

Lowbit kernels

Introduction

- We developed low-bit (1-8 bit) operators for ARM CPU:
 - Linear operators with 8-bit dynamically quantized activations and low-bit weights (1-8 bit)
 - Embedding operators with low-bit weights (1-8 bit)
- The operators work across all PyTorch surfaces, including eager, torch.compile, AOTI, and ExecuTorch, and are [available to use in torchchat](#) and [ExecuTorch](#)

Quantization

- In affine quantization, we represent a group of FP32 weights w_1, \dots, w_n with x -bit integers q_1, \dots, q_n , an FP32 scale s , and an optional x -bit zero z :
 - $w_i = s * (q_i - z)$ [with zero]
 - $w_i = s * q_i$ [without zero]

$$\begin{bmatrix} -6.6 \\ -2.2 \\ 1.1 \\ -1.1 \end{bmatrix} = 1.1 \times \left(\begin{bmatrix} -4 \\ 0 \\ 3 \\ 1 \end{bmatrix} - 2 \right)$$

Low-bit operators work in quantized domain to use less memory

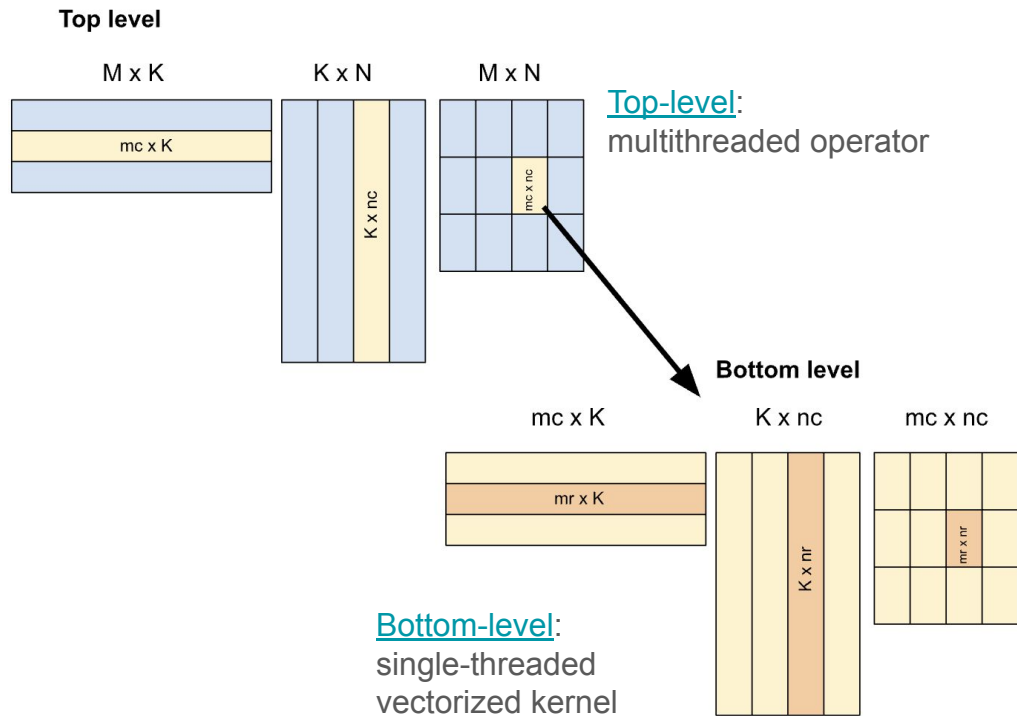
Quantized dot product

- The dot product between weights and activations in FP32 domain can be computed by a dot product in intx domain (integer arithmetic)

$$\begin{aligned} v^a \cdot v^w &= \sum_{i=1}^k v_i^a v_i^w \\ &= \sum_{g=1}^G \sum_{i:g(i)=g} v_i^a v_i^w \\ &= \sum_{g=1}^G s^a s_g^w \sum_{i:g(i)=g} [q_i^a q_i^w - q_i^a z_g^w - z^a q_i^w + z^a z_g^w] \\ &= \sum_{g=1}^G s^a s_g^w \left[\left(\sum_{i:g(i)=g} q_i^a q_i^w \right) - z_g^w \left(\sum_{i:g(i)=g} q_i^a \right) - z^a \left(\sum_{i:g(i)=g} q_i^w \right) + G z^a z_g^w \right] \end{aligned}$$

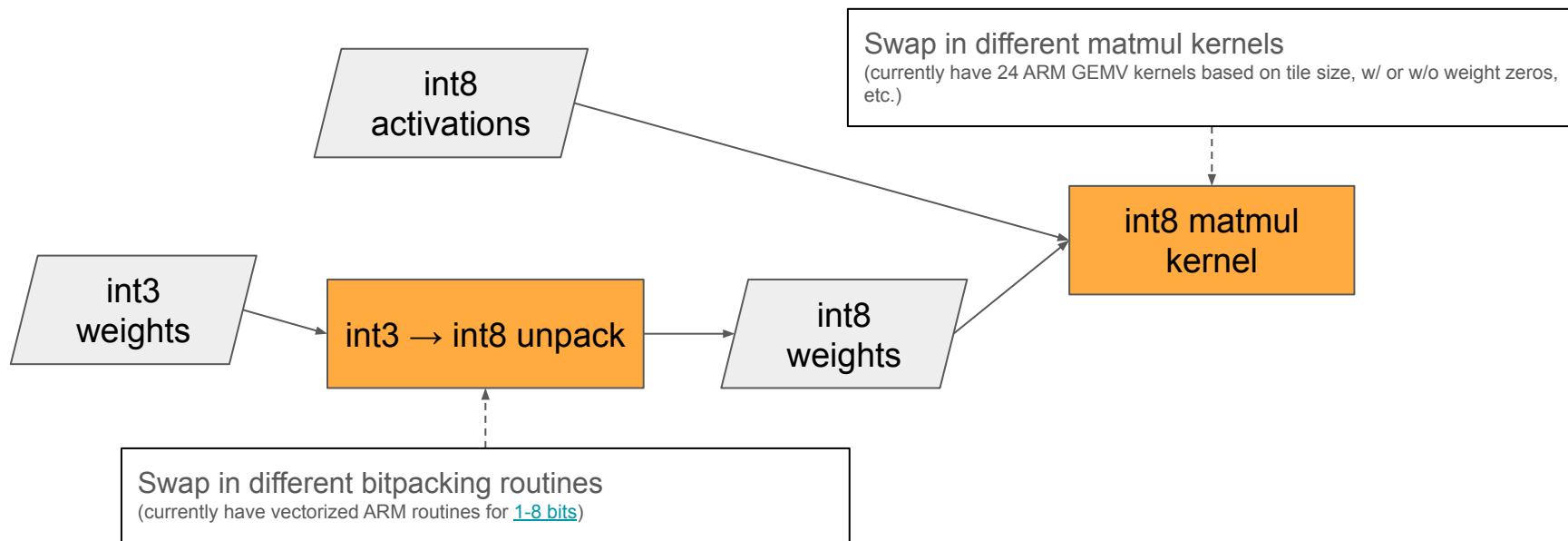
Operator design

- The “top-level” multi-threaded linear operator is agnostic to bottom-level single-threaded kernels, allowing **third-party vendors to swap in** their own kernels
- The interface between the operator and kernel is defined by a [ukernel config](#), which specifies kernel function pointers for preparing activation data, preparing weight data, and running the kernel.



















































Universal kernel design

As an example low-level kernel that uses the operator framework, we implemented universal low-bit GEMV kernels for ARM CPUs. These universal kernels work for bitwidths 1-8.



Thinking about bitpacking

Pack 8 3-bit values into 3 bytes

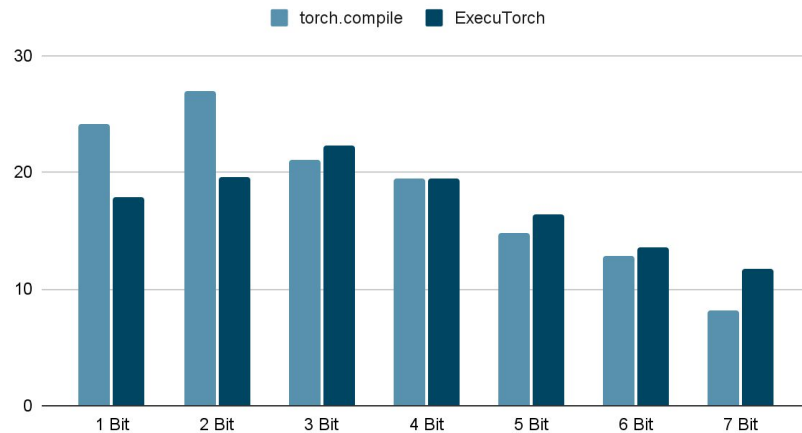
	Format 1								Format 2							
3-bit																
																
																
SHIFTs	13								7							
ANDs	15								10							
ORs	8								2							

- Unpacking format 2 is 1.60x faster than format 1
- Using format 2 improves end-to-end LLM decode performance by 1.23x

Performance

Bitwidth x	torch.compile (Decode tokens/sec)	ExecuTorch (Decode tokens/sec)	ExecuTorch PTE size (GiB)
1	24.18	17.86	1.46
2	27.02	19.65	2.46
3	21.01	22.25	3.46
4	19.51	19.47	4.47
5	14.78	16.34	5.47
6	12.80	13.61	6.47
7	8.16	11.73	7.48

Llama3.1 8B decode (tokens/sec)



Measurements were done with torchchat using 6 threads on an M1 Mac Pro with 8 performance cores, 2 efficiency cores, and 32GB RAM.

Shared code between PyTorch and ExecuTorch

- We wanted to share code between PyTorch and ExecuTorch, but PyTorch and ExecuTorch do not share operator registration, tensors, or parallel compute.
- To share code,
 - Kernels were implemented with [using raw pointers instead of PyTorch tensors](#)
 - We implemented the [linear operator in a header](#) that is included in separate [PyTorch](#) and [ExecuTorch](#) operator registration code
 - For multi-threaded compute, we introduced [torchao::parallel_1d](#), which compiles to either [at::parallel_for](#) or [ExecuTorch's threadpool](#) based on compile-time flags.

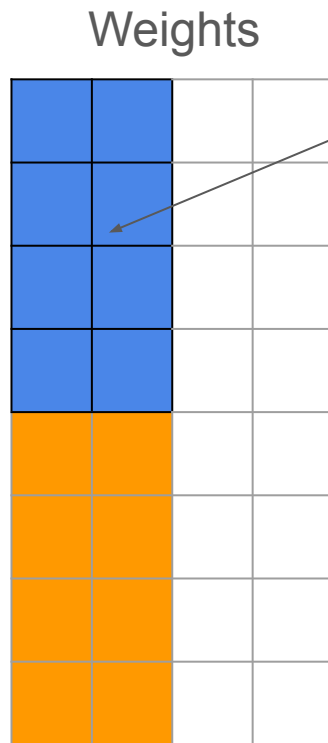
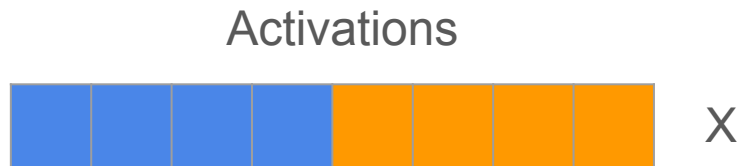
Potential Next steps

- Add low-bit kernels for other CPU ISAs like x86, and accelerators like Apple Metal
- Add universal low-bit GEMM kernels for ARM CPU, reusing the same bitpacking routines from the universal GEMV kernels.
- Add kernels for look-up table based quantization schemes.
- Integrate third-party libraries like KleidiAI with the operator framework.
- Improve runtime selection of [ukernel configs](#) based on ISA, packing format, and activation shape.

Valpacking

Repeat for each 2 weight columns:

1. Load **4 blue activations** and **8 blue weights**
2. Compute **2 blue dot products**
3. Load **4 orange activations** and **8 orange weights**
4. **Compute 2 orange dot products**



8 blue weights are stored contiguously in memory (matrix not stored in row-major or column major format)

=

