

Perplexity. The perplexity scores, calculated as the average over the Wikitext-2 and C4 datasets, are detailed in Table II. The application of 8-bit quantization has a negligible impact on perplexity. However, unstructured pruning demonstrated a significant influence on perplexity. At 50% sparsity, the perplexity increases by 22.9% and 9.0% for LLaMA 2-13B and OPT-13B, respectively. Beyond this point, continued increases in sparsity result in an unacceptable rise in perplexity.

Model Performance. We evaluate the end-to-end inference throughput for LLaMA 2-13B and OPT-13B at 40% and 50% sparsity levels. Fig. 9 presents the results, with the x-axis representing the batch size and sparsity (Batch-Sparsity) and the y-axis denoting the number of tokens per second (#tokens/s). SparQuant achieved a speedup of 1.40-2.77× over cuBLAS_INT8 and 1.33-1.83× over SpInfer. As discussed in Section II-A, the bottleneck of LLM inference is the skinny MatMul. Therefore, kernel improvements can directly translate into enhanced end-to-end model inference performance.

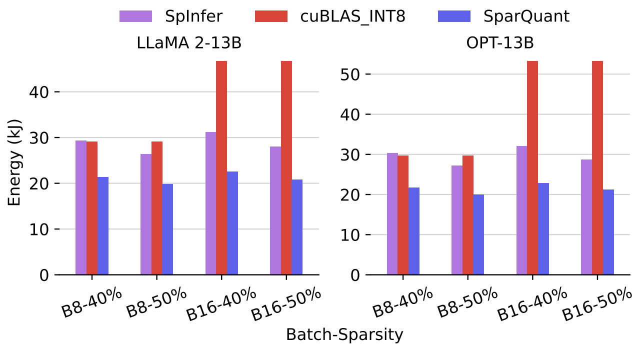


Fig. 10. End-to-end energy consumption for LLaMA 2-13B and OPT-13B.

Model Energy. The `nvidia-smi` tool is employed to monitor energy consumption during the experiments. Fig. 10 illustrates the results, with the y-axis representing the energy consumption for each inference. SparQuant achieves a 26%-60% energy reduction compared to cuBLAS_INT8 and a 25%-28% reduction compared to SpInfer. This reduction can be attributed to the reduced inference time achieved through our efficient kernel implementation. Since the total energy consumed is a product of power and execution time, shortening the computation duration directly lowers the overall energy required per inference.

V. CONCLUSION

We propose SparQuant, a framework to effectively accelerate LLM inference by combining unstructured sparsity with 8-bit quantization. Our primary innovation, the Tiled-Bitmap compression format, facilitates the efficient storage of compressed matrices and execution of a highly optimized SpMM kernel on GPUs. We have experimentally demonstrate the success of our framework. Compared to the baseline, SparQuant provides a mean kernel-level speedup of 1.57× and achieves 1.33× to 1.83× speedup for end-to-end model inference. Our work confirms that a dedicated framework for sparse-quantized models can overcome existing performance limitations and significantly improve the inference throughput of LLMs.

References

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

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

[3] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, 2019, pp. 4171–4186.

[4] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[5] J. Lin, J. Tang, H. Tang, S. Yang, W.-M. Chen, W.-C. Wang, G. Xiao, X. Dang, C. Gan, and S. Han, "Awq: Activation-aware weight quantization for on-device llm compression and acceleration," *Proceedings of machine learning and systems*, vol. 6, pp. 87–100, 2024.

[6] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, "Gptq: Accurate post-training quantization for generative pre-trained transformers," *arXiv preprint arXiv:2210.17323*, 2022.

[7] E. Frantar and D. Alistarh, "Sparsegpt: Massive language models can be accurately pruned in one-shot," in *International conference on machine learning*. PMLR, 2023, pp. 10 323–10 337.

[8] Y. Chen, B. Cheng, J. Han, Y. Zhang, Y. Li, and S. Zhang, "Dlp: Dynamic layerwise pruning in large language models," *arXiv preprint arXiv:2505.23807*, 2025.

[9] G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth, and S. Han, "Smoothquant: Accurate and efficient post-training quantization for large language models," in *International conference on machine learning*. PMLR, 2023, pp. 38 087–38 099.

[10] J. Li, J. Xu, S. Li, S. Huang, J. Liu, Y. Lian, and G. Dai, "Fast and efficient 2-bit llm inference on gpu: 2/4/16-bit in a weight matrix with asynchronous dequantization," in *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, 2024, pp. 1–9.

[11] Y. Zhao, C.-Y. Lin, K. Zhu, Z. Ye, L. Chen, S. Zheng, L. Ceze, A. Krishnamurthy, T. Chen, and B. Kasikci, "Atom: Low-bit quantization for efficient and accurate llm serving," *Proceedings of Machine Learning and Systems*, vol. 6, pp. 196–209, 2024.

[12] Z. Liu, C. Zhao, H. Huang, S. Chen, J. Zhang, J. Zhao, S. Roy, L. Jin, Y. Xiong, Y. Shi *et al.*, "Paretoq: Scaling laws in extremely low-bit llm quantization," *arXiv preprint arXiv:2502.02631*, 2025.

[13] S. Ashkboos, M. L. Croci, M. G. d. Nascimento, T. Hoefler, and J. Hensman, "Slicegpt: Compress large language models by deleting rows and columns," *arXiv preprint arXiv:2401.15024*, 2024.

[14] M. Sun, Z. Liu, A. Bair, and J. Z. Kolter, "A simple and effective pruning approach for large language models," *arXiv preprint arXiv:2306.11695*, 2023.

[15] Y. Zhang, H. Bai, H. Lin, J. Zhao, L. Hou, and C. V. Cannistraci, "Plug-and-play: An efficient post-training pruning method for large language models," *The Twelfth Interntional Conference on Learning Representations*, 2024.

[16] H. Xia, Z. Zheng, Y. Li, D. Zhuang, Z. Zhou, X. Qiu, Y. Li, W. Lin, and S. L. Song, "Flash-llm: Enabling cost-effective and highly-efficient large generative model inference with unstructured sparsity," *arXiv preprint arXiv:2309.10285*, 2023.

[17] R. Fan, X. Yu, P. Dong, Z. Li, G. Gong, Q. Wang, W. Wang, and X. Chu, "Spinfer: Leveraging low-level sparsity for efficient large language model inference on gpus," in *Proceedings of the Twentieth European Conference on Computer Systems*, 2025, pp. 243–260.

[18] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. de las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. R. Lavaud, M.-A. Lachaux, P. Stock, T. L. Scao, T. Lavril, T. Wang, T. Lacroix, and W. E. Sayed, "Mistral 7b," 2023. [Online]. Available: https://arxiv.org/abs/2310.06825

[19] N. Zheng, B. Lin, Q. Zhang, L. Ma, Y. Yang, F. Yang, Y. Wang, M. Yang, and L. Zhou, "{SparTA}:{Deep-Learning} model sparsity via {Tensor-with-Sparsity-Attribute}," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 213–232.

[20] T. Gale, M. Zaharia, C. Young, and E. Elsen, "Sparse gpu kernels for deep learning," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–14.

[21] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin *et al.*, "Opt: Open pre-trained transformer language models," *arXiv preprint arXiv:2205.01068*, 2022.

[22] NVIDIA, "cuBLAS Library," https://docs.nvidia.com/cuda/cublas/, 2024.

[23] ——, "Nsight Compute," https://developer.nvidia.com/nsight-compute, 2024.

[24] S. Merity, C. Xiong, J. Bradbury, and R. Socher, "Pointer sentinel mixture models," *arXiv preprint arXiv:1609.07843*, 2016.

[25] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *Journal of machine learning research*, vol. 21, no. 140, pp. 1–67, 2020.