QUANTIZATION BEYOND MATRIX MULTIPLICATION FOR EFFICIENT LLM INFERENCE

Anonymous authors
Paper under double-blind review

ABSTRACT

Quantization is widely used to accelerate LLMs inference, but existing approaches primarily optimize matrix multiplication while leaving surrounding operators and data movement in high precision. This limits end-to-end benefits of quantization, particularly for memory-bound operators. In this paper, we proposed an INT8 pipeline that extends quantization beyond matrix multiplication. Our design preserves intermediate tensors in INT8 across both matmul and non-matmul operators, reducing global-memory traffic. We further introduce a hierarchical warp–block reduction strategy that effectively utilizes on-chip resources and maintains high GPU occupancy during compute deep learning operators. We evaluate our approach on Qwen3 and Llama-3.2 models across multiple task and workloads. Compared with BF16 inference, our approach achieves up to a $1.6\times$ inference speedup and reduces energy consumption by up to 39%, while maintaining comparable accuracy. The benefits remain consistent across different GPUs, decoding workloads, and batch sizes, demonstrating that quantization beyond matmul is important for efficient end-to-end LLM inference.[1]

1 INTRODUCTION

Transformer-based large language models (LLMs) Vaswani et al. (2017); Brown et al. (2020) are increasingly deployed in interactive assistants and agentic systems, where they process user requests and generate responses. In these applications, inference is executed repeatedly during deployment and directly affects response latency, computational workload, memory traffic, and energy consumption Taherin et al. (2026). During inference, model weights and intermediate tensors are repeatedly transferred through the memory Shazeer (2019); Pope et al. (2023). As a result, end-to-end inference performance is determined not only by computation, but also by the cost of data movement and available memory bandwidth.

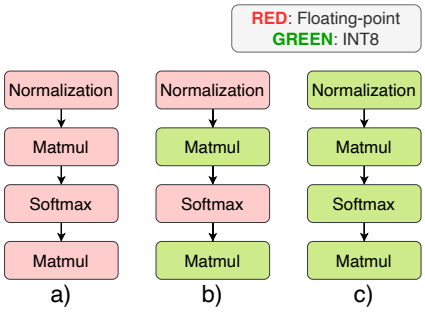


Figure 1: Comparison of tensor representations across Transformer operators. (a) Floating-point execution. (b) Matmul-centric INT8 quantization. (c) Our method preserves INT8 tensor across both matmul and non-matmul operators.

Quantization is widely used to reduce the memory usage and computational requirements of LLM inference by representing model tensors with fewer bits Gholami et al. (2022). Weight-only methods such as GPTQ Frantar et al. (2023) and AWQ Lin et al. (2024) store model weights in low-precision formats, while computation is typically performed with higher-precision. Weight–activation methods, such as LLM.int8() Dettmers et al. (2022), and SmoothQuant Xiao et al. (2023), further quantize activations to enable low-precision matrix multiplication (matmul). Most existing methods have concentrated on a numerical question: how can model accuracy be preserved under low-precision matmul? In this work, we ask a complementary systems question: once matmul operates in INT8, what happens to the other operators in the inference pipeline?

[1] We plan to release the source code upon publication.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

In existing quantized pipelines, intermediate tensors and surrounding operators, such as normalization, softmax, and activation functions, are typically processed in higher precision (Figure 1(b)). Consequently, although quantization reduces the computation associated with matmul, surrounding operators continue to read and write higher-precision tensors to global memory. This limitation affects both prefill and decoding. During prefill, matmul accounts for only 30%–50% of the latency in our profiling of LLMs (Section 2.2), leaving a substantial fraction of inference latency unchanged. During decoding, many operators are memory-bound (Section 2.1), so the benefit of INT8 arithmetic largely disappears and operators are primarily limited by memory bandwidth. Operator fusion, as in FlashAttention Dao et al. (2022) and graph compilers Ansel et al. (2024), addresses this problem by avoiding the materialization of intermediate tensors *inside* a fused region. However, tensors that outside fusion boundaries still travel in BF16. Our approach is complementary to operator fusion: storing these tensors in INT8 reduces their memory traffic regardless of how operators are fused.

We propose an INT8 inference pipeline, which maintains INT8 representations across matmul and non-matmul operators (Figure 1(c)). Each kernel reads INT8 inputs, dequantizes them on chip, evaluates the operator in higher precision using registers and on-chip memory, and requantizes the result before anything is written back. Higher-precision intermediate values remain on chip, and only quantized outputs are written to global memory; the method requires no retraining. For GEMM, we fuse requantization into the output stage of a CUTLASS INT8 kernel Thakkar et al. (2023); for GEMV, we use a dot-product kernel that keeps its INT32 accumulators in registers.

The main challenge lies in non-matmul operators. Following the taxonomy of Ivanov et al. (2021), we divide them into *reduction-based* operators, such as softmax and normalization, and *element-wise* operators. Reduction-based operators are particularly challenging because they require a statistic over the entire input vector before the output can be produced. The straightforward implementation materializes the dequantized floating-point vector in the GPU's on-chip *shared memory* while performing the reduction. However, this approach require substantial on-chip storage for higher-precision intermediate values, increasing shared-memory pressure and reducing GPU occupancy. To address this problem, we introduce a hierarchical warp–block reduction that avoids materializing the full floating-point vector and substantially reduces shared-memory usage (Section 4).

In summary, our main contributions are:

- **Analysis.** We show that matmul-centric quantization leaves the memory-bound part of LLM inference untouched: non-matmul operators account for 50%–70% of prefill latency in the models we profile, and during decoding every operator falls far below the GPU's compute-to-bandwidth ratio. This reframes activation quantization as a data-movement optimization rather than only a compute optimization.
- **INT8 pipeline.** We design an inference pipeline that preserves intermediate tensors in INT8 across supported operators, reducing global-memory traffic while performing numerically sensitive computation in higher precision on chip.
- **Resource-efficient kernels.** We develop INT8-in/INT8-out CUDA kernels for GEMM, GEMV, reduction-based, and element-wise operators, including a hierarchical warp–block reduction that bounds shared-memory usage and maintains high GPU occupancy.
- **Evaluation.** On five Qwen3 and Llama-3.2 models, four GPUs, and batch sizes from 1 to 16, our pipeline achieves up to $1.6\times$ speedup and up to 39% lower energy than BF16 inference while maintaining comparable perplexity and zero-shot accuracy.

2 Background and Motivation

Our work is motivated by two observations: *i*) matrix multiplications account for only part of LLM inference latency (Sec. 2.2), and *ii*) there is a large gap between GPU compute throughput and memory bandwidth (Sec. 2.1). Considered together, these observations motivate maintaining low-precision representations across both matmul and non-matmul operators.

2.1 The Imbalance Between GPU Computation and Data Movement

The roofline model characterizes an operator by its arithmetic intensity, the amount of computation performed per byte transferred from memory Williams et al. (2009). An operator whose arith-

Table 1: Profiling of operators in Qwen3-14B with a sequence length of 2048 tokens.

(a) Prefill stage

Operator	FLOPs	Memory Traffic	Arithmetic Intensity	Bound
q_proj & o_proj	107G	90M	1137	Compute
k_proj & v_proj	21G	34M	602	Compute
qk_matmul	42G	344M	119	Memory
softmax	0.84G	640M	1.2	Memory
sv_matmul	43G	344M	119	Memory
gate_proj & up_proj	365G	258M	1349	Compute
SiLU	0.14G	136M	1.1	Memory
down_proj	365G	258M	1349	Compute

(b) Decode stage

Operator	FLOPs	Memory Traffic	Arithmetic Intensity	Bound
q_proj & o_proj	52M	50M	1	Memory
k_proj & v_proj	10M	10M	1	Memory
qk_matmul	20M	4M	5	Memory
softmax	0.4M	0.3M	1.2	Memory
sv_matmul	20M	4M	5	Memory
gate_proj & up_proj	178M	170M	1	Memory
SiLU	0.07M	0.06M	1	Memory
down_proj	178M	170M	1	Memory

metic intensity is below the GPU's compute-to-memory-bandwidth ratio is memory-bound. Modern GPUs provide substantially more arithmetic throughput than global-memory bandwidth, making low-arithmetic-intensity operators sensitive to data movement. For example, the NVIDIA A40 provides 149 TFLOP/s of peak BF16 throughput but only 696 GB/s of global-memory bandwidth, corresponding to a compute-to-memory ratio of approximately 215 FLOP/byte.

Table 1 compares the arithmetic intensity of representative Transformer operators during prefill and decoding. During prefill, large matmuls exhibit high arithmetic intensity and are primarily compute-bound. In contrast, non-matmul operators have arithmetic intensities close to 1 FLOP/byte and are therefore memory-bound. During decoding, the arithmetic intensity of the matmul operators drop significantly as GEMMs become GEMVs. As a result, both matmul and non-matmul operators become memory-bound, making global-memory traffic a major performance bottleneck.

2.2 LLM Inference Is Not Dominated by Matmul Alone

Autoregressive LLM inference consists of two phases: *prefill*, which processes the input prompt in parallel, and *decoding*, which generates one token at a time Patel et al. (2024). Figure 2 shows the contribution of different operators to prefill latency under PyTorch eager execution. We group Softmax and normalization as reduction operators, while element-wise operators include masking, SiLU, arithmetic operations, and tensor-copy operations. Across five LLMs, matmul accounts for approximately 30%–50% of latency, the remaining latency is distributed across non-matmul operators. This observation exposes a limitation of existing LLM quantization approaches: optimizing matmul alone leaves a substantial fraction of prefill latency unaddressed. Therefore, quantization should also consider non-matmul operators.

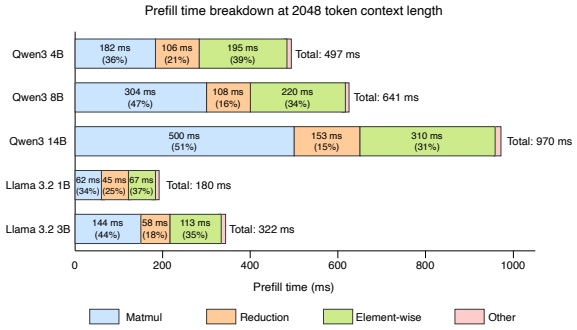


Figure 2: Breakdown of prefill latency across different operator categories for five LLMs.

3 INT8 Pipeline Design

Our pipeline maintains INT8 representations across matmul and non-matmul operators within the attention and feed-forward modules. We retain the normalization layers between these modules in BF16 because prior work has shown that those normalization layers are very sensitive to quantization Wei et al. (2023); Chang et al. (2025). Section 3.1 describes our quantized GEMM and GEMV kernels. Section 3.2 then presents the design for quantized non-matmul operators. The efficient GPU implementation of these non-matmul operators is then described in Section 4.

3.1 Quantized Matmul Operation

GEMM. We implement GEMM using an INT8 Tensor Core kernel and modify the output stage to produce INT8 tensors directly. Given quantized inputs X_q and weights W_q, the quantized output is computed as: $Y_q[m,n] = \frac{s_X[m]s_W[n]}{s_Y[m]} \sum_{k=1}^{K} X_q[m,k]W_q[k,n]$, where s_X and s_W denote the input

and weight scales, respectively, and s_Y is the output scale determined during calibration Williams & Aletras (2024). Our implementation builds on the CUTLASS INT8 GEMM kernel Thakkar et al. (2023) with a customized epilogue visitor tree. INT8 products are accumulated in INT32 using Tensor Cores. The epilogue then applies the input, weight, and output scales, and requantizes the result directly to INT8 before writing to the global memory. Consequently, neither INT32 nor floating-point output tensors are materialized in the global memory.

GEMV is memory-bound due to its low arithmetic intensity. Compared with large GEMMs, GEMV has much less parallelism along the matrix dimensions, making it more difficult to fully utilize Tensor Cores Hanindhito & John (2024). We therefore implement GEMV as a SIMT dot-product kernel in which each warp computes one output element $Y[n]$. INT8 input and weight values are accumulated into INT32 registers; the accumulator is then rescaled using the input and weight scales, requantized using the output scale, and written directly to global memory as INT8. Intermediate values remain in registers, avoiding the materialization of higher-precision tensors in global memory. Our implementation follows a similar strategy to the high-performance GGML library ggml-org, while additionally fusing output quantization to preserve INT8 tensors across operators.

3.2 Quantized Non-Matmul Operations

Figure 3 illustrates the INT8 dataflow for non-matmul operators. Both the input X_q and output Y_q are stored in INT8 in global memory. Because operators such as softmax and normalization require higher-precision computation, the input is dequantized only after being loaded on chip $X = \mathrm{dequantize}(X_q, s_X)$. The operator is then evaluated in floating point, $(Y = f(X))$, and the result is requantized before being written back to global memory. Only the resulting INT8 tensor Y_q and its scale s_Y are written to global memory. Compared with floating-point execution, this dataflow requires additional on-chip computation. Section 4 describes our operator-specific kernel designs for efficiently managing on-chip resources.

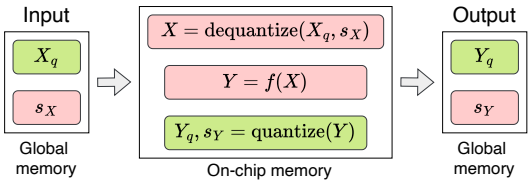


Figure 3: On-chip computation strategy for INT8 operators. Here, f may represent Softmax, RMSNorm, SiLU, or Sigmoid.

4 Resource-Efficient INT8 Kernel Design

Efficiently implementing the INT8 non-matmul dataflow described in Section 3.2 is challenging because higher-precision intermediate computation must be performed under limited on-chip memory space. The first challenge is shared-memory capacity. For example, the NVIDIA A40 provides 48 KiB of shared memory per block by default. Storing a single FP32 vector with $d = 12288$ elements requires 49152 bytes $= 48$ KiB, which exhausts the per-block shared-memory budget and leaves no room for additional temporary storage. The second challenge is occupancy. High shared-memory usage limits the number of concurrently blocks on each SM. For example, if each block consumes 48 KiB, an SM with 100 KiB of shared memory can accommodate at most two blocks. This reduces block-level concurrency and limits the GPU's ability to hide latency. Therefore, although INT8 input/output reduces global-memory traffic, inefficient management of on-chip memory can offset these benefits. To address this issue, we develop a hierarchical warp–block computation strategy that avoids materializing the full floating-point vector on chip.

4.1 Efficient INT8 GPU Kernel

Given an input tensor $X_q \in \mathbb{Z}^{R\times d}$, where R denotes the number of rows and d is the vector dimension, each CUDA block independently processes one row $X_q[r, :]$. This row-wise mapping naturally matches Transformer operators such as RMSNorm or Softmax, whose computations are performed independently along the final dimension. For higher-order tensors, all leading dimensions are flattened into R. Within each block, the d elements are distributed across B threads. Thread t processes the index set $\mathcal{J}_t = \{j \mid j = t+kB,\ j < d,\ k \geq 0\}$. After each thread computes a local partial result over $\mathcal{J}_t$, these partial results are combined using a hierarchical warp–block reduction. Next, within

Algorithm 1 INT8 RMSNorm with Hierarchical Warp–Block Reduction

Require: Input $x_q \in \mathbb{Z}^d$, input scale s_x, RMSNorm weight $\gamma \in \mathbb{R}^d$, CUDA block size B
Ensure: Quantized output $y_q \in \mathbb{Z}^d$ and output scale s_y
Thread t processes $\mathcal{J}_t = \{j \mid j = t + kB,\ j < d,\ k \geq 0\}$

Step 1: Compute the inverse RMS factor
$r_t \leftarrow 0$
for all $j \in \mathcal{J}_t$ **do**
    $x_j \leftarrow s_x x_q[j]$
    $r_t \leftarrow r_t + x_j^2$
end for
$R \leftarrow \textsc{BlockReduction}_{\text{sum}}(r_t)$
$\rho \leftarrow \textsc{Broadcast}\left((R/d + \epsilon)^{-1/2}\right)$

Step 2: Compute the output scale
$m_t \leftarrow 0$
for all $j \in \mathcal{J}_t$ **do**
    $y_j \leftarrow (s_x x_q[j])\, \rho\, \gamma[j]$
    $m_t \leftarrow \max(m_t, |y_j|)$
end for
$M \leftarrow \textsc{BlockReduction}_{\max}(m_t)$
$s_y \leftarrow \textsc{Broadcast}\left(\max(M/127, \epsilon_q)\right)$

Step 3: Recompute and quantize the output
for all $j \in \mathcal{J}_t$ **do**
    $y_j \leftarrow (s_x x_q[j])\, \rho\, \gamma[j]$
    $y_q[j] \leftarrow \text{clip}\left(\text{round}\left(y_j / s_y\right), -127, 127\right)$
end for
return y_q, s_y

each warp, partial values are reduced using register-level shuffle instructions. Each warp then writes one value to shared memory, and the first warp performs a second reduction to get the block-level results. Consequently, shared-memory usage scales with the number of warps, $O(N_{\text{warp}})$, rather than with the hidden dimension d. Detailed pseudocode for the warp- and block-level reduction are provided in Appendix A.2.

INT8 RMSNorm. Algorithm 1 shows how the hierarchical warp–block strategy is applied to INT8 RMSNorm computations, where the kernel makes three passes over the input. In the first pass, each thread dequantizes its assigned input elements and accumulates a local sum of squares. A block-level sum reduction produces the row-wide sum, from which the inverse RMS factor is computed. The second pass evaluates the normalized values and performs a maximum reduction to determine the output quantization scale. The final pass recomputes the normalized values, quantizes them to INT8, and writes the output to global memory.

The recomputation in the final pass introduces additional arithmetic, but avoids materializing the floating-point output vector in global memory. This trade-off is favorable for memory-bound operators. Our analysis in Appendix C.2 shows that INT8 RMSNorm achieves more than $2.6\times$ the arithmetic intensity of BF16 under the assumed execution model. The same hierarchical structure applies to other reduction-based operators, although the required statistics and number of passes differ; for example, Softmax requires maximum and sum reductions.

4.2 Efficient INT8 GPU Kernels for Element-Wise Operations

Element-wise operators apply a function independently to each input element. For example, SiLU computes $y_i = x_i \sigma(x_i)$. With INT8 input and output, the computation becomes more complex. Each input element is first dequantized as $x_i = s_x x_{q,i}$, and the resulting output must be requantized before being written to global memory. Under symmetric quantization, the output scale is $s_y = \max_i |y_i|/127$, which requires a reduction across the output elements. Thus, an element-wise operator that is fully independent in floating-point execution becomes a multi-stage INT8 computation. For SiLU, the kernel first dequantizes the input, evaluates the function, and performs a maxi-

mum reduction to determine s_y. It then recomputes SiLU in a second pass and writes the requantized INT8 output. Detail about floating-point and INT8 SiLU algorithm are provided in Appendix B.

This design introduces a trade-off, where the INT8 kernel performs more computation, but reads and writes fewer bytes than its floating-point counterpart. As shown in our evaluation, the advantage in memory traffic can outweigh the additional computation. Moreover, our analytical model in Appendix C.1 explains this behavior: INT8 SiLU has more than $2.6\times$ the arithmetic intensity of BF16 under the assumed execution model.

5 Experimental Results

5.1 Experimental Setup

We implement our approach in PyTorch 2.6 with CUDA 12.4 and GCC 11.5. For matrix multiplication, we use CUTLASS INT8 GEMM kernels, while non-matmul operators, including RMSNorm, Softmax, and element-wise operations, are implemented as custom CUDA kernels exposed through PyTorch wrappers. In the end-to-end model, supported operators within the attention and feed-forward regions use our INT8 pipeline. Only the normalization layers separating the attention and feed-forward modules are retained in BF16 for numerical robustness.

We evaluate Qwen3-4B, Qwen3-8B, and Qwen3-14B Yang et al. (2025), as well as Llama-3.2-1B and Llama-3.2-3B Grattafiori et al. (2024). Model quality is evaluated using perplexity (PPL) and zero-shot accuracy on ARC-Easy, ARC-Challenge, and PIQA. System efficiency is measured using end-to-end inference latency and energy consumption. Unless otherwise stated, experiments are conducted on an NVIDIA A40 GPU; we additionally evaluate RTX 5000 Ada, A30, and A100 GPUs to assess robustness across hardware platforms.

5.2 Quantization performance

Table 2 compares our method with the BF16 baseline and state-of-the-art quantization methods, including LLM.int8() Dettmers et al. (2022), SmoothQuant Xiao et al. (2023), and QuaRot Ashkboos et al. (2024). These technique use different classes of quantization: LLM.int8() uses mixed precision computations, SmoothQuant applies a smoothing technique, and QuaRot applies the rotation matrix to maintain model quality. In contrast, our method extends INT8 storages to both matrix and non-matrix operators (W8A8O8), targeting inference efficiency with minimal accuracy loss. Across all five models, our method consistently achieves the lowest inference latency and energy consumption. Compared with BF16, it reduces latency up to 37% and energy consumption up to 39%. We also consistently outperform SmoothQuant and QuaRot in system efficiency. These gains come with only a modest quality degradation: perplexity increases slightly, while zero-shot accuracy generally remains within 1–2 percentage points of the BF16 result. Overall, our results show that extending quantization beyond matrix multiplication provides substantial latency and energy benefits, while preserving competitive model quality.

5.3 Performance Across Inference Scenarios

We further evaluate our INT8 pipeline across different GPUs, batch sizes, and autoregressive decoding workloads. Figure 4 shows that INT8 reduces both matmul and non-matmul latency across all four evaluated GPUs, indicating that the benefits are not architecture-specific. Table 3 shows consistent latency reductions for batch sizes of 1, 4, and 16 across all evaluated models, with speedups of $1.4\times$–$1.6\times$ at batch size 1, and similar gains at larger batches.

For autoregressive decoding with KV caching, Table 4 shows that INT8 reduces inter-token latency and total generation time across all evaluated sequence lengths. Although INT8 incurs TTFT overhead at the shortest prompt length, this overhead is amortized as the prompt length increases. To better understand the decoding improvement, we compare the BF16 and INT8 GEMV kernels in Table 5. Across all dimensions, the INT8 kernel achieves approximately $2.4\times$ lower latency, due to the benefits of reducing memory traffic during memory-bound decoding.

5.4 Optimized Model with FlashAttention and Torch Compile

Table 2: Comparison of model quality, inference latency, and energy consumption.

Model	Method	Precision	Quality				Latency (ms) ↓	Energy (J) ↓
			PPL ↓	ARC-E ↑	ARC-C ↑	PIQA ↑		
Qwen3-4B	Float	BF16	7.41	78%	70%	78%	504	124
	LLM.int8()	Mixed	7.44	78%	70%	77%	680	177
	SmoothQuant	W8A8	7.48	78%	69%	77%	401	105
	QuaRot	W8A8	7.43	78%	70%	78%	460	111
	Ours	W8A8O8	7.54	77%	68%	77%	**337**	**82**
Qwen3-8B	Float	BF16	6.65	84%	79%	80%	651	174
	LLM.int8()	Mixed	6.71	83%	79%	80%	879	205
	SmoothQuant	W8A8	6.78	84%	79%	79%	502	125
	QuaRot	W8A8	6.67	84%	79%	80%	565	142
	Ours	W8A8O8	6.81	82%	77%	79%	**420**	**109**
Qwen3-14B	Float	BF16	6.07	89%	84%	81%	986	278
	LLM.int8()	Mixed	6.11	89%	84%	81%	1245	328
	SmoothQuant	W8A8	6.15	89%	84%	80%	705	198
	QuaRot	W8A8	6.08	89%	84%	81%	817	219
	Ours	W8A8O8	6.21	88%	84%	80%	**620**	**168**
Llama-3.2-1B	Float	BF16	11.99	40%	37%	75%	180	36
	LLM.int8()	Mixed	12.12	40%	37%	75%	307	64
	SmoothQuant	W8A8	12.11	39%	37%	74%	158	31
	QuaRot	W8A8	12.03	40%	36%	75%	172	34
	Ours	W8A8O8	12.56	39%	35%	74%	**130**	**22**
Llama-3.2-3B	Float	BF16	9.84	46%	45%	76%	321	77
	LLM.int8()	Mixed	9.82	46%	44%	75%	476	99
	SmoothQuant	W8A8	9.88	46%	44%	75%	257	61
	QuaRot	W8A8	9.86	46%	44%	76%	290	68
	Ours	W8A8O8	9.95	45%	44%	74%	**218**	**49**

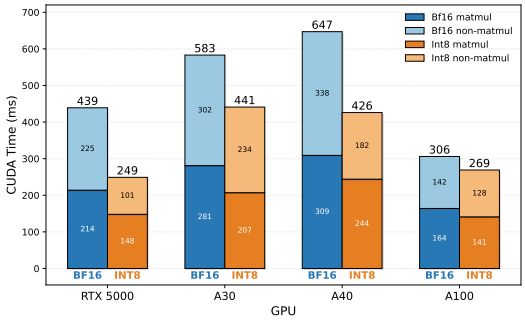


Figure 4: Operator-level latency breakdown of BF16 and our INT8 model across different GPUs.

All experiments presented so far use PyTorch eager, where INT8 kernels reduce latency to comparable BF16. Under graph-level optimization, `torch.compile` can fuse BF16 PyTorch operators and reduce kernel-launch Ansel et al. (2024). Thus, directly comparing compiled BF16 with custom INT8 would conflate compiler optimizations and quantization effects. FlashAttention Dao et al. (2022) presents a related case, where it reduces data movement by fusing the attention computation. We view this as an integration limitation rather than a limitation of the INT8 dataflow. Developing compiler-aware INT8 kernels is outside the scope of this work.

Table 6: End-to-end performance with FlashAttention and compiler optimization.

Model	Latency (ms)			Energy (J)		
	BF16	INT8	Gain	BF16	INT8	Save
Llama-3.2-1B	72	**55**	**1.3**×	11	**7**	**36%**
Llama-3.2-3B	185	**125**	**1.5**×	43	**26**	**39%**
Qwen3-4B	177	**160**	**1.1**×	40	**35**	**12%**
Qwen3-8B	315	**260**	**1.2**×	80	**64**	**20%**
Qwen3-14B	535	**400**	**1.3**×	147	**111**	**24%**

Rather than competing with these optimizations, we evaluate whether our INT8 pipeline remains beneficial alongside them. Accordingly, we use the same `torch.compile` and FlashAttention for both precisions and retain our INT8 pipeline only for the surrounding operators, including QKV projections, RMSNorm, feed-forward layers, and SiLU. Table 6 shows that our INT8 pipeline remains beneficial under this optimized setting, achieving up to 1.5× speedup and reducing energy consumption by up to 39%.

Table 3: TTFT latency under different batch sizes.

Model	Batch 1			Batch 4			Batch 16		
	BF16	INT8	Speedup	BF16	INT8	Speedup	BF16	INT8	Speedup
Llama 3.2–1B	180 ms	**130 ms**	**1.4**×	0.74 s	**0.52 s**	**1.4**×	2.94 s	**2.09 s**	**1.4**×
Llama 3.2–3B	321 ms	**218 ms**	**1.5**×	1.34 s	**0.89 s**	**1.5**×	5.24 s	**3.56 s**	**1.5**×
Qwen3–4B	504 ms	**337 ms**	**1.5**×	2.05 s	**1.35 s**	**1.5**×	8.15 s	**5.45 s**	**1.5**×
Qwen3–8B	651 ms	**420 ms**	**1.5**×	2.65 s	**1.70 s**	**1.6**×	10.55 s	**6.89 s**	**1.5**×
Qwen3–14B	986 ms	**620 ms**	**1.6**×	4.16 s	**2.59 s**	**1.6**×	16.61 s	**10.36 s**	**1.6**×

Table 4: Decoding performance of Qwen3-14B under BF16 and INT8 inference.

Generated Length	Prec.	TTFT (ms) ↓	ITL (ms/tok) ↓	Thr. (tok/s) ↑	Total (s) ↓
128	BF16	**78**	61	16	7.6
	INT8	125	**55**	**18**	**6.8**
512	BF16	206	64	15	32
	INT8	**195**	**56**	**18**	**28**
1024	BF16	411	67	14	68
	INT8	**320**	**60**	**16**	**61**

Table 5: BF16 and INT8 GEMV latency during decoding.

Dimension		Latency (ms) ↓		
M	N	BF16	INT8	Speedup
4096	4096	0.077	**0.031**	**2.4**×
4096	8192	0.139	**0.059**	**2.4**×
8192	8192	0.279	**0.115**	**2.4**×
8192	16384	0.540	**0.224**	**2.4**×

5.5 Efficiency of the INT8 Hierarchical Reduction

We compare BF16 reduction kernels with two INT8 designs for Softmax and RMSNorm: a naive implementation that stores the full d-dimensional vector in shared memory, and our hierarchical warp–block design. As shown in Table 7, naive INT8 can be slower and less energy-efficient than BF16 computation, despite operating on lower-precision data. This behavior indicates that its performance is dominated by inefficient on-chip usage rather than memory bandwidth. At $d = 16384$, the naive kernels cannot be launched because its shared-memory requirements exceed the default per-block limit. In contrast, the hierarchical kernels achieve the lowest latency for both operators.

To explain latency differences, Table 8 presents the NVIDIA Nsight Compute NVIDIA (2026) profiling for two INT8 designs. We report shared-memory allocation per block, achieved warp occupancy (the fraction of active warps during execution), and the shared-memory block limit (maximum number of blocks that can reside concurrently on SMs when only shared-memory capacity is considered). The naive design increases shared-memory usage from 17 KiB to 33 KiB as d grows from 4096 to 8192, reducing the shared-memory block limit from five to two, indicating that inefficient shared-memory usage restricts block concurrency. On the other hand, the hierarchical design only allocates 272 bytes across all dimensions because it stores only one partial result per warp. As a result, the kernel achieved 90%–92% warp occupancy and allows up to 11 blocks per SM.

5.6 Element-Wise Kernel Performance

Table 9 compares the BF16 and INT8 latency of SiLU. The profiling results show that BF16 execution is strongly memory-bound, with high memory utilization, but low compute utilization. In contrast, INT8 substantially increases arithmetic intensity and compute utilization. Despite the additional on-chip computation, INT8 consistently achieves lower latency and energy consumption across all evaluations. This demonstrates that, for element-wise operators, reducing global-memory traffic provides a larger benefit than the additional computation.

6 Related Work

Weight–activation quantization methods can be broadly grouped into mixed-precision, scaling/smoothing, and rotation-based approaches. Mixed-precision methods, such as LLM.int8() Dettmers et al. (2022) and MosaicQuant Hu et al. (2026), isolate outliers and process them at higher precision. Scaling-based methods, including SmoothQuant Xiao et al. (2023), OutlierTune Wang et al. (2024), and QServe Lin et al. (2025), rescale activations and weights to improve quantization robustness. Rotation-based methods, such as QuaRot Ashkboos et al. (2024), SpinQuant Liu et al. (2025), AffineQuant Ma et al. (2024), and ReSpinQuant Kim et al. (2026), transform

Table 7: Performance of BF16, naive INT8, and hierarchical INT8 reduction kernels.

(a) Softmax

d	Kernel	Latency (ms)	Energy (mJ)
4096	BF16	0.07	5
	Naive INT8	0.07	5
	Hierarchical INT8	0.07	5
8192	BF16	0.21	18
	Naive INT8	0.23	20
	Hierarchical INT8	**0.11**	**9**
16384	BF16	0.52	55
	Naive INT8	Exceeds SMEM	
	Hierarchical INT8	**0.30**	**30**

(b) RMSNorm

d	Kernel	Latency (ms)	Energy (mJ)
4096	BF16	0.07	4
	Naive INT8	0.07	4
	Hierarchical INT8	0.07	4
8192	BF16	0.19	16
	Naive INT8	0.27	23
	Hierarchical INT8	**0.11**	**9**
16384	BF16	0.39	39
	Naive INT8	Exceeds SMEM	
	Hierarchical INT8	**0.32**	**32**

Table 8: NCU profiling of naive and hierarchical INT8 reduction kernels.

(a) Softmax

d	Kernel	SMEM	Occ.	Blocks limit
4096	Naive	17 KiB	78%	5
	Hierarchical	**272 B**	**90%**	**11**
8192	Naive	33 KiB	33%	2
	Hierarchical	**272 B**	**90%**	**11**
16384	Naive	$>$ limit	–	–
	Hierarchical	**272 B**	**90%**	**11**

(b) RMSNorm

d	Kernel	SMEM	Occ.	Blocks limit
4096	Naive	17 KiB	80%	5
	Hierarchical	**272 B**	**92%**	**11**
8192	Naive	33 KiB	33%	2
	Hierarchical	**272 B**	**92%**	**11**
16384	Naive	$>$ limit	–	–
	Hierarchical	**272 B**	**92%**	**11**

Table 9: SiLU performance under BF16 and INT8 execution.

Dim.	Precision	Latency (ms)	Speedup	Energy (mJ)	AI	Compute / Memory Util.
4096	BF16	0.07	1.00×	4	0.7	16% / 84%
	INT8	**0.04**	**1.75**×	**3**	4.2	82% / 62%
8192	BF16	0.13	1.00×	10	0.7	15% / 87%
	INT8	**0.08**	**1.63**×	**8**	3.5	81% / 74%
16384	BF16	0.25	1.00×	22	0.6	15% / 88%
	INT8	**0.18**	**1.39**×	**20**	2.7	75% / 85%

activations and weights into more quantization-friendly representations. These approaches primarily focus on improving the numerical robustness of low-precision matmul.

Several works have extended quantization beyond matrix multiplication. I-BERT Kim et al. (2021) uses polynomial approximations for integer-only nonlinear operations, Jetfire Xi et al. (2024) introduces an INT8 dataflow for Transformer pretraining while retaining FP16 attention and Softmax, and IntAttention Zhong et al. (2026) develops an integer attention pipeline with lookup-table-based Softmax. In contrast, our work preserves INT8 tensors across both matmul and non-matmul operators, while performing numerically sensitive computation in higher precision on chip. Because on-chip memory is limited, we design resource-aware kernels that reduce shared-memory pressure. Together, our techniques extend quantization beyond matrix multiplication by jointly optimizing global-memory traffic and on-chip resource utilization.

7 Conclusion

We present an INT8 inference pipeline that extends quantization beyond matrix multiplication by preserving intermediate tensors in INT8 across both matmul and non-matmul operators, while performing numerically sensitive computation in higher precision on chip. To make this pipeline efficient, we develop operator-specific kernels for GEMM, GEMV, reduction-based, and element-wise operators, including a hierarchical warp–block reduction strategy that reduces shared-memory pressure and maintains high GPU occupancy. Across Qwen3 and Llama-3.2 models, our approach achieves up to $1.6\times$ inference speedup and 39% lower energy consumption, with consistent benefits across GPUs, decoding workloads, and batch sizes.

AI USE STATEMENT

We used generative AI tools, including ChatGPT, to assist with manuscript editing and readability, code-related troubleshooting, checking mathematical derivations, and discussing the presentation. All AI-assisted material was reviewed by the authors. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

REFERENCES

Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. PyTorch 2: Faster machine learning through dynamic Python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 929–947, 2024.

Saleh Ashkboos, Amirkeivan Mohtashami, Maximilian L Croci, Bo Li, Pashmina Cameron, Martin Jaggi, Dan Alistarh, Torsten Hoefler, and James Hensman. Quarot: Outlier-free 4-bit inference in rotated llms. *Advances in Neural Information Processing Systems*, 37:100213–100240, 2024.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pp. 1877–1901, 2020.

Ting-Yun Chang, Muru Zhang, Jesse Thomason, and Robin Jia. Why do some inputs break low-bit llm quantization? In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 3410–3429, 2025.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*, volume 35, pp. 16344–16359, 2022.

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale. *Advances in neural information processing systems*, 35: 30318–30332, 2022.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. GPTQ: Accurate post-training quantization for generative pre-trained transformers. In *International Conference on Learning Representations*, 2023.

ggml-org. ggml: Tensor library for machine learning. URL `https://github.com/ggml-org/ggml`. GitHub repository.

Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. In *Low-Power Computer Vision*, pp. 291–326. Chapman and Hall/CRC, 2022.

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

Bagus Hanindhito and Lizy K John. Accelerating ml workloads using gpu tensor cores: The good, the bad, and the ugly. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering*, pp. 178–189, 2024.

Yangjia Hu, Haodong Wang, Zicong Hong, Qianli Liu, Quanxin Shou, Jian Lin, Song Guo, Xiaowei Shen, Xiangjun Huang, Dian Wang, et al. Mosaicquant: Inlier-outlier disaggregation for unified 4-bit llm quantization. *arXiv preprint arXiv:2606.15652*, 2026.

Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. Data movement is all you need: A case study on optimizing transformers. *Proceedings of Machine Learning and Systems*, 3:711–732, 2021.

Sehoon Kim, Amir Gholami, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. I-bert: Integer-only bert quantization. In *International conference on machine learning*, pp. 5506–5518. PMLR, 2021.

Suyoung Kim, Sunghyun Wee, Hyeonjin Kim, Kyomin Hwang, Hyunho Lee, and Nojun Kwak. Respinquant: Efficient layer-wise llm quantization via subspace residual rotation approximation. *arXiv preprint arXiv:2604.11080*, 2026.

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. *Proceedings of Machine Learning and Systems*, 6:87–100, 2024.

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

Zechun Liu, Changsheng Zhao, Igor Fedorov, Bilge Soran, Dhruv Choudhary, Raghuraman Krishnamoorthi, Vikas Chandra, Yuandong Tian, and Tijmen Blankevoort. Spinquant: Llm quantization with learned rotations. In *International Conference on Learning Representations*, volume 2025, pp. 92009–92032, 2025.

Yuexiao Ma, Huixia Li, Xiawu Zheng, Feng Ling, Xuefeng Xiao, Rui Wang, Shilei Wen, Fei Chao, and Rongrong Ji. Affinequant: Affine transformation quantization for large language models. In *International Conference on Learning Representations*, volume 2024, pp. 50932–50951, 2024.

Stefan K Muller and Jan Hoffmann. Modeling and analyzing evaluation cost of cuda kernels. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–31, 2021.

NVIDIA. *CUDA Programming Guide*. NVIDIA, 2026. URL `https://docs.nvidia.com/cuda/cuda-programming-guide/index.html`. Version 13.3. Last updated May 27, 2026.

NVIDIA. NVIDIA Nsight Compute. `https://developer.nvidia.com/nsight-compute`, 2026. Accessed: 2026-07-10.

Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pp. 118–132. IEEE, 2024.

Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems*, 5:606–624, 2023.

Noam Shazeer. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*, 2019.

Amir Taherin, Sana Taghipour Anvari, Charles Amante, Yixiao Chen, Ruben Noroian, Zlatan Feric, Nicolas Bohm Agostini, Pu Zhao, José Cano, Bin Ren, et al. Hydra: Phase-aware workload characterization of llm inference across edge soc generations, backends, and quantization levels. *arXiv preprint arXiv:2608.25053*, 2026.

Vijay Thakkar, Pradeep Ramani, Cris Cecka, Aniket Shivam, Honghao Lu, Ethan Yan, Jack Kosaian, Mark Hoemmen, Haicheng Wu, Andrew Kerr, Matt Nicely, Duane Merrill, Dustyn Blasig, Aditya Atluri, Fengqi Qiao, Piotr Majcher, Paul Springer, Markus Hohnerbach, Jin Wang, and Manish Gupta. CUTLASS, January 2023. URL `https://github.com/NVIDIA/cutlass`.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

Jinguang Wang, Yuexi Yin, Haifeng Sun, Qi Qi, Jingyu Wang, Zirui Zhuang, Tingting Yang, and Jianxin Liao. Outliertune: Efficient channel-wise quantization for large language models. *arXiv preprint arXiv:2406.18832*, 2024.

Xiuying Wei, Yunchen Zhang, Yuhang Li, Xiangguo Zhang, Ruihao Gong, Jinyang Guo, and Xianglong Liu. Outlier suppression+: Accurate quantization of large language models by equivalent and effective shifting and scaling. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 1648–1665, 2023.

Miles Williams and Nikolaos Aletras. On the impact of calibration data in post-training quantization and pruning. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 10100–10118, 2024.

Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

Haocheng Xi, Yuxiang Chen, Kang Zhao, Kai Jun Teh, Jianfei Chen, and Jun Zhu. Jetfire: Efficient and accurate transformer pretraining with int8 data flow and per-block quantization. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235, pp. 54049–54063. PMLR, 2024.

Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International conference on machine learning*, pp. 38087–38099. PMLR, 2023.

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

Wanli Zhong, Haibo Feng, Zirui Zhou, Hanyang Peng, and Shiqi Yu. Intattention: A fully integer attention pipeline for efficient edge inference. *Proceedings of Machine Learning and Systems*, 8: 155–168, 2026.

Algorithm 2 Warp-Level Reduction

Require: Local value v_{local} held by lane ℓ, warp size $W = 32$, reduction operator OP
Ensure: Lane 0 holds the warp reduction result
$s \leftarrow W/2$
while $s > 0$ **do**
  $v_{\text{stride}} \leftarrow \textsc{ShuffleDown}(v_{\text{local}}, s)$
  if $\ell < s$ **then**
    $v_{\text{local}} \leftarrow \text{OP}(v_{\text{local}}, v_{\text{stride}})$
  end if
  $s \leftarrow s/2$
end while
return v_{local} ▷ Valid reduction result in lane 0

Algorithm 3 Block-Level Reduction

Require: Local value v_i held by thread i, block size B, warp size $W = 32$, reduction operator OP
Ensure: Reduction result for block
Allocate shared memory $\texttt{smem}[32]$
$\ell \leftarrow i \bmod W$ ▷ Lane index within warp
$w \leftarrow \lfloor i/W \rfloor$ ▷ Warp index within block
$v_{\text{warp}} \leftarrow \textsc{WarpReduction}(v_i, \text{OP})$
if $\ell = 0$ **then**
  $\texttt{smem}[w] \leftarrow v_{\text{warp}}$
end if
SYNCHRONIZETHREADS()
if $w = 0$ **then**
  $v_{\text{block}} \leftarrow \textsc{WarpReduction}(\texttt{smem}[\ell], \text{OP})$
end if
return v_{block}

A GPU Method Details

A.1 GPU Memory Hierarchy

NVIDIA GPUs expose multiple memory spaces with different capacity and latency NVIDIA (2026). *Global memory* is accessible by all threads and provides the main storage for model weights and tensors. Although global memory has large capacity, accesses are substantially more expensive than accesses to on-chip storage. For this reason, our INT8 pipeline stores tensors in compact INT8 format in global memory and performs higher-precision computation only after the data has been loaded on chip. *Shared memory* is an on-chip memory space shared by all threads within a CUDA block. Compared with global memory, shared memory provides much lower-latency. However, its capacity is limited.

This limitation is particularly important for reduction-based operators. A straightforward implementation may dequantize an entire d-dimensional INT8 vector and materialize the resulting FP32 values in shared memory, requiring $4d$ bytes of shared memory per block. For example, when $d = 12{,}288$, storing one FP32 vector requires $12{,}288 \times 4 = 49{,}152$ bytes $= 48$ KiB. On the NVIDIA A40 used in our experiments, this reaches the standard 48-KiB per-block shared-memory allocation limit. Our hierarchical reduction is designed to avoid materializing the entire floating-point vector in shared memory.

A.2 Hierarchical Warp–Block Reduction

This subsection presents the warp- and block-level reduction primitives used by the INT8 kernels described in Section 4.1.

CUDA Execution Hierarchy. A CUDA kernel is executed by a grid of thread blocks, where each block contains multiple threads NVIDIA (2026). Within a block, threads are organized by the hardware into groups of $W = 32$ threads, called *warps*. Threads within the same block can cooperate through shared memory and block-wide synchronization, while threads within a warp can directly exchange register-resident values using warp-level primitives.

Given an input tensor $X \in \mathbb{R}^{R\times d}$, where R is the sequence length and d is the hidden dimension, we assign one CUDA block to process each row $X_r \in \mathbb{R}^d$. Within the block, the elements of the row are distributed across B threads. Each thread first applies a reduction operator OP, such as addition or maximum, over its assigned elements to produce a thread-local partial result. Thread t processes the strided subset

$$\mathcal{J}_t = \{j \mid j = t + kB,\ j < d,\ k \geq 0\}.$$

The choice of block size B affects kernel performance because it determines the amount of work assigned to each thread and the number of warps. We therefore empirically tune the block size over a set of candidate configurations $\mathcal{B} = \{64, 128, 256, 512, 1024\}$ and select

$$B^* = \arg\min_{B\in\mathcal{B}} L(B),$$

where $L(B)$ denotes the measured kernel latency for block size B. Across the evaluated kernels, $B = 256$ consistently achieves the lowest latency. This configuration is also consistent with common CUDA benchmarks that use 256-thread blocks Muller & Hoffmann (2021).

Warp-Level Reduction. Algorithm 2 shows the warp-level reduction. Starting from a thread-local value, the reduction proceeds in a tree-based manner with strides $W/2, W/4, \ldots, 1$. At each step, each lane obtains a value from another lane that is s positions away and combines it with its current value using OP. After $\log_2 W$ steps, lane 0 contains the reduction result for the warp. Warp-level communication is performed through register shuffle operations and therefore does not require intermediate shared-memory accesses or block-wide synchronization.

Block-Level Reduction. To combine results across multiple warps, we use the block-level reduction shown in Algorithm 3. Let $N_w = B/W$ denote the number of warps in a block, assuming that B is a multiple of the warp size. In the first stage, each warp independently reduces the thread-local values using Algorithm 2. Lane 0 of each warp then writes its warp-level result to shared memory. After a block-wide synchronization, the first warp then performs a second warp-level reduction, producing the final block-wide result in thread 0.

Because a CUDA block contains at most 1024 threads, it contains at most 32 warps. Therefore, each block-level reduction requires only 32 shared-memory elements to store the warp-level partial results. In the normalization kernel in Algorithm 1, block-level reduction is used twice, requiring two 32-element FP32 buffers. The reduction therefore requires only $2 \times 32 \times 4 = 256$ bytes of shared memory per block. This explains the small shared-memory footprint observed in the NCU profiling results in Table 8, where the compiled kernel uses 272 bytes per block. Of these, 256 bytes correspond to the reduction buffers, while the remaining 16 bytes arise from additional shared-memory allocation in the compiled kernel.

INT8 Softmax. Although Algorithm 1 illustrates the hierarchical reduction using RMSNorm, the same principle applies to other row-wise operators that require global statistics over the hidden dimension. For example, RMSNorm requires a sum-of-squares reduction, whereas Softmax requires two reductions: a maximum to obtain a numerically stable offset and a summation to compute the normalization factor. Algorithm 4 shows how these two hierarchical reductions are integrated into the INT8 Softmax kernel.

B INT8 SiLU

Algorithm 5 compares floating-point and INT8 SiLU execution. The floating-point kernel evaluates $y_j = x_j\sigma(x_j)$ independently for each element and writes the result directly to memory. In contrast, the INT8 kernel must first dequantize the input, evaluate SiLU, and compute the maximum output magnitude to determine the output scale s_y. It then performs a second pass to recompute the output and quantize it to INT8 before writing to global memory. Thus, INT8 SiLU introduces additional

Algorithm 4 INT8 Softmax with Hierarchical Warp–Block Reduction

Require: Input $x_q \in \mathbb{Z}^d$, input scale s_x, CUDA block size B
Ensure: Quantized output $y_q \in \mathbb{Z}^d$ and output scale s_y
Thread t processes $\mathcal{J}_t = \{j \mid j = t + kB,\ j < d,\ k \geq 0\}$

Step 1: Compute the maximum value
$m_t \leftarrow -\infty$
for all $j \in \mathcal{J}_t$ **do**
    $x_j \leftarrow s_x x_q[j]$
    $m_t \leftarrow \max(m_t, x_j)$
end for
$M \leftarrow \textsc{BlockReduction}_{\max}(m_t)$
$M \leftarrow \textsc{Broadcast}(M)$

Step 2: Compute output scale
$r_t \leftarrow 0$
for all $j \in \mathcal{J}_t$ **do**
    $x_j \leftarrow s_x x_q[j]$
    $r_t \leftarrow r_t + \exp(x_j - M)$
end for
$R \leftarrow \textsc{BlockReduction}_{\text{sum}}(r_t)$
$s_y \leftarrow \textsc{Broadcast}\left(\max\left(\frac{1}{127R}, \epsilon_q\right)\right)$

Step 4: Recompute and quantize the output
for all $j \in \mathcal{J}_t$ **do**
    $x_j \leftarrow s_x x_q[j]$
    $y_j \leftarrow \exp(x_j - M)/R$
    $y_q[j] \leftarrow \text{clip}\left(\text{round}\left(y_j/s_y\right), 0, 127\right)$
end for
return y_q, s_y

Algorithm 5 Floating-point and INT8 SiLU execution

(a) Floating-point SiLU
Require: Input $x \in \mathbb{R}^d$
Ensure: Output $y \in \mathbb{R}^d$
Thread t processes
$$\mathcal{J}_t = \{j \mid j = t + kB,\ j < d,\ k \geq 0\}$$
for all $j \in \mathcal{J}_t$ **do**
    $y_j \leftarrow x_j \sigma(x_j)$
end for

(b) INT8 SiLU
Require: Input $x_q \in \mathbb{Z}^d$, scale s_x
Ensure: Output $y_q \in \mathbb{Z}^d$, scale s_y
$m_t \leftarrow 0$
Thread t processes
$$\mathcal{J}_t = \{j \mid j = t + kB,\ j < d,\ k \geq 0\}$$
for all $j \in \mathcal{J}_t$ **do**
    $x_j \leftarrow s_x x_q[j]$
    $y_j \leftarrow x_j \sigma(x_j)$
    $m_t \leftarrow \max(m_t, |y_j|)$
end for
$M \leftarrow \textsc{BlockReduction}_{\max}(m_t)$
$s_y \leftarrow M/127$
for all $j \in \mathcal{J}_t$ **do**
    $x_j \leftarrow s_x x_q[j]$
    $y_j \leftarrow x_j \sigma(x_j)$
    $y_q[j] \leftarrow \text{clip}(\text{round}(y_j/s_y), -127, 127)$
end for

on-chip computation and a reduction, but avoids materializing the floating-point output tensor in global memory. Our analytical model in Appendix C.1 explains this behavior: INT8 SiLU has more than $2.7\times$ the arithmetic intensity of BF16 under the assumed execution model.

C Arithmetic-intensity Analysis

C.1 Element-wise Operation

We analyze the arithmetic intensity of BF16 and INT8 SiLU under the execution schemes in Algorithm 5. Let d denote the number of elements in a row and C_σ the arithmetic cost of evaluating the sigmoid function. Because the cost of transcendental functions is hardware dependent, we keep C_σ symbolic. In this analysis, we use an operation-count model in which each scalar multiplication, division, absolute-value operation, maximum comparison, rounding operation, and clipping operation is counted as one arithmetic operation.

Arithmetic intensity is defined as the number of arithmetic operations per byte transferred to or from global memory. This model captures relative arithmetic work rather than exact hardware FLOPs. Mathematically,

$$\mathrm{AI} = \frac{\text{arithmetic operations}}{\text{bytes transferred to/from global memory}}.$$

Proposition 1 *Under the above operation-count model and neglecting the $O(1)$ scale-storage overhead per row, the arithmetic intensities of BF16 and INT8 SiLU are*

$$\mathrm{AI}_{\mathrm{BF16}} = \frac{C_\sigma + 1}{4}, \qquad \mathrm{AI}_{\mathrm{INT8}} = \frac{2C_\sigma + 9}{3}.$$

Consequently, for any finite $C_\sigma \geq 0$,

$$\frac{\mathrm{AI}_{\mathrm{INT8}}}{\mathrm{AI}_{\mathrm{BF16}}} = \frac{8}{3} + \frac{28}{3(C_\sigma + 1)} > \frac{8}{3}.$$

Thus, INT8 SiLU has more than $8/3$ times the arithmetic intensity of BF16 SiLU.

Proof of Proposition 1. BF16 SiLU evaluates each element once as $y_j = x_j\sigma(x_j)$, requiring $C_\sigma + 1$ operations per element. Since the kernel reads and writes one BF16 value per element, its arithmetic work and global-memory traffic are

$$W_{\mathrm{BF16}} = d(C_\sigma + 1), \qquad Q_{\mathrm{BF16}} = 4d.$$

The INT8 scheme requires two passes because the output scale $s_y = \max_j |y_j|/127$ must be determined before quantization. In the first pass, each element is dequantized, evaluated by SiLU, and converted to its magnitude, requiring $C_\sigma + 3$ operations. The maximum reduction requires $d - 1$ comparisons, followed by one division to compute s_y. Hence,

$$W_{\mathrm{pass1}} = d(C_\sigma + 3) + (d - 1) + 1 = d(C_\sigma + 4).$$

In the second pass, each element is dequantized and evaluated again, followed by division by s_y, rounding, and clipping. This requires $C_\sigma + 5$ operations per element, giving

$$W_{\mathrm{pass2}} = d(C_\sigma + 5), \qquad W_{\mathrm{INT8}} = d(2C_\sigma + 9).$$

Under the assumed execution scheme, the first pass reads d INT8 values, while the second pass reads d INT8 values and writes d INT8 values. Neglecting scale-storage overhead, the total global-memory traffic is therefore

$$Q_{\mathrm{INT8}} = 3d.$$

Substituting the work and traffic counts yields

$$\mathrm{AI}_{\mathrm{BF16}} = \frac{C_\sigma + 1}{4}, \qquad \mathrm{AI}_{\mathrm{INT8}} = \frac{2C_\sigma + 9}{3}.$$

Consequently,

$$\frac{\mathrm{AI}_{\mathrm{INT8}}}{\mathrm{AI}_{\mathrm{BF16}}} = \frac{8}{3} + \frac{28}{3(C_\sigma + 1)}.$$

Since $C_\sigma \geq 0$, the second term is positive for any C_σ, establishing the claimed lower bound. □

C.2 RMSNorm Analysis

In this subsection, we analyze the arithmetic intensity of BF16 and INT8 RMSNorm. Let d denote the number of elements in a row and C_{rsqrt} the arithmetic cost of an inverse-square-root operation. We use the same operation-count model as in Section C.1. The INT8 kernel follows the three-stage execution scheme in Algorithm 1: the first stage computes the inverse RMS factor, the second determines the output scale, and the third recomputes and quantizes the output. Constant-size scale-storage is neglected.

Proposition 2 *Under the above operation-count and memory-traffic assumptions, the arithmetic intensities of BF16 and INT8 RMSNorm are*

$$\mathrm{AI}_{\mathrm{BF16}} = \frac{4d + 1 + C_{\mathrm{rsqrt}}}{6d}, \qquad \mathrm{AI}_{\mathrm{INT8}} = \frac{14d + 2 + C_{\mathrm{rsqrt}}}{8d}.$$

Consequently, the ratio of their arithmetic intensities is

$$\frac{\mathrm{AI}_{\mathrm{INT8}}}{\mathrm{AI}_{\mathrm{BF16}}} = \frac{3(14d + 2 + C_{\mathrm{rsqrt}})}{4(4d + 1 + C_{\mathrm{rsqrt}})} = \frac{21}{8} - \frac{9 + 15C_{\mathrm{rsqrt}}}{8(4d + 1 + C_{\mathrm{rsqrt}})}.$$

For fixed C_{rsqrt}, the correction term vanishes as $d \to \infty$. Hence,

$$\lim_{d\to\infty} \frac{\mathrm{AI}_{\mathrm{INT8}}}{\mathrm{AI}_{\mathrm{BF16}}} = \frac{21}{8} = 2.625.$$

Thus, the INT8 execution has approximately $2.63\times$ the arithmetic intensity of BF16 RMSNorm.

Proof of Proposition 2. BF16 RMSNorm computes

$$R = \sum_{j=1}^{d} x_j^2, \qquad \rho = (R/d + \epsilon)^{-1/2}, \qquad y_j = x_j \rho \gamma_j.$$

The computation requires d squares, $d - 1$ additions for the reduction, one division, one addition of ϵ, one inverse-square-root operation, and $2d$ multiplications for the output. Therefore,

$$W_{\mathrm{BF16}} = 4d + 1 + C_{\mathrm{rsqrt}}.$$

Reading the BF16 input and weight vectors and writing the BF16 output requires

$$Q_{\mathrm{BF16}} = 2d + 2d + 2d = 6d$$

bytes of global-memory traffic.

The INT8 implementation introduces an additional reduction to determine the output scale and a third stage to quantize the output. In the first stage, each element is dequantized and squared, and the squared values are reduced to compute the inverse RMS factor. The arithmetic work is

$$\begin{aligned} W_1 &= d + d + (d - 1) + 1 + 1 + C_{\mathrm{rsqrt}} \\ &= 3d + 1 + C_{\mathrm{rsqrt}}. \end{aligned}$$

Here, the first two terms account for dequantization and squaring, the third for the sum reduction, and the remaining terms for division by d, addition of ϵ, and inverse square root.

In the second stage, each element is dequantized and normalized as

$$y_j = (s_x x_{q,j}) \rho \gamma_j.$$

This requires three multiplications and one absolute-value operation per element. Computing the maximum magnitude requires $d - 1$ comparisons, followed by one division and one maximum comparison to obtain

$$s_y = \max\left(\frac{\max_j |y_j|}{127}, \epsilon_q\right).$$

Thus,

$$W_2 = 4d + (d - 1) + 1 + 1 = 5d + 1.$$

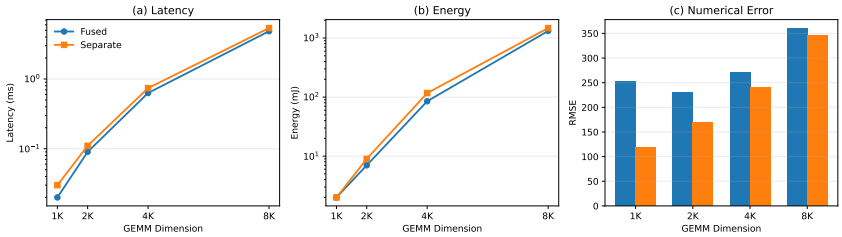


Figure 5: Comparison between fused and separate W8A8O8 implementations across different GEMM dimensions. The fused implementation has slightly lower latency and energy consumption, while the separate implementation achieves slightly lower numerical error. Latency and energy are shown on a logarithmic scale.

The third stage recomputes the normalized output and applies division by s_y, rounding, and clipping, requiring six operations per element:

$$W_3 = 6d.$$

The total INT8 arithmetic work is therefore

$$W_{\text{INT8}} = W_1 + W_2 + W_3 = 14d + 2 + C_{\text{rsqrt}}.$$

Under the assumed memory-access scheme, the first stage reads d INT8 input values. The second and third stages each read d INT8 input values and d BF16 weights, while the third stage additionally writes d INT8 output values. Hence,

$$Q_{\text{INT8}} = d + (d + 2d) + (d + 2d + d) = 8d.$$

Substituting the work and traffic counts gives the arithmetic intensities stated in Proposition 2:

$$\frac{\text{AI}_{\text{INT8}}}{\text{AI}_{\text{BF16}}} = \frac{3(14d + 2 + C_{\text{rsqrt}})}{4(4d + 1 + C_{\text{rsqrt}})},$$

which approaches $21/8$ as d increases for fixed C_{rsqrt}. □

Both analysis in subsection C.1 and C.2 shows that INT8 SiLU and INT8 RMSNorm has higher arithmetic intensity than its BF16 counterpart. The increased arithmetic intensity moves the operator toward the compute-limited region of the roofline model, consistent with the higher compute utilization and lower memory utilization observation in Table 9.

D Additional Experimental Analysis

D.1 Fused vs. Separate W8A8O8 GEMM

We evaluate two implementations of W8A8O8 GEMM. The fused implementation performs scaling and requantization directly in the GEMM epilogue, avoiding the need to write the intermediate BF16 output to global memory. This implementation uses a precomputed output scale s_Y obtained from calibration Williams & Aletras (2024). In contrast, the separate implementation first produces the GEMM output and then applies a separate quantization kernel, where s_Y is computed from the current activation. This introduces additional memory traffic and kernel-launch overhead, but allows the output scale to adapt to the runtime output distribution.

Figure 5 compares the two implementations. The fused implementation generally achieves slightly lower latency and energy consumption because it avoids materializing the intermediate output and launching an additional quantization kernel. The separate implementation achieves slightly lower RMSE because its output scale is computed directly from the current GEMM output. However, the differences in both efficiency and numerical error are small across the evaluated dimensions. Therefore, the choice between fused and separate W8A8O8 has no substantial effect on the overall latency or numerical accuracy in our experiments. By default, we use the fused implementation to minimize latency, while the separate implementation can be selected when numerical accuracy is prioritized.