Transformer Inference using MAD vs LUT Kernels: A Comparative Benchmark of MAD and LUT Kernels for Binary and Ternary Dot Products on CPU and Edge Platforms

Mustafa Batu Eren Responsible Prof.: Qing Wang Supervisor: Braden Refalo
EEMCS, Delft University of Technology

1. Background & Motivation

Transformer language models rely on dense floating-point matrix multiplications, expensive in compute and energy. Running them on resource-constrained hardware demands cheaper inference.

Extreme quantization compresses weights to very few bits:

- **BitNet** [1]: binary weights $\mathbf{w} \in \{-1, +1\}^n$ (1 bit/param).
- **BitNet b1.58** [2]: ternary weights $\mathbf{w} \in \{-1, 0, +1\}^n$ ($\approx$1.58 bits/param), matching full-precision Transformers in perplexity at a fraction of the cost.

These regimes **eliminate floating-point multiply-accumulate**: the dot product degenerates to sign manipulation and zero-masking. Two CPU strategies exploit this: **MAD** computes $\mathbf{w} \cdot \mathbf{x}$ directly; **LUT** precomputes partial sums into a table and replaces arithmetic with a lookup. Prior systems (T-MAC [3], Bitnet.cpp [4]) report end-to-end speedups but **confound** the comparison: their kernels differ simultaneously in data layout, format, table depth, and software stack.

Research Question

To what extent do LUT-based dot-product kernels provide a performance advantage over MAD-based kernels for binary and ternary Transformer inference across different hardware resource constraints?

2. Kernel Approaches: MAD vs. LUT

MAD: Direct Computation

Compute $\mathbf{w} \cdot \mathbf{x}$ directly with SIMD integer sign-manipulation: binary $\times$ becomes a sign flip, ternary zeros are masked. No tables; everything stays in registers.

$$\underbrace{\begin{bmatrix} 3 \\ -5 \end{bmatrix}}_{\mathbf{x}\ \text{int8}} \cdot \underbrace{\begin{bmatrix} +1 \\ -1 \end{bmatrix}}_{\mathbf{w}\ \{-1,+1\}} = \underbrace{(+1)\cdot 3}_{\text{keep}} + \underbrace{(-1)\cdot(-5)}_{\text{negate}} = 8$$

LUT: Lookup Table

Group weights into blocks of g elements; precompute all partial sums once per activation vector $\mathbf{x}$. Each block then indexes its table slice with a **single hardware shuffle** (`pshufb` on AVX2, `vqtbl` on NEON), replacing g ops with one lookup. Tables grow exponentially: K^g entries/block ($K=2$ binary, $K=3$ ternary).

Build table from $\mathbf{x} = [3, -5]$, $g = 2$ ($2^2 = 4$ entries):

$\mathbf{w}$ pattern	$T[\mathbf{w}]$
$[-1, -1]$	2
$[-1, +1]$	-8
$[+1, -1]$	**8**
$[+1, +1]$	-2

$\mathbf{w} = [+1, -1] \longrightarrow T[\mathbf{w}] = \mathbf{8}$

	MAD	LUT
Compute / block	$O(g)$ ops	$O(1)$ lookup
Memory	registers only	K^g entries / block
Cache pressure	none	grows with depth g

Depth swept over $g \in \{2, 3, 4\}$: larger g cuts lookups but grows the table exponentially. **Where the crossover lands is an empirical question.**

3. Benchmark Setup & Instruction Cost

We implemented eight C++ SIMD kernels: MAD (both plain and BitNet-adapted) and LUT, each at depths $g \in \{2, 3, 4\}$ in binary and ternary variants. Every kernel was validated against a scalar reference, then swept over $N \in \{1024, 2048, 4096\}$ under two timing modes: *KernelOnly*, where the table cost is amortized, and *Full*, which rebuilds the table for each matrix-vector multiplication. We ran the sweep on an x86 desktop (AVX2) and an ARM edge core (Pi 4B, NEON).

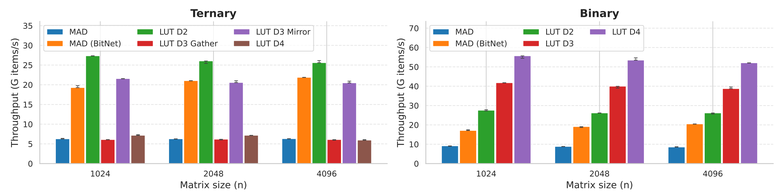
Larger tables force slower instructions

A single `pshufb` resolves a 16-entry byte lookup in **one cycle** — but only while a block's K^g table fits 16 entries. Binary tables stay small (2^g: $4 \to 8 \to 16$), so even depth-4 rides the fast shuffle. Ternary overruns the limit early (3^g: $9 \to 27 \to 81$): past 16 entries a kernel must re-pack the table to fit `pshufb` or fall back to the **5$\times$ slower** `vpgatherdd`, and by depth-4 only gather remains. **Table size picks the instruction; the instruction sets the speed.**

Instr.	Role	cyc/op	lat.
`pshufb`	LUT lookup, ≤ 16 entries	**1.0**	1
`vpgatherdd`	LUT lookup, > 16 entries	**5.0**	22
`vpmaddubsw`	MAD multiply–add	0.5	5
`vpaddd`	MAD accumulate	0.33	1

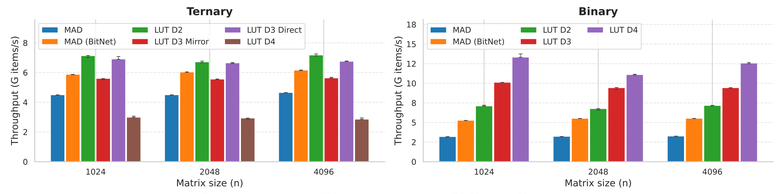
Table 1: AVX2 instruction cost on Comet Lake (uops.info); *cyc/op* is reciprocal throughput, lower is faster.

4. Throughput Results



Bars: median throughput. Whiskers: interquartile range (Q1-Q3) across 30 repetitions.

Figure 1: KernelOnly throughput on **x86 (AVX2)**, ternary (left) and binary (right).



Bars: median throughput. Whiskers: interquartile range (Q1-Q3) across 30 repetitions.

Figure 2: KernelOnly throughput on the **ARM edge** platform (Pi 4B, Cortex-A72, NEON).

The instruction story plays out as predicted: **binary throughput rises with depth, ternary peaks early**. On x86 the fastest binary kernel (LUT D4) hits **104.4 GOPS, 2.6$\times$ the strongest MAD**; ternary LUT D2 leads by only 1.2–1.4$\times$. On the narrow NEON edge core the expected depth ordering breaks: LUT D2 stays ahead of the deeper LUT D3 Direct even though D3 Direct issues fewer lookups, because the two-register `vqtbl2q` is $\approx 3\times$ costlier than single-register `vqtbl1q`.

5. Roofline Analysis

The **roofline model** [5] plots achieved performance (GOPS) against **arithmetic intensity**, the operations performed per byte of memory traffic. Diagonal **memory-bandwidth** slopes (DRAM, L2, L1) bound memory-limited kernels on the left; horizontal **hardware ceilings** bound compute-limited kernels from above. A kernel lying on a diagonal is *memory-bound*; one reaching a ceiling is *compute-bound*. Raising arithmetic intensity shifts a kernel rightward toward the compute roof; the "elbow" where slope meets ceiling marks the intensity needed to saturate compute.

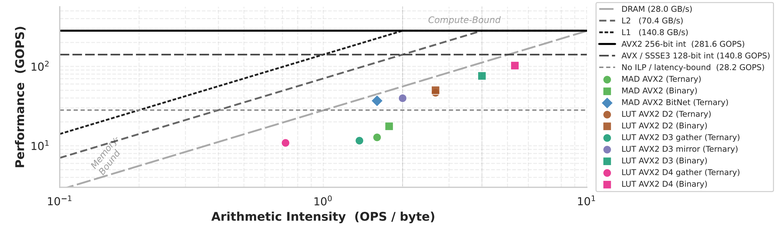


Figure 3: AVX2 roofline, all kernels ($N = 4096$). Diagonals are bandwidth ceilings; horizontals are compute ceilings.

Every kernel sits **far below the compute ceiling** (281.6 GOPS) *and* below its DRAM-bandwidth slope, so **none is bandwidth-bound**. Instead the binding limit is **instruction throughput**, not memory.

Key observations:

- **Depth cuts both ways**: for binary, nibble-packing shrinks the weight stream faster than the 2^g table grows, so deeper LUTs *gain* intensity and shift rightward; for ternary the 3^g table must be re-streamed and balloons faster than the operation count, so deeper LUTs *lose* intensity and drift leftward. Either way every kernel stays left of the elbow, so the shift never buys compute-bound speed.
- **The lookup instruction decides the winner**: kernels whose inner loop is a **single-cycle shuffle** on an L1-resident table (LUT D2, binary D3/D4) clear the latency line by the widest margin; gather-based ternary D3/D4 and unpack-heavy MAD kernels fall *below* it.
- **Cache pressure**: the table grows exponentially (2^g/3^g entries); past a critical depth it spills L1$\to$L2, adding latency that erodes the intensity gain.

6. Conclusion

The LUT advantage over MAD is **conditional, not uniform**, governed by:

- **Weight alphabet**: binary wins big (104.4 GOPS, 2.6$\times$ the strongest MAD); ternary narrow (1.2–1.4$\times$).
- **Table depth & cache tier**: deeper tables help only while they fit fast cache.
- **Lookup throughput**: single-cycle shuffle wins; gather/multi-register loses.

Optimal kernel selection is a **function of the deployment target**.

References

[1] H. Wang et al. *BitNet: Scaling 1-bit Transformers for LLMs.* arXiv:2310.11453, 2023.
[2] S. Ma et al. *The Era of 1-bit LLMs: All LLMs are in 1.58 Bits.* arXiv:2402.17764, 2024.
[3] J. Wei et al. *T-MAC: CPU Renaissance via Table Lookup for Low-bit LLM Deployment on Edge.* EuroSys, 2025.
[4] J. Wang et al. *Bitnet.cpp: Efficient Edge Inference for Ternary LLMs.* arXiv:2502.11880, 2025.
[5] S. Williams et al. *Roofline: An Insightful Visual Performance Model.* Commun. ACM 52(4), 2009.