Hardware costs of vector quantization

Semyon Savkin
semyon@mit.edu

Jose Ramos
jricramc@mit.edu

I. Introduction

When running large language models (LLMs), especially the ones which have reasoning capabilities, most compute resources are spent in generation phase. In this regime, the model performs the inference for generating one token, so the most time-consuming operation is matrix-vector multiplication. This workload has the property of being memory-bound, which means that that more resources are spent on loading the weight matrix from the DRAM than on doing the computation itself.

One common method of reducing the energy consumed by loading the weights is quantization. Quantization is a technique of reducing the precision of the data (in our case, model weights) to decrease the storage and reduce the memory bandwidth bottleneck. While quantizing the components of the model could lead to a degradation of performance, the LLMs can be compressed from 16 bits to 4 bits with small deterioration in evaluation metrics.

An important property of quantization is the efficiency of recovering the original values from the compressed representations. Most commonly used quantization methods for running LLM inference on GPUs utilize uniform scalar quantization. The uniform scalar quantization methods have the following quantization functions for given value bounds L and R and bit rate R:

$$\text{encoder: } Q_{ij} = \left\lfloor 2^R \cdot \frac{A_{ij} - L}{R - L} \right\rfloor$$

$$\text{decoder: } \hat{A}_{ij} = L + \frac{(R-L)Q_{ij}}{2^R}$$

The benefit of uniform quantization is that during inference we can operate solely on quantized representations, using the integer multipliers, which is good for speed. However, the tradeoff between bitwidth and accuracy for this method is far from optimal. The figure 5 displays the gap in MSE between an optimal scalar quantizers (i.e. the ones that quantize each weight independently into one of 2^R values), and an optimal rate distortion function, on Gaussian noise.

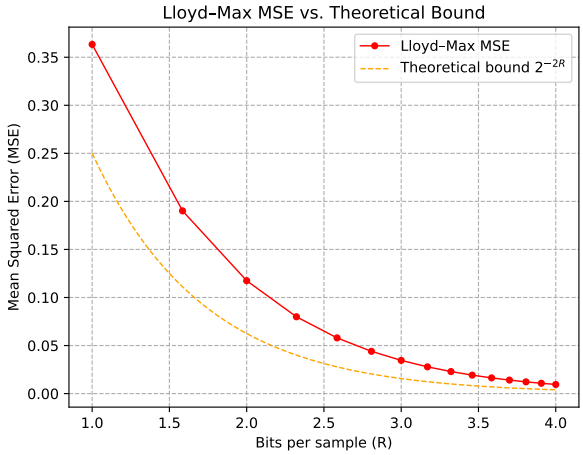


Fig. 1: Gaussian source rate distortion tradeoff

Practical quantizers that achieve better rate distortion function often split the data (weight matrix) into a sequence of low-dimensional vectors, and apply some quantization method on each vector independently. While using vector quantization (VQ) may offer better accuracy, loading models quantized with VQ on GPU requires running a costly dequantization procedure consisting of multiple instructions.

Designing custom accelerators for running neural networks opens the possibility of embedding a VQ dequantizer directly into the hardware. We model the dequantizer as a circuit with multiple pipeline stages, which takes a quantized representation as the input, and returns a vector of dequantized weights, which are consequently used in MACs. We show that the benefits of higher accuracy from better quantization outweigh the overhead in area and energy from having the dequantizer component.

In this work, we choose NestQuant [1] as the VQ method. It has state-of-the-art accuracy results in 4-bit weight quantization among low-dimensional VQ methods. In addition, its dequantization algorithm can be modified to have a very efficient hardware implementation.

Our paper is structured as follows. First, we give an overview of vector quantization methods and hardware simulations of effects of quantization in section II. Then, we have a brief overview of the NestQuant scheme in section A. In section III, we propose a circuit that implements VQ in hardware with small area and latency. Finally, in section IV we model the integration of our dequantizer into hardware using CiMLoop [2].

II. Related Work

LLM.INT8() [3] showed that 8-bit weights and activations are feasible if a handful of large-magnitude "outlier" channels are kept in FP16. Post-training methods have also shown promising results using 4–5 bits: GPTQ [4] uses second-order error compensation, SmoothQuant [5] rebalances per-channel scales, while AWQ [6] outlier-weights to adaptively mixed precisions. There have also been attempts at 3-bit or even 2-bit regimes by using blockwise Hessian information [7]. To avoid expensive floating-point non–linearities, I-BERT [8] reformulated GELU, Softmax and LayerNorm for exact INT8 arithmetic, enabling *fully-integer* Transformer inference. Collectively, these works establish that state-of-the-art LLMs can run at 4-bit weight precision with sub-percent perplexity loss, motivating a hardware pipeline in which int4 matrices feed integer MACs.

Scalar quantizers leave a large rate–distortion gap, especially below 4 bits. Cluster-based *vector quantization* (VQ) closes that gap by sharing representative centroids across weights. Deep Compression combined pruning, VQ and Huffman coding to compress CNNs by $35\times$ and inspired the *EIE* ASIC, which performed table look-ups on-chip to reconstruct weights on the fly [9]. Recent work revisits VQ for transformers: AQLM [10] encodes each weight as the sum of multiple codebook vectors, achieving 2–3 bits/weight with minimal perplexity loss, while Huff-LLM [11] integrates a Huffman decoder next to a systolic array, reducing weight traffic by 26% at $< 6\%$ area cost. NestQuant [1] employs *nested lattices* (E_8) for extremely efficient 4-bit VQ and provides bit-shift-only decoding logic.

To optimize for quantized data, several ASICs revisit the datapath organization. *OliVe* [12] encodes each "outlier–victim" pair inside a 4-bit group, allowing outlier handling *locally* with negligible control overhead and delivering $4.5\times$ faster inference than a GPU baseline. *OPAL* [13] mixes 3-/5-/8-bit activations and dedicates a small FP16 component for the high-magnitude values, attaining $> 2\times$ energy savings on GPT-J. On FPGAs, Byun *et al.* [7] constrain each row to two discrete bit-widths, enabling a dual-systolic architecture that stores all weights on-chip and realises up to $17\times$ energy efficiency over prior BERT accelerators.

Analog or digital CIM moves the dot-product into weight storage arrays, bypassing DRAM altogether. *ATT* [14] and *iMCAT* [15] exploit ReRAM crossbars to execute attention matrix multiplications in situ, yielding a $\mathcal{O}(100\times)$ improvement over GPUs. *H3D-Transformer* vertically integrates FeFET CIM tiles with digital tensor cores, resulting in 10 TOPS/W on GPT-2 [16]. These CIM designs operate at 4–8-bit precision, but suffer from device non-idealities and limited codebook flexibility. Our fully-digital approach provides a drop-in module for existing integer accelerators and can coexist with CIM back-ends by supplying int4 weights directly to mixed-signal MACs.

Rapid evaluation of mapping strategies and memory hierarchies is enabled by `Timeloop` and `Accelergy` [17]. Recent LLM accelerators such as Huff-LLM [11] and CiMLoop [2] adopt this toolchain to estimate cycle, energy and area metrics early in design. We follow the same methodology: the NestQuant dequantizer is modelled as a three-stage pipeline in `Timeloop`, and its access counters feed `Accelergy` to obtain post-layout energy projections. This ensures our results are directly comparable to prior art that reports `Timeloop` / `Accelergy` figures.

III. Dequantization circuit

In this section, we develop a pipelined efficient circuit for NestQuant dequantization. The dequantization consists of two stages: first, we need to perform a matrix multiplication between the generating matrix G and the coordinates. Then, we need to find the closest point in E_8 lattice scaled by q and subtract it. For a more detailed discussion on the algorithm, please refer to Appendix A.

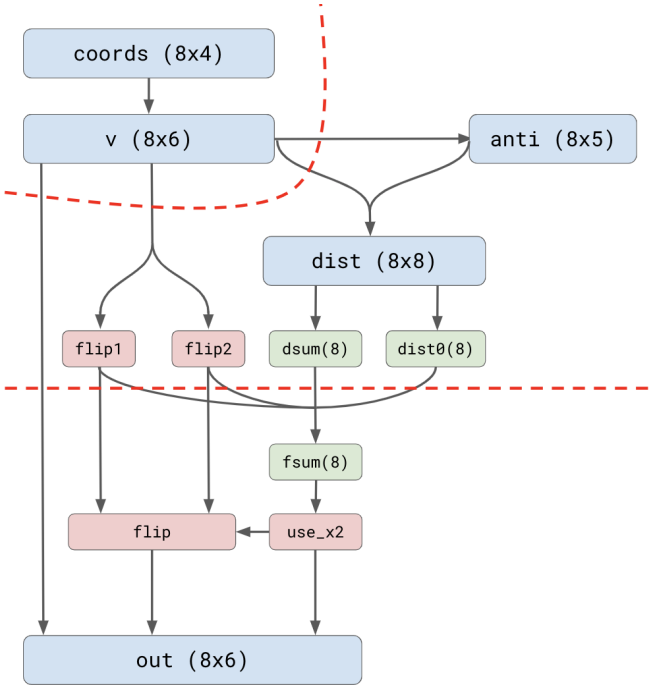
We make the circute for 4-bit quantization with $q = 16$. The matrix G is equal to the following:

$$G = \begin{pmatrix} 2 & 1 & 1 & 1 & 1 & 1 & 1 & \frac{1}{2} \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & \frac{1}{2} \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & \frac{1}{2} \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & \frac{1}{2} \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & \frac{1}{2} \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & \frac{1}{2} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} \end{pmatrix}$$

Instead of multiplying by G, we will multiply by $2G$ to keep having integer values, then we need to subtract the closest value in $32E_8$. Note that if $f(x) = x - \mathcal{V}_{32E_8}(x)$, then $f(x + 64t) = f(x)$ for all $t \in \mathbb{Z}^8$. So, it means that we can keep only 6 bits of matrix multiplication result, interpreting it as an unsigned number. We don't need to do any multiplications due to sparsity of the matrix, the fixed bitshifts for multiplying by 2 and 4, as well as 6-bit sums are sufficient.

Now, let's describe the circuit for computing $f(x)$. To round the point x to the closest element of $32E_8$, we consider two candidates: closest x_1 in $32D_8$ (recall that this set contains all integer 8-vectors with coordinates divisible by 32 such that their sum is divisible by 64) and closest x_2 in $32D_8 + 16$ (equal to $32D_8$ with every vector with 16 added to every coordinate). Then, the answer is x minus best candidate among x_1 and x_2 by Eucledian distance to x.

We describe the procedure for obtaining x_1. If we drop the requirement for sum of coordinates being divisible by 64, we simply have to round each coordinate to the closest value divisible by 32. There are two ways to round each number: floor and ceil. We should choose the best rounding for each component. However, if the sum of coordinates of optimal rounding fails the divisibility by 64, we need to "flip" one of the rounding directions. In the full algorithm, we need to choose the optimal component to flip, however, acoording to [1], always flipping the first component is close to optimal in the end-to-end quantization benchmarks, and we will use this approach.



(a) Circuit diagram of NestQuant dequantization circuit

use_x2	$v^i[4]$	flip	value	target	out	rep
0	0	0	p	0	p	$00p$
1	0	0	p	16	$p-16$	$11p$
0	1	0	$16+p$	32	$p-16$	$11p$
1	1	0	$16+p$	16	p	$00p$
0	0	1	p	32	$p-32$	$10p$
1	0	1	p	-16	$p+16$	$01p$
0	1	1	$16+p$	0	$p+16$	$01p$
1	1	1	$16+p$	48	$p-32$	$10p$

(b) 6-bit two complement representation of the result depending on the parameters. We denote 4 least significant bits of v^i as p

We denote x^i as i-th component of vector x and $y[k]$ as k-th least significant bit of y. Let L^i be x^i with last 5 bits set to 0. We denote $\tilde{x}_1$ and $\tilde{x}_2$ to be x_1 and x_2 before flipping rounding. Then, $\tilde{x}_1^i$ is equal to L^i or L^i+32 depending on $x^i[4]$ and $\tilde{x}_2^i$ is L^i+16. Let d_i be the absolute distance between x^i and L^i+16 the the MSE between $\tilde{x_1}$ and x is $\sum_i (16-d_i)^2$ and MSE between $\tilde{x_2}$ and x is $\sum_i d_i^2$.

$$\tilde{x_1} \text{ is better } \Leftrightarrow \sum d_i^2 \geq \sum (16-d_i)^2 \Leftrightarrow \sum d_i^2 - 256 + 32d_i - d_i^2 \geq 0 \Leftrightarrow \left(32\sum d_i\right) - 2048 \geq 0 \Leftrightarrow \sum d_i \geq 64$$

We set $\Delta = \sum d_i$, and we compare Δ with 64 to select the candidate. Note that we are choosing between x_1 and x_2, not between $\tilde{x_1}$ and $\tilde{x_2}$, so we need to update Δ accordingly. One can show that if we flip first coordinate of $\tilde{x_1}$, we need to subtract $2d_1$ and if we flip $\tilde{x_2}$, we add $32-2d_1$, and texe comparison between Δ and 64 will become the valid criterion.

Using this information, we design the dequantization circuit with the following variables:

- **coords:** The initial coordinates given as an input
- **v:** The coordinates of the point after matmul
- **anti:** i-th value is equal to $16 - v^i[3:0]$
- **d:** d_i is computed with a MUX between v^i and $anti^i$, based on $v^i[4]$, padded to 8 bits
- **dsum** is sum of d_i, **dist0** is d_1
- **flip1** and **flip2** correspond to whether rounding direction of first component in $\tilde{x}_1$ and $\tilde{x}_2$ needs to be flipped. One can show that $\text{flip}_2 = \bigoplus_i v^i[5]$, $\text{flip}_1 = \text{flip}_2 \oplus \bigoplus_i v^i[4]$.
- **fsum** is the updated Δ to account for flips
- **use_x2** is true when we choose x_2, otherwise we choose x_1. **flip** is true when the candidate we chose has a flip
- **out** is the final result. Table 2b show that 6-bit 2-complement representation of out^i is equal to the concatentation of $(\text{flip} \wedge (i=1)) \oplus \text{use\_x2} \oplus v^i[4]$, $\text{use\_x2} \oplus v^i[4]$, and $v^i[3:0]$.

The dashed red lines in circuit figure indicate the places where pipelines are separated. We have synthesized two versions of the circuit: with 3 pipeline stages and with 2 pipeline stages (without the second separtion). We have implement the circuit

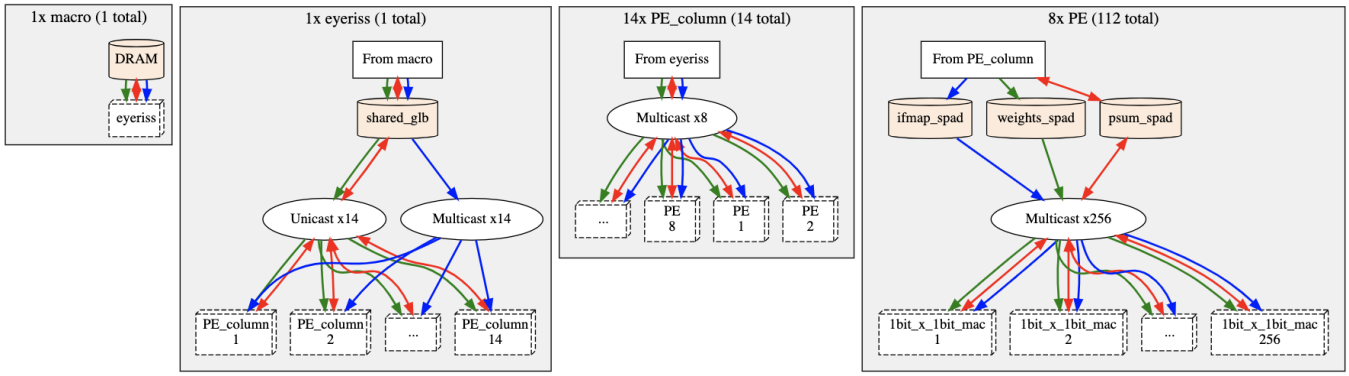


Fig. 3: The diagram of Eyeriss architecture

using Minispec. The synthesis results (area, critical path delay, and gate count are presented in Table I). The area is computed for 45nm technology.

Pipeline stages	Area	Delay	Gate count
2	1117.47 um^2	549.62 ps	903
3	1350.48 um^2	354.59 ps	963

TABLE I: Simulation results

IV. VQ ACCELERATOR DESIGN

We use CiMLoop to estimate the area and energy of running the workloads. First, we describe our estimation methods for area and energy of the quantization circuit. We will be using 65nm technology for our estimation, so to scale our area data, we will be multiply it by the squared ratio of technologies. To estimate the energy, we use the formula $\alpha C_{gate} V^2 G$, where C_{gate} is the capacitance of the gate (we estimate it as 2 fF), α is the rate of wire changes, $V = 1$ Volt. We use 3-pipelined version, and our estimations result in energy of $\sim$1 pJ and area $\sim$3000 um^2.

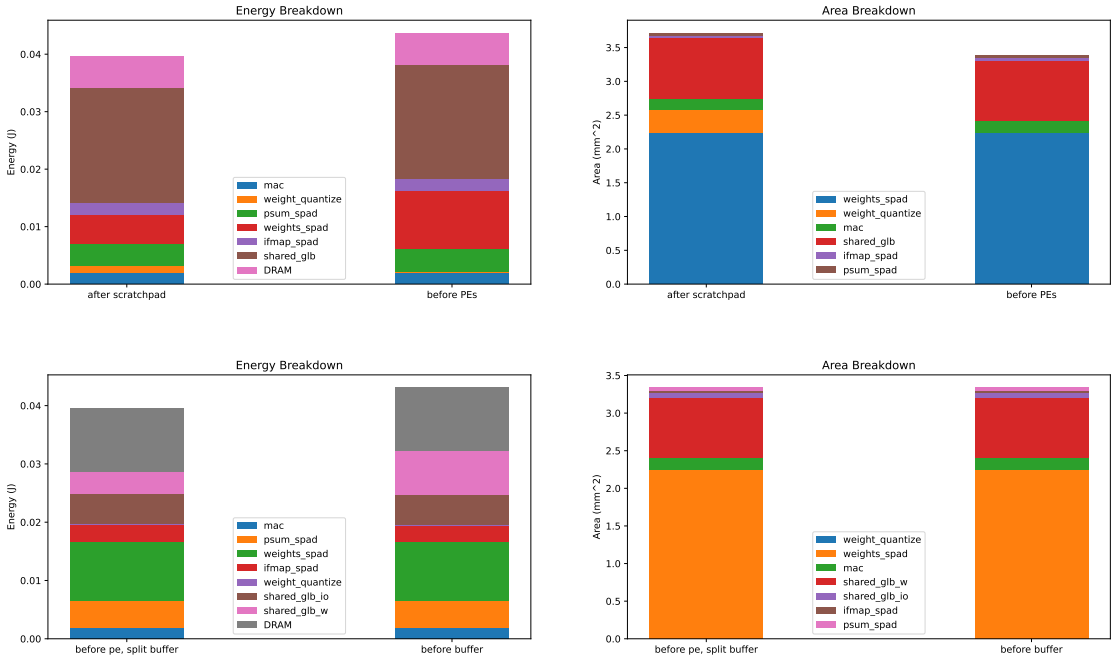


Fig. 4: Top: energy and area comparison for placing the dequantizer before PEs or after scratchpad with shared buffer. Bottom: energy and area comparison for placing the dequantizer before or after buffer with split buffer

We use Eyeriss [18] architecture as the basic accelerator. The architecture consists of the following components: DRAM, shared global buffer. Then, there is a 14 by 8 array of PEs. Each PE contains scratchpads for weights, activations, and outputs. Finally, each PE contains a MAC that is modeled by several virtual MACs. We assume that the dequatnized numbers are 8-bit, while the quantized numbers are 4-bit.

We model the dequantization component as a buffer with width 32, with the dequantization action attached to the read. There are three possible placements of the dequantizer: after the scratchpad, before the PEs, and before the buffer. We explore the tradeoffs of these options. Note that in order to model the dequantization before the shared global buffer, we would need to have different (relative) datawidth for weights and activations, which CiMLoop does not support. So, we will compare the placement of dequantizer before or after global buffer in the setup, where it is split into weight buffer and input/output buffer.

The figure 4 shows the evaluation results for area and energy of the accelerator with dequantizer on a problem of multiplying two 2048×2048 matrices. We can note that having the dequantizer closer to MAC makes its energy larger, because the dequantization would have to run larger number of times. In addition, we would have more instances of the dequantizer, so the area grows as well. However, the benefit is that the buffers that are higher in the memory hierarchy would be able to have lower number of bits, so the memory operations with those buffers would consume less energy.

We can note that even in the scenario where the the dequantizer is placed right before MAC, the energy used by dequantizer is quite negligible compared to the buffers. One way to see this is to notice that the dequantizer energy is comparable to the energy of MAC, but dequantizer can process eight elements instead of one, so its energy is bounded by energy of MAC, which is still much smaller than global buffer energy. However, having a dequantizer in each PE leads to a large area consumption. So, placing a dequantizer before PE saves area, but increases energy due to weight buffer having to store twice more bits.

Since the energy of dequantization is negligible compared to the savings from having the weight buffer to store two times less data, we should prefer putting the dequantizer after the weight buffer as compared to before the weight buffer.

In addition to evaluating on GEMM, we run all of our configurations on convolutional workloads, specifically ResNet. The energy and area distributions across various configs are in table II. The configs are: “w0” — dequantizer after scratchpaad, “w3” — dequantizer before PE, “w3s” — w3 config with split buffer, “w4” — dequantizer before the buffer. The convolution experiments exhibit similar patterns as GEMM.

V. Conclusion

Embedding a pipelined NestQuant dequantizer adds only $\sim$3 000 um^2 and 1 pJ per 8-word decode—negligible compared to DRAM/SRAM energy when we halve buffer bit-widths. Placing the dequantizer just after the weight buffer maximizes these savings without multiplying hardware instances. This shows that vector-quantization modules can deliver the accuracy of 4-bit VQ at minimal area/energy cost, unlocking more aggressive low-bit schemes in DNN accelerators.

	Workload	Config	DRAM (area)	DRAM (energy)	ifmap_spad (area)	ifmap_spad (energy)	mac (area)	mac (energy)	psum_spad (area)	psum_spad (energy)	shared_glb (area)	shared_glb (energy)	shared_glb_io (area)	shared_glb_io (energy)	shared_glb_w (area)	shared_glb_w (energy)	weight_quantize (area)	weight_quantize (energy)	weights_spad (area)	weights_spad (energy)	Total (area)	Total (energy)
0	GEMM	w0	nan	5.50e-03 (13.89%)	3.40e-08 (0.91%)	1.97e-03 (4.97%)	1.68e-07 (4.52%)	1.97e-03 (4.99%)	4.17e-08 (1.12%)	3.98e-03 (10.05%)	8.97e-07 (24.12%)	2.00e-02 (50.44%)	nan	nan	nan	nan	3.38e-07 (9.09%)	1.10e-03 (2.78%)	2.24e-06 (60.24%)	5.10e-03 (12.89%)	3.72e-06 (100.00%)	3.96e-02 (100.00%)
1	GEMM	w3	nan	5.50e-03 (12.57%)	3.40e-08 (1.00%)	1.97e-03 (4.50%)	1.68e-07 (4.97%)	1.97e-03 (4.52%)	4.17e-08 (1.23%)	3.98e-03 (9.10%)	8.97e-07 (26.50%)	2.00e-02 (45.66%)	nan	nan	nan	nan	3.02e-09 (0.09%)	1.38e-04 (0.31%)	2.24e-06 (66.20%)	1.02e-02 (23.33%)	3.38e-06 (100.00%)	4.37e-02 (100.00%)
2	GEMM	w3s	nan	1.09e-02 (27.49%)	3.40e-08 (1.02%)	2.81e-03 (7.12%)	1.68e-07 (5.03%)	1.97e-03 (5.00%)	4.17e-08 (1.25%)	4.51e-03 (11.41%)	nan	nan	5.51e-08 (1.65%)	5.28e-03 (13.36%)	7.99e-07 (23.92%)	3.74e-03 (9.46%)	3.02e-09 (0.09%)	1.38e-04 (0.35%)	2.24e-06 (67.04%)	1.02e-02 (25.82%)	3.34e-06 (100.00%)	3.95e-02 (100.00%)
3	GEMM	w4	nan	1.09e-02 (25.19%)	3.40e-08 (1.02%)	2.81e-03 (6.53%)	1.68e-07 (5.03%)	1.97e-03 (4.58%)	4.17e-08 (1.25%)	4.51e-03 (10.45%)	nan	nan	5.51e-08 (1.65%)	5.28e-03 (12.24%)	7.99e-07 (23.92%)	7.48e-03 (17.34%)	3.02e-09 (0.09%)	5.37e-07 (0.00%)	2.24e-06 (67.04%)	1.02e-02 (23.66%)	3.34e-06 (100.00%)	4.31e-02 (100.00%)
4	DNN	w0	nan	1.35e-03 (18.13%)	3.40e-08 (0.91%)	3.54e-04 (4.74%)	1.68e-07 (4.52%)	4.17e-04 (5.59%)	4.17e-08 (1.12%)	8.19e-04 (10.98%)	8.97e-07 (24.12%)	3.22e-03 (43.14%)	nan	nan	nan	nan	3.38e-07 (9.09%)	2.32e-04 (3.11%)	2.24e-06 (60.24%)	1.07e-03 (14.31%)	3.72e-06 (100.00%)	7.46e-03 (100.00%)
5	DNN	w3	nan	1.34e-03 (16.29%)	3.40e-08 (1.00%)	3.45e-04 (4.18%)	1.68e-07 (4.97%)	4.17e-04 (5.05%)	4.17e-08 (1.23%)	8.19e-04 (9.93%)	8.97e-07 (26.50%)	3.16e-03 (38.33%)	nan	nan	nan	nan	3.02e-09 (0.09%)	2.74e-05 (0.33%)	2.24e-06 (66.20%)	2.14e-03 (25.89%)	3.38e-06 (100.00%)	8.25e-03 (100.00%)
6	DNN	w3s	nan	3.19e-03 (30.27%)	3.40e-08 (1.02%)	4.37e-04 (4.15%)	1.68e-07 (5.03%)	4.17e-04 (3.96%)	4.17e-08 (1.25%)	9.84e-04 (9.35%)	nan	nan	5.51e-08 (1.65%)	1.41e-03 (13.40%)	7.99e-07 (23.92%)	1.56e-03 (14.84%)	3.02e-09 (0.09%)	5.62e-05 (0.53%)	2.24e-06 (67.04%)	2.47e-03 (23.50%)	3.34e-06 (100.00%)	1.05e-02 (100.00%)
7	DNN	w4	nan	3.06e-03 (27.41%)	3.40e-08 (1.02%)	4.37e-04 (3.91%)	1.68e-07 (5.03%)	4.17e-04 (3.73%)	4.17e-08 (1.25%)	9.91e-04 (8.87%)	nan	nan	5.51e-08 (1.65%)	1.10e-03 (9.84%)	7.99e-07 (23.92%)	2.77e-03 (24.78%)	3.02e-09 (0.09%)	1.72e-06 (0.02%)	2.24e-06 (67.04%)	2.40e-03 (21.45%)	3.34e-06 (100.00%)	1.12e-02 (100.00%)

	Workload	Config	DRAM (area)	DRAM (energy)	ifmap_spad (area)	ifmap_spad (energy)	mac (area)	mac (energy)	psum_spad (area)	psum_spad (energy)	shared_glb (area)	shared_glb (energy)	shared_glb_io (area)	shared_glb_io (energy)	shared_glb_w (area)	shared_glb_w (energy)	weight_quantize (area)	weight_quantize (energy)	weights_spad (area)	weights_spad (energy)	Total (area)	Total (energy)
0	GEMM	w0	0.00e+00 (0.00%)	1.34e-04 (77.67%)	3.40e-08 (0.91%)	9.60e-07 (0.56%)	1.68e-07 (4.52%)	9.64e-07 (0.56%)	4.17e-08 (1.12%)	1.69e-06 (0.98%)	8.97e-07 (24.12%)	2.92e-05 (16.90%)	nan	nan	nan	nan	3.38e-07 (9.09%)	5.37e-07 (0.31%)	2.24e-06 (60.24%)	5.24e-06 (3.03%)	3.72e-06 (100.00%)	1.73e-04 (100.00%)
1	GEMM	w3	0.00e+00 (0.00%)	1.34e-04 (75.38%)	3.40e-08 (1.00%)	9.60e-07 (0.54%)	1.68e-07 (4.97%)	9.64e-07 (0.54%)	4.17e-08 (1.23%)	1.69e-06 (0.95%)	8.97e-07 (26.50%)	2.92e-05 (16.40%)	nan	nan	nan	nan	3.02e-09 (0.09%)	5.37e-07 (0.30%)	2.24e-06 (66.20%)	1.05e-05 (5.89%)	3.38e-06 (100.00%)	1.78e-04 (100.00%)
2	GEMM	w3s	0.00e+00 (0.00%)	1.68e-04 (78.88%)	3.40e-08 (1.02%)	7.51e-07 (0.35%)	1.68e-07 (5.03%)	9.64e-07 (0.45%)	4.17e-08 (1.25%)	2.72e-06 (1.28%)	nan	nan	5.51e-08 (1.65%)	2.56e-06 (1.20%)	7.99e-07 (23.92%)	2.68e-05 (12.64%)	3.02e-09 (0.09%)	5.37e-07 (0.25%)	2.24e-06 (67.04%)	1.05e-05 (4.94%)	3.34e-06 (100.00%)	2.12e-04 (100.00%)
3	GEMM	w4	0.00e+00 (0.00%)	1.68e-04 (70.06%)	3.40e-08 (1.02%)	7.51e-07 (0.31%)	1.68e-07 (5.03%)	9.64e-07 (0.40%)	4.17e-08 (1.25%)	2.72e-06 (1.14%)	nan	nan	5.51e-08 (1.65%)	2.56e-06 (1.07%)	7.99e-07 (23.92%)	5.36e-05 (22.41%)	3.02e-09 (0.09%)	5.37e-07 (0.22%)	2.24e-06 (67.04%)	1.05e-05 (4.39%)	3.34e-06 (100.00%)	2.39e-04 (100.00%)
4	DNN	w0	0.00e+00 (0.00%)	4.91e-05 (7.04%)	3.40e-08 (0.91%)	3.02e-05 (4.32%)	1.68e-07 (4.52%)	2.66e-05 (3.81%)	4.17e-08 (1.12%)	5.00e-05 (7.16%)	8.97e-07 (24.12%)	4.26e-04 (61.05%)	nan	nan	nan	nan	3.38e-07 (9.09%)	1.48e-05 (2.12%)	2.24e-06 (60.24%)	1.01e-04 (14.50%)	3.72e-06 (100.00%)	6.98e-04 (100.00%)
5	DNN	w3	0.00e+00 (0.00%)	4.91e-05 (6.20%)	3.40e-08 (1.00%)	3.02e-05 (3.81%)	1.68e-07 (4.97%)	2.66e-05 (3.36%)	4.17e-08 (1.23%)	5.00e-05 (6.32%)	8.97e-07 (26.50%)	4.26e-04 (53.82%)	nan	nan	nan	nan	3.02e-09 (0.09%)	7.40e-06 (0.93%)	2.24e-06 (66.20%)	2.02e-04 (25.56%)	3.38e-06 (100.00%)	7.92e-04 (100.00%)
6	DNN	w3s	0.00e+00 (0.00%)	1.32e-04 (22.74%)	3.40e-08 (1.02%)	3.79e-05 (6.51%)	1.68e-07 (5.03%)	2.66e-05 (4.57%)	4.17e-08 (1.25%)	6.07e-05 (10.43%)	nan	nan	5.51e-08 (1.65%)	1.95e-04 (33.48%)	7.99e-07 (23.92%)	9.97e-06 (1.71%)	3.02e-09 (0.09%)	3.02e-07 (0.05%)	2.24e-06 (67.04%)	1.19e-04 (20.50%)	3.34e-06 (100.00%)	5.82e-04 (100.00%)
7	DNN	w4	0.00e+00 (0.00%)	1.02e-04 (22.73%)	3.40e-08 (1.02%)	3.79e-05 (8.48%)	1.68e-07 (5.03%)	2.66e-05 (5.95%)	4.17e-08 (1.25%)	6.07e-05 (13.58%)	nan	nan	5.51e-08 (1.65%)	6.57e-05 (14.71%)	7.99e-07 (23.92%)	3.23e-05 (7.22%)	3.02e-09 (0.09%)	7.55e-08 (0.02%)	2.24e-06 (67.04%)	1.22e-04 (27.30%)	3.34e-06 (100.00%)	4.47e-04 (100.00%)

TABLE II: The Timeloop results for GEMM and Resnet workloads

References

[1] S. Savkin, E. Porat, O. Ordentlich, and Y. Polyanskiy, “Nestquant: Nested lattice quantization for matrix products and llms,” *arXiv preprint arXiv:2502.09720*, 2025. [Online]. Available: https://arxiv.org/abs/2502.09720

[2] T. Andrulis, J. S. Emer, and V. Sze, "Cimloop: A flexible, accurate, and fast compute-in-memory modeling tool," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2024, pp. 10–23.
[3] T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer, "LLM.int8(): 8-bit matrix multiplication for transformers at scale," *arXiv preprint arXiv:2208.07339*, 2022. [Online]. Available: https://arxiv.org/abs/2208.07339
[4] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, "GPTQ: Accurate post-training quantization for generative pre-trained transformers," *arXiv preprint arXiv:2210.17323*, 2022. [Online]. Available: https://arxiv.org/abs/2210.17323
[5] G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth, and S. Han, "Smoothquant: Accurate and efficient post-training quantization for large language models," *arXiv preprint arXiv:2211.10438*, 2022. [Online]. Available: https://arxiv.org/abs/2211.10438
[6] J. Lin, J. Tang, H. Tang, S. Yang, W.-M. Chen, W.-C. Wang, G. Xiao, C. Gan, and S. Han, "AWQ: Activation-aware weight quantization for llm compression and acceleration," *arXiv preprint arXiv:2306.00978*, 2023. [Online]. Available: https://arxiv.org/abs/2306.00978
[7] W. Byun, J. Woo, and S. Mukhopadhyay, "Hardware-friendly hessian-driven row-wise quantization and fpga acceleration for transformer-based models," in *Proceedings of the 61st ACM/IEEE Design Automation Conference (DAC)*, 2024.
[8] S. Kim, A. Gholami, Z. Yao, M. W. Mahoney, and K. Keutzer, "I-bert: Integer-only bert quantization," in *International Conference on Machine Learning (ICML)*, 2021. [Online]. Available: https://arxiv.org/abs/2101.01321
[9] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," *ISCA*, 2016.
[10] V. Egiazarian, A. Panferov, D. Kuznedelev, E. Frantar, A. Babenko, and D. Alistarh, "Extreme compression of large language models via additive quantization," *arXiv preprint arXiv:2401.06118*, 2024. [Online]. Available: https://arxiv.org/abs/2401.06118
[11] A. Naga, C. Zhang, P. Gupta, Y.-J. Hsu, T. Hoefler, and D. Alistarh, "Huff-LLM: End-to-end lossless compression for efficient llm inference," *arXiv preprint arXiv:2502.00922*, 2025. [Online]. Available: https://arxiv.org/abs/2502.00922
[12] C. Guo, J. Tang, W. Hu, J. Leng, C. Zhang, F. Yang, Y. Liu, M. Guo, and Y. Zhu, "OliVe: Accelerating large language models via hardware-friendly outlier-victim pair quantization," in *Proceedings of the 50th International Symposium on Computer Architecture (ISCA)*, 2023.
[13] J. Koo, D. Park, S. Jung, and J. Kung, "OPAL: Outlier-preserved microscaling quantization accelerator for generative large language models," in *Design Automation Conference (DAC)*, 2024. [Online]. Available: https://arxiv.org/abs/2409.05902
[14] H. Guo, L. Peng, J. Zhang, Q. Chen, and T. D. LeCompte, "ATT: A fault-tolerant reram accelerator for attention-based neural networks," in *IEEE International Conference on Computer Design (ICCD)*, 2020, pp. 213–221.
[15] A. F. Laguna, A. Kazemi, M. Niemier, and X. S. Hu, "In-memory computing based accelerator for transformer networks for long sequences," in *Design, Automation and Test in Europe (DATE)*, 2021.
[16] Y. Luo and S. Yu, "H3d-transformer: A heterogeneous 3d (h3d) computing platform for transformer model acceleration on edge devices," *ACM Trans. Design Autom. Electr. Syst.*, 2024.
[17] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara *et al.*, "Timeloop: A systematic approach to dnn accelerator evaluation," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019, pp. 304–315.
[18] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.

APPENDIX A
NESTQUANT BACKGROUND

NestQuant is a lattice-based vector quantization method. We fix a lattice Λ in dimension d, which will be the dimension of our VQ. When quantizing a point $x \in \mathbb{R}^d$, we first find the closest (by Eucledian distance) to x lattice element $y \in \Lambda$, we denote it as $y = \mathcal{V}_\Lambda(x)$. Ideally, we would want to assign quantized bitstrings to all the elements of Λ so that we can always reconstruct y, but the number of lattice elements is infinite. So, we need to pick a subset of Λ to which we assign bitstrings.

Since the vectors we quantize usually lie within certain distance of the origin, one reasonable choice of assignment would be to explicitly enumerate all the lattice elements in a sphere around origin. However, then we won't be able to do dequantization without a huge lookup table. Instead, we use a nested lattice approach, which assigns bitstrings to points, which are closer to 0 than to any other element of scaled lattice. Specifically, let R be the bitrate, $q = 2^R$, then y is assigned a bitstring iff $\mathcal{V}_{q\Lambda}(y) = 0$.

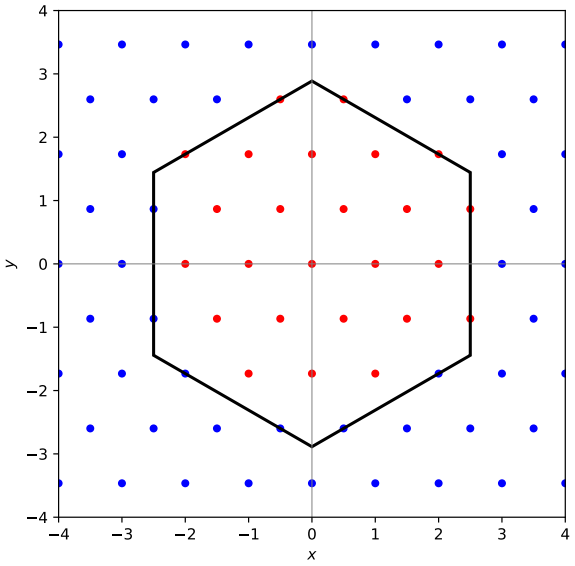


Fig. 5: Red dots are NestQuant codebook elements for hexagonal lattice and $q = 5$

We specify the encoder and decoder functions. Let G be the generator matrix of the lattice. Then, the encoder maps a point to the coordinates of closest lattice point modulo q: $x \to [G^{-1}\mathcal{V}_\Lambda(x)] \bmod q$. Then, the decoder reconstructs the point y from these coordinates c, and subtracts the closest point in $q\Lambda$, ($y = Gc$, $\hat{x} = y - \mathcal{V}_{q\Lambda}(y)$). ensuring that reconstructed value $\hat{x}$ satisfies $\mathcal{V}_{q\Lambda}(\hat{x}) = 0$. It can be proved that these encoder and decoder functions correctly implement the quantization algorithm specified above. In NestQuant, we choose Λ to be E_8 lattice. $E_8 = D_8 \cup D_8 + \frac{1}{2}$, where D_8 contains all vectors in $\mathbb{Z}^8$ with even coordinate sum.