Benchmarking Notes

1 benchmarking_script

(a) Written in `benchmark.py`.

(b) **Commands:**

```
# small
python benchmark.py --d-model 768 --d-ff 3072 --num-layers 12 --num-heads 12

# medium
python benchmark.py --d-model 1024 --d-ff 4096 --num-layers 24 --num-heads 16

# large
python benchmark.py --d-model 1280 --d-ff 5120 --num-layers 36 --num-heads 20

# xl
python benchmark.py --d-model 1600 --d-ff 6400 --num-layers 48 --num-heads 25

# 2.7B
python benchmark.py --d-model 2560 --d-ff 10240 --num-layers 32 --num-heads 32
```

Results:

Table 1: Benchmark results for small and medium models.

Model	Pass	Warmup	Mean (s)	Std (s)
Small	Forward	0	0.072297	0.121894
Small	Backward	0	0.106601	0.121139
Small	Forward	1	0.031458	0.000598
Small	Backward	1	0.067038	0.001763
Medium	Forward	0	0.136340	0.117727
Medium	Backward	0	0.241945	0.122108
Medium	Forward	1	0.097735	0.001412
Medium	Backward	1	0.202035	0.001391

```
Cannot do rest due to memory limitations (8GB).
```

(c) Minor increase in measured time with 0 warmup steps. This happens because some optimizations are done based on the first pass, so warming up lets the correct cache/shapes be known in advance for the next passes.

2 nsys_profile

Table 2: Benchmark results on forward pass on Nsys vs Python standard library.

Model	Context Length	Nsys Mean (ms)	Python Mean (ms)
Small	128	40.780	40.479
Small	256	41.320	43.825
Small	512	40.996	70.388

(a) Total time is roughly 40 ms for all context sizes (did small model only due to memory constraints) but our measured time in Python keeps increasing due to device sync overhead.

(b) `ampere_sgemm_128x64_nn` takes up the most time in both forward and backward passes. It is called 52 times in the forward pass.

(c) **Forward:**

```
Time    Total Time    Instances    Avg         Med         Min         Max         StdDev
    Name
5.6%    2.167 ms      94           23.053 \textmus   22.688 \textmus   21.920 \textmus
    30.304 \textmus   1.260 \textmus  void at::native::elementwise_kernel<(int)128, (int)2,
    void at::native::gpu_kernel_impl_nocast<...>>
```

Backward:

```
Time     Total Time    Instances    Avg         Med         Min         Max         StdDev
    Name
13.7%    7.872 ms      11           715.670 \textmus  597.540 \textmus  592.869 \textmus
    1.914 ms    397.501 \textmus void cutlass::Kernel2<
    cutlass_80_simt_sgemm_128x64_8x5_nt_align1>(T1::Params)
10.5%    6.012 ms      68           88.410 \textmus   36.176 \textmus   1.152 \textmus
    274.627 \textmus  85.005 \textmus  void at::native::vectorized_elementwise_kernel<(int)4,
     ...>
```

(d) Optimizer takes up a huge chunk of time but overall, kernel contribution remains the same.

(e) Matrix multiplication takes approximately 762 μs while computing softmax takes approximately 800 μs. The matrix multiplication has much more FLOPs than softmax.

3 mixed_precision_accumulation

```
ans.
tensor(10.0001)
tensor(9.9531, dtype=torch.float16)
tensor(10.0021)
tensor(10.0021)
```

Accumulating in FP32 lets us retain a more accurate result when adding floats of lower precision, regardless of whether we upscale the lower precision float or not.

4 benchmarking_mixed_precision

(a)
- Model parameters: FP32
- Output of first feedforward layer: FP16
- Output of layer norm: FP16
- Predicted logits: FP16
- Loss: FP32
- Gradients: FP16

(b) The mean and variance calculations in layernorm are sensitive to mixed precision. The subtraction in mean, squaring in variance and sqrt in normalization are all sensitive. Using BF16 is okay as it has the same range as FP32 so we could treat layernorm with mixed precision then.

(c) With bigger batch sizes, model sizes and context length, mixed precision is significantly faster.

5 benchmarking_mixed_precision

Using only small + 128 context length due to memory limitations.

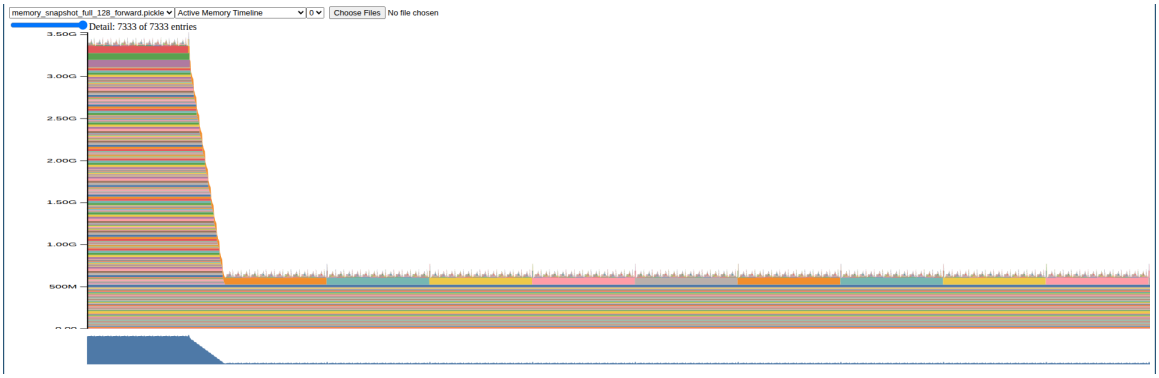

Figure 1: Forward only.

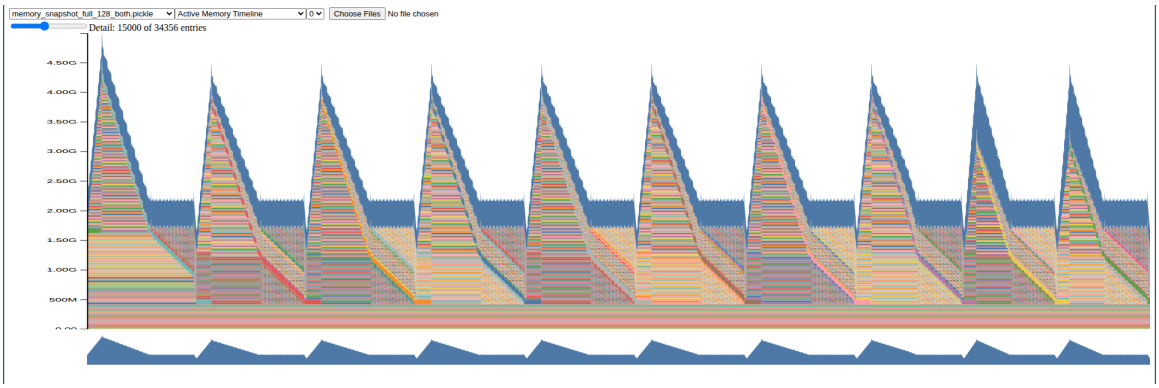

Figure 2: Full step.

(a) The peaks for forward pass have very sharp ends which makes them identifiable. Memory rises and falls for each pass. We also see a big chunk being cleared when we are clearing gradients in our full step.

(b) Peak memory usage of Forward Pass: 3.4GB Peak memory usage of Full Step: 4.7GB

(c) Peak memory usage (mixed precision) of Forward Pass: 3GB Peak memory usage (mixed precision) of Full Step: 4.5GB

There is a small improvement in memory usage but not a major one.

(d) Considering for small sized model: (using 16 batch size) 16 x 128 x 768 x 2 bytes = 0.75 MiB

(e) Yes by calculating the memory size of a matrix at a particular layer, we actually can find it in the graph and also figure out where we are in the taining layers. Just looking at the biggest allocations also helps figure this out e.g. in my case its the logits shape (16 x 128 x 10000) which indicates the end of training when its allocation ends. We can also look at some of the smaller repetitive patterns to see our transformer layers in action.

6 pytorch_attention

d_model	Sequence Length	Forward Mean (ms)	Backward Mean (ms)	Mem. Usage (MB)
16	256	0.3204	2.3416	41.8000
16	1024	0.4285	1.9817	239.3200
16	4096	0.5548	19.3975	3139.7600
32	256	0.3811	1.2181	339.9800
32	1024	0.4645	2.1281	245.2800
32	4096	0.5351	19.5532	3145.7600
64	256	0.3615	1.1155	369.9800
64	1024	0.5135	2.0453	239.3600
64	4096	0.5414	22.9726	3175.7600
128	256	0.3671	1.1074	459.9200
128	1024	0.4364	2.2736	273.0400
128	4096	0.5530	43.8644	3701.5200

(a) OOM on sequence lengths 8192 and 16384 for all d_models. For any QK where sequence length is 8192: batch size x sequence length x sequence length = 8 x 8192 x 8192 x 4 bytes = 2048 MB

We need 2x that for the backward pass.

Memory reserved before backward pass is being dominated by sequence length. We can see that regardless of d_model, a sequence length of 4096 results in memory usage upwards of 3GB. Reducing the quadratic memory nature of QK in attention would help in reducing memory significantly.