

SSR: Sparse Segment Reduction for Ternary GEMM Acceleration

Adeline Pittet, Shien Zhu, Valérie Verdan, Gustavo Alonso
Systems Group, ETH Zurich, Switzerland

apittet@ethz.ch, shien.zhu@inf.ethz.ch, verdanv@student.ethz.ch, alonso@inf.ethz.ch

Abstract—Large Language Models (LLMs) require substantial computational resources, limiting their deployment on resource-constrained hardware. Ternary LLMs mitigate these demands through weight quantization via ternary values, achieving significant compression often with 50 – 90% sparsity. However, existing approaches have limitations: methods optimized for ternary weights, such as BitNet, redundant segment reduction (RSR), and its improved version RSR++, do not exploit sparsity structures, while conventional sparse formats neglect ternary characteristics, foregoing dual optimization opportunities.

In this paper, we introduce *Sparse Segment Reduction (SSR)*, a ternary matrix multiplication method designed to accelerate the inference of ternary LLMs and general Ternary Weight Networks (TWNs). SSR has a dedicated optimized ternary data format and an algorithm that systematically exploits sparsity patterns through computation trees that scale with the sparsity. SSR provides theoretical gains with asymptotically faster inference than RSR++ for sparsity above 50%, while practical evaluations reveal performance improvements across all sparsity levels. Evaluation results show that SSR achieves 2.1-11.3× speedup over RSR++ on ternary GEMM with 45-95% sparsity. Furthermore, SSR achieves 3.5-6.3× end-to-end speedup and 4.9% of memory saving over RSR++ on the Llama-3 1B model inference.

Index Terms—Sparse GEMM, Ternary LLM, Edge Computing

I. INTRODUCTION

Large Language Models (LLMs) have emerged as a powerful tool across many domains, yet their substantial computational and memory requirements limit their usage on resource-constrained devices [1]. Quantization addresses this [2], [3] by reducing bit precision of weights and activations, decreasing the memory usage, the energy consumption, and the inference time while aiming to close the accuracy gap. For example, Nvidia and AMD GPUs adopt FP8, FP6, and even FP4 quantization for LLM inference.

Binary Weight Networks (BWNs) constrain weights to $\{-1, +1\}$ for extreme compression, which introduces an accuracy drop and requires dense computation [4]. *Ternary Weight Networks (TWNs)* expand the weight space to $\{-1, 0, +1\}$, enabling higher accuracy and efficiency while introducing sparsity [5], [6]. Compared with FP8 quantization, TWNs achieve a 4× compression and replace the multiplication operations in GEMM with lightweight addition and subtraction operations [5], [7]. *Ternary-Weight LLMs* apply TWNs to large language models, reducing size and accelerating inference [8]. Ternary-Weight LLMs inherit the features of TWNs in the LLM era and share the same addition-based sparse GEMM on ternary weights and dense activations.

Ternary LLM and general TWNs exhibit high sparsity which frequently exceeds 50% and even surpasses 90% in practice [9]. For example, induced weight sparsity by learning binary gate variables reaches 95.84% [10], Faraone et al. achieve 97.6% sparsity in ternary networks using quantization threshold regularization and pruning [11], and up to 99.58% has been reported in recent work [9]. Since zero-valued weights contribute nothing to matrix multiplications, this inherent sparsity provides significant opportunities for computational and memory optimizations. Conservative methods typically target around 50% sparsity for balanced trade-offs [12], [13], whereas in practice 90–95% sparsity is achievable without notable accuracy loss [9].

However, the ternary-weight dense-activation sparse GEMM (ternary GEMM) is under-optimized. Traditional sparse formats such as CSR, JDS, and COO [14]–[16] are designed for general scientific computing and cannot exploit the structural properties of ternary networks. Similarly, general-purpose libraries such as Eigen and PyTorch Sparse provide optimized sparse GEMM kernels but do not exploit the ternary structure of TWNs. Consequently, existing inference frameworks fail to translate the given structural properties into meaningful computational benefits. For example in our experiments, PyTorch Sparse GEMM based on CSC is slower than PyTorch Dense GEMM with 90% or lower sparsity on CPUs, which is much lower than the 10× theoretical speedup of sparse GEMM. Though related works have proposed algorithms that use the ternary structure such as BitNet.cpp [17], RSR, and RSR++ [18], they do not fully exploit the sparsity feature of ternary weights to reduce the computation complexity. In addition, their data formats have redundant values that brings larger memory footprint in ternary GEMM.

In this paper, we propose *Sparse Segment Reduction (SSR)* to achieve efficient ternary GEMM for ternary LLMs and general TWNs. We achieve asymptotic improvements over BitNet [17] and RSR++ [18] in ternary GEMM independent of hardware or structured sparsity constraints. SSR systematically leverages the inherent sparsity patterns of TWNs through preprocessing stages that exclude zero-pattern elements, yielding compact computation trees that scale well with sparsity. During inference, these optimizations eliminate redundant operations and reduce memory traffic, achieving asymptotically superior performance for matrices with sparsity levels exceeding 50%, while delivering practical performance gains across low and high sparsity levels compared to RSR++ [18].

We implement the SSR algorithm with code optimizations like unrolling, AVX2 SIMD, and multi-threading on CPUs (code is available at [19]). Evaluation results show that SSR achieves 2.1-11.3 \times speedup over RSR++ on ternary GEMM with 45-95% sparsity. SSR also achieves up to 59.4% faster speed over PyTorch Sparse GEMM. SSR achieves 1.8-7.5 \times speedup up to 17.8% of memory saving over RSR++ at the layer level on Llama-MLP blocks. Furthermore, SSR achieves 3.5-6.3 \times end-to-end speedup and 4.9% memory saving over RSR++ on Llama-3 1B model inference.

II. BACKGROUND AND RELATED WORKS

A. Target Problem

We target ternary LLMs and general Ternary-Weight Deep Neural Networks, and Binary Weight Networks (BWNs) can be seen as a special case of TWNs. Let $X \in \mathbb{R}^{M \times K}$ denote a dense activation matrix and $W \in \{-1, 0, 1\}^{K \times N}$ a ternary weight matrix with a given sparsity level p . The primary computation in these networks involves the matrix product

$$Y = X \cdot W,$$

which forms the core of the inference workload. We represent the elements, rows, and columns of a weight matrix W using $W[i, j]$, $W[i, :]$, and $W[:, j]$, respectively. And \bar{x}_j is the j -th row of a matrix X . We denote the sparsity level of a matrix by p , representing the fraction of zero weights. Let σ_n be a permutation of n elements in $[n]$, i.e., a bijective mapping. Finally, let pattern_0 be the random variable counting the number of rows in a matrix that match the all-zero bit pattern.

B. Quantized Neural Networks

Quantization of neural networks has evolved from binary weights [4], [20], [21] to ternary weights [5], [9], [22], with recent work extending these approaches to large language models [8]. Advances also include hardware-friendly implementations [23] and new training methods for ternary networks [24]. Hardware-optimized solutions for quantized networks include specialized accelerators using low-bit arithmetic [25]–[27], configurable FPGAs [28], and symmetry-aware in-memory computing [29]. Optimizations for general-purpose platforms leverage ARM NEON SIMD [30], GPUs [31], and GPU Tensor Cores for sparse ternary operations [32].

C. Ternary LLM Inference

Ternary LLMs such as BitNet b1.58 [33] achieve full-precision performance using ternary weights $\{-1, 0, 1\}$, offering a 2.71 \times speedup and 3.55 \times memory reduction, demonstrating the potential of extreme quantization for large language models. Current state-of-the-art ternary GEMM methods include T-MAC [26], which uses lookup tables to achieve 4–5 \times speedups via precomputed partial sums, and RSR++ [18], which achieves logarithmic theoretical complexity improvement through redundant segment reduction. However, RSR++ faces significant practical limitations: it ignores weight sparsity by treating zeros as non-zero weights, transfers large amounts of unnecessary data via dense permutation arrays, performs

many computations on elements that do not affect the output, and causes non-linear memory access patterns that degrade cache performance and increase computational overhead.

III. METHOD

We introduce SSR, a new sparsity-aware method inspired by RSR++ that constructs block-wise indices excluding zero-pattern elements during preprocessing, enabling compact computation trees. During inference, these tree indices provide direct access to non-zero elements, avoiding redundant computations. Both stages are shown in Fig. 1.

A. Pre-processing

Given a weight matrix $W \in \{-1, 0, +1\}^{K \times N}$, we decompose W into two binary matrices W_1 and W_2 such that $W = W_1 - W_2$, where $W_1[i, j] = 1$ if $W[i, j] = +1$, otherwise 0; similarly, $W_2[i, j] = 1$ if $W[i, j] = -1$, otherwise 0. Then W_1 and W_2 go through the following preprocessing steps. See Fig. 1 for an example with $W = W_1 - W_2$.

1) *Column Blocking*: First, we partition a binary matrix into consecutive column blocks of L columns each. The last block may contain fewer columns in certain cases.

Definition 1 (*L-Column Block*). Given a binary matrix $W \in \{0, 1\}^{K \times N}$ and a blocking factor $L \in \mathbb{N}$, partition W into column blocks $W_i^{[L]}$ for $i \in \{1, \dots, \lceil N/L \rceil\}$, where each block $W_i^{[L]}$ consists of columns $W[:, (i-1) \cdot L]$ to $W[:, \min(i \cdot L - 1, N - 1)]$.

If L is clear from the context, we simply write W_i instead of $W_i^{[L]}$ to denote a L -column block. In Fig. 1 with $L = 3$, we partition W_2 into the column blocks $W_{2,1}$ and $W_{2,2}$.

2) *Permutation*: To efficiently aggregate grouped binary patterns, we process the column blocks W_i of a binary matrix according to a permutation order that sorts the binary patterns in lexicographic ascending order.

Definition 2 (*Permutation Sequence*). Let $W_i \in \{0, 1\}^{K \times L}$ be a binary matrix. For each row $r \in \{0, \dots, K-1\}$ of W_i define its *key value* $\kappa_i(r)$ as the integer value obtained by taking the concatenation of $W_i[r, 0], \dots, W_i[r, L-1]$. The *permutation sequence* of W_i , denoted by σ_i , is a permutation of $\{0, \dots, L-1\}$ such that

$$\kappa_i(\sigma_i(0)) \leq \kappa_i(\sigma_i(1)) \leq \dots \leq \kappa_i(\sigma_i(L-1)).$$

Essentially, the permutation sequence σ_i for a column block W_i specifies how the rows of W_i should be reorganized so that their binary sequences, read as integer values, are arranged in ascending order. In Fig. 1, $W_{2,1}$ has binary patterns 000, 001, and 101 from rows 2, 1, and 0. Their ascending order ($000 < 001 < 101$) gives the row permutation $\sigma_{2,1} = \{2, 1, 0\}$.

Definition 3 (*Reduced Permutation Sequence*). Given a permutation sequence σ_i of a column block W_i , the *Reduced Permutation Sequence* σ_i' is obtained by removing a prefix of σ_i . Formally, let q be the number of all-zero rows in W_i . For all $j = 0, \dots, q-1$, we have

$$\kappa_i(\sigma_i(j)) = 0,$$

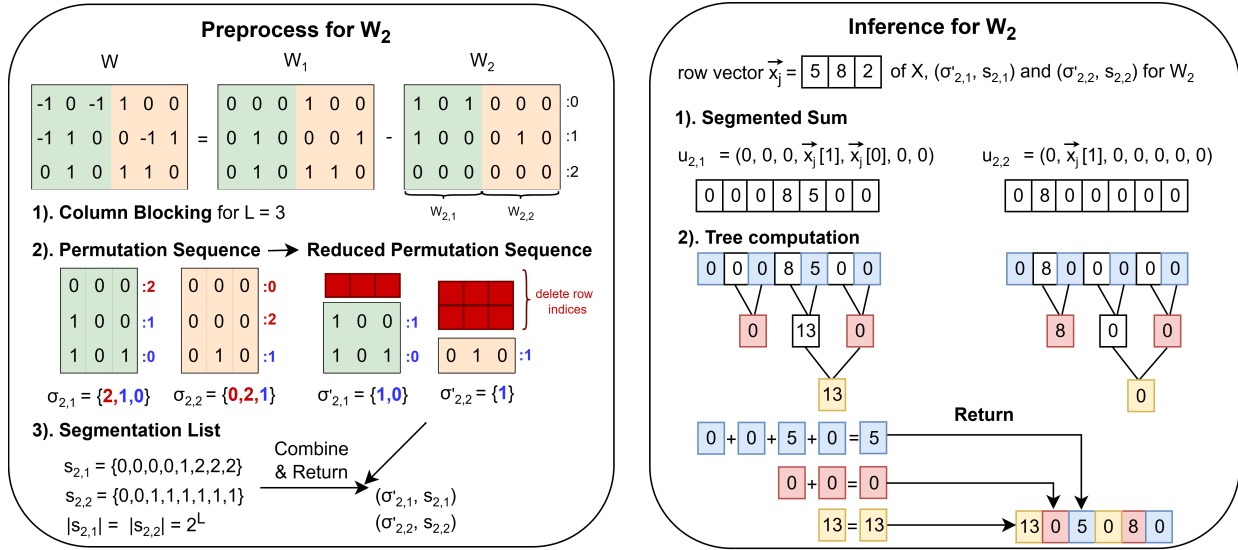


Fig. 1: Pre-processing of the binary matrix W_2 and inference with the row vector \vec{x}_j of the input matrix X for $L = 3$. The length of the reduced permutation sequence equals the number of nonzero patterns in the column blocks: 2 for $W_{2,1}$ and 1 for $W_{2,2}$. Row indices of all-zero rows are deleted and shown in red, while the remaining row indices are shown in blue.

and σ'_i is the subsequence of σ_i obtained by excluding these first q indices.

For the example in Fig. 1, $W_{2,1}$ has the permutation sequence $\sigma_{2,1} = \{2, 1, 0\}$. Since the matrix contains a single row with the all-zero pattern (row 2), this index can be removed to form the reduced permutation $\sigma'_{2,1} = \{1, 0\}$.

3) *Segmentation*: To facilitate efficient computation on grouped binary patterns, we segment the sorted column blocks W_i using boundary indices that mark transitions between different binary keys.

Definition 4 (Segmentation List). Let W_i be a column block with reduced permutation sequence σ'_i , and let $\pi_{\sigma'_i}(W_i)$ be the permuted block. The *Segmentation Sequence* s_i consists of 2^L bounding indices of the interval containing the pattern

$$s_i = \{b_0, \dots, b_{2^L-1}\},$$

such that, in the sorted smaller matrix $\pi_{\sigma'_i}(W_i)$, for all $j \in \{0, \dots, 2^L - 2\}$ and $r \in [b_j, b_{j+1})$, we have $\kappa_i(r) = j + 1$, as we do not consider the all-zero pattern.

A segment s encodes binary patterns as cumulative counts, where consecutive differences give the number of occurrences of each pattern. For $W_{2,1}$ in Fig. 1, ignoring the all-zero pattern 000, we get $s = [0, 0, 0, 0, 1, 2, 2, 2]$ representing cumulative counts for the bit pattern 001 to 111: the first four positions remain 0 (no pattern 001–011), positions 4–5 increase from 0 to 1 to account for the single occurrence of 100, positions 5–6 go from 1 to 2 for the single occurrence of 101, and the remaining positions stay 2 (no further patterns).

Algorithm 1 combines the three pre-processing steps. Given fixed ternary weights and block size L , preprocessing is applied once, and inference repeatedly reuses the resulting structures for runtime computation. The parameter L is selected empirically per machine; in practice, optimal values are

Algorithm 1 Preprocessing

- 1: **Input:** Binary matrix $W \in \{0, 1\}^{K \times N}$, factor $L \in \mathbb{N}$
- 2: $\{W_1, W_2, \dots, W_{\lceil N/L \rceil}\} \leftarrow$ Column Blocking of W
- 3: **for** each block W_i **do**
- 4: $\sigma_i \leftarrow$ Permutation Sequence of W_i
- 5: $\sigma'_i \leftarrow$ Reduced Permutation Sequence from σ_i
- 6: $s_i \leftarrow$ Segmentation Sequence of $\pi_{\sigma'_i}(W_i)$
- 7: **Return:** $\{(\sigma'_i, s_i) \mid i = 1, \dots, \lceil N/L \rceil\}$

small and strictly below $\log_2 N$ for the theoretical bound to hold. Further, optimal L is decreasing with higher sparsity and increasing with matrix size. Once determined, both preprocessing and inference run with a fixed configuration.

B. Inference

During inference, given an input matrix $X \in \mathbb{R}^{M \times K}$ with row vectors \vec{x}_j for $j \in \{0, \dots, M - 1\}$, we compute the matrix product XW by performing row-wise vector-matrix multiplications $\vec{x}_j \cdot W_i^{[L]}$, where $i \in \{1, \dots, \lceil N/L \rceil\}$, taking advantage of the preprocessed structure of each $W_i^{[L]}$. It involves two steps: segmented sum and tree computation.

1) *Segmented Sum*: Given $\vec{x}_j \in \mathbb{R}^K$ and (σ'_i, s_i) for $W_i^{[L]}$, we compute the segmented sum.

Definition 5 (Segmented Sum). Let σ'_i be the reduced permutation and s_i the segmentation sequence of $W_i^{[L]}$. The *segmented sum* \vec{u}_i is defined as

$$\vec{u}_i[\ell] = \sum_{r=b_\ell}^{b_{\ell+1}-1} \vec{x}_j[\sigma'_i(r)], \quad \ell \in \{0, \dots, 2^L - 2\},$$

where $s_i = \{b_0, \dots, b_{2^L-1}\}$.

Each entry of \vec{u}_i thus aggregates the contributions of all elements in a row \vec{x}_j sharing the same binary pattern in a column block $W_i^{[L]}$.

2) *Tree Computation*: The tree computation can be viewed as a recursive reduction that accumulates contributions from elements according to the binary patterns of their positions.

Definition 6 (Tree Computation). Let $L \in \mathbb{N}$ and let \vec{u}_i be the segmented sum vector of length $2^L - 1$. Define a sequence of vectors $\vec{v}_0, \vec{v}_1, \dots, \vec{v}_{L-1}$ recursively as follows:

$$\begin{aligned} \vec{v}_0 &:= \vec{u}_i, \\ \vec{v}_{\ell+1} &:= (\vec{v}_\ell[2], \vec{v}_\ell[4], \dots, \vec{v}_\ell[|\vec{v}_\ell| - 1]) \text{ for } \ell = \{0, \dots, L-2\}. \end{aligned}$$

The *tree computation output* is the concatenation of the sums of the even-indexed entries of the vectors in reverse order:

$$\vec{z}_i := \left(\sum_{j=0}^{\lfloor \frac{|\vec{v}_{L-1}|}{2} \rfloor} \vec{v}_{L-1}[2j], \dots, \sum_{j=0}^{\lfloor \frac{|\vec{v}_0|}{2} \rfloor} \vec{v}_0[2j] \right).$$

The vectors $\vec{v}_0, \vec{v}_1, \dots, \vec{v}_{L-1}$ have odd length, with $\vec{v}_0 = \vec{u}_i$ of length $2^L - 1$. For $\ell \geq 0$, $\vec{v}_{\ell+1}$ is formed recursively by taking every second element of \vec{v}_ℓ from positions 2 to $|\vec{v}_\ell| - 1$, roughly halving the length until $|\vec{v}_{L-1}| = 1$. The tree output \vec{z}_i concatenates the even-index sums of each vector in the sequence, i.e. at each level in the tree, in reverse order, as illustrated in Fig. 1. The corresponding sums are shown in blue, red, and yellow and the concatenation in reverse order.

Algorithm 2 Inference

- 1: **Input:** $X \in \mathbb{R}^{M \times K}$ and $\{(\sigma'_i, s_i)\}$ for all $W_i^{[L]}$
 - 2: **for** each row vector $\vec{x}_j \in \mathbb{R}^K$ **do**
 - 3: **for** each (σ'_i, s_i) corresponding to $W_i^{[L]}$ **do**
 - 4: $\vec{u}_i \leftarrow$ Segmented Sum of \vec{x}_j using (σ'_i, s_i)
 - 5: $\vec{z}_i \leftarrow$ Tree Computation of \vec{u}_i
 - 6: $\vec{Z}_j \leftarrow (\vec{z}_1, \dots, \vec{z}_{\lceil N/L \rceil})$
 - 7: **Return:** $\vec{Z} \in \mathbb{R}^{M \times N}$, where the j -th row is \vec{Z}_j
-

C. Theoretical Computation Complexity Analysis

The proposed SSR improves inference efficiency and achieves asymptotically better operational complexity than RSR++ for sparsity $p > 0.5$, as it requires fewer additions and less memory accesses. The preprocessing complexity of SSR remains $O(K \cdot N)$ like in RSR++.

TABLE I: Operation counts of SSR and related works.

BitNet.cpp	RSR++	SSR
$\frac{2MNK}{3}$	$\frac{2MNK}{\log_2 K}$	$\log_2 \left(\frac{1}{1-p} \right) \cdot \frac{4MN \cdot K^{-1/\log_2(1-p)}}{\log_2 K}$

RSR++ requires, in the worst case, $\frac{2MNK}{\log_2(K+2)-1}$ operations [18], which for $N \geq 2$ gives the lower bound $2 \cdot \frac{MNK}{\log_2 K}$, as shown in Table I. SSR exploits that elements mapped to all-zero bit patterns contribute nothing to the final result, allowing their exclusion from computation. Hence, for each row and block $W_i^{[L]}$, the number of additions drop from K to

$K - \mathbb{E}[\text{pattern}_0]$, with $\mathbb{E}[\text{pattern}_0] = p^L \cdot K$ assuming entries are independently zero with probability p . Setting $L = \left\lceil \log_2 K / \log_2 \left(\frac{1}{1-p} \right) \right\rceil$ and following the derivation in [18] yields an upper bound on the computational cost, as shown in Table I. The performance difference between RSR++ and SSR arises from both the exponent of K and the constant factor. For $p > 0.5$, we have:

$$-\log_2(1-p) > 1 \quad \Rightarrow \quad \frac{-1}{\log_2(1-p)} < 1$$

Hence, SSR achieves strictly better asymptotic performance for $p > 0.5$, coinciding with RSR++ at $p = 0.5$. While SSR has a larger constant factor, the reduced exponent dominates for sufficiently large K . Specifically, for sparsity $p \in (0.5, 1]$, SSR is more efficient when

$$K \geq (2c)^{\frac{c}{c-1}} \text{ with } c = \log_2 \left(\frac{1}{1-p} \right).$$

SSR further reduces computation and memory access. Excluding the all-zero bit pattern shortens the segmented sum array by one entry, saving one addition per row and per column block $W_i^{[L]}$ at each tree level, i.e., at least $M \cdot \frac{N}{L}$ additions. Moreover, the permutation array and segmentation list are shortened, reducing both the number of indices processed and the corresponding memory accesses.

Similar to BitNet [17] and RSR++ [18], our SSR algorithm is independent of the underlying hardware architecture. Therefore, SSR can be implemented on both CPUs and GPUs. We provide the benchmarking results on CPU only in the next section because of the high engineering effort required.

IV. EVALUATION

We optimize the C++ implementation of SSR with loop unrolling, blocking, AVX2 SIMD intrinsics, and multi-threading to achieve faster speed. We benchmark the SSR CPU implementation with ternary weight matrices at the GEMM level, the layer level, and the model level. The experiments were conducted on a Windows 11 Home 23H2 laptop with an AMD Ryzen 7 8845HS@3.8GHz. The compiler is Microsoft Visual Studio 2022 v143. All experiments are executed for 10 times and we use the average time to draw the graphs. We extend the experiments to Llama Multi-Layer Perceptron (MLP) block and Llama models to systematically evaluate the performance gain of our implementation.

A. Ternary Matrix Multiplication Evaluation

We perform a matrix-level evaluation against RSR and RSR++ [18] for the single-threaded implementation, then against more related work for the multi-threaded one. Figure 2 presents the ternary GEMM for both the basic SSR implementation and the AVX2-vectorized version. These experiments are conducted with single-threaded code. The activation matrix has dimensions 512×1024 , and the ternary weight matrix has dimensions 1024×4096 . RSR and RSR++ are insensitive to sparsity, as they have almost the same execution time across different sparsity levels. In contrast, our SSR and Eigen

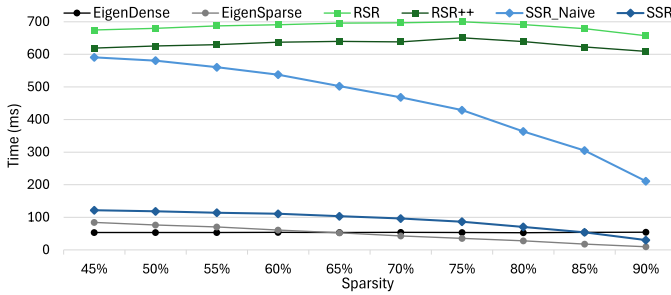


Fig. 2: Execution time of single-thread ternary GEMM (X: 512×1024 , W: 1024×4096 , L=6).

Sparse GEMM have shorter execution time with the increasing sparsity consistently.

RSR++ consistently has 7.6%-9.1% improvement over RSR. Our SSR Naive implementation has 2.9%-189.0% faster speed over RSR++ thanks to the sparsity. The optimized SSR implementation achieves 4.9-20.2 \times speedup over RSR++ when the sparsity ranges from 45%-95%. Eigen Dense and Sparse GEMM serve as a very good baseline. Our optimized SSR implementation with AVX2 SIMD achieves similar speed as the Eigen Sparse but cannot outperform it.

Next, we evaluate the multi-threaded, AVX2-vectorized SSR implementation against established baselines, including PyTorch Dense, PyTorch Sparse CSR and CSC, as well as Eigen Dense and Eigen Sparse implementations.

We conducted both float and INT8 precision for all experiments. However, the INT8 version of PyTorch and Eigen, including Dense and Sparse GEMM, performed worse than expected. Their INT8 implementations are much slower than their FP32 versions, making these experiment results unusable. For example, the Eigen Sparse GEMM INT8 is 2.2-4.5 \times slower than its FP32 version, while others may be even slower than PyTorch Sparse CSR. Therefore, though our INT8 speedup is relatively better than the speedups in FP32, we exclude these INT8 benchmarking results and only provide Eigen Sparse GEMM INT8 for reference.

Figure 3 shows the ternary GEMM time with a small weight matrix of size 1024×4096 . PyTorch Dense GEMM is a strong baseline as it has been optimized by many engineers and widely applied in the industry. However, the PyTorch Sparse CSR (Compressed Sparse Row) performs worse than other sparse methods. It only outperforms RSR++ with 75% or higher sparsity. Our SSR achieves 14.0%-43.8% faster speed

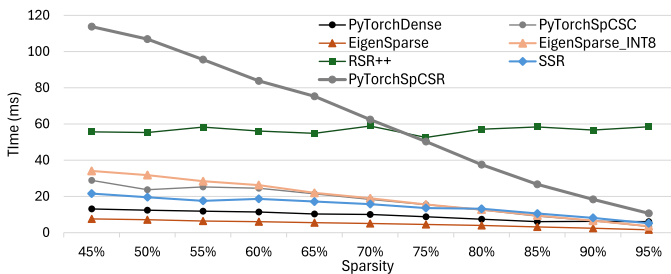


Fig. 3: Execution time of multi-thread ternary GEMM (X: 256×1024 , W: 1024×4096 , L=4).

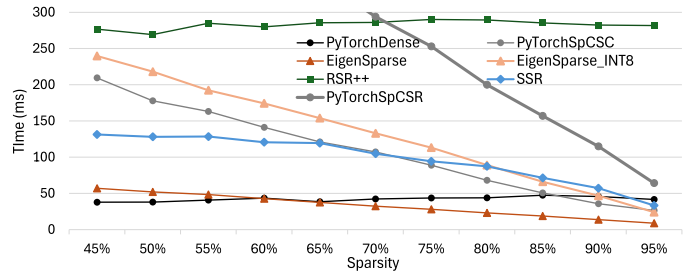


Fig. 4: Execution time of multi-thread ternary GEMM (X: 512×2048 , W: 2048×8192 , L=6).

than PyTorch Sparse CSC (Compressed Sparse Column) with 45%-75% sparsity, but is up to 35% slower than it with very high sparsity. Similarly, SSR is 14.7%-62.2% faster than Eigen Sparse INT8 GEMM when the sparsity is lower than 80%. As RSR++ has similar execution time for different sparsity levels, our SSR achieves 2.6-11.3 \times speedup over RSR++.

Fig. 4 shows ternary GEMM with a larger weight matrix, zoomed in to highlight performance trends. Eigen Sparse GEMM tends to perform better on smaller matrices, as its relative speed compared with PyTorch Dense GEMM shifts slower compared with Fig. 3. The trend is similar to the previous graph. SSR is up to 59.4% and 82.5% faster than PyTorch Sparse CSC and Eigen Sparse INT8 with 70% or lower sparsity, but becomes slower at higher sparsity. SSR achieves 2.1-8.4 \times speedup over RSR++ for 45-95% sparsity, due to large matrix sizes ($N \gg 2$) and lower memory footprint, which we evaluate later in layer-level benchmarking.

The performance curves in Fig. 3 and Fig. 4 show that our SSR seems to do not utilize the sparsity as good as other methods. The reason is that these experiments use a fixed L factor. Based on our experiments, the optimal L value changes with both the matrix size and the sparsity level. Therefore, dynamically selecting the L value based on a performance profiling can overcome this limitation.

B. Layer-Level Evaluation

We evaluate the ternary GEMM performance at Llama-3 Multi-Layer Perceptron blocks. As one MLP block contains one up projection (e.g., up_proj in Fig. 5: 1024×4096), one gate projection (1024×4096), and one down projection (4096×1024), it reflects how the GEMM performance contributes to the layer-level performance. The activation function like GeLU or SiLU are omitted here to only evaluate the ternary GEMMs. The sizes of other projections changes with the up projection accordingly in Fig. 6.

Fig. 5 presents the execution time of one small MLP block. Though the MLP block execution time is about 2.5-3 \times as long as one single GEMM across different methods, it shared similar trend as the GEMM evaluation results. SSR is up to 75.8% faster but up to 31.9% slower than PyTorch CSC, which shows a better scaling curve across the sparsity than the single GEMM case. The crossing point is still below 80% sparsity. SSR has 2.6-7.5 \times speedup over RSR++ at this MLP block.

Fig. 6 presents the execution time of one Llama-3 1B MLP block, which is about 8 \times as large as the small MLP block.

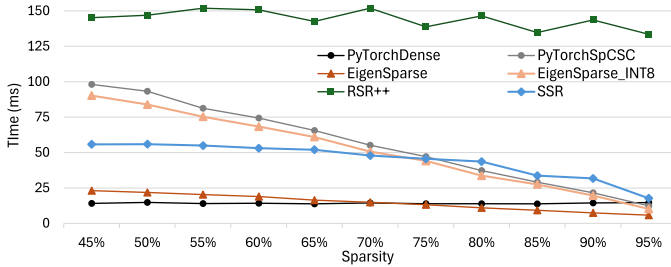


Fig. 5: Execution time of one small Llama MLP block (X: 256×1024 , up_proj: 1024×4096 , L=6).

Similar to the GEMM benchmarking, the Eigen Sparse INT8 has a worse speed than the small case, and is 13.7-34.2% slower than the PyTorch CSC. SSR is up to 38.5% faster and 31.5% slower than PyTorch CSC. Compared with PyTorch Dense, it achieves faster speed at 95% sparsity, which is the same as PyTorch CSC and Eigen Sparse INT8. SSR achieves 1.8-7.5 \times speedup over RSR++.

TABLE II: Memory usage of one Llama-3 1B MLP block

	After Initialization	During Inference
RSR++	129-146 MB	177-259 MB
SSR	122-124 MB	160-213 MB

In addition to the speed analysis, we provide the memory usage of the MLP block in Table II. The memory usage after initialization shows the data format size that store the ternarized weights. Compared with RSR++, SSR reduces 5.4-15.1% of memory thanks to the efficient data format that removes redundant all-zero indices. The memory usage during inference shows the dynamic memory in execution. As SSR can utilize the sparsity to skip computation with zero weights, we reduces the memory usage by 9.6-17.8% compared with RSR++. Such a smaller memory footprint not only enhances performance but also lowers energy consumption due to fewer memory accesses

C. End-to-End Evaluation

We report the Time-To-First-Token latency of a ternary llama-3.1 1B model in Table III. The MLP blocks of the llama model are replaced with corresponding ternary MLP blocks, and the activations like SiLU are added back to these MLP blocks. The attention part are implemented using libTorch C++ APIs with dedicated PyTorch native Multi-Head-Attention functions to maximize its efficiency. We set the

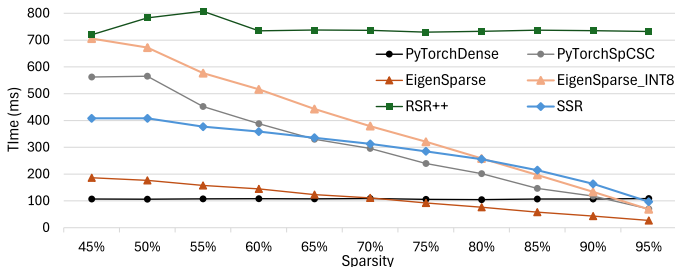


Fig. 6: Execution time of one Llama-3 1B MLP block (X: 512×2048 , up_proj: 2048×8192 , L=6).

TABLE III: Time-To-First-Token of one Llama-3.1 1B model (Sequence Length = 512)

Sparsity	50%	70%	90%
RSR++	36.23 s	36.30 s	36.19 s
SSR	10.49 s	8.80 s	5.71 s

sequence length to 512 to reflect a relative long user query to that language model.

The Time-To-First-Token latency of RSR++ ternary Llama is very stable across the tested sparsity levels, as it does not exploit sparsity to remove redundant computations. In contrast, our SSR achieves lower latency and faster inference, with a 1.84 \times speedup at 90% sparsity compared to at 50%. This relative speedup is smaller than the 5 \times theoretical speedup because the attention layers are not ternarized. Compared with RSR++, SSR achieves a 3.45-6.34 \times speedup for long queries with 512 tokens, with further improvements possible by ternarizing Multi-Head Attention and dynamically selecting L instead of using a fixed value.

At the full model level, RSR++ requires up to 3113 MB to execute the ternary LLaMA-3.1 1B model with a sequence length of 512, whereas SSR reduces this to 2960 MB, achieving a consistent memory overhead reduction as the GEMM-level and layer-level benchmarking results. The 4.9% of memory reduction percentage is smaller than the MLP blocks due to the addition of the attention functions.

V. CONCLUSION

In this paper, we present Sparse Segment Reduction (SSR) as an optimized ternary GEMM method for ternary LLMs and general TWNs. Related works BitNet [17] RSR++ [18] reduce the complexity of ternary GEMM but overlook zero values. Therefore, we leverage the zero weights in the ternary GEMM to reduce storage and eliminate redundant computations. SSR features a specialized compact data format that reduces up to 15% memory size than RSR++ and a sparse GEMM algorithm that achieves another log-scale improvement on the computation complexity. We present the pre-processing algorithm for the data format and the inference algorithm with tree-reduction in mathematic equations. We also mathematically show the computation complexity of SSR and the boundary conditions. We implement the proposed SSR in C++ and optimize the code with standard optimizations like loop unrolling, AVX2 SIMD, and multi-threading. Evaluation results show that SSR achieves a speed up to 75.8% faster than PyTorch Sparse CSC and a speedup of 2.1-11.3 \times over RSR++ in the ternary GEMM. SSR achieves 3.5-6.3 \times end-to-end speedup over RSR++ on the ternary Llama-3 1B model with 4.9% of total memory reduction.

We hope that this work can facilitate the adoption of ternary LLMs and general TWNs. The limitation of this work is the lack of implementation for GPUs and more LLM architectures. This points to future work in GPU implementation and inference demonstration on open-weight ternary LLMs. Given the strong performance of related works BitNet and RSR++ on GPUs, SSR is also expected to yield further improvements with GPU-specific optimizations.

REFERENCES

- [1] L. Seymour, B. Kutukcu, and S. Baidya, "Large language models on small resource-constrained systems: Performance characterization, analysis and trade-offs," 2024. [Online]. Available: <https://arxiv.org/abs/2412.15352>
- [2] 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," in *Proceedings of Machine Learning and Systems*, P. Gibbons, G. Pekhimenko, and C. D. Sa, Eds., vol. 6, 2024, pp. 87–100. [Online]. Available: https://proceedings.mlsys.org/paper_files/paper/2024/file/42a452cbafa9dd64e9ba4aa95cc1ef21-Paper-Conference.pdf
- [3] J. Kong, L. Hu, F. Ponzina, and T. Rosing, "Tinyagent: Quantization-aware model compression and adaptation for on-device LLM agent deployment," in *Workshop on Efficient Systems for Foundation Models II @ ICML/2024*, 2024. [Online]. Available: <https://openreview.net/forum?id=ntf0eq0urB>
- [4] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Advances in Neural Information Processing Systems*, C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, Eds., vol. 28. Curran Associates, Inc., 2015. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2015/file/3e15cc11f979ed25912dff5b0669f2cd-Paper.pdf
- [5] F. Li, B. Liu, X. Wang, B. Zhang, and J. Yan, "Ternary weight networks," 2022. [Online]. Available: <https://arxiv.org/abs/1605.04711>
- [6] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained ternary quantization," in *International Conference on Learning Representations*, 2017.
- [7] R. Razani, G. Morin, E. Sari, and V. P. Nia, "Adaptive binary-ternary quantization," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2021, pp. 4613–4618.
- [8] S. Ma, H. Wang, L. Ma, L. Wang, W. Wang, S. Huang, L. Dong, R. Wang, J. Xue, and F. Wei, "The era of 1-bit llms: All large language models are in 1.58 bits," 2025, equal contribution. Corresponding author: Furu Wei. [Online]. Available: <https://arxiv.org/html/2402.17764v1>
- [9] X. Deng and Z. Zhang, "Sparsity-control ternary weight networks," *Neural Networks*, vol. 145, pp. 221–232, 2022. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0893608021004093>
- [10] S. Srinivas, A. Subramanya, and R. Venkatesh Babu, "Training sparse neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, July 2017.
- [11] J. Faraone, N. Fraser, G. Gambardella, M. Blott, and P. H. W. Leong, "Compressing low precision deep neural networks using sparsity-induced regularization in ternary networks," in *Neural Information Processing*, D. Liu, S. Xie, Y. Li, D. Zhao, and E.-S. M. El-Alfy, Eds. Cham: Springer International Publishing, 2017, pp. 393–404.
- [12] C. Jin, H. Sun, and S. Kimura, "Sparse ternary connect: Convolutional neural networks using ternarized weights with enhanced sparsity," in *2018 23rd Asia and South Pacific Design Automation Conference (ASPDAC)*, 2018, pp. 190–195.
- [13] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained ternary quantization," 2017. [Online]. Available: <https://arxiv.org/abs/1612.01064>
- [14] TensorFlow Community, "Csr sparse matrix rfc," <https://github.com/tensorflow/community/blob/master/rfcs/20200519-csr-sparse-matrix.md>, 2020.
- [15] E. Montagne and A. Ekambaram, "An optimal storage format for sparse matrices," *Information Processing Letters*, vol. 90, no. 2, pp. 87–92, 2004. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0020019004000249>
- [16] NVIDIA, "Sparse matrix formats," 2024. [Online]. Available: https://docs.nvidia.com/nvpl/latest/sparse/storage_format/sparse_matrix.html
- [17] J. Wang, H. Zhou, T. Song, S. Cao, Y. Xia, T. Cao, J. Wei, S. Ma, H. Wang, and F. Wei, "Bitnet.cpp: Efficient edge inference for ternary llms," 2025. [Online]. Available: <https://arxiv.org/abs/2502.11880>
- [18] M. Dehghankar, M. Erfanian, and A. Asudeh, "An efficient matrix multiplication algorithm for accelerating inference in binary and ternary neural networks," 2025. [Online]. Available: <https://arxiv.org/abs/2411.06360>
- [19] [Online]. Available: <https://github.com/fpgasystems/ternaryLLM>
- [20] Y. Wang, J. Lin, and Z. Wang, "An energy-efficient architecture for binary weight convolutional neural networks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 2, pp. 280–293, 2018.
- [21] K. Huang, B. Ni, and X. Yang, "Efficient quantization for neural networks with binary weights and low bitwidth activations," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, 2019, pp. 3854–3861.
- [22] A. Ardakani, Z. Ji, S. C. Smithson, B. H. Meyer, and W. J. Gross, "Learning recurrent binary/ternary weights," 2019. [Online]. Available: <https://arxiv.org/abs/1809.11086>
- [23] Y. Boo and W. Sung, "Structured sparse ternary weight coding of deep neural networks for efficient hardware implementations," in *2017 IEEE International Workshop on Signal Processing Systems (SiPS)*, 2017, pp. 1–6.
- [24] H. Zheng, X. Bai, X. Liu, Z. M. Mao, B. Chen, F. Lai, and A. Prakash, "Learn to be efficient: Build structured sparsity in large language models," in *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, Eds., vol. 37. Curran Associates, Inc., 2024, pp. 101969–101991. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2024/file/b8f10193cab43d45df9bb810637333fd-Paper-Conference.pdf
- [25] H. Guo, W. Brandon, R. Cholakov, J. Ragan-Kelley, E. P. Xing, and Y. Kim, "Fast matrix multiplications for lookup table-quantized llms," 2025. [Online]. Available: <https://arxiv.org/abs/2407.10960>
- [26] S. Eetha, S. P.K., V. Pant, S. Vikram, M. Mody, and M. Purnaprajna, "Tilenet: Hardware accelerator for ternary convolutional neural networks," *Microprocessors and Microsystems*, vol. 83, p. 104039, 2021. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0141933121002106>
- [27] H. Park and K. Choi, "Cell division: weight bit-width reduction technique for convolutional neural network hardware accelerators," in *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, ser. ASPDAC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 286–291. [Online]. Available: <https://doi.org/10.1145/3287624.3287721>
- [28] A. Prost-Boucle, A. Bourge, and F. Pétrot, "High-efficiency convolutional ternary neural networks with custom adder trees and weight compression," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 11, no. 3, Dec. 2018. [Online]. Available: <https://doi.org/10.1145/3270764>
- [29] K. Ando, K. Ueyoshi, K. Orimo, H. Yonekawa, S. Sato, H. Nakahara, S. Takamaeda-Yamazaki, M. Ikebe, T. Asai, T. Kuroda, and M. Motomura, "Brein memory: A single-chip binary/ternary reconfigurable in-memory deep neural network accelerator achieving 1.4 tops at 0.6 w," *IEEE Journal of Solid-State Circuits*, vol. 53, no. 4, pp. 983–994, 2018.
- [30] A. Trusov, E. Limonova, D. Nikolaev, and V. V. Arlazarov, "Fast matrix multiplication for binary and ternary cnns on arm cpu," 2022. [Online]. Available: <https://arxiv.org/abs/2205.09120>
- [31] Z. Wang, "Sparsert: Accelerating unstructured sparsity on gpus for deep learning inference," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 31–42. [Online]. Available: <https://doi.org/10.1145/3410463.3414654>
- [32] Y. Ogiwara and H. Kawashima, "Sparse ternary matrix multiplication with tensor core for transformer," in *2024 Twelfth International Symposium on Computing and Networking Workshops (CANDARW)*, 2024, pp. 150–156.
- [33] S. Ma, H. Wang, L. Ma, L. Wang, W. Wang, S. Huang, L. Dong, R. Wang, J. Xue, and F. Wei, "The era of 1-bit llms: All large language models are in 1.58 bits," 2024. [Online]. Available: <https://arxiv.org/abs/2402.17764>