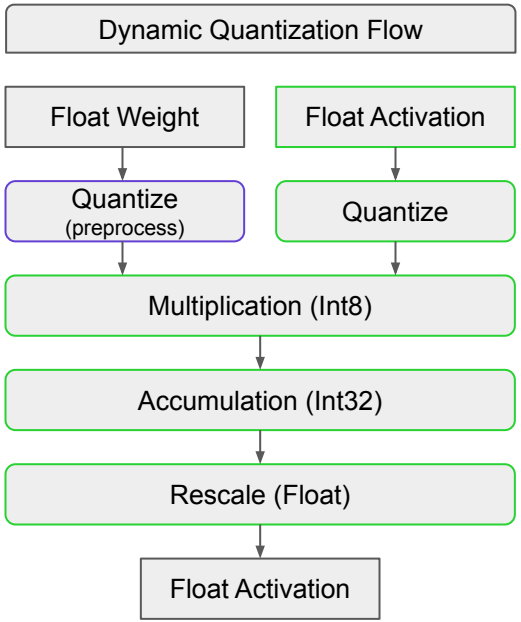
Quantization Cuda vs Triton

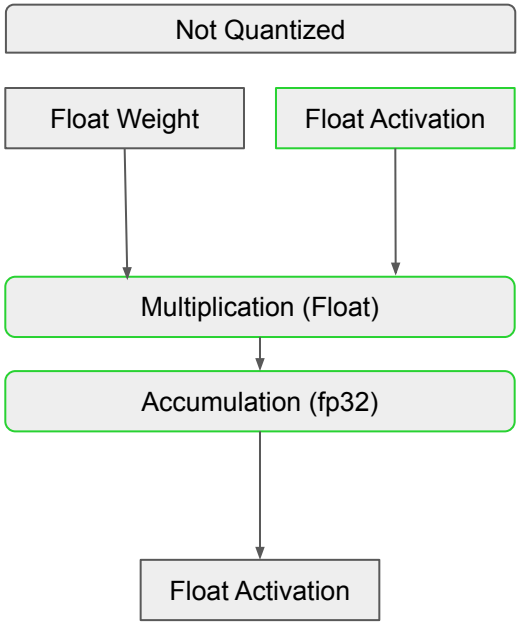
Charles Hernandez
HDCharles

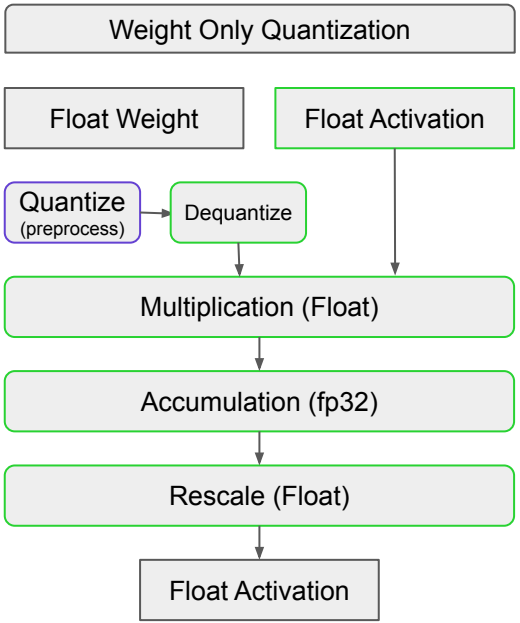
Background

- Pytorch Core
  - AO (Architecture Optimization) Team
    - Quantization
    - Pruning
- My Recent Focus
  - GPU Quantization
    - Segment-anything-fast, gpt-fast, sdxl-fast
    - TorchAO - https://github.com/pytorch-labs/ao
      - Int8 Dynamic Quantization
        - i8i8->i32 vs i8i8bf16->bf16
      - Int 8 Weight Only Quantization
        - bf16i8->bf16
      - Int 4 Weight Only Quantization
        - bf16i4->bf16

Techniques


Dynamic Quantization Flow
Float Weight
Float Activation
Quantize
(preprocess)
Quantize
Multiplication (Int8)
Accumulation (Int32)
Rescale (Float)
Float Activation


Not Quantized
Float Weight
Float Activation
Multiplication (Float)
Accumulation (fp32)
Float Activation


Weight Only Quantization
Float Weight
Float Activation
Quantize
(preprocess)
Dequantize
Multiplication (Float)
Accumulation (fp32)
Rescale (Float)
Float Activation

Quantization
Cuda vs Triton

01 Dynamic Quantization

02 Int8 Weight Only Quantization

03 Int4 Weight Only Quantization

Quantization
Cuda vs Triton

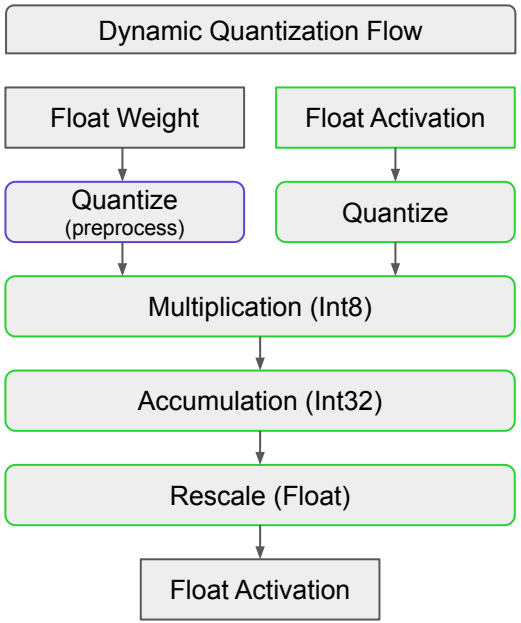
01 Dynamic Quantization

02 Int8 Weight Only Quantization

03 Int4 Weight Only Quantization

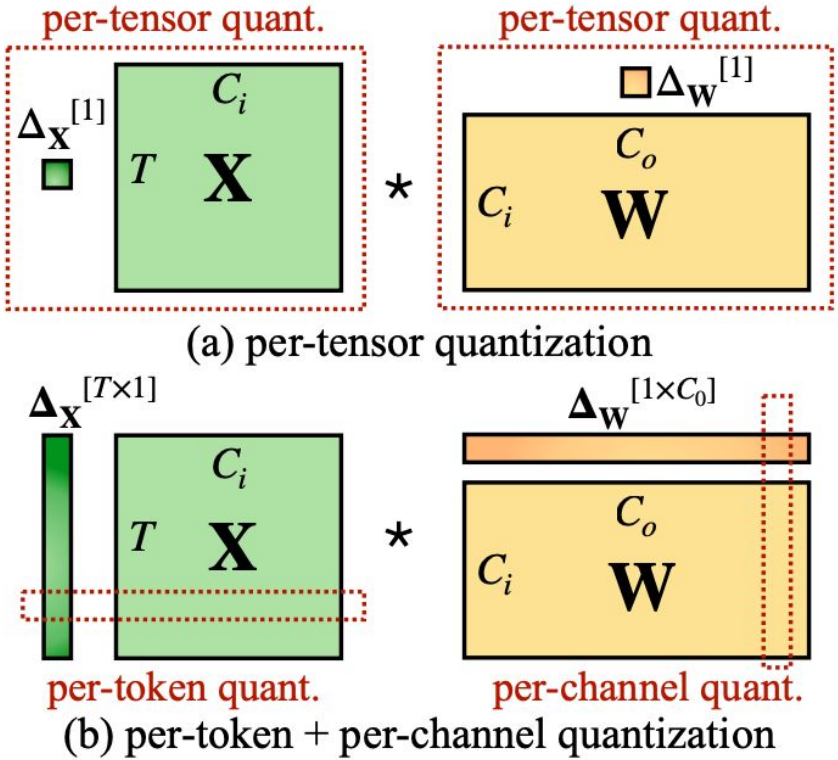
Dynamic Quantization

$$Y = X.W$$
$$Y = (Sx * Xint).(Wint * Sw)$$
$$Y = Sx * (Xint.Wint) * Sw$$



Dynamic Quantization

$$Y = X.W$$
$$Y = (Sx * Xint).(Wint * Sw)$$
$$Y = Sx * (Xint.Wint) * Sw$$



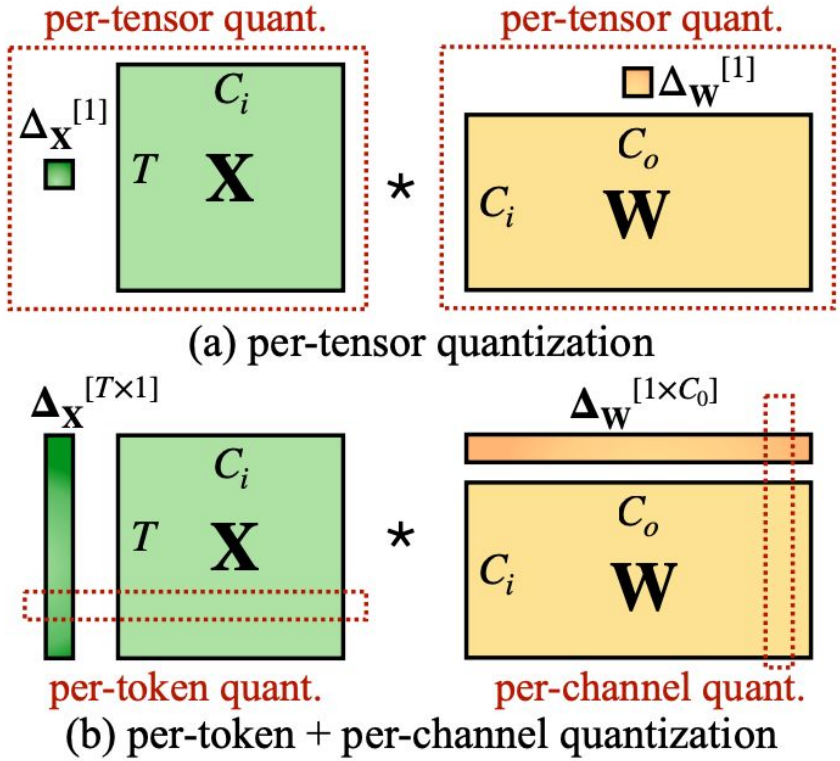
Dynamic Quantization

Y=X.W
Y = (Sx*Xint).(Wint * Sw)
Y = Sx * (Xint.Wint) * Sw

Y = Sx * (XWint) * Sw

https://triton-lang.org/main/getting-started/tutorials/03-matrix-multiplication.html

SAM, vit_h, bsz=16		
technique	**runtime (ms)**	**peak memory**
no quantization	785.313	15.279
dynamic quantization	731.649	18.631



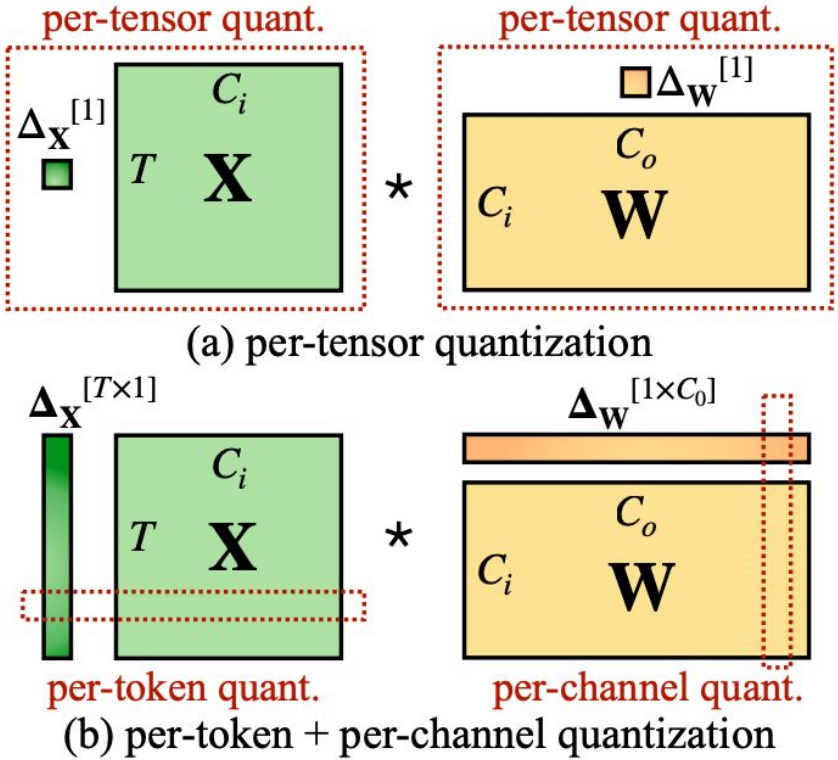
Dynamic Quantization

int8
bf16
int32

Y=X.W
Y = (Sx*Xint).(Wint * Sw)
Y = Sx * (Xint.Wint) * Sw

Y = Sx * (XWint) * Sw

SAM, vit_h, bsz=16		
technique	**runtime (ms)**	**peak memory**
no quantization	785.313	15.279
dynamic quantization	721.649	18.631



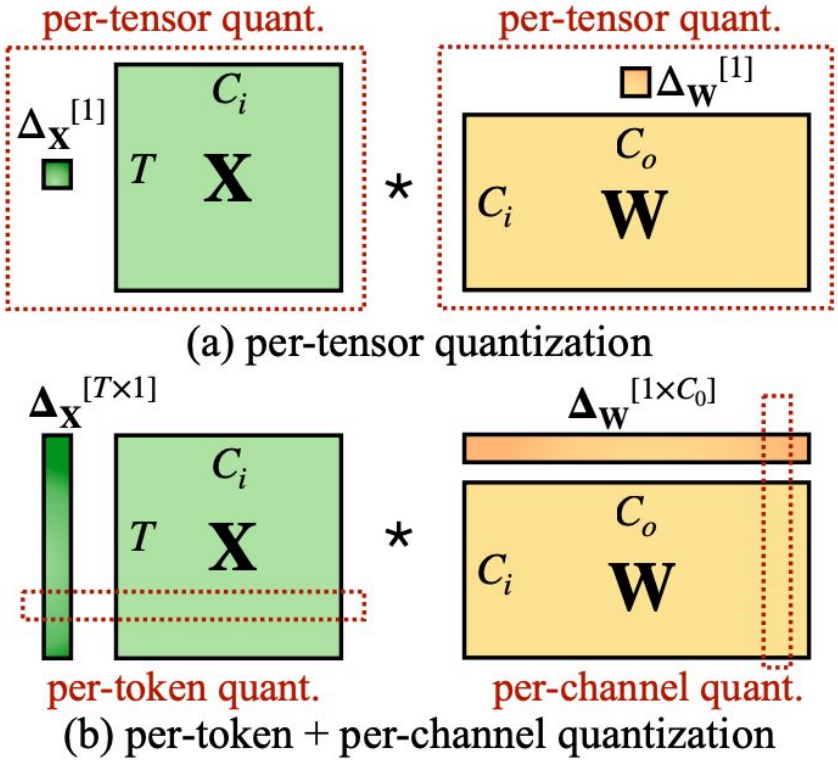
Dynamic Quantization

int8
bf16
int32

Y=X.W
Y = (Sx*Xint).(Wint * Sw)
Y = Sx * (Xint.Wint) * Sw

~~Y = Sx * (XWint) * Sw~~
Sx * (Xint.Wint * Sw)
Sx * (XWrescaled)

SAM, vit_h, bsz=16		
technique	**runtime (ms)**	**peak memory**
no quantization	785.313	15.279
dynamic quantization	731.649	18.631
dynamic quantization with fusion	695.115	14.941



```python
# This op is a special case of the int_mm op which we use based on the pattern
# _int_mm -> mul (defined in ../fx_passes/post_grad.py) in order to prevent
# realization of the int32 _int_mm output by forcing fusion with the mul op.
# This is only used when config.force_fuse_int_mm_with_mul = True
def tuned_fused_int_mm_mul(mat1, mat2, mat3, out_dtype, *, layout=None):
    out_dtype = (
        torch.promote_types(mat3.get_dtype(), torch.int32)
        if out_dtype is None
        else out_dtype
    )
    m, n, k, layout, mat1, mat2, mat3 = mm_args(
        mat1, mat2, mat3, layout=layout, out_dtype=out_dtype
    )
    choices: List[Dict[Any, Any]] = []
    for config in int8_mm_configs(m, n, k):
        mm_template.maybe_append_choice(
            choices,
            input_nodes=(mat1, mat2, mat3),
            layout=layout,
            **dict(mm_options(config, m, n, k, layout), ACC_TYPE="tl.int32"),
            suffix_args=1,
            epilogue_fn=V.ops.mul,
        )
    return autotune_select_algorithm("int_mm", choices, [mat1, mat2, mat3], layout)
```

(Forced epilogue)

https://github.com/pytorch/pytorch/blob/main/torch/_inductor/kernel/mm.py#L295

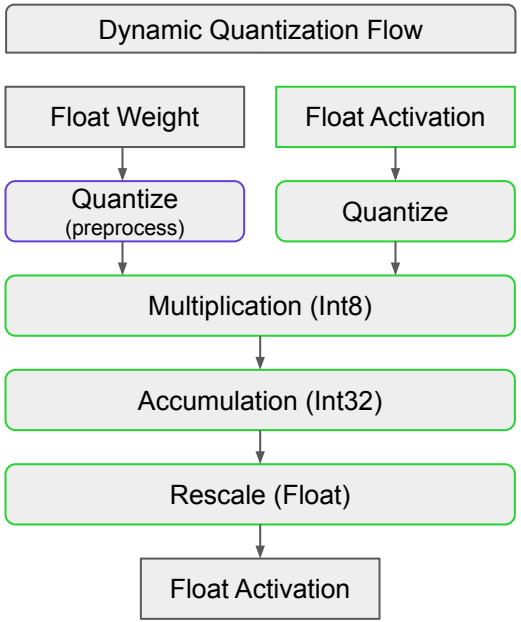
Quantization
Cuda vs Triton

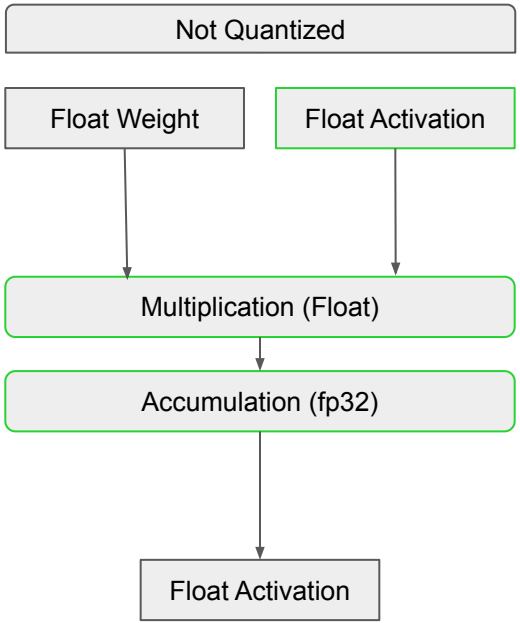
01 Dynamic Quantization

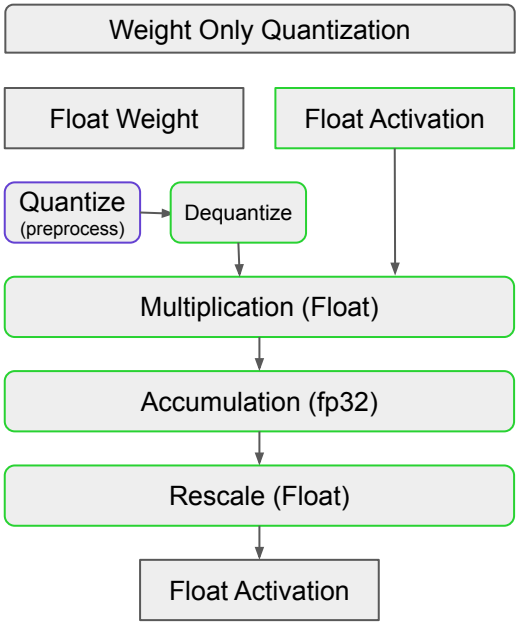
02 Int8 Weight Only Quantization

03 Int4 Weight Only Quantization

Techniques


Dynamic Quantization Flow
Float Weight
Float Activation
Quantize
(preprocess)
Quantize
Multiplication (Int8)
Accumulation (Int32)
Rescale (Float)
Float Activation


Not Quantized
Float Weight
Float Activation
Multiplication (Float)
Accumulation (fp32)
Float Activation

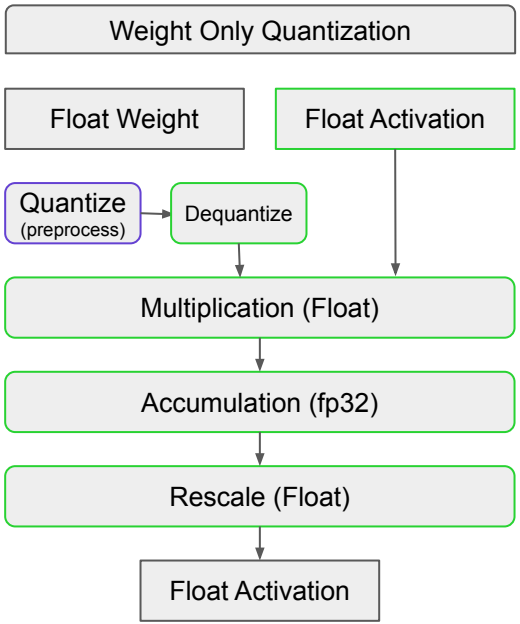

Weight Only Quantization
Float Weight
Float Activation
Quantize
(preprocess)
Dequantize
Multiplication (Float)
Accumulation (fp32)
Rescale (Float)
Float Activation

Int8 Weight Only Quantization

$$Y = X.W$$
$$Y = X.(Wint * Sw)$$
$$Y = (X.Wint) * Sw$$

https://triton-lang.org/main/getting-started/tutorials/03-matrix-multiplication.html

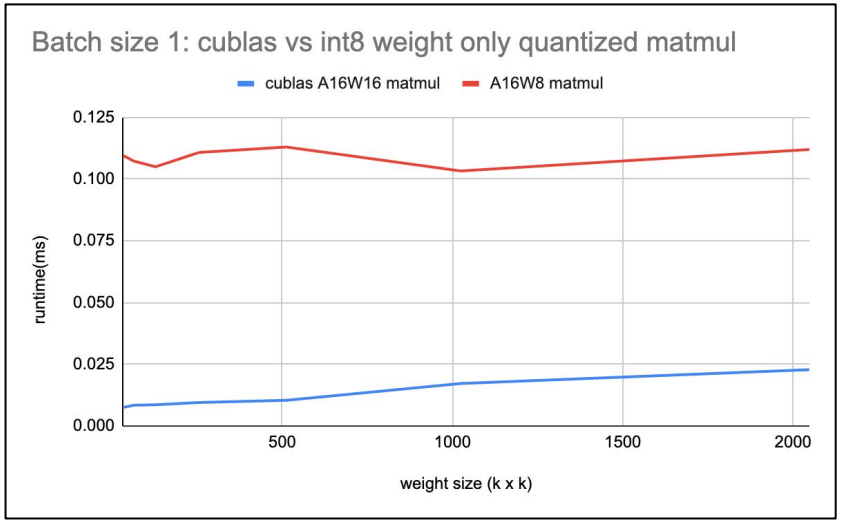
https://github.com/pytorch/pytorch/blob/main/torch/_inductor/kernel/mm.py#L76



Int8 Weight Only Quantization

Once again the kernel performs poorer than expected:

LLaMA-7B, bsz=1	
technique	**perf (tokens/s)**
no quantization	93.08
int8 weight-only quant	40.59



Int8 Weight Only Quantization

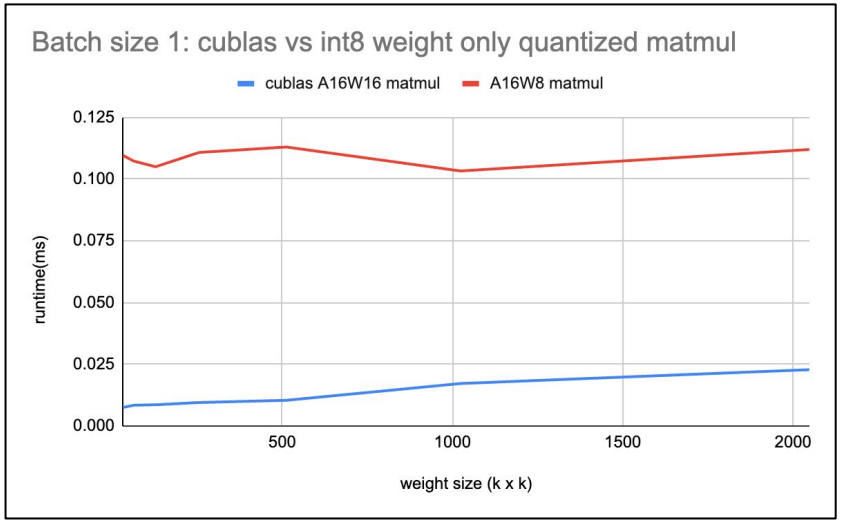
Why is the kernel so slow?

1) Its doing more work than the base matmul

```
    a = tl.load(A, mask=rk[None, :] < k, other=0.)
    b = tl.load(B, mask=rk[:, None] < k, other=0.)
if B_PROLOGUE_CAST_TYPE is not None:
    b = b.to(B_PROLOGUE_CAST_TYPE)
acc += tl.dot(a, b, allow_tf32=ALLOW_TF32)
```

2) The Blocksize is limited to be >= 16, meaning

The grid for this launch is configured to execute only 64 blocks, which is less than the GPU's 108 multiprocessors. This can underutilize some multiprocessors. If you do not intend to execute this kernel



Int8 Weight Only Quantization

https://github.com/pytorch/pytorch/blob/main/torch/_inductor/decomposition.py#L231

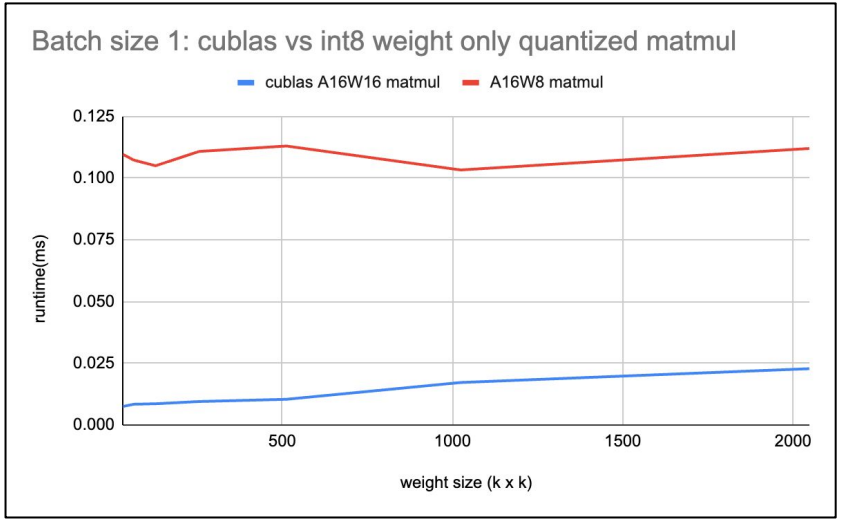
However this is a problem that can be solved by torch.compile:

1) Its doing more work than the base matmul

```
    a = tl.load(A, mask=rk[None, :] < k, other=0.)
    b = tl.load(B, mask=rk[:, None] < k, other=0.)
if B_PROLOGUE_CAST_TYPE is not None:
    b = b.to(B_PROLOGUE_CAST_TYPE)
acc += tl.dot(a, b, allow_tf32=ALLOW_TF32)
```

2) The Blocksize is limited to be >= 16, meaning

The grid for this launch is configured to execute only 64 blocks, which is less than the GPU's 108 multiprocessors. This can underutilize some multiprocessors. If you do not intend to execute this kernel



```
def triton_(in_ptr0, in_ptr1, in_ptr2, in_ptr3, out_ptr1, xnumel, rnumel, XBLOCK : tl.constexpr, RBLOCK
    xnumel = 4096
    rnumel = 4096
    xoffset = tl.program_id(0) * XBLOCK
    xindex = xoffset + tl.arange(0, XBLOCK)[:, None]
    xmask = xindex < xnumel
    rbase = tl.arange(0, RBLOCK)[None, :]
    x0 = xindex
    _tmp6 = tl.full([XBLOCK, RBLOCK], 0, tl.float32)
    for roffset in range(0, rnumel, RBLOCK):
        rindex = roffset + rbase
        rmask = rindex < rnumel
        r1 = rindex
        tmp0 = tl.load(in_ptr0 + (r1), None, eviction_policy='evict_last').to(tl.float32)
        tmp2 = tl.load(in_ptr1 + (r1 + (4096*x0)), xmask, eviction_policy='evict_first', other=0.0)
        tmp1 = tmp0.to(tl.float32)
        tmp3 = tmp2.to(tl.float32)
        tmp4 = tmp1 * tmp3
        tmp5 = tl.broadcast_to(tmp4, [XBLOCK, RBLOCK])
        tmp7 = _tmp6 + tmp5
        _tmp6 = tl.where(xmask, tmp7, _tmp6)
    tmp6 = tl.sum(_tmp6, 1)[:, None]
    tmp9 = tl.load(in_ptr2 + (x0), xmask, eviction_policy='evict_last').to(tl.float32)
    tmp11 = tl.load(in_ptr3 + (x0), xmask, eviction_policy='evict_last').to(tl.float32)
    tmp8 = tmp6.to(tl.float32)
    tmp10 = tmp8 * tmp9
    tmp12 = tmp10 + tmp11
    tl.store(out_ptr1 + (x0), tmp12, xmask)
```

X corresponds to the N dimension
R corresponds to the K dimension

XBlock is always 1 so each program_id processes a single value of the output

Load full activation tensor (in fp32)

Load a column of weight

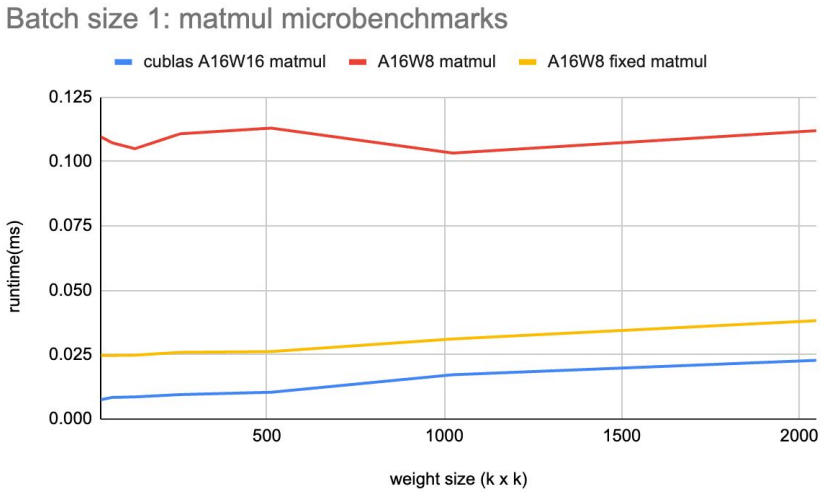
Convert weight column to fp32

Int8 Weight Only Quantization

LLaMA-7B, bsz=1	
technique	**perf (tokens/s)**
no quantization	93.08
int8 weight-only quant	40.59
int8 weight-only quant opt	135.01

However this is a problem that can be solved by torch.compile:

- Note we're still not matching default bf16 performance for the microbenchmarks even though we're much closer.
  - This is mostly due to torch.compile overhead and goes away in e2e tests
- In this case we ran directly into some of the limitations of triton but were able to route around them by avoiding using tensor cores.
- We still lack a performant kernel for bsz>1



Quantization
Cuda vs Triton

01 Dynamic Quantization

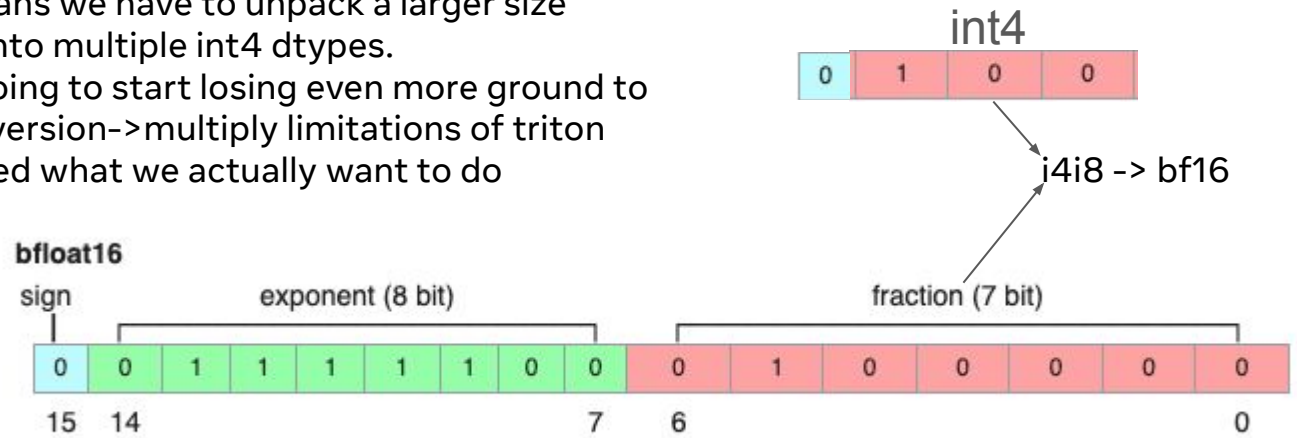
02 Int8 Weight Only Quantization

03 Int4 Weight Only Quantization

Int4 Weight Only Quantization

Int4 quantization brings new complications compared to int8

- Currently no native int4/uint4 torch dtype
- This means we have to unpack a larger size tensor into multiple int4 dtypes.
- We're going to start losing even more ground to the conversion->multiply limitations of triton compared what we actually want to do



Int4 Weight Only Quantization

Int4 quantization brings new complications compared to int8

- Currently no native int4/uint4 torch dtype
- This means we have to unpack a larger size tensor into multiple int4 dtypes.
- We're going to start losing even more ground to the conversion->multiply limitations of triton compared what we actually want to do

But we can see how far we can get with just triton

int4x2

AB	CD
EF	GH

We can pack/unpack several different ways

A	C	B	D
E	G	G	H

A	B	C	D
E	G	G	H

A	C
E	G
B	D
G	H

A	C
B	D
E	G
G	H

Int4 Weight Only Quantization

- If we're doing a matmul, since this will be our weight, we'd like information that is contiguous in the int4x2 to be contiguous in the unpacked form.
- This means using one of the two rightmost options.
- Since our matmul implementations tend to have a single thread process all of K, we chose the bottom right option to avoid situations where A thread has to load unnecessary data due to the packing.

int4x2

AB	CD
EF	GH

We can pack/unpack several different ways

A	C	B	D
E	G	G	H

A	B	C	D
E	G	G	H

A	C
E	G
B	D
G	H

A	C
B	D
E	G
G	H

Int4 Weight Only Quantization

- If we're doing a matmul, since this will be our weight, we'd like information that is contiguous in the int4x2 to be contiguous in the unpacked form.
- This means using one of the two rightmost options.
- Since our matmul implementations tend to have a single thread process all of K, we chose the bottom right option to avoid situations where A thread has to load unnecessary data due to the packing.
- We can pack/unpack a uint8 and int4 like so

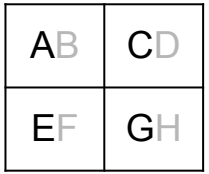
  int4[2*k,n]=(uint4x2[k,n] & 0xF) - 8
  int4[2*k+1,n]=(uint4x2[k,n] >> 4) - 8

  (15 in binary)

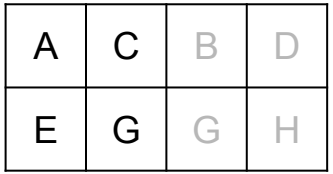
- We chose uint8 over int8 because triton's bit shifts are broken for int8

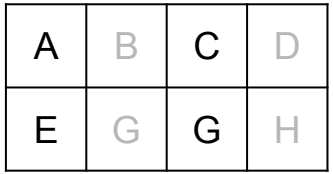
https://github.com/pytorch/pytorch/blob/main/torch/_inductor/kernel/unpack_mixed_mm.py#L9

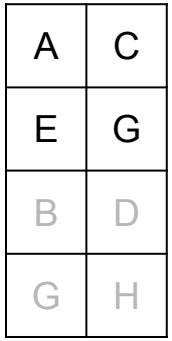
int4x2

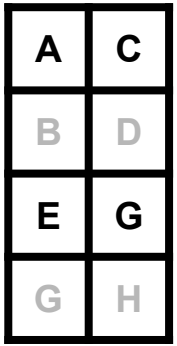


We can pack/unpack several different ways









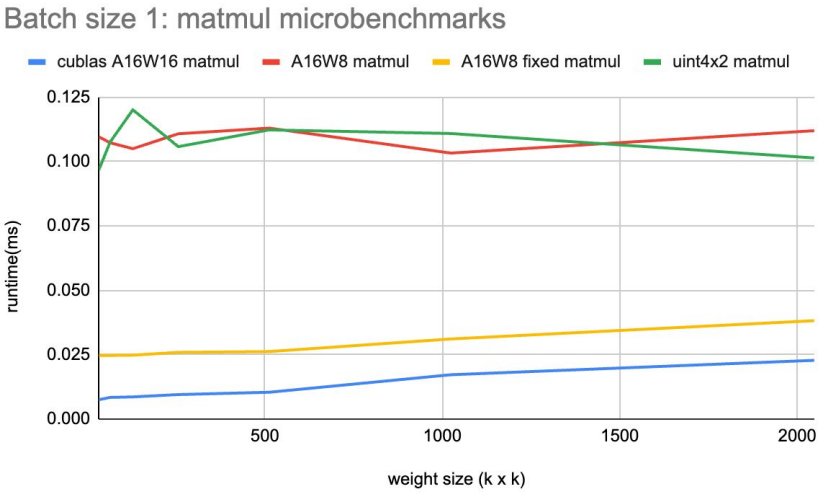
Int4 Weight Only Quantization

LLaMA-7B, bsz=1	
technique	**perf (tokens/s)**
no quantization	93.08
int8 weight-only quant	40.59
int8 weight-only quant opt	135.01
uint4x2 weight-only quant	43.59

Once again the kernel performs poorer than expected:

- Instead of being 4x faster than the no quantization case, uint4x2 quantization is ½ as fast.
- If I were to try this today, i would write the kernel similar to the fast int8 kernel rather than the slow int8 kernel
- That direction of investigation was halted due to Jeff Johnson (the person who wrote the PyTorch GPU backend) developing an int4 kernel that made its way to pytorch

https://github.com/pytorch/pytorch/blob/main/aten/src/ATen/native/native_functions.yaml#L4097



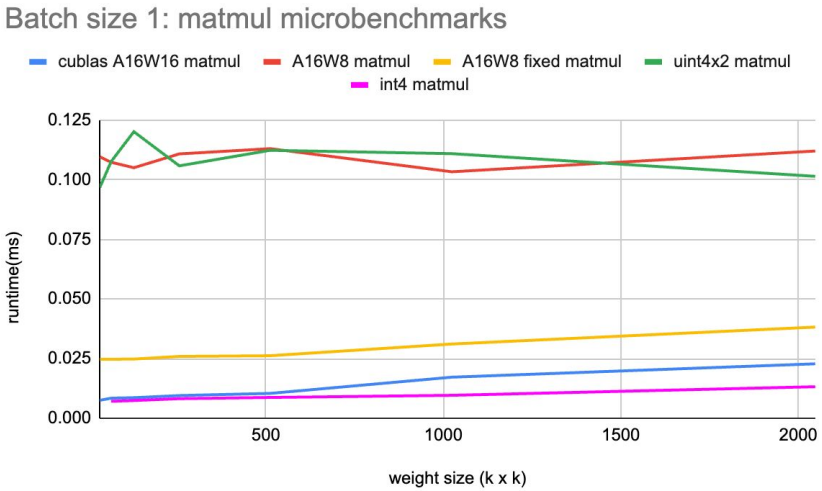
Int4 Weight Only Quantization

Once again the kernel performs poorer than expected:

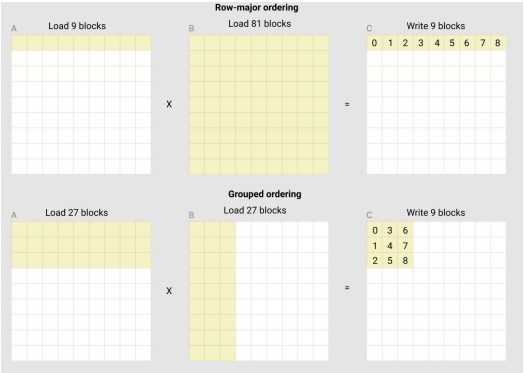
- Instead of being 4x faster than the no quantization case, uint4x2 quantization is ½ as fast.
- If I were to try this today, i would write the kernel similar to the fast int8 kernel rather than the slow int8 kernel
- That direction of investigation was halted due to Jeff Johnson (the person who wrote the PyTorch GPU backend) developing an int4 kernel that made its way to pytorch

https://github.com/pytorch/pytorch/blob/main/aten/src/ATen/native/native_functions.yaml#L4097

LLaMA-7B, bsz=1	
technique	**perf (tokens/s)**
no quantization	93.08
int8 weight-only quant	40.59
int8 weight-only quant opt	135.01
uint4x2 weight-only quant	43.59
Int4 groupwise quant	187.8



Triton Limitations



- It runs into trouble when trying to work with complicated operations and nonstandard dtypes
  - Int4
  - batchsize>1 int8/int4 weight only
    - L2 Cache Optimization
- Config consistency
  - Issues with Heuristics, in some of the tests I ran,
  - the best configurations aren’t available or are heuristically discarded.

Triton Strengths

- Great at composing “simple” operations together
  - Fused_int_mm_mul
  - SAM flash attention
    - https://github.com/facebookresearch/segment-anything/blob/main/segment_anything/modeling/image_encoder.py#L358
    - https://github.com/pytorch-labs/segment-anything-fast/blob/main/segment_anything_fast/flash_4.py#L13
- Getting you ~75% of the way to cuda speed without having to look at a single .cu file

Learn More

- Reproducing my experiments:
  - https://github.com/pytorch-labs/ao
    - Current best way to access all the pytorch native gpu quantization work,
      - Used by sdxl-fast and segment-anything-fast
  - https://github.com/pytorch-labs/gpt-fast
    - (has GPTQ support for int4)
    - Best for weight only quant
  - https://github.com/pytorch-labs/segment-anything-fast
    - Best for dynamic quant
  - https://gist.github.com/HDCharles/287ac5e997c7a8cf031004aad0e3a941
    - triton microbenchmarks

Questions?

Questions?

Reserve Slides

Results - SAM

- Dynamic Quant achieves ~13% speedup over baseline
- Weight Only Quant doesn't move the needle significantly due to model being primarily compute bound and kernels not being oriented toward large batch sizes
- All quantization techniques resulted in minimal accuracy degradation

	SAM vit_h				
	image_encoder				e2e
model	bs 32 (s)	img/sec	speedup over SDPA	peak memory (GB)	coco 2017 val accuracy
fp16	2.37	13.52	0.67	56.5	0.5842
compiled	1.64	19.55	0.97	47.3	0.5842
SDPA	1.58	20.24	1.00	29.2	0.5842
int8 weight only quant	1.58	20.26	1.00	28.6	0.5837
int8 dynamic (weight + activation) quant	1.40	22.79	1.13	29.2	0.5846
2:4 pruned cusparselt	1.39	23.03		28.0	0.5331

Results - Llama2

- Achieved 45% and 86% speedups using weight only int8 and int4 quantization
- No accuracy drop for int8 weight-only quantization
- Int4 weight-only quantization results in a small accuracy drop, half of which is recovered using GPTQ
- (note) Dynamic Quantization is not in the table, it was tested but had worse accuracy and perf than weight only quantization due to the model being memory bound

	Llama2 7b				
	perf		accuracy		
model	**bs 1 (tok/s)**	**speedup over compiled**	**hellaswag acc_norm**	**wikitext bits_per_byte**	**winogrande acc**
bf16	25.36	0.24	75.96%	0.674	68.67%
compiled	104.11	1.00	75.98%	0.674	68.03%
int8 weight only quant	150.82	1.45	76.04%	0.674	68.27%
int4g128 weight only groupwise quant	193.39	1.86	75.07%	0.695	67.48%
GPTQ (100 examples from each task)	193.39	1.86	75.67%	0.682	68.67%

Results - Llama2 - Simulated Low Precision

- To understand how groupsize, bit number and GPTQ affect accuracy, we compared their effect on wikitext bits_per_byte perplexity
- GPTQ seems to recover half of the lost PPL in all cases
  - except G=64, 2 bit case where the no GPTQ PPL seems anomalously high
- We also got perf numbers for different group sizes notably G=32 results in less than 3% perf reduction for the 4 bit kernel and halves loss by nearly an additional 50%

	Llama2 7b						
	e2e perf	wikitext bits_per_byte					
	bs1	4 Bits		3 Bits		2 Bits	
	tok/s	GPTQ	no GPTQ	GPTQ	no GPTQ	GPTQ	no GPTQ
G=128	193.390	0.683	0.695	0.718	0.910	2.167	3.741
G=64	194.310	0.681	0.689	0.707	0.813	1.501	3.808
G=32	187.800	0.678	0.684	0.702	0.747	1.150	3.397

Note1: lower is better for bits_per_byte perplexity, compare these values to bf16 and int8 value of .674

Note2: 2 bit and 3 bit numbers are simulated using the 4 bit kernel while restricting Qmax during quantization

GPU Quantization for Generative AI

Quantization APIs - Dynamic Quantization

- Quantization APIs are available in the torchao repository
  - https://github.com/pytorch-labs/ao

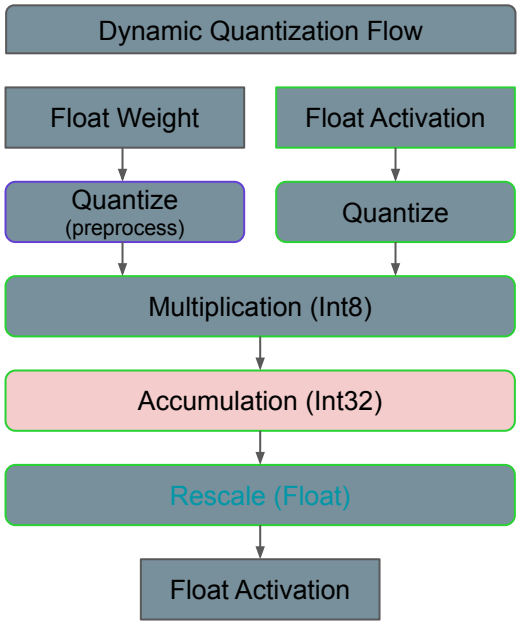
```
def apply_dynamic_quant(model):
    """
    Applies dynamic symmetric per-token activation and per-channel weight
    quantization to all linear layers in the given model using
    module swaps.
    """
    _replace_with_custom_fn_if_matches_filter(
        model,
        lambda mod: DynamicallyPerAxisQuantizedLinear.from_float(mod),
        lambda mod, fqn: isinstance(mod, torch.nn.Linear),
    )
```

Quantization APIs - Dynamic Quantization

- Quantization APIs are available in the torchao repository
  - https://github.com/pytorch-labs/ao

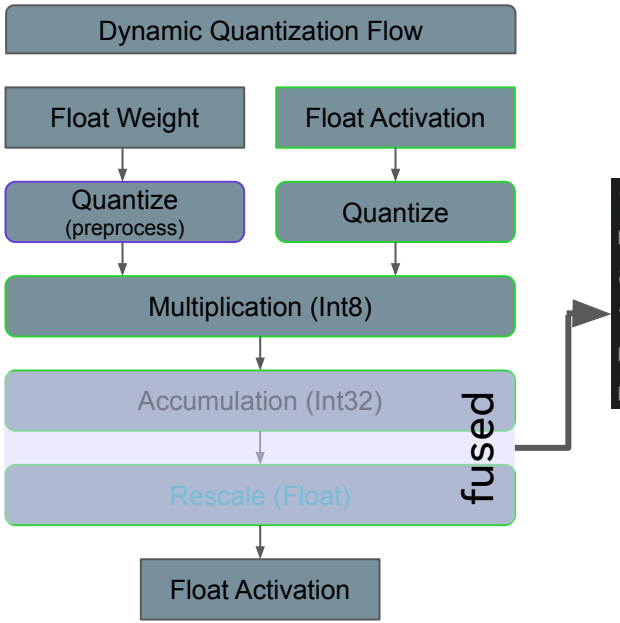
```
from torchao.quantization import apply_dynamic_quant
model2 = deepcopy(model)
apply_dynamic_quant(model2)
torch._inductor.config.force_fuse_int_mm_with_mul = True
model_c = torch.compile(model2, mode='max-autotune')
model_c(image)
```

Quantization APIs - Dynamic Quantization




```
from torchao.quantization import apply_dynamic_quant
model2 = deepcopy(model)
apply_dynamic_quant(model2)
torch._inductor.config.force_fuse_int_mm_with_mul = True
model_c = torch.compile(model2, mode='max-autotune')
model_c(image)
```

Quantization APIs - Dynamic Quantization




```
from torchao.quantization import apply_dynamic_quant
model2 = deepcopy(model)
apply_dynamic_quant(model2)
torch._inductor.config.force_fuse_int_mm_with_mul = True
model_c = torch.compile(model2, mode='max-autotune')
model_c(image)
```

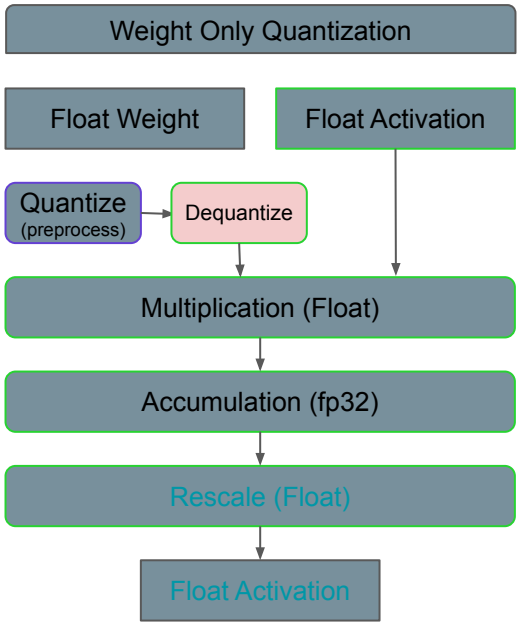
Quantization APIs - Weight Only Quantization

```python
def apply_weight_only_int8_quant(model):
    """
    Applies weight-only symmetric per-channel int8 quantization to all linear layers
    in the given model using module swaps.
    """
    _replace_with_custom_fn_if_matches_filter(
        model,
        WeightOnlyInt8QuantLinear.from_float,
        lambda mod, fqn: isinstance(mod, torch.nn.Linear),
    )
```

Quantization APIs - Weight Only Quantization

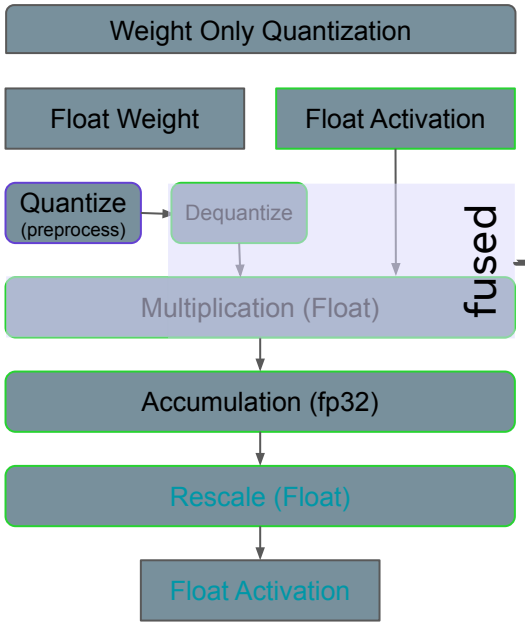
```
from torchao.quantization import apply_weight_only_int8_quant
model2 = deepcopy(model)
apply_weight_only_int8_quant(model2)
torch._inductor.config.use_mixed_mm = True
model_c = torch.compile(model2, mode='max-autotune')
model_c(image)
```

Quantization APIs - Weight Only Quantization




```
from torchao.quantization import apply_weight_only_int8_quant
model2 = deepcopy(model)
apply_weight_only_int8_quant(model2)
torch._inductor.config.use_mixed_mm = True
model_c = torch.compile(model2, mode='max-autotune')
model_c(image)
```

Quantization APIs - Weight Only Quantization




```
from torchao.quantization import apply_weight_only_int8_quant
model2 = deepcopy(model)
apply_weight_only_int8_quant(model2)
torch._inductor.config.use_mixed_mm = True
model_c = torch.compile(model2, mode='max-autotune')
model_c(image)
```

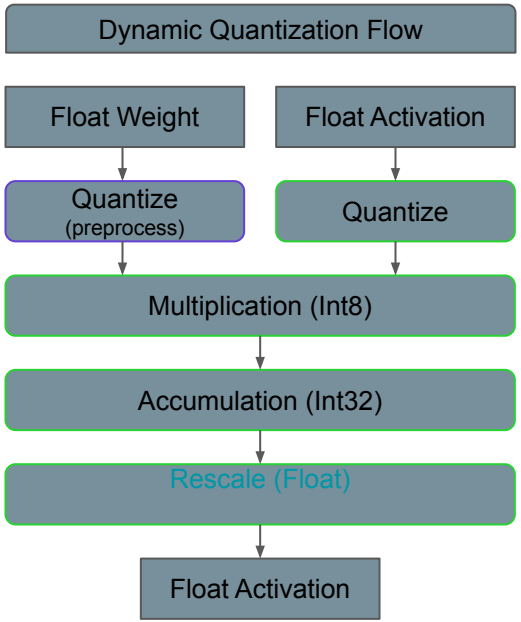
Code Pointers

- Quantization APIs are available in the torchao repository
  - https://github.com/pytorch-labs/ao
- The segment-anything-fast repository demonstrates how to use these APIs in concert with other techniques
  - https://github.com/pytorch-labs/segment-anything-fast
- The gpt-fast repository uses related APIs to perform quantization
  - It also has implementations of int4 quantization and GPTQ that currently don't live elsewhere
  - https://github.com/pytorch-labs/gpt-fast

Next Steps

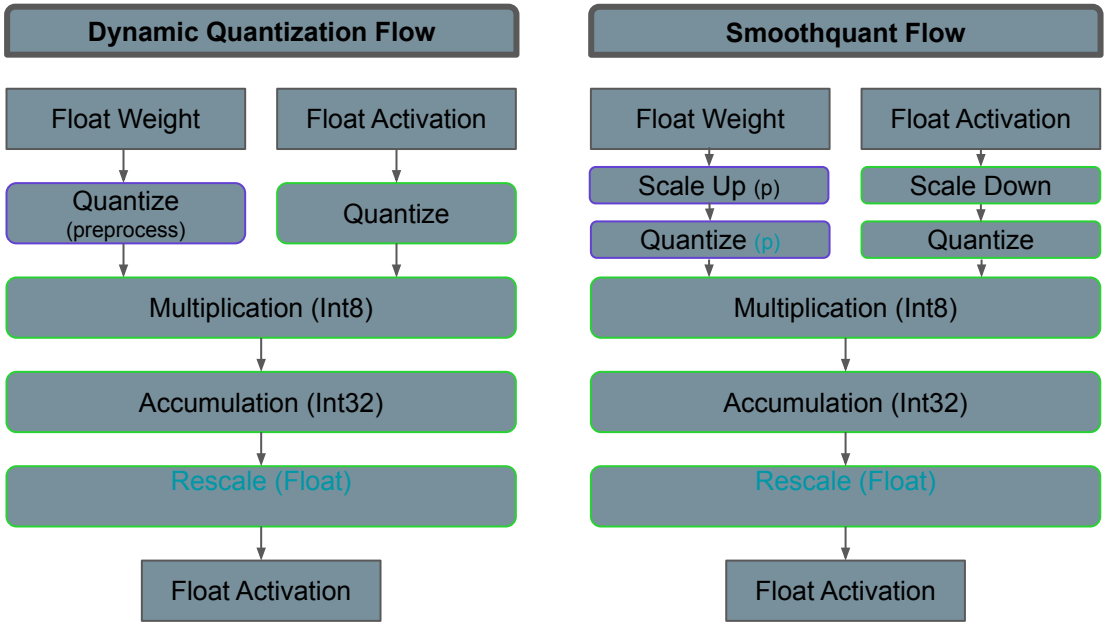
- Add features to https://github.com/pytorch-labs/ao APIs
  - Use subclasses instead of module swaps
  - Enable save/load support
- Apply quantization to models in torchbench and see how our coverage looks
- Finish tutorial for torchao APIs and how to optimize your own model
- Figure out plan for GPTQ and int4 quantization in either pt2e flow or something else less model dependent
- APIs for static quantization

Dynamic Quantization



- Recalculates quantization parameters for each sample
  - Insensitive to non-stationary distributions
  - Sensitive to frequent outliers
- Floating point activations
  - Drop in replacement for non quantized op
  - Can be slower than techniques which allow for a series of quantized ops without dequantizing

Smoothquant

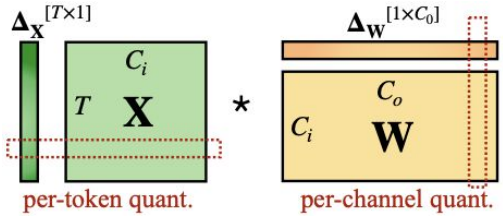


[Smoothquant]

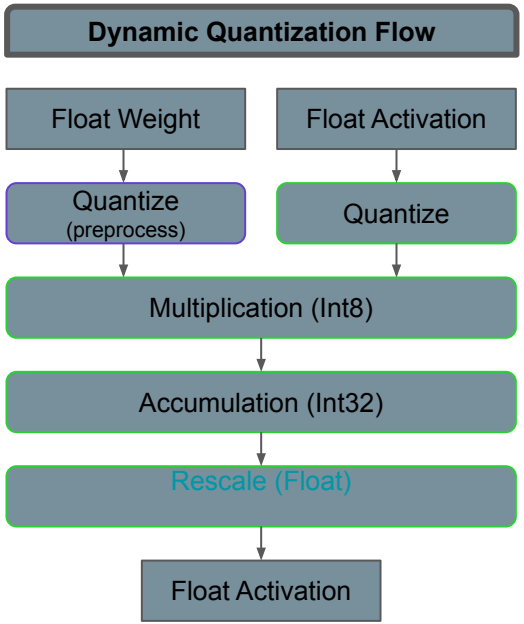
- Input-weight equalization

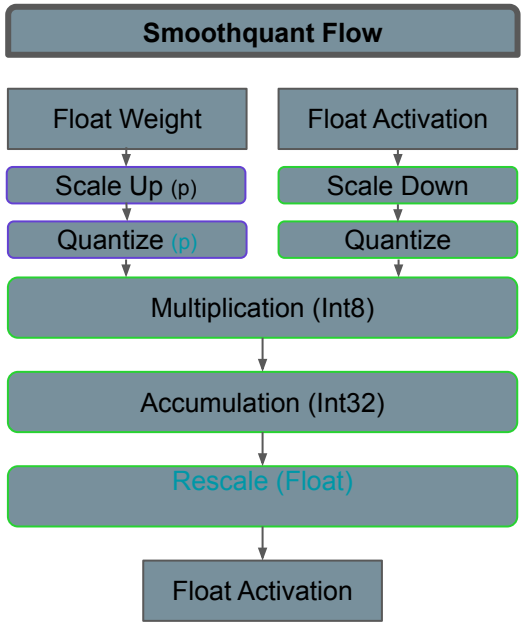
We combine it with [LLM.int8]()

- per-token quant activations
- per-channel quant weights



Smoothquant





Method	OPT-175B	BLOOM-176B	GLM-130B*
FP16	71.6%	68.2%	73.8%
W8A8	32.3%	64.2%	26.9%
ZeroQuant	31.7%	67.4%	26.7%
`LLM.int8()`	71.4%	68.0%	73.8%
Outlier Suppression	31.7%	54.1%	63.5%
SmoothQuant-O1	**71.2%**	68.3%	**73.7%**
SmoothQuant-O2	71.1%	**68.4%**	72.5%
SmoothQuant-O3	71.1%	67.4%	72.8%

Smoothquant

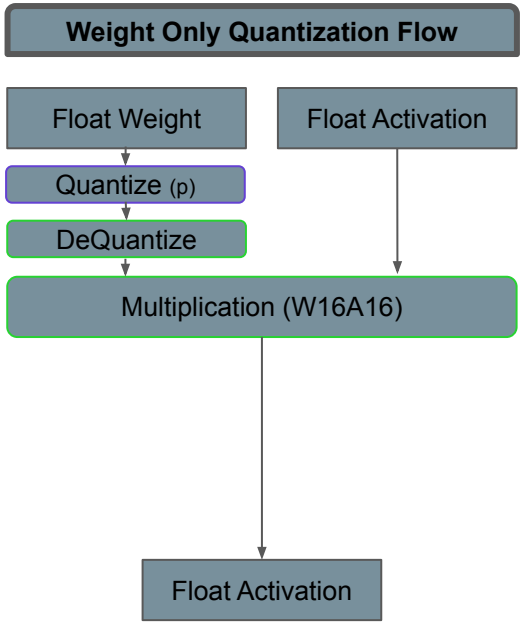
- Input-weight equalization

We combine it with LLM.int8()

- per-token quant activations
- per-channel quant weights

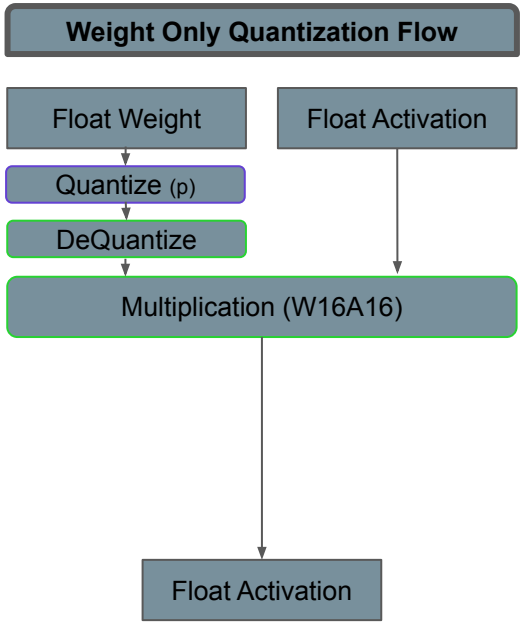
Method	Weight	Activation
W8A8	per-tensor	per-tensor dynamic
ZeroQuant	group-wise	per-token dynamic
`LLM.int8()`	per-channel	per-token dynamic+FP16
Outlier Suppression	per-tensor	per-tensor static
SmoothQuant-O1	per-tensor	per-token dynamic
SmoothQuant-O2	per-tensor	per-tensor dynamic
SmoothQuant-O3	per-tensor	per-tensor static

Weight Only Quantization Int8

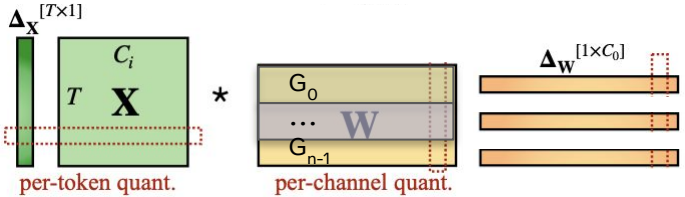


- More accurate than methods with activation quantization
  - Activations tend to be the difficult piece to quantize in practice
- Mixed dtype matmul is more expensive than fp16-fp16 matmul computationally, though more efficient in terms of memory
- In practice we use per-channel quantization for int8

Weight Only Quantization Int4



- Mixed dtype matmul is more expensive than fp16-fp16 matmul computationally, though more efficient in terms of memory
- In practice we use group-wise quantization for int4 since the accuracy is worse
  - In addition to quantizing the weight per-channel, group-wise quantization breaks C_i into n groups G_0 to G_{n-1} with each group having its own



Note: could still do smoothquant style input-weight equalization here which may be useful for the int4 case

GPTQ

- GPTQ Uses Expected Hessian of

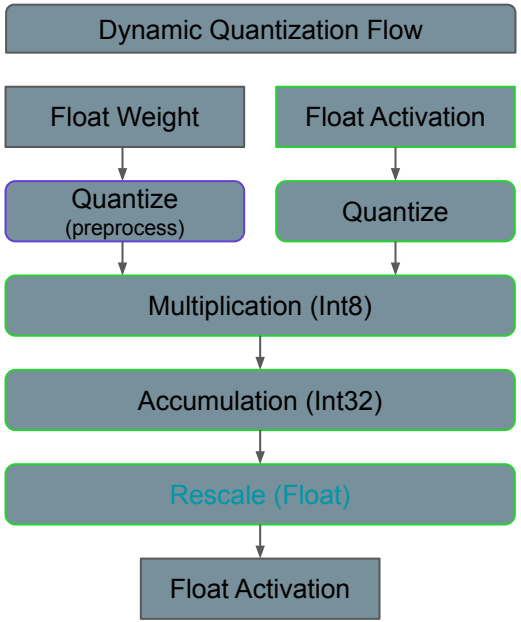
$$\text{argmin}_{\widehat{\mathbf{W}}} ||\mathbf{WX} - \widehat{\mathbf{W}}\mathbf{X}||_2^2.$$

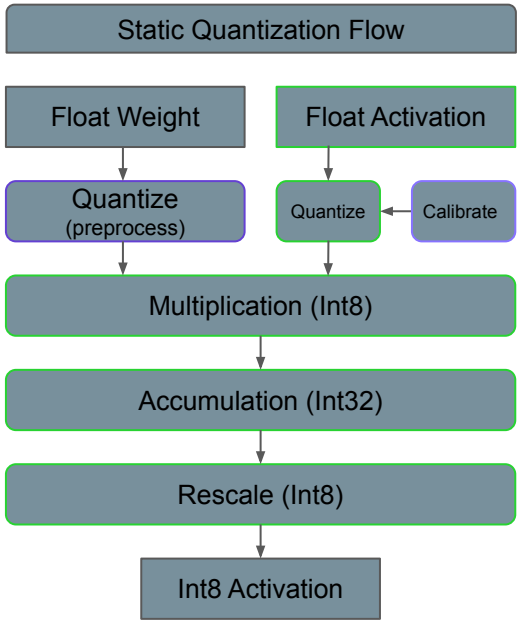
  to quantize W given a predefined quantization scheme (group-wise, per-channel)
- Necessary to get decent int4 weight-only accuracy

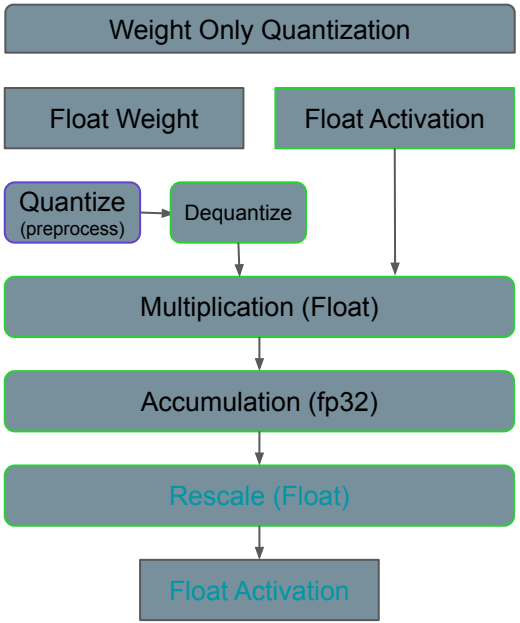
GPTQ High Level Algorithm:

1. estimate Hessian over several batches for some layer
2. Quantize a column of W
3. Use H to update the non-quantized columns of W (to keep the above equation minimized)
4. Repeat 2,3 until no more columns left to quantize

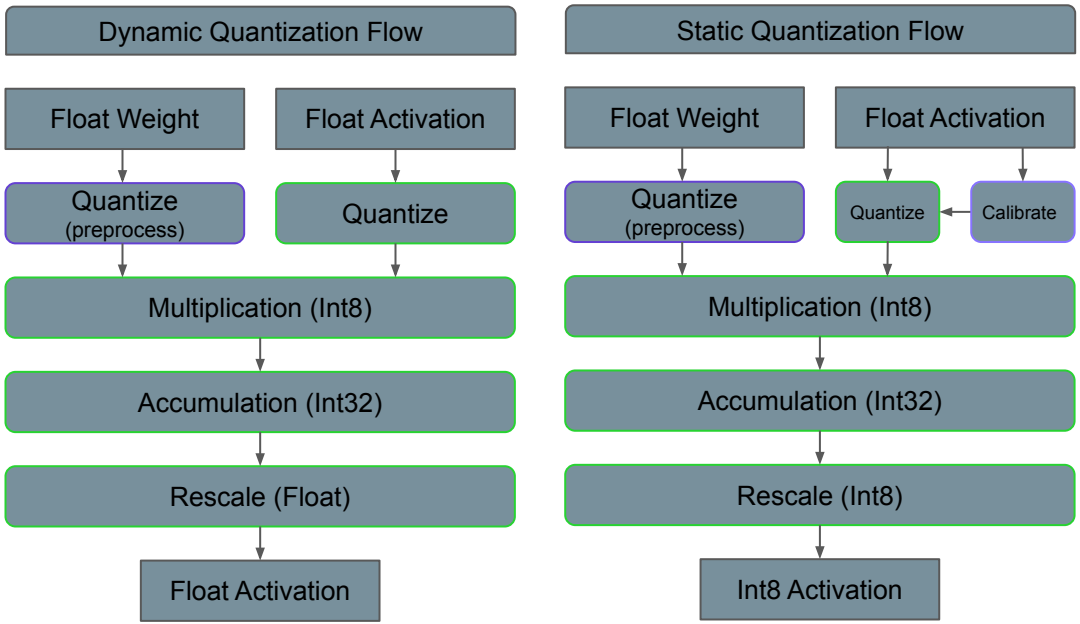
Techniques


Dynamic Quantization Flow
Float Weight
Float Activation
Quantize
(preprocess)
Quantize
Multiplication (Int8)
Accumulation (Int32)
Rescale (Float)
Float Activation


Static Quantization Flow
Float Weight
Float Activation
Quantize
(preprocess)
Quantize
Calibrate
Multiplication (Int8)
Accumulation (Int32)
Rescale (Int8)
Int8 Activation


Weight Only Quantization
Float Weight
Float Activation
Quantize
(preprocess)
Dequantize
Multiplication (Float)
Accumulation (fp32)
Rescale (Float)
Float Activation

Static Quantization



- Calculates best quantization parameters over calibration set
  - More sensitive to non-stationary distributions
  - Less sensitive to frequent outliers
- Integer activations
  - Best if have a sequence of quantizable ops

Static Quantization with Float Output