# BitLLM: Efficient Low-Bit, Conversion-Hiding LLM Decoding on Mobile GPUs

Hangyu Zheng
hyzheng@uga.edu
University of Georgia
Athens, GA, USA

Zhihao Shu
Zhihao.Shu@uga.edu
University of Georgia
Athens, GA, USA

Kunxiong Zhu
Kunxiong.Zhu@uga.edu
University of Georgia
Athens, GA, USA

Minghai Qin
qinminghai@gmail.com
Western Digital Research
San Jose, CA, USA

Xiaolong Ma
xiaolongma@arizona.edu
University of Arizona
Tucson, AZ, USA

Gagan Agrawal
gagrawal@uga.edu
University of Georgia
Athens, GA, USA

Wei Niu
wniu@uga.edu
University of Georgia
Athens, GA, USA

## Abstract

The rapid scaling of large language models (LLMs) has unlocked remarkable capabilities across various domains. However, deploying LLMs on mobile devices remains challenging because of their high computational and memory requirements. Low-bit quantization (e.g., 4-bit weights, 8-bit activations) can reduce model size while preserving accuracy, but its practical efficacy on mobile GPUs has been limited so far. In this paper, we identify that the performance gap stems primarily from *frequent data type conversions* between quantized and floating-point values, and *dynamic and inefficient memory access patterns* caused by the growing Key-Value (KV) cache during autoregressive decoding. We introduce `BitLLM`, a mobile inference engine that addresses type conversion overhead and suboptimal memory accesses in quantized LLMs. Our approach features an adaptive layout and packing strategy for the KV cache that ensures efficient memory access throughout LLM decoding. This is coupled with an integer-native execution pipeline that defers and overlaps type conversion outside the compute *hot path*, keeping the critical `MatMul` stages in the integer domain. Implemented on off-the-shelf mobile GPUs, `BitLLM` outperforms leading engines, i.e., LLAMA.CPP, MNN-LLM, LiteRT-LM, and MLC-LLM, achieving average speedups of 1.78×, 1.73×, 1.58×, and 1.47×, respectively, across various models and quantization levels. Under long-context generation, `BitLLM` reduces average energy consumption (per token) by up to 88.9%.



This work is licensed under a Creative Commons Attribution 4.0 International License.

*ASPLOS '27, Heraklion, Greece.*

© 2027 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-3012-2/2027/04
https://doi.org/10.1145/3845814.3850028

***CCS Concepts:*** • **Human-centered computing** → **Ubiquitous and mobile devices**; • **Computing methodologies** → **Neural networks**.

***Keywords:*** Large language models, Quantization, Mobile computing

**ACM Reference Format:**
Hangyu Zheng, Zhihao Shu, Kunxiong Zhu, Minghai Qin, Xiaolong Ma, Gagan Agrawal, and Wei Niu. 2027. BitLLM: Efficient Low-Bit, Conversion-Hiding LLM Decoding on Mobile GPUs. In *Proceedings of the 32nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '27), April 11–15, 2027, Heraklion, Greece.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3845814.3850028

## 1 Introduction

Large language models (LLMs) are now central to modern AI, having deeply impacted areas such as natural language processing, multimodal reasoning, and human-computer interaction [26, 64, 67]. This has driven growing demand to run LLMs on mobile devices [41, 53, 60], where on-device inference offers privacy, lower latency, resilience to connectivity issues [35, 44], and lower data-center energy costs by shifting inference away from centralized infrastructure.

However, LLMs are inherently resource-intensive, requiring billions of parameters, multi-gigaflop computations per token, and substantial memory bandwidth for expanding Key-Value (KV) caches [77]. These demands strain the limited computational capacity and memory budget of mobile processors [57, 80]. Bridging this gap requires both algorithmic compression and hardware-aware system design. Quantization, converting high-precision floating-point parameters and activations into compact integer formats, has been shown to maintain model accuracy at 4–8-bit precision while significantly reducing memory footprint, bandwidth

requirements, and energy consumption per token [20, 21, 27, 36, 77, 79, 87]. On server platforms, quantization delivers substantial speedups by leveraging native low-bit execution paths such as Tensor Cores [37, 62, 69, 73, 84, 88]. In contrast, mobile GPUs often lack native support for low-bit arithmetic. As shown in Table 1, INT8 inference on a Snapdragon 8 Gen 3 GPU is up to 2× *slower* than FP16, while INT8 on the CPU (with native low-bit instructions) of the same SoC is 2× faster. Without dedicated integer execution instructions, every low-bit operation requires type promotion and conversion: unpacking eight INT4 values from a 32-bit word costs approximately 16 shift and 16 mask operations, exceeding the 8 multiply and 7 add operations of the dot-product itself (more details in Section 3).

Beyond these general DNN considerations, LLMs face a second bottleneck on mobile: the dynamically growing KV cache. Mobile GPUs organize memory as 2D textures with RGBA vector lanes [46, 85], and the optimal data layout depends on the current decoding sequence length. Existing frameworks [68, 75, 76] apply a fixed layout throughout decoding, leading to misaligned memory accesses and periodic latency spikes as the KV cache grows.T-MAC [75] and BitNet.cpp [68] target CPU table-lookup computation, which produces uncoalesced accesses on GPUs and cannot exploit the 2D texture memory hierarchy. T-MAN [76] relies on NPU-specific table-lookup units and static data layouts that cannot adapt to the dynamically growing KV cache during decoding. Efficient quantized execution on mobile GPUs thus remains an open problem, particularly when the optimal layout shifts at runtime with changing sequence length.

We propose `BitLLM`, a mobile GPU inference engine that realizes this formulation through runtime layout adaptation, hardware-aligned packing, and conversion-hiding integer execution. We target 4-/8-bit group-wise quantization, which prior work [20, 37] has shown to achieve best balance of efficiency and accuracy, rather than aggressive formats (e.g., 2-bit) [5, 6, 68] that often compromise accuracy. Although layout selection [28], low-bit packing [75], and kernel pipelining [14] have each been studied, prior approaches generally optimize them independently or assume fixed execution configurations [76]. In contrast, we formulate mobile-GPU low-bit decoding as a dynamic, quantization-aware layout-selection problem and use a lightweight Layout Efficiency Proxy (LEP) to guide runtime layout and packing choices as the KV cache grows. Our contributions are as follows:

- An empirical analysis showing that layout mismatch in the growing KV cache and data type conversion overhead are the key bottlenecks causing low-bit quantization to underperform on mobile GPUs.
- A novel LEP-guided framework that combines quantization with a layout-aware runtime, enabling efficient low-bit LLM execution on mobile GPUs.
- A set of novel mechanisms, including adaptive layouts for KV cache, hardware-aligned value packing, and an integer-native pipeline that defers type conversion outside the compute hot path.
- Extensive evaluation showing `BitLLM` outperforms leading frameworks across multiple models and mobile GPUs.

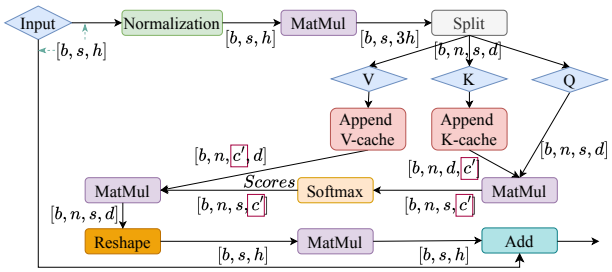



**Figure 1. Computational graph of an autoregressive transformer decoder.** Each new query ($Q$) attends to all cached keys (*K-cache*) and values (*V-cache*), causing the KV cache to grow and dynamically alter tensor shapes.

`BitLLM` is evaluated on four LLMs with *grouped-query attention (GQA)* and *traditional multi-head attention (MHA)*, namely Llama-3.2 [24], Qwen3 [82], Gemma-2 [65], and Phi-2 [29], under multiple precisions (FP16, W8A8, W4A8)[1], context lengths, and mobile GPU platforms. `BitLLM` achieves average speedups of 1.78× over LLAMA.CPP [11], 1.73× over MNN-LLM [74], 1.58× over LiteRT-LM [12, 23], and 1.47× over MLC-LLM [9, 42], while eliminating the latency spikes observed in other frameworks during KV cache growth. Average per-token energy consumption is reduced by 56–76% relative to the state-of-the-art baselines. `BitLLM` enables stable execution of long-context models on resource-constrained devices where other frameworks exhibit severe degradation.

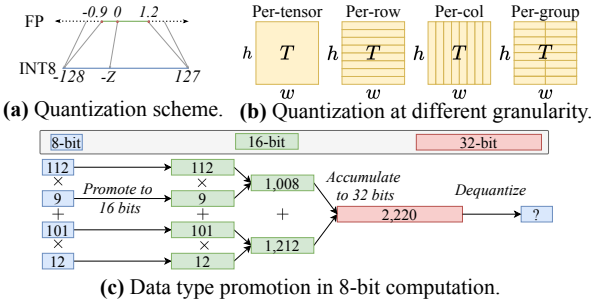
## 2 LLM and Quantization Background

### 2.1 Computation Flow in Transformer Decoder

An autoregressive LLM generates tokens sequentially, with each step depending on all previously produced tokens. At step $N$, the model processes the current input embedding while appending new projections to a growing KV cache. This cache helps avoid recomputation but its growing size introduces challenges to relate to total memory and memory access costs/efficiency.

Figure 1 illustrates the computation graph of a Transformer decoder layer during autoregressive decoding. Let $b$ denote the batch size and $s$ the number of tokens processed at the current step (typically $s$=1). Let $c'$ be the updated *cache* length ($c' = c_{\text{prev}} + s$). For the model, $n$ is the *number of attention heads*, $d$ is the *number of dimensions per head*, and $h = n \cdot d$ is the *hidden size*. The input embedding has shape $[b, s, h]$. After normalization and a linear projection, the representation becomes $[b, s, 3h]$, which is split into queries ($Q$),

[1]FP16: 16-bit floating-point for weight and KV cache. W8A8: 8-bit weights and 8-bit KV cache. W4A8: 4-bit weights and 8-bit KV cache.




**(a)** Quantization scheme. **(b)** Quantization at different granularity.

**(c)** Data type promotion in 8-bit computation.

**Figure 2. Overview of quantization techniques.** (a) Quantization scheme. (b) Quantization at different granularities. (c) Type conversion pipeline for INT8 computation, involving promotion to INT16/INT32 and dequantization back.

keys ($K$), and values ($V$), and reshaped into

$$Q, K, V \in \mathbb{R}^{[b,n,s,d]}. \tag{1}$$

The new keys and values are appended to the caches,

$$K_{\text{cache}} \in \mathbb{R}^{[b,n,d,c']}, \qquad V_{\text{cache}} \in \mathbb{R}^{[b,n,c',d]}, \tag{2}$$

The memory layouts for these caches are discussed later; broadly, they are chosen to avoid transpose operations in the subsequent attention computation. The attention mechanism then computes similarity scores between the current queries and the cached keys:

$$Scores = QK^{\top} \in \mathbb{R}^{[b,n,s,c']}. \tag{3}$$

After applying a softmax along the $c'$ dimension, the result is multiplied by the value cache to obtain the context tensor

$$S = \text{Softmax}(Scores), \qquad SV \in \mathbb{R}^{[b,n,s,d]}. \tag{4}$$

This result is reshaped back to $[b, s, h]$, passed through an output projection, and combined with the residual connection to form the block output $[b, s, h]$. In the final decoder layer, the output is projected into the vocabulary space to produce *logits* for next-token prediction.

While $s$ remains small and fixed in decoding, the cache length $c'$ grows linearly with the number of generated tokens. This growth causes the attention computations to increase linearly and introduces different shapes of matrix multiplication. Thus, although the KV cache enables efficient reuse of past computations, its expansion also inflates memory consumption and complicates GPU execution optimization strategies such as static tiling and memory coalescing.

### 2.2 LLM Quantization

Quantization maps high-precision floating-point values into discrete and compact integer domains, reducing model size and lowering arithmetic cost. A real-valued tensor $x$ can be quantized using a uniform affine mapping with floating-point scale $s > 0$ and integer zero-point $z \in \mathbb{Z}$:

$$q = \text{clip}\left(\text{round}(x/s) + z,\ q_{\min}, q_{\max}\right), \qquad \hat{x} \approx s \cdot (q - z), \tag{5}$$

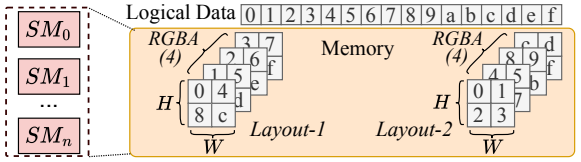



**Figure 3. Illustration of 2D texture memory layout.**

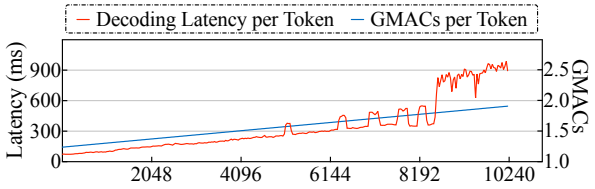



**Figure 4. Decoding latency and computational workload per token as functions of decode sequence index.**

where $q$ is an integer within the target bit-width. As shown in Figure 2(a), this affine mapping projects floating-point values into a bounded integer range (i.e., -128 to 127).

To balance accuracy and efficiency, various quantization schemes have been proposed. They also differ in the granularity at which the scale and zero-point are chosen. Figure 2(b) illustrates some of these different approaches. *Per-tensor quantization* applies a single scale across the entire tensor, offering low overhead but suffering from sensitivity to outliers [20]. *Per-row/column quantization* assigns independent scales per row or column, improving accuracy for tensors with heterogeneous distributions. *Per-group quantization* generalizes this idea by partitioning tensors into groups of fixed size, balancing error reduction and implementation cost. Typical group sizes are 32 to 128 [36].

### 2.3 Constraints of Mobile GPUs

As shown in Figure 3, the memory subsystem of mobile GPUs can be viewed as a 2-dimensional texture space [46]: a 2D grid with dimensions width $W$ and height $H$, where each point stores a fixed-width vector – typically four elements, like an RGBA pixel. This structure, optimized for workloads with inherent 2D spatial locality, allows the dedicated cache to efficiently fetch and cache data in 2D blocks – a fundamental divergence from the linear memory space in CPUs. However, the ability to benefit from this feature depends on data layouts that align with the 2D memory space, as misalignment leads to mandatory padding, uncoalesced accesses, and bandwidth under-utilization. The optimal layout is generally dictated by the specific computation being performed. As an illustration of possibilities, Figure 3 also illustrates two distinct data layouts for a tensor of length 16, where Layout-1 uses row-major ordering, and Layout-2 leverages column-major ordering.

On server-class platforms, quantization is effectively supported through optimized low-bit instructions such as those

**Table 1. Comparison of LLM decoding latency on mobile platforms with FP16 versus low-bit representation.**

| Model | Data Type | Latency per Token | | GPU Speedup |
|---|---|---|---|---|
| | | CPU | GPU | |
| TinyLLama-1B [86] | FP16 | 61.9ms | 56ms | 1.11× |
| | INT8 | 31.2ms | 67.8ms | **0.46**× |
| Qwen3-1.7B [82] | FP16 | 78.8ms | 73.6ms | 1.07× |
| | INT8 | 45.4ms | 130.2ms | **0.35**× |
| Llama-3.2-3B [24] | FP16 | 140.5ms | 137.4ms | 1.02× |
| | INT8 | 80.1ms | 159.2ms | **0.50**× |

seen in AVX-512, VNNI, and CUDA-INT4/INT8 extensions. However, mobile GPUs are primarily designed for FP16/FP32 pipelines. Figure 2(c) shows the sequence of data type conversion required for INT8 operands in the absence of specialized instructions: (i) unpacking, (ii) promotion to INT16 for multiplication, (iii) accumulation into INT32, and (iv) dequantization back to FP32 for subsequent operations. These steps add significant computational cost and register pressure.

## 3 Empirical Study on LLM Inference

To understand the performance impact of the steps outlined above, we profile the decoding latency of various LLMs on a commercial mobile device (OnePlus 12) with MNN-LLM [74], comparing FP16 and INT8 on both CPU and GPU. The results, summarized in Table 1, first confirm the expected pattern for FP16 execution: GPUs consistently outperform CPUs, leveraging their superior floating-point throughput. However, INT8 quantization reverses this trend. INT8 reliably accelerates inference on the CPU (i.e., as compared to FP16), by leveraging reduced memory bandwidth and efficient integer arithmetic. However, on the mobile GPU, INT8 consistently results in degraded performance, with execution that is even slower than the CPU's INT8 path. This inversion stems from the overhead of type promotion and bit-level manipulation that low-bit computation requires on mobile GPUs – operations that existing frameworks do not handle efficiently.

The dynamically growing KV cache introduces a second bottleneck in the memory subsystem. Figure 4 highlights the impact of KV cache size on latency. The blue curve shows that the arithmetic workload (measured in GMACs per token) increases in a consistent linear fashion with sequence length. In contrast, the red curve reveals that decoding latency per token grows much faster and exhibits frequent stair-step spikes. These spikes occur when changes in KV tensor size and shape cause the existing layout to become suboptimal.

## 4 System Design

The key idea of `BitLLM` is to make low-bit LLM execution layout-adaptive on mobile GPUs. The motivation for this approach comes from the fact that layout and unpacking costs interact on the critical path, optimizing one in isolation yields limited gains. More specifically, we first formalize the notion of the Layout Efficiency Proxy (LEP) (Section 4.1), which models the memory and data unpacking costs that dominate the runtime. Grounded in this formalization, we introduce three complementary mechanisms: i) adaptive layouts that evolve dynamically with sequence length (Section 4.2), ii) quantization-aware packing designed to minimize unpacking overhead (Section 4.3), and iii) a type conversion-hiding computation pipeline that preserves these gains through low-bit integer execution (Section 4.4). We use $H$ (attention heads), $D$ (hidden dimension per head), and $T$ (current sequence length) for the transformer dimensions from Section 2.1.

### 4.1 Optimization Overview

Two cost factors jointly determine runtime performance and guide our optimizations. The first is *data access cost*. Different attention operators favor different stride-1 axes: $QK^T$ benefits from $D$-contiguous layouts for stride-1 query access, while $SV$ benefits from $T$-contiguous layouts for efficient token traversal. A misaligned layout forces strided fetches and under-utilizes the GPU's RGBA vector lanes. The data access cost, in summary, depends on the layout configuration and further changes with sequence length. The second factor is *unpacking overhead*. As shown in Figure 5, an FP32 8-element dot product uses 8 `FMA` instructions, but low-bit values must be unpacked into byte- or word-aligned operands before the mobile GPU can perform arithmetic on them. INT8 values are already byte-aligned, requiring only 8 `mul` and 7 `add` for accumulation. INT4 values, however, are sub-byte: eight 4-bit values share a single 32-bit word, and extracting them requires roughly 16 `shift` and 16 `mask` operations on top of the 8 `mul` and 7 `add`, which can rival or exceed the arithmetic cost itself. Because this overhead depends on the packing scheme, it must be modeled explicitly rather than folded into a generic compute cost.

To guide our optimization decisions, we define the Layout Efficiency Proxy (LEP) for a layout $\mathcal{L}$ of tensor $t$. Here, a layout configuration $\mathcal{L} = (\pi, \rho, \phi)$ specifies three choices:

- $\pi$ (contiguous-axis order) determines which logical tensor axis is stride-1 in memory, such as head $H$, hidden dimension $D$, or token $T$.
- $\rho$ (2D texture placement) determines how the ordered tensor axes are mapped to texture rows and columns on the mobile GPU.
- $\phi$ (low-bit value packing) determines how 4-/8-bit values are arranged within a machine word or RGBA lanes.

The LEP is formulated as a weighted sum of two dominant cost factors:

$$\mathrm{LEP}(\mathcal{L}, t) = \alpha_1 \cdot \frac{\mathrm{Bytes}(\pi, \rho, \phi, t)}{\mathrm{BW}_{\mathrm{tex}}} + \alpha_2 \cdot \mathrm{UnpackOps}(\phi) \quad (6)$$

The key is that both LEP components can be efficiently computed by combining analytical modeling with static analysis.

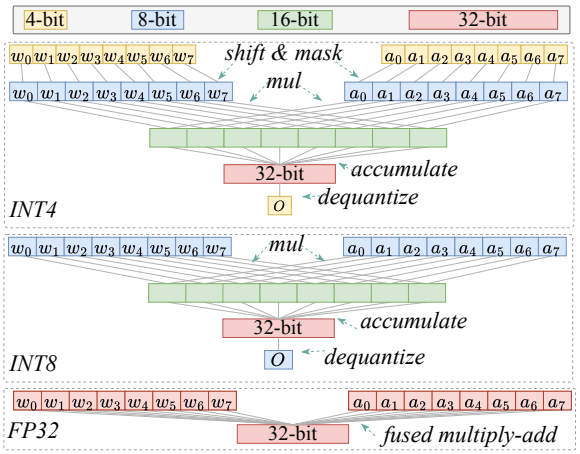



**Figure 5. Execution pipelines for a length-8 dot product with INT4, INT8, and FP32 formats.** FP32 uses direct fused multiply-add (`fma`); INT8 performs integer `mul` with 32-bit `accumulation`; INT4 additionally unpacks sub-byte weights with `shift` and `mask` instructions.

- Data access cost (Bytes$(\cdot)$/BW$_{\text{tex}}$): this term is derived *analytically* by modeling the tensor ($t$) access patterns against the memory mapping defined by $\pi$ and $\rho$. For an operator and $t$, we simulate the memory traversal pattern to count the bytes moved between DRAM and on-chip cache, considering cache line sizes and bank conflicts.
- Unpacking cost (UnpackOps$(\phi)$): this term is quantified through *static code analysis*. For each quantization packing scheme $\phi$, we generate the corresponding kernel code and count the number of integer arithmetic and bitwise operations (shifts, masks, AND/OR) required to unpack a single packed word. This operation count is normalized to represent the overhead per arithmetic operation.
- Weight calibration ($\alpha_1$, $\alpha_2$): $\alpha_1$ and $\alpha_2$ are device-specific normalization weights calibrated once per GPU through offline microbenchmarks (<10 minutes) that separately stress the memory subsystem and integer ALU. The weights are model-independent and remain fixed across all models and sequence lengths. On our evaluated GPUs, $\alpha_1 = 0.5$ and $\alpha_2 = 0.7$ yield the best layout selection accuracy.

To make LEP evaluation efficient at runtime, we first pre-compute Bytes$(\pi, \rho, \phi, t)$ and UnpackOps$(\phi)$ for all supported layout configurations during an offline profiling phase. During runtime selection, `BitLLM` queries these existing values and performs the weighted sum in Equation 6. This hybrid analytical-static approach allows `BitLLM` to efficiently evaluate thousands of $(\mathcal{L}, t)$ combinations with minimal overhead. The LEP serves as a comparative metric to identify the most efficient layout configuration for the dominant operators in transformers (i.e., `MatMul`).

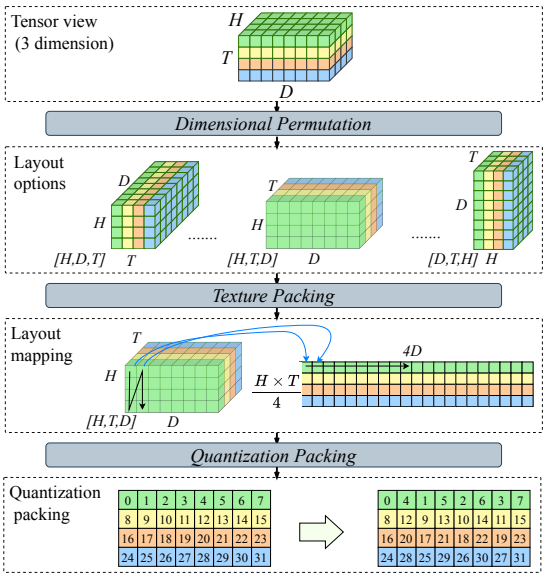



**Figure 6. Layout optimization pipeline.**

Figure 6 shows an example of the layout optimization pipeline for tensor $K$ with shape $[H, D, T]$ under configuration $\mathcal{L}$. First, the dimension $D$ is chosen as the stride-1 axis since computation is performed along $D$, ensuring consecutive cache access. The tensor is then mapped into 2D texture space along the $H$ dimension, and finally, the low-bit quantized values are rearranged and packed according to $\phi$ for efficient dequantization. The subsequent sections describe our detailed mechanisms for minimizing the LEP.

### 4.2 Adaptive Layouts for Decoding

Having established the LEP as our efficiency metric, we now present the adaptive layout mechanism that minimizes this proxy. The challenge translates to dynamically selecting, for each KV cache operation, the layout configuration $\mathcal{L} = (\pi, \rho, \phi)$ that yields the lowest LEP given the current tensor $t$ and KV cache.

***Layout adaptation with sequence length.*** This section addresses the Bytes/BW$_{\text{tex}}$ term through runtime adaptation. Specifically, in layout adaptation with sequence length, the optimal stride-1 axis for the KV cache is a function of sequence length. *Short contexts* (e.g., < 256) are dominated by the $QK^\top$ operation, favoring a $D$-contiguous layout. *Long contexts* (e.g., > 512) are dominated by $SV$, favoring a $T$-contiguous layout. Choosing a static layout leads to inefficient memory access and severe latency spikes (Figure 4).

We adopt a greedy layout evolution strategy. Algorithm 1 formalizes this procedure, which periodically evaluates candidate layouts and selects the one minimizing LEP for the current and near-future sequence lengths. The algorithm generates a set of candidate layouts $\mathcal{C} = \{(\pi, \rho, \phi)\}$ by enumerating permissible stride-1 axis permutations (e.g., $D$-major vs. $T$-major) along with corresponding texture packing schemes $\rho$ that comply with the mobile GPU's memory constraints.

**Algorithm 1:** Greedy Layout Evolution

**Input: :** Current layout $\mathcal{L}_{current}$, sequence length $t$, cache dimensions $(H, D, T)$
**Output: :** New layout $\mathcal{L}_{new}$

$\mathcal{C} \leftarrow \text{GenerateCandidates}(H, D, T)$
**foreach** *candidate* $\mathcal{L}_c \in \mathcal{C}$ **do**
  $\text{LEP}_c \leftarrow \alpha_1 \cdot \frac{\text{Bytes}(\mathcal{L}_c, t)}{\text{BW}_{\text{tex}}} + \alpha_2 \cdot \text{UnpackOps}(\mathcal{L}_c)$
$\mathcal{L}_{best} \leftarrow \arg\min_{\mathcal{L}_c \in \mathcal{C}} \text{LEP}_c$
**if** $LEP_{best} < LEP_{current}$ **then**
  $\mathcal{L}_{new} \leftarrow \mathcal{L}_{best}$
**else**
  $\mathcal{L}_{new} \leftarrow \mathcal{L}_{current}$

It evaluates each candidate by computing its LEP score at the current sequence length $t$, estimating the efficiency of the predominant operator (e.g., $QK^\top$ or $SV$) under that layout. Because LEP uses precomputed values and one weighted sum per candidate, selection completes in microseconds with negligible decoding overhead. The lowest-LEP candidate is adopted only if it measurably improves upon the current configuration; otherwise, the current layout is retained.

***Folding along non-reduction axes.*** Mobile GPUs impose strict limits on the width and height of texture memory (e.g., ≤ 16K pixels on each dimension). As the KV cache grows in long context, tensor dimensions often exceed these limits, making direct mapping into 2D textures infeasible. A naive folding strategy partitions oversized dimensions arbitrarily. However, if folding occurs along a *reduction axis* – e.g., folding the hidden dimension $D$ in $QK^\top$ or folding the token dimension $T$ in $SV$ – the reduction spans across multiple tiles. This introduces strided, cross-cache-tile fetches, increasing memory traffic and degrading cache locality.

We adopt an operator-aware folding policy. The key insight is that *folding should preserve contiguous access along the dimension that is reduced during a computation-intensive operation (e.g.,* `MatMul`*)*, as it is critical for achieving a stride-1 memory access pattern and high memory bandwidth utilization. Consequently, whenever folding is necessary, it is applied exclusively along non-reduction axes. This ensures the reduction dimension remains contiguous, preserving stride-1 access. As shown in Figure 7, for $QK^\top$ (reduction over $D$), we fold along $T$; and for $SV$ (reduction over $T$), we fold along $D$. This principle extends to other operators like `Softmax`, where reduction occurs along the last dimension, so we fold along preceding dimensions. This policy leverages operator semantics to make a locally optimal choice with zero search overhead.

***Append-only segments and phase-aware transitions.*** As the sequence length increases, the initially optimal KV cache layout can become suboptimal. Conventional systems typically resort to one of two costly approaches: 1) performing a re-layout of the entire cache – a process that copies hundreds of megabytes of data and imposes significant overhead, or 2) leaving the layout unchanged and accepting suboptimal performance. Both result in sharp latency spikes. A naive alternative is to switch layouts greedily at every small improvement in the Layout Efficiency Proxy (LEP). While this avoids re-layout, it produces many small fragments of the cache, significantly increasing iterator overhead and complicating memory management.

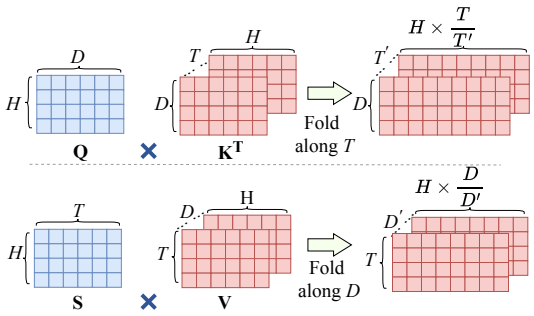



**Figure 7. Two folding examples.**

We propose an append-only segmentation strategy coupled with phase-aware transitions. The key idea is *Append-only Segments*, where KV cache is divided into immutable segments. Each segment records its layout $(\pi, \rho, \phi)$ and kernel schedule parameters (e.g., unroll factor, block size). When LEP indicates that a more efficient configuration is available, new tokens are appended into a new segment configured with that new layout. Previous segments remain intact, eliminating the need for copying or re-layout. To prevent excessive fragmentation, we design *phase-aware transitions*. Specifically, we apply a heuristic to LEP evaluation and trigger layout switches only when improvements persist over a sufficient sequence length. This compresses short-term fluctuations into a few stable computational phases, often aligned with operator-level shifts, such as transitions from $QK^\top$-heavy to $SV$-heavy workloads. This design avoids copying or relayout during transitions; switching only initializes metadata for a new segment. Across our 10K-token runs, it creates 2.25 segments on average and at most five. Because each segment is contiguous, the runtime traverses only a small number of segments, and total segmentation overhead remains below 1% of end-to-end decoding latency.

### 4.3 Quantization-Aware Packing

Low-bit quantization reduces memory footprint but introduces significant unpacking overhead when data formats are misaligned with hardware execution units. Modern mobile GPUs expose native integer dot-product instructions (e.g., `dp4a` and `udot8`), which multiply multiple INT8 pairs and accumulate the result into an INT32 register in a single cycle. To exploit these primitives effectively, operands must be packed into specific 32-bit vector layouts.

Naively packing low-bit values (e.g., INT4) contiguously into a 32-bit word misaligns them with byte-addressable

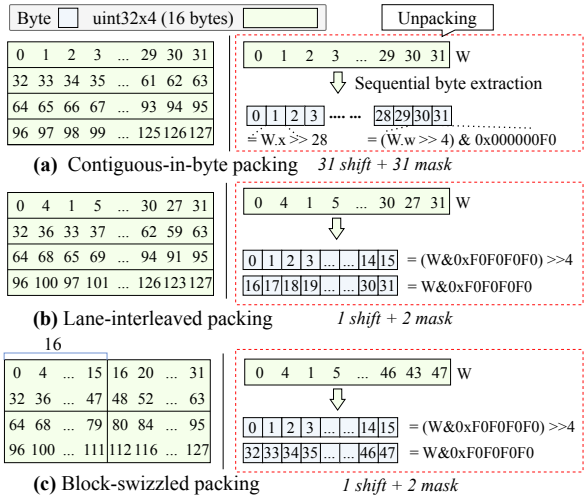



**Figure 8. Different quantization packing schemes.**

hardware. Eight linearly packed INT4 values cross byte boundaries, forcing the GPU to extract each value and expand it to INT8 before integer arithmetic. As quantified in Section 4.1, this expansion can cost more than the computation itself.

To mitigate this inefficiency, we treat quantization packing as a schedulable dimension and redesign it to minimize the `Unpacking` term in LEP. Specifically, we introduce a set of structured packing families. Figure 8 illustrates the three schemes. The *contiguous-in-byte scheme* maximizes storage density but requires about 31 shift and 31 mask operations to extract values from a four-word vector, the highest unpack cost. *Lane-interleaved packing* rearranges nibbles so that each byte holds two aligned INT4 values, matching the four RGBA lanes of texture memory. This reduces unpacking to a fixed sequence of one shift and two masks per word, independent of tensor shape. *Block-swizzled packing* further organizes values into $2 \times 2$ texel tiles aligned with texture cache blocks, combining the constant-time unpacking of lane interleaving with improved memory coalescing.

Formally, if packed operands are expressed as

$$A = \sum_{i=0}^{3} a_i \cdot 2^{8i}, \qquad B = \sum_{i=0}^{3} b_i \cdot 2^{8i}, \quad a_i, b_i \in \mathbb{Z}_8,$$

the `dp4a` instruction computes

$$\mathrm{dp4a}(A, B, r_{\mathrm{acc}}) = r_{\mathrm{acc}} + \sum_{i=0}^{3} (a_i \times b_i), \quad r_{\mathrm{acc}} \in \mathbb{Z}_{32}.$$

Our packing strategies guarantee that unpacked INT4 values expand directly into the aligned INT8 operands required by such primitives, with only constant-time bit operations.

Compared to naive packing, where each word requires up to 20 unpacking operations, our reordered packing reduces the requirement to approximately three instructions. Microbenchmarks show that this reduces unpacking overhead by about 80% and increases integer pipeline utilization accordingly. Without this, the instruction cost of unpacking can negate the memory savings from quantization.

***Discussion: applicability to other GPUs.*** Our implementation targets Adreno and Mali GPUs (Section 5.4), but the packing design requires only 8-bit integer dot-product accumulate instructions. Table 2 shows that similar primitives exist on AMD, Intel, and NVIDIA GPUs, making the approach portable beyond mobile. Extension to CPUs and NPUs is less straightforward: low-bit systems on CPU/NPU [68, 75, 76] use table-lookup over SIMD lanes co-designed with linear memory hierarchies, whereas our packing and layout co-design targets warp-parallel execution. Comparison across these platforms is a direction for future work.

**Table 2.** 8-bit dot-product primitives on different platforms.

| Platform | Instruction |
|---|---|
| Adreno/MALI GPU | `udot8_acc` / `dot8_acc` |
| AMD GPU (GCN/RDNA) | `v_dot4_u32_u8` / `v_dot4_i32_i8` |
| Intel GPU (Xe) | `dpas.s32.s8.s8` |
| NVIDIA GPU | `dp4a.s32.s8.s8` |

### 4.4 Type Conversion-Hiding Computation

Conventional low-bit computation on mobile GPUs incurs repeated data type conversions. INT8 multiplication requires upcasting to INT16 to avoid overflow, and many frameworks further convert quantized tensors to FP16 for `MatMul`. These conversions account for over 30% of arithmetic instructions in INT4 execution and leave the integer dot-product units underutilized, because conversion overhead sits directly on the critical path.

We introduce an integer-native, tile-based pipeline that eliminates conversions from the core computation path. We reorganize the computation around tiles, which are small and contiguous blocks of the weight or activation matrix that serve as the fundamental unit of data transfer and parallel computation. Our design is built on three key principles: 1) maintaining all hot-path computations in the integer domain, 2) deferring dequantization and floating-point conversion to an epilogue stage, and 3) overlapping this epilogue asynchronously with the computation of subsequent tiles. This represents a paradigm shift from *converting-then-computing* to *computing-then-converting*.

As shown in Figure 9, the pipeline is organized at the tile granularity and comprises three tightly overlapped stages. First (*Load & Unpack*), tiles are loaded in their compactly-packed form from texture memory. Leveraging the reordered packing scheme from the previous section, we unpack INT4 data into byte-aligned INT8 values using only ≈3 bitwise operations (1 `shift` and 2 `mask`) per 32-bit word – a 5× improvement over the 16 operations needed for naive contiguous packing. This unpacking occurs uniformly in registers, eliminating thread divergence and minimizing UnpackOps.

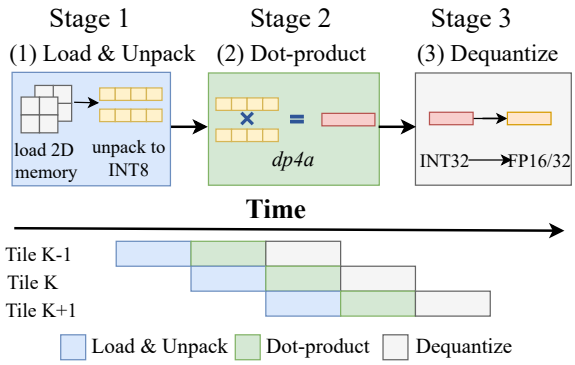



**Figure 9. Tile-based pipelining for type conversion-hiding computation.** The pipeline consists of three stages: (1) *Load & Unpack*, (2) *Integer Compute*, and (3) *Deferred Conversion*. Operations overlap across tiles, ensuring the hot path remains fully in the integer domain.

In the second stage (*Dot-product*), the unpacked, byte-aligned tiles are immediately fed into integer dot-product primitives (e.g., `dp4a`, `udot8`), which accumulate results directly into INT32 registers. This stage is entirely free of type conversions, allowing the integer units to operate at peak utilization. Tile sizes are co-designed with the adaptive layouts (Section 4.2) to balance register pressure and memory coalescing, keeping traffic near the LEP lower bound.

In the third stage (*Dequantization*), the integer accumulations are converted to FP16/FP32 and fused with dequantization. This epilogue runs only at tile boundaries, asynchronously via warp specialization. As shown in Figure 9, warp group A computes on tile $k$ while group B loads tile $k{+}1$ and group C converts the results of tile $k{-}1$. Synchronization is needed only at tile boundaries, which effectively removes conversion latency from the critical path. Except when parallelism or compute slack is limited, as in very short contexts (sequence lengths below 12 in our kernels) or small matrices, the pipeline effectively overlaps conversion with computation. Such cases are uncommon in our evaluated workloads. Even under these conditions, conversion remains outside the inner dot-product loop, and only a portion of the epilogue latency is exposed on the critical path.

The conventional pipeline requires two conversions per tile (unpacking before computation and dequantization after), which creates register traffic and leaves the integer ALUs idle. Our pipeline eliminates conversions from the hot path; the single deferred dequantization per tile is hidden behind the next tile's computation. In practice, this reduces conversion instructions by over an order of magnitude and unpack overhead by ≈80% (Section 5.3).

## 5 Evaluation

This section evaluates `BitLLM`'s effectiveness in terms of latency, energy consumption, memory/cache efficiency, optimization contributions, and portability across mobile GPUs.

**Table 3. Model perplexity on the WikiText2 [40] dataset (lower is better) for different precisions.**

| Model | #Params | Perplexity ↓ | | |
|---|---|---|---|---|
| | | FP16 | W8A8 | W4A8 |
| Llama-3.2-1B [24] | 1.24B | 13.75 | 14.31 | 15.34 |
| Qwen3-1.7B [82] | 1.72B | 9.84 | 9.97 | 10.28 |
| Gemma-2-2B [65] | 2.61B | 18.09 | 20.33 | 21.40 |
| Phi-2-2B [29] | 2.78B | 17.41 | 18.15 | 19.94 |

### 5.1 Experiment Setup

***Workloads.*** We select four representative LLMs that are suitable for mobile GPU deployment: *Llama-3.2-1B* [24], *Qwen3-1.7B* [82], *Gemma-2-2B* [65], and *Phi-2-2B* [29]. They cover three attention mechanisms: grouped-query attention (GQA) as in Llama-3.2 and Qwen3, multi-head attention (MHA) as in Phi-2, and alternating local-global attention in Gemma-2. These models are widely used in on-device studies and fit within the 8–16GB unified memory (shared with the OS and other apps) of current mobile GPUs; larger models (e.g., 7–8B) remain impractical at full precision. For each model, we evaluate two low-bit precision settings: *W8A8*, with 8-bit weights, activations, and KV cache under group-wise scaling; and *W4A8*, with 4-bit weights and 8-bit activations and KV cache — a balance between compression and accuracy validated by prior work [37, 77]. The group size is 32, following [21, 36]. Although our main focus is low-bit quantization, we include *FP16* to show our adaptive layouts benefit higher-precision computation as well.

Before focusing on latency, energy, and memory behavior in the rest of this section, we demonstrate that the quantization levels chosen here result in high accuracy. Table 3 reports model-level accuracy on standard benchmarks. Across all four models, W8A8 incurs only a modest perplexity increase over FP16 (0.1–2.2 points), while W4A8 remains within 0.4–3.3 points of FP16, confirming that group-wise quantization at these precisions preserves model quality.

`BitLLM` is independent of the quantization algorithm and focuses on execution efficiency. Therefore, for a given quantization scheme, `BitLLM` provides the same model accuracy as other frameworks. The remainder of our evaluation therefore focuses on system-level performance.

***Baselines.*** We compare `BitLLM` with 4 widely used LLM inference engines on mobile GPUs: LLAMA.CPP [11] (31914624), MNN-LLM [74] (7f1197e0), LiteRT-LM [23] (26f2178, powered by TensorFlow-Lite [12]), and MLC-LLM [42] (8f49ea6b, powered by TVM [9]). Several related frameworks, such as bitnet.cpp [68] and other works including Transformer-Lite [34], Ladder [69], PowerInfer-2 [81], and ExecuTorch [51], do not support the tested models on mobile GPUs. This is because they lack operator implementations or otherwise have hardware dependencies. In the reported results, we enable the auto-tuning capabilities available in all baselines to

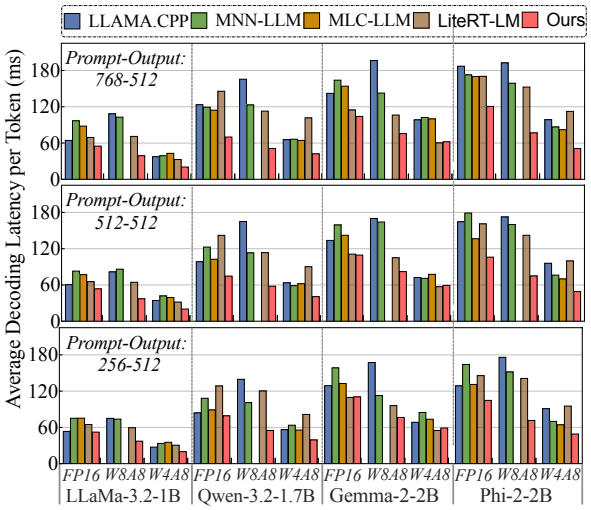



**Figure 10. Average latency per token comparison across models, precisions, and Prompt–Output ($P$–$O$) window.**

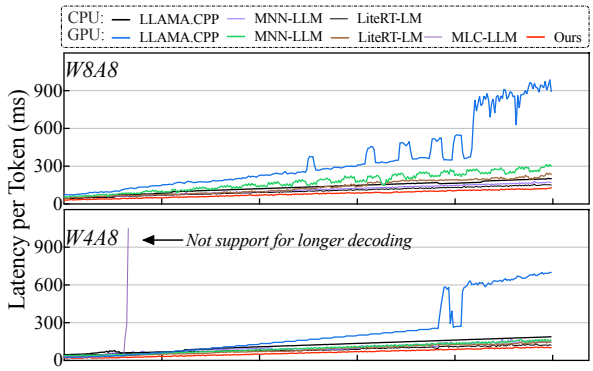



**Figure 11. Per-token decoding latency as sequence length grows (up to $10K$ tokens).** MLC-LLM does not support W8A8 or long contexts at W4A8. Our method (red) achieves the lowest latency and reduces the stair-step spikes observed in baselines.

achieve the best possible performance. The batch size is set to 1 in mobile environments. Energy consumption is collected through *Android system interface*. We run each experiment 50 times and report the average, as the variance is negligible.

***Testbed.*** Our primary platform is a OnePlus 12 with a Snapdragon 8 Gen 3 SoC, an Adreno 750 GPU, and 16 GB of unified memory. To assess portability (Section 5.4), we also evaluate a OnePlus 11 with a Snapdragon 8 Gen 2 SoC, an Adreno 740 GPU, and 16 GB of memory to test same-vendor generalization, and a Google Pixel 8 with a Tensor G3 SoC, a Mali-G715 GPU, and 8 GB of memory to validate cross-vendor portability under tighter resource constraints.

### 5.2 End-to-End Performance Comparison

This section compares the end-to-end performance of `BitLLM` with the state-of-the-art baselines on mobile GPUs across various context lengths and model precisions.

***Average latency with fixed context length.*** We report average per-token *decoding* latency across four models and four baselines under three Prompt–Output ($P$–$O$) windows: 768–512, 512–512, and 256–512, where, e.g., 768–512 denotes a 768-token prompt followed by 512 autoregressively decoded tokens. As shown in Figure 10, `BitLLM` achieves lower per-token latency than all baselines across all configurations. Against LLAMA.CPP, the average speedups are 1.29× (FP16), 2.47× (W8A8), and 1.59× (W4A8); against MNN-LLM, 1.55×, 2.06×, and 1.60×; against LiteRT-LM, 1.37×, 1.84×, and 1.63×; against MLC-LLM, 1.38× (FP16) and 1.56× (W4A8) (this framework does not support W8A8). FP16 also benefits from our adaptive layouts, though the gains are more modest because conversion overhead is not a major factor here.

Notably, `BitLLM` achieves 1.66× (W8A8) and 2.36× (W4A8) speedups over FP16, recovering 83% and 59% of the theoretical 2×/4× bounds. The gap stems from non-quantized components (e.g., `Softmax` and `Normalization`) and fixed runtime overheads that are not reduced with bit-width. In contrast, baselines capture far less: LLAMA.CPP achieves only 0.75× at W8A8 (even slower than FP16) and 1.56× at W4A8; MNN-LLM reaches 1.25×/1.86×; LiteRT-LM reaches 1.14×/1.79×; MLC-LLM attains 1.76× at W4A8. This demonstrates that `BitLLM` translates quantization potential into end-to-end speedups more effectively than prior systems.

Moreover, `BitLLM`'s advantage widens in longer prompts: against LLAMA.CPP, the average speedup grows from 1.62× (256–512) to 1.98× (768–512); similar trends hold for MNN-LLM (1.61×→1.85×), LiteRT-LM (1.50×→1.68×), and MLC-LLM (1.34×→1.62×). Longer prompts intensify prefill computations and expand the KV cache, increasing sensitivity to memory layout. Baselines apply a fixed layout regardless of shape, incurring growing penalties, whereas our cache-aware runtime adapts layouts as tensor shapes evolve and our conversion-hiding execution mitigates the overhead.

***Working with a fixed latency budget.*** Under a fixed latency budget, `BitLLM` can use its efficiency gains to run higher-precision configurations and improve model quality. Across the four models and three $P$–$O$ windows in Figure 10, `BitLLM` at FP16 achieves geometric-mean speedups of 1.75×, 1.46×, and 1.24× over LLAMA.CPP, MNN-LLM, and LiteRT-LM at W8A8, respectively. MLC-LLM is omitted because it does not support W8A8. Similarly, `BitLLM` at W8A8 outperforms LLAMA.CPP, MNN-LLM, LiteRT-LM, and MLC-LLM at W4A8 by 1.08×, 1.07×, 1.04×, and 1.02×, respectively. Table 3 confirms the corresponding quality benefit: FP16 reduces perplexity by 0.13–2.24 points over W8A8, while W8A8 reduces it by 0.31–1.79 points over W4A8. Thus, `BitLLM` can either reduce latency at fixed precision or improve model quality under a fixed latency budget.

***Latency scaling with increasing context length.*** Next, we analyze how speedups evolve with decoding length. We fix $P$ and vary $O$ from 1 to $10K$. Due to space constraints, we

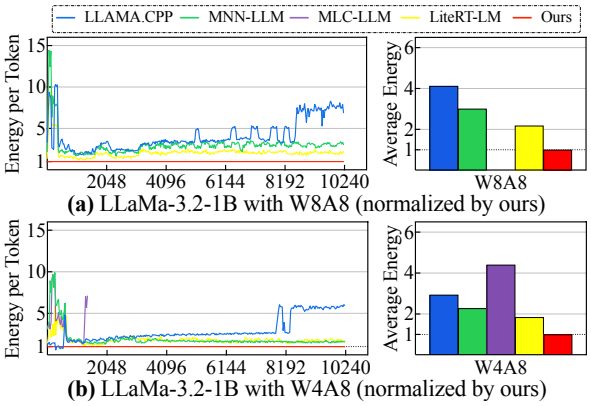



**Figure 12. Normalized (by ours) per-token and average energy consumption for low-bit precisions.**

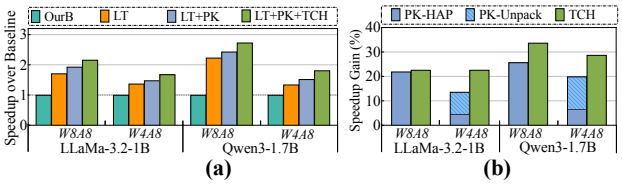



**Figure 13. Optimization breakdown on mobile GPU.** (a) Cumulative speedup over `OurB` (baseline = 1) as each optimization is enabled. `LT`, `PK`, and `TCH` denote adaptive layout, quantization-aware packing, and type-conversion hiding, respectively. (b) Fine-grained breakdown of the incremental speedup gains from `PK` and `TCH` over `OurB`. The `PK` gain is separated into hardware-aligned packing (`PK-HAP`) and reduced INT4 unpacking (`PK-Unpack`).

report results for a representative model (Llama-3.2-1B), as other models exhibit similar trends. We present results for the low-bit precisions W8A8 and W4A8.

Figure 11 shows that `BitLLM` consistently outperforms all baselines as $O$ increases. Our per-token latency remains flat, whereas all baselines exhibit periodic spikes from increasingly inefficient layouts as the KV cache grows. `BitLLM` achieves average speedups of 3.25×, 1.68×, and 1.50×, respectively, over LLAMA.CPP, MNN-LLM, and LiteRT-LM at W4A8. MLC-LLM degrades catastrophically at longer contexts (speedup over 43.81×) due to its static layout design; we therefore exclude it from the primary long-context comparison. At W8A8, the speedups are 3.96×, 2.14×, and 1.81× against LLAMA.CPP, MNN-LLM, and LiteRT-LM. Notably, for these baselines the CPU backend often outperforms their own GPU backend at low-bit precisions, owing to richer native support for low-precision integer arithmetic on CPUs. Even against their best results across platforms, which happen to be the CPU backends, `BitLLM` on GPU still achieves substantial speedups: 2.11×, 1.65×, and 1.56× over LLAMA.CPP, MNN-LLM, and LiteRT-LM at W8A8, and 1.74×, 1.42×, and 1.38× at W4A8.

Speedups grow monotonically with $O$ as KV cache expansion amplifies layout sensitivity. From $O$=1 to $O$=10$K$, the gain over LLAMA.CPP rises from 2.0× to 8.35× (W8A8) and 1.1× to 6.93× (W4A8); over MNN-LLM, from 1.49× to 2.66× (W8A8) and 1.38× to 2.48× (W4A8); over LiteRT-LM, from 1.24× to 2.03× (W8A8) and 1.21× to 1.98× (W4A8). The trend is pronounced against MLC-LLM at W4A8, where speedup escalates from 1.36× to 527×, as its static layout incurs increasingly severe penalties with KV cache growth. These results confirm that our adaptive layout and integer-native execution mitigate the overhead of growing context length.

***Energy consumption per token and average profile.*** Figure 12 reports normalized per-token energy (left) and average energy (right) for Llama-3.2-1B under W8A8 and W4A8, with $O$ from 1 to 10$K$ and a fixed prompt of 512. `BitLLM` reduces average per-token energy by 75.7%/66.0% (W8A8/W4A8) over LLAMA.CPP, 66.7%/56.1% over MNN-LLM, 54.4%/44.2% over LiteRT-LM, and 88.9% over MLC-LLM at W4A8. Consistent with the latency, our energy consumption remains stable as $O$ grows, whereas baselines show rising energy overhead with longer sequences.

### 5.3 Performance Breakdown and Analysis

***Optimization breakdown.*** Figure 13(a) reports the cumulative speedup of each optimization for two representative models (Llama-3.2-1B and Qwen3-1.7B) under a fixed $P$–$O$ window of 512–512 at W8A8 and W4A8 precisions. Other models exhibit similar trends and are omitted for brevity. All bars are normalized to `OurB`, a variant of `BitLLM` with all key optimizations disabled; each subsequent bar adds one optimization cumulatively. Accordingly, the differences between adjacent bars quantify the incremental contribution of each component at each precision: `OurB`→`LT` for adaptive layout, `LT`→`LT + PK` for quantization-aware packing, and `LT + PK`→`LT + PK + TCH` for the type-conversion-hiding pipeline. At W8A8, `LT` (adaptive layout) delivers 1.71× and 2.23× speedups on Llama-3.2-1B and Qwen3-1.7B, respectively. Adding `PK` (quantization-aware packing) yields an additional 1.22× and 1.26×; `TCH` (type-conversion hiding) contributes a further 1.23× and 1.34×. At W4A8, the corresponding incremental gains are 1.37× and 1.34× (`LT`), 1.1× and 1.2× (`PK`), 1.2× and 1.29× (`TCH`). Among all components, `LT` contributes the largest improvement by preserving data locality through layout adaptation, enabling memory bandwidth to be utilized more effectively (further quantified in the following analysis).

***Quantization-aware packing breakdown.*** Figure 13(b) further shows the incremental gains over `OurB` that are the cumulative speedups reported in Figure 13(a). It decomposes `PK` gain into hardware-aligned packing (`PK-HAP`) and reduced sub-byte unpacking (`PK-Unpack`), and reports `TCH` separately as the contribution of the subsequent conversion-hiding kernel pipeline. `PK-HAP` captures the benefit of matching the

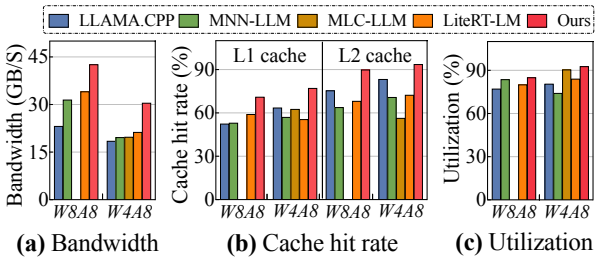
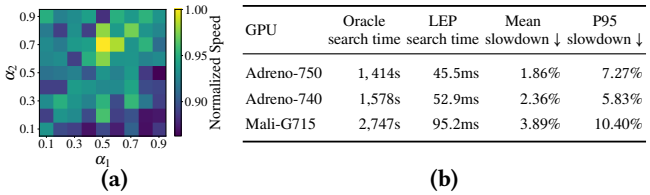



**Figure 14. Memory, cache, and ALU utilization analysis.**




| GPU | Oracle search time | LEP search time | Mean slowdown ↓ | P95 slowdown ↓ |
|---|---|---|---|---|
| Adreno-750 | 1,414s | 45.5ms | 1.86% | 7.27% |
| Adreno-740 | 1,578s | 52.9ms | 2.36% | 5.83% |
| Mali-G715 | 2,747s | 95.2ms | 3.89% | 10.40% |

**Figure 15. LEP calibration and selection analysis.** (a) Impact of different LEP weights $(\alpha_1, \alpha_2)$. (b) LEP search overhead and performance gap to oracle across mobile GPUs.

stored operand order to the RGBA texture lanes and dot-product instructions, whereas PK-Unpack isolates the reduction in shift-and-mask operations for packed INT4 values. At W8A8, PK-HAP contributes 21.9% and 25.7% on Llama-3.2-1B and Qwen3-1.7B, respectively, while PK-Unpack has no contribution because INT8 values require no sub-byte extraction. At W4A8, PK-HAP contributes 4.63% and 6.71%, and PK-Unpack contributes 8.95% and 13.2%, respectively. The subsequent TCH optimization contributes 22.6% and 33.7% at W8A8 and 22.6% and 28.7% at W4A8.

The W8A8 packing gain therefore comes entirely from PK-HAP. Although the default W8A8 representation is byte-aligned, its element order does not match the fixed lane order in which a 32-bit dot-product instruction consumes the four values returned by an RGBA texture fetch. This mismatch requires lane rearrangement. PK-HAP instead stores each group in consumption order, allowing the fetched values to feed the instruction directly. For the profiled Llama-3.2-1B workload($P$–$O$=512–512, Adreno 750), Figure 14 shows that W8A8 exhibits 25.4%–60.2% higher measured bandwidth than W4A8 across baselines supporting both precisions, with mean L1/L2 cache-hit-rate decreases of 3.9%/6.3%. Given this higher memory pressure, BitLLM achieves a larger bandwidth gain at W8A8, reaching up to 1.84× the measured baseline bandwidth, compared with 1.65× at W4A8. This contrast explains the larger PK-HAP benefit in Figure 13(b), while W4A8 gains more from reduced INT4 unpacking.

***Memory, cache, and ALU analysis.*** The preceding breakdown quantifies each optimization's contribution; we now profile the hardware-level metrics that explain these gains. Figure 14 reports memory bandwidth, cache behavior, and ALU utilization measured via Snapdragon Profiler [52]. Figure 14 (a) shows GPU memory bandwidth at W8A8 and W4A8 (MLC-LLM is excluded at W8A8 as it does not support this precision). BitLLM achieves the highest bandwidth across both precisions, reflecting the aligned access patterns produced by adaptive layouts and operator-aware folding (Section 4.2). Figure 14 (b) shows cache hit rates: BitLLM achieves 13.6%–18.6% higher L1 and 10.4%–37.3% higher L2 hit rates than all baselines.

W4A8 yields higher hit rates than W8A8 due to the reduced memory footprint. This also explains why W4A8 baselines can outperform W8A8 on mobile GPUs. In the profiled configuration, relative to W8A8, W4A8 reduces memory traffic by 39.8%, increases the L1 and L2 cache hit rates by 21.14% and 12.3%, respectively, and increases unpacking overhead by 9.04%. For memory- and cache-sensitive decoding, reduced data movement and improved cache locality can outweigh the additional INT4 unpacking cost. Conversely, the higher memory pressure of W8A8 increases the benefit of hardware-aligned packing, consistent with the larger PK-HAP contribution in Figure 13(b). The improved cache locality follows from stride-1 access preservation through non-reduction-axis folding and length-aware layout selection. Figure 14 (c) reports shader ALU utilization. BitLLM achieves the highest utilization at both precisions, with improvements of up to 18.6% at W8A8 and 7.9% at W4A8. This gain results from quantization-aware packing (reducing unpacking instructions by ≈80%) and conversion-hiding execution, which reduce unpacking work and remove type conversion from the compute critical path, respectively.

***Analysis of LEP weights.*** Figure 15(a) presents the sensitivity of LEP to different $\alpha_1$ and $\alpha_2$ settings on our primary testbed. The specific values are obtained from the offline calibration (in Section 4.1), which completes in less than 10 minutes. The parameters are model-independent, and the best-performing setting ($\alpha_1 = 0.5$, $\alpha_2 = 0.7$) is selected and used across all models. Even drastically different parameter values result in only about a 10%–15% performance degradation. Incorporating automated tuning strategies (e.g., Bayesian optimization) to further streamline the calibration process is left as future work. Figure 15(b) compares LEP with exhaustive oracle search across representative mobile GPUs. Oracle search benchmarks every candidate layout, requiring 1413.6–2747.3 s across the three GPUs. LEP completes layout selection in only 45.5–95.2 ms, reducing search time by more than four orders of magnitude. The selected layouts remain close to the oracle, with mean slowdowns of 1.86–3.89% and P95 slowdowns of 5.83–10.40%. These results demonstrate stable selection quality across mobile-GPU architectures.

### 5.4 Portability Evaluation

We assess portability on two additional platforms: the OnePlus 11 (same vendor, earlier generation) and the Google

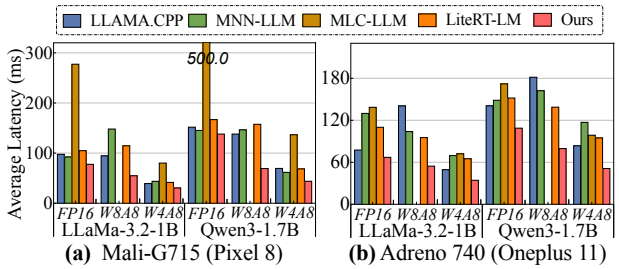



**Figure 16. Portability evaluation.** Empty bars indicate that the framework does not support the configuration due to missing operator support or insufficient device memory.

Pixel 8 (cross-vendor, more constrained resources). Figure 16 reports per-token latency for Llama-3.2-1B and Qwen3-1.7B at FP16, W8A8, and W4A8 under $P$–$O$=512–512. `BitLLM` runs all model–precision combinations on both devices. At low-bit precisions, `BitLLM` achieves 1.3×–2.9× speedups on OnePlus 11 and 1.4×–3.6× on Pixel 8 (average 1.7× vs. 1.9×, as tighter resources amplify the benefit of our optimizations). At FP16, speedups of 1.1×–2.1× and 1.1×–3.6× confirm that adaptive layouts alone provide significant benefits.

## 6 Related Work

***Quantization for LLMs.*** Classical quantization methods for deep learning have evolved into LLM-specific schemes that preserve model accuracy at aggressively low-bit widths [58]. LLM.int8 [18] halves memory relative to FP16 with an outlier-aware 8-bit scheme. SmoothQuant [77] migrates activation scaling into weights for W8A8 quantization. GPTQ [21] performs one-shot weight quantization for accurate 3–4 bit deployment. MobileQuant [63] develops mobile-friendly quantization algorithms for on-device applications, which is complementary to our setting. Quantization has also been applied to reduce KV cache costs: KVQuant [27] provides calibration-based 4-bit KV quantization, and Q-Hitter [87] incorporates token-oracle signals for better compression. SNIP [49] explores sub-byte mixed-precision for LLM training, while AWQ [36] rescales salient channels for 4-bit quantization. These works primarily target server-class platforms and their specialized low-bit instruction sets, such as CUDA INT4/INT8 and AVX-512 VNNI. Mainstream mobile GPUs (e.g., Adreno, Mali) lack such dedicated instruction support; our work is designed for these architectures.

***Low-bit computation optimizations.*** A line of work on precision-scalable hardware [2, 17, 31, 54, 55] scales execution width to operand precision, but largely targets custom accelerators. T-MAC [75] and BitNet.cpp [68] explore table-lookup-based low-bit computation on edge-device CPUs, but their access patterns map poorly to GPUs due to uncoalesced memory accesses and misalignment with GPU memory hierarchies. PMPD [7] focuses on mixed-precision scheduling, adapting precision across inference phases and over the generated sequence to improve efficiency on resource-constrained hardware. On the server side, Atom [88] and QServer [38] integrate mixed-precision quantization with CUDA tensor-core kernels unavailable on mobile GPUs. At format and language level, M2XFP [28] proposes a metadata-augmented microscaling format for low-bit quantization, and Tilus [19] provides a tile-level GPU programming language for low-precision computation. Other closely related work generates quantized kernels whose *lane-interleaving* or *bit-packing layouts* are dictated by specific SIMD instructions (e.g., ARM NEON [14], x86 VNNI [56], and Tensor Core MMA [22]), and are resolved *statically* at compile time or during offline preprocessing.

***LLM serving and compiler frameworks.*** Several frameworks target LLM inference on server GPUs [10, 13, 50, 78]. DeepSpeed-Inference [4] combines kernel fusion, tensor parallelism, and quantization-aware execution. FlashAttention [15, 16] improves attention kernel utilization through optimized work partitioning. SGLang [90] and Vidur [1] explore scheduling and serving strategies. oFFN [61] accelerates inference with an outlier-aware structured FFN design. For KV cache management, vLLM [32] reduces fragmentation via paged attention. Paged KV-cache management and speculative decoding have been developed primarily for server-class LLM serving, where memory management in high-throughput request processing are central. In contrast, `BitLLM` targets efficient low-bit decoding on resource-constrained mobile GPUs, but these techniques remain complementary. A paged KV cache could be integrated by treating pages or page groups as layout-managed segments. Speculative decoding would require additional KV-state management; we leave their implementation and evaluation to future work. Data layout also matters for efficient DNN execution [46]. TVM [9] optimizes graph-level fusion and layout selection; Ansor [89] and TMModel [25] search over loop transformations and tiling with learned cost models. LADDER [69] transforms computations into statically optimized low-precision pipelines with memory-layer-specific tensor transformations. Triton [66] enables tile-centric code generation with near-vendor-library performance.

These compiler systems primarily optimize code generation and static schedules for predetermined layouts. `BitLLM` instead uses LEP to select quantization-aware layouts dynamically across decoding phases and couples the selected layouts with a type-conversion-hiding execution pipeline.

***On-device LLM inference.*** While earlier mobile inference systems focused on convolution-heavy workloads [30, 33, 45, 83, 85], the shift to LLMs has driven a new generation of on-device runtimes [3, 39, 47, 70]. LLAMA.CPP [11] and LiteRT-LM [23] provide cross-platform runtimes leveraging vector instructions and low-bit formats. MLC-LLM [42], built on TVM, enables low-bit LLM execution on mobile GPUs. However, as our evaluation reveals, these frameworks inherit static layout assumptions that do not adapt to the

dynamic memory access patterns of autoregressive decoding, leaving a performance gap on mobile GPUs. SoD$^2$ [43] and Transformer-Lite [34] target dynamic-shape inference and memory planning for mobile deployment, achieving significant speedups through static shape propagation and operator-level optimizations. FlashMem [59] optimizes the mobile GPU memory hierarchy for LLM workloads, though it targets floating-point operations and does not handle the growing KV cache during decoding. Neuralink [72] reduces on-device LLM compute through neuron co-activation linking, which is orthogonal to the layout and type conversion overhead we focus on. Production frameworks including ExecuTorch [51] and ONNXRuntime [48] primarily target mobile CPUs, with limited low-bit GPU backend support. Other on-device efforts target NPUs and heterogeneous SoC execution, including PowerInfer [62, 81], Ripple [71], LLM.NPU [80], HeteroLLM [8], and T-MAN [76], but do not address the GPU-oriented, low-bit decoding challenges we focus on; extending our optimizations to such heterogeneous scenarios is a promising future direction. Overall, `BitLLM`'s distinction lies not in any individual layout, packing, or pipelining technique, but in their sequence-aware coordination for mobile-GPU decoding. The underlying framework, LEP, dynamically makes layout and packing decisions as the KV cache grows, while the conversion-hiding pipeline preserves these gains during execution.

## 7 Discussion and Future Work

***Generality across execution environments.*** At the LEP level, supporting a new mobile GPU requires updating the supported layout candidates and recalibrating the device-specific weights once offline, as detailed in Section 4.1 and evaluated across GPUs in Section 5.3. For *a heterogeneous SoC* (i.e., with CPU, GPU, and NPU), LEP can be extended by scoring backend–layout pairs and incorporating an inter-backend transition term:

$$\mathrm{LEP}_{b'\to b}(\mathcal{L},t) = \alpha_{1,b}\frac{\mathrm{Bytes}_b(\pi,\rho,\phi,t)}{\mathrm{BW}_b} + \alpha_{2,b}\,\mathrm{UnpackOps}_b(\phi) + T_{\mathrm{trans}}(b'\to b,\mathcal{L},t).$$

Here, $b', b \in \{\mathrm{CPU},\mathrm{GPU},\mathrm{NPU}\}$ denote the source and target backends. The current GPU implementation assumes 2D texture and RGBA access, GPU-oriented packing, and low-bit dot-product instructions. Extending LEP to CPUs and NPUs would preserve the cost-model structure but require backend-specific layouts, memory models, and execution primitives (i.e., instruction sets). The transition term $T_{\mathrm{trans}}$ captures synchronization, data movement, and other required layout conversion between backends; it is zero when $b' = b$. The heterogeneous extension would retain LEP's lightweight selection procedure. For a fixed source backend $b'$, LEP evaluates $\sum_b |\mathcal{C}_b|$ backend–layout candidates. Because the backend-specific LEP parameters are precomputed, each candidate is evaluated with lightweight computation.

*Concurrent workloads* can change the effective backend and bandwidths, causing parameters obtained through offline calibration to become suboptimal. A *contention-aware* extension could update these parameters at decoding-phase boundaries based on resource-availability information provided by the runtime scheduler and then rescore the affected candidates. Extending to contention-aware heterogeneous execution is a promising direction.

***Extending to new quantization schemes.*** `BitLLM` currently evaluates W8A8 and W4A8 using a group size of 32; however, its layout-selection and packing formulation is not specific to these configurations. Supporting an additional quantization format requires incorporating compatible packing candidates and extending the LEP cost components to capture format-specific overheads. Specifically, the Bytes term could account for quantized data, whereas the UnpackOps term accounts for bit extraction, operand rearrangement, and type conversion. These costs may depend on both group size and bit width: group size determines dequantization granularity, while bit width determines packing density, unpacking cost, and the availability of compatible integer primitives. For each target device, a packing candidate is associated with a supported execution path; when the packed representation cannot be consumed directly by a native primitive, LEP can further incorporate the required unpacking and conversion costs. LEP then ranks the extended set of candidates using the resulting cost model. We leave additional quantization formats to future work.

## 8 Conclusion

We presented `BitLLM`, which addresses the unrealized potential of low-bit LLM inference on mobile GPUs through adaptive KV cache layouts, hardware-aligned packing, and an integer-native pipeline that defers type conversion outside the compute hot path. Extensive evaluation demonstrates that `BitLLM` delivers significant speedups (e.g., 1.78× over LLAMA.CPP on average with fixed context length) and substantial per-token energy reductions (up to 88.9%), while effectively eliminating latency spikes during KV cache growth to ensure predictable long-context decoding. Future work includes combining other low-bit quantization schemes with pruning co-optimization and support for concurrent workloads on heterogeneous mobile hardware (NPUs, TPUs).

## Acknowledgments

The authors would like to thank the anonymous reviewers and the shepherd for their valuable and constructive feedback. This work was supported in part by the National Science Foundation (NSF) under the awards of CCF-2428108, CCF-2333895, OAC-2403090, CSR-2341378, IIS-2540429, CNS-2555319, and CCF-2553684. Any errors and opinions are not those of the NSF and are attributable solely to the author(s).

## References


[1] Amey Agrawal, Nitin Kedia, Jayashree Mohan, Ashish Panwar, Nipun Kwatra, Bhargav S Gulavani, Ramachandran Ramjee, and Alexey Tumanov. 2024. Vidur: A large-scale simulation framework for llm inference. *Proceedings of Machine Learning and Systems* 6 (2024), 351–366.

[2] Jorge Albericio, Alberto Delmás, Patrick Judd, Sayeh Sharify, Gerard O'Leary, Roman Genov, and Andreas Moshovos. 2017. Bit-pragmatic deep neural network computing. In *Proceedings of the 50th annual IEEE/ACM international symposium on microarchitecture*. 382–394.

[3] Keivan Alizadeh, Seyed Iman Mirzadeh, Dmitry Belenko, S Khatamifard, Minsik Cho, Carlo C Del Mundo, Mohammad Rastegari, and Mehrdad Farajtabar. 2024. Llm in a flash: Efficient large language model inference with limited memory. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 12562–12584.

[4] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, et al. 2022. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[5] Jerry Chee, Yaohui Cai, Volodymyr Kuleshov, and Christopher M De Sa. 2023. Quip: 2-bit quantization of large language models with guarantees. *Advances in Neural Information Processing Systems* 36 (2023), 4396–4429.

[6] Hong Chen, Chengtao Lv, Liang Ding, Haotong Qin, Xiabin Zhou, Yifu Ding, Xuebo Liu, Min Zhang, Jinyang Guo, Xianglong Liu, et al. 2024. Db-llm: Accurate dual-binarization for efficient llms. *arXiv preprint arXiv:2402.11960* (2024).

[7] Hao Mark Chen, Fuwen Tan, Alexandros Kouris, Royson Lee, Hongxiang Fan, and Stylianos I Venieris. 2024. Progressive mixed-precision decoding for efficient llm inference. *arXiv preprint arXiv:2410.13461* (2024).

[8] Le Chen, Dahu Feng, Erhu Feng, Rong Zhao, Yingrui Wang, Yubin Xia, Haibo Chen, and Pinjie Xu. 2025. HeteroLLM: Accelerating Large Language Model Inference on Mobile SoCs platform with Heterogeneous AI Accelerators. *arXiv preprint arXiv:2501.14794* (2025).

[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, and Luis Ceze. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *OSDI*. 578–594.

[10] FasterTransformer contributors. 2022. *FasterTransformer*. https://github.com/NVIDIA/FasterTransformer

[11] LLaMA.CPP contributors. 2023. *LLaMA.CPP*. https://github.com/ggerganov/llama.cpp

[12] TensorFlow contributors. 2017. *LiteRT*. https://ai.google.dev/edge/litert

[13] TensorRT-LLM contributors. 2022. *TensorRT-LLM*. https://github.com/NVIDIA/TensorRT-LLM

[14] Meghan Cowan, Thierry Moreau, Tianqi Chen, James Bornholt, and Luis Ceze. 2020. Automatic generation of high-performance quantized machine learning kernels. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. 305–316.

[15] Tri Dao. 2023. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691* (2023).

[16] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems* 35 (2022), 16344–16359.

[17] Alberto Delmas, Patrick Judd, Sayeh Sharify, and Andreas Moshovos. 2017. Dynamic stripes: Exploiting the dynamic precision requirements of activation values in neural networks. *arXiv preprint arXiv:1706.00504* (2017).

[18] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. LLM.int8 (): 8-bit matrix multiplication for transformers at scale. *Advances in neural information processing systems* 35 (2022), 30318–30332.

[19] Yaoyao Ding, Bohan Hou, Xiao Zhang, Allan Lin, Tianqi Chen, Cody Hao Yu, Yida Wang, and Gennady Pekhimenko. 2026. Tilus: A Tile-Level GPGPU Programming Language for Low-Precision Computation. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 281–297.

[20] Shichen Dong, Wen Cheng, Jiayu Qin, and Wei Wang. 2024. Qaq: Quality adaptive quantization for llm kv cache. *arXiv preprint arXiv:2403.04643* (2024).

[21] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2022. GPTQ: Accurate post-training compression for generative pretrained transformers. *arXiv preprint arXiv:2210.17323* (2022).

[22] Elias Frantar, Roberto L Castro, Jiale Chen, Torsten Hoefler, and Dan Alistarh. 2025. Marlin: Mixed-precision auto-regressive parallel inference on large language models. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 239–251.

[23] Google team. 2026. *LiteRT-LM*. https://github.com/google-ai-edge/LiteRT-LM

[24] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[25] Jiexiong Guan, Zhenqing Hu, Christos D Antonopoulos, Nikolaos Bellas, Spyros Lalis, Evgenia Smirni, Gang Zhou, Gagan Agrawal, and Bin Ren. 2025. TMModel: Modeling Texture Memory and Mobile GPU Performance to Accelerate DNN Computations. In *Proceedings of the 39th ACM International Conference on Supercomputing*. 205–220.

[26] Yanchu Guan, Dong Wang, Zhixuan Chu, Shiyu Wang, Feiyue Ni, Ruihua Song, Longfei Li, Jinjie Gu, and Chenyi Zhuang. 2023. Intelligent virtual assistants with llm-based process automation. *arXiv preprint arXiv:2312.06677* (2023).

[27] Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W Mahoney, Yakun S Shao, Kurt Keutzer, and Amir Gholami. 2024. Kvquant: Towards 10 million context length llm inference with kv cache quantization. *Advances in Neural Information Processing Systems* 37 (2024), 1270–1303.

[28] Weiming Hu, Zihan Zhang, Haoyan Zhang, Chen Zhang, Cong Guo, Yu Feng, Tianchi Hu, Guanglin Li, Guipeng Hu, Junsong Wang, et al. 2026. M2XFP: A Metadata-Augmented Microscaling Data Format for Efficient Low-bit Quantization. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 1151–1167.

[29] Mojan Javaheripi, Sébastien Bubeck, Marah Abdin, Jyoti Aneja, Sebastien Bubeck, Caio César Teodoro Mendes, Weizhu Chen, Allie Del Giorno, Ronen Eldan, Sivakanth Gopi, et al. 2023. Phi-2: The surprising power of small language models. *Microsoft Research Blog* 1, 3 (2023), 3.

[30] Joo Seong Jeong, Jingyu Lee, Donghyun Kim, Changmin Jeon, Changjin Jeong, Youngki Lee, and Byung-Gon Chun. 2022. Band: coordinated multi-dnn inference on heterogeneous mobile processors. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*. 235–247.

[31] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M Aamodt, and Andreas Moshovos. 2016. Stripes: Bit-serial deep neural network computing. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 1–12.

[32] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 611–626.

[33] Royson Lee, Stylianos I Venieris, Lukasz Dudziak, Sourav Bhattacharya, and Nicholas D Lane. 2019. Mobisr: Efficient on-device super-resolution through heterogeneous mobile processors. In *The 25th Annual International Conference on Mobile Computing and Networking*. 1–16.

[34] Luchang Li, Sheng Qian, Jie Lu, Lunxi Yuan, Rui Wang, and Qin Xie. 2024. Transformer-lite: High-efficiency deployment of large language models on mobile phone gpus. *arXiv preprint arXiv:2403.20041* (2024).

[35] Xiangyu Li, Yuanchun Li, Yuanzhe Li, Ting Cao, and Yunxin Liu. 2024. FlexNN: Efficient and Adaptive DNN Inference on Memory-Constrained Edge Devices. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*. 709–723.

[36] 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. *Proceedings of machine learning and systems* 6 (2024), 87–100.

[37] Yujun Lin, Haotian Tang, Shang Yang, Zhekai Zhang, Guangxuan Xiao, Chuang Gan, and Song Han. 2024. Qserve: W4a8kv4 quantization and system co-design for efficient llm serving. *arXiv preprint arXiv:2405.04532* (2024).

[38] Yujun Lin, Haotian Tang, Shang Yang, Zhekai Zhang, Guangxuan Xiao, Chuang Gan, and Song Han. 2025. Qserve: W4a8kv4 quantization and system co-design for efficient llm serving. *Proceedings of Machine Learning and Systems* 7 (2025).

[39] llamafile contributors. 2023. *llamafile*. https://github.com/Mozilla-Ocho/llamafile

[40] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843* (2016).

[41] Guanhong Miao, A Adam Ding, and Samuel S Wu. 2022. Real-time privacy preserving disease diagnosis using ECG signal. *arXiv preprint arXiv:2202.03652* (2022).

[42] MLC team. 2023-2025. *MLC-LLM*. https://github.com/mlc-ai/mlc-llm

[43] Wei Niu, Gagan Agrawal, and Bin Ren. 2024. Sod2: Statically optimizing dynamic deep neural network execution. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 386–400.

[44] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. 2021. DNNFusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 883–898.

[45] Wei Niu, Xiaolong Ma, Sheng Lin, Shihao Wang, Xuehai Qian, Xue Lin, Yanzhi Wang, and Bin Ren. 2020. PatDNN: Achieving Real-Time DNN Execution on Mobile Devices with Pattern-based Weight Pruning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 907–922.

[46] Wei Niu, Md Musfiqur Rahman Sanim, Zhihao Shu, Jiexiong Guan, Xipeng Shen, Miao Yin, Gagan Agrawal, and Bin Ren. 2024. SmartMem: Layout Transformation Elimination and Adaptation for Efficient DNN Execution on Mobile. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, Vol. 3. La Jolla, CA, USA. doi:10.1145/3620666.3651384

[47] ollama contributors. 2023. *ollama*. https://ollama.com/

[48] ONNX Runtime developers. 2018. ONNX Runtime. https://github.com/microsoft/onnxruntime.

[49] Yunjie Pan, Yongyi Yang, Hanmei Yang, and Scott Mahlke. 2026. SNIP: An Adaptive Mixed Precision Framework for Subbyte Large Language Model Training. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '26)*.

[50] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2023. Efficiently scaling transformer inference. *Proceedings of machine learning and systems* 5 (2023), 606–624.

[51] Pytorch. 2023. *On-device AI across mobile, embedded and edge for PyTorch*. pytorch.org/executorch

[52] Qualcomm. 2016. Snapdragon Profiler. https://developer.qualcomm.com/software/snapdragon-profiler. https://developer.qualcomm.com/software/snapdragon-profiler

[53] Muhammad Kashif Saeed, Alanoud Al Mazroa, Bandar M Alghamdi, Fouad Shoie Alallah, Abdulrhman Alshareef, and Ahmed Mahmud. 2024. Predictive analytics of complex healthcare systems using deep learning based disease diagnosis model. *Scientific Reports* 14, 1 (2024), 27497.

[54] Sayeh Sharify, Alberto Delmas Lascorz, Kevin Siu, Patrick Judd, and Andreas Moshovos. 2018. Loom: Exploiting weight and activation precisions to accelerate convolutional neural networks. In *Proceedings of the 55th Annual Design Automation Conference*. ACM, 20.

[55] Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Joon Kyung Kim, Vikas Chandra, and Hadi Esmaeilzadeh. 2018. Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 764–775.

[56] Haihao Shen, Hanwen Chang, Bo Dong, Yu Luo, and Hengyu Meng. 2025. Efficient llm inference on cpus. In *Enhancing LLM Performance: Efficacy, Fine-Tuning, and Inference Techniques*. Springer, 33–46.

[57] Xuan Shen, Peiyan Dong, Lei Lu, Zhenglun Kong, Zhengang Li, Ming Lin, Chao Wu, and Yanzhi Wang. 2024. Agile-Quant: Activation-Guided Quantization for Faster Inference of LLMs on the Edge. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 18944–18951.

[58] Xuan Shen, Weize Ma, Jing Liu, Changdi Yang, Rui Ding, Quanyi Wang, Henghui Ding, Wei Niu, Yanzhi Wang, Pu Zhao, Jun Lin, and Jiuxiang Gu. 2025. QuartDepth: Post-Training Quantization for Real-Time Depth Estimation on the Edge. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[59] Zhihao Shu, Md Musfiqur Rahman Sanim, Hangyu Zheng, Kunxiong Zhu, Miao Yin, Gagan Agrawal, and Wei Niu. 2026. FlashMem: Supporting Modern DNN Workloads on Mobile with GPU Memory Hierarchy Optimizations. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[60] Zhihao Shu, Xiaowei Yu, Zihao Wu, Wenqi Jia, Yinchen Shi, Miao Yin, Tianming Liu, Dajiang Zhu, and Wei Niu. 2024. Real-time core-periphery guided ViT with smart data layout selection on mobile devices. *Advances in Neural Information Processing Systems* 37 (2024), 95744–95763.

[61] Geunsoo Song, Hoeseok Yang, and Youngmin Yi. 2026. oFFN: Outlier and Neuron-aware Structured FFN for Fast yet Accurate LLM Inference. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 1301–1315.

[62] 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*. 590–606.

[63] Fuwen Tan, Royson Lee, Łukasz Dudziak, Shell Xu Hu, Sourav Bhattacharya, Timothy Hospedales, Georgios Tzimiropoulos, and Brais Martinez. 2024. Mobilequant: Mobile-friendly quantization for on-device language models. In *Findings of the Association for Computational Linguistics: EMNLP 2024*. 9761–9771.

[64] Karan Taneja, Pratyusha Maiti, Sandeep Kakar, Pranav Guruprasad, Sanjeev Rao, and Ashok K Goel. 2024. Jill Watson: A Virtual Teaching Assistant powered by ChatGPT. In *International Conference on Artificial Intelligence in Education*. Springer, 324–337.

[65] Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, et al. 2024. Gemma 2: Improving open language models at a practical size. *arXiv preprint arXiv:2408.00118* (2024).

[66] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 10–19.

[67] Minh Duc Vu, Han Wang, Zhuang Li, Jieshan Chen, Shengdong Zhao, Zhenchang Xing, and Chunyang Chen. 2024. GPTVoiceTasker: LLM-powered virtual assistant for smartphone. *arXiv preprint arXiv:2401.14268* (2024).

[68] Jinheng Wang, Hansong Zhou, Ting Song, Shijie Cao, Yan Xia, Ting Cao, Jianyu Wei, Shuming Ma, Hongyu Wang, and Furu Wei. 2025. Bitnet. cpp: Efficient edge inference for ternary llms. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 9305–9322.

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

[70] Rongxiang Wang, Zhiming Xu, and Felix Xiaozhu Lin. 2024. WhisperFlow: speech foundation models in real time. *arXiv preprint arXiv:2412.11272* (2024).

[71] Tuowei Wang, Ruwen Fan, Minxing Huang, Zixu Hao, Kun Li, Ting Cao, Youyou Lu, Yaoxue Zhang, and Ju Ren. 2024. Ripple: Accelerating LLM Inference on Smartphones with Correlation-Aware Neuron Management. *arXiv preprint arXiv:2410.19274* (2024).

[72] Tuowei Wang, Ruwen Fan, Minxing Huang, Zixu Hao, Kun Li, Ting Cao, Youyou Lu, Yaoxue Zhang, and Ju Ren. 2025. Neuralink: Fast on-Device LLM Inference with Neuron Co-Activation Linking. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 147–162.

[73] Yuke Wang, Boyuan Feng, and Yufei Ding. 2022. QGTC: accelerating quantized graph neural networks via GPU tensor core. In *Proceedings of the 27th ACM SIGPLAN symposium on principles and practice of parallel programming*. 107–119.

[74] Zhaode Wang, Jingbang Yang, Xinyu Qian, Shiwen Xing, Xiaotang Jiang, Chengfei Lv, and Shengyu Zhang. 2024. MNN-LLM: A Generic Inference Engine for Fast Large Language Model Deployment on Mobile Devices. In *Proceedings of the 6th ACM International Conference on Multimedia in Asia Workshops*. 1–7.

[75] Jianyu Wei, Shijie Cao, Ting Cao, Lingxiao Ma, Lei Wang, Yanyong Zhang, and Mao Yang. 2025. T-mac: Cpu renaissance via table lookup for low-bit llm deployment on edge. In *Proceedings of the Twentieth European Conference on Computer Systems*. 278–292.

[76] Jianyu Wei, Qingtao Li, Shijie Cao, Lingxiao Ma, Zixu Hao, Yanyong Zhang, Xiaoyan Hu, and Ting Cao. 2025. T-MAN: Enabling End-to-End Low-Bit LLM Inference on NPUs via Unified Table Lookup. *arXiv preprint arXiv:2511.11248* (2025).

[77] 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*.

[78] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2023. Efficient streaming language models with attention sinks. *arXiv preprint arXiv:2309.17453* (2023).

[79] Daliang Xu, Hao Zhang, Liming Yang, Ruiqi Liu, Gang Huang, Mengwei Xu, and Xuanzhe Liu. 2024. Empowering 1000 tokens/second on-device llm prefilling with mllm-npu. *arXiv preprint arXiv:2407.05858* (2024).

[80] Daliang Xu, Hao Zhang, Liming Yang, Ruiqi Liu, Gang Huang, Mengwei Xu, and Xuanzhe Liu. 2025. Fast On-device LLM Inference with NPUs. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 445–462.

[81] Zhenliang Xue, Yixin Song, Zeyu Mi, Xinrui Zheng, Yubin Xia, and Haibo Chen. 2024. Powerinfer-2: Fast large language model inference on a smartphone. *arXiv preprint arXiv:2406.06282* (2024).

[82] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388* (2025).

[83] Hyunho Yeo, Chan Ju Chong, Youngmok Jung, Juncheol Ye, and Dongsu Han. 2020. Nemo: enabling neural-enhanced video streaming on commodity mobile devices. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*. 1–14.

[84] Chong Yu, Tao Chen, and Zhongxue Gan. 2023. Boost transformer-based language models with GPU-friendly sparsity and quantization. In *Findings of the Association for Computational Linguistics: ACL 2023*. 218–235.

[85] Li Lyna Zhang, Shihao Han, Jianyu Wei, Ningxin Zheng, Ting Cao, Yuqing Yang, and Yunxin Liu. 2021. nn-Meter: towards accurate latency prediction of deep-learning model inference on diverse edge devices. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*. 81–93.

[86] Peiyuan Zhang, Guangtao Zeng, Tianduo Wang, and Wei Lu. 2024. Tinyllama: An open-source small language model. *arXiv preprint arXiv:2401.02385* (2024).

[87] Zhenyu Zhang, Shiwei Liu, Runjin Chen, Bhavya Kailkhura, Beidi Chen, and Zhangyang Wang. 2024. Q-hitter: A better token oracle for efficient llm inference via sparse-quantized kv cache. *Proceedings of Machine Learning and Systems* 6 (2024), 381–394.

[88] Yilong Zhao, Chien-Yu Lin, Kan Zhu, Zihao Ye, Lequn Chen, Size Zheng, Luis Ceze, Arvind Krishnamurthy, Tianqi Chen, and Baris Kasikci. 2024. Atom: Low-bit quantization for efficient and accurate llm serving. *Proceedings of Machine Learning and Systems* 6 (2024), 196–209.

[89] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, and Koushik Sen. 2020. Ansor: Generating {High-Performance} tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*. 863–879.

[90] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. 2024. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems* 37 (2024), 62557–62583.