Slaying ⚔️ OOMs

Jane Xu
Mark Saroufim

PyTorch Devs

How people deal with OOMs

Smaller batch size
Smaller model

RuntimeError: CUDA out of memory

https://en.wikipedia.org/wiki/Bazooka#/media/File:Soldier_with_Bazooka_M1.jpg

VRAM

24 GB of VRAM

40 or 80 GB of VRAM

Memory crash course

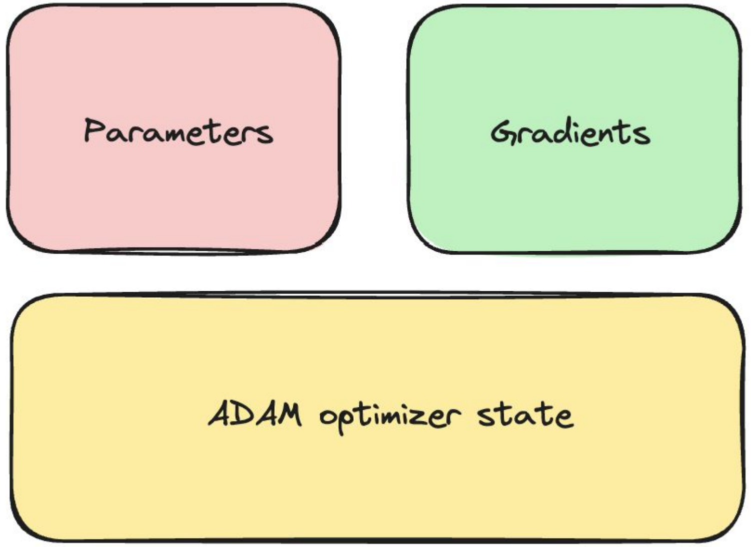
Llama 7B has 7B parameters in fp16

Each parameter is 2 bytes so parameters is 14GB

Gradients memory = parameter memory

Adam Optimizer State = 2 * parameter memory

Total = 14GB + 14GB + 28GB = 56GB

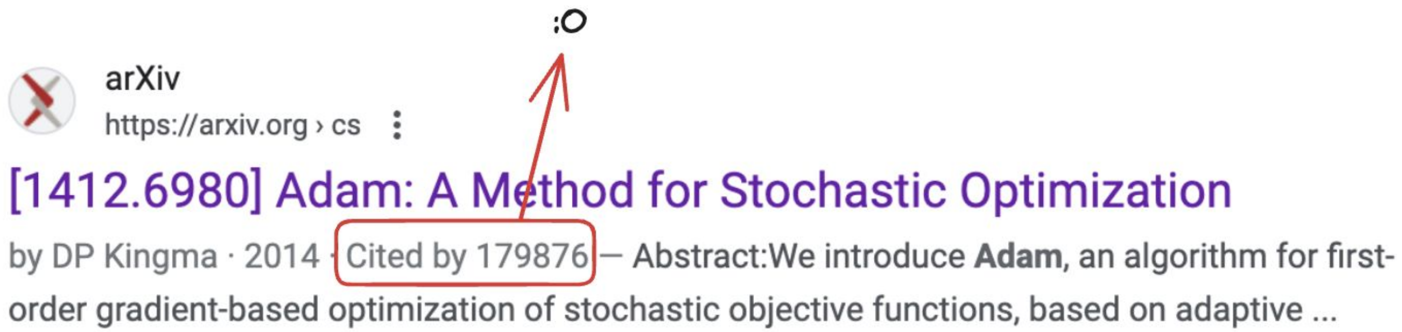


Larger batch sizes and context lengths

Bottleneck is almost always Activations that’s why Flash Attention is important

Paper math is great but Papers don’t tell us when we’re wrong
https://dev-discuss.pytorch.org/t/how-to-measure-memory-usage-from-your-model-without-running-it/2024

Let’s optimize the bottleneck!


:O

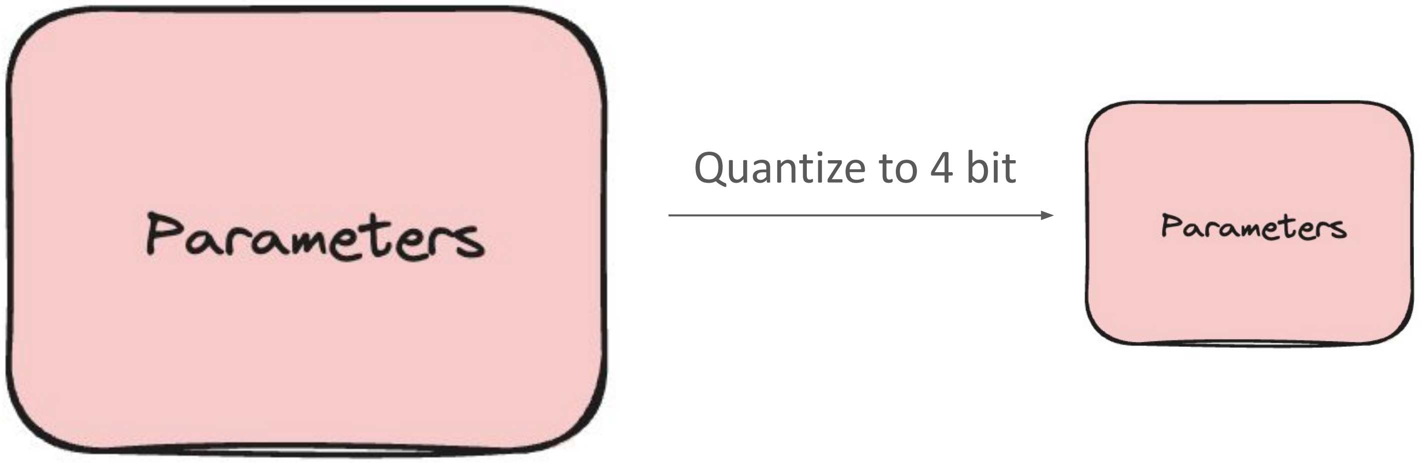
arXiv

https://arxiv.org › cs

[1412.6980] Adam: A Method for Stochastic Optimization

by DP Kingma · 2014 · Cited by 179876 — Abstract:We introduce **Adam**, an algorithm for first-order gradient-based optimization of stochastic objective functions, based on adaptive ...

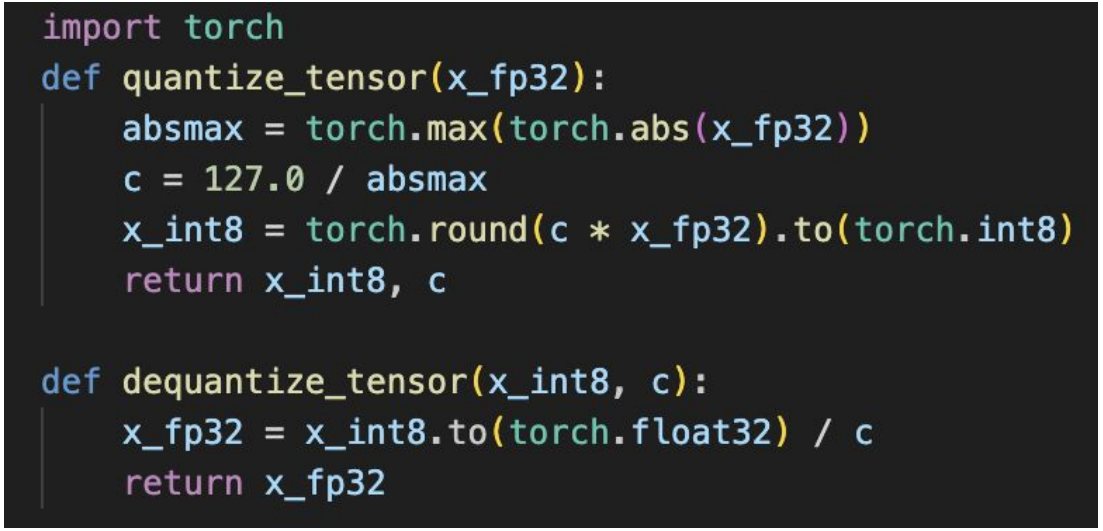
Ok let’s take a look at params



14 GB at fp16

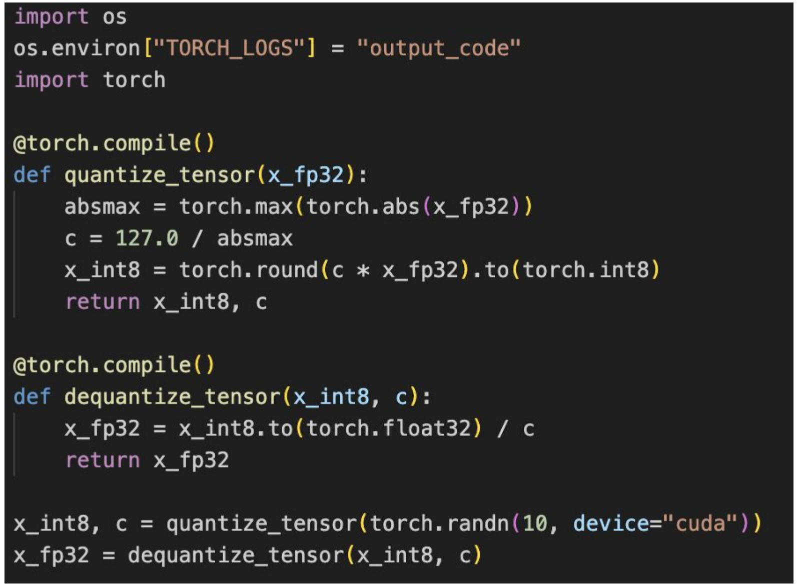
3.5 GB at int4
Each int4 needs ½ byte*

Hello quantization

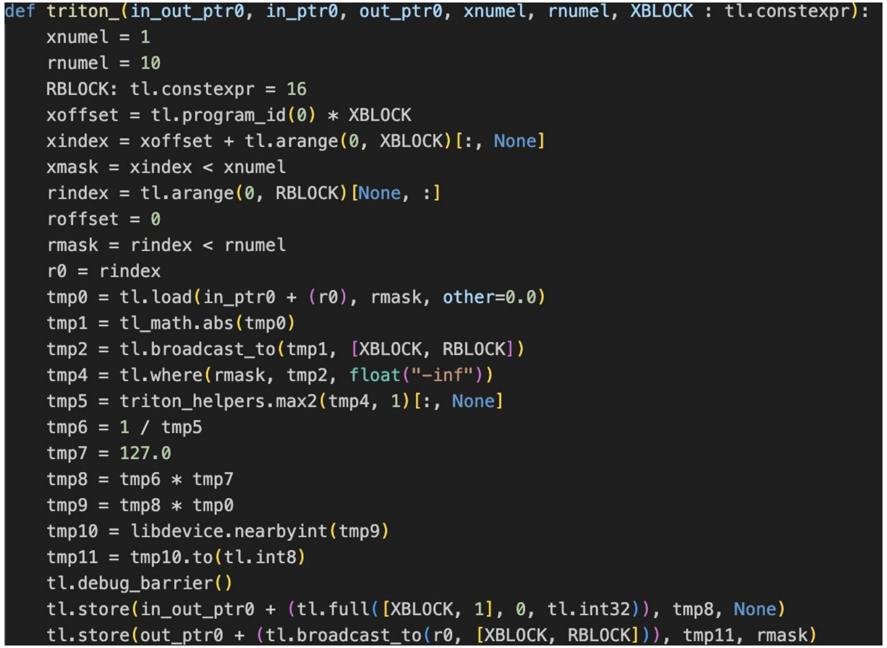


```python
import torch
def quantize_tensor(x_fp32):
    absmax = torch.max(torch.abs(x_fp32))
    c = 127.0 / absmax
    x_int8 = torch.round(c * x_fp32).to(torch.int8)
    return x_int8, c

def dequantize_tensor(x_int8, c):
    x_fp32 = x_int8.to(torch.float32) / c
    return x_fp32
```

torch.compile



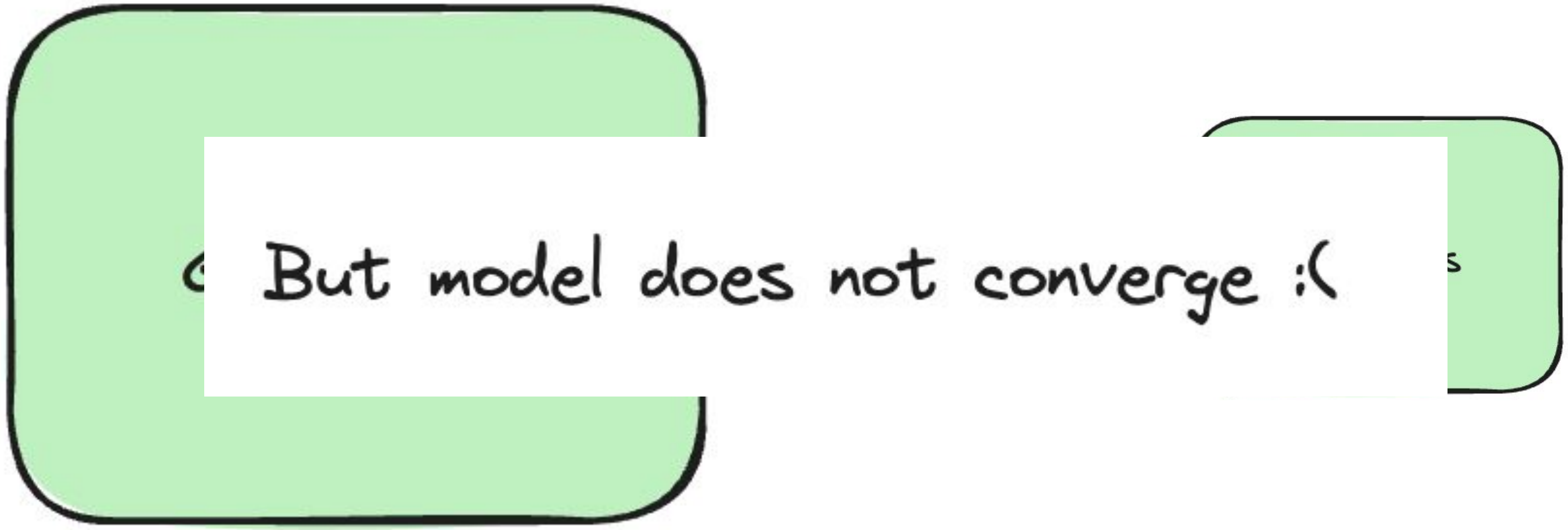
```python
import os
os.environ["TORCH_LOGS"] = "output_code"
import torch

@torch.compile()
def quantize_tensor(x_fp32):
    absmax = torch.max(torch.abs(x_fp32))
    c = 127.0 / absmax
    x_int8 = torch.round(c * x_fp32).to(torch.int8)
    return x_int8, c

@torch.compile()
def dequantize_tensor(x_int8, c):
    x_fp32 = x_int8.to(torch.float32) / c
    return x_fp32

x_int8, c = quantize_tensor(torch.randn(10, device="cuda"))
x_fp32 = dequantize_tensor(x_int8, c)
```



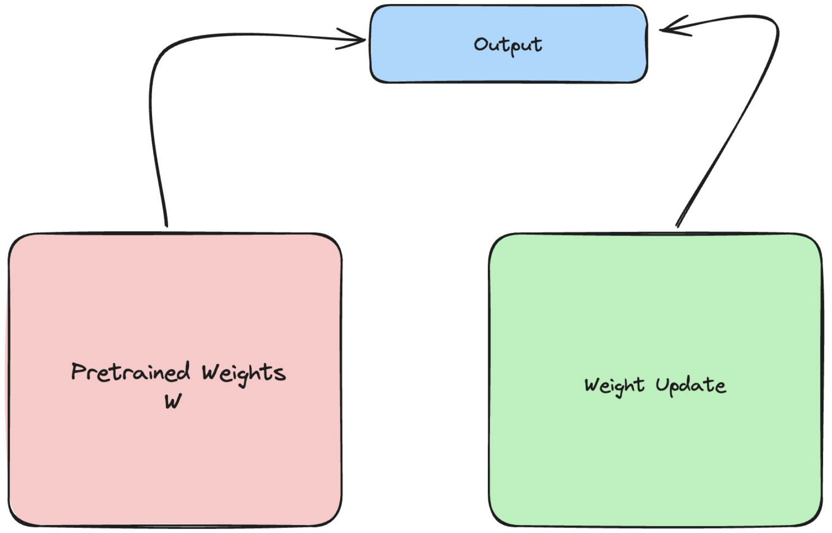
```python
def triton_(in_out_ptr0, in_ptr0, out_ptr0, xnumel, rnumel, XBLOCK : tl.constexpr):
    xnumel = 1
    rnumel = 10
    RBLOCK: tl.constexpr = 16
    xoffset = tl.program_id(0) * XBLOCK
    xindex = xoffset + tl.arange(0, XBLOCK)[:, None]
    xmask = xindex < xnumel
    rindex = tl.arange(0, RBLOCK)[None, :]
    roffset = 0
    rmask = rindex < rnumel
    r0 = rindex
    tmp0 = tl.load(in_ptr0 + (r0), rmask, other=0.0)
    tmp1 = tl_math.abs(tmp0)
    tmp2 = tl.broadcast_to(tmp1, [XBLOCK, RBLOCK])
    tmp4 = tl.where(rmask, tmp2, float("-inf"))
    tmp5 = triton_helpers.max2(tmp4, 1)[:, None]
    tmp6 = 1 / tmp5
    tmp7 = 127.0
    tmp8 = tmp6 * tmp7
    tmp9 = tmp8 * tmp0
    tmp10 = libdevice.nearbyint(tmp9)
    tmp11 = tmp10.to(tl.int8)
    tl.debug_barrier()
    tl.store(in_out_ptr0 + (tl.full([XBLOCK, 1], 0, tl.int32)), tmp8, None)
    tl.store(out_ptr0 + (tl.broadcast_to(r0, [XBLOCK, RBLOCK])), tmp11, rmask)
```

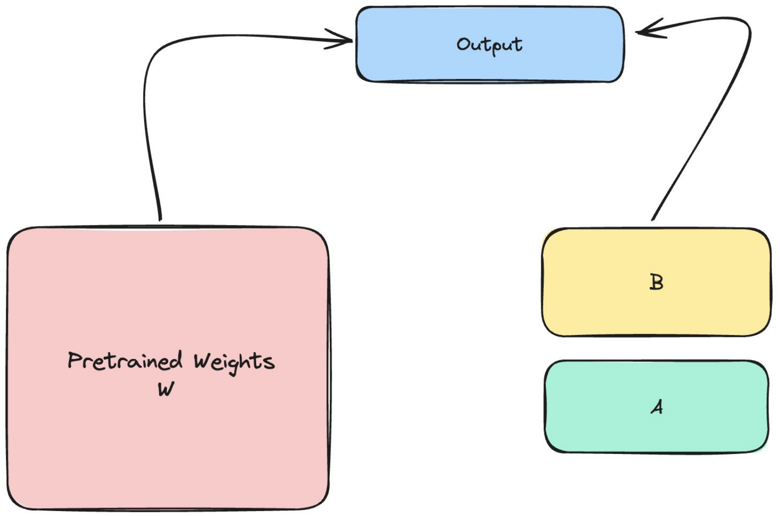
https://github.com/pytorch/ao

Back to gradients



Full finetuning vs LORA

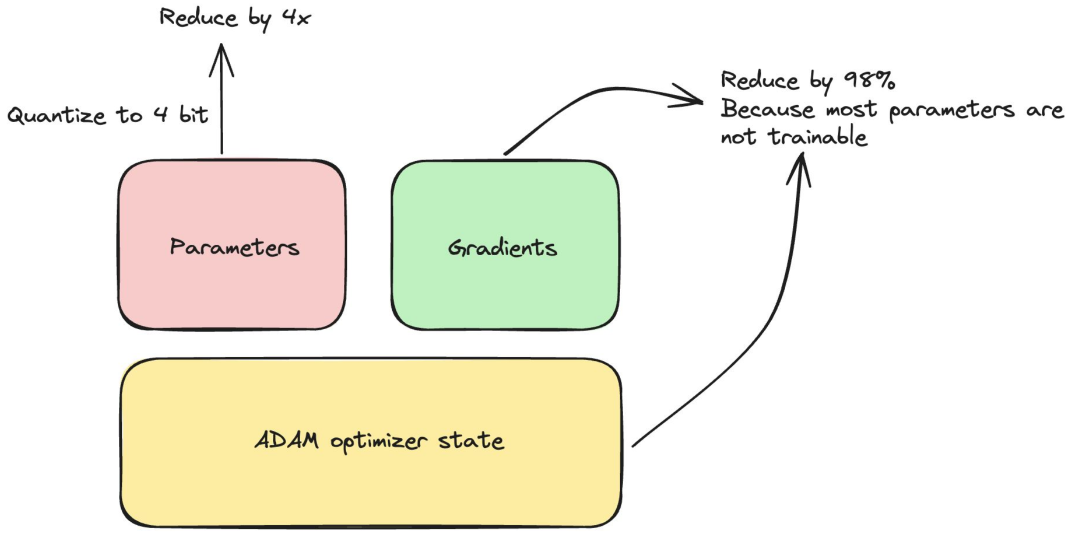




weight += (lora_B @ lora_A) * scaling

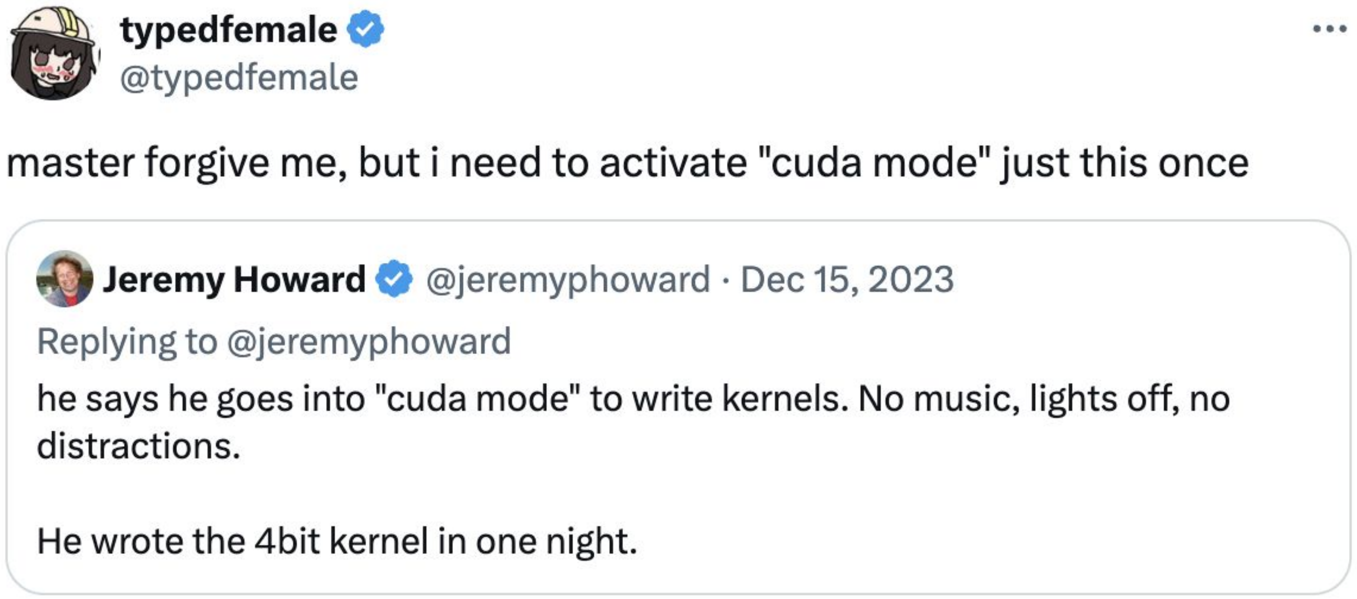
QLoRA

All winning entries for https://llm-efficiency-challenge.github.io/ used QLoRA



Implementing QLoRA

4000 lines of CUDA code
https://github.com/TimDettmers/bitsandbytes/blob/main/csrc/kernels.cu



typedfemale
@typedfemale

master forgive me, but i need to activate "cuda mode" just this once

Jeremy Howard @jeremyphoward · Dec 15, 2023
Replying to @jeremyphoward
he says he goes into "cuda mode" to write kernels. No music, lights off, no distractions.

He wrote the 4bit kernel in one night.

Forgot to mention some details

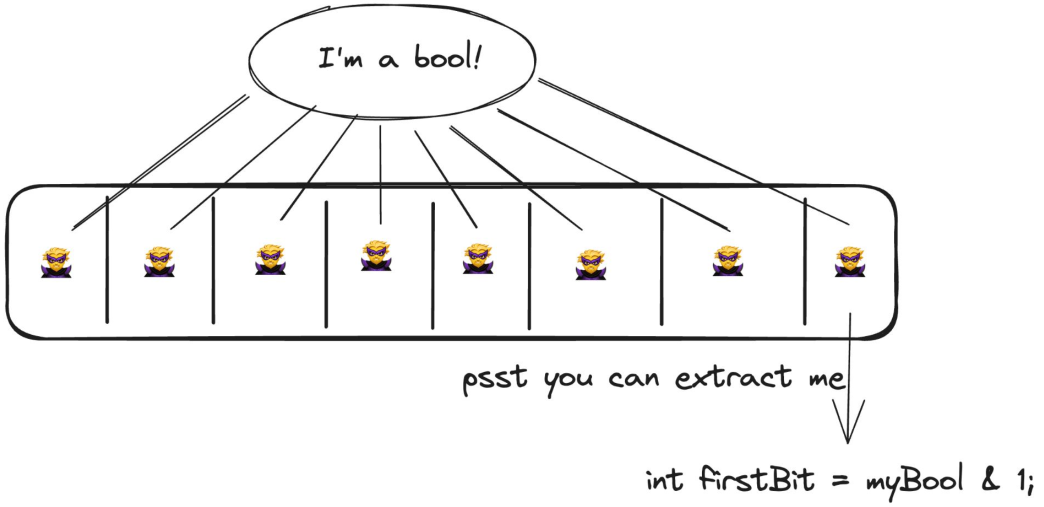
- Weights aren’t in int4 but NF4 which is closer to a normal distribution
- Can’t matrix multiply NF4 tensors, need to dequantize and matmul
- Remember how important the max is when doing the quant? Well you can’t use the same max for everything otherwise you’re too sensitive to outliers
- Quantization typically done in blocks with independent scales
- QLoRA quantizes the scales, double quantization!
- 😵‍💫
- Let’s look at some code https://github.com/pytorch/ao/blob/main/torchao/dtypes/nf4tensor.py

Bitpacking

PyTorch supports down to int8 https://pytorch.org/docs/stable/tensors.html

Elements of a tensor need to be byte (8 bit) addressable

C++ is the same a bool takes 8 bits of memory



But what if we wanted to implement a real Tensor

Probably feature PyTorch devs are most excited about
https://github.com/albanD/subclass_zoo/

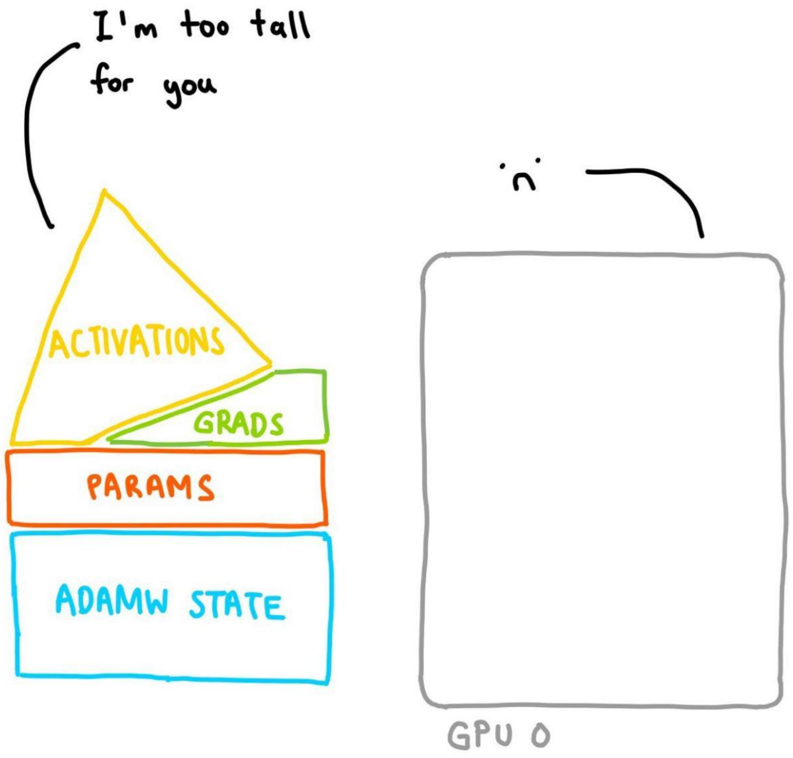
We can define what matrix multiplication over NF4 means using Python
https://github.com/pytorch/ao/pull/37 by @drisspg

```
return F.linear(input, weight.to(input.dtype))
```

But we can also define how FSDP would handle an NF4 Tensor
https://github.com/pytorch/ao/pull/150 i.e aten.split by @weifengpy

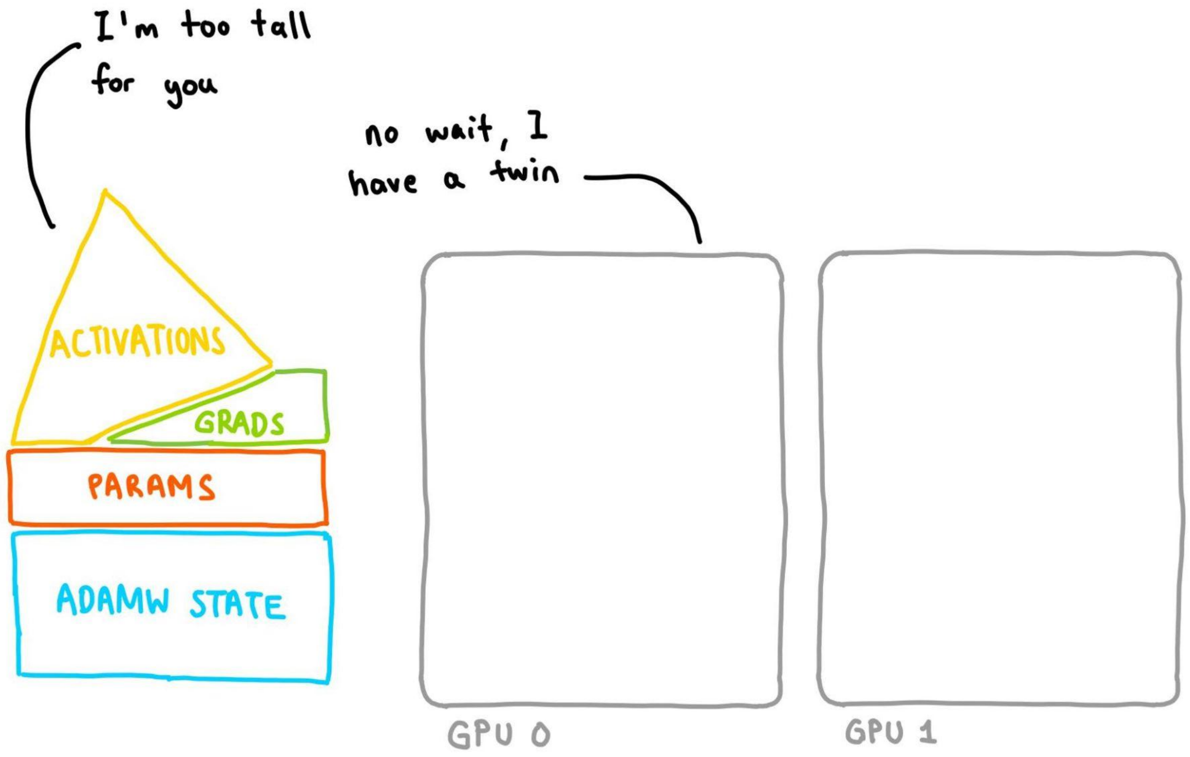
https://pytorch.org/docs/stable/tensors.html

One GPU was not enough…

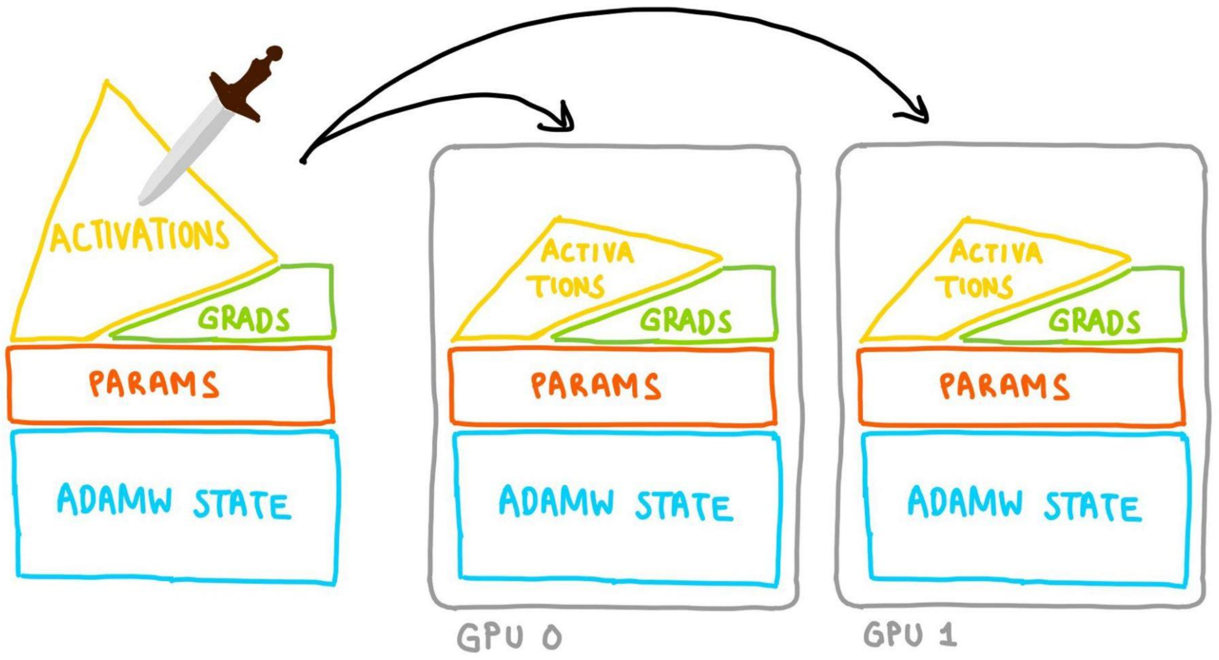


* DISCLAIMER: the model memory to the left does *not* include literally everything that'll take up memory during training, but is meant to be illustrative of the significant pieces.

But what if you had 2?

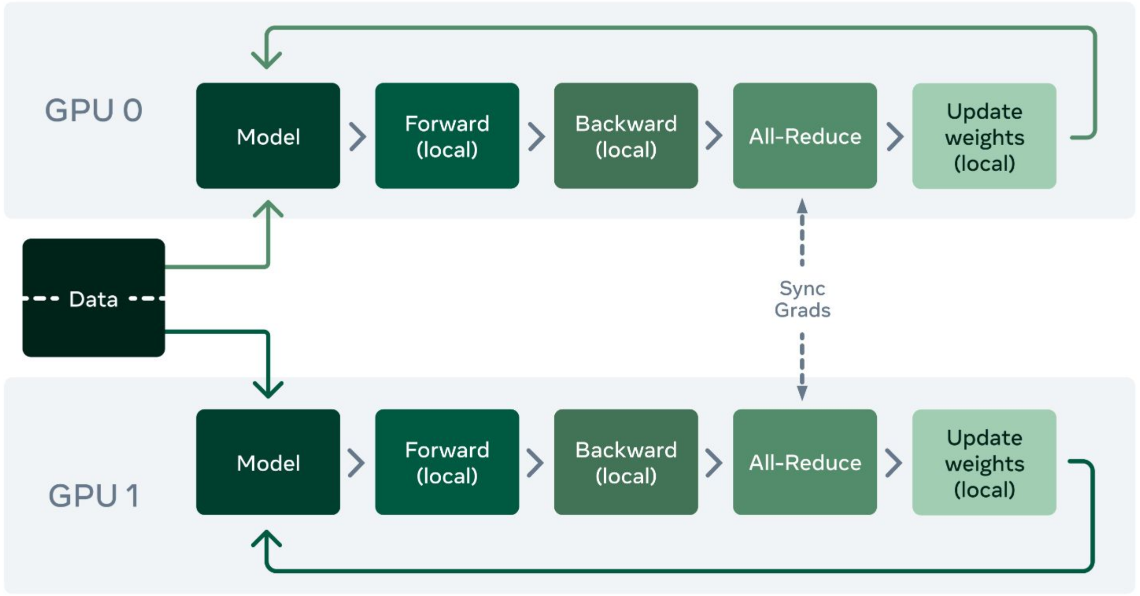


Let’s start with the obvious: parallelize the data (batch size)



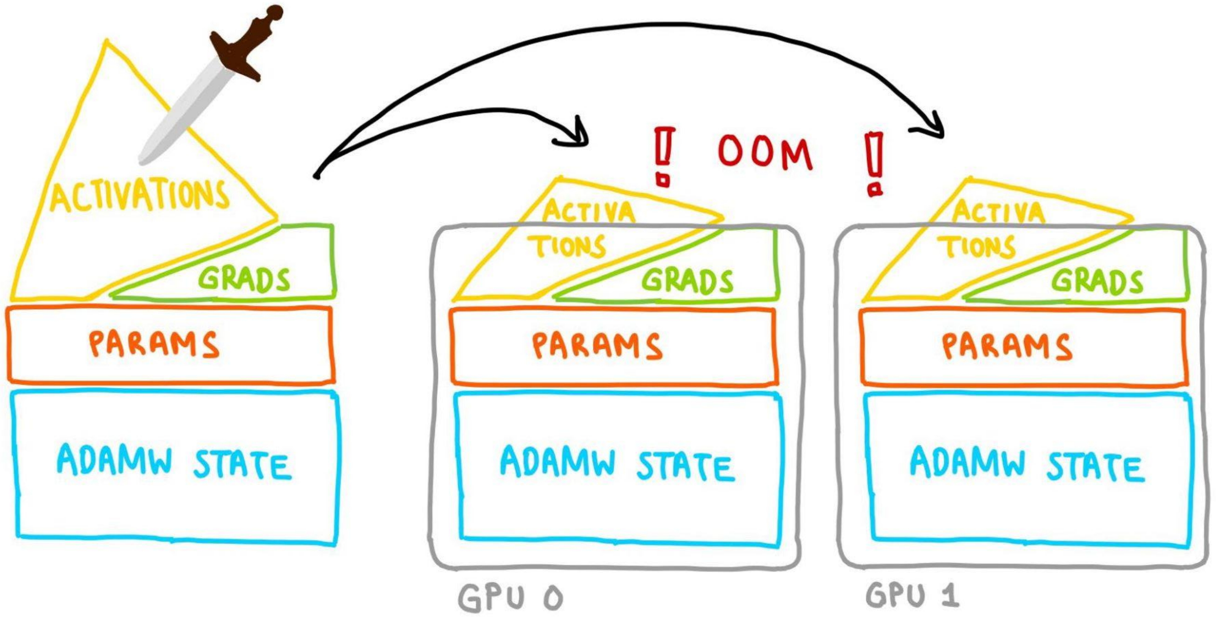
Sharding the batch size halves the activations. Everything else is duplicated.

Let’s start with the obvious: parallelize the data (batch size)



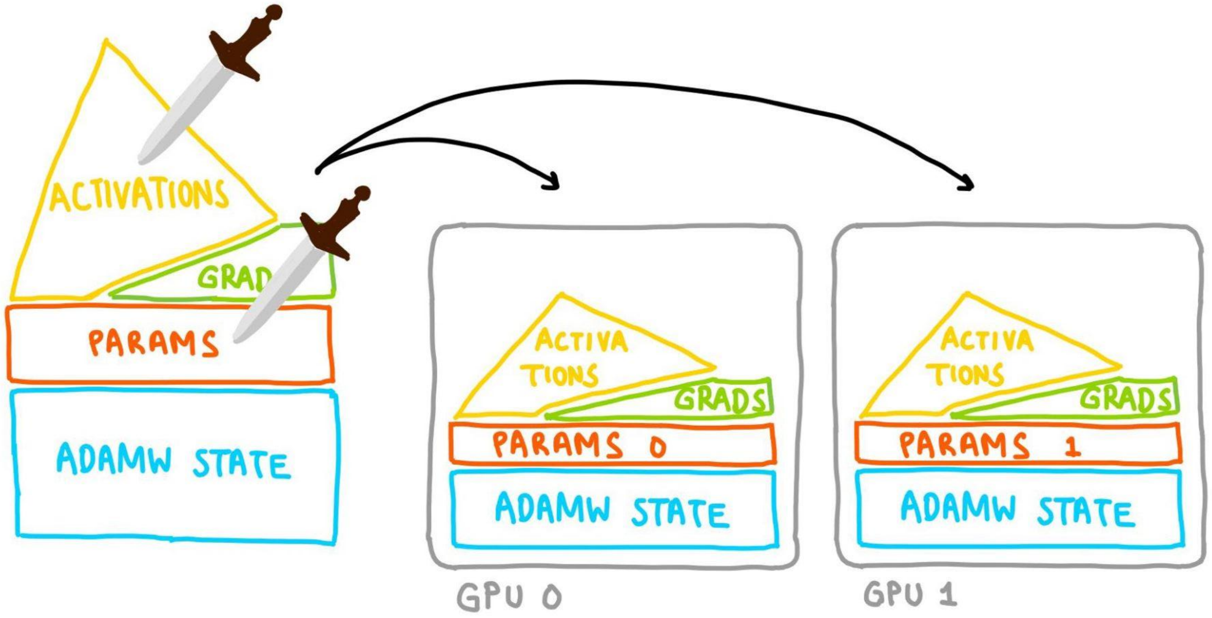
Note that we need to sync/sum the grads before the optim step with an all-reduce!
In general, techniques to lower memory require additional compute and management.

But what if that wasn’t enough?



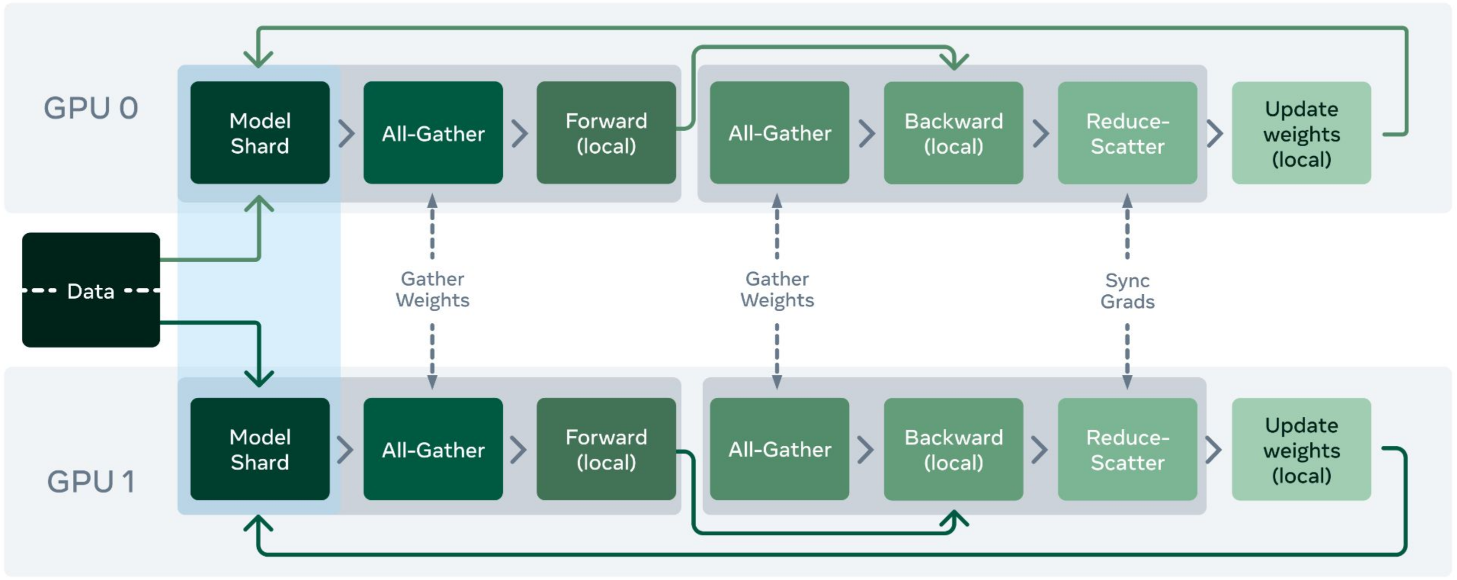
What else can we do?

Let's keep parallelizing! Shard the params too.



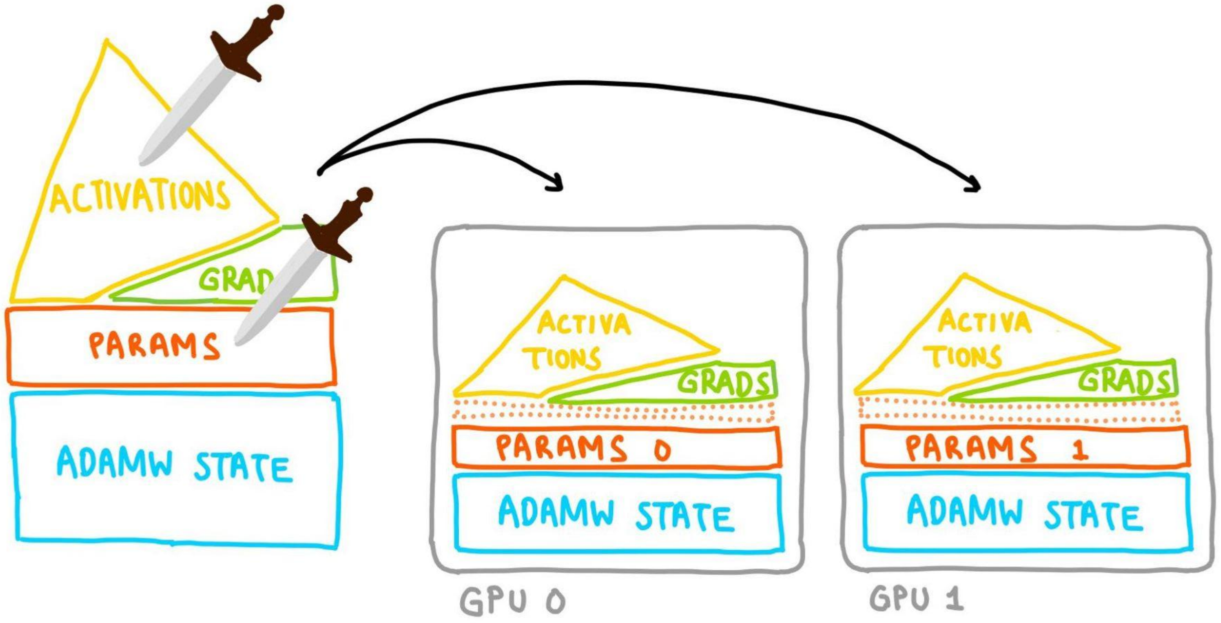
Sharding the params will in turn reduce gradient and optimizer memory.

Congrats you have discovered FSDP! - **f**ully **s**harded **d**ata **p**arallel



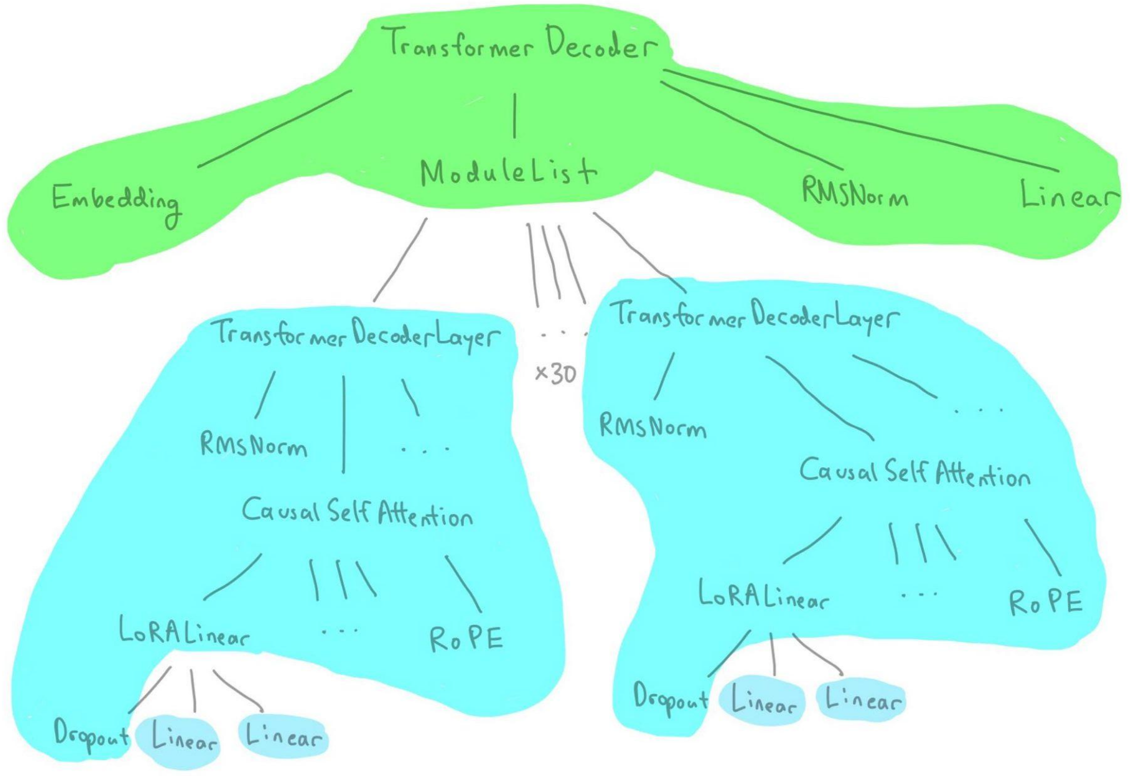
FSDP will bring in only a layer's weights at a time to avoid using too much memory.
As a result, we need more collectives to shuffle tensors between GPUs.

A slightly more accurate depiction of memory for a step in FSDP



⬚ Memory corresponding to the layer getting processed. Will be freed when the layer is done.

What constitutes a layer in FSDP?

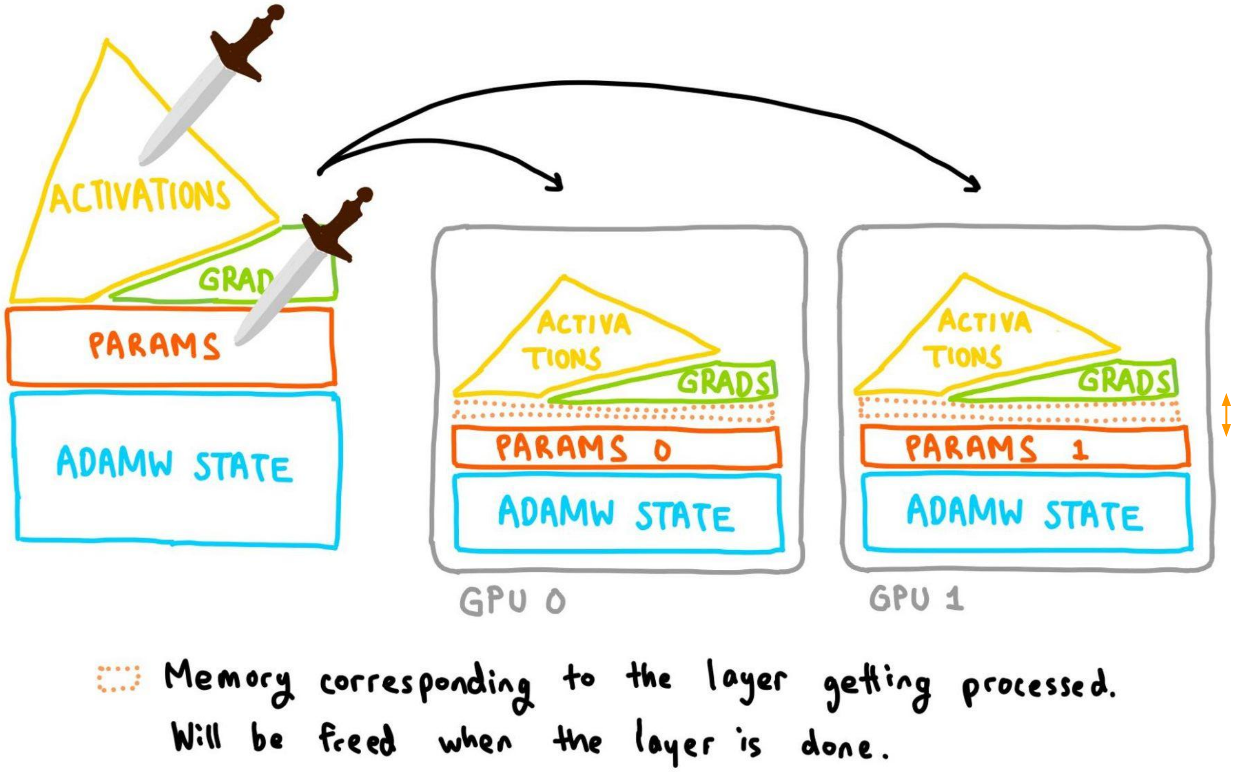


Every nn module is a tree of more nn modules.

The user's wrapping policy determines what gets treated as its own "layer".

This depicts a wrapping policy where `TransformerDecoderLayer` and `Linear` are specified.

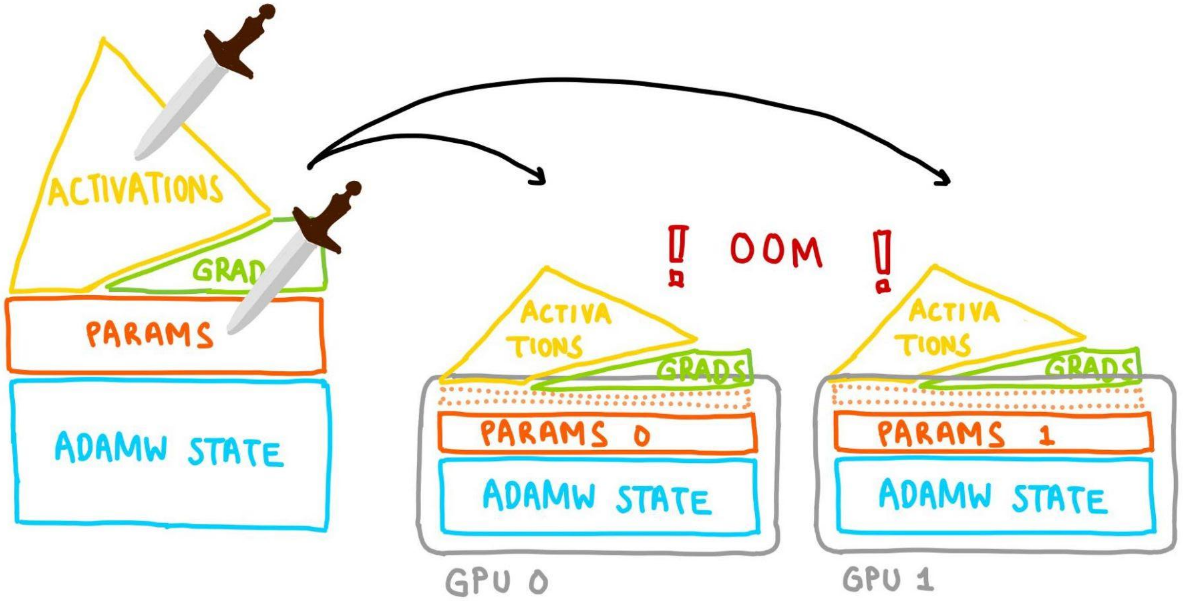
What you decide to wrap influences memory usage (and more)



The more "fine-grained" you wrap, the smaller that dotted memory will be.

Smaller blobs = less memory needs to be all-gathered at a time.

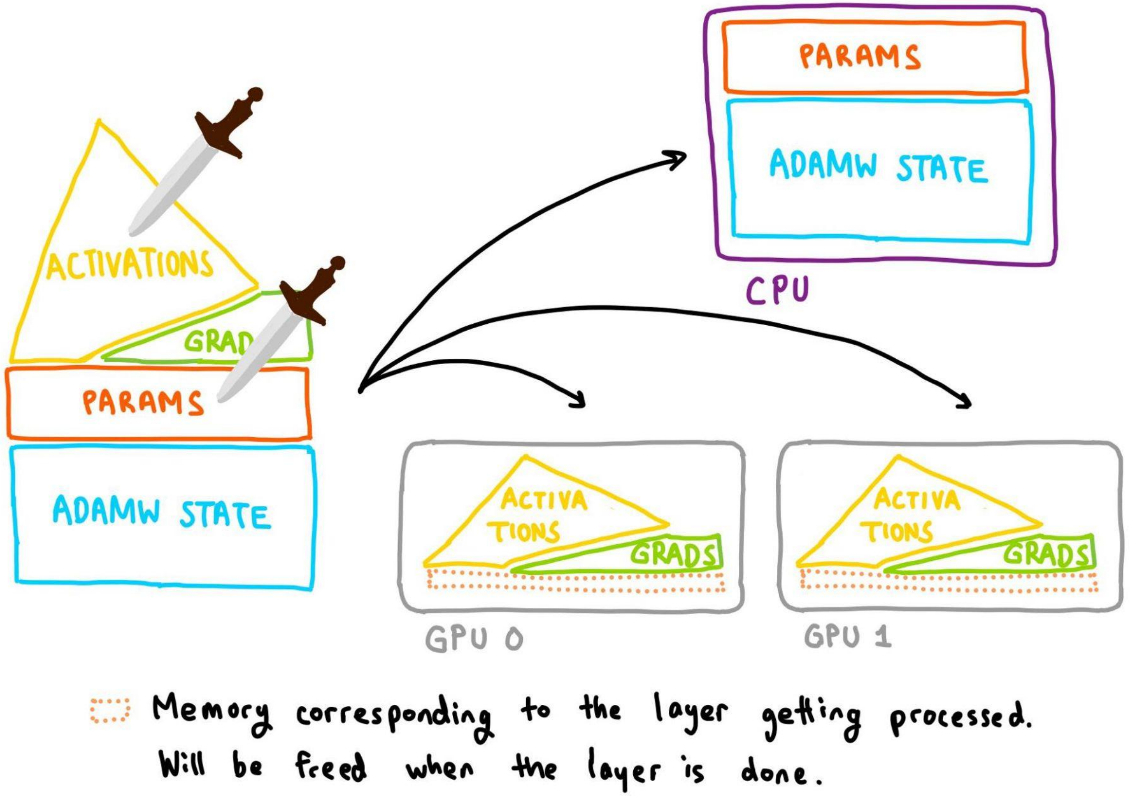
But what if after all your tweaking, you still OOM?



⬚ Memory corresponding to the layer getting processed. Will be freed when the layer is done.

What *else* can we do?

In comes CPU offloading!



Don’t forget about the CPU!

Just keep parameters on the CPU and move them to the GPU when computing forward + backward.

Note that the optimizer update will be done on CPU, so the optim state lives there too.

None of this is quite new…right?

I mean…okay, yes, FSDP has existed for a while, with all the features mentioned above.

And wonderful people have been using these features, like Answer.AI who built fsdp_qlora with FSDP x bnb to compose qLoRA and distributed.

BUT we’ve recently come out with *per-parameter* FSDP!

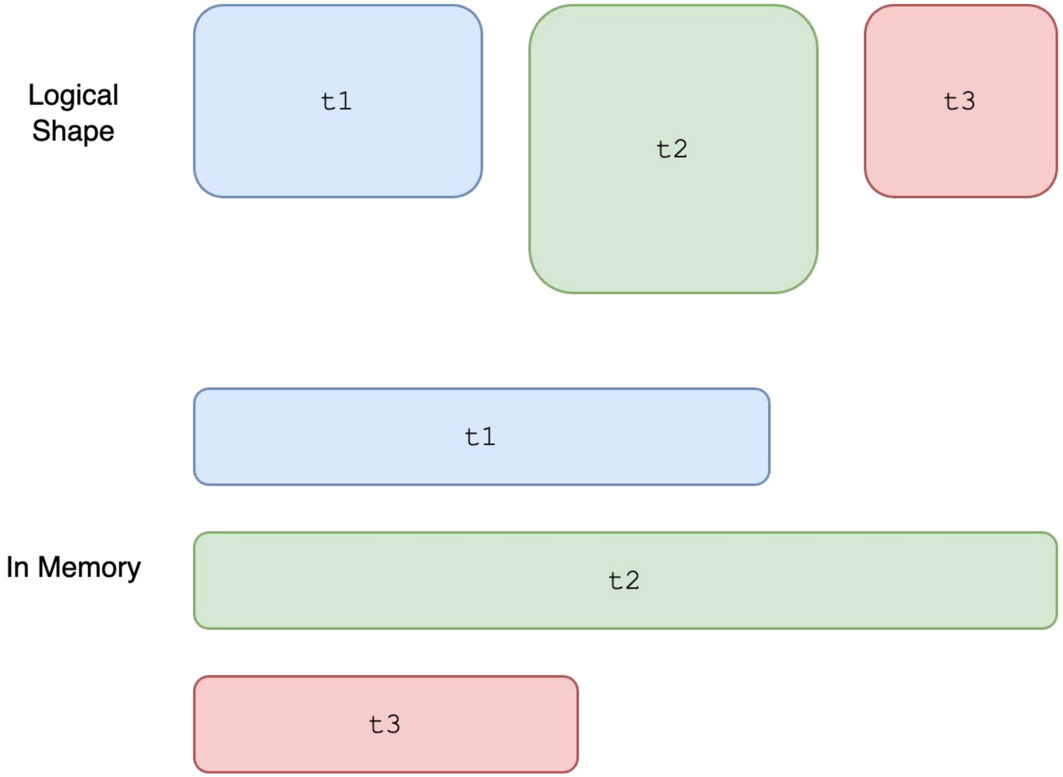
What is per-parameter FSDP?

Let’s start with the status quo: **flat-parameter FSDP1**. Say you have these params to shard across our two GPUs:

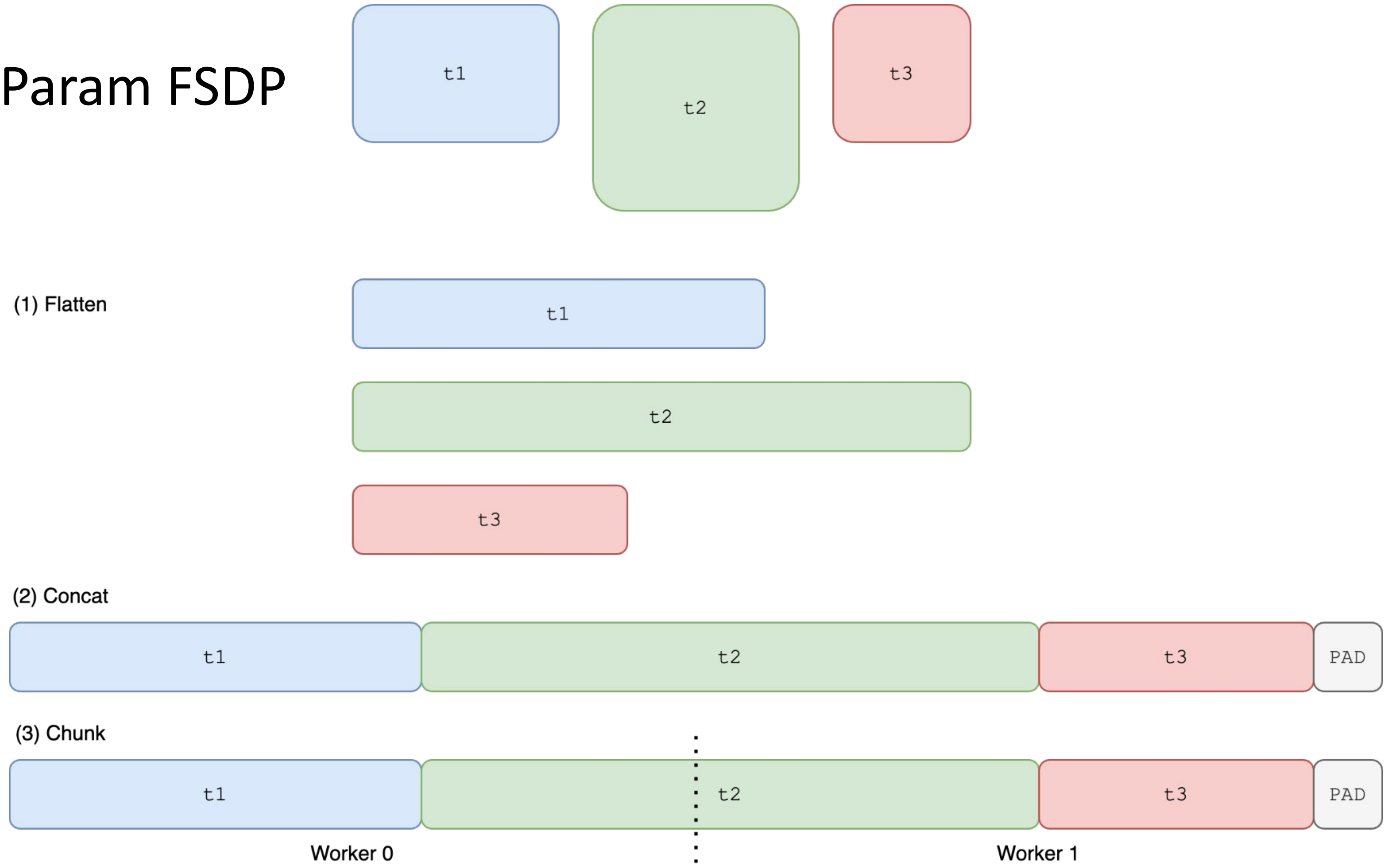
```
t1: (2, 3)
t2: (3, 3)
t3: (2, 2)
```

Goal: make all-gather efficient

Constraint: NCCL requires each GPU contribute same-size Tensors



FlatParam FSDP



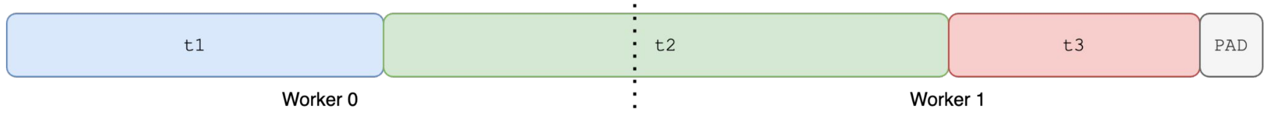
FlatParam FSDP

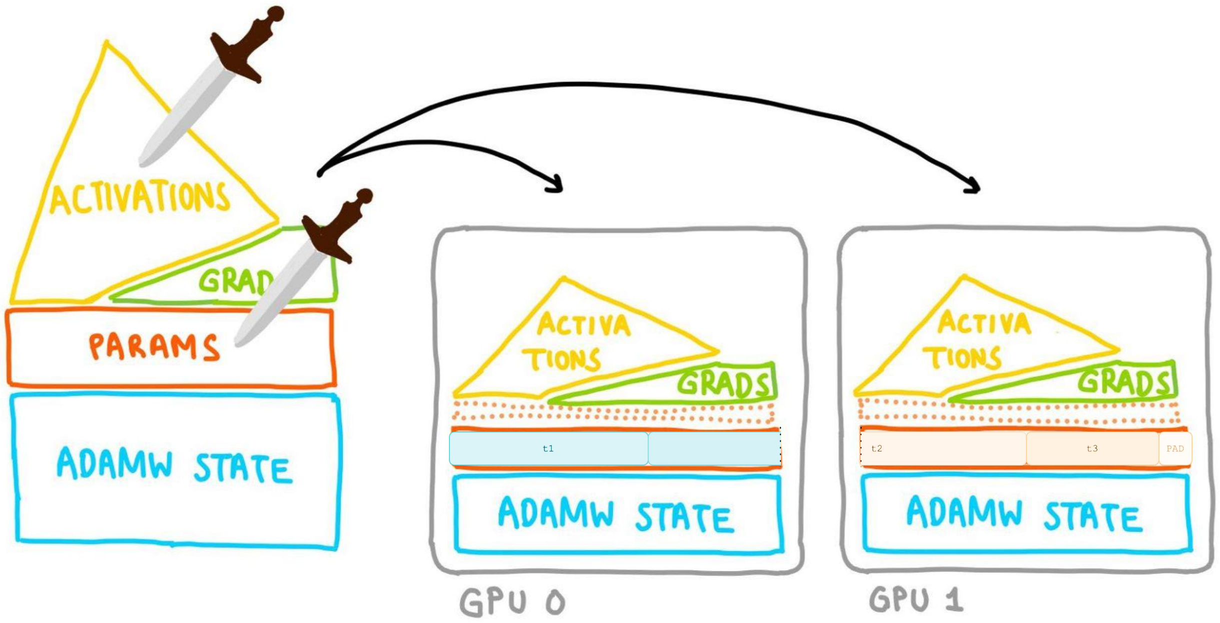
Each chunk is smooshed into **one** Tensor, which we call a FlatParameter.

This approach has its pros:

- Contiguous memory
- One can use **views** to retrieve `t1`, `t2`, `t3` (vs copy's)

but also its cons…





⬚ Memory corresponding to the layer getting processed. Will be freed when the layer is done.

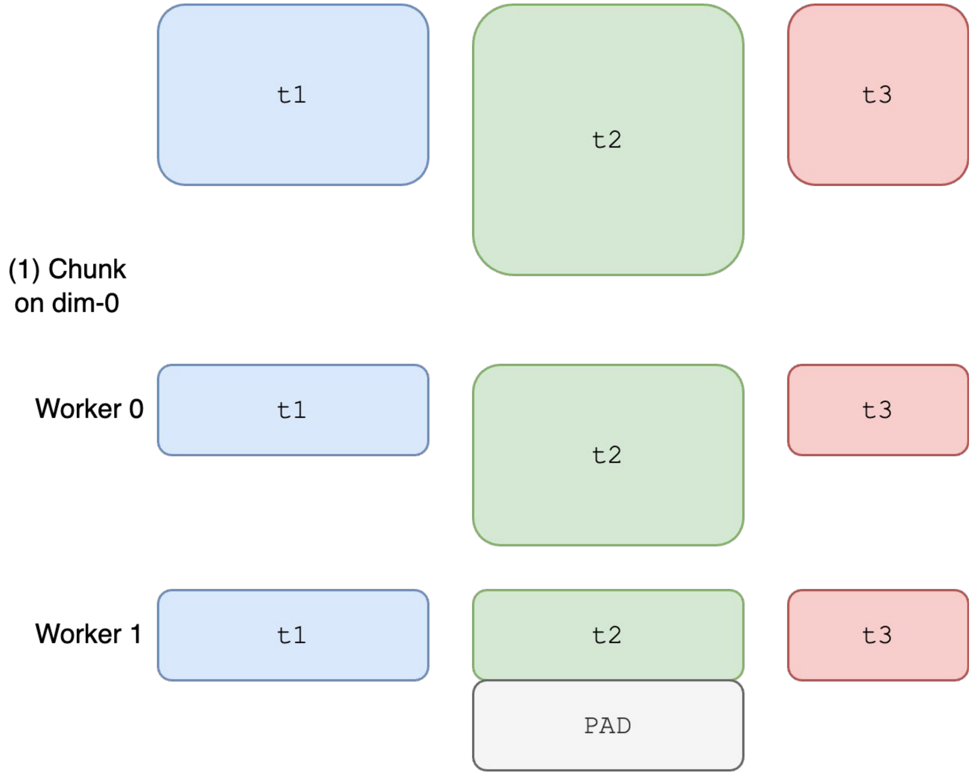
Another way to shard, dubbed “per parameter”

Remember you have these params to shard across our GPUs:

```
t1: (2, 3)
t2: (3, 3)
t3: (2, 2)
```

We can slice each parameter in half over dimension 0, and pad uneven slices.

This way, every param has representation on every machine.

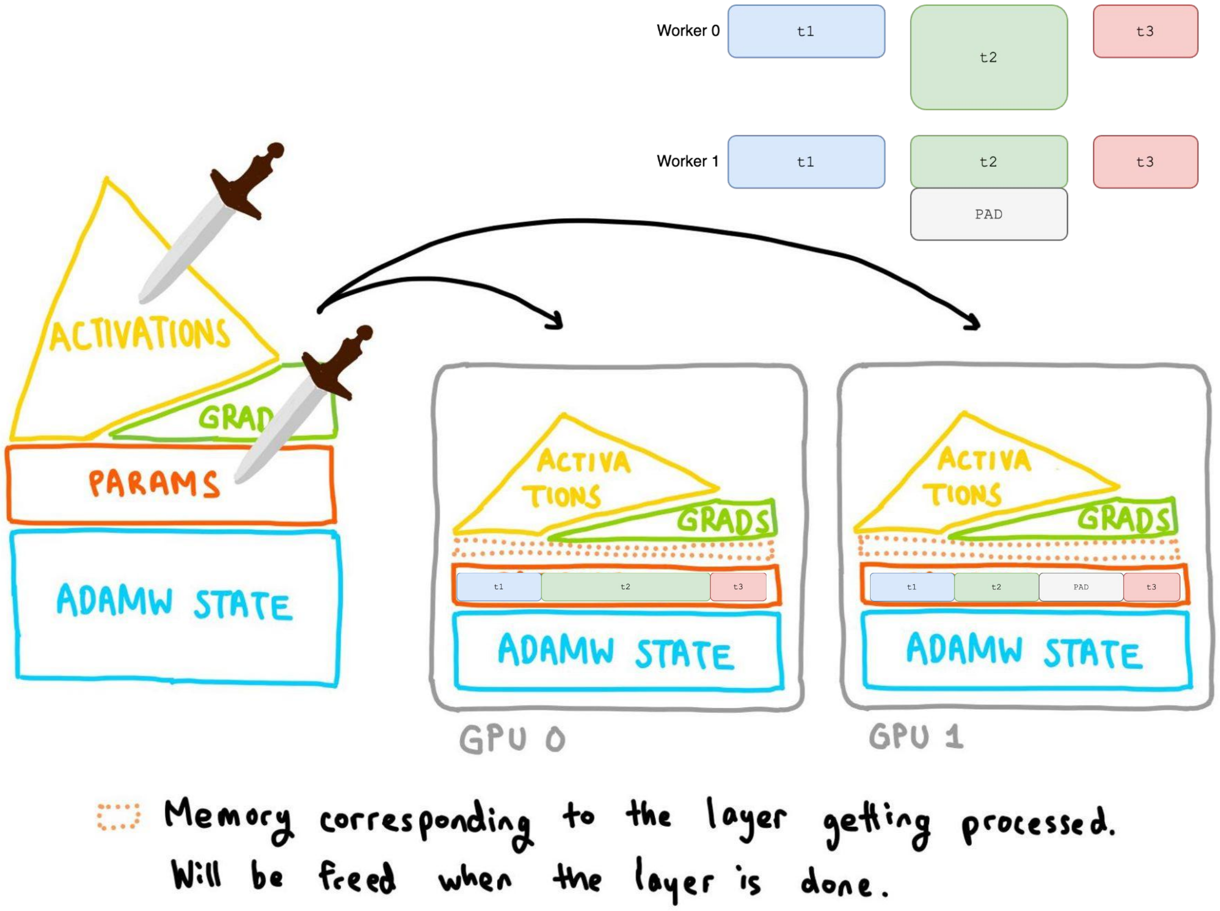


Per-param FSDP2

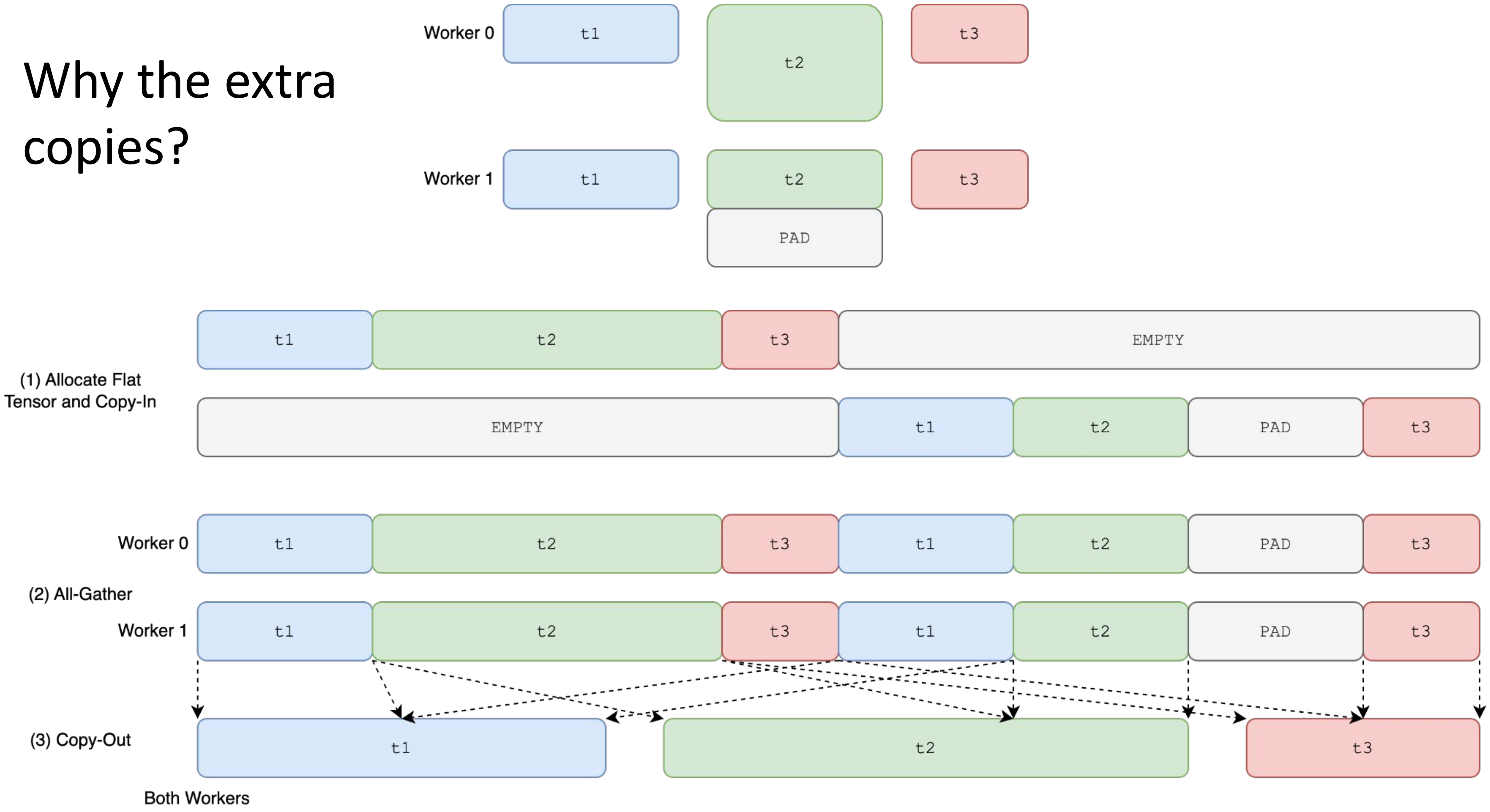
Each chunk is **many** Tensors, each a DTensor (D is for distributed).

This approach has a major pro: each param maintains its own identity (dtype, subclass, metadata).

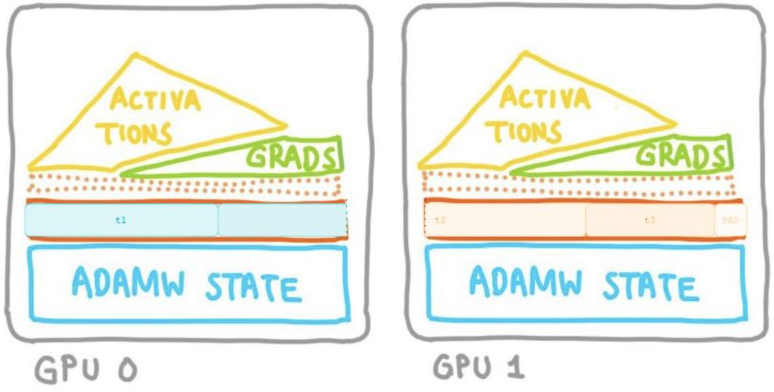
BUT does require extra copies ($$$ > views) during All-Gather.



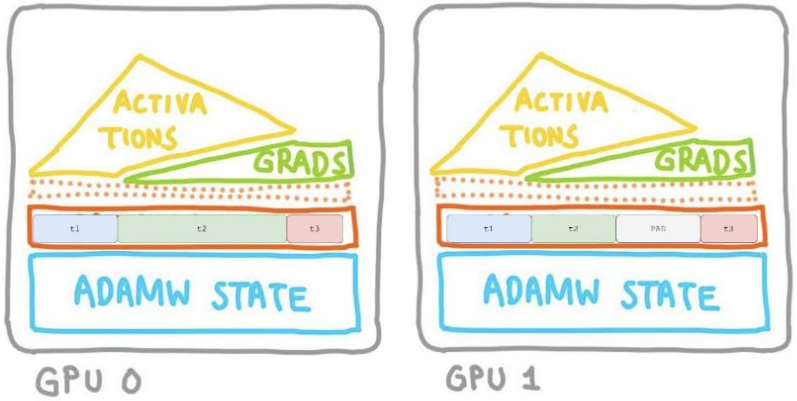
Why the extra copies?



Why do we think the “per-parameter-ness” is worth it?



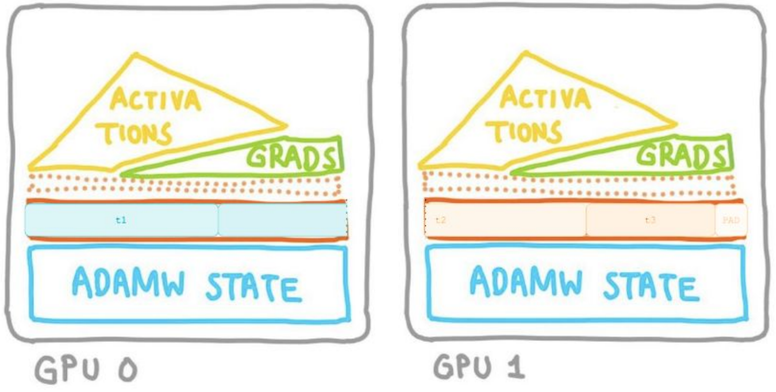
vs



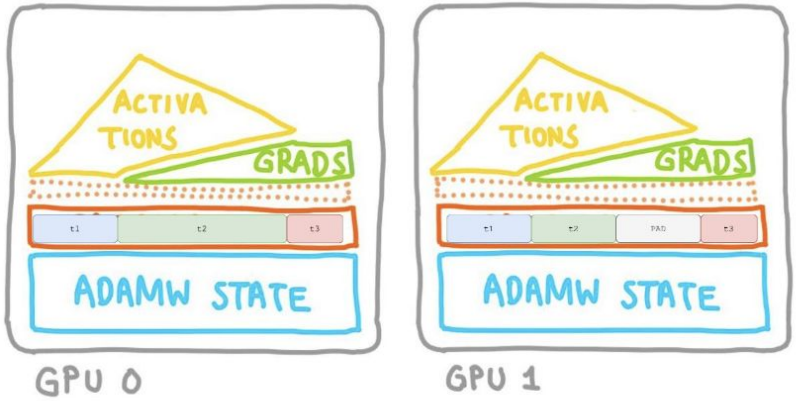
FlatParameter forces `t1`, `t2`, and `t3` to share dtype, requires_grad as it is one Tensor.

In per-params, `t1`, `t2`, and `t3` here can be themselves! They can have their own dtype, requires_grad.

Why do we think the "per-parameter-ness" is worth it?



vs



Think **quantization**: what if you wanted `t2` to be uint8 + `t1` to remain fullsized bf16?
Think **parameter freezing/LoRA**: what if `t2` is a frozen base weight while `t3` is the LoRA adapter?

You'd have to hack around FSDP1 concepts you'd get for free in FSDP2.

FSDP2 also has other cool pros, like deterministic memory

This is another major implementation change that actually guarantees deterministic memory:

Only 2 layers worth of memory will coexist at a time.

But it'll take another lecture to explain xD, for more details, see https://dev-discuss.pytorch.org/t/fsdp-cudacachingallocator-an-outsider-newb-perspective/1486

Why do we think the “per-parameter-ness” is worth it?

Well, it…

1. **just makes more sense**
   a. Every parameter is an evenly sliced version of itself in FSDP2
   b. Whereas in FSDP1, some parameters are entirely on 1 machine while others could be split across arbitrarily. Plus, every parameter belonging to a FlatParam must share dtype and subclass and requires_grad.
2. widens what could be wrapped by FSDP into a layer
3. unlocks param-wise optimizers, like AdaFactor
4. **composes with other distributed parallelisms** (TP, PP) through DTensor, as tensor structure is maintained

FSDP2 also has other cool pros, like deterministic memory

Due to how FSDP1 implemented its rate limiter on CPU, it couldn’t actually guarantee:

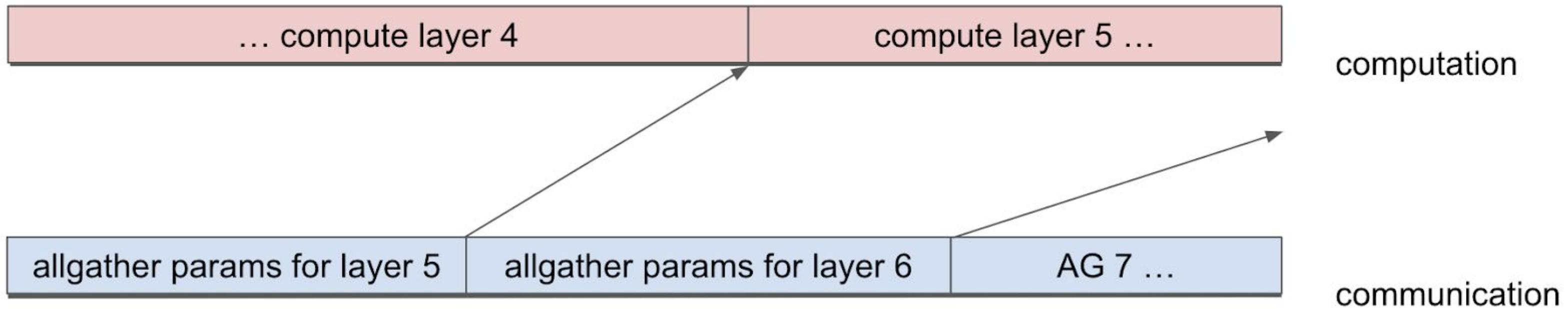
Only 2 layers worth of memory will coexist at a time.

For example, using CPU offloading sometimes caused *more* memory usage!

FSDP2 moved the burden of rate limiting from CPU to CUDA events, so now this guarantee can actually be met :D

For more details, see https://dev-discuss.pytorch.org/t/fsdp-cudacachingallocator-an-outsider-newb-perspective/1486

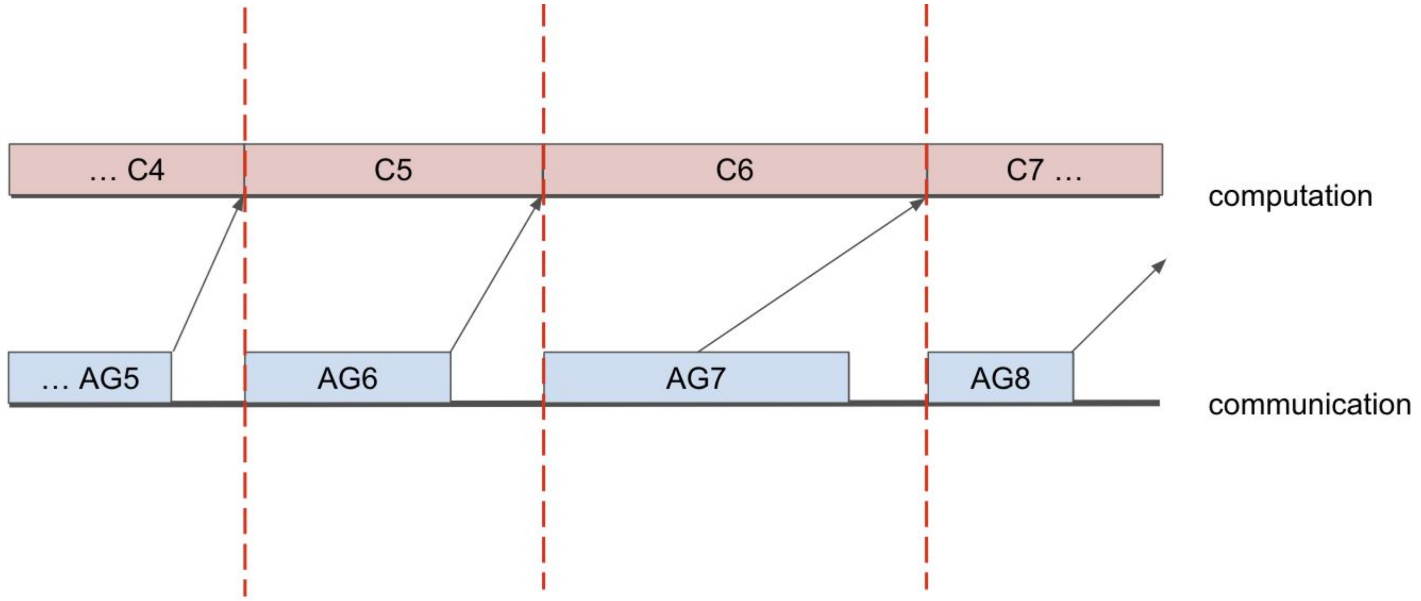
Our implementation overlaps communication with computation



This way, the all-gathers are imperceptible in terms of runtime!

Note that this requires **prefetching** the next layer's parameters so that they could be ready by the time its compute starts.

But we do it methodically to avoid peaking memory



The FSDP rate limiter forces prefetching to wait until the previous layer is freed.

Desired guarantee: only 2 layers worth of memory will coexist at a time.

So let's take FSDP2 out for a swim

Answer.AI had already successfully composed FSDP1 with QLoRA, but only after expertly maneuvering through its limitations.

e.g., "FSDP was not copying the quantization information needed for each shard to use the model! That's because FSDP is quite opinionated on the subset of data it will sync between GPUs"

We want to offer cleaner, more general solutions to composing distributed with low precision parameters, so why not start here, with FSDP2 x NF4?

So we did! https://github.com/pytorch/torchtune/pull/909

Cleaner and more composable is always good, but how do we do on perf? Let's find out!

The plan

1. Get some GPUs
2. Run a benchmark on Answer.AI’s train.py
3. Run the same benchmark on Wei’s torchtune recipe
4. Wait…were those actually the same benchmark?
5. Make sure what I’m measuring was 🍎 ⇔ 🍏 and not 🍎 ⇔ 🍊
6. Record the gaps
7. Investigate and fill the gaps if possible

Getting some GPUs

I rented myself a dual setup on vast.ai

- 2 RTX 3090s, 24 GB VRAM each
- 117 GB RAM
- 12 cores => required `torch.set_num_threads(8)`
- PCIE 3.0 16x, with 9.0 GB/s bandwidth each
- CUDA 12.2

Running a benchmark on Answer.AI’s train.py

llama2-7B, context length 2048

batch size	peak memory	runtime for a step
8	15.03 GiB	13.9s
10	18.05 GiB	16.9s
below needs PYTORCH_CUDA_ALLOC_CONF=expandable_segments:True		
12	21.06 GiB	19.9s
+	OOM	N/A

llama2-7B, context length 2048, with CPU offloading

batch size	peak memory	runtime for a step
8	12.88 GiB	14.0s
10	15.89 GiB	17.5s
below needs PYTORCH_CUDA_ALLOC_CONF=expandable_segments:True		
12	18.91 GiB	20.9s
14	21.92 GiB	23.6s
+	OOM	N/A

Thanks Answer.AI peeps on CUDA MODE for sending me benchmarks to try!

Running a benchmark on Answer.AI's train.py

llama2-7B, context length 2048

batch size	peak memory	runtime for a step
8	15.03 GiB	13.9s
10	18.05 GiB	16.9s
below needs PYTORCH_CUDA_ALLOC_CONF=expandable_segments:True		
12	21.06 GiB	19.9s
+	OOM	N/A

llama2-7B, context length 2048, with CPU offloading

batch size	peak memory	runtime for a step
8	12.88 GiB	14.0s
10	15.89 GiB	17.5s
below needs PYTORCH_CUDA_ALLOC_CONF=expandable_segments:True		
12	18.91 GiB	20.9s
14	21.92 GiB	23.6s
+	OOM	N/A

I decided to focus on just one of these to do an apples to apples comparison.

```
python train.py --model_name meta-llama/Llama-2-7b-hf --batch_size 8
--context_length 2048 --train_type qlora --use_gradient_checkpointing True
--reentrant_checkpointing True --dataset dummy --dataset_samples 48
```

Running the same benchmark on Wei's torchtune recipe

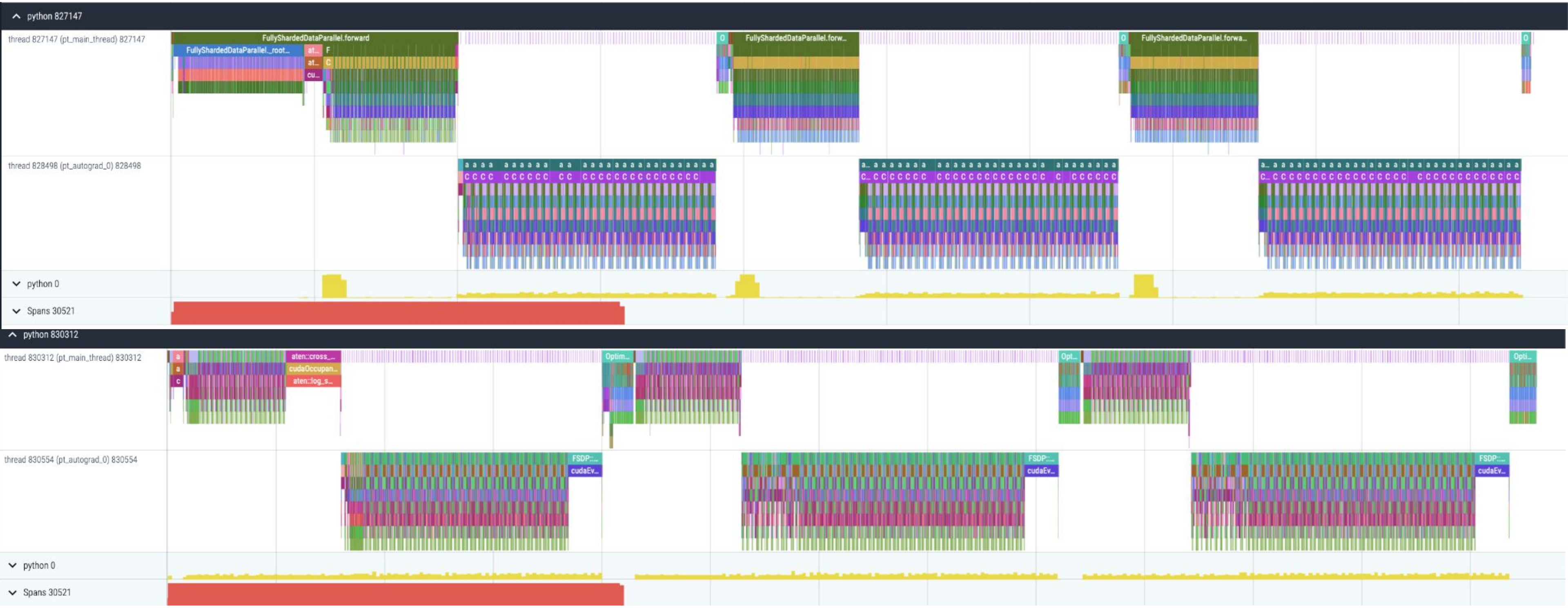
`tune run --nnodes 1 --nproc_per_node 2 lora_finetune_fsdp2 --config recipes/configs/dev/llama2/7B_qlora_fsdp2.yaml` * with tweaks to align the configs

	batch size	peak memory	runtime for a step
train.py	8	12.88 GiB	14.0s
torchtune	8	12.60 GiB	16.5s

Since FSDP2 is stricter about memory and requires extra copies, it would be easy to chalk up the differences above as expected.

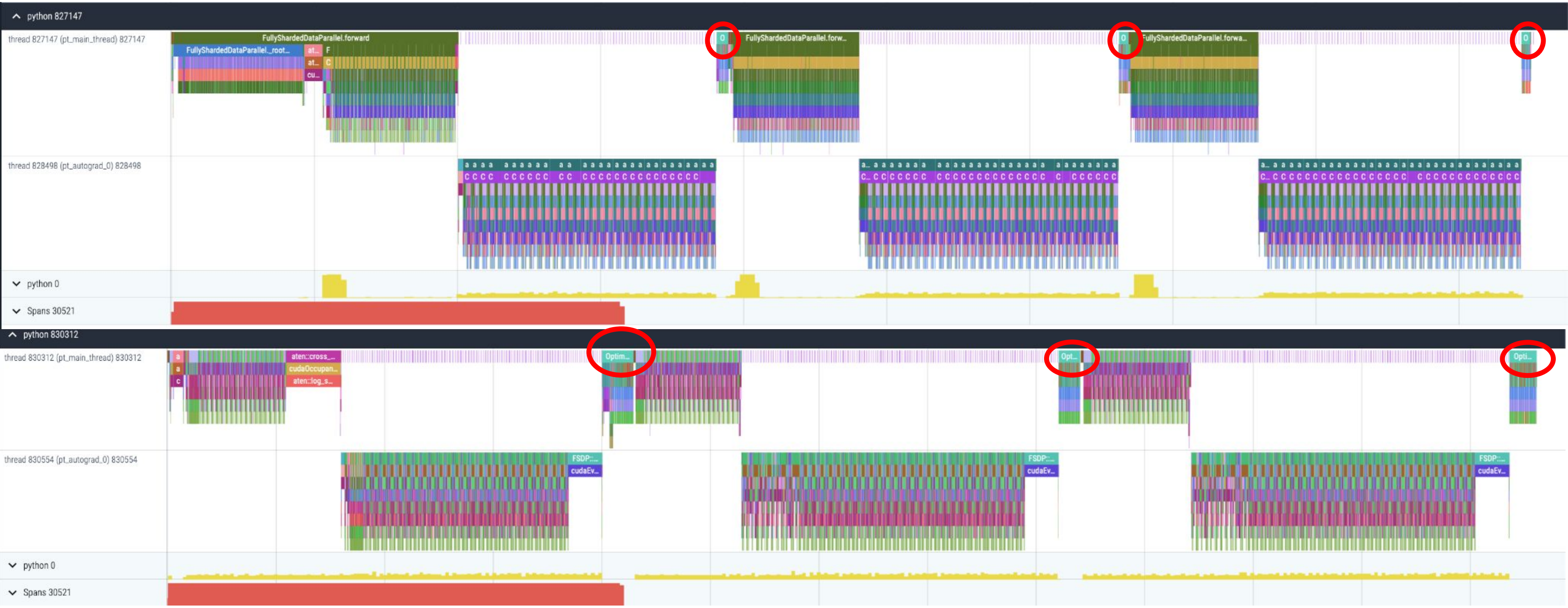
But, nah, we have to be diligent! And very quickly, one glance at the trace revealed troubling shenanigans.

Wait…are those actually the same benchmark?


python 827147
thread 827147 (pt_main_thread) 827147
FullyShardedDataParallel.forward
FullyShardedDataParallel._root_
FullyShardedDataParallel.forw…
FullyShardedDataParallel.forwa…
thread 828498 (pt_autograd_0) 828498
python 0
Spans 30521
python 830312
thread 830312 (pt_main_thread) 830312
aten::cross_…
cudaOccupan…
aten::log_s…
Optim…
thread 830554 (pt_autograd_0) 830554
FSDP::…
cudaEv…
python 0
Spans 30521

Spot the difference!

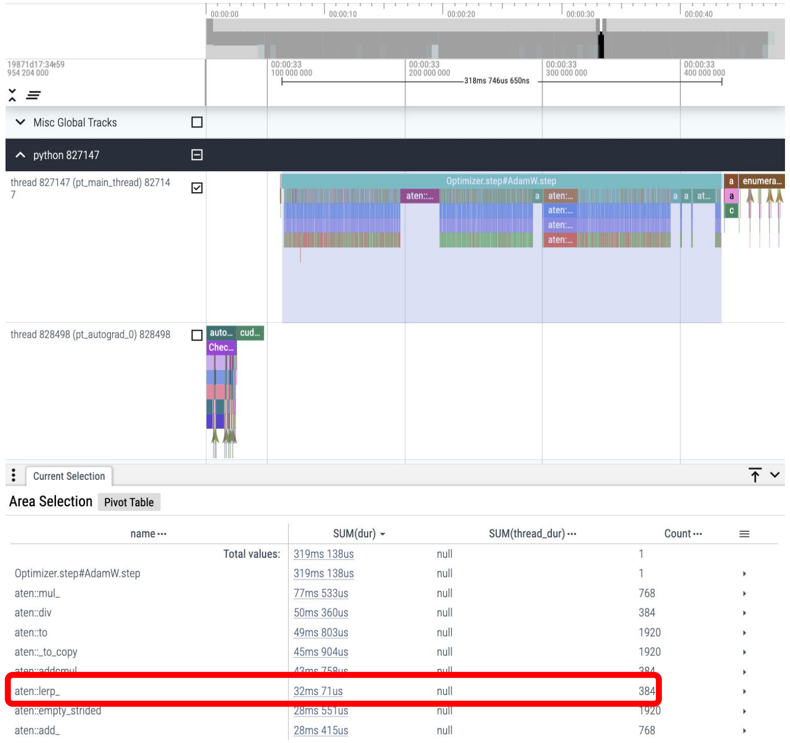
Wait…are those actually the same benchmark?


python 827147
thread 827147 (pt_main_thread) 827147
FullyShardedDataParallel.forward
FullyShardedDataParallel._root_
FullyShardedDataParallel.forw...
FullyShardedDataParallel.forwa...
thread 828498 (pt_autograd_0) 828498
python 0
Spans 30521
python 830312
thread 830312 (pt_main_thread) 830312
aten::cross_...
cudaOccupan...
aten::log_s...
Optim...
Opt...
Opti...
thread 830554 (pt_autograd_0) 830554
FSDP::...
cudaEv...
python 0
Spans 30521

Why were the optimizer steps so much bigger in the torchtune trace?

Wait…are those actually the same benchmark?

train.py trace



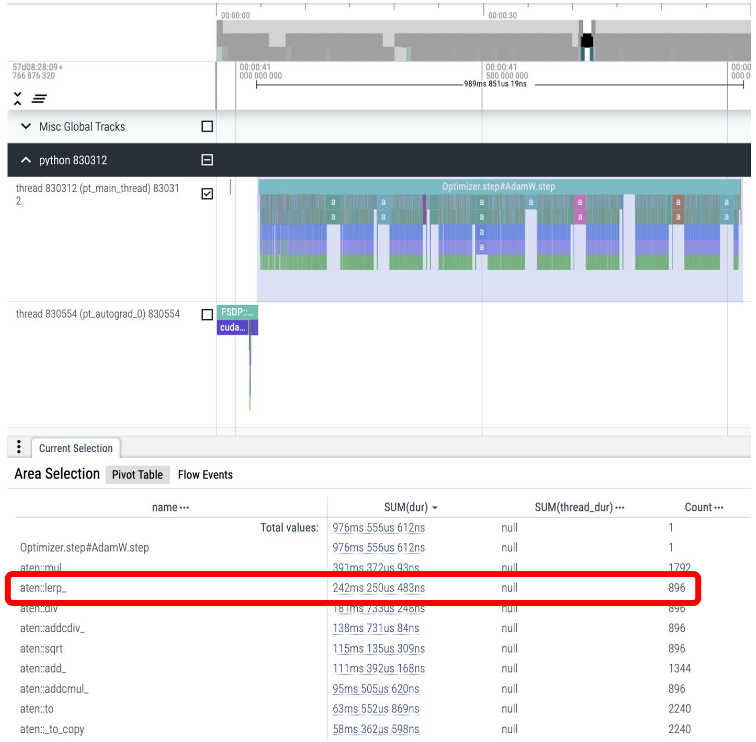
19871d17:34:59 954 204 000
Misc Global Tracks
python 827147
thread 827147 (pt_main_thread) 827147
thread 828498 (pt_autograd_0) 828498
Optimizer.step#AdamW.step
318ms 746us 650ns

Current Selection

Area Selection | Pivot Table

name	SUM(dur)	SUM(thread_dur)	Count
Total values:	319ms 138us	null	1
Optimizer.step#AdamW.step	319ms 138us	null	1
aten::mul_	77ms 533us	null	768
aten::div	50ms 360us	null	384
aten::to	49ms 803us	null	1920
aten::_to_copy	45ms 904us	null	1920
aten::addcmul_	43ms 758us	null	384
aten::lerp_	32ms 71us	null	384
aten::empty_strided	28ms 551us	null	1920
aten::add_	28ms 415us	null	768

torchtune trace



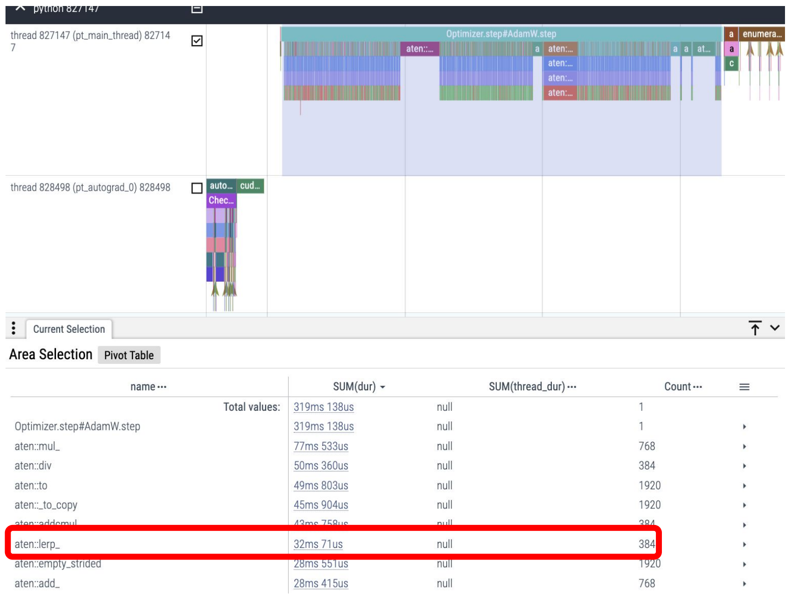
57d08:28:09+766 876 320
Misc Global Tracks
python 830312
thread 830312 (pt_main_thread) 830312
thread 830554 (pt_autograd_0) 830554
Optimizer.step#AdamW.step
989ms 851us 19ns

Current Selection

Area Selection | Pivot Table | Flow Events

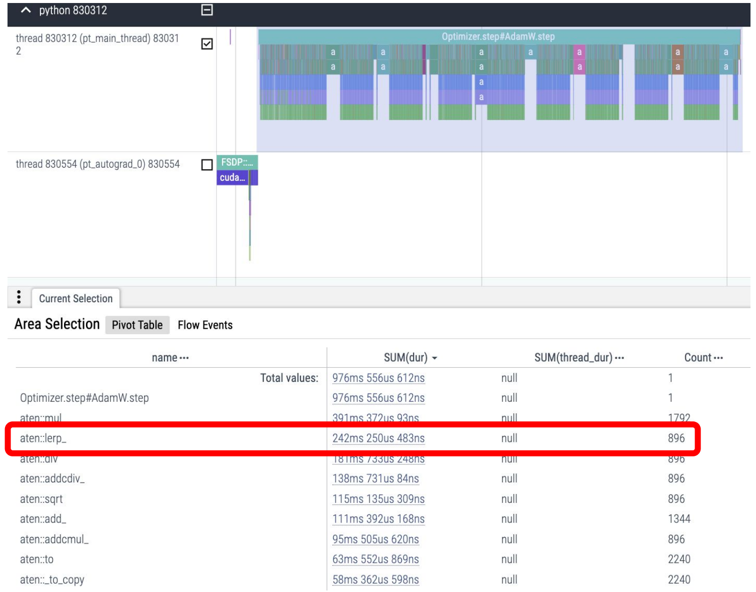
name	SUM(dur)	SUM(thread_dur)	Count
Total values:	976ms 556us 612ns	null	1
Optimizer.step#AdamW.step	976ms 556us 612ns	null	1
aten::mul	391ms 372us 93ns	null	1792
aten::lerp_	242ms 250us 483ns	null	896
aten::div	181ms 733us 248ns	null	896
aten::addcdiv_	138ms 731us 84ns	null	896
aten::sqrt	115ms 135us 309ns	null	896
aten::add_	111ms 392us 168ns	null	1344
aten::addcmul_	95ms 505us 620ns	null	896
aten::to	63ms 552us 869ns	null	2240
aten::_to_copy	58ms 362us 598ns	null	2240

Wait…are those actually the same benchmark?


python 827147
thread 827147 (pt_main_thread) 827147
thread 828498 (pt_autograd_0) 828498
Optimizer.step#AdamW.step
aten::...
enumera...
auto...
cud...
Chec...
Current Selection

Area Selection Pivot Table

name	SUM(dur)	SUM(thread_dur)	Count
Total values:	319ms 138us	null	1
Optimizer.step#AdamW.step	319ms 138us	null	1
aten::mul_	77ms 533us	null	768
aten::div	50ms 360us	null	384
aten::to	49ms 803us	null	1920
aten::_to_copy	45ms 904us	null	1920
aten::addcmul_	43ms 758us	null	384
aten::lerp_	32ms 71us	null	384
aten::empty_strided	28ms 551us	null	1920
aten::add_	28ms 415us	null	768

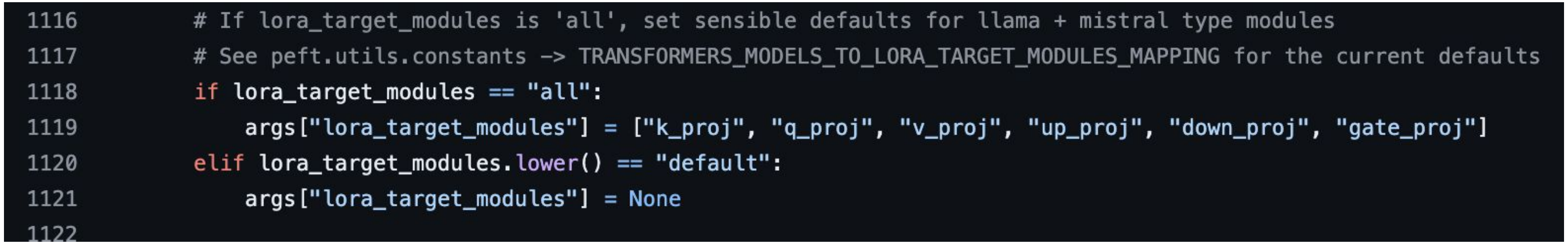

python 830312
thread 830312 (pt_main_thread) 830312
thread 830554 (pt_autograd_0) 830554
Optimizer.step#AdamW.step
FSDP::...
cuda...
Current Selection

Area Selection Pivot Table Flow Events

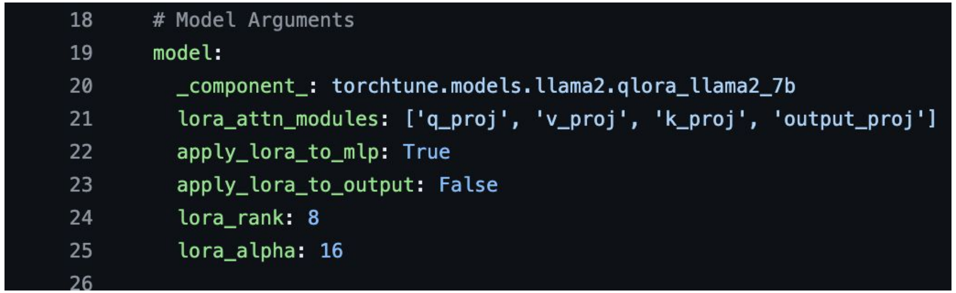
name	SUM(dur)	SUM(thread_dur)	Count
Total values:	976ms 556us 612ns	null	1
Optimizer.step#AdamW.step	976ms 556us 612ns	null	1
aten::mul	391ms 372us 93ns	null	1792
aten::lerp_	242ms 250us 483ns	null	896
aten::div	181ms 733us 248ns	null	896
aten::addcdiv_	138ms 731us 84ns	null	896
aten::sqrt	115ms 135us 309ns	null	896
aten::add_	111ms 392us 168ns	null	1344
aten::addcmul_	95ms 505us 620ns	null	896
aten::to	63ms 552us 869ns	null	2240
aten::_to_copy	58ms 362us 598ns	null	2240

Aha! torchtune was training more parameters than Answer.AI's train.py config.
448 - 384 = 64 extra params! Any guesses where they came from?

Wait…are those actually the same benchmark?



```
        # If lora_target_modules is 'all', set sensible defaults for llama + mistral type modules
        # See peft.utils.constants -> TRANSFORMERS_MODELS_TO_LORA_TARGET_MODULES_MAPPING for the current defaults
        if lora_target_modules == "all":
            args["lora_target_modules"] = ["k_proj", "q_proj", "v_proj", "up_proj", "down_proj", "gate_proj"]
        elif lora_target_modules.lower() == "default":
            args["lora_target_modules"] = None

```

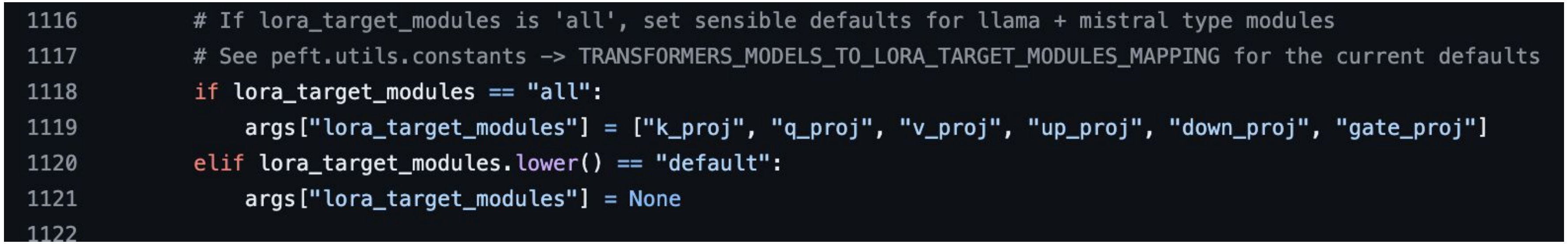


```
# Model Arguments
model:
  _component_: torchtune.models.llama2.qlora_llama2_7b
  lora_attn_modules: ['q_proj', 'v_proj', 'k_proj', 'output_proj']
  apply_lora_to_mlp: True
  apply_lora_to_output: False
  lora_rank: 8
  lora_alpha: 16

```

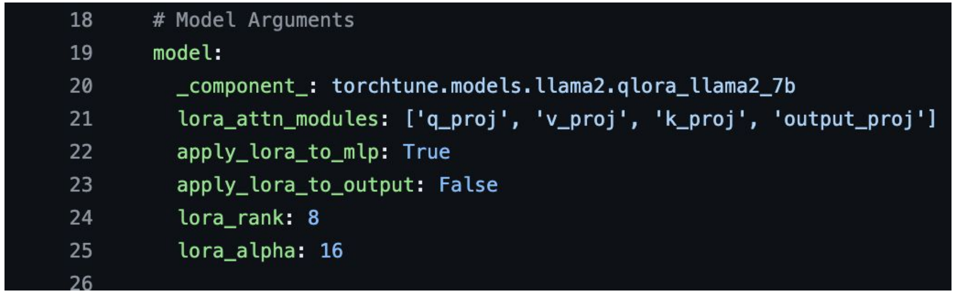
Spot the difference!

answer is on next slide :D

Wait…are those actually the same benchmark?



```
        # If lora_target_modules is 'all', set sensible defaults for llama + mistral type modules
        # See peft.utils.constants -> TRANSFORMERS_MODELS_TO_LORA_TARGET_MODULES_MAPPING for the current defaults
        if lora_target_modules == "all":
            args["lora_target_modules"] = ["k_proj", "q_proj", "v_proj", "up_proj", "down_proj", "gate_proj"]
        elif lora_target_modules.lower() == "default":
            args["lora_target_modules"] = None
```



```
# Model Arguments
model:
  _component_: torchtune.models.llama2.qlora_llama2_7b
  lora_attn_modules: ['q_proj', 'v_proj', 'k_proj', 'output_proj']
  apply_lora_to_mlp: True
  apply_lora_to_output: False
  lora_rank: 8
  lora_alpha: 16
```

Spot the difference!

torchtune LoRA-fied the output_proj when the train.py did not.
LoRA-fying = adding 2 low rank adapters to the o of every qkv.
32 TransformerDecoderLayers * 2 more params each = 64 extra params to train.

Making sure what I’m measuring is 🍎 ⇔ 🍏 and not 🍎 ⇔ 🍊

I took a pause cuz it wasn’t going to be fruitful if the items getting measured weren’t sufficiently aligned!

Steps I took:

- Stopped LoRA-fying the output_proj in my torchtune recipe
- Changed FSDP2 wrapping policy to wrap the same layers
- Replicated the same “dummy” dataset for my benchmark
- Took another pass ensuring max seq len + other hyperparams for model construction were the same

Making sure what I’m measuring is 🍎 ⇔ 🍏 and not 🍎 ⇔ 🍊

I then ran the benchmark after my changes…and FSDP2 x NF4 still looked mighty slow.

	batch size	peak memory	runtime for a step
train.py	8	12.88 GiB	14.0s
torchtune	8	10.70 GiB	16.6s

Even though it may feel like we took a mini step back, we’ve made a giant leap unblocking our official first step: understanding the problem (gaps).

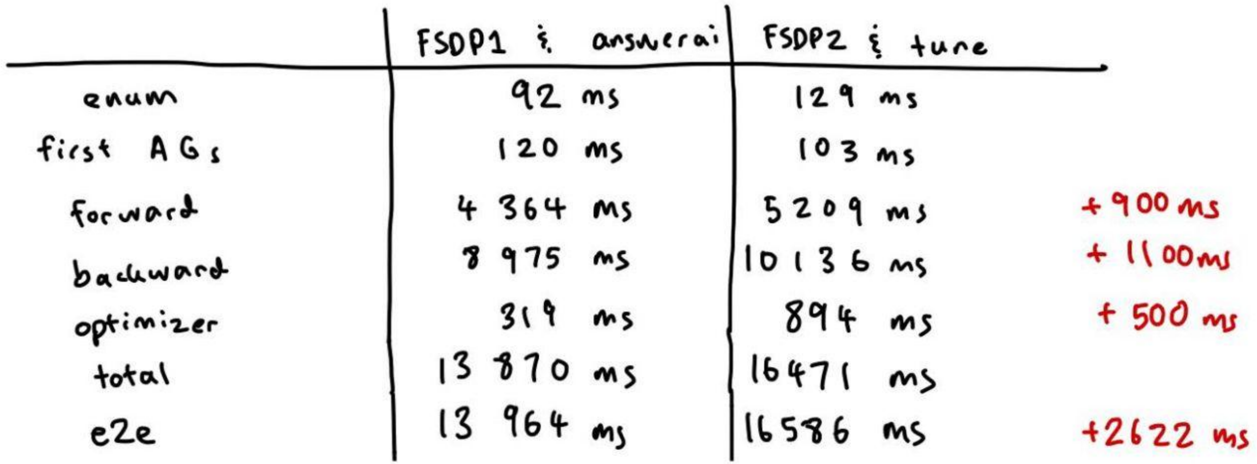
I could finally start a very long game of Spot the Difference.

train.py trace

torchtune trace

Recording the gaps

I first did a survey of the land, and derived this chart:

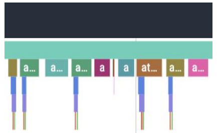


	FSDP1 & answerai	FSDP2 & tune	
enum	92 ms	129 ms	
first AGs	120 ms	103 ms	
forward	4 364 ms	5 209 ms	+ 900 ms
backward	8 975 ms	10 136 ms	+ 1100ms
optimizer	319 ms	894 ms	+ 500 ms
total	13 870 ms	16471 ms	
e2e	13 964 ms	16586 ms	+2622 ms

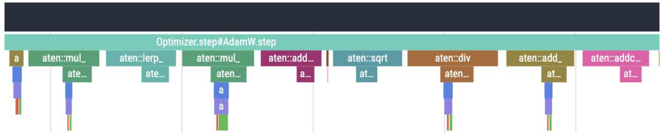
We see that we should focus on the forward and the optimizer step kernels.

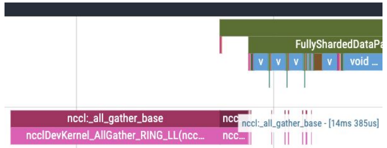
Recording the gaps

the optimizer step is still slower

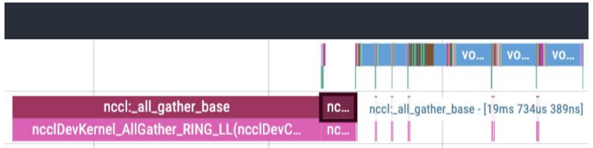


vs



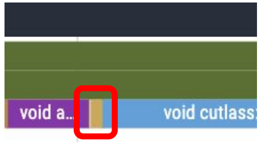

FullyShardedDataPa
v
v
v
v
void ...
nccl:_all_gather_base
ncc
ncclDevKernel_AllGather_RING_LL(ncc...
ncc...
nccl:_all_gather_base - [14ms 385us]

vs

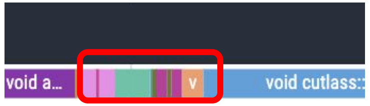

vo...
vo...
vo...
nccl:_all_gather_base
nc...
ncclDevKernel_AllGather_RING_LL(ncclDevC...
nc...
nccl:_all_gather_base - [19ms 734us 389ns]

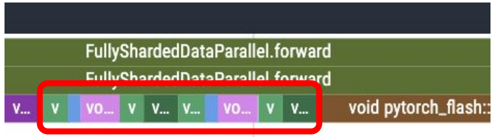
the 2nd AG was 5ms longer

additional overhead right before the gemms

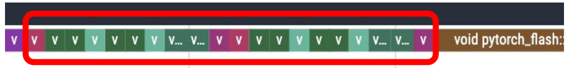


vs



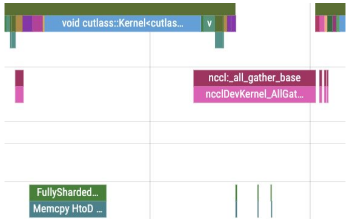

FullyShardedDataParallel.forward
FullyShardedDataParallel.forward
V...
V
VO...
V
V...
V...
VO...
V
V...
void pytorch_flash::

vs

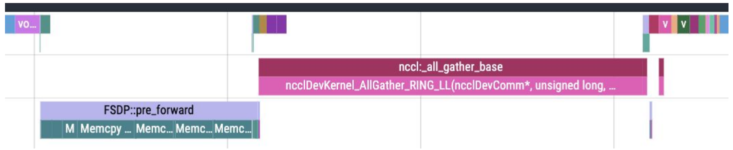


differing ops before sdpa

exposed AGs/mem H2Ds



vs



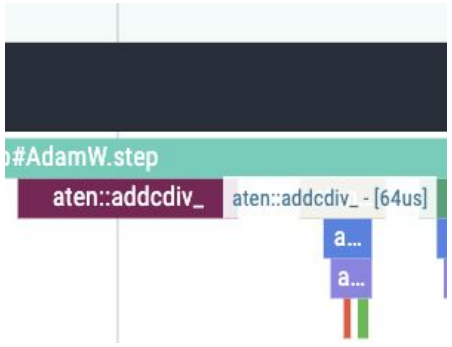
Investigating and filling the gaps if possible

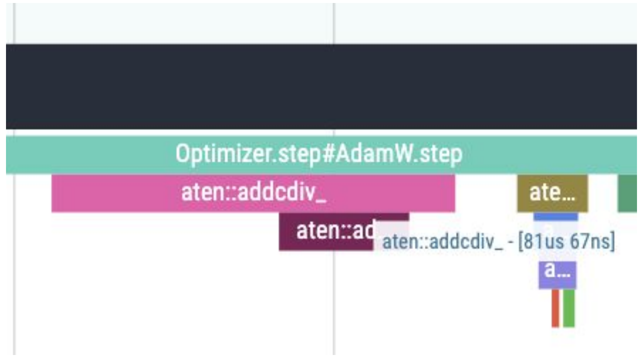
lesgo

traces:

https://drive.google.com/drive/u/3/folders/1HmGNC4v4L5nXhtdDMVCpUBrme1ELp-2C

Gap: the optimizer step is still slower



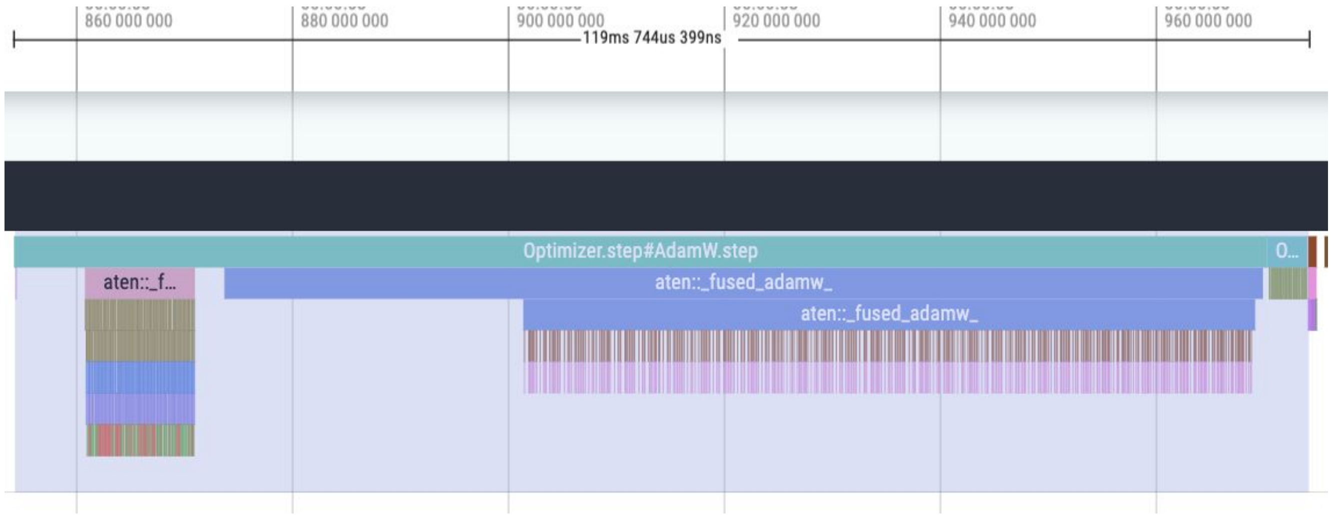


Understanding why:

- DTensor overhead
- parameter is not necessarily contiguous

Solution: used fused! (thanks Intel)

Gap filled: the optimizer step is now faster

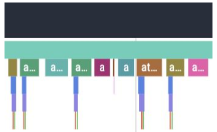


Solution: used fused! (thanks Intel)

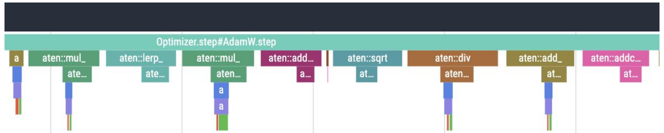
- avoid DTensor overhead by only dispatching 1 fused kernel!
- leverage vectorization
- goes from ~1s -> 120ms, speeding up 8x

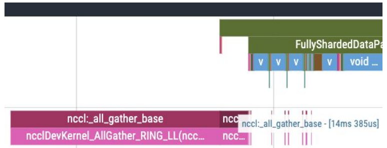
The gaps

~~the optimizer step is still slower~~

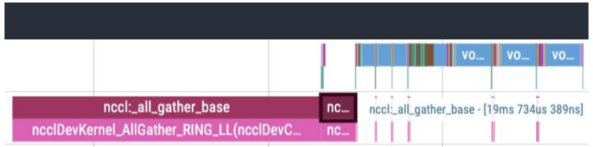


vs



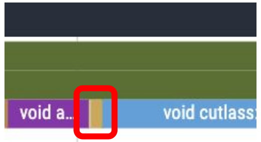

FullyShardedDataPa
v
v
v
v
void ...
nccl:_all_gather_base
ncclDevKernel_AllGather_RING_LL(ncc...
ncc
ncc...
nccl:_all_gather_base - [14ms 385us]

vs

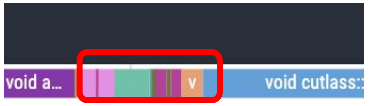

vo...
vo...
vo...
nccl:_all_gather_base
ncclDevKernel_AllGather_RING_LL(ncclDevC...
nc...
nc...
nccl:_all_gather_base - [19ms 734us 389ns]

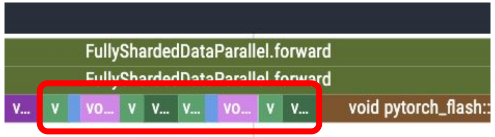
the 2nd AG was 5ms longer

additional overhead right before the gemms

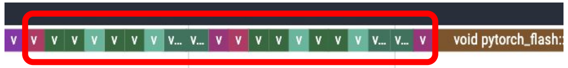


vs



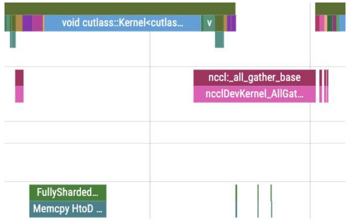

FullyShardedDataParallel.forward
FullyShardedDataParallel.forward
V...
V
VO...
V
V...
V...
VO...
V
V...
void pytorch_flash::

vs

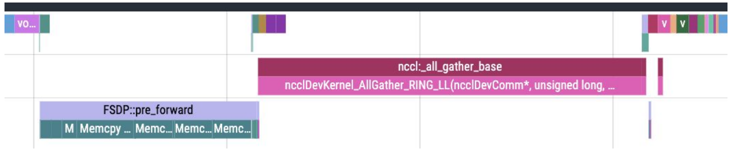


differing ops before sdpa

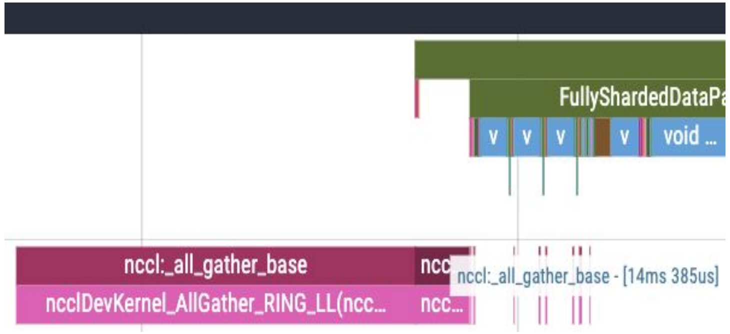
exposed AGs/mem H2Ds

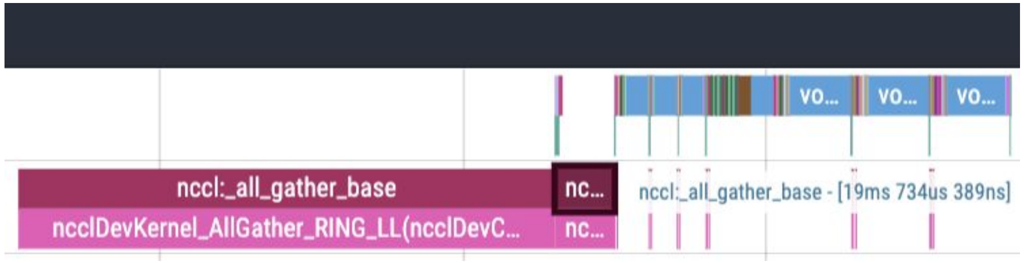


vs



Gap: the 2nd AG was 5ms longer


FullyShardedDataPa
v
v
v
v
void ...
nccl:_all_gather_base
ncc
nccl:_all_gather_base - [14ms 385us]
ncclDevKernel_AllGather_RING_LL(ncc...
ncc...


vo...
vo...
vo...
nccl:_all_gather_base
nc...
nccl:_all_gather_base - [19ms 734us 389ns]
ncclDevKernel_AllGather_RING_LL(ncclDevC...
nc...

Check the all-gather input arguments!

Realize that 25,300,992 bf16s != 64,646,208 Bytes

In FSDP1, print out `_fqns` of a FlatParameter. In FSDP2, print all_gather_inputs.

Lining up the parameters revealed…

Gap understood: the 2nd AG was much larger

answerai:

	4194304
	4194304
	4194304
4194304 (o.proj.weight)	
	11272192
4096	11272192
4096	11272192

totaling 50,601,984 ÷2
= 25,300,992

tune	scalers	2x factor	data
4096	131072	1024	4194304
	131072	1024	4194304
	131072	1024	4194304
16777216 (output_proj)			
	1352256	27521	11272192
4096	1352256	27521	11272192
	1352256	27521	11272192

$\underbrace{\text{scalers} + \text{2x factor}}_{1,461,312}$

totaling 64,646,208

Why the heavy load?

- our NF4 all-gathers the NF4 metadata whereas bnb Params4bit does not
- more significantly, after opting out of LoRA, our output_proj remained frozen but full sized. train.py froze their output_proj too, but quantized it

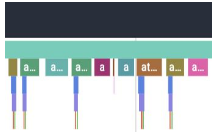
Gap to be filled: detangle the q from qLoRA

Why the heavy load?

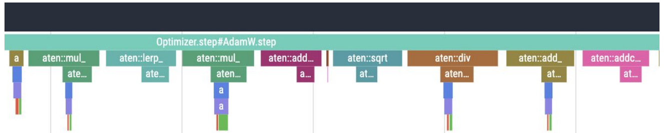
- our NF4 all-gathers the NF4 metadata whereas bnb Params4bit does not
  - This is intended! FSDP2 allows NF4Tensor subclass to decide which of its inner tensors are all-gathered
- more significantly, after opting out of LoRA, our output_proj remained frozen but full sized. train.py froze their output_proj too, but quantized it.
  - This is not intended!
  - This is a next step for torchtune to allow base weights to be quantized even if they opt out of LoRA

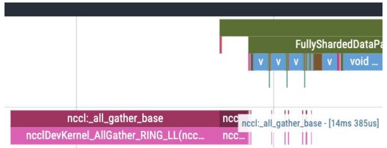
The gaps

~~the optimizer step is still slower~~

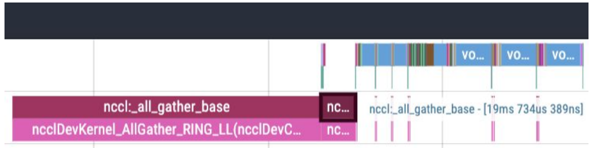


vs



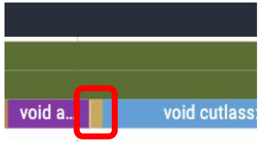

FullyShardedDataP
nccl:_all_gather_base
ncclDevKernel_AllGather_RING_LL(ncc...
nccl:_all_gather_base - [14ms 385us]

vs

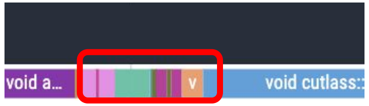

nccl:_all_gather_base
ncclDevKernel_AllGather_RING_LL(ncclDevC...
nccl:_all_gather_base - [19ms 734us 389ns]

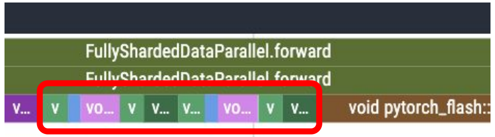
~~the 2nd AG was 5ms longer~~

additional overhead right before the gemms

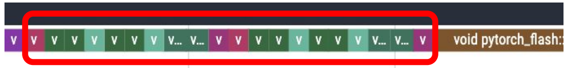


vs



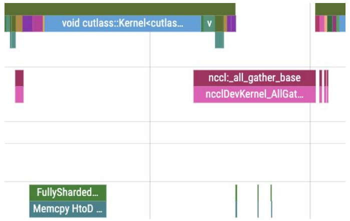

FullyShardedDataParallel.forward
FullyShardedDataParallel.forward
void pytorch_flash::

vs

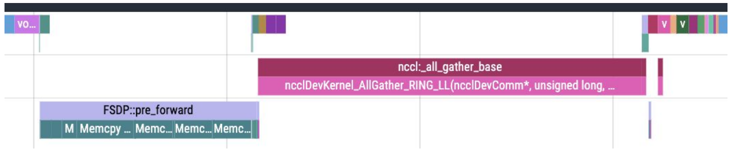


differing ops before sdpa

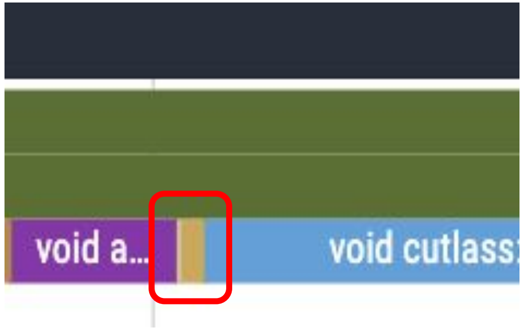
exposed AGs/mem H2Ds

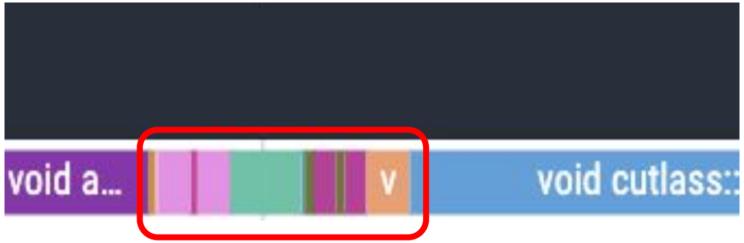


vs


nccl:_all_gather_base
ncclDevKernel_AllGather_RING_LL(ncclDevComm*, unsigned long, ...
FSDP::pre_forward
Memcpy ...

Gap: additional overhead right before the gemms

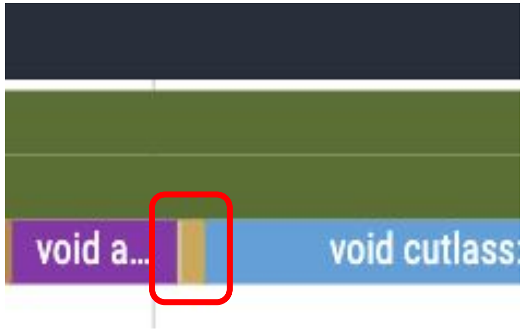


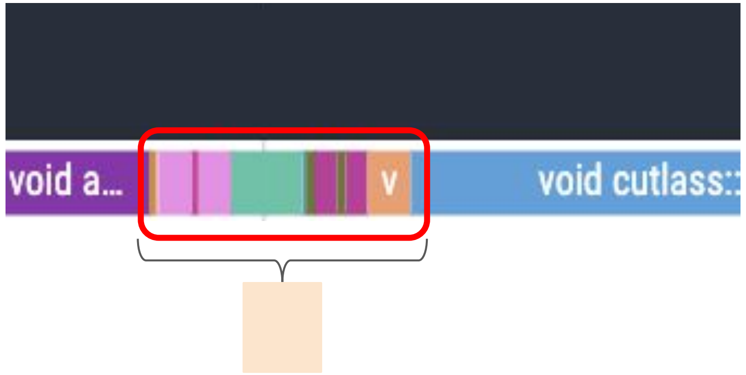


Understanding why:

- NF4Tensor overrides the mm in order to dequantize before calling the gemm
- bnb has a CUDA kernel for the dequantization work

Gap to be filled: fuse LinearNF4 overhead



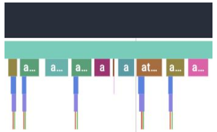


Solutions:

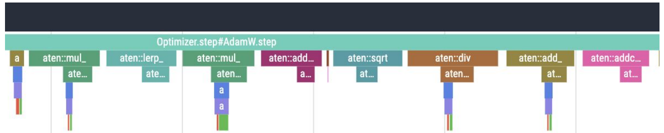
- A next step is to leverage torch.compile. I did try it, but it does not play well with activations checkpointing at the moment
- Another next step is to package and use the Triton kernels that Driss wrote

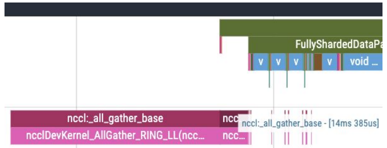
The gaps

~~the optimizer step is still slower~~

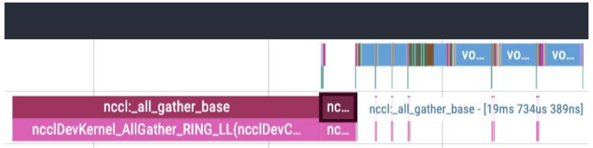


vs



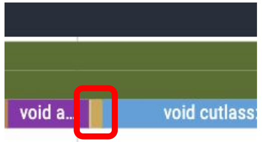

FullyShardedDataPa
void ...
nccl:_all_gather_base
ncclDevKernel_AllGather_RING_LL(ncc...
nccl:_all_gather_base - [14ms 385us]

vs

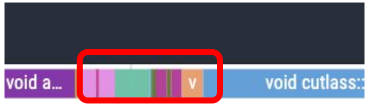

vo...
nccl:_all_gather_base
ncclDevKernel_AllGather_RING_LL(ncclDevC...
nccl:_all_gather_base - [19ms 734us 389ns]

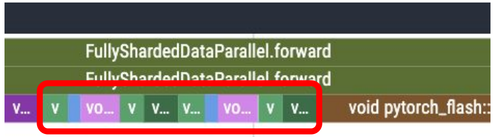
~~the 2nd AG was 5ms longer~~

~~additional overhead right before the gemms~~

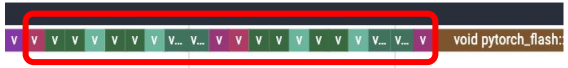


vs



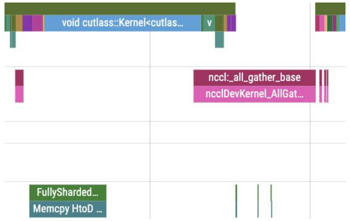

FullyShardedDataParallel.forward
FullyShardedDataParallel.forward
vo...
void pytorch_flash::

vs

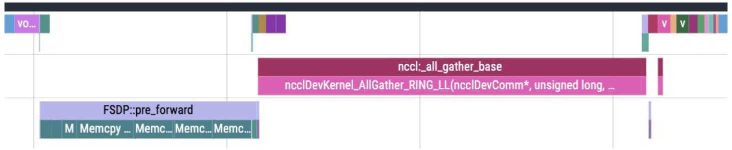


differing ops before sdpa

exposed AGs/mem H2Ds



vs


nccl:_all_gather_base
ncclDevKernel_AllGather_RING_LL(ncclDevComm*, unsigned long, ...
FSDP::pre_forward
M Memcpy ... Memc... Memc... Memc...

Gap: differing ops before sdpa (costing us 6ms per layer!)

This gap is the most boring of them all: torchtune and the default LLaMa2 config simply use different RoPE algorithms.

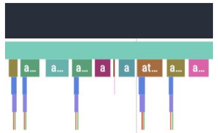
- torchtune uses the original Meta algorithm with no numerical differences.
- the default LlamaRotaryEmbedding is 2-3x faster (for our trace 6ms faster) but is not the same numerically.

Solution:

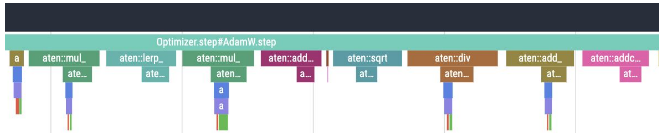
- A next step is for torchtune to offer more options for more optimized but less faithful Embedding algos if desired.

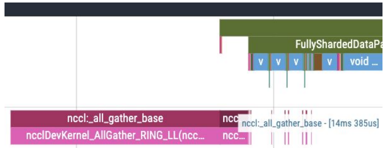
The gaps

~~the optimizer step is still slower~~

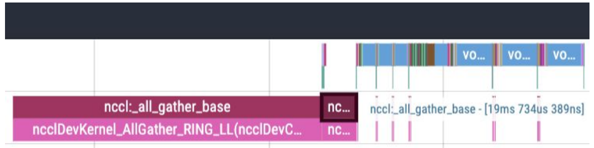


vs



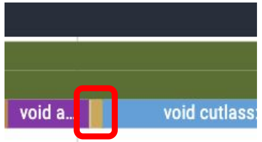

FullyShardedDataPa
v
v
v
v
void …
nccl:_all_gather_base
ncc
ncclDevKernel_AllGather_RING_LL(ncc…
ncc…
nccl:_all_gather_base - [14ms 385us]

vs

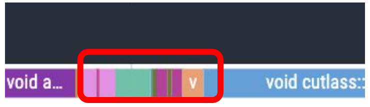

vo…
vo…
vo…
nccl:_all_gather_base
nc…
ncclDevKernel_AllGather_RING_LL(ncclDevC…
nc…
nccl:_all_gather_base - [19ms 734us 389ns]

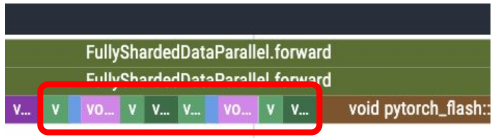
~~the 2nd AG was 5ms longer~~

~~additional overhead right before the gemms~~

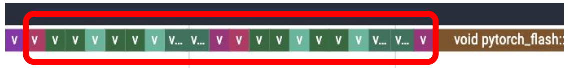


vs



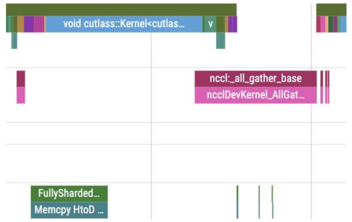

FullyShardedDataParallel.forward
FullyShardedDataParallel.forward
V…
V
VO…
V
V…
V…
VO…
V
V…
void pytorch_flash::

vs

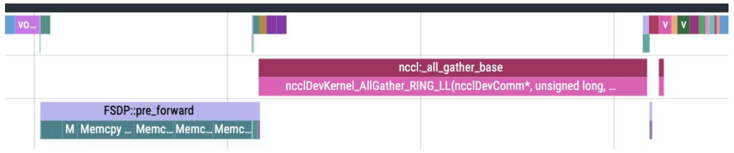


~~differing ops before sdpa~~

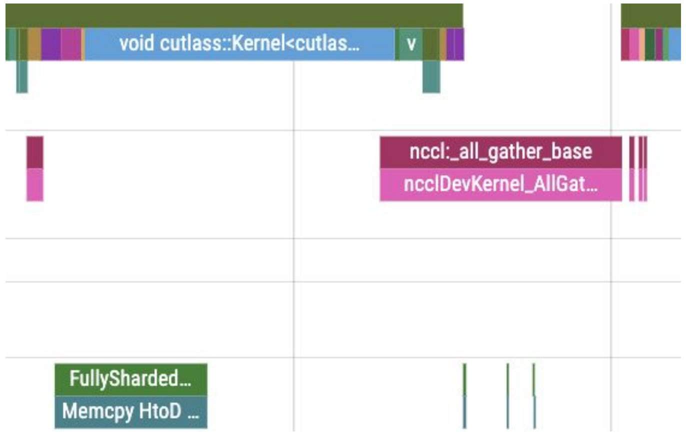
exposed AGs/mem H2Ds

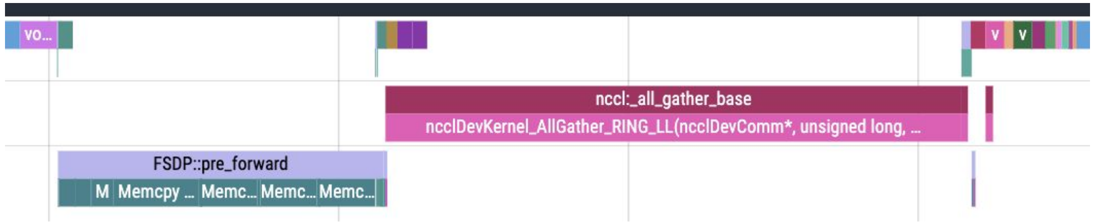


vs



Gap: exposed AGs/mem H2Ds





We wonder: why is the left side Memcpy hidden in FSDP1, but very exposed in FSDP2?

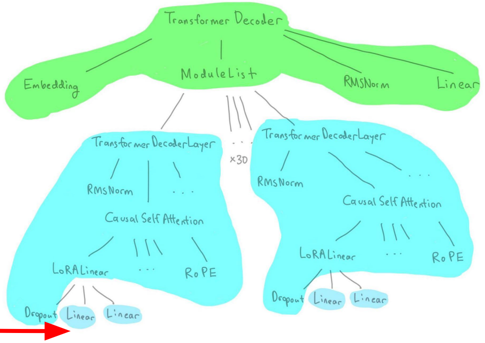
Answer: the stricter memory restraints!

- Memcpy is used to bring offloaded params from CPU to GPU
- FSDP2 is guaranteeing the constraint that only 2 layers of params will be allowed at a time by having the Memcpy wait as well.

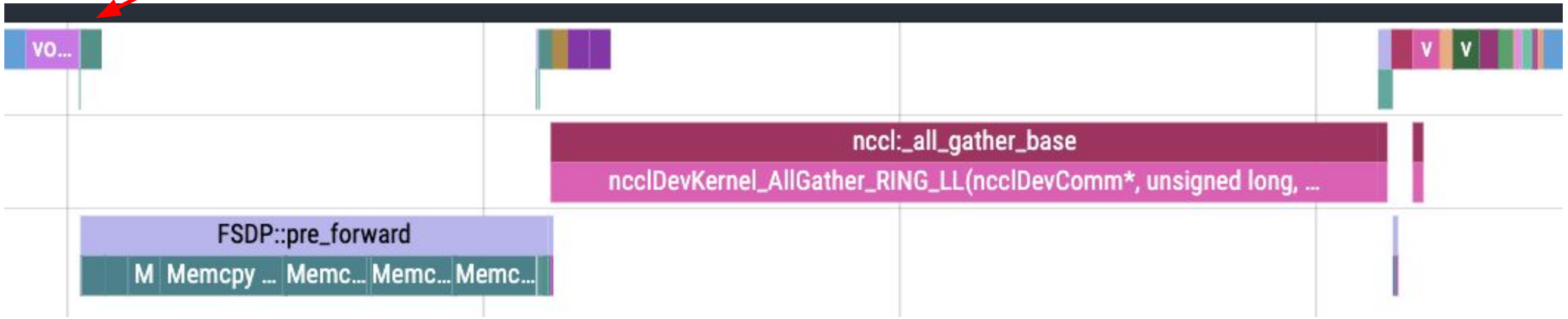
Gap: exposed AGs/mem H2Ds

Note that the problem here isn't that FSDP2 is too strict. It's that the computation is too small to properly hide the communication!

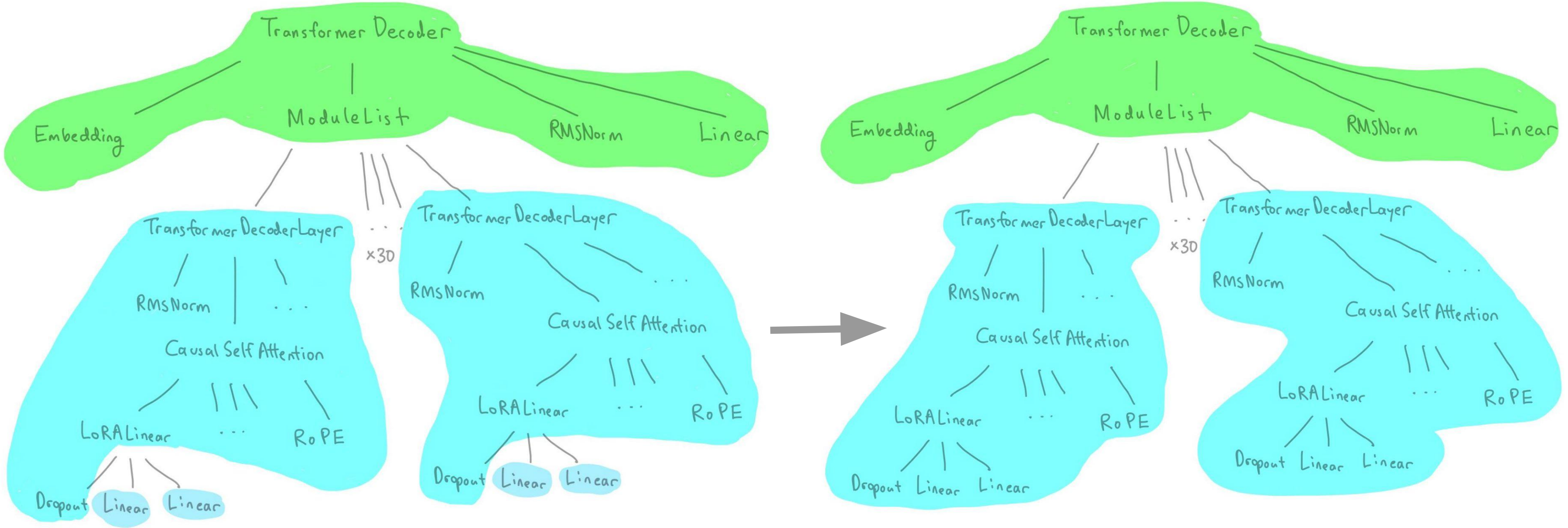
Solution: wrap more granularly. Have bigger layers.



too tiny!



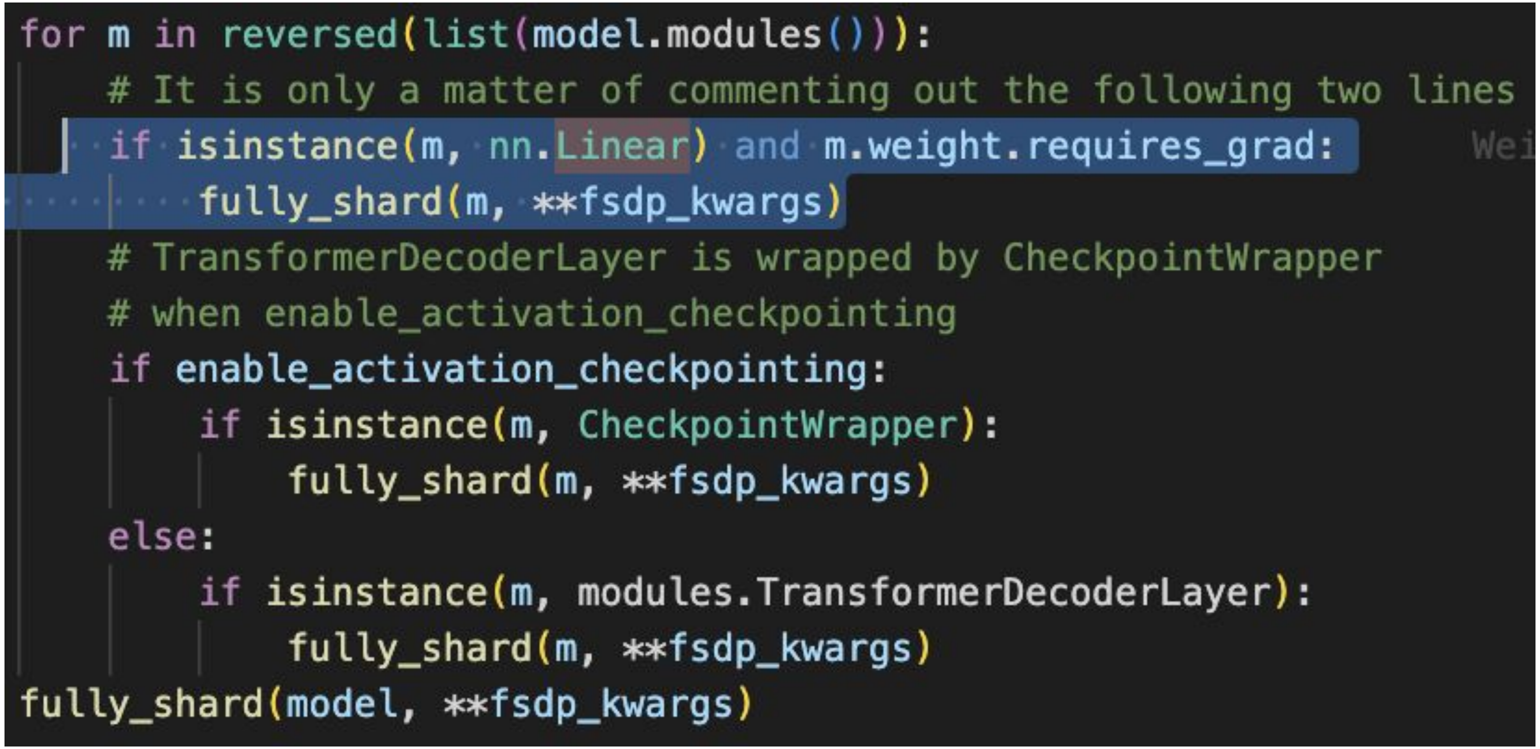
We want a new wrapping policy:



This new wrapping policy is only possible with FSDP2!

As now, both NF4Tensors and plain Tensors can coexist in 1 layer.

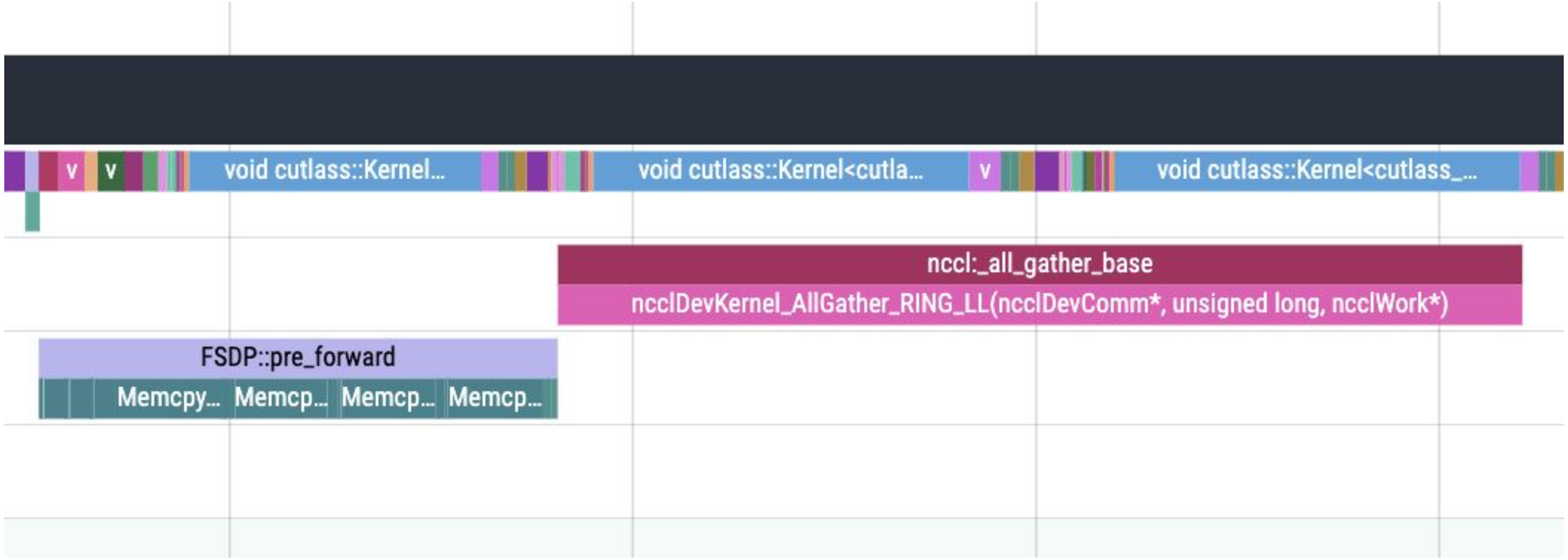
Side note: this is very easy to do in FSDP2



```
for m in reversed(list(model.modules())):
    # It is only a matter of commenting out the following two lines
    if isinstance(m, nn.Linear) and m.weight.requires_grad:          Wei
        fully_shard(m, **fsdp_kwargs)
    # TransformerDecoderLayer is wrapped by CheckpointWrapper
    # when enable_activation_checkpointing
    if enable_activation_checkpointing:
        if isinstance(m, CheckpointWrapper):
            fully_shard(m, **fsdp_kwargs)
    else:
        if isinstance(m, modules.TransformerDecoderLayer):
            fully_shard(m, **fsdp_kwargs)
fully_shard(model, **fsdp_kwargs)
```

Gap filled: hidden AGs/mem H2Ds

Solution: wrap more granularly. Have bigger layers.

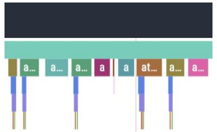

v
v
void cutlass::Kernel...
void cutlass::Kernel<cutla...
v
void cutlass::Kernel<cutlass_...
nccl:_all_gather_base
ncclDevKernel_AllGather_RING_LL(ncclDevComm*, unsigned long, ncclWork*)
FSDP::pre_forward
Memcpy...
Memcp...
Memcp...
Memcp...

Now both are overlapped!

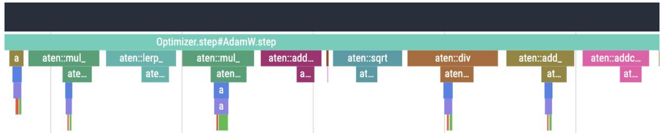
final_torchtune_trace.json

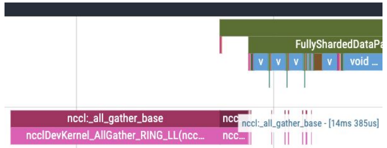
The gaps

~~the optimizer step is still slower~~

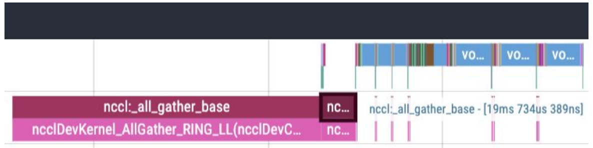


vs



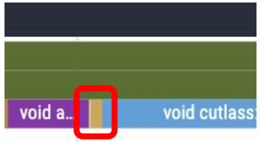

FullyShardedDataP
void …
nccl:_all_gather_base
ncclDevKernel_AllGather_RING_LL(ncc…
nccl:_all_gather_base - [14ms 385us]

vs

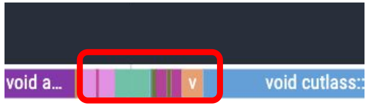

vo…
nccl:_all_gather_base
ncclDevKernel_AllGather_RING_LL(ncclDevC…
nc…
nccl:_all_gather_base - [19ms 734us 389ns]

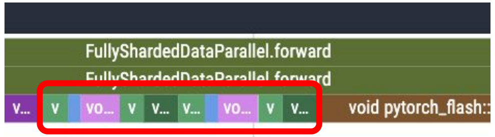
~~the 2nd AG was 5ms longer~~

~~additional overhead right before the gemms~~

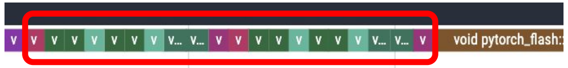


vs



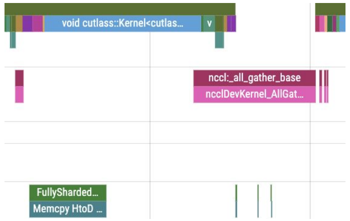

FullyShardedDataParallel.forward
FullyShardedDataParallel.forward
v…
vo…
void pytorch_flash::

vs

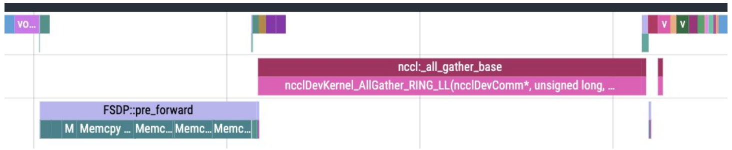


~~differing ops before sdpa~~

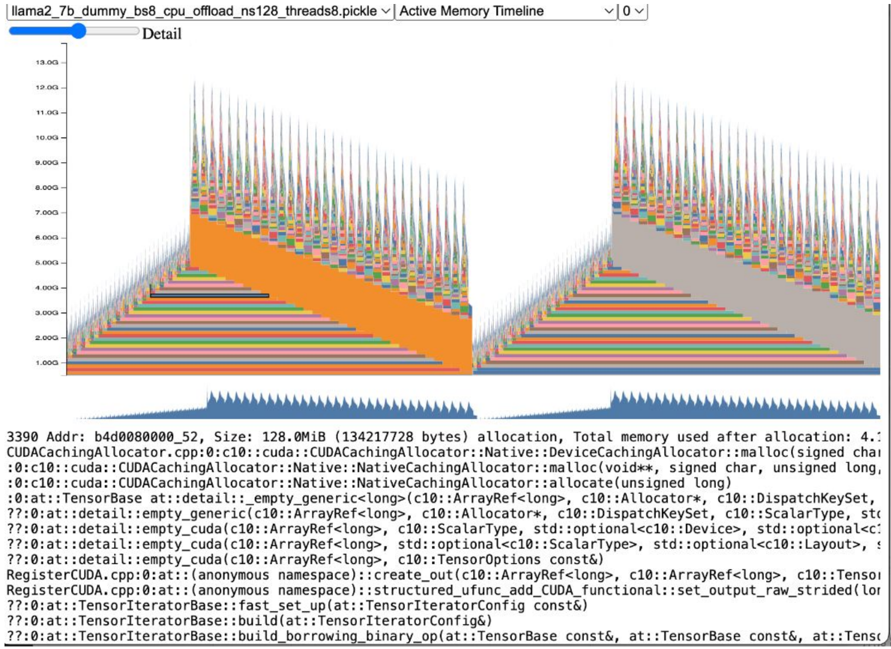
~~exposed AGs/mem H2Ds~~



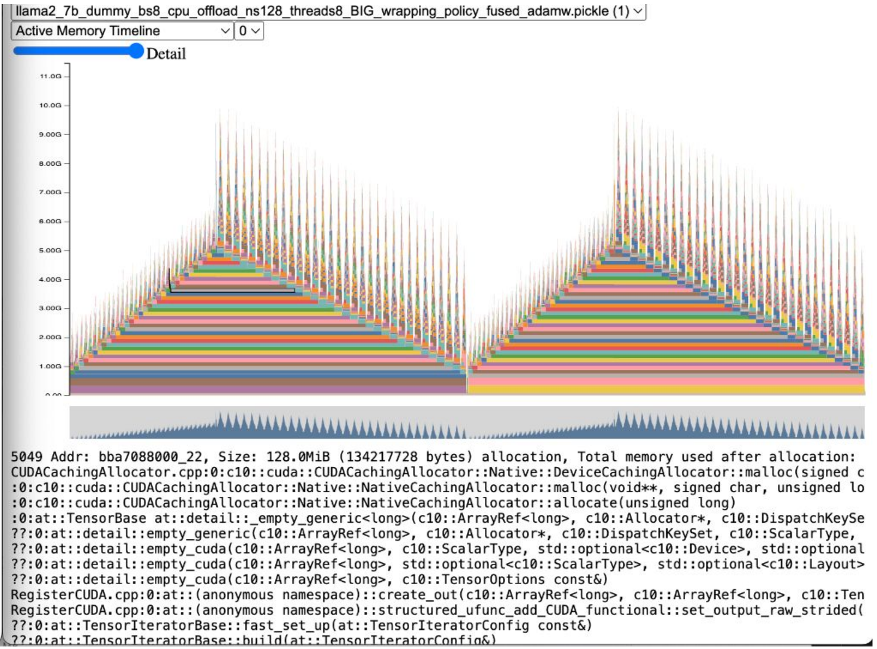
vs


vo…
nccl:_all_gather_base
ncclDevKernel_AllGather_RING_LL(ncclDevComm*, unsigned long, …
FSDP::pre_forward
M Memcpy … Memc… Memc… Memc…

Positive gap: how come torchtune uses less memory?



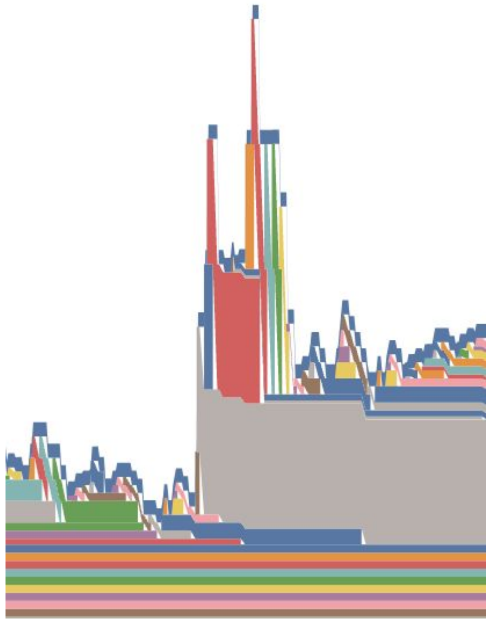
```
3390 Addr: b4d0080000_52, Size: 128.0MiB (134217728 bytes) allocation, Total memory used after allocation: 4.1
CUDACachingAllocator.cpp:0:c10::cuda::CUDACachingAllocator::Native::DeviceCachingAllocator::malloc(signed char
:0:c10::cuda::CUDACachingAllocator::Native::NativeCachingAllocator::malloc(void**, signed char, unsigned long,
:0:c10::cuda::CUDACachingAllocator::Native::NativeCachingAllocator::allocate(unsigned long)
:0:at::TensorBase at::detail::_empty_generic<long>(c10::ArrayRef<long>, c10::Allocator*, c10::DispatchKeySet,
??:0:at::detail::empty_generic(c10::ArrayRef<long>, c10::Allocator*, c10::DispatchKeySet, c10::ScalarType, std
??:0:at::detail::empty_cuda(c10::ArrayRef<long>, c10::ScalarType, std::optional<c10::Device>, std::optional<c1
??:0:at::detail::empty_cuda(c10::ArrayRef<long>, std::optional<c10::ScalarType>, std::optional<c10::Layout>, s
??:0:at::detail::empty_cuda(c10::ArrayRef<long>, c10::TensorOptions const&)
RegisterCUDA.cpp:0:at::(anonymous namespace)::create_out(c10::ArrayRef<long>, c10::ArrayRef<long>, c10::Tenso
RegisterCUDA.cpp:0:at::(anonymous namespace)::structured_ufunc_add_CUDA_functional::set_output_raw_strided(lon
??:0:at::TensorIteratorBase::fast_set_up(at::TensorIteratorConfig const&)
??:0:at::TensorIteratorBase::build(at::TensorIteratorConfig&)
??:0:at::TensorIteratorBase::build_borrowing_binary_op(at::TensorBase const&, at::TensorBase const&, at::Tens
```

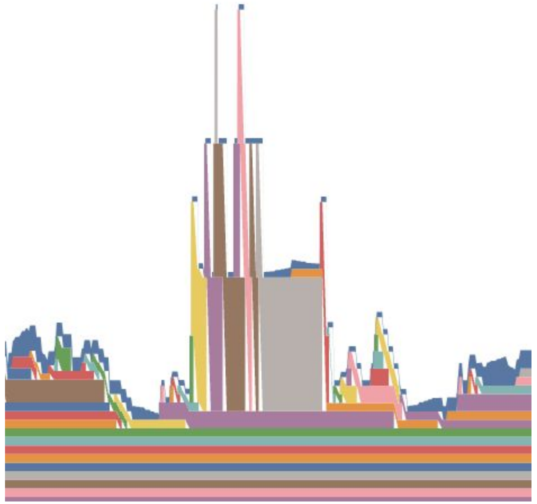

llama2_7b_dummy_bs8_cpu_offload_ns128_threads8_BIG_wrapping_policy_fused_adamw.pickle (1)
Active Memory Timeline
0
Detail
11.0G
10.0G
9.00G
8.00G
7.00G
6.00G
5.00G
4.00G
3.00G
2.00G
1.00G

```
5049 Addr: bba7088000_22, Size: 128.0MiB (134217728 bytes) allocation, Total memory used after allocation:
CUDACachingAllocator.cpp:0:c10::cuda::CUDACachingAllocator::Native::DeviceCachingAllocator::malloc(signed c
:0:c10::cuda::CUDACachingAllocator::Native::NativeCachingAllocator::malloc(void**, signed char, unsigned lo
:0:c10::cuda::CUDACachingAllocator::Native::NativeCachingAllocator::allocate(unsigned long)
:0:at::TensorBase at::detail::_empty_generic<long>(c10::ArrayRef<long>, c10::Allocator*, c10::DispatchKeySe
??:0:at::detail::empty_generic(c10::ArrayRef<long>, c10::Allocator*, c10::DispatchKeySet, c10::ScalarType,
??:0:at::detail::empty_cuda(c10::ArrayRef<long>, c10::ScalarType, std::optional<c10::Device>, std::optional
??:0:at::detail::empty_cuda(c10::ArrayRef<long>, std::optional<c10::ScalarType>, std::optional<c10::Layout>
??:0:at::detail::empty_cuda(c10::ArrayRef<long>, c10::TensorOptions const&)
RegisterCUDA.cpp:0:at::(anonymous namespace)::create_out(c10::ArrayRef<long>, c10::ArrayRef<long>, c10::Ten
RegisterCUDA.cpp:0:at::(anonymous namespace)::structured_ufunc_add_CUDA_functional::set_output_raw_strided(
??:0:at::TensorIteratorBase::fast_set_up(at::TensorIteratorConfig const&)
??:0:at::TensorIteratorBase::build(at::TensorIteratorConfig&)
```

One, yes, FSDP2 has better guarantees. But here, it's that torchtune frees the loss early!

Positive gap: how come torchtune uses less memory?





Zooming in, the gap is the size of the loss.

torchtune can get up to bs=16 for llama2-7b, 2048 context len

llama2-7B, context length 2048, with CPU offloading

batch size	peak memory	runtime for a step
8	12.88 GiB	14.0s
10	15.89 GiB	17.5s
below needs PYTORCH_CUDA_ALLOC_CONF=expandable_segments:True		
12	18.91 GiB	20.9s
14	21.92 GiB	23.6s
+	OOM	N/A

train.py

batch size	peak memory	runtime for a step
8	10.7 GiB	14.8s
10	13.2 GiB	18.2s
12	15.7 GiB	21.7s
14	18.3 GiB	25.3s
below needs PYTORCH_CUDA_ALLOC_CONF=expandable_segments:True		
16	20.8 GiB	28.8s
+	OOM	N/A

torchtune

Try this out in torchtune!

https://github.com/pytorch/torchtune

though HUGE DISCLAIMER checkpointing is not working yet

The rest of the team

@drisspg: Driss wrote the original NF4 tensor implementation

@awgu: Andrew is the main architect of FSDP2

@weifengpy: Wei showed how to compose new dtypes w/ FSDP2

@rohan-varma/@ebsmothers: wrote the LoRA recipes and merged code in tune

Thanks!

Implement new dtypes that work with compile and FSDP: https://github.com/pytorch/ao

Compile them: https://pytorch.org/docs/main/torch.compiler

Author them as subclasses so they work like real PyTorch tensors: https://github.com/albanD/subclass_zoo/

Go from 1 GPU to N GPUs with FSDP2: https://github.com/pytorch/pytorch/issues/114299

End to end finetuning examples: https://github.com/pytorch/torchtune

End to end training examples: https://github.com/pytorch/torchtitan

And remember to profile your memory: https://pytorch.org/blog/understanding-gpu-memory-1/

If you have any questions reach out to us on Discord.

If you’re doing research at the intersection of quantization and distributed we’d loooove to hear from you