Bitnet.cpp: Efficient Edge Inference for Ternary LLMs

Jinheng Wang[1,4], Hansong Zhou[1,4], Ting Song[4], Shijie Cao[4], Yan Xia[4], Ting Cao[4], Jianyu Wei[2,4], Shuming Ma[4], Hongyu Wang[3,4], Furu Wei[4],
[1]Peking University, [2]University of Science and Technology of China,
[3]University of Chinese Academy of Sciences, [4]Microsoft Research

Abstract

The advent of 1-bit large language models (LLMs), led by BitNet b1.58, has spurred interest in ternary LLMs. Despite this, research and practical applications focusing on efficient edge inference for ternary LLMs remain scarce. To bridge this gap, we introduce Bitnet.cpp, an inference system optimized for BitNet b1.58 and ternary LLMs. Given that **mixed-precision matrix multiplication (mpGEMM)** constitutes the bulk of inference time in ternary LLMs, Bitnet.cpp incorporates a novel mpGEMM library to facilitate sub-2-bits-per-weight, efficient and lossless inference. The library features two core solutions: **Ternary Lookup Table (TL)**, which addresses spatial inefficiencies of previous bit-wise methods, and **Int2 with a Scale (I2_S)**, which ensures lossless edge inference, both enabling high-speed inference. Our experiments show that Bitnet.cpp achieves up to a 6.25x increase in speed over full-precision baselines and up to 2.32x over low-bit baselines, setting new benchmarks in the field. Additionally, we expand TL to element-wise lookup table (ELUT) for low-bit LLMs in the appendix, presenting both theoretical and empirical evidence of its considerable potential. Bitnet.cpp is publicly available at `https://github.com/microsoft/BitNet/tree/paper`, offering a sophisticated solution for the efficient and practical deployment of edge LLMs.

1 Introduction

In recent years, large language models have garnered widespread attention due to their exceptional performance across a variety of tasks. However, the growing demand for efficient deployment on edge devices, particularly driven by data privacy concerns, poses challenges due to the limited computational power and bandwidth of these devices.

To address these challenges, model compression techniques are frequently employed. Notable examples benefiting from such techniques include

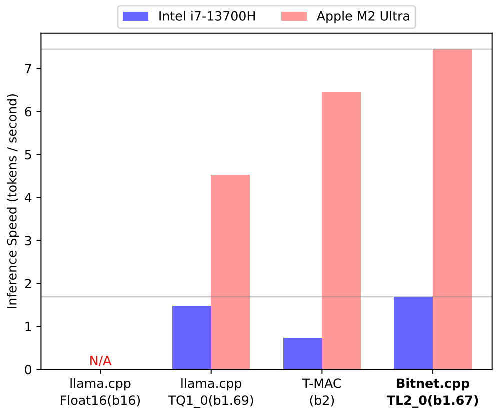
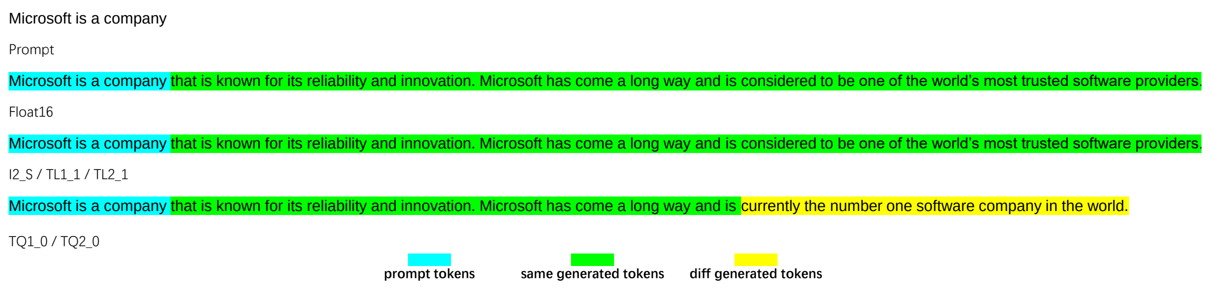


Figure 1: A comparison of end-to-end inference speeds on a 100B ternary LLM. (bx) represents the bits per weight, where x denotes specific value. "N/A" indicates that the tested CPU cannot host the specified model size with the given kernel.

Gemini-Nano (Anil et al., 2024) and Phi3-mini (Abdin et al., 2024), both designed for mobile and personal devices. Furthermore, recent advancements by BitNet b1.58 (Wang et al., 2023; Ma et al., 2024) represent a significant development in edge LLM inference, introducing 1-bit LLMs by quantizing all weights to ternary values therefore reducing the bits per weight (bpw) to 1.58, while preserving accuracy comparable to full-precision LLMs. Subsequent releases of ternary LLMs, including TriLM (Kaushal et al., 2024), Llama3-8B-1.58 (Mekkouri et al., 2024), and BitNet a4.8 (Wang et al., 2024a), have demonstrated the effectiveness and applicability of ternary architectures, thereby extending the boundaries of the 1-bit era. Despite the burgeoning interest in ternary LLMs, the conversion of their theoretical benefits into practical advantages during inference is still understudied.

To fill this gap, we aim to enhance the performance of ternary LLMs edge inference by optimizing mpGEMM (e.g., 8-bit activation and 1.58-bit weight). However, the non-integer bpw charac-


Microsoft is a company

Prompt

Microsoft is a company that is known for its reliability and innovation. Microsoft has come a long way and is considered to be one of the world's most trusted software providers.

Float16

Microsoft is a company that is known for its reliability and innovation. Microsoft has come a long way and is considered to be one of the world's most trusted software providers.

I2_S / TL1_1 / TL2_1

Microsoft is a company that is known for its reliability and innovation. Microsoft has come a long way and is currently the number one software company in the world.

TQ1_0 / TQ2_0

🟦 prompt tokens 🟩 same generated tokens 🟨 diff generated tokens

Figure 2: An example to show lossless inference for BitNet b1.58 with Bitnet.cpp.

teristic of ternary weights conflicts with the rules for computer memory access alignment, thus posing challenges in designing a sub-2-bit-per-weight, efficient edge mpGEMM for ternary LLMs. Currently, TQ1_0 in llama.cpp(lla) utilizes 1.69 bits to store ternary weights, but it is slower compared to TQ2_0 and T-MAC(Wei et al., 2024), which use 2 bits to maintain alignment. Moreover, prior implementations of mpGEMM have not achieved lossless inference for BitNet b1.58, as they fail to fully align with BitNet b1.58 training schemes during inference.

To address these issues, we develop Bitnet.cpp, which incorporates a novel mpGEMM library. Our key idea is to avoid intricate bit-level manipulations by directly operating the weight elements when designing mpGEMM, while strictly aligning with BitNet b1.58 training schemes. Based on our ideas, the library not only resolves spatial inefficiencies, but also surpasses existing solutions in terms of performance (Figure 1), achieving lossless inference for BitNet b1.58 (Figure 2). To this end, our work makes several contributions:

- First, we conduct a comprehensive survey of current cutting-edge mpGEMM methods and identify their limitations when applied to ternary LLMs. (Section 2)

- To overcome these limitations, we design and implement a ternary mpGEMM library incorporating our innovative kernels, **TL** and **I2_S**, which we integrate into Bitnet.cpp. This library facilitates fast and lossless inference through element-wise design and precise alignment with training schemes. (Section 3)

- We evaluate Bitnet.cpp on multiple edge devices and demonstrate that it achieves a up to 6.25x speedup compared to state-of-the-art baselines, while realizing lossless inference for BitNet b1.58. (Section 4)

- We use TL2_0 to discuss the advantages of Bitnet.cpp compared to prior approaches with respect to computational and memory access efficiency.(Section 5)

- Finally, in the appendix, we extend TL beyond ternary LLMs, to element-wise lookup table (ELUT) for low-bit LLMs. We perform detailed analyses of ELUT (Appendix A), demonstrating its high efficiency and untapped potential. (Appendix B).

2 Ternary LLM & mpGEMM on Edge

In this section, we present a detailed examination of the characteristics of ternary LLMs and introduce a systematic taxonomy of current edge mpGEMM methods, as illustrated in Figure 3. We aim to delineate the limitations of existing mpGEMM approaches in handling ternary LLMs, informed by our comprehensive survey, with the objective of guiding future optimizations.

2.1 Ternary LLM: Features

Ternary Weights A distinctive characteristic of ternary LLMs is that the weights in the transformer layers are ternary, allowing only three possible values: {-1, 0, 1}. Consequently, the information content of these weights is approximately 1.58 bits per weight, as calculated by $\log(3)/\log(2)$. This substantial compression not only markedly reduces the model size, but also enables opportunities for further optimization with existing mpGEMM methods, such as those employed in llama.cpp and T-MAC.

Lossless Inference for BitNet b1.58 BitNet b1.58 performs ternary quantization on weights and int8 per-tensor quantization on activations during training. Based on this, if the training constraints are preserved during inference, lossless inference can be achieved for BitNet b1.58, as shown in Figure 2.

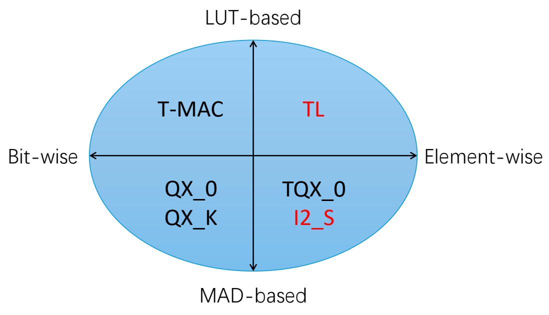


Figure 3: A taxonomy of mpGEMM solutions for ternary LLMs on edge devices. TL and I2_S are integrated in Bitnet.cpp, while QX and TQX are integrated in llama.cpp.

2.2 mpGEMM on Edge: Definitions

MAD-based and LUT-based We classify edge mpGEMM methods into two computational strategies: **multiply-then-add (MAD)-based** and **lookup table (LUT)-based**. The MAD-based strategy performs dot product calculations, while the LUT-based strategy employs lookup tables to store precomputed values, thereby enabling rapid accumulation via table lookups.

Bit-wise and Element-wise Edge mpGEMM methods are additionally classified based on the fundamental unit of computation into **Bit-wise** and **Element-wise** categories. Bit-wise methods process data at the bit level, focusing solely on bit operation without considering the attributes of weight elements, precluding non-integer bits per weight. In contrast, element-wise methods perform computations at the element level, taking into account the distinct properties of each weight element, which enables non-integer bits per weight.

2.3 mpGEMM on Edge: Taxonomy (Figure 3)

Bit-wise LUT-based (Up left) Recent research by T-MAC has shown that bit-wise LUT-based methods significantly outperform MAD-based approaches in edge inference, particularly emphasizing their efficiency for low-bit LLMs. However, when applied to ternary LLMs, these bit-wise LUT-based methods exhibit spatial inefficiencies, leading to a substantial performance decline in environments with limited bandwidth.

Bit-wise MAD-based (Down left) As a foundational framework for LLM edge inference, llama.cpp has pioneered several low-bit edge mpGEMM methods, predominantly bit-wise MAD-based, including the QX_0 and QX_K series. For instance, Q2_K utilizes the K-quants method to quantize weights to 2 bits, thereby ensuring the universality and correctness of the quantization. However, the application of Q2_K to ternary weights introduces complications: in addition to wasted space, maintaining accuracy with K-quants necessitates a multi-step dequantization process prior to performing the dot product, consequently increasing the overall latency.

Element-wise MAD-based (Down right) In fact, llama.cpp introduces two element-wise MAD-based methods for ternary LLMs: TQ1_0 and TQ2_0, with bits per weight of 1.69 and 2.06, respectively. These methods leverage the ternary nature of the weights to avoid the multi-step dequantization required by K-quants, thereby significantly boosting performance. Despite these advancements, the lack of support for tensor-wide quantization means llama.cpp relies on per-block quantization with a static block length of 256 for activations (e.g., Q8_K). To accommodate this limitation, TQX_0 also utilizes the block quantization scheme. However, this approach is inconsistent with the computational methods used during BitNet b1.58 training, thus hindering TQX_0 from achieving lossless inference.

3 Methodology

Kernel	type	bpw	Lossless
TL1_0	LUT-based	2	×
TL1_1	LUT-based	2	✓
TL2_0	LUT-based	1.67	×
TL2_1	LUT-based	1.67	✓
I2_S	MAD-based	2	✓

Table 1: Bitnet.cpp ternary mpGEMM library.

This section addresses the limitations of existing edge mpGEMM methods, as previously discussed, through the design and implementation of a novel ternary mpGEMM library, summarized in Table 1. We aim to showcase our pioneering techniques for efficient edge inference of ternary LLMs, focusing on two key dimensions: fast and lossless.

3.1 Fast Edge Inference

For MAD-based methods, llama.cpp has implemented TQ1_0 and TQ2_0, which facilitate rapid ternary LLM edge inference. However, the current bit-wise approach for LUT-based methods does not fully exploit the potential of ternary LLMs

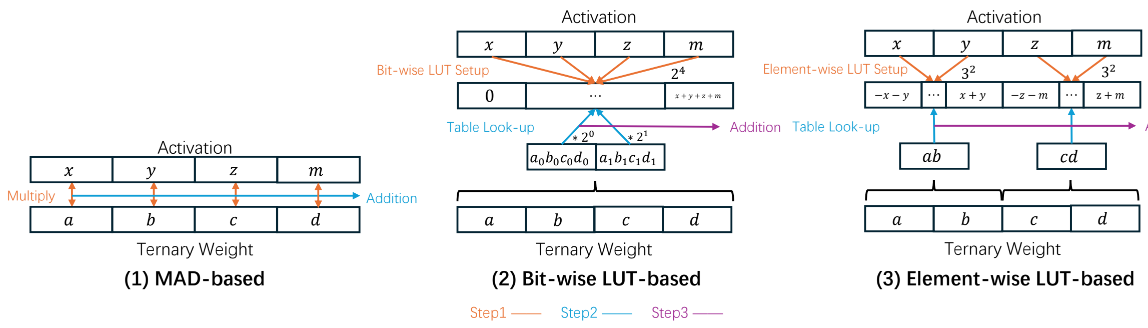


Figure 4: A simple example to explain the differences between various methods for completing mpGEMM when $K = 4$: (1) represents the MAD-based solution, where the result is obtained via the dot product; (2) represents the bit-wise LUT-based solution, where the weights are split into different bit indices, and the result is obtained by performing a lookup in the LUT, followed by bit-shifting and accumulation; (3) represents the element-wise LUT-based solution, where all possible values of the weights are enumerated to obtain the index, and the result is obtained by performing a lookup in the LUT, followed by accumulation. A_x refers to the x_{th} bit in weight A. In (2), $g = 4$ and $b = 2$; whereas in (3) $g = 2$ and $C = 3$.

for fast edge inference. Consequently, we have developed the **element-wise LUT-based (ELUT) mpGEMM**, which not only reduces bpw but also addresses the spatial inefficiencies inherent in bit-wise methods through **element-wise mirror consolidation**. To effectively implement ELUT in ternary LLMs, noted as **TL**, we mitigate issues such as misaligned memory access through **signed-unsigned weight splitting**, overcome hardware instruction support deficiencies with **1bit sign operation**, and resolve misaligned block computations via **block-fitting weight splitting**. This subsection will elaborate on our design and implementation strategies. For an in-depth analysis of the reasons behind ELUT's acceleration and its broader implications beyond ternary LLMs, please refer to Appendix A.

3.1.1 Design: TL

Element-wise LUT-based mpGEMM The bit-wise LUT-based mpGEMM, designed for generality, uses 2-bit storage for ternary weights, leading to space inefficiency, thus negatively affecting speed. To overcome these limitations, we introduce an element-wise LUT-based mpGEMM approach. In the following, we delineate the key distinctions among MAD-based, bit-wise LUT-based, and element-wise LUT-based mpGEMM methods.

$$R = \sum_{i \leftarrow 1}^{K} A_i W_i \tag{1}$$

$$R = \sum_{i \leftarrow 1}^{b} \sum_{j \leftarrow 1}^{K/g} \text{Look-up}(bLUT_j, W_{ij}) \tag{2}$$

$$R = \sum_{i \leftarrow 1}^{K/g} \text{Look-up}(eLUT_i, W_i) \tag{3}$$

$$W \in \mathbb{Z},\ |W| = C \tag{4}$$

Consider a simple GEMM computation involving two input matrices: A (1, K) and B (K, 1). As shown in Equation 1, MAD-based mpGEMM computes the result using the dot product. In LUT-based mpGEMM, the conventional approach employs a bit-wise representation of the LUT, as shown in Equation 2, where b denotes the bit-width of the weight (2 for ternary weights, as $3 < 2^2$), and g represents the group size. The bit-wise LUT ($bLUT$) has a size of b^g. By relaxing the bit-width restriction and adopting an element-wise representation of the LUT, as shown in Equation 3, a finer-grained expression is obtained. In this case, the element-wise LUT ($eLUT$) has a size of C^g, where C denotes the cardinality of the weight set (3 for ternary weights). Figure 4 illustrates a simple example highlighting these differences.

Element-wise Mirror Consolidation (Wei et al., 2024) introduced the concept of mirror consolidation, positing that during LUT enumeration, half of the values for b^g are inversely related to the other half, effectively halving the LUT size. Extending this concept to C^g results in what we term element-wise mirror consolidation. For the element-wise LUT-based solution, due to the 128-bit SIMD register instruction length (e.g., AVX2 vpshufb), C^g is constrained to a maximum of 16 ($16 \times int8 = 128$). Without element-wise mirror consolidation, the maximum value of g for ternary LLMs remains at 2, maintaining the same bpw as the bit-wise

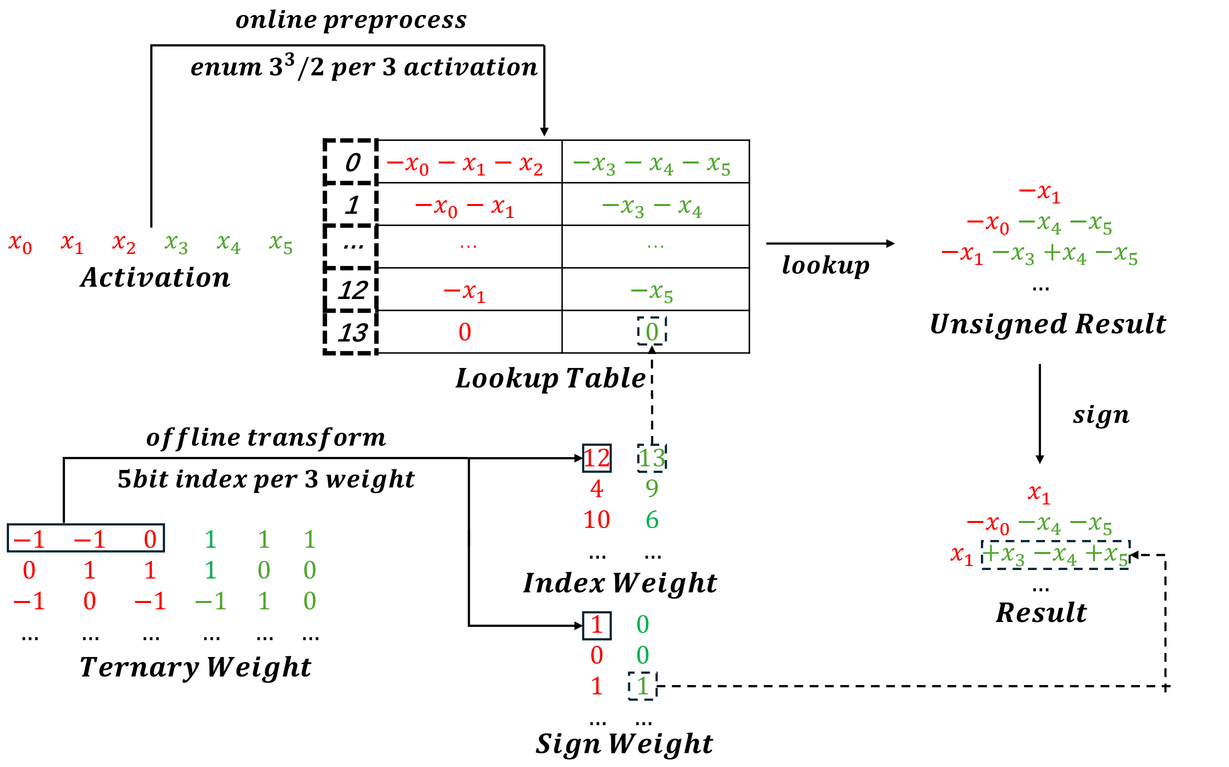


Figure 5: The TL2 design uses signed-unsigned weight splitting. First, a 4-bit index weight is used to look up the table and obtain the unsigned result. Then, the corresponding 1-bit sign weight is applied to perform the sign operation on the unsigned result, yielding the final output.

LUT-based method (4 bits for 2 weights, $3^2 < 2^4$). However, employing element-wise mirror consolidation increases the maximum g to 3, thus compressing bpw to 1.67 (5 bits for 3 weights, $\frac{3^3}{2} < 2^4$). Consequently, we have developed two practical designs for TL. We refer to the design with $g = 2$ as TL1 and the design with $g = 3$, which incorporates element-wise mirror consolidation, as TL2. Algorithm 3 details the design of TL1, while Algorithm 4 outlines that of TL2.

3.1.2 Implementation: TL

Signed-Unsigned Weight Splitting To implement element-wise mirror consolidation, we introduce signed-unsigned weight splitting, where we use a separate 1-bit sign weight to store the sign of the enumeration, and a 4-bit index weight to store the corresponding LUT index for unsigned enumeration. It is evident that using continuous 5-bit storage for 3 weights would cause severe memory access misalignment. Since LUT-based mpGEMM is inherently memory-intensive, the additional memory accesses caused by misalignment would significantly degrade performance. In contrast, signed-unsigned weight splitting allows three weights to be represented using 5 bits, adhering to the element-wise approach, while simultaneously avoiding misalignment issues in computation and memory access. Figure 5 demonstrates the detailed computation flow of TL2, using signed-unsigned weight splitting.

1bit Sign Operation Determining the sign of the value indexed from the LUT using only 1 bit is challenging, as values are represented in two's complement, and the design must ensure compatibility with SIMD instructions.

$$\begin{gathered} x = \text{sign} \oplus (\text{sign} + x) \\ x \in \text{int8},\ \text{sign} \in \{0, 1\} \end{gathered} \tag{5}$$

After evaluating multiple methods, we selected the approach presented in Equation 5 to address the issue. This sequence of operations, which includes the XOR and ADD operations, enables the sign to be determined by a single bit and is fully compatible with both the AVX2 and NEON instructions. When the bit of sign is 0, the result remains unchanged; otherwise, the result is converted to its negative value.

Block-fitting Weight Splitting The TL series employs an LUT-centric data layout for mpGEMM to address inefficiencies in memory storage and access, as introduced by T-MAC. When adopting

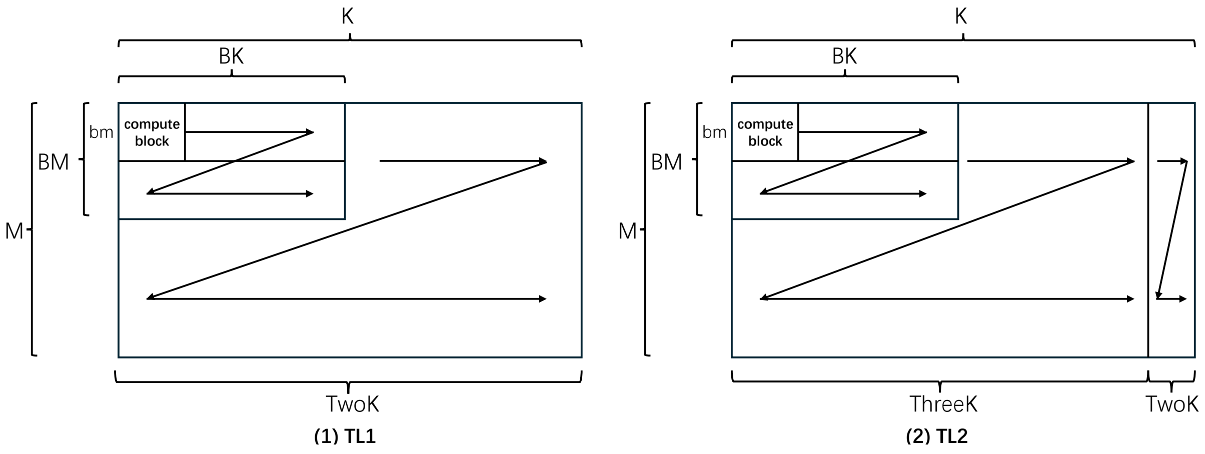


Figure 6: The computation sequences for TL1 and TL2. The left side represents TL1, and the right side represents TL2. Arrows indicate the order of computation, with the smallest computational unit being the compute block. M and K refer to the dimensions of the weights. $bm \times by$ refers to the number of weights involved in the minimal compute block. bm can be selected from 16 or 32. In TL1, by is $\frac{256}{bm}$, and in TL2, by is $\frac{192}{bm}$.

this layout, it is crucial to ensure that the minimal compute blocks align precisely with the weight matrix. As illustrated on the left side of Figure 6, for TL1, the block length BK must be divisible by the matrix dimension K. This condition is easily met in TL1, as $g = 2$, meaning K only needs to be a multiple of 2. However, the situation differs for TL2. Most LLM weight shapes do not have K as a multiple of 3 when using TL2, where $g = 3$. To address this, we introduce block-fitting weight splitting, which statically divides the weight into two parts to fit the blocks. After splitting, as shown on the right side of Figure 6, one portion of the weight, with dimensions $ThreeK = \lfloor \frac{K}{BK3} \rfloor \times BK3$, is computed using TL2, while the remaining portion, $TwoK = K - ThreeK$, is computed using TL1. By applying block-fitting weight splitting, we resolve the block mismatch issue without requiring additional padding, thereby preventing potential latency increases.

3.2 Lossless Edge Inference

To achieve lossless inference for BitNet b1.58, this subsection first identifies the gaps between existing methods and lossless inference. It then presents innovative approaches for achieving lossless inference using both MAD-based and LUT-based methods.

3.2.1 Design & Implementation: TL

Since table lookups require SIMD instructions operating on 8-bit data, a potential conflict arises when enumerating sums that might overflow if stored in 8-bit integers. T-MAC addresses this issue by quantizing the accumulated sum to int8; however, this approach introduces additional losses, preventing lossless inference. To resolve this, we introduce the pack-and-unpack technique. First, we maintain the sums as int16 without additional quantization and split the int16 enumerated sums into two parts using the pack instruction. Then, during the indexing process, we apply the table lookup twice. Afterward, we use the unpack instruction to concatenate the two parts, ultimately obtaining the desired int16 result. Kernels that utilize typical additional quantization are TL1_0 and TL2_0, whereas those that use the pack-and-unpack technique are TL1_1 and TL2_1.

3.2.2 Design & Implementation: I2_S

Due to inconsistency with training schemes, existing element-wise MAD-based methods do not enable lossless inference for BitNet b1.58. In Bitnet.cpp, I2_S is designed based on the element-wise approach, adhering strictly to the ternary weight and per-tensor int8 activation quantization settings of BitNet b1.58 training, thereby ensuring lossless inference. Furthermore, I2_S performs comparably with TQ2_0 and supports mpGEMM dimensions K that are multiples of 128, while TQ2_0 only supports multiples of 256. As a result, we have optimized the MAD-based solutions and integrated the implementation into Bitnet.cpp.

4 Experiments

We evaluated the performance of Bitnet.cpp for end-to-end edge inference for ternary LLM. Compared to state-of-the-art methods, Bitnet.cpp signif-

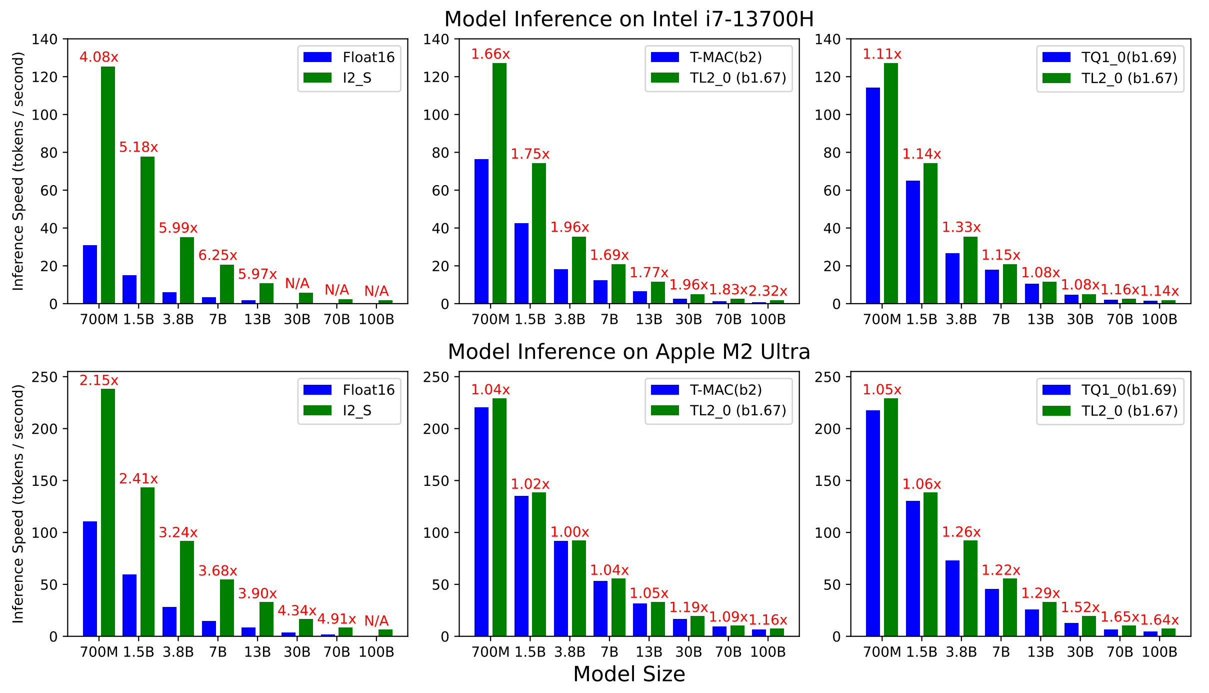


Figure 7: End-to-end performance for inference on multiple model sizes, specific details of the model size are referenced in (Wang et al., 2024b). Here, (bx) denotes the bpw value, where x represents the respective bpw. Detailed performance information can be found on Table 7.

icantly improves ternary LLM edge inference performance across different CPU architectures and model sizes under the sub-2-bits-per-weight condition. For quality evaluation, compared to Float16, TL1_0 and TL2_0 exhibit negligible loss, whereas I2_S, TL1_1, and TL2_1 achieve lossless in BitNet b1.58.

4.1 Speed Evaluation

4.1.1 Devices

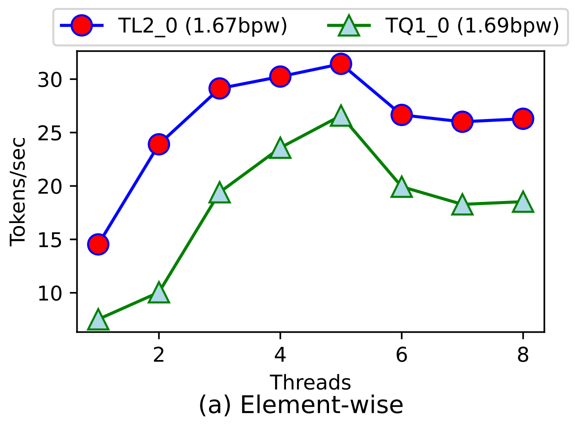
We conducted a performance evaluation of Bitnet.cpp on two devices: the Apple M2 Ultra and the Intel i7-13700H. These devices represent the ARM and x86 architectures, respectively, covering most edge devices and ensuring broad applicability and reliable performance results for Bitnet.cpp.

4.1.2 Baselines

We conducted experiments from two perspectives: lossless inference and fast inference. For the lossless inference aspect, we chose llama.cpp Float16 as the baseline and compared it with I2_S from Bitnet.cpp. This comparison evaluates the lossless inference performance of Bitnet.cpp, demonstrating its improvements in both accuracy and speed. For the fast inference aspect, we conducted experiments based on the two features of TL2_0: element-wise and LUT-based. llama.cpp includes two element-wise MAD-based solutions, TQ1_0 and TQ2_0. To neutralize the effect of bpw, TQ1_0, which has a bpw nearly identical to TL2_0, was selected for comparison. This comparison aims to evaluate the performance differences between MAD-based and LUT-based solutions. For T-MAC, a bit-wise LUT-based solution, the 2-bit kernel was selected for comparison with TL2_0 to assess performance differences between element-wise and bit-wise methods.

4.1.3 End-to-end Inference Speed

We evaluated the token generation speed of Bitnet.cpp and observed a significant speed advantage across different CPU architectures and model sizes compared to baselines. As illustrated in Figure 7, I2_S achieves up to a 6.25x speedup compared to Float16, demonstrating that Bitnet.cpp provides a comprehensive advantage in both accuracy and speed. Furthermore, TL2_0 outperforms T-MAC by up to 2.32x on the Intel i7-13700H and by up to 1.19x on the Apple M2 Ultra, indicating a notable improvement in LUT-based mpGEMM performance. Moreover, TL2_0 surpasses TQ1_0, with up to 1.33x speedup on the Intel i7-13700H and 1.65x on the Apple M2 Ultra, further improv-



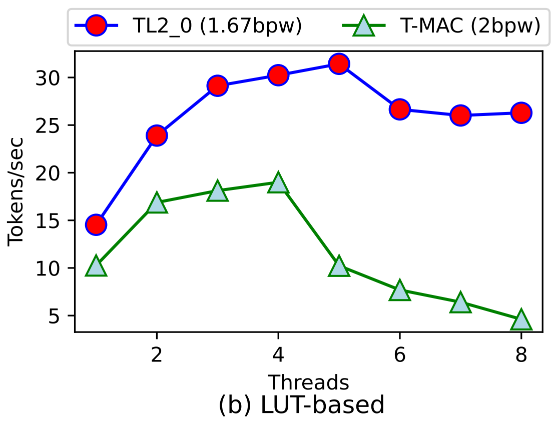


Figure 8: Multi-threaded end-to-end inference performance of the 3.8B model on Intel i7 13700H.

ing performance in element-wise mpGEMM with bpw below 2. As detailed in Table 7, TL2_0 reaches 7.45 tokens/s on the Apple M2 Ultra and 1.69 tokens/s on the Intel i7-13700H, outperforming previous ternary kernels in 100B ternary LLM inference on edge devices. These findings highlight the significant inference benefits of Bitnet.cpp.

4.2 Quality Evaluation

We used the bitnet_b1_58-large[1] model and the perplexity[2] tool from llama.cpp for quality evaluation. For baselines, Float16 and Q4_0 from llama.cpp were selected for comparison with Bitnet.cpp. For tasks, we used WikiText2(Merity et al., 2016) to measure perplexity (the lower, the better), HellaSwag(Zellers et al., 2019) and WinoGrande(Sakaguchi et al., 2021) to measure accuracy (the higher, the better). As shown in Table 2, both TL1_0 and TL2_0 achieve nearly identical perplexity compared to Float16 on WikiText2 and maintain accuracy comparable to Float16 on WinoGrande and HellaSwag. I2_S, TL1_1, and TL2_1 exhibit lossless performance relative to Float16 across all tasks. These results indicate that the loss introduced by Bitnet.cpp is negligible.

Method	**WikiText2** Perplexity↓	**Winograd** Accuracy↑	**HellaSwag** Accuracy↑
Float16	11.29	55.32	43.0
Q4_0	11.57	55.09	42.25
TL1_0	11.30	55.32	43.0
TL2_0	11.30	55.32	43.0
TL1_1	11.29	55.32	43.0
TL2_1	11.29	55.32	43.0
I2_S	11.29	55.32	43.0

Table 2: End-to-end inference quality.

5 Discussion

5.1 Memory-Computation Trade-off

The execution speed of a kernel is governed by both instruction computation speed and data access speed. The former is influenced by computational complexity, instruction types, and pipeline depth, while the latter depends on memory access complexity, data locality, and the type of memory accessed. Ultimately, kernel execution speed is constrained by the slower of the two. Naturally, we refer to computation-related costs as **computation** overhead and data-access-related costs as **memory** overhead. Thus, optimizing kernel performance is essentially a matter of exploring the compute-memory trade-off.

5.2 Towards Compute: Compared to TQ1_0

For the computation overhead comparison, we selected TQ1_0, which has a bpw value nearly identical to that of TL2_0 to ensure a fair comparison. As shown in Figure 8(a), the performance curves of TL2_0 and TQ1_0 in a multi-threaded environment exhibit similar trends, with TL2_0 consistently outperforming TQ1_0 across all threads. These results demonstrate that LUT-based solutions consistently outperform MAD-based alternatives in computation-related overhead, reinforcing their advantage in computational efficiency and resulting in notable performance improvements.

[1] `https://huggingface.co/1bitLLM/bitnet_b1_58-large`

[2] `https://github.com/ggerganov/llama.cpp/tree/master/examples/perplexity`

5.3 Towards Memory: Compared to T-MAC

For the memory overhead comparison, TL2_0 has an advantage over T-MAC in terms of bpw, thereby modestly increasing the performance ceiling of memory-intensive LUT-based solutions. As a result, significant performance improvements are achieved, particularly in low-bandwidth environments. As shown in Figure 8(b), TL2_0 outperforms T-MAC in a multi-threaded setting. Notably, TL2_0 continues to scale as the number of threads increases to five, whereas T-MAC's performance starts to degrade. This indicates that TL2_0 reaches the memory-bound state later than T-MAC, thereby delaying bandwidth saturation and extending the performance ceiling.

5.4 Generalization: ELUT

To enhance generality, we extend the design concept of TL beyond ternary LLMs to ELUT (element-wise LUT-based mpGEMM). Detailed information ELUT can be found in the Appendix A and Appendix B. ELUT is applicable to LLMs whose weights assume a finite set of values and provides a finer-grained alternative to bit-wise LUT-based mpGEMM. For LLMs with an integer bpw, bit-wise LUT-based methods and ELUT are theoretically equivalent. The primary contribution of ELUT lies in its ability to support mpGEMM design for LLMs with non-integer bpw, such as ternary LLMs.

6 Related Work

LUT-based mpGEMM Previous research has explored the application of LUT-based mpGEMM in deep learning. (Ganji et al., 2023) employs LUT-based mpGEMM to accelerate computations in convolutional neural networks, while (Davis Blalock, 2021; Tang et al., 2023) utilize this approach to process vector-quantized activations. For LLM inference, (Park et al., 2024; Maleki, 2023; Frantar, 2023) apply LUT-based GEMM on GPUs. However, in practice, these methods are often slower than MAD-based approaches, such as (cut; bit), due to the inefficiency of rapid table access on GPU.

LLM Inference FlashAttention (Dao et al., 2022; Dao, 2023) introduces an innovative approach to GPU attention kernel design. VLLM (Kwon et al., 2023) and TensorRT-LLM (trt) have optimized end-to-end inference performance using systematic techniques. Powerinfer (Song et al., 2024; Xue et al., 2024) proposes novel strategies that intelligently balance workloads across heterogeneous devices, improving overall inference efficiency.

LLM Quantization Post-training quantization (PTQ) refers to converting a full-precision LLM to a low-precision without retraining, with related works including (Xiao et al., 2023; Lin et al., 2024; Chee et al., 2023; Frantar et al., 2023; Dettmers et al., 2023, 2022; Shao et al., 2024). However, PTQ inevitably results in quantization loss. In contrast, Quantization-Aware Training (QAT) effectively avoids this issue. QAT involves retraining a pretrained model to obtain a quantized model, thus mitigating quantization loss. Relevant works include (Liu et al., 2023; Chen et al., 2024; Du et al., 2024). BitNet B1.58 adopts QAT, creating conditions for lossless inference in the system.

7 Conclusion

In this paper, by optimizing mpGEMM, we address the inefficiencies caused by the conflicts of non-integer bpw in ternary LLMs with memory access alignment rules, and enable lossless inference for BitNet b1.58. Our key idea is to utilize a finer-grained element-wise scheme instead of bit-wise, and consistent with BitNet b1.58 training schemes. Based on our key ideas, we develop Bitnet.cpp, featuring TL, the first element-wise LUT-based mpGEMM kernel for ternary LLMs, and I2_S, the first lossless MAD-based kernel for BitNet b1.58. The practical outcomes of our research are noteworthy. We have demonstrated that Bitnet.cpp achieves up to 6.25x speedup compared to baselines and provided lossless inference for BitNet b1.58. To enhance the generality of our research, we extended the TL to ELUT for low-bit LLMs, highlighting its efficiency and potential. This paper presents extensive work on optimizing edge inference for ternary LLMs from both algorithmic and engineering perspectives. It offers the research community new insights into handling ternary and non-integer bpw weights, shows the practical advantages of ternary LLMs and presents the industry with innovative solutions for deploying fast, lossless LLMs on edge devices.

Limitations

Bitnet.cpp has the following limitations:

- Bitnet.cpp currently only provides a practical solution for ternary LLM inference on edge devices. In the future, we plan to extend the Bitnet.cpp to offer efficient inference solutions for ternary LLMs across multiple devices.

- Bitnet.cpp is specifically designed for ternary LLMs, with a relatively narrow range of applicable model architectures. In response to this, we have expanded the element-wise LUT-based (ELUT) method to cover low-bit ranges in the appendix. However, it still lacks support from actual LLMs other than ternary ones.

- Bitnet.cpp does not discuss in detail the acceleration specifics of LLMs during the prefilling stage, as there has been a shift in the resource bottleneck from being memory-bound during the decoding stage to computation-bound during the prefilling stage. Therefore, the original optimization methods are no longer applicable, and we will continue to explore optimization methods for the prefilling stage.

References

Bitblas. `https://github.com/microsoft/bitblas`.

Cutlass. `https://github.com/NVIDIA/cutlass`.

llama.cpp. `https://github.com/ggerganov/llama.cpp`.

Pcm. `https://github.com/intel/pcm`.

Tensorrt-llm. `https://github.com/NVIDIA/TensorRT-LLM`.

Marah Abdin, Jyoti Aneja, Hany Awadalla, Ahmed Awadallah, and Ammar Ahmad Awan et al. 2024. Phi-3 technical report: A highly capable language model locally on your phone. *Preprint*, arXiv:2404.14219.

Jerry Chee, Yaohui Cai, Volodymyr Kuleshov, and Christopher M De Sa. 2023. Quip: 2-bit quantization of large language models with guarantees. In *Advances in Neural Information Processing Systems*, pages 4396–4429. Curran Associates, Inc.

Mengzhao Chen, Wenqi Shao, Peng Xu, Jiahao Wang, Peng Gao, Kaipeng Zhang, and Ping Luo. 2024. Efficientqat: Efficient quantization-aware training for large language models. *Preprint*, arXiv:2407.11062.

Tri Dao. 2023. Flashattention-2: Faster attention with better parallelism and work partitioning. *Preprint*, arXiv:2307.08691.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. In *Advances in Neural Information Processing Systems*, pages 16344–16359. Curran Associates, Inc.

John Guttag Davis Blalock. 2021. Multiplying matrices without multiplying. In *Proceedings of the 38th International Conference on Machine Learning*, pages 992–1004. PMLR.

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

Tim Dettmers, Ruslan Svirschevski, Vage Egiazarian, Denis Kuznedelev, Elias Frantar, Saleh Ashkboos, Alexander Borzunov, Torsten Hoefler, and Dan Alistarh. 2023. Spqr: A sparse-quantized representation for near-lossless llm weight compression. *Preprint*, arXiv:2306.03078.

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

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2023. Gptq: Accurate post-training quantization for generative pre-trained transformers. *Preprint*, arXiv:2210.17323.

Darshan C. Ganji, Saad Ashfaq, Ehsan Saboori, Sudhakar Sah, Saptarshi Mitra, MohammadHossein AskariHemmat, Alexander Hoffman, Ahmed Hassanien, and Mathieu Léonardon. 2023. Deepgemm: Accelerated ultra low-precision inference on cpu architectures using lookup tables. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, pages 4656–4664.

Ayush Kaushal, Tejas Pandey, Tejas Vaidhya, Aaryan Bhagat, and Irina Rish. 2024. Spectra: A comprehensive study of ternary, quantized, and fp16 language models. *Preprint*, arXiv:2407.12327.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. 2024.

Awq: Activation-aware weight quantization for on-device llm compression and acceleration. In *Proceedings of Machine Learning and Systems*, pages 87–100.

Zechun Liu, Barlas Oguz, Changsheng Zhao, Ernie Chang, Pierre Stock, Yashar Mehdad, Yangyang Shi, Raghuraman Krishnamoorthi, and Vikas Chandra. 2023. Llm-qat: Data-free quantization aware training for large language models. *Preprint*, arXiv:2305.17888.

Shuming Ma, Hongyu Wang, Lingxiao Ma, Lei Wang, Wenhui Wang, Shaohan Huang, Li Dong, Ruiping Wang, Jilong Xue, and Furu Wei. 2024. The era of 1-bit llms: All large language models are in 1.58 bits. *arXiv preprint arXiv:2402.17764*.

Saeed Maleki. 2023. Look-up mai gemm: Increasing ai gemms performance by nearly 2.5x via msgemm. *Preprint*, arXiv:2310.06178.

Elias Frantar, Dan Alistarh. 2023. QMoE: Practical Sub-1-Bit Compression of Trillion-Parameter Models. *Preprint*, arXiv:2310.16795.

Mohamed Mekkouri, Marc Sun, Leandro von Werra, and Thomas Wolf. 2024. 1.58-bit llm: A new era of extreme quantization.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer sentinel mixture models. *Preprint*, arXiv:1609.07843.

Zhiwen Mo, Lei Wang, Jianyu Wei, Zhichen Zeng, Shijie Cao, Lingxiao Ma, Naifeng Jing, Ting Cao, Jilong Xue, Fan Yang, and Mao Yang. 2024. Lut tensor core: Lookup table enables efficient low-bit llm inference acceleration. *Preprint*, arXiv:2408.06003.

Gunho Park, Baeseong Park, Minsub Kim, Sungjae Lee, Jeonghoon Kim, Beomseok Kwon, Se Jung Kwon, Byeongwook Kim, Youngjoo Lee, and Dongsoo Lee. 2024. Lut-gemm: Quantized matrix multiplication based on luts for efficient inference in large-scale generative language models. *Preprint*, arXiv:2206.09557.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. 2021. Winogrande: an adversarial winograd schema challenge at scale. *Commun. ACM*, 64(9):99–106.

Wenqi Shao, Mengzhao Chen, Zhaoyang Zhang, Peng Xu, Lirui Zhao, Zhiqian Li, Kaipeng Zhang, Peng Gao, Yu Qiao, and Ping Luo. 2024. Omniquant: Omnidirectionally calibrated quantization for large language models. *Preprint*, arXiv:2308.13137.

Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. 2024. Powerinfer: Fast large language model serving with a consumer-grade gpu. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 590–606, New York, NY, USA. Association for Computing Machinery.

Xiaohu Tang, Yang Wang, Ting Cao, Li Lyna Zhang, Qi Chen, Deng Cai, Yunxin Liu, and Mao Yang. 2023. Lut-nn: Empower efficient neural network inference with centroid learning and table lookup. In *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*, ACM MobiCom '23, New York, NY, USA. Association for Computing Machinery.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M. Dai, and et al. Anja Hauth. 2024. Gemini: A family of highly capable multimodal models. *Preprint*, arXiv:2312.11805.

Hongyu Wang, Shuming Ma, Li Dong, Shaohan Huang, Huaijie Wang, Lingxiao Ma, Fan Yang, Ruiping Wang, Yi Wu, and Furu Wei. 2023. Bitnet: Scaling 1-bit transformers for large language models. *arXiv preprint arXiv:2310.11453*.

Hongyu Wang, Shuming Ma, and Furu Wei. 2024a. Bitnet a4.8: 4-bit activations for 1-bit llms. *Preprint*, arXiv:2411.04965.

Jinheng Wang, Hansong Zhou, Ting Song, Shaoguang Mao, Shuming Ma, Hongyu Wang, Yan Xia, and Furu Wei. 2024b. 1-bit ai infra: Part 1.1, fast and lossless bitnet b1.58 inference on cpus. *Preprint*, arXiv:2410.16144.

Jianyu Wei, Shijie Cao, Ting Cao, Lingxiao Ma, Lei Wang, Yanyong Zhang, and Mao Yang. 2024. T-mac: Cpu renaissance via table lookup for low-bit llm deployment on edge. *arXiv preprint arXiv:2407.00088*.

Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. 2023. SmoothQuant: Accurate and efficient post-training quantization for large language models. In *Proceedings of the 40th International Conference on Machine Learning*, pages 38087–38099.

Yanyue Xie, Zhengang Li, Dana Diaconu, Suranga Handagala, Miriam Leeser, and Xue Lin. 2024. Lut-mul: Exceed conventional fpga roofline limit by lut-based efficient multiplication for neural network inference.

Zhenliang Xue, Yixin Song, Zeyu Mi, Le Chen, Yubin Xia, and Haibo Chen. 2024. Powerinfer-2: Fast large language model inference on a smartphone. *Preprint*, arXiv:2406.06282.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. 2019. Hellaswag: Can a machine really finish your sentence? *Preprint*, arXiv:1905.07830.

In the appendix, we extend the concept of element-wise LUT-based solutions beyond ternary LLMs, analyzing its capabilities and potential from a more general perspective.

A Insight

In this section, we will analyze the computational complexity and memory access complexity of the element-wise LUT-based (ELUT) mpGEMM algorithm. Based on this analysis, we will compare our results with those of MAD-based solutions and bit-wise LUT-based solutions, drawing the conclusion that the ELUT algorithm exhibits comprehensive advantages in both computation and memory access compared to previous algorithms.

A.1 Complexity

In general, mpGEMM requires two steps to complete: the preprocessing stage and the accumulation stage. As shown in Algorithm 1, for the MAD-based solution, the preprocessing stage involves quantizing the floating-point activations to integers, with a computational complexity of $O(NK)$ and a memory access complexity of $O(NK)$. In the accumulation stage, the MAD-based solution performs element-wise multiplication and accumulation for the K corresponding elements across M rows and N columns, resulting in a computational complexity of $O(MNK)$ and a memory access complexity of $O(MNK)$.

As shown in Algorithm 2, for ELUT, the preprocessing stage involves first performing quantization to quantize the floating-point activations into NK/g groups, and then enumerating the C^g possible values within each group to construct the Lookup Table. The computational complexity of this process is $O(NKC^g/g)$, and the memory access complexity is also $O(NKC^g/g)$. In the accumulation stage, ELUT performs lookup and accumulation operations group by group. The computational complexity of this process is $O(MNK/g)$, while the memory access complexity is $O(MNKC^g/g)$ because the entire Lookup Table must be loaded for each group.

Through theoretical analysis, we can identify several interesting insights. First, ELUT has an advantage over the MAD-based solution in terms of computational complexity for LLM inference. The overall computational complexity of the MAD-based solution is $O(MNK)$, while ELUT is $max(O(NKC^g/g), O(MNK/g))$. This implies

Algorithm 1: MAD-based mpGEMM

```
Input: Activation $A$ of shape $N, K$
Input: Weights $W$ of shape $M, K$,
       $W \in \mathbb{Z},\ |W| = C$
Output: Result matrix $R$ of shape $M, N$
/* C-complexity $\rightarrow$ Computational
   Complexity                              */
/* M-complexity $\rightarrow$ Memory Access
   Complexity                              */
/* Phase1 : Preprocessing                  */
/* C-complexity : $O(NK)$ /
   M-complexity : $O(NK)$                  */
1 $A_q$ = Quantization($A$)
/* Phase2 : Accumulation                   */
/* C-complexity : $O(MNK)$ /
   M-complexity : $O(MNK)$                 */
2 for $m, n \leftarrow 1$ to $M, N$ do
3     $R[n, m] = \sum_{k=1}^{K} (A_q[n, k] * W[m, k])$
4 end
/* Overall C-complexity : $O(MNK)$
   */
/* Overall M-complexity : $O(MNK)$
   */
```

Algorithm 2: ELUT mpGEMM

```
Input: Activation $A$ of shape $N, K$
Input: Weights $W$ of shape $M, K$,
       $W \in \mathbb{Z},\ |W| = C$
Input: Group size $g$
Output: Result matrix $R$ of shape $M, N$
/* C-complexity $\rightarrow$ Computational
   Complexity                              */
/* M-complexity $\rightarrow$ Memory Access
   Complexity                              */
/* Phase1 : Preprocessing                  */
/* C-complexity : $O(NKC^g/g)$ /
   M-complexity : $O(NKC^g/g)$             */
1 $A_q$ = Tbl-quantization($A$)
2 $LUT_A$ = Table-setup($A_q$)
/* Phase2 : Accumulation                   */
/* C-complexity : $O(MNK/g)$ /
   M-complexity : $O(MNKC^g/g)$            */
3 for $m, n \leftarrow 1$ to $M, N$ do
4     $R[n, m] =$
        $\sum_{k=1}^{K/g}$ Lookup($LUT_A[n, k], W[m, k]$)
5 end
/* Overall C-complexity :
   $max(O(NKC^g/g), O(MNK/g))$             */
/* Overall M-complexity :
   $O(MNKC^g/g)$                           */
```

that as long as $C^g < M$ and $g > 1$, ELUT requires fewer computations for mpGEMM. In LLMs, the value of M, i.e., the hidden size, is generally large. In contrast, the C value for ternary LLMs is only 3 and g is only 2 or 3. Therefore, ELUT is computationally more efficient than the MAD-based solution.

However, ELUT has a disadvantage in terms of memory access complexity compared to the MAD-based solution. The memory access complexity of the MAD-based solution is $O(MNK)$, while the LUT-based solution has a memory access complexity of $O(MNKC^g/g)$. In practical implementations, we employ optimization techniques such as element-wise mirror consolidation and LUT-centric data layout to reduce memory access complexity, thereby significantly mitigating the overhead caused by memory access.

A.2 Compared to MAD-based: More Practical

In fact, when deploying LLMs on current edge devices, we often face the limitation of using only a very small number of threads. Under such circumstances, the constraints on computational resources are maximized, making computational complexity a critical factor. In contrast, due to the limited number of threads, memory access is unlikely to reach bandwidth limits. In this context, ELUT, with its computational complexity being only $\frac{1}{g}$ of that of the MAD-based solution in most cases, is expected to outperform the MAD-based solution in real-world inference scenarios for LLMs. Therefore, ELUT is more suitable for deployment in practical scenarios than the MAD-based solution.

A.3 Compared to Bit-Wise : More Fine-grained

C	g	bpw_b	bpw_e
3	3	2	**1.67**
4	2	2	2
5	2	3	**2.5**
...	...	...	...

Table 3: A comparison table of bpw from bit-wise and element-wise for different weight cardinality. C represents the weight cardinality, g indicates to group size, bpw_b denotes bit-wise bpw, bpw_e refers to element-wise bpw.

Although we have demonstrated that ELUT outperforms MAD-based solutions in terms of performance with low thread counts, the bit-wise LUT-based solution also exhibits this advantage. The advantage of the ELUT method over the bit-wise method lies in its finer granularity of LUTs, shifting from bit-based to element-based, ensuring a more information-preserving compression of weights.

Returning to the computational complexity, in most cases, the computational complexity of the LUT method is $O(MNK/g)$. For ternary LLMs, when $g = 3$, the complexity is reduced by a factor of $\frac{1}{6}$ compared to $g = 2$. In terms of memory access complexity, since mirror consolidation is used when $g = 3$, we can compute the memory access complexity for $g = 2$ and $g = 3$ as follows.

$$O(\frac{MNK3^2}{2}) = O(\frac{MNK3^3/2}{3})$$

Based on this, since the bpw when $g = 3$ is approximately 1/6 lower than when $g = 2$ and memory access complexity is similar, we observe that when using the ELUT method on ternary LLMs inference, both computation and memory access are reduced compared to the bit-wise method. Similarly, as Table 3 shown, the same conclusion can be extended to the case where $C \neq 2^n$. This provides theoretical guidance for TL implementation.

B Potential

After evaluating the performance of ELUT, we have observed that it has a comprehensive advantage over other methods. However, we believe that ELUT has not yet reached its theoretical performance limit. In the following, we will analyze the hardware limitations affecting ELUT and estimate its theoretical performance in the absence of such constraints. This analysis aims to explore the potential of ELUT and provide insights for future hardware designs targeting low-bit LLMs inference.

B.1 Bandwidth

Bandwidth is the data transfer rate between memory and the processor, and it also determines the execution rate of kernels. Considering that ELUT has a higher memory access complexity than the MAD-based solution, bandwidth has a significant influence on overall end-to-end inference speed. As shown in Figure 7, it is evident that TL2_0 demonstrates a more pronounced acceleration effect on T-MAC for Intel i7-13700H compared to Apple M2 Ultra. The main reason for this phenomenon lies in the significant difference in maximum bandwidth between the two edge devices. In fact, the Apple

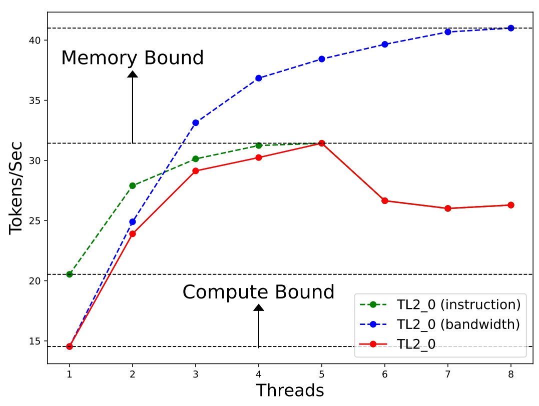


Figure 9: ELUT performance potential curve.

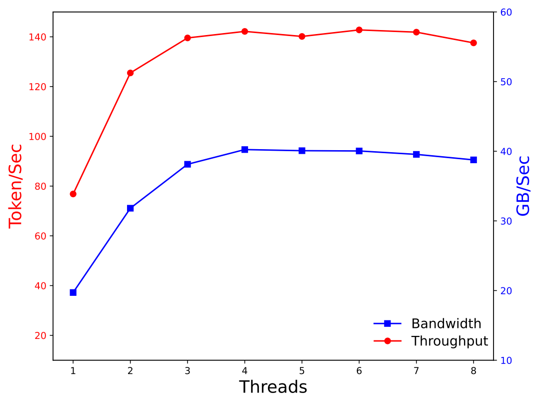


Figure 10: Throughput and Bandwidth curve, tested with bitnet-b1.58-large on intel core i5-13400F.

M2 Ultra has a maximum bandwidth exceeding 800 GB/s, while the maximum bandwidth of the Intel i7-13700H is less than 100 GB/s. As shown in Figure 10, we used PCM (PCM) tool to measure the token throughput and bandwidth at different thread counts and compared them side by side. It is clear that the shape of token throughput and bandwidth curves are nearly identical. When the thread count reaches 4, the token throughput also reaches its maximum value due to the saturation of the bandwidth, causing the end-to-end inference speed to reach its peak. Therefore, we can conclude that the maximum bandwidth limits the potential of ELUT. Building on this, as shown in Figure 9, we estimated the end-to-end inference speed when the bandwidth is increased. We anticipate that, with the increase in maximum bandwidth, ELUT will reach the memory-bound state later, resulting in a higher end-to-end inference speed, with the upper bound still determined by the theoretical maximum bandwidth. This estimation validates our theoretical analysis of ELUT. Moreover, we are pleased to note that there is currently a trend towards increasing the bandwidth of edge devices, which will further unlock the potential of ELUT.

B.2 Instructions Throughput

SIMD instructions are commonly used to implement kernels on CPUs, as SIMD allows a single instruction to process multiple data elements simultaneously, achieving computation parallelism and acceleration. For SIMD instructions, two metrics determine the performance of the instruction: instruction throughput, which determines the number of instructions that can be completed in a single clock, and instruction latency, which determines the number of clocks required to complete a single instruction. On modern CPUs, since MAD operations are widely used, common architectures such as x86 and ARM have made specific optimizations to ensure high instruction throughput for these operations (as shown in Table 4). For example, in the x86 architecture with AVX2 instructions, a single MAD instruction can complete an int8 multiply-accumulate operation and convert the result to int16. However, for ELUT, we need to use three types of instructions—TBL (table lookup), ADD (accumulation), and CVT (type conversion)—to accomplish the same task. Although the AVX documentation [3] states that the latency of the MAD instruction is 5 cycles, which is greater than the latency of the TBL instruction, both instructions have the same throughput. This implies that, under reasonable pipeline scheduling, the theoretical completion time for MAD and TBL instructions is the same. We validated this on an Intel i5-13400F, where the completion time for a single MAD instruction was **3.77 ns**, and for a single TBL instruction, it was **3.70 ns**, which is nearly identical. However, since the table lookup must be followed by addition and conversion (TBL+ADD+CVT), this sequence inevitably leads to a reduction in throughput. We observed that completing the same task with TBL+ADD+CVT took **6.20 ns**, approximately 68% longer than the raw latency of a single MAD instruction. This highlights that, in terms of throughput, the table lookup followed by the accumulation method suffers significant performance loss due to insufficient hardware support.

In previous work, (Mo et al., 2024; Xie et al.,

[3] https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

Instruction Set	LUT-based	MAD-based
AVX2	_mm256_shuffle_epi8	_mm256_maddubs_epi16
NEON	vqtbl1q_u8	vmlal_s8 / vmull_s16 + vaddq_s32

Table 4: Core instructions in AVX2 and Neon for LUT-based and MAD-based mpGEMM.

2024) was implemented in hardware on GPUs and FPGAs, respectively, as solutions similar to ELUT, and they achieved performance improvements over MAD-based solutions. This suggests that providing better hardware support for ELUT on edge devices is highly promising. As shown in Figure 9, we estimated the performance of ELUT with hardware support, and the results indicate a significant performance boost when bandwidth is not a bottleneck. We sincerely hope that the exploration of ELUT's potential can inspire future hardware designs to fully unlock ELUT's capabilities.

B.3 Register Length

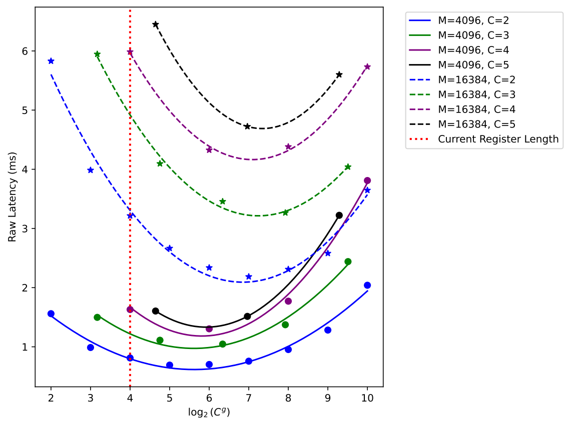


Figure 11: Register length and raw latency relationship graph.

The length of registers also imposes a limitation on the performance of ELUT. Taking AVX2 as an example, the lookup width of the TBL SIMD instruction is 128 bits, which means that it can look up 16 int8 values in one operation. Clearly, from an element-wise perspective, all the possible values of C^g that we enumerate need to be covered in a single lookup. Otherwise, we would need to use a bit-wise approach, performing bit-by-bit lookups, which sacrifices the memory access benefits obtained from the element-wise method. For example, in the case of ternary LLMs, with the limitation of 128-bit register length, we can enumerate at most $\frac{3^3}{2}$ possible values in the lookup table, which restricts $g \leq 3$. Assuming we disregard the limitation of instruction length, we simulate a longer instruction length using the original instructions without considering precision. As shown in Figure 11, as the length of SIMD registers increases, the number of enumerable g values grows, thereby significantly reducing computational complexity. Theoretically, when $C^g = M$, the computational complexity introduced by enumerating LUTs surpasses that of table lookup and accumulation, and further increasing the length of SIMD registers no longer yields additional benefits. It is significant that the g values we can currently enumerate are still far from the intersection point. Therefore, increasing the register length provides a definite benefit in terms of computational complexity. This also indicates that the potential of ELUT has not yet reached its theoretical limit.

C TL Algorithm

Unpack		Pack
-1	-1	0000
-1	0	0001
-1	1	0010
0	-1	0011
0	0	0100
0	1	0101
1	-1	0110
1	0	0111
1	1	1000

Table 5: TL1 Kernel transforms every two full-precision weights into 4-bit index and performs LUT computation.

Algorithm 3: TL1 mpGEMM

```
Input: Activation $A$ of shape $N, K$
Input: Weights $W$ of shape $M, K$
Output: Result matrix $R$ of shape $M, N$
1  IndexWeight = PreprocessWeights(W, M, K)
2  LUT = PreCompute(A, N, K)
3  for $n, m \leftarrow 1$ to $N, M$ do
4      $R[n, m] = \sum_{k=1}^{K/2} \text{Lookup}(LUT, IndexWeight, n, m, k)$
5  end
6  Function PreCompute($A, N, K$):
7      for $n, k \leftarrow 1$ to $N, K/2$ do
8          for $i \leftarrow 1$ to $3^2$ do
               /* $Unpack$ shows in Table 5                          */
9              $LUT[n, k, i] = Unpack_i(A[n, 2k], A[n, 2k+1])$
10         end
11     end
12     return $R$
13 Function PreprocessWeights($W, M, K$):
14     for $m, k \leftarrow 1$ to $M, K/2$ do
           /* $Pack$ shows in Table 5                                */
15         $IndexWeight[m, k] = Pack(W[m, 2k], W[m, 2k+1])$
16     end
17     return $IndexWeight$
```

Unpack			Pack
-1	-1	-1	1 1101
-1	-1	0	1 1100
-1	-1	1	1 1011
-1	0	-1	1 1010
...			
0	0	0	0 0000
...			
1	0	1	0 1010
1	1	-1	0 1011
1	1	0	0 1100
1	1	1	0 1101

Table 6: TL2 Kernel compresses every three full-precision weights into a 1-bit sign (0 or 1) and a 4-bit index.

Algorithm 4: TL2 mpGEMM

```
Input: Activation $A$ of shape $N, K$
Input: Weights $W$ of shape $M, K$
Output: Result matrix $R$ of shape $M, N$
IndexWeight, Signweight = PreprocessWeights($W$, $M$, $K$)
LUT = PreCompute($A$, $N$, $K$)
for $n, m \leftarrow 1$ to $N, M$ do
    $R[n, m] = \sum_{k \leftarrow 1}^{K/3}$ Lookup$(LUT, IndexWeight, n, m, k)$
    $R[n, m] = Signweight \times R[n, m]$
end
Function PreCompute($A, N, K$):
    for $n, k \leftarrow 1$ to $N, K/3$ do
        for $i \leftarrow 1$ to $3^3/2$ do
            /* $Unpack$ shows in Table 6                                */
            $LUT[n, k, i] = Unpack_i(A[n, 3k], A[n, 3k+1], A[n, 3k+2])$
        end
    end
    return $R$
Function PreprocessWeights($W, M, K$):
    $SignWeight = Sign(W)$
    $W = |W|$
    for $m, k \leftarrow 1$ to $M, K/3$ do
        /* $Pack$ shows in Table 6                                      */
        $IndexWeight[m, k] = Pack(W[m, 3k], W[m, 3k+1], W[m, 3k+2])$
    end
    return $IndexWeight, SignWeight$
```

D Performance

CPU	Model Size	Kernels							
		general kernels			ternary kernels				
		Float16	Q4_0	T-MAC	TQ1_0	TQ2_0	TL1_0	TL2_0	I2_S
		b(16)	b(4.5)	b(2)	b(1.69)	b(2.06)	b(2)	**b(1.67)**	b(2)
Intel i7-13700H 20C 64G	700M	30.73	67.57	76.29	114.20	123.94	75.62	**126.99**	125.37
	1.5B	15.02	35.46	42.38	64.86	71.92	43.44	74.16	**77.75**
	3.8B	5.85	16.33	18.12	26.59	33.19	17.91	**35.43**	35.04
	7B	3.30	9.09	12.29	17.96	19.92	11.89	**20.72**	20.62
	13B	1.78	5.04	6.44	10.55	11.21	6.32	**11.41**	10.62
	30B	N/A	2.13	2.54	4.62	5.25	2.65	4.99	**5.70**
	70B	N/A	0.94	1.32	2.09	2.32	1.49	**2.42**	2.30
	100B	N/A	0.67	0.73	1.48	1.61	0.75	**1.69**	1.65
APPLE M2	700M	110.65	197.38	220.22	217.64	237.61	214.53	229.21	**238.16**
	1.5B	59.49	117.77	135.27	130.10	**145.68**	132.68	138.28	143.43
	3.8B	28.31	71.89	91.84	73.14	88.66	90.73	**92.12**	91.65
	7B	14.87	39.47	53.37	45.55	54.90	52.77	**55.42**	54.74
	13B	8.42	23.28	31.72	25.83	**34.63**	32.12	33.22	32 .88
	30B	3.78	10.98	16.40	12.85	15.46	15.02	**19.59**	16.41
	70B	1.71	5.16	9.48	6.30	8.16	9.23	**10.37**	8.39
	100B	N/A	3.56	6.45	4.53	6.18	6.34	**7.45**	6.50

Table 7: Comparison of inference speed across different CPU (Unit: Tokens/Second) in an unlimited thread setting. $b(x)$ represents the bits per weight, where x denotes specific value. "N/A" indicates that the tested CPU cannot host the specified model size with the given kernel. The token generation speed was determined by calculating the average results from 10 tests conducted across different devices using diverse methodologies.