# Efficient Matrix Multiplication Algorithms for Quantized LLMs


Johannes Gäßler, 21.02.24

# About Me

- Will soon start a PhD in experimental particle physics at KIT
- Additional bachelor’s degree in informatics
- Background in computing and statistics
- One of the llama.cpp core developers (mostly CUDA)

# CUDA

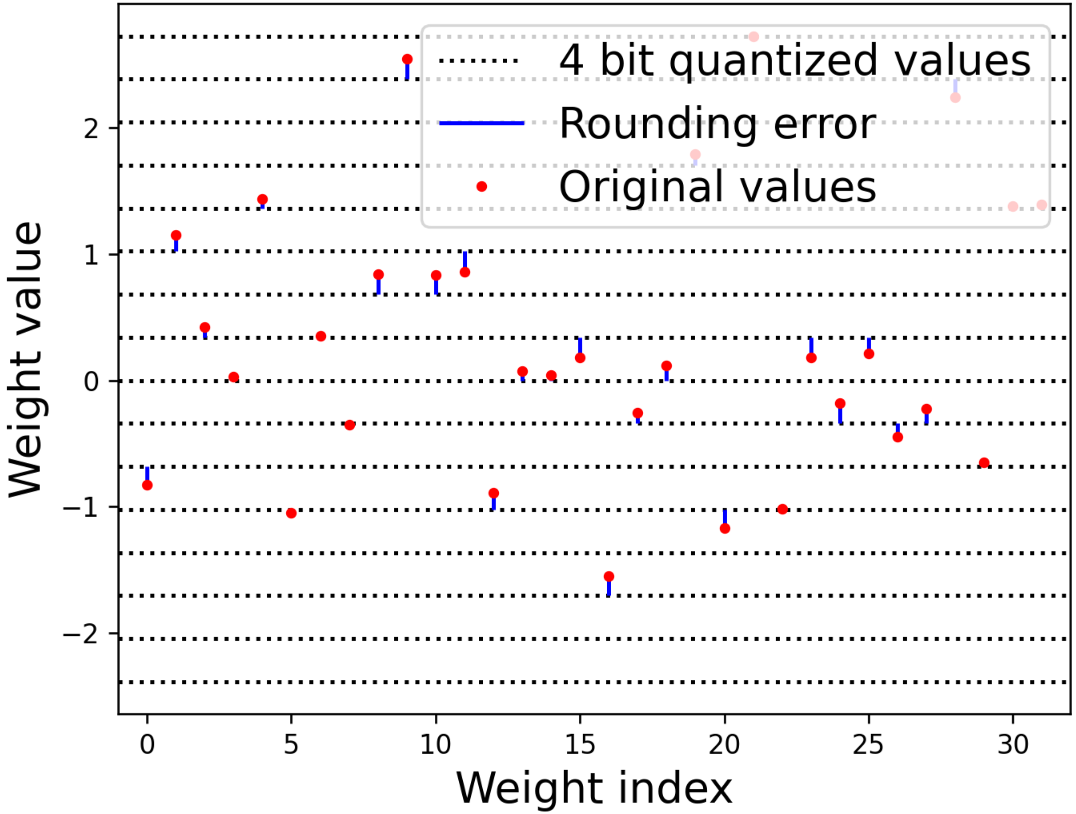
- NVIDIA’s first-party platform for general-purpose computing on GPUs
- Massive parallelism
- Suitable for artificial neural networks, e.g. transformers

# llama.cpp

- Open-source C/C++ program for LLM inference
- Wide support across hardware and OSs
- Quantization support for 2-8.5 bits per weight (1.5 BPW is WIP)

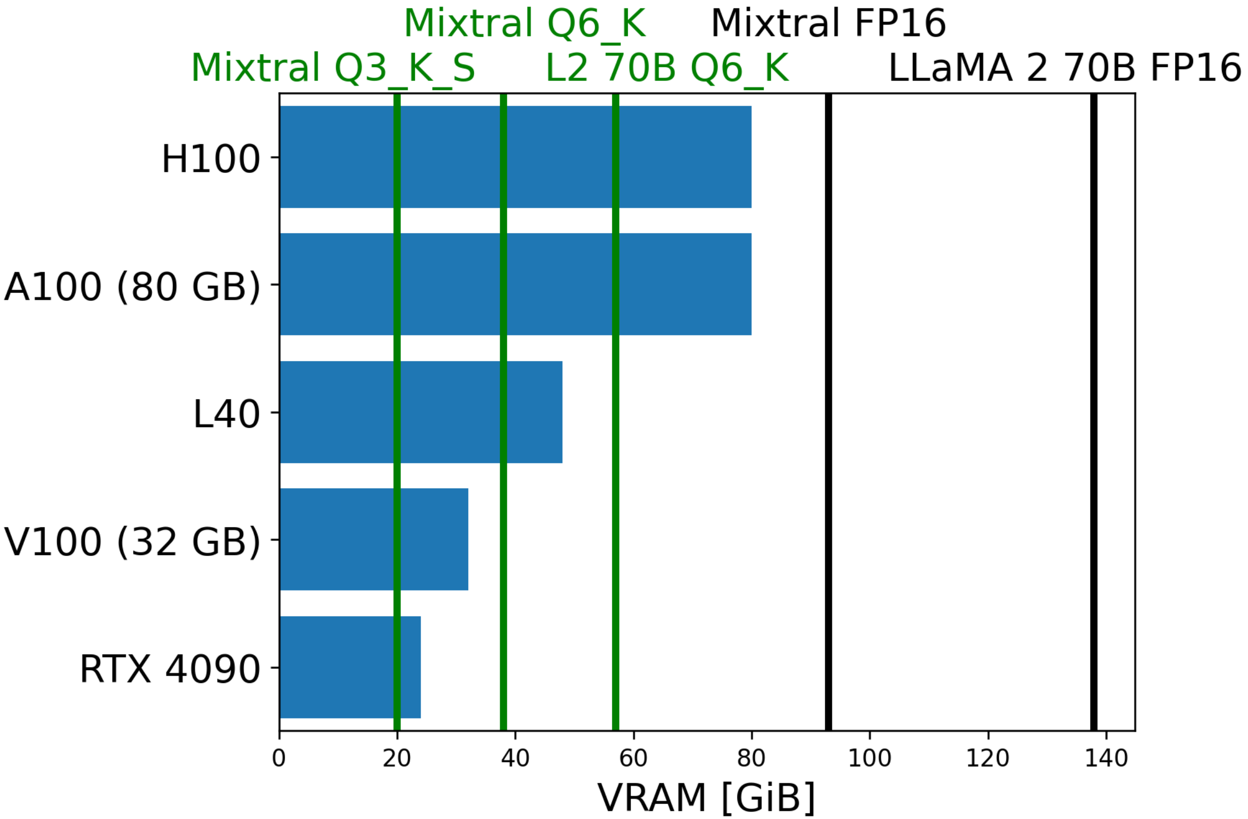
# Weight Quantization

- Original LLaMA weights are FP16
- Can be quantized to 2-8 bit ints with some quality loss
- Simple scheme: round to nearest



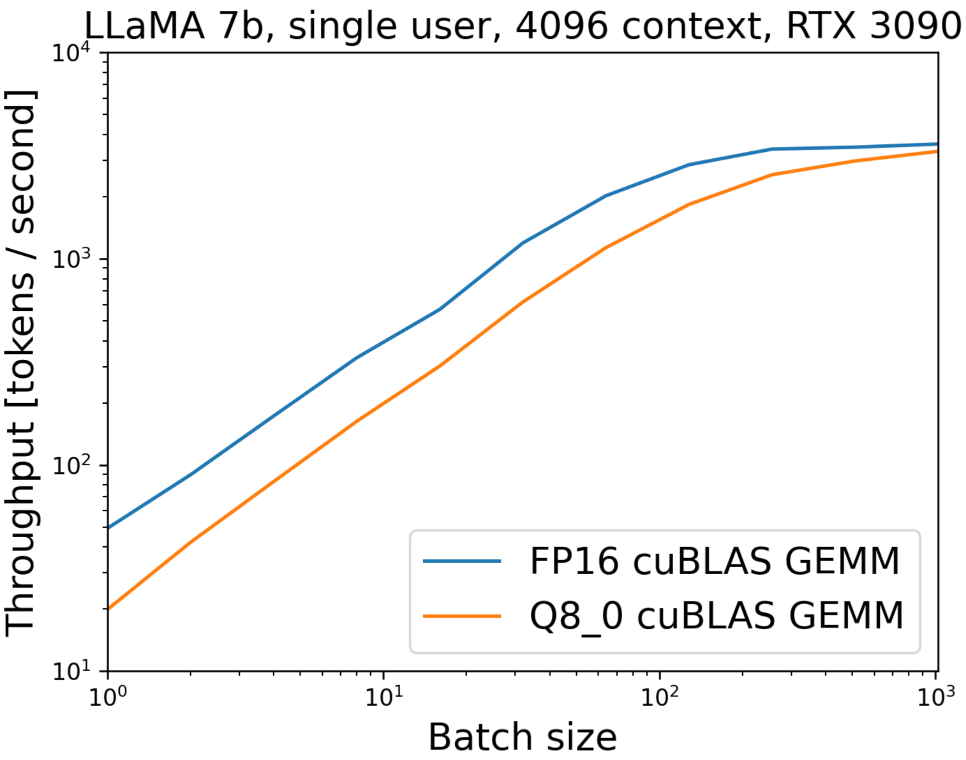
# Weight Quantization

- State-of-the-art models don’t fit on single GPU at FP16
- But quantized models do



# Simple Matrix Multiplication

- Weights as blocks of ints $q_k$ with per-block scale $d$
- FP16: $a_k = d \cdot q_k$.
- Can use FP16 GEMM
- Decoding only half as fast as FP16



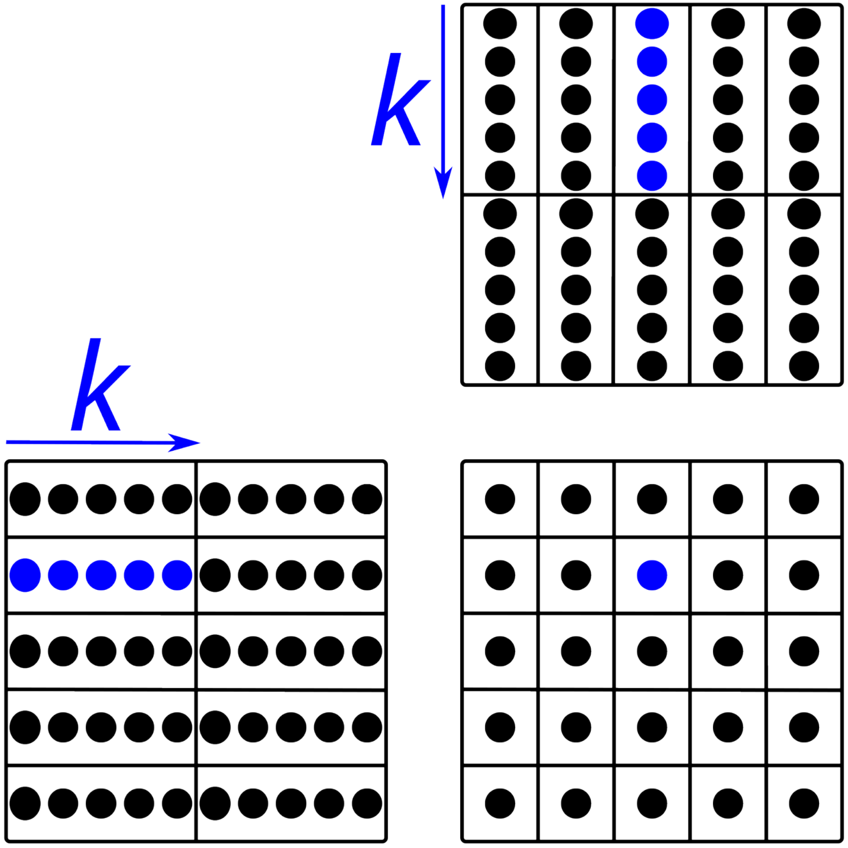
# Matrix Shape Matters

- 2x square matrix: $O(N^2)$ data, $O(N^3)$ compute
- Square matrix + vector: $O(N^2)$ data, $O(N^2)$ compute
- Encoding: batch size >> 1, compute bound
- Decoding: batch size 1, I/O bound

# Fused Matrix Multiplication

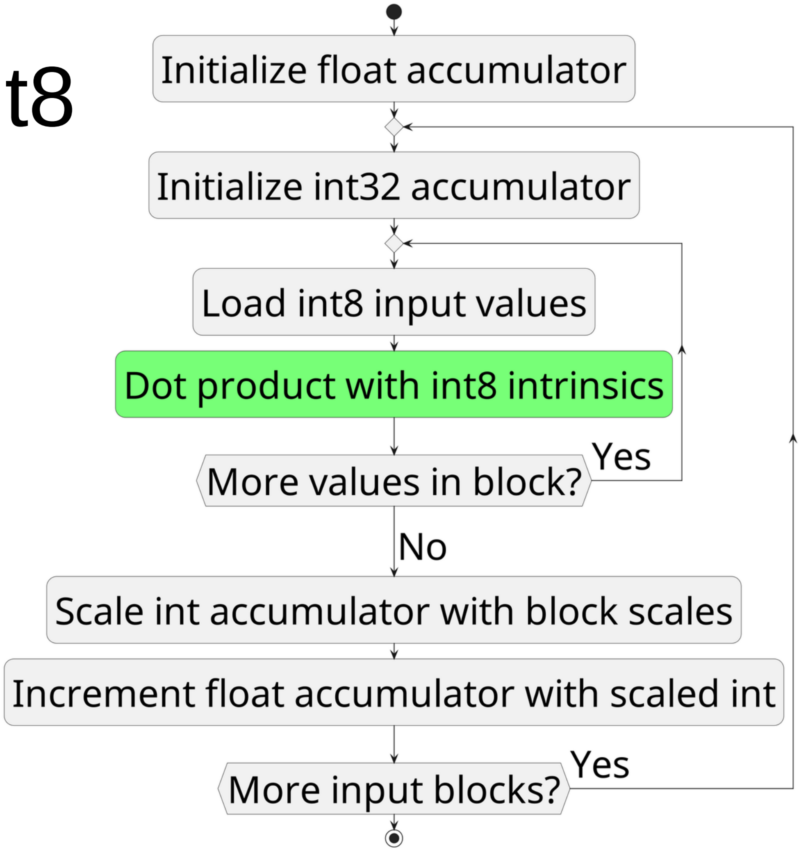
- Quantize hidden state on-the-fly
- Dot product of two quantized blocks:

$$\sum_k a_k b_k = \sum_k d_a q_{ak} d_b q_{bk}$$
$$= \underset{\text{float}}{d_a d_b} \sum_k \underset{\text{int8}}{q_{ak} q_{bk}}.$$



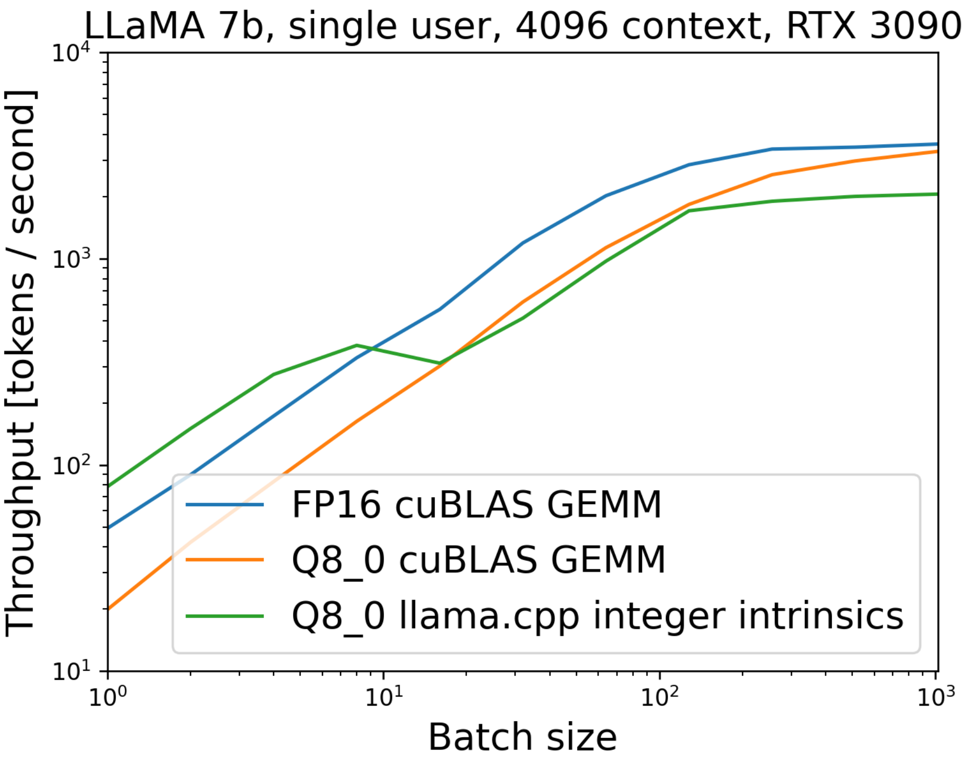
# Fused Matrix Multiplication

- Can do most operations as int8
- Faster than float
- Hardware support for int8 intrinsics since Pascal



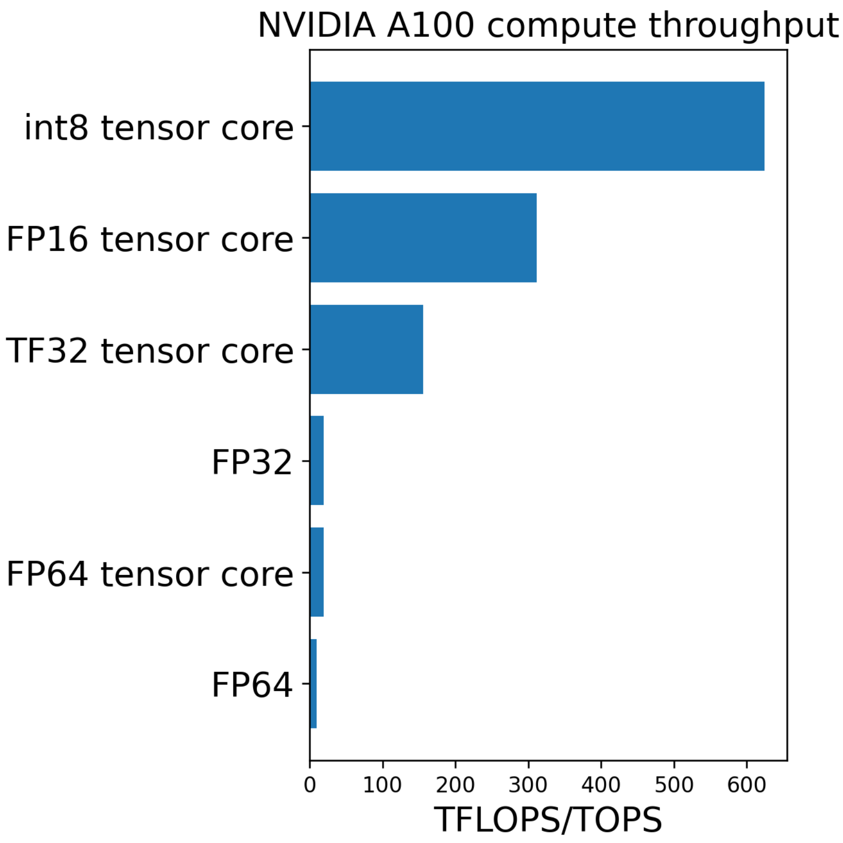
# Fused Matrix Multiplication

- Avoid dequantizing to FP16 to reduce I/O
- Dequantize + FP 16 GEMM: 8.5 bits read + 16 bits write + 16 bits read per weight
- Directly use quantized: 8.5 bits read per weight



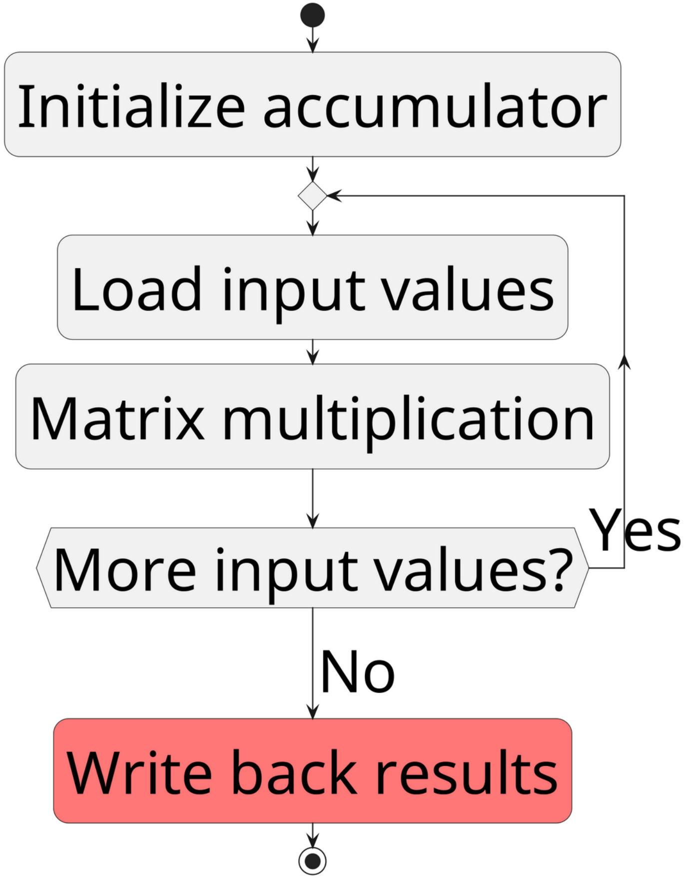
# Tensor Cores Are Fast

- int8 intrinsics only ~60% throughput of FP16 GEMM at 1024 batch size
- Reason: FP16 tensor cores
- But: int8 tensor cores 2x faster than FP16



# Tensor Core Usage Patterns

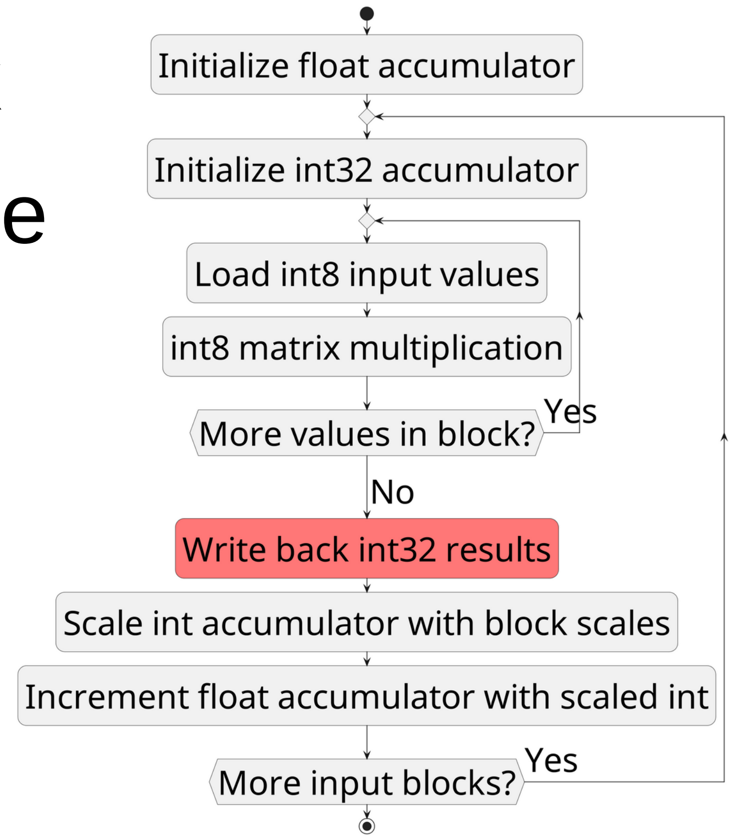
- Tensor cores are fast but restrictive
- Fast: Initialize, load data, matrix multiplication
- Slow: Write back results



# Problems With Quantized Blocks

- Slow writeback for each block
- Small blocks: bad performance
- Big blocks: bad precision

$$a_k = d \cdot q_k.$$

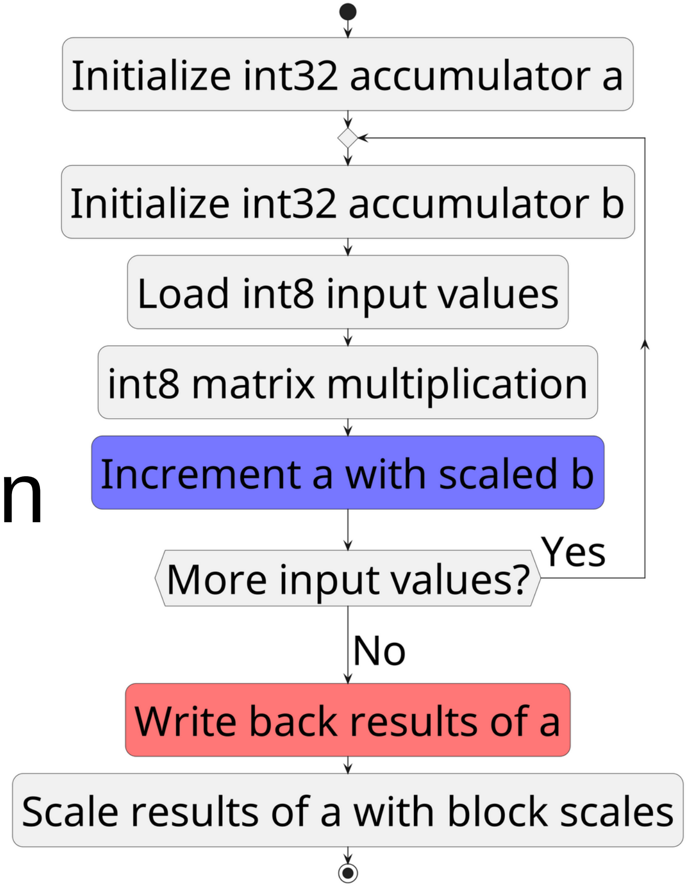


# Tensor Core Details

- CUDA organizes threads as “warps” of size 32
- Each thread holds 256/32 = 8 tensor core accumulator values
- **No guarantees** for which 8 values thread holds
- But can apply scalar operations

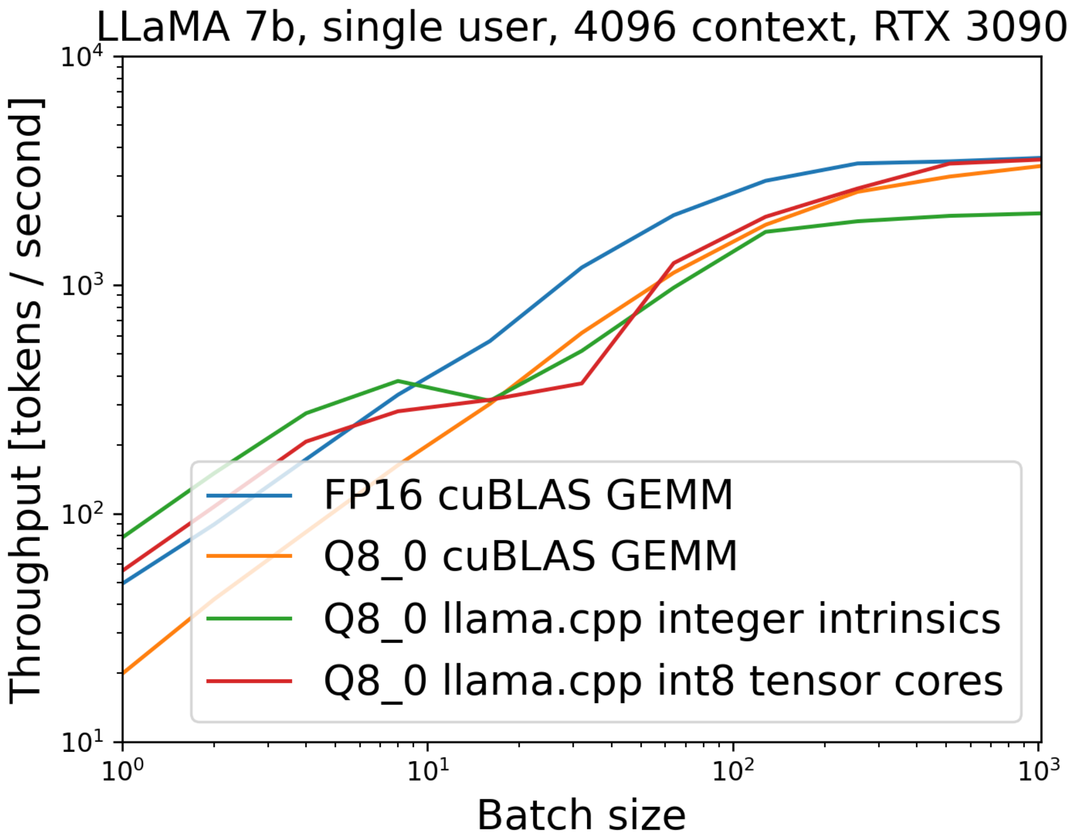
# llama.cpp int8 Tensor Cores

- Define additional scales for tensor core fragments
- Better precision for large blocks
- Single float scale per row/column
- Only need to write back results once



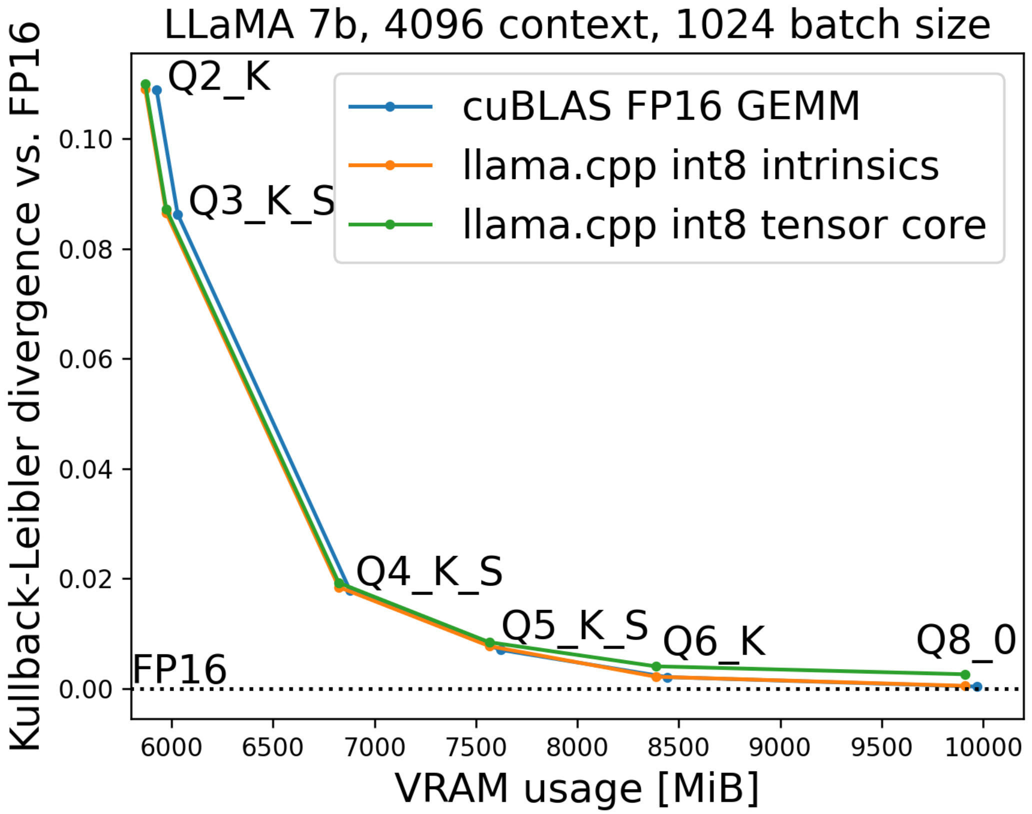
# Llama.cpp int8 Tensor Cores

- Llama.cpp int8 15% faster than cuBLAS FP16 GEMM
- 6-13% end-to-end
- Negligible precision loss



# Precision/VRAM

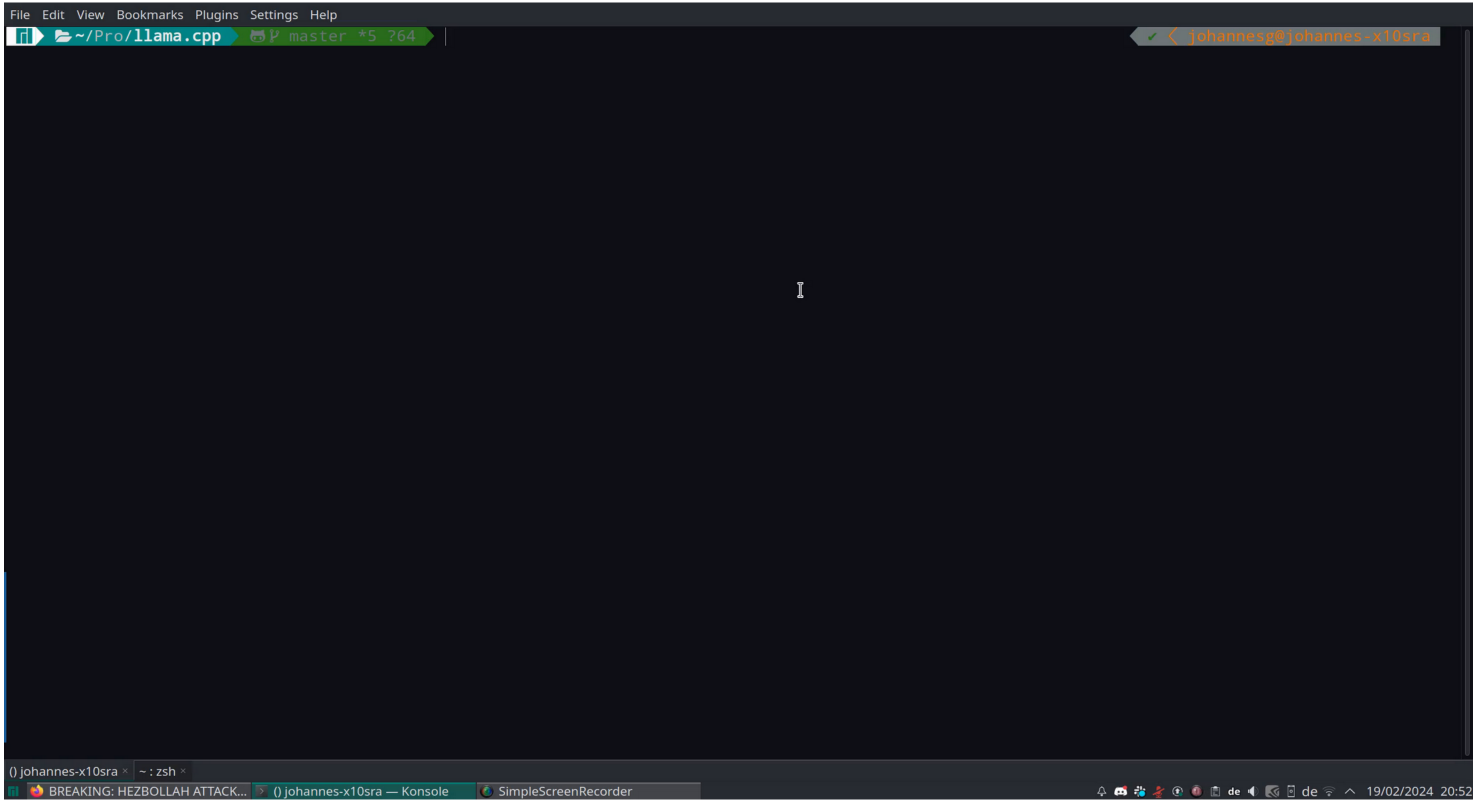
- Precision with int8 slightly worse
- But memory usage also slightly lower
- Change to top token probabilities < 1%



# Further Work

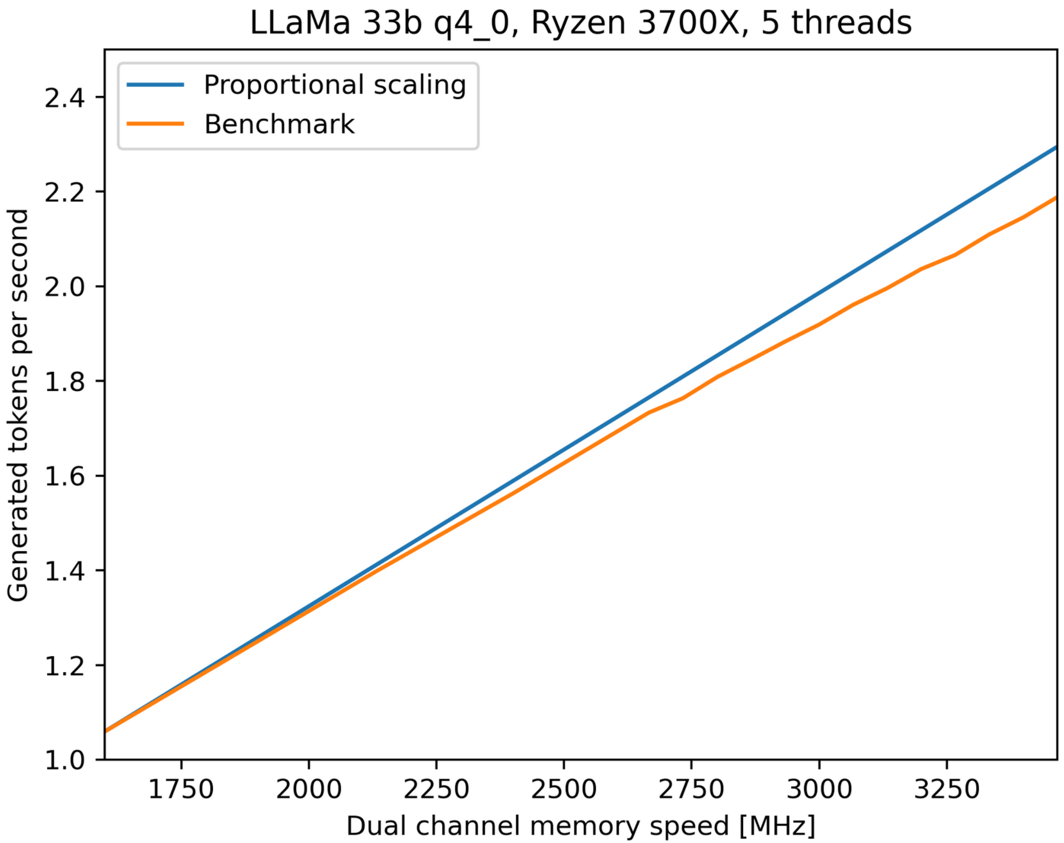
- Only ~23% int8 tensor core utilization
  => More performance optimization
- Fused operations (e.g. FlashAttention, mixture of experts)
- Implement backwards pass (or training at all)

# Live Demonstration (Hopefully)



Thank you for your attention!

# Appendix: Memory Scaling

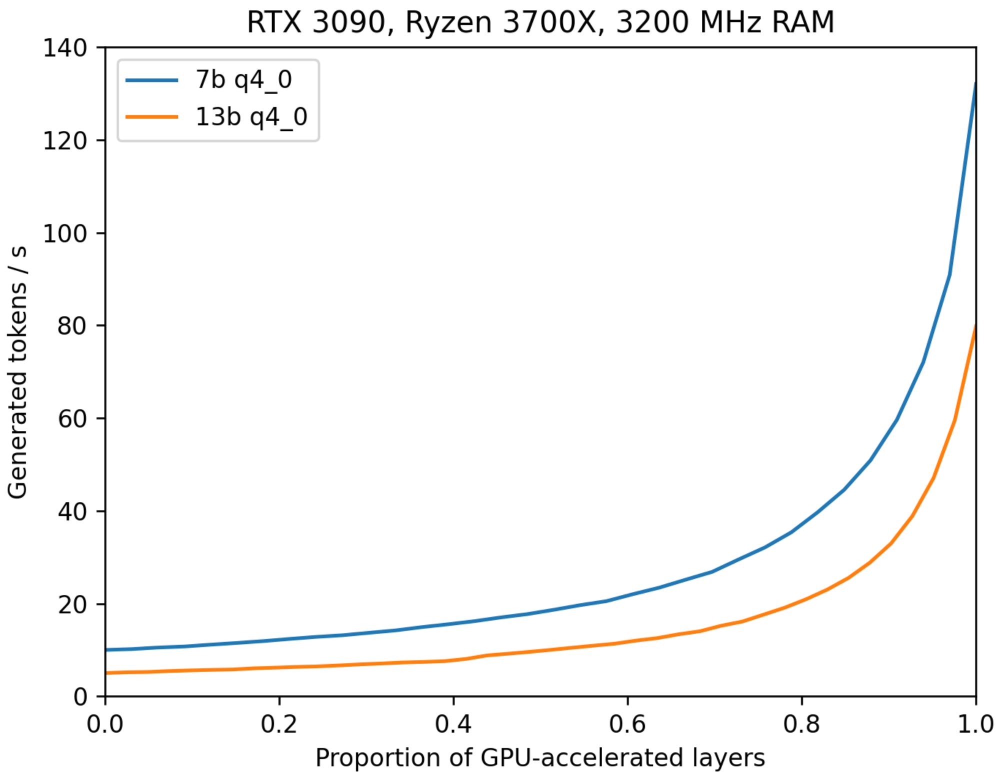


# Kullback-Leibler Divergence

- Measures difference between probability distributions $P$ and $Q$
- Definition:

$$D_{\mathrm{KL}}(P \parallel Q) = \sum_x P(x) \ \log\left(\frac{P(x)}{Q(x)}\right).$$

# Appendix: CPU+GPU Scaling



# Appendix: CPU+GPU Scaling