GPU Sparsity

03/21/24

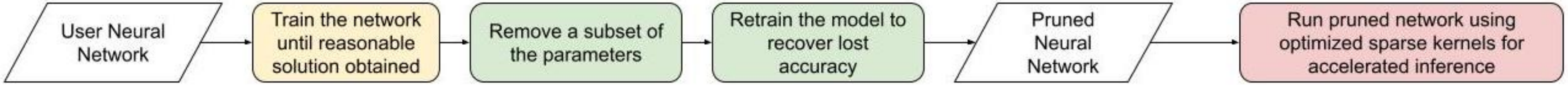
About Me

- PyTorch Core
  - Architecture Optimization
  - Quantization / Sparsity
  - GenAI (LLMs, ViT)
- Based in NYC
- UCLA (Go Bruins!)

- Email: jessecai@meta.com
- Github: @jcaip

What is Sparsity / Pruning?

- Why waste compute use many parameters when few parameter do trick ?



- Two parts:
  - Accuracy: zero out parameters from the model
  - Performance: how to make multiplying by zero fast

- Dates back to [Optimal Brain Damage (Hinton 89)]

Sparsity (Performance)

- Multiplying by zero is fast!
  - 0 * 1231231 vs 123123123 * 123123
- But not if you still do the computation
  - You can still do long multiplication for 0 * 1231231 and this will take a similar amount of time as 2 * 1231231
- So let’s add a bunch of zeros to our model and skip those computations

How should we add zeros?

- Different Sparsity Patterns
- We want flexibility for accuracy
- But we want structure for performance reasons

Sparsity Pattern	**Mask Visualization (50% sparsity level)**
Unstructured Sparsity	1 0 1 1 0 1 0 1 0 0 1 1 1 1 1 0 1 0 0 0 1 0 1 0 0 1 1 0 0 0 0 1 *Fig 2.3: unstructured sparsity*
2:4 Semi-Structured	0 1 1 0 1 0 1 0 0 0 1 1 1 1 0 0 1 0 0 1 0 1 0 1 0 1 0 1 1 0 1 0 *Fig 2.4: 2:4 semi-structured sparsity*
Block Sparsity	0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 *Fig 2.5: 4x4 block-wise structured sparsity*
Structured Sparsity	1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 *Fig 2.6: row-wise structured sparsity*

Table 4.4: Description of some common sparsity patterns

Sparsity (Performance)

- How can we do that for tensors multiplication?
  - [Sparse representations](#) + Sparse kernels
  - Store the data and structure independently
- COO representation
  - [1 2 3
0 0 0 -> index: < (0,0) (0,1) (0,2) >
0 0 0] data: < 1 2 3 >
- Sparse matmul
  - Only faster at high sparsity levels (>99%)

Sparsity Pattern	**Mask Visualization (50% sparsity level)**
Unstructured Sparsity	1 0 1 1 0 1 0 1 0 0 1 1 1 1 1 0 1 0 0 0 1 0 1 0 0 1 1 0 0 0 0 1 *Fig 2.3: unstructured sparsity*
2:4 Semi-Structured	0 1 1 0 1 0 1 0 0 0 1 1 1 1 0 0 1 0 0 1 0 1 0 1 0 1 0 1 1 0 1 0 *Fig 2.4: 2:4 semi-structured sparsity*
Block Sparsity	0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 *Fig 2.5: 4x4 block-wise structured sparsity*
Structured Sparsity	1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 *Fig 2.6: row-wise structured sparsity*

Table 4.4: Description of some common sparsity patterns

OK but how to parallelize?

- Unstructured sparsity is cool and accuracy preserving but you can’t make it fast on GPUs
    - Not easy to parallelize
    - GPUs work on blocks - we can’t have blocks with structure with unstructured sparsity
- How to make it fast on GPU
    - What if we remove a row instead of just a single param? structured pruning
    - We can reuse dense kernels (yay)
    - But the accuracy impact is large and difficult to deal with

GPU Sparsity

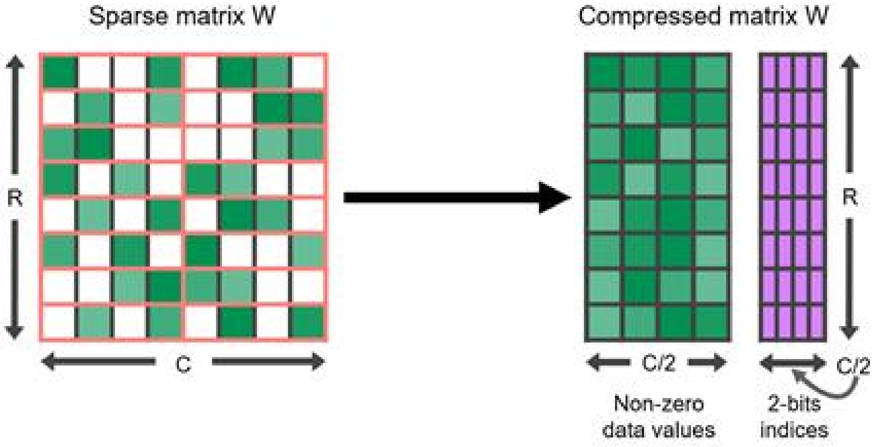
- Semi-structured (2:4) sparsity
  - Fixed 50% sparsity level, up to 2x theoretical max speedup
  - Relatively easy to recover accuracy.
- Block sparsity
  - Block based on block size, speedups of ~3.4x at 90% sparsity
  - Requires more advanced algorithms to recover accuracy (DRESS)

Sparsity Pattern	**Mask Visualization (50% sparsity level)**
Unstructured Sparsity	1 0 1 1 0 1 0 1 0 0 1 1 1 1 1 0 1 0 0 0 1 0 1 0 0 1 1 0 0 0 0 1 *Fig 2.3: unstructured sparsity*
2:4 Semi-Structured	0 1 1 0 1 0 1 0 0 0 1 1 1 1 0 0 1 0 0 1 0 1 0 1 0 1 0 1 1 0 1 0 *Fig 2.4: 2:4 semi-structured sparsity*
Block Sparsity	0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 *Fig 2.5: 4x4 block-wise structured sparsity*
Structured Sparsity	1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 *Fig 2.6: row-wise structured sparsity*

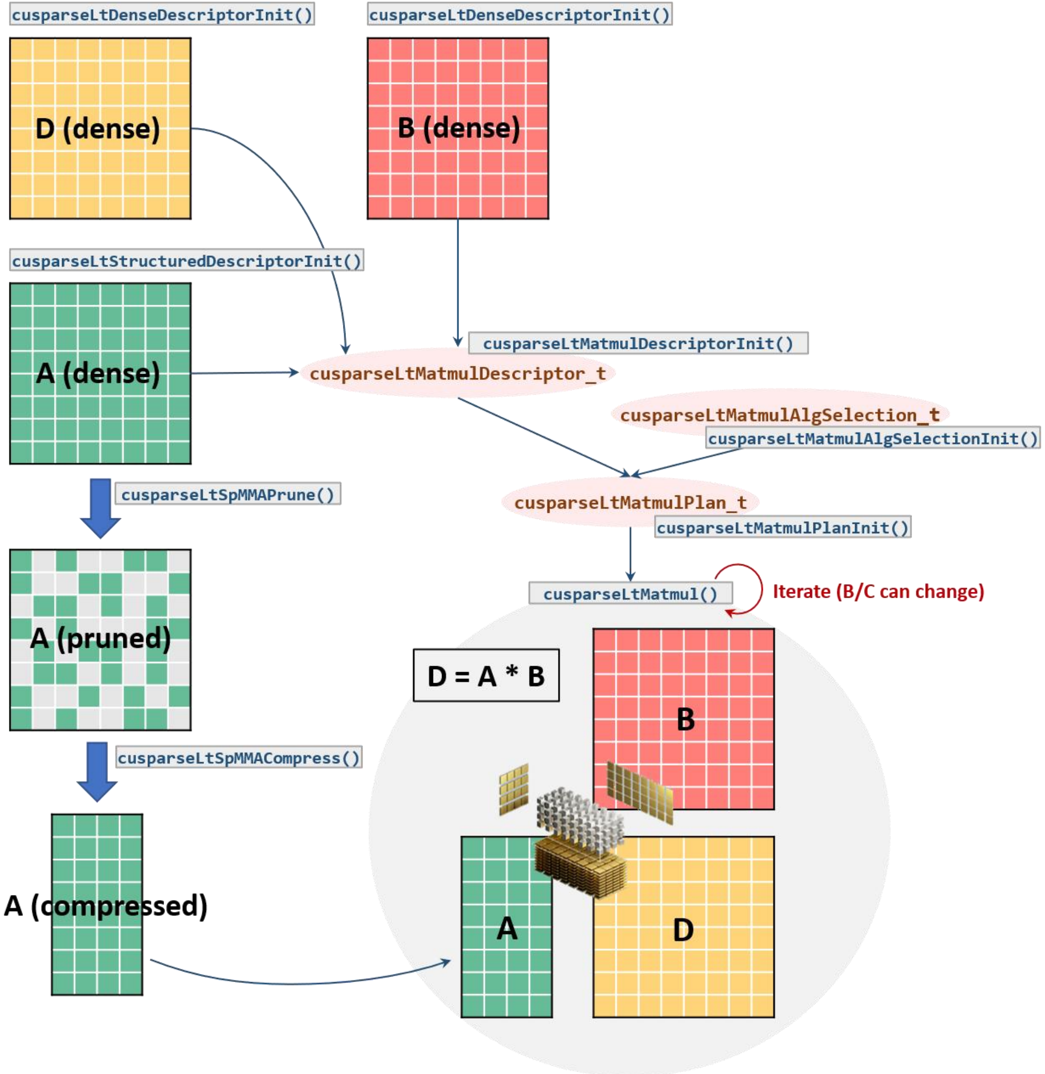
Table 4.4: Description of some common sparsity patterns

Semi-Structured (2:4) Sparsity

- Also known as M:N / fine-grained structured sparsity
- 2 in every 4 elements are 0
- Can be applied to STRIP or TILE



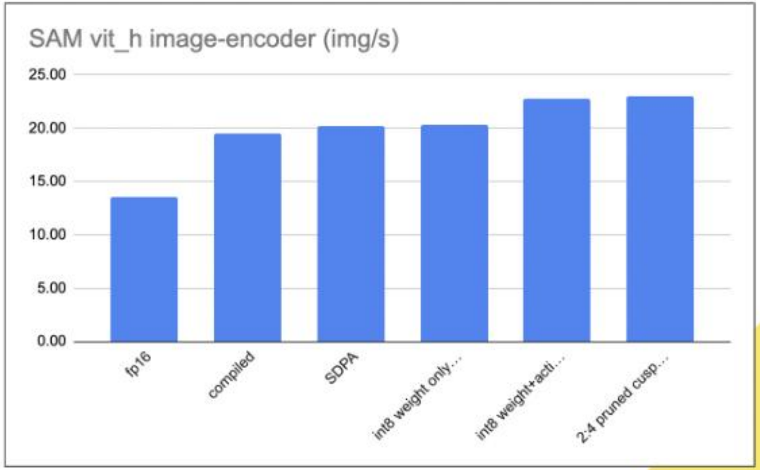
- Theoretical 2x max acceleration
  - 1.6x on average in practice
  - Depends on matrix shapes
- Simple retraining recipe
  - Prune once the retrain
- cuSPARSELt vs CUTLASS backend


cusparseLtDenseDescriptorInit()
D (dense)
cusparseLtDenseDescriptorInit()
B (dense)
cusparseLtStructuredDescriptorInit()
A (dense)
cusparseLtMatmulDescriptorInit()
cusparseLtMatmulDescriptor_t
cusparseLtMatmulAlgSelection_t
cusparseLtMatmulAlgSelectionInit()
cusparseLtSpMMAPrune()
cusparseLtMatmulPlan_t
cusparseLtMatmulPlanInit()
A (pruned)
cusparseLtMatmul()
Iterate (B/C can change)
D = A * B
B
cusparseLtSpMMACompress()
A (compressed)
A
D

E2E Results

Network	Data Set	Metric	Dense FP16	Sparse FP16
ResNet-50	ImageNet	Top-1	76.1	76.2
ResNeXt-101_32x8d	ImageNet	Top-1	79.3	79.3
Xception	ImageNet	Top-1	79.2	79.2
SSD-RN50	COCO2017	bbAP	24.8	24.8
MaskRCNN-RN50	COCO2017	bbAP	37.9	37.9
FairSeq Transformer	EN-DE WMT'14	BLEU	28.2	28.5
BERT-Large	SQuAD v1.1	F1	91.9	91.9

Table 2. Sample accuracy of 2:4 structured sparse networks trained with our recipe.



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

Easy API

- https://gist.github.com/jcaip/44376cd69d3a05cbe16610b4379d9b70
- Get zeros into the right spots, then call
  - to_sparse_semi_structured
- Works with torch.compile!
  - Needed to fuse transpositions
  - Only first matrix sparse
  - xW’ = (xW’)’’ = (Wx’)’

sparsify.py

```python
import torch
from torch.sparse import to_sparse_semi_structured, SparseSemiStructuredTensor

# Sparsity helper functions
def apply_fake_sparsity(model):
    """
    This function simulates 2:4 sparsity on all linear layers in a model.
    It uses the torch.ao.pruning flow.
    """
    # torch.ao.pruning flow
    from torch.ao.pruning import WeightNormSparsifier
    sparse_config = []
    for name, mod in model.named_modules():
        if isinstance(mod, torch.nn.Linear):
            sparse_config.append({"tensor_fqn": f"{name}.weight"})

    sparsifier = WeightNormSparsifier(sparsity_level=1.0,
                                      sparse_block_shape=(1,4),
                                      zeros_per_block=2)
    sparsifier.prepare(model, sparse_config)
    sparsifier.step()

    sparsifier.step()
    sparsifier.squash_mask()


def apply_sparse(model):
    apply_fake_sparsity(model)
    for name, mod in model.named_modules():
        if isinstance(mod, torch.nn.Linear):
            mod.weight = torch.nn.Parameter(to_sparse_semi_structured(mod.weight))
```

Block Sparsity

- Use Superblock to recover accuracy
- Microbenchmarks for ViT-L Layers:

Batch size = 256 MLP 1				
	blocksize=8	blocksize=16	blocksize=32	blocksize=64
sparsity_level = 0.9	0.007	0.509	1.439	2.202
sparsity_level = 0.8	0.003	0.262	0.666	1.776
Batch size = 256 MLP 2				
	blocksize=8	blocksize=16	blocksize=32	blocksize=64
sparsity_level = 0.9	0.008	0.429	1.073	1.631
sparsity_level = 0.8	0.004	0.28	0.795	1.251

- E2E result: ImageNet ViT-L 1.44x speedup with minimal acc loss (78 -> 76)

Current Work

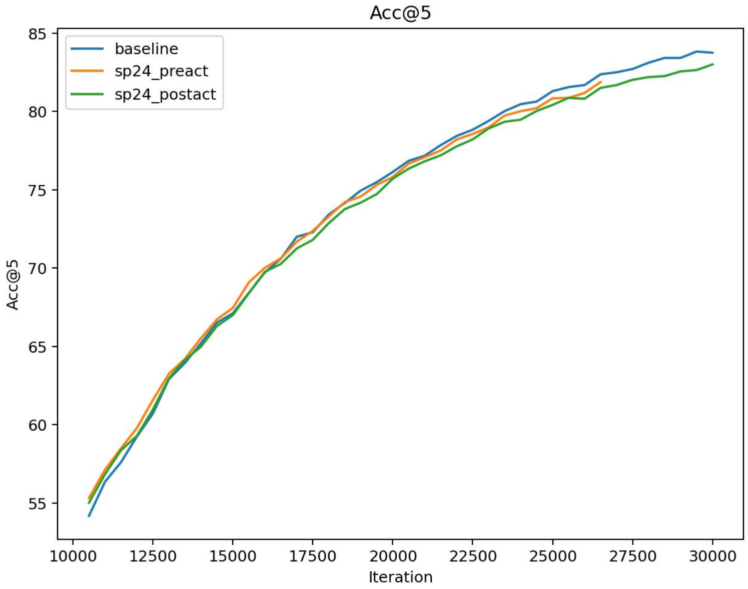
- Composing with quantization
  - Performance
  - Accuracy
- 2:4 sparse training
- Pruning algorithm experimentation (torchao)
  - We need help here!!! This is a the current sticking point

2:4 Sparse Training

2:4 sparsity for training

- Can we use 2:4 sparsity for training? Yes!
- Main Idea:
  - Sparsify + sparse_mm < dense_mm
  - Need both W and W_t for forward / backward pass
- xFormers ran experiments with DINO show 10-20% e2e speedup with 0.5% acc gap (81.6 -> 80.5) on ImageNet-1k
  - Applying 2:4 sparsity to activations

Initial results - ViT-Base on ImageNet / MLP with GeLU



baseline

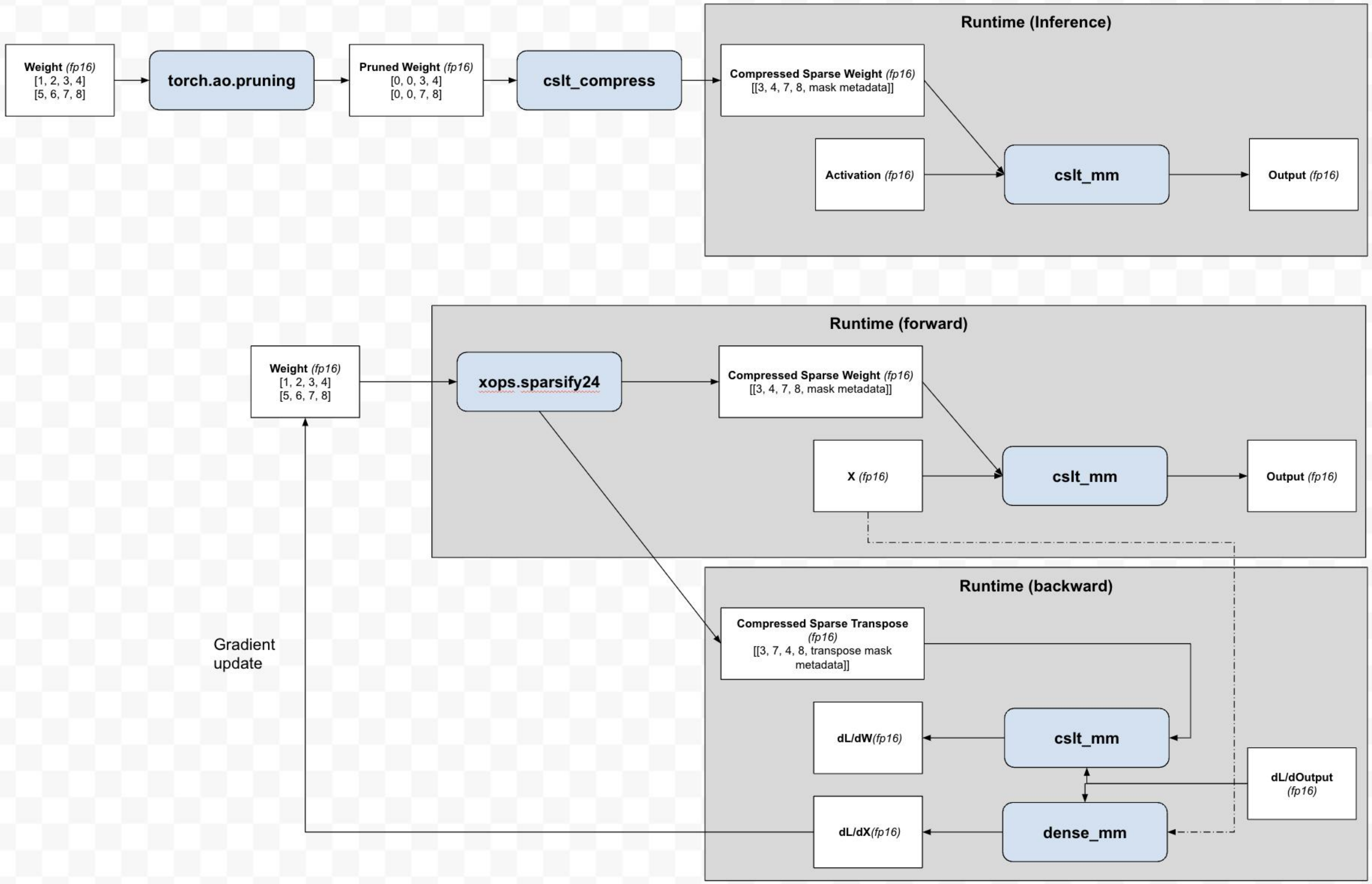
```python
def forward(self, x: Tensor) -> Tensor:
    x = self.fc1(x)
    x = self.act(x)
    x = self.drop(x)
    x = self.fc2(x)
    x = self.drop(x)
    return x
```

sp24_postact

```python
def forward(self, x: Tensor) -> Tensor:
    x = self.fc1(x)
    x = self.act(x)
    x = self.drop(x)

    x = self.sparsify_24(x.abs())

    x = self.fc2(x)
    x = self.drop(x)
    return x
```

sp24_preact

```python
def forward(self, x: Tensor) -> Tensor:
    x = self.fc1(x)

    x = self.sparsify_24(x)

    x = self.act(x)
    x = self.drop(x)
    x = self.fc2(x)
    x = self.drop(x)
    return x
```



Runtime (Inference)
Weight (fp16)
[1, 2, 3, 4]
[5, 6, 7, 8]
torch.ao.pruning
Pruned Weight (fp16)
[0, 0, 3, 4]
[0, 0, 7, 8]
cslt_compress
Compressed Sparse Weight (fp16)
[[3, 4, 7, 8, mask metadata]]
Activation (fp16)
cslt_mm
Output (fp16)
Runtime (forward)
Weight (fp16)
[1, 2, 3, 4]
[5, 6, 7, 8]
xops.sparsify24
Compressed Sparse Weight (fp16)
[[3, 4, 7, 8, mask metadata]]
X (fp16)
cslt_mm
Output (fp16)
Runtime (backward)
Compressed Sparse Transpose (fp16)
[[3, 7, 4, 8, transpose mask metadata]]
Gradient update
dL/dW(fp16)
cslt_mm
dL/dOutput (fp16)
dL/dX(fp16)
dense_mm

Difference between inference and training

Training:

- Only compute benefit, slight memory penalty (9/8) *
- Sparsification happens at runtime
- Need both W and W_t
- Needs contiguous output for distributed collective

Inference:

- Compute + memory speedup
- Sparsification happens offline
- Just W is sufficient
- Can return a view and torch.compile away subsequent transposition

2:4 sparse training components

- Fast sparsification ops:
  - Allow us to quickly calculate X_compressed and X_t_compressed.
  - Note that we need both for the forward and backward pass respectively.
- Custom autograd.Function
  - Forwards: returns sparse tensor subclass (X_compressed)
    - output = X @ W.T
  - Backwards: gradient update (use X_t_compressed)
    - input.grad = output.grad @ W.grad = output.grad.T @ input
- cuSPARSELt
  - Transposition fusion for subsequent distributed collective.

```python
class _Sparsify24Func(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x: torch.Tensor, algo: str, gradient: str, backend: str):  # type: ignore[override]
        if gradient not in [GRADIENT_SP24, GRADIENT_DENSE]:
            raise ValueError(
                f"Invalid gradient type: '{gradient}'. "
                f"Expected '{GRADIENT_SP24}' or '{GRADIENT_DENSE}'"
            )
        if not isinstance(x, Sparse24Tensor):
            (packed, meta, packed_t, meta_t, threads_masks) = SparsifyBothWays.OPERATOR(
                x, algorithm=algo, backend=backend
            )
            cls = (
                Sparse24TensorCutlass
                if backend == BACKEND_CUTLASS
                else Sparse24TensorCuSparseLt
            )
            out = cls(
                x.shape,
                packed=packed,
                meta=meta,
                packed_t=packed_t,
                meta_t=meta_t,
                threads_masks=threads_masks,
                requires_grad=False,
            )
        else:
            if x.threads_masks is None:
                raise ValueError("!!")
            out = x
        ctx.threads_masks = out.threads_masks
        ctx.meta = out.meta
        ctx.meta_t = out.meta_t
        ctx.dtype = out.dtype
        ctx.gradient = gradient
        return out
```

```python
@staticmethod
def backward(ctx, grad_out: torch.Tensor):  # type: ignore[override]
    if isinstance(grad_out, Sparse24Tensor):
        return grad_out, None, None, None
    assert not isinstance(grad_out, Sparse24Tensor)
    assert grad_out.dtype == ctx.dtype
    if ctx.gradient == GRADIENT_SP24:
        packed, packed_t = SparsifyApply.OPERATOR(grad_out, ctx.threads_masks)
        grad_in: torch.Tensor = Sparse24TensorCutlass(
            grad_out.shape,
            packed,
            ctx.meta,
            packed_t,
            ctx.meta_t,
            ctx.threads_masks,
            requires_grad=grad_out.requires_grad,
        )
    elif ctx.gradient == GRADIENT_DENSE:
        assert ctx.threads_masks.is_contiguous()
        grad_in = SparsifyApplyDenseOutput.OPERATOR(grad_out, ctx.threads_masks)
    else:
        assert False, f"Unsupported gradient type: {ctx.gradient}"
    return (
        grad_in,
        None,
        None,
        None,
    )
```

Open Questions

- Custom autograd functions + torch.compile
  - See here for more info
  - Dynamo when speculating expects that the gradient flowing in is a subclass, because the output of forward is a subclass for the autograd Function
- Can we reuse the same memory for specified elements of the sparse tensor?
  - Specified elements look to be the same, but swizzled between A_compressed and A_t_compressed
- Can we rewrite fast sparsification with torch.compile?
  - Alek from Quantsight has written fast torch.compile routines for CUTLASS, could we do the same thing but for cuSPARSELt?

Future Work

Future Work

- Block sparsity + 2:4 sparsity
  - Blocks that are 2:4 sparse themselves
  - 2:4 sparse arrangement of blocks
- Fusing a shuffle into the sparsity pattern (SHFL-BW)
  - More flexibility for accuracy
  - We can fuse the shuffle with torch.compile -> minimal perf hit
- 2:4 Sparse dropout
  - Do people still use dropout anymore?

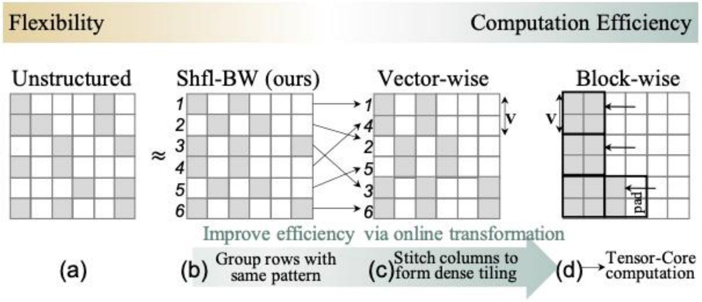


Figure 3: Different sparse patterns and how to transform from *Shfl-BW* to block-wise sparsity.

Future Work (cuda-mode specific)

- Closing CUTLASS and cuSPARSELt gap
    - Aleksandar Samardžić has done a great job of this
    - Fuse transposition into matmul
    - Fuse dequant into CUTLASS kernel
- Autotuning parameters for sparse kernels
- Writing larger fused kernels
- More flexible sparsity patterns
- Load-as-sparse kernels
- Add to triton ?

Conclusion

- It’s early but we’re building towards something that could 2x (or more) the size of models.
- Accuracy is the main blocker right now
  - We can give researchers more flexible sparsity patterns
- Performance
  - We’ve show that sparsity works on GPUs and is faster
  - We can push performance by stacking sparsity / working on the corner cases
- Accuracy and Performance are getting more intertwined