

T-SAR: A Full-Stack Co-design for CPU-Only Ternary LLM Inference via In-Place SIMD ALU Reorganization

Hyunwoo Oh, KyungIn Nam, Rajat Bhattacharjya, Hanning Chen, Tamoghno Das, Sanggeon Yun,
 Suyeon Jang, Andrew Ding, Nikil Dutt, and Mohsen Imani
 Department of Computer Science, University of California, Irvine
 Email: {hyunwoo, m.imani}@uci.edu

Abstract—Recent advances in LLMs have outpaced the computational and memory capacities of edge platforms that primarily employ CPUs, thereby challenging efficient and scalable deployment. While ternary quantization enables significant resource savings, existing CPU solutions rely heavily on memory-based lookup tables (LUTs) which limit scalability, and FPGA or GPU accelerators remain impractical for edge use. This paper presents T-SAR, the first framework to achieve scalable ternary LLM inference on CPUs by repurposing the SIMD register file for dynamic, in-register LUT generation with minimal hardware modifications. T-SAR eliminates memory bottlenecks and maximizes data-level parallelism, delivering $5.6\text{--}24.5\times$ and $1.1\text{--}86.2\times$ improvements in GEMM latency and GEMV throughput, respectively, with only 3.2% power and 1.4% area overheads in SIMD units. T-SAR achieves up to $2.5\text{--}4.9\times$ the energy efficiency of an NVIDIA Jetson AGX Orin, establishing a practical approach for efficient LLM inference on edge platforms.

Index Terms—Large Language Models, SIMD, Instruction Set Architecture, Quantization, Ternary LLM.

I. INTRODUCTION

Large Language Models (LLMs) have become ubiquitous today, with numerous applications, including those in coding assistants, document analysis, and interactive conversational interfaces across consumer and enterprise systems [1]. However, LLMs require substantial resources for inference, with billions of parameters driving extensive matrix operations and creating significant latency during autoregressive generation, where each token requires a full model forward pass [1].

Traditionally, LLMs have been deployed on cloud servers with high-power GPUs, NPUs, or specialized accelerators to handle their extreme compute and memory demands (e.g., >140 GB memory for Llama2-70B [2]). Increasingly, however, there is a need to run LLMs directly on the edge for scenarios such as coding copilots with proprietary source code, document analysis of confidential business data, and personalized assistants handling sensitive information [3]. In these settings, reliance on cloud computing is often infeasible due to intellectual property concerns, data privacy regulations like GDPR and HIPAA [4], limited network connectivity, or prohibitive cloud costs for continuous inference workloads.

Thus, to enable standalone LLM deployment on edge devices at lower cost, several techniques have been proposed, including pruning [5], [6], quantization [7]–[10], knowledge distillation [11], [12], and weight binarization [13]–[15].

Within quantization, ternary quantization has emerged as a particularly promising approach [16]–[18]. By constraining

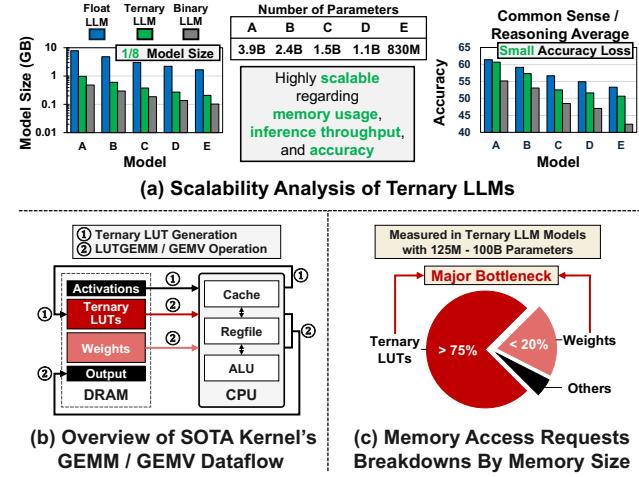


Fig. 1: Motivation for scalable ternary LLM acceleration. (a) Ternary LLMs provide $8\times$ size reduction with minimal accuracy loss, making them suitable for edge deployment. (b) GEMM/GEMV dataflow: SOTA LUT-based kernels store TLUTs in DRAM, causing frequent memory access requests. (c) Memory access breakdown: TLUTs dominate system memory requests—over 75%—across models from 125M to 100B parameters, creating a major bottleneck for CPU inference.

weights to $\{-1, 0, 1\}$, it achieves $8\times$ memory compression (Fig. 1(a)) while maintaining 93–99% of full-precision accuracy, presenting even better model-size scaling than binary LLMs [18]. This enables dramatic cost reduction and practical deployment on resource-constrained edge devices.

However, traditional (edge-) CPUs face a fundamental architectural misalignment when ternary LLMs are deployed, resulting in degraded performance. State-of-the-art (SOTA) methods such as T-MAC [19] and BitNet.cpp [20] replace multiply-accumulate (MAC) operations with dynamic lookup tables (LUTs) stored in memory (caches and DRAM), shifting workloads from compute-bound to memory-bound ones, achieving 2.4–6.2 \times the throughput compared to the FP16-based kernels [20]. As shown in Fig. 1(c), ternary LUT (TLUT) accesses account for over 75% of system memory requests, creating bandwidth pressure that underutilizes Single-Instruction Multiple-Data (SIMD) execution units ubiquitous in commodity processors [21]. This excessive memory traffic cancels out much of the computational benefit of ternary quantization.

While dedicated AI accelerators and FPGAs [22]–[24] achieve high ternary inference efficiency, their cost and integration complexity preclude widespread edge deployment. Likewise, server-class features such as Intel AMX [25] or

custom ISA extensions with large compute arrays [26]–[28] remain unavailable on edge CPUs due to area and power constraints. In contrast, the SIMD units that current approaches underutilize already provide high-bandwidth register files with datapaths naturally aligned to ternary operations. Yet no prior work has exploited these SIMD registers for in-register LUT computation, leaving an opportunity to overcome memory bottlenecks and fully harness existing parallel hardware.

Therefore, in this paper we introduce T-SAR, a full-stack co-design framework for scalable, high-throughput ternary LLM inference on edge CPUs, achieved by leveraging existing SIMD hardware with only minimal modifications. **The core innovation of T-SAR is repurposing the SIMD vector register file for dynamic, in-register LUT generation—eliminating costly memory traffic and maximizing data-level parallelism without new compute arrays or complex datapath extensions.** While we focus on the x86 AVX2 ISA, the idea extends naturally to other SIMD ISAs (e.g., ARM NEON, RISC-V Vector [29]), requiring only parameter retuning. T-SAR’s design spans four tightly integrated layers:

- **Algorithmic Layer:** A ternary-to-binary decomposition and data packing scheme for efficient LUT computing.
- **ISA Layer:** Minimal extensions that add register-to-register LUT-based General Matrix Multiplication (GEMM)/General Matrix-Vector Multiplication (GEMV), supporting dynamic computation within SIMD units.
- **Microarchitecture Layer:** Power and area overhead analyses based on lightweight wiring/multiplexing adjustments for SIMD units, validated by ASIC synthesis.
- **Software Layer:** An adaptive kernel dataflow that maximizes throughput across diverse models and platforms.

From high-performance to low-power edge CPUs, T-SAR achieves $5.6\text{--}24.5\times$ GEMM latency reduction and $1.1\text{--}86.2\times$ GEMV throughput improvement over SOTA CPU baselines [19], [20] on ternary LLMs from 125M to 100B parameters, with only 3.2% power and 1.4% area overhead in SIMD units. Notably, T-SAR also delivers $2.5\text{--}4.9\times$ higher energy efficiency than the Jetson AGX Orin GPU on Llama-8B [17] and Falcon3-10B [30]. These results demonstrate that systematic co-design can unlock the latent potential of ubiquitous SIMD hardware for practical edge LLM deployment, narrowing the gap with specialized accelerators while leveraging the world’s most widely deployed compute platform: CPUs.

II. MOTIVATION

We now characterize the bottleneck in SOTA ternary kernels that motivates the co-design framework T-SAR.

While ternary LLMs promise efficient inference by quantizing weights to $\{-1, 0, 1\}$ and replacing costly floating-point multiplications with low-cost integer operations, the SOTA approach for executing them introduces a critical bottleneck. As shown in Fig. 2(a,b), these models are implemented in ‘BitLinear’ layers, where the core matrix multiplication is handled by an **LUT-based GEMM/GEMV** method [19], [20]. This technique partitions each input vector (with K total inputs) into atomic blocks of size c . For each block, all 3^c

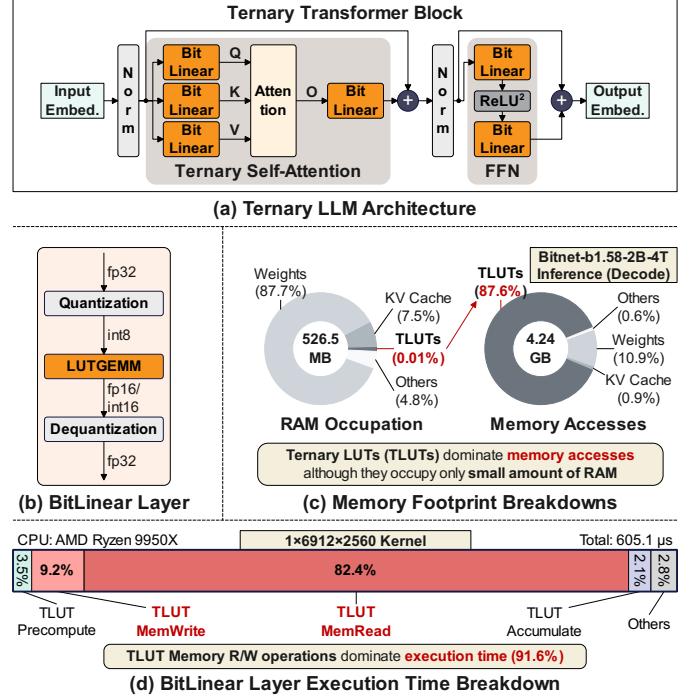


Fig. 2: **Ternary LLMs: Architecture and bottleneck analysis.** (a) Ternary transformer with BitLinear layers. (b) BitLinear layer workflow including quantization and LUTGEMM. (c) BitNet-b1.58-2B-4T memory footprints: TLUTs, though tiny in RAM, dominate memory accesses. (d) BitLinear GEMV time breakdown: Memory R/W dominates execution.

possible dot product results are pre-computed and stored in an LUT, transforming the computation from $O(K)$ arithmetic operations per output to $O(K/c)$ table lookups. The total LUT storage per layer becomes $O((K/c) \cdot 3^c)$, that trades off arithmetic complexity for memory accesses, resulting in the fastest CPU implementations today.

Though theoretically efficient, this trade-off introduces a critical **memory access bottleneck**. As Fig. 3(a) illustrates, the SOTA dataflow relies on pre-computing all LUT values and storing them in system memory. During inference, weights are used to index these LUTs, requiring frequent, random memory accesses that fail to efficiently utilize modern cache hierarchies and powerful SIMD hardware. Our analysis reveals the severity of this issue. Although ternary LUTs (TLUTs) occupy less than 0.01% of total RAM in a representative model, they are accessed so frequently that they account for a staggering **87.6% of all memory transactions** (Fig. 2(c)). This memory saturation means the workload becomes **memory-bound rather than compute-bound**, with **91.6% of the execution time** spent on memory read/write (R/W) operations (Fig. 2(d)).

To overcome these limitations, T-SAR aims to **eliminate external LUT loads and fully exploit the SIMD datapath**, shown in Fig. 3(b). T-SAR generates compressed LUTs on-the-fly inside SIMD registers via custom ISA extensions, eliminating memory TLUT traffic and supporting fused GEMV-accumulation operations to increase throughput. However, achieving in-register LUT computation is non-trivial: the required LUT size (3^c) does not match the fixed 2^n bitwidth of SIMD registers, and the limited register file size constrains the

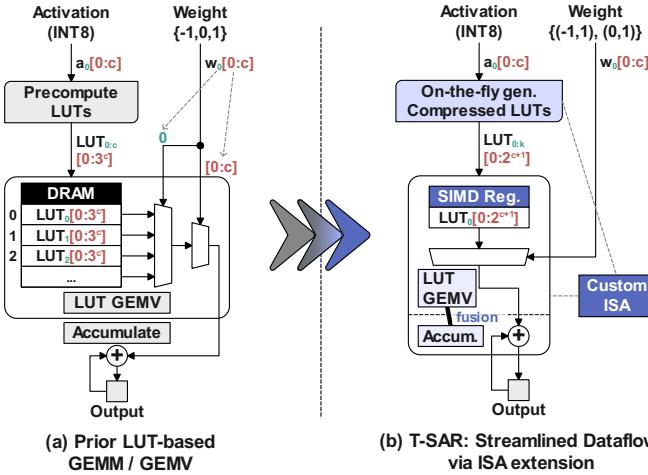


Fig. 3: **Prior LUT-based CPU solution vs. T-SAR.** (a) Prior: precomputed LUTs loaded from DRAM. (b) T-SAR: on-the-fly compressed LUTs generated in SIMD registers.

number of LUTs that are held simultaneously. This necessitates the careful co-design detailed in the following sections.

III. T-SAR Co-DESIGN STACK

To resolve the memory bottleneck mentioned in Section II, T-SAR introduces a full-stack co-design that transforms LUT-based operations from memory-bound to compute-bound tasks. The core innovation is that we eliminate memory traffic by generating LUTs on-the-fly, directly within the CPU’s high-speed SIMD register file. This section details the tightly integrated algorithmic, ISA, and microarchitectural layers that enable this transformation.

A. Algorithm: Ternary-to-Binary Decomposition

The primary challenge of in-register LUT generation is the architectural mismatch between the base-3 nature of ternary weights and the base-2 structure of SIMD hardware. A naive LUT for a block of c weights would require 3^c entries, which does not align with SIMD register bitwidths.

T-SAR overcomes this with a novel **LUT compression via weight transformation**, as shown in Fig. 4. We decompose a ternary weight block $\mathbf{w} \in \{-1, 0, 1\}^c$ and its corresponding input activations $\mathbf{a} \in \mathbb{R}^c$ into two separate binary forms:

- **Dense weights:** $\mathbf{w}_D \in \{-1, +1\}^c$, where $w_{D,i} = w_i$ if $w_i \neq 0$, else $+1$.
- **Sparse weights:** $\mathbf{w}_S \in \{0, 1\}^c$, where $w_{S,i} = 1$ if $w_i = 0$, else 0.

This allows the original dot product to be re-expressed as a subtraction of two binary dot products, removing the influence of the zero-weighted elements:

$$y = \sum_{i=1}^c w_i a_i = \sum_{i=1}^c w_{D,i} a_i - \sum_{i=1}^c w_{S,i} a_i$$

With this transformation, instead of one ternary LUT with 3^c entries, we only need two binary LUTs (for \mathbf{w}_D and \mathbf{w}_S), each of size 2^c . The total storage per block becomes 2^{c+1} , which perfectly matches the power-of-two width of SIMD registers and avoids significant data-path augmentation.

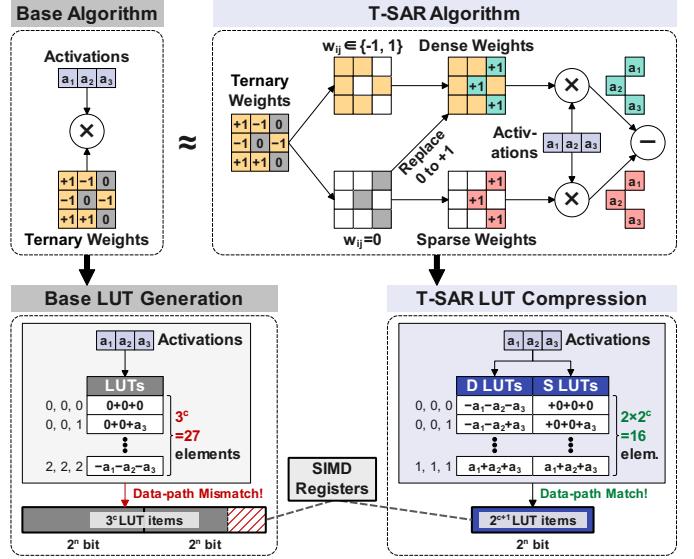


Fig. 4: **Proposed LUT GEMV Algorithm for LUT compression, matching the LUT size to the data-path.**

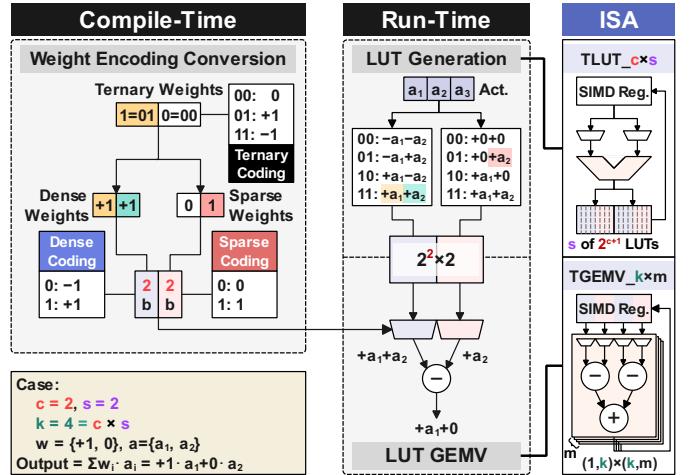


Fig. 5: **T-SAR's LUT-based kernel framework overview.**

B. ISA Extensions for In-Register Execution

As shown in Fig. 5, the decomposition mentioned above enables a two-phase kernel framework. At compile time, ternary weights are encoded into dense and sparse binary forms. At run time, a minimal set of ISA extensions execute the LUT-based GEMV directly within SIMD registers. These instructions are parameterized by:

- c : the block size.
- s : the number of input blocks processed per instruction.
- k : the number of input channels processed per instruction ($k = c \times s$).
- m : the number of output channels.

The workflow is: (1) the `TLUT_cxs` instruction generates two binary LUTs from activations and places them in SIMD registers; (2) the `TGEMV_kxm` uses these register-resident LUTs with pre-encoded weights to perform the $(1, k) \times (k, m)$ GEMV; and (3) the final ternary result is reconstructed by

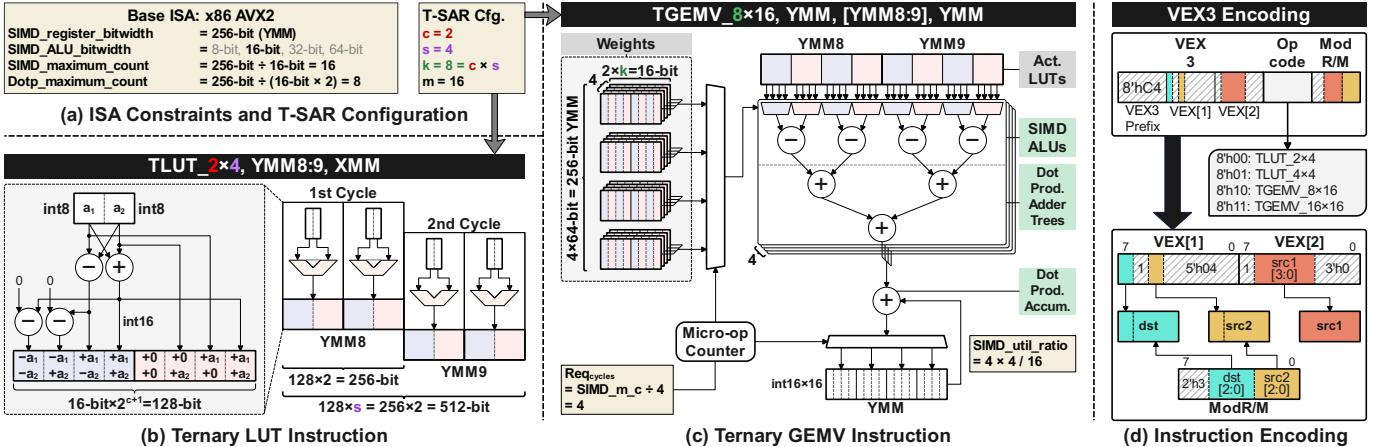


Fig. 6: T-SAR’s ISA extension applied to x86 AVX2 SIMD ISA, demonstrating how the T-SAR instruction primitives are realized with only minimal hardware changes, utilizing existing SIMD ALUs and adder trees. (a) T-SAR configuration and base ISA constraints. (b) TLUT_{c×s} example. (c) TGEMV_{k×m} example. (d) Instruction encoding details for AVX2, with designed examples of TLUT_{c×s} and TGEMV_{k×m} instructions.

subtraction. This register-resident design aligns computation with the SIMD datapath, eliminating the memory bottleneck.

C. Microarchitectural Implementation

Fig. 6 demonstrates a practical ISA extension realization on the x86 AVX2 ISA. For the example configuration shown in Fig. 6(a), we set $c=2$, $s=4$, $k=8$, and $m=16$.

The TLUT_{2×4} instruction (Fig. 6(b)) generates four register-resident LUTs, each with $2^{c+1} = 8$ 16-bit entries, occupying two 256-bit YMM registers [31] ($4 \times 8 \times 16 = 512$ bits). To minimize hardware changes, the operation is split into two μ -ops, each writing 256 bits per cycle.

The TGEMV_{8×16} instruction (Fig. 6(c)) performs a $(1, 8) \times (8, 16)$ GEMV, producing 16 outputs. It involves $s \times m = 64$ subtractions and $m = 16$ s-to-1 adder tree (ADT) operations, distributed over four μ -ops. These instructions reuse the existing 256-bit YMM datapath, including all 16 16-bit SIMD ALUs and 4-to-1 ADTs originally for dot product instructions. Only minor wiring and multiplexer additions are required, keeping area and power overhead minimal.

The instruction encoding, detailed in Fig. 6(d), uses standard VEX3 fields for both TLUT and TGEMV primitives. For instructions that span multiple registers (e.g., TLUT_{2×4} writing to YMM8:9 or TGEMV_{8×16} reading from YMM8:9), the destination or source register is interpreted as a register pair: if dst is 0x1000, the operation uses YMM8 and YMM9.

D. Software Kernel and Dataflow Optimization

The performance of GEMM/GEMV operations is highly dependent on data reuse patterns, which vary across different layers of an LLM. To maximize throughput, T-SAR employs an adaptive kernel scheduling strategy. We implement two microkernel dataflows—*activation-persistent* (AP) and *output-persistent* (OP)—to flexibly match these patterns (Fig. 7).

The AP dataflow (Fig. 7(a)) retains input activations in registers across the inner loop, minimizing LUT recomputation and increasing input and weight cache hits. This is effective for layers with high activation and weight reuse (i.e., high N

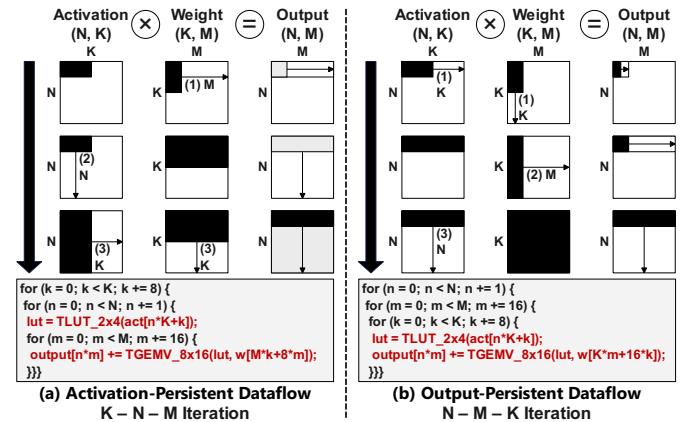


Fig. 7: The T-SAR’s kernel dataflow selections. (a) Activation-persistent dataflow minimizing the TLUT_{c×s} invocations and increases input/weight cache hits. (b) Output-persistent dataflow reducing total memory footprints.

and K). In contrast, the OP dataflow (Fig. 7(b)) keeps output accumulators local until computation completes, reducing memory write-back traffic. This is beneficial in layers with a high number of output channels (high M). At compile-time, T-SAR’s inference framework empirically selects the fastest kernel for each layer, ensuring maximum performance across the entire model.

IV. EXPERIMENTS AND EVALUATION

We now evaluate T-SAR across diverse models and platforms to validate three key claims: (1) T-SAR delivers significant end-to-end speedups for both prefill (GEMM-heavy) and decode (GEMV-heavy) phases in autoregressive LLMs; (2) these improvements arise from fundamentally reducing the memory bottleneck of SOTA LUT-based methods; and (3) they are achieved with minimal hardware overhead, making T-SAR highly efficient even compared to edge GPUs.

A. Experimental Setup

ISA and Simulator: We extend gem5-AVX 20.1.0.0 [32], [33] (DerivO3CPU) to model the T-SAR ISA. New

TABLE I: gem5 simulator configurations for evaluation platforms.

System Type	CPU Model	Simulation Mode	Cores	Freq.	L1 I/D Cache	L2 Cache	L3 Cache	DRAM
Workstation	AMD Ryzen 9950X		16	5.7 GHz	32 KB / 48 KB	1 MB/core	64 MB shared	DDR5-6400 MHz
Laptop	AMD Ryzen 7840U	DerivO3CPU	8	5.1 GHz	32 KB / 32 KB	1 MB/core	16 MB shared	DDR5-4400 MHz
Mobile	Intel Processor N250		4	3.8 GHz	64 KB / 32 KB	2 MB shared	6 MB shared	DDR5-4400 MHz

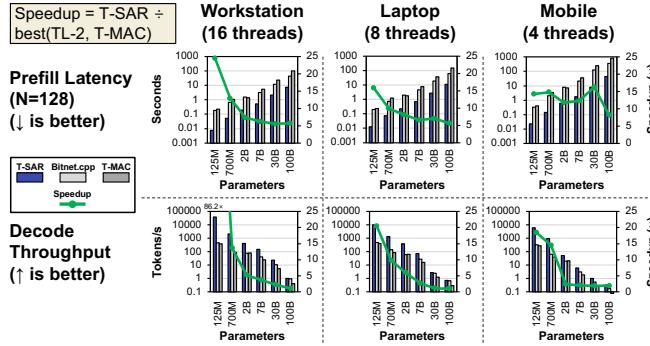


Fig. 8: End-to-end performance across platforms.

TLUT_c × s and TGEMV_k × m operations are added to the AVX2 pipeline with cycle-accurate μ -op sequencing and register-pair reads/writes. Each instruction was verified by executing hand-written assembly with byte-pattern encodings.

Kernels: We design three kernel variants for each LUT-GEMV pair (TLUT_2 × 4 + TGEMV_8 × 16, TLUT_4 × 4 + TGEMV_16 × 16), resulting in six kernels: (1) AP-min: activation-persistent with minimal register usage; (2) AP-max: activation-persistent with maximal register use to reduce iterations; (3) OP: output-persistent for minimal write-back traffic. All are implemented in C++ with inline assembly and compiled with GCC 9.4.0.

Baselines: We compare against two SOTA LUT-based baselines: Bitnet.cpp TL-2 [20] and T-MAC [19]. To ensure fairness, our kernels include both input quantization and output dequantization stages (shown in Fig. 2(b)).

Models and Protocol: We evaluate BitNet models from 125M to 100B parameters [34]. Prefill runs with $N = 128$ tokens (batch=1) to build the KV cache; decode measures steady-state throughput using the KV cache. Thread counts are fixed at {16, 8, 4} for {Workstation, Laptop, Mobile}.

Metrics: We report prefill latency, decode throughput, kernel execution time, and kernel memory access requests.

Platforms: Three representative x86 CPU classes are modeled (Table I): **Workstation** (Ryzen 9950X, 16 cores), **Laptop** (Ryzen 7840U, 8 cores), **Mobile** (Intel N250, 4 cores).

B. End-to-End Results

Prefill (GEMM-heavy): As shown in Fig. 8 (top), T-SAR delivers geo-mean prefill speedups of 8.8x (Workstation), 8.4x (Laptop), and 12.4x (Mobile) across all models (125M–100B). Since GEMM is compute-bound, register-resident LUT generation and fused accumulation let efficiency gains translate almost directly into throughput until cache/memory contention emerges. This explains why prefill benefits exceed decode gains. For example, Mobile’s 7B prefill drops from >20 s to under 1.7 s—enabling interactive LLM use on devices where GPUs cannot be deployed.

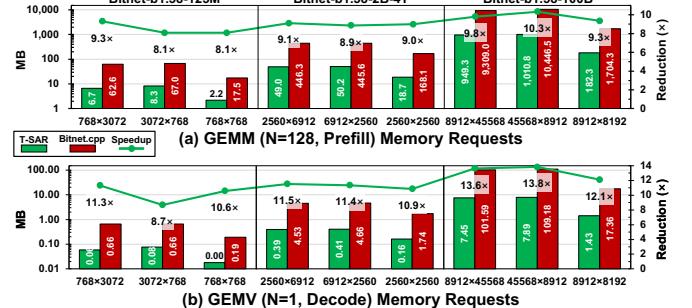


Fig. 9: Memory request volume (MB) of the kernels executed in the representative BitNet model inference (125M, 2B-4T, 100B). (a) GEMM (N=128, Prefill). (b) GEMV (N=1, Decode). Memory Requests

Decode (GEMV-heavy): Fig. 8 (bottom) shows that baseline GEMV is dominated by repeated TLUT fetches. T-SAR removes this traffic entirely, exposing SIMD compute throughput. Relative gains peak on Workstation (6.4x), due to larger caches delaying memory-system bandwidth saturation; Laptop and Mobile reach 4.1–4.2x. Mobile’s smaller gain reflects earlier bandwidth saturation despite the request-volume reduction.

Link to bottlenecks: The prefill/decode gap mirrors GEMM’s compute-bound and GEMV’s bandwidth-bound nature—setting up the trends seen in memory-system analysis.

C. Memory-System Impact

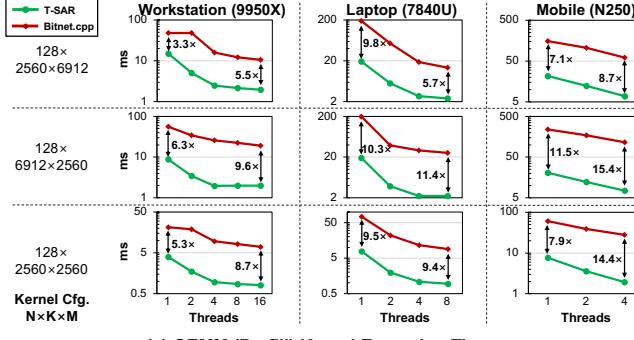
The central claim of T-SAR is that it removes the memory bottleneck by generating LUTs directly in registers. As shown in Fig. 9, this reduces memory request volume (MB) by 8.7–13.8x compared to TL-2 [20], with GEMV showing larger relative cuts because the baseline is TLUT-dominated and TLUT fetches are eliminated. For GEMM, TL-2’s denser weight packing (1.67 bits/weight) limits relative reductions, but the resulting stall decreases still improve ALU utilization.

Request reduction grows with model size ($K \times M$) as more LUT calls are avoided, but latency gains diverge: (1) **GEMV case-** Large cuts yield smaller returns as lower Last-level cache (LLC) hit rate reduces effective bandwidth and forces early saturation—most evident in Mobile 1 × 8192 × 45568: 89% → 62%—capping latency drops. (2) **GEMM case-** Even modest cuts yield large drops since compute-bound phases convert freed cycles and higher locality into utilization; LLC hit rate stays high (89% → 91%) until contention. These effects predict the thread-scaling behavior shown in Fig. 10.

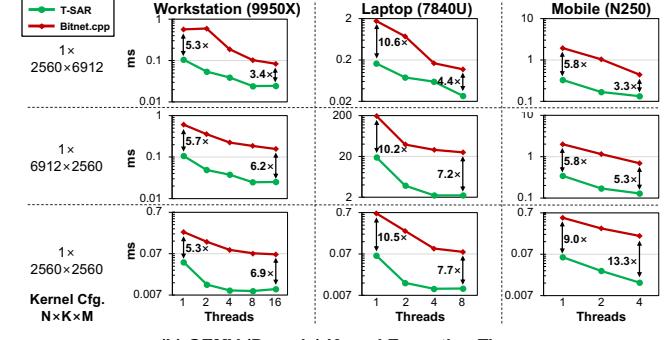
D. Kernel Microbenchmarks and Thread Scaling

From Fig. 10, we make the following observations:

GEMM case: For large shapes (128 × 2560 × 6912, 128 × 6912 × 2560), T-SAR sustains scaling up to 8–16 threads (Workstation) and 4–8 (Laptop) before L3/DRAM contention dominates. Scaling is not perfectly linear, but flattens later



(a) GEMM (Prefill) Kernel Execution Times



(b) GEMV (Decode) Kernel Execution Times

Fig. 10: Multi-thread scaling for BitNet-b1.58-2B-4T. T-SAR vs. TL-2 for GEMM (left) and GEMV (right). Solid lines denote absolute latency (log scale). Arrows show relative speedup.

than GEMV due to the compute-bound nature and higher data locality, allowing freed cycles from reduced stalls to be fully exploited. This yields up to $13\times$ speedup at 4 threads—matching the large prefill gains in Fig. 8.

GEMV case: Despite the largest proportional memory savings, GEMV saturates effective bandwidth quickly—often by 2–4 threads on Mobile and 4–8 on Workstation/Laptop—leading to early plateaus and smaller decode-time improvements. Here, extra cores cannot offset bandwidth limits.

Hence, we see that compute-bound kernels (GEMM) sustain scaling and achieve larger absolute gains across platforms, while memory-bound kernels (GEMV) plateau quickly—highlighting T-SAR’s impact and limits for real-world deployment for edge platforms.¹

E. Hardware Overheads

To size the cost of T-SAR, we synthesized a 256-bit SIMD unit (vector add/mul/dot-product, write-back interface) at 1 GHz in TSMC 28 nm using Cadence Genus 21.10, both *without* and *with* the T-SAR logic (shown in Table II). Multi-mode multi-corner (MMMC) synthesis covered $ssg \leftrightarrow ffg$, $V_{DD} \in [0.81, 1.05]$ V, $T \in [0, 125]$ °C; Area and power are reported at $t\tau_0p_9v25c$. The T-SAR instructions reuse existing ALU lanes and register file—no new arithmetic units or scratchpads. Additions are: (i) a 256-bit vector *write-back* MUX to inject TLUT words into the register file, (ii) small operand-bus wires and input MUXes for TLUT/TGEMV (no extra read ports), and (iii) a tiny control/scoreboard block to sequence TLUT writes and fused accumulation.

Result: As shown in Table II, the area increases by **+1.4%** ($73,560 \rightarrow 74,590 \mu\text{m}^2$) and active power consumption under kernel-like switching rises by **+3.2%** ($5,904 \rightarrow 6,090 \text{ mW}$), dominated by toggling on the new MUX paths. Results correspond to a single 256-bit SIMD slice (no SRAM arrays)

¹Note: We report prefill with $N=128$ due to simulation cost. TL-2’s weight packing (1.67-bit) is denser than our 1+1-bit split, resulting in approximately 20% more static memory occupation, but end-to-end is dominated by TLUT traffic rather than weight RAM size, hence T-SAR’s advantage. We demonstrate our framework to x86 due to the broad applicability, but retargeting to NEON or RISC-V Vector (RVV) [29] only requires c, s, k, m tuning due to the different SIMD lane width but extant dot product extensions. For instance, existing **ARM NEON’s 128-bit datapath** with SDOT/UDOT instruction support (since ARMv8.2-A [35]) realizes the **TLUT_2x4 + TGEMV_8x8**.

TABLE II: Synthesis of a 256-bit SIMD slice (TSMC 28 nm, 1 GHz) with and without T-SAR ISA.

Block	Area (μm^2)			Power (mW)		
	Base	T-SAR	Δ	Base	T-SAR	Δ
SIMD ALUs + write-back interface	73,560	73,560	0.0%	5,904	5,904	0.0%
T-SAR \rightarrow write-back MUX	0	588	+0.8%	0	41	+0.7%
Operand-bus wires and input MUX	0	147	+0.2%	0	24	+0.4%
Others (control/scoreboard, decode)	0	295	+0.4%	0	121	+2.0%
Total	73,560	74,590	+1.4%	5,904	6,090	+3.2%

and exclude caches and register files; absolute area will scale with the number of slices and integration.

F. Cross-Platform Comparison

To further evaluate our CPU-based solution, we compare it directly against an edge GPU in the NVIDIA Jetson AGX Orin [36] SoC, using identical model checkpoints and runtime settings (batch=1; steady-state decode). For CPUs, T-SAR power is estimated from CPU *package* power under TL-2 decode via the measured dynamic overhead in Table. II: $P_{\text{T-SAR}} = 1.032 \cdot P_{\text{TL-2}}$. Energy per token is $E = P_{\text{T-SAR}} / (\text{tokens/s})$ from the measured T-SAR throughput from gem5 simulator.

TABLE III: Cross-platform decode throughput and energy/token (batch=1). Power boundary: CPU package; GPU module.

Platform	Llama-b1.58-8B		Falcon3-b1.58-10B	
	tokens/s	J/token	tokens/s	J/token
Workstation CPU (9950X, 4nm, T-SAR)	128.96	0.616	103.93	0.795
Laptop CPU (7840U, 4nm, T-SAR)	61.00	0.405	49.65	0.540
Mobile CPU (N250, 10nm, T-SAR)	5.18	0.733	4.30	0.953
Jetson AGX Orin GPU (8nm, llama.cpp)	16.78	1.839	13.25	2.620

Takeaways. With matched per-model checkpoints and settings, T-SAR on CPUs outperforms Jetson on **Workstation** and **Laptop** across both families: Llama-b1.58-8B shows $7.7 \times / 3.0 \times$ (tokens/s / lower J/token) on workstation and $3.6 \times / 4.5 \times$ on laptop; Falcon3-b1.58-10B shows $7.8 \times / 3.3 \times$ and $3.7 \times / 4.9 \times$, respectively. On **Mobile**, throughput is lower than Jetson ($0.31\text{--}0.32 \times$), yet energy/token remains $2.5\text{--}2.75 \times$ lower, consistent with our decode memory-bandwidth analysis.

V. CONCLUSION

We presented T-SAR, a full-stack co-design framework for CPU-only ternary LLM inference. The key idea is to move LUT generation from memory into SIMD registers, turning TLUT fetches into in-register compute and fusing

accumulation so BitLinear layers shift from bandwidth-bound to datapath-bound execution. This yields portable speedups across CPUs with no new ALUs and only minor mux/control logic, supported by AP/OP kernels that adapt per layer to each model and platform. Beyond the reported gains, the approach generalizes to RVV and NEON [29], invites advanced LUT compression and compiler scheduling, and integrates naturally with sparsity or further quantization. In short, a small ISA extension realigns the bottleneck and enables interactive LLM inference on CPUs—even mobile ones—while keeping hardware changes minimal.

REFERENCES

- [1] Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. Large language models: A survey. *arXiv preprint arXiv:2402.06196*, 2024.
- [2] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open Foundation and Fine-Tuned Chat Models. *arXiv preprint arXiv:2307.09288*, 2023.
- [3] Yue Zheng, Yuhao Chen, Bin Qian, Xiufang Shi, Yuanchao Shu, and Jiming Chen. A review on edge large language models: Design, execution, and applications. *ACM Comput. Surv.*, 57(8), March 2025.
- [4] Kiarash Ahi. Risks & Benefits of LLMs & GenAI for Platform Integrity, Healthcare Diagnostics, Cybersecurity, Privacy & AI Safety: A Comprehensive Survey, Roadmap & Implementation Blueprint. *arXiv preprint arXiv:2506.12088*, 2025.
- [5] Xinyin Ma, Gongfan Fang, and Xinchao Wang. LLM-Pruner: On the Structural Pruning of Large Language Models. In *Adv. Neural Inf. Process. Syst. 36 (NeurIPS)*, pages 21702–21720, New Orleans, LA, USA, 2023.
- [6] Qichen Fu, Minsik Cho, Thomas Merth, Sachin Mehta, Mohammad Rastegari, and Mahyar Najibi. LazyLLM: Dynamic Token Pruning for Efficient Long Context LLM Inference. *arXiv preprint arXiv:2407.14057*, 2024.
- [7] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. AWQ: Activation-aware weight quantization for on-device LLM compression and acceleration. In *Annu. Conf. Mach. Learn. Syst. 6 (MLSys)*, volume 6, pages 87–100, Santa Clara, CA, USA, 2024.
- [8] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. GPT3.int8(): 8-Bit Matrix Multiplication for Transformers at Scale. In *Adv. Neural Inf. Process. Syst. 35 (NeurIPS)*, pages 30318–30332, New Orleans, LA, USA, 2022.
- [9] Yuzhuang Xu, Xu Han, Zonghan Yang, Shuo Wang, Qingfu Zhu, Zhiyuan Liu, Weidong Liu, and Wanxiang Che. Onebit: Towards extremely low-bit large language models. In *Adv. Neural Inf. Process. Syst. 37 (NeurIPS)*, pages 66357–66382, Vancouver, BC, Canada, December 2024.
- [10] Pedro Palacios, Rafael Medina, Giovanni Ansaloni, and David Atienza. HEEPstor: an Open-Hardware Co-design Framework for Quantized Machine Learning at the Edge. In *22nd ACM Int. Conf. Comput. Front.: Workshops Spec. Sess. (CF)*, page 22–25, Cagliari, Italy, 2025.
- [11] Achintya Kundu, Fabian Lim, Aaron Chew, Laura Wynter, Penny Chong, and Rhui Dih Lee. Efficiently Distilling LLMs for Edge Applications. *arXiv preprint arXiv:2404.01353*, 2024.
- [12] Yiming Ma, Chaoyao Shen, Linfeng Jiang, Tao Xu, and Meng Zhang. TKD: An Efficient Deep Learning Compiler with Cross-Device Knowledge Distillation. In *Des. Autom. Test Eur. (DATE)*, pages 1–7, Lyon, France, 2025.
- [13] Haoli Bai, Wei Zhang, Lu Hou, Lifeng Shang, Jin Jin, Xin Jiang, Qun Liu, Michael Lyu, and Irwin King. BinaryBERT: Pushing the Limit of BERT Quantization. In *Jt. Conf. 59th Annu. Meet. Assoc. Comput. Linguist. 11th Int. Jt. Conf. Nat. Lang. Process. (ACL-IJCNLP)*, pages 4334–4348, Virtual, August 2021.
- [14] Haotong Qin, Yifu Ding, Mingyuan Zhang, Qinghua YAN, Aishan Liu, Qingqing Dang, Ziwei Liu, and Xianglong Liu. BiBERT: Accurate Fully Binarized BERT. In *Int. Conf. Learn. Represent. (ICLR)*, Virtual, April 2022.
- [15] Zhihang Yuan, Yuzhang Shang, and Zhen Dong. PB-LLM: Partially binarized large language models. In *Int. Conf. Learn. Represent. (ICLR)*, Vienna, Austria, May 2024.
- [16] Hongyu Wang, Shuming Ma, Li Dong, Shaohan Huang, Huajie Wang, Lingxiao Ma, Fan Yang, Ruiping Wang, Yi Wu, and Furu Wei. BitNet: Scaling 1-bit Transformers for Large Language Models. *arXiv preprint arXiv:2310.11453*, 2023.
- [17] Shuming Ma, Hongyu Wang, Lingxiao Ma, Lei Wang, Wenhui Wang, Shaohan Huang, Li Dong, Ruiping Wang, Jilong Xue, and Furu Wei. The Era of 1-Bit LLMs: All Large Language Models Are in 1.58 Bits. *arXiv preprint arXiv:2402.17764*, 2024.
- [18] Ayush Kaushal, Tejas Vaidhya, Arnab Kumar Mondal, Tejas Pandey, Aaryan Bhagat, and Irina Rish. Surprising Effectiveness of Pretraining Ternary Language Model at Scale. In *Int. Conf. Learn. Represent. (ICLR)*, pages 1–48, 2025.
- [19] Jianyu Wei, Shijie Cao, Ting Cao, Lingxiao Ma, Lei Wang, Yanyong Zhang, and Mao Yang. T-MAC: CPU Renaissance via Table Lookup for Low-Bit LLM Deployment on Edge. In *21th European Conference on Computer Systems (EuroSys)*, pages 278–292, Rotterdam, Netherlands, March 2025.
- [20] Jinheng Wang, Hansong Zhou, Ting Song, Shijie Cao, Yan Xia, Ting Cao, Jianyu Wei, Shuming Ma, Hongyu Wang, and Furu Wei. Bitnet.cpp: Efficient Edge Inference for Ternary LLMs. *arXiv preprint arXiv:2502.11880*, 2025.
- [21] Diana Vuță-Popescu, Ionuț Cătălin Antofă, Cătălin Bogdan Ciobanu, and Csaba Zoltán Kertész. SIMD extensions—a historical perspective. In *2024 IEEE 30th International Symposium for Design and Technology in Electronic Packaging (SIITME)*, pages 108–115. IEEE, 2024.
- [22] Chenyang Yin, Zhenyu Bai, Pranav Venkatram, Shivam Aggarwal, Zhaoying Li, and Tulika Mitra. TerEffic: Highly Efficient Ternary LLM Inference on FPGA. *arXiv preprint arXiv:2502.16473*, 2025.
- [23] Ye Qiao, Zhiheng Chen, Yifan Zhang, Yian Wang, and Sitoa Huang. TeLLMe: An Energy-Efficient Ternary LLM Accelerator for Prefilling and Decoding on Edge FPGAs. *arXiv preprint arXiv:2504.16266*, 2025.
- [24] Ruiqi Chen, Jiayu Liu, Shidi Tang, Yang Liu, Yanxiang Zhu, Ming Ling, and Bruno Da Silva. ATE-GCN: An FPGA-Based Graph Convolutional Network Accelerator with Asymmetrical Ternary Quantization. In *Des. Autom. Test Eur. (DATE)*, pages 1–6, Lyon, France, 2025.
- [25] Seonjin Na, Geonhwa Jeong, Byung Hoon Ahn, Aaron Jezghani, Jeffrey Young, Christopher J. Hughes, Tushar Krishna, and Hyeseon Kim. FlexInfer: Flexible LLM Inference with CPU Computations. In *Annu. Conf. Mach. Learn. Syst. 7 (MLSys)*, 2025.
- [26] Giorgos Armenakos, Alexis Maras, Sotirios Xydis, and Dimitrios Soudris. Mixed-Precision Neural Networks on RISC-V Cores: ISA Extensions for Multi-Pumped Soft SIMD Operations. In *IEEE/ACM Int. Conf. Comput. Aided Des. (ICCAD)*, pages 1–9, Newark, NJ, USA, October 2024.
- [27] Angelo Garofalo, Giuseppe Tagliavini, Francesco Conti, Davide Rossi, and Luca Benini. XpulpNN: Accelerating Quantized Neural Networks on RISC-V Processors Through ISA Extensions. In *Des. Autom. Test Eur. (DATE)*, pages 186–191, Virtual, March 2020.
- [28] Navaneeth Kunhi Purayil, Matteo Perotti, Tim Fischer, and Luca Benini. AraXL: A Physically Scalable, Ultra-Wide RISC-V Vector Processor Design for Fast and Efficient Computation on Long Vectors. In *Des. Autom. Test Eur. (DATE)*, pages 1–7, Lyon, France, 2025.
- [29] Ju-Hung Li, Jhih-Kuan Lin, Yung-Cheng Su, Chi-Wei Chu, Lai-Tak Kuok, Hung-Ming Lai, Chao-Lin Lee, and Jenq-Kuen Lee. SIMD Everywhere Optimization from ARM NEON to RISC-V Vector Extensions. *arXiv preprint arXiv:2309.16509*, 2023.
- [30] Falcon-LLM Team. The Falcon 3 Family of Open Models, December 2024. Available: <https://huggingface.co/blog/falcon3>.
- [31] Chris Lomont. Introduction to intel advanced vector extensions. *Intel white paper*, 23(23):1–21, 2011.
- [32] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 Simulator. *ACM SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.
- [33] Seungmin Lee, Youngsok Kim, Dukyun Nam, and Jong Kim. Gem5-AVX: Extension of the Gem5 Simulator to Support AVX Instruction Sets. *IEEE Access*, 12:20767–20778, 2024.
- [34] Jinheng Wang, Hansong Zhou, Ting Song, Shaoguang Mao, Shuming Ma, Hongyu Wang, Yan Xia, and Furu Wei. 1-bit AI Infra: Part 1.1,

- Fast and Lossless BitNet b1.58 Inference on CPUs. *arXiv preprint arXiv.2410.16144*, 2024.
- [35] Arm Ltd. A64 SIMD Vector Instructions. <https://developer.arm.com/documentation/100069/0609/A64-SIMD-Vector-Instructions>. Accessed: March 26, 2025.
- [36] Mark Barnell, Courtney Raymond, Steven Smile, Darrek Isereau, and Daniel Brown. Ultra Low-Power Deep Learning Applications at the Edge with Jetson Orin AGX Hardware. In *IEEE High Perform. Extreme Comput. Conf. (HPEC)*, pages 1–4, 2022.