CS336 Assignment 2

Dmitry Frumkin

December 2, 2025

Contents

1.1 Profiling and Benchmarking

Code: `cs336_systems/benchmark.py`

Problem (benchmarking_script)

Code:

- `cs336_systems/benchmark.sh`
- `cs336_systems/benchmark.ipynb`

(a) **Write a script to perform basic end-to-end benchmarking of the forward and backward passes in your model.**

I used an H-100 via lambda.ai and performed sweeps over forward or forward-backward, model size, context window size, number of warmup iterations. Every configuration was a separate process. For cleaner results, I reset the GPU in the beginning of every run.

(b) **Time the forward and backward passes for the model sizes described in §1.1.2. Use 5 warmup steps and compute the average and standard deviation of timings over 10 measurement steps. How long does a forward pass take? How about a backward pass? Do you see high variability across measurements, or is the standard deviation small?**

The forward pass takes longer in the training mode (also includes the loss), but usually not by much. The backward pass takes longer than the forward pass. Computation expectedly takes longer for larger models and larger contexts (scaling linearly with the context length). The standard deviation is generally small.

(c) **One caveat of benchmarking is not performing the warm-up steps. Repeat your analysis without the warm-up steps. How does this affect your results? Why do you think this happens? Also try to run the script with 1 or 2 warm-up steps. Why might the result still be different?**

Without warm-up, the running time as well as the standard deviation are much higher. In general, increasing the number of warmup steps led to a smaller standard deviation, but not always. Having more warmup steps helps because there is some intentional lazy initialization and auto-tuning (different behavior for frequent vs. one-off tasks).

small 128

warmup	forward infer ($\mu \pm \sigma$)	forward train ($\mu \pm \sigma$)	backward ($\mu \pm \sigma$)
0	228.3 ± 641.920 ms	334.9 ± 888.234 ms	134.3 ± 224.827 ms
1	24.6 ± 0.802 ms	47.3 ± 7.034 ms	46.8 ± 3.046 ms
2	24.5 ± 0.247 ms	44.2 ± 6.182 ms	58.2 ± 7.435 ms
5	24.7 ± 0.881 ms	43.0 ± 4.437 ms	43.9 ± 0.374 ms

small 256

warmup	forward infer ($\mu \pm \sigma$)	forward train ($\mu \pm \sigma$)	backward ($\mu \pm \sigma$)
0	90.9 ± 209.537 ms	135.8 ± 298.914 ms	78.1 ± 105.088 ms
1	24.7 ± 0.251 ms	49.0 ± 12.162 ms	43.3 ± 1.736 ms
2	24.8 ± 0.221 ms	46.8 ± 5.747 ms	62.6 ± 2.820 ms
5	24.8 ± 0.928 ms	42.6 ± 1.648 ms	44.8 ± 1.204 ms

small 512

warmup	forward infer ($\mu \pm \sigma$)	forward train ($\mu \pm \sigma$)	backward ($\mu \pm \sigma$)
0	94.5 ± 212.860 ms	135.2 ± 277.018 ms	98.2 ± 130.555 ms
1	27.2 ± 0.080 ms	38.3 ± 3.909 ms	56.7 ± 0.376 ms
2	27.2 ± 0.160 ms	37.3 ± 2.747 ms	56.6 ± 0.273 ms
5	27.5 ± 0.599 ms	36.0 ± 1.777 ms	56.7 ± 0.127 ms

small 1024

warmup	forward infer ($\mu \pm \sigma$)	forward train ($\mu \pm \sigma$)	backward ($\mu \pm \sigma$)
0	127.4 ± 208.374 ms	160.6 ± 305.183 ms	161.3 ± 105.880 ms
1	62.1 ± 0.137 ms	63.9 ± 0.979 ms	127.8 ± 2.262 ms
2	62.0 ± 0.072 ms	64.2 ± 1.039 ms	128.3 ± 2.348 ms
5	61.9 ± 0.478 ms	64.6 ± 1.126 ms	128.8 ± 1.908 ms

medium 128

warmup	forward infer ($\mu \pm \sigma$)	forward train ($\mu \pm \sigma$)	backward ($\mu \pm \sigma$)
0	141.4 ± 292.052 ms	145.2 ± 261.547 ms	89.8 ± 80.164 ms
1	48.8 ± 0.438 ms	72.9 ± 17.787 ms	66.1 ± 2.163 ms
2	48.6 ± 0.203 ms	68.0 ± 10.055 ms	67.0 ± 4.887 ms
5	27.7 ± 0.155 ms	64.8 ± 3.898 ms	69.8 ± 7.430 ms

medium 256

warmup	forward infer ($\mu \pm \sigma$)	forward train ($\mu \pm \sigma$)	backward ($\mu \pm \sigma$)
0	116.5 ± 213.767 ms	150.3 ± 274.864 ms	107.1 ± 72.944 ms
1	40.6 ± 0.200 ms	48.2 ± 10.115 ms	85.2 ± 1.726 ms
2	49.3 ± 0.502 ms	66.8 ± 4.666 ms	97.8 ± 4.334 ms
5	48.8 ± 0.200 ms	63.4 ± 1.758 ms	84.4 ± 1.162 ms

medium 512

warmup	forward infer ($\mu \pm \sigma$)	forward train ($\mu \pm \sigma$)	backward ($\mu \pm \sigma$)
0	124.9 ± 133.877 ms	160.8 ± 233.043 ms	194.4 ± 71.426 ms
1	84.6 ± 0.700 ms	87.1 ± 1.118 ms	172.7 ± 4.832 ms
2	85.0 ± 1.361 ms	87.4 ± 1.513 ms	172.4 ± 3.338 ms
5	85.5 ± 1.585 ms	88.2 ± 1.942 ms	172.7 ± 3.040 ms

medium 1024

warmup	forward infer ($\mu \pm \sigma$)	forward train ($\mu \pm \sigma$)	backward ($\mu \pm \sigma$)
0	228.9 ± 134.339 ms	276.6 ± 268.558 ms	402.7 ± 58.270 ms
1	187.2 ± 4.805 ms	192.4 ± 3.552 ms	385.6 ± 5.887 ms
2	188.3 ± 4.807 ms	192.9 ± 2.959 ms	387.1 ± 5.172 ms
5	189.1 ± 4.027 ms	194.1 ± 1.641 ms	386.8 ± 3.267 ms

large 128

warmup	forward infer ($\mu \pm \sigma$)	forward train ($\mu \pm \sigma$)	backward ($\mu \pm \sigma$)
0	92.8 ± 155.813 ms	124.0 ± 147.006 ms	143.3 ± 138.398 ms
1	73.0 ± 0.219 ms	115.5 ± 2.531 ms	119.1 ± 14.977 ms
2	43.8 ± 0.807 ms	122.3 ± 1.452 ms	118.9 ± 4.525 ms
5	43.6 ± 0.065 ms	113.5 ± 4.165 ms	118.5 ± 7.382 ms

large 256

warmup	forward infer ($\mu \pm \sigma$)	forward train ($\mu \pm \sigma$)	backward ($\mu \pm \sigma$)
0	128.5 ± 132.664 ms	272.9 ± 487.343 ms	206.4 ± 91.674 ms
1	87.1 ± 1.960 ms	115.9 ± 4.300 ms	193.3 ± 2.345 ms
2	88.5 ± 2.952 ms	122.9 ± 14.651 ms	179.4 ± 4.372 ms
5	89.1 ± 3.012 ms	116.2 ± 1.939 ms	178.7 ± 0.988 ms

large 512

warmup	forward infer ($\mu \pm \sigma$)	forward train ($\mu \pm \sigma$)	backward ($\mu \pm \sigma$)
0	238.3 ± 224.752 ms	330.1 ± 501.610 ms	378.3 ± 77.678 ms
1	166.4 ± 5.387 ms	173.5 ± 5.626 ms	353.2 ± 2.239 ms
2	167.7 ± 4.840 ms	171.8 ± 3.322 ms	353.4 ± 1.146 ms
5	169.7 ± 3.755 ms	173.1 ± 2.343 ms	353.5 ± 1.343 ms

large 1024

warmup	forward infer ($\mu \pm \sigma$)	forward train ($\mu \pm \sigma$)	backward ($\mu \pm \sigma$)
0	434.3 ± 134.639 ms	438.6 ± 131.433 ms	836.7 ± 67.380 ms
1	392.3 ± 6.728 ms	397.9 ± 5.313 ms	815.3 ± 2.741 ms
2	394.6 ± 6.250 ms	399.7 ± 3.111 ms	816.6 ± 1.816 ms
5	394.0 ± 5.575 ms	401.8 ± 1.838 ms	819.2 ± 2.926 ms

xl 128

warmup	forward infer ($\mu \pm \sigma$)	forward train ($\mu \pm \sigma$)	backward ($\mu \pm \sigma$)
0	125.6 ± 149.707 ms	147.7 ± 175.869 ms	208.2 ± 80.354 ms
1	78.5 ± 0.183 ms	166.9 ± 22.613 ms	185.1 ± 0.940 ms
2	78.6 ± 0.838 ms	153.1 ± 1.525 ms	183.8 ± 0.696 ms
5	80.1 ± 1.363 ms	151.3 ± 1.207 ms	184.2 ± 0.965 ms

xl 256

warmup	forward infer ($\mu \pm \sigma$)	forward train ($\mu \pm \sigma$)	backward ($\mu \pm \sigma$)
0	199.4 ± 142.765 ms	251.4 ± 271.645 ms	373.9 ± 122.028 ms
1	154.2 ± 4.648 ms	162.0 ± 11.039 ms	332.1 ± 2.788 ms
2	154.7 ± 4.406 ms	161.0 ± 3.197 ms	334.9 ± 1.552 ms
5	156.9 ± 3.746 ms	160.9 ± 3.502 ms	334.9 ± 1.207 ms

xl 512

warmup	forward infer ($\mu \pm \sigma$)	forward train ($\mu \pm \sigma$)	backward ($\mu \pm \sigma$)
0	372.1 ± 138.889 ms	398.7 ± 192.962 ms	701.9 ± 55.230 ms
1	328.6 ± 6.085 ms	337.8 ± 4.857 ms	686.5 ± 5.499 ms
2	329.3 ± 4.709 ms	341.4 ± 2.191 ms	689.3 ± 3.921 ms
5	330.0 ± 4.597 ms	338.8 ± 1.022 ms	692.3 ± 1.447 ms

xl 1024

warmup	forward infer ($\mu \pm \sigma$)	forward train ($\mu \pm \sigma$)	backward ($\mu \pm \sigma$)
0	785.4 ± 126.616 ms	OOM	OOM
1	746.4 ± 4.099 ms	OOM	OOM
2	746.7 ± 3.556 ms	OOM	OOM
5	749.6 ± 3.396 ms	OOM	OOM

2.7B 128

warmup	forward infer ($\mu \pm \sigma$)	forward train ($\mu \pm \sigma$)	backward ($\mu \pm \sigma$)
0	171.5 ± 140.348 ms	201.3 ± 235.047 ms	291.1 ± 58.769 ms
1	125.6 ± 4.335 ms	126.9 ± 3.288 ms	271.9 ± 4.430 ms
2	125.7 ± 4.283 ms	127.5 ± 1.662 ms	272.3 ± 3.595 ms
5	126.5 ± 3.620 ms	128.1 ± 1.512 ms	274.6 ± 1.642 ms

2.7B 256

warmup	forward infer ($\mu \pm \sigma$)	forward train ($\mu \pm \sigma$)	backward ($\mu \pm \sigma$)
0	280.5 ± 137.766 ms	309.2 ± 223.872 ms	527.4 ± 78.354 ms
1	235.8 ± 8.065 ms	239.3 ± 4.057 ms	503.0 ± 6.747 ms
2	237.4 ± 7.061 ms	240.1 ± 3.730 ms	504.2 ± 4.773 ms
5	238.6 ± 6.753 ms	241.5 ± 3.193 ms	506.0 ± 2.973 ms

2.7B 512

warmup	forward infer ($\mu \pm \sigma$)	forward train ($\mu \pm \sigma$)	backward ($\mu \pm \sigma$)
0	537.6 ± 129.439 ms	573.0 ± 225.172 ms	1052.0 ± 54.629 ms
1	498.0 ± 10.498 ms	504.0 ± 6.766 ms	1036.7 ± 2.376 ms
2	498.6 ± 9.292 ms	507.2 ± 2.809 ms	1041.7 ± 2.128 ms
5	498.4 ± 5.943 ms	509.3 ± 2.356 ms	1043.9 ± 1.739 ms

2.7B 1024

warmup	forward infer ($\mu \pm \sigma$)	forward train ($\mu \pm \sigma$)	backward ($\mu \pm \sigma$)
0	1082.7 ± 133.297 ms	OOM	OOM
1	1042.1 ± 2.513 ms	OOM	OOM
2	1043.1 ± 2.084 ms	OOM	OOM
5	1043.2 ± 2.831 ms	OOM	OOM

Problem (nsys_profile)

Code:

- `cs336_systems/profile.sh`
- `cs336_systems/profile_reports.sh`
- `cs336_systems/profile.ipynb`

(a) **What is the total time spent on your forward pass? Does it match what we had measured before with the Python standard library?**

Forward infer / train time (ms)

Context Model	128	256	512	1024
small	35.4 / 42.8	34.7 / 42.2	40.5 / 46.7	69.3 / 72.7
medium	69.7 / 85.4	69.6 / 82.4	99.0 / 99.0	197.2 / 192.9
large	60.4 / 71.8	96.6 / 127.7	177.0 / 177.3	400.4 / 395.0
xl	97.0 / 104.9	165.7 / 169.9	339.4 / 334.5	755.7 / OOM
2.7B	134.1 / 131.5	243.5 / 235.4	507.4 / 493.2	1046.4 / OOM

The forward time matches what we measured earlier (is a little higher).

(b) **What CUDA kernel takes the most cumulative GPU time during the forward pass? How many times is this kernel invoked during**

a single forward pass of your model? Is it the same kernel that takes the most runtime when you do both forward and backward passes?

For an xl model with context length 512, `sm80_xmma_gemm_f32f32...` matrix multiplication kernels took the most time (123 calls for the most popular one). In the backward pass, the most time is taken by a similar matrix multiplication kernel (a transposed version on the forward pass and a non-transposed one on the backward pass).

(c) **What other kernels besides matrix multiplies do you see accounting for non-trivial CUDA runtime in the forward pass?**

I see elementwise operations, such as multiplication and addition, (`at::native::elementwise_kernel` and `at::native::vectorized_elementwise_kernel`) and reductions, such as mean and max (`at::native::reduce_kernel`).

(d) **Profile running one complete training step with your implementation of AdamW (i.e., the forward pass, computing the loss and running a backward pass, and finally an optimizer step, as you'd do during training). How does the fraction of time spent on matrix multiplication change, compared to doing inference (forward pass only)? How about other kernels?**

For an xl model with context length 512, during inference, 4 matrix multiplication kernels take about 80% of the time with the rest going to elementwise (most) and reduction operations. Over a training step, there are many more kernels run: matrix multiplication still dominates, but now takes only about 60% of the time, while pointwise operations take a relatively larger share of time.

(e) **Compare the runtime of the softmax operation versus the matrix multiplication operations within the self-attention layer of your model during a forward pass. How does the difference in runtimes compare to the difference in FLOPs?**

For an xl model with context length 512, focusing on `scaled_dot_product_attention`, softmax takes nearly as much time as computing attention scores and the final matrix multiplication combined (556 ns vs 388 and 236 ns). This is very high given that the fraction of FLOPs taken by softmax is tiny ($\sim 512^2$ vs $\sim 1280 \times 512^2$ for the other two operations). The reason for high running time is that unlike highly optimized matrix multiplications, softmax is very memory-bound.

Problem (mixed_precision_accumulation)

```
import torch

s = torch.tensor(0,dtype=torch.float32)
for i in range(1000):
    s += torch.tensor(0.01,dtype=torch.float32)
print(s)

s = torch.tensor(0,dtype=torch.float16)
for i in range(1000):
    s += torch.tensor(0.01,dtype=torch.float16)
print(s)

s = torch.tensor(0,dtype=torch.float32)
for i in range(1000):
    s += torch.tensor(0.01,dtype=torch.float16)
print(s)

s = torch.tensor(0,dtype=torch.float32)
for i in range(1000):
    x = torch.tensor(0.01,dtype=torch.float16)
    s += x.type(torch.float32)
print(s)

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

There are two sources of error: from 0.01 stored in binary (higher in float16) and from the result of addition stored in binary (again higher in float16). In the first case (everything in float32), we get minimal error; in the second case (everything in float16) we get the highest error from both sources; in the third and fourth cases (equivalent: implicit or explicity upcasting to float32) we get only the higher error from storing 0.01 in float16, but lower error from addition.

Problem (benchmarking_mixed_precision)

Code:

- `cs336_systems/benchmark_mp.sh`
- `cs336_systems/mixed_precision.ipynb`

(a) Parameters and gradients are always in float32 for reasons demonstrated in the previous problem. Activations are computed in float16 for linear layers (fc1, fc2) and relu, but float32 for LayerNorm. Logits are activations of fc2

(float16) and the loss is float32 because it's usually numerically sensitive (MSE, Cross-Entropy), but could in theory be float16, e.g. if we use some primitive loss, such as `max()`.

(b) LayerNorm contains sensitive operations: accumulations (mean, standard deviation), epsilon addition and square root. Compared to float16, bfloat16 would help avoid over- and underflows, but it has fewer bits for the mantissa leading to loss in precision. Therefore, we always use float32 for LayerNorm. In theory, hardware could fuse LayerNorm computations to produce float16/bfloat16 activations, but the benefits would be much smaller than from the fused matmul.

(c) Mixed precision did not help with OOM issues; however, as the model and the context window sizes increased, we got super-linear savings in running time as a result of better register / cache / memory utilization and more efficient tensor cores / kernels. Surprisingly, with mixed precision, the 2.7B model was often faster than the xl or even large model at small context window sizes.

Forward Inference (bf16 / fp32 ms)

context model	128	256	512	1024
small	27.7 / 24.7	16.4 / 24.8	33.2 / 27.5	35.4 / 61.9
medium	32.1 / 27.7	34.3 / 48.8	68.1 / 85.5	84.6 / 189.1
large	48.3 / 43.6	51.1 / 89.1	72.0 / 169.7	156.7 / 394.0
xl	65.7 / 80.1	119.5 / 156.9	105.2 / 330.0	262.4 / 749.6
2.7B	43.8 / 126.5	50.0 / 238.6	102.8 / 498.4	261.7 / 1043.2

Forward Training (bf16 / fp32 ms)

context model	128	256	512	1024
small	43.3 / 43.0	49.4 / 42.6	46.0 / 36.0	56.3 / 64.6
medium	58.7 / 64.8	67.7 / 63.4	83.4 / 88.2	97.5 / 194.1
large	78.6 / 113.5	79.8 / 116.2	99.5 / 173.1	164.3 / 401.8
xl	101.6 / 151.3	103.8 / 160.9	126.0 / 338.8	274.1 / OOM
2.7B	78.4 / 128.1	121.2 / 241.5	117.0 / 509.3	274.9 / OOM

Backward (bf16 / fp32 ms)

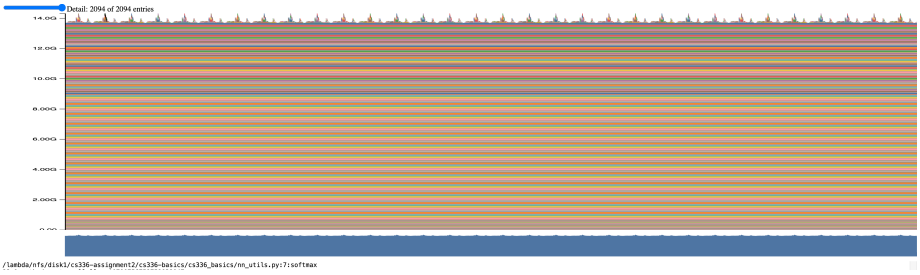
context model	128	256	512	1024
small	43.0 / 43.9	68.8 / 44.8	45.0 / 56.7	73.8 / 128.8
medium	92.7 / 69.8	76.1 / 84.4	109.2 / 172.7	172.3 / 386.8
large	106.4 / 118.5	109.2 / 178.7	153.4 / 353.5	316.3 / 819.2
xl	179.8 / 184.2	173.5 / 334.9	213.9 / 692.3	539.4 / OOM
2.7B	123.3 / 274.6	130.1 / 506.0	237.6 / 1043.9	554.7 / OOM

Problem (memory_profiling)

Code: `cs336_systems/profile_memory.sh`

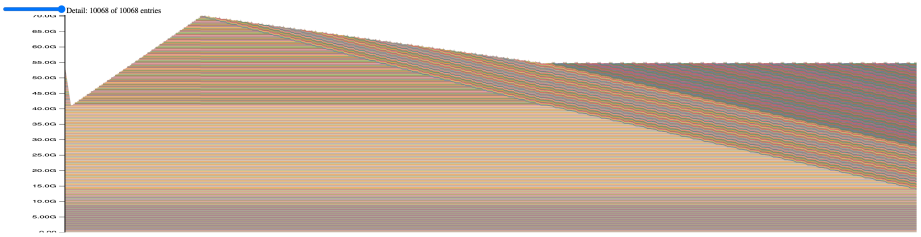
(a) **How do your memory timelines look like? Can you tell which stage is running based on the peaks you see?**

For the forward pass during inference, most memory has already been allocated and we see only small bursts. The largest peaks corresponds to `softmax`. Smaller ones correspond to other transformer block operations, such as `SiLU`.



Forward pass

For the full step, we see a small spike for `zero_grad` in the beginning, then a gradual increase in memory use during the forward pass (store activations with spikes for softmax, etc.). During the backward pass, we deallocate activations while allocating / reusing memory for gradients. At context length 128, the peak is during the backward pass (fewer attention activations), whereas at 512, the peak is achieved during the forward pass.



Training step

(b) **What is the peak memory usage of each context length when doing a forward pass? What about when doing a full training step?**

Code: `cs336_systems/memory_peaks.ipynb`

There is not much difference between the context lengths 128 and 256 because in both cases memory is mostly used by the backward pass.

Peak memory in GB (bf16 / fp32)

Context	128	256	512
forward pass	19.2 / 12.9	19.3 / 13.0	19.6 / 13.4
training step	51.5 / 51.5	52.1 / 51.5	62.7 / 65.6

(c) **Does mixed-precision significantly affect memory usage?**

Not significantly. We instantiate the model in float32 and then use autocast for the forward pass, so we increase memory consumption during warmup - before taking measurements. At the same time, AMP decreases the size of activations, which matters only with larger contexts (we see the difference at 512).

(d) **At our reference hyperparameters, what is the size of a tensor of activations in the Transformer residual stream, in single-precision?**

$\frac{d_k \times context\_length \times batch\_size \times bytes\_in\_float}{bytes\_in\_megabyte} = \frac{2560 \times 512 \times 4 \times 4}{1024^2} = 20$ (MB).

(e) **What is the size of the largest allocations shown? Looking through the stack trace, can you tell where those allocations come from?**

The largest alloctions have size 128 MB and come from attention's softmax.

$\frac{batch\_size \times num\_heads \times context\_size^2 \times bytes\_in\_float}{bytes\_in\_megabyte} = \frac{4 \times 32 \times 512^2 \times 4}{1024^2} = 128$

1.2 Optimizing Attention with FlashAttention-2

Problem (pytorch_attention)

(a) **Report the timings (or out-of-memory errors) you get for these configurations. At what size do you get out-of-memory errors? Do the accounting for the memory usage of attention in one of the smallest configurations you find that runs out of memory (you can use the equations for memory usage of Transformers from Assignment 1). How does the memory saved for backward change with the sequence length? What would you do to eliminate this memory cost?**

Code:

- `cs336_systems/benchmark_attention.py`
- `cs336_systems/benchmark_att.sh`
- `cs336_systems/attention.ipynb`

Forward time in ms (eager / compiled)

Context Dimension	256	1024	4096	8192	16384
16	0.2 / 0.1	0.3 / 0.2	3.9 / 1.4	15.2 / 5.4	59.9 / 21.5
32	0.2 / 0.1	0.3 / 0.2	4.0 / 1.6	15.8 / 6.0	62.9 / 24.3
64	0.2 / 0.1	0.4 / 0.2	4.5 / 2.0	17.6 / 7.8	69.9 / 31.8
128	0.2 / 0.1	0.4 / 0.3	5.4 / 3.0	21.9 / 12.4	85.9 / 48.4

Backward time in ms (eager / compiled)

Context Dimension	256	1024	4096	8192	16384
16	1.5 / 1.4	1.5 / 1.3	9.5 / 3.6	34.6 / 12.1	134.6 / 50.3
32	1.6 / 1.5	1.6 / 1.7	9.2 / 3.7	35.0 / 12.7	137.7 / 53.6
64	1.6 / 1.6	1.1 / 1.8	10.1 / 4.2	36.9 / 14.4	144.6 / 60.6
128	1.5 / 1.5	1.7 / 1.7	10.9 / 5.0	41.1 / 18.8	160.5 / 76.4

Memory size in GB (eager / compiled)

Context Dimension	256	1024	4096	8192	16384
16	66.5 / 66.5	98.1 / 98.1	584.4 / 584.5	2128.8 / 2129.0	8289.5 / 8290.0
32	67.0 / 67.0	100.1 / 100.1	592.4 / 592.5	2144.8 / 2145.0	8321.5 / 8322.0
64	68.0 / 68.0	104.1 / 104.1	608.4 / 608.5	2176.8 / 2177.0	8385.5 / 8386.0
128	70.0 / 70.0	112.1 / 112.1	640.4 / 640.5	2240.8 / 2241.0	8513.5 / 8514.0

On an H-100, we do not run out of memory from a single attention, though the memory usage increases dramatically with the context length. The memory saved is:

- inputs (Q, K, V): $context \times d_{\text{model}}$ each;
- attention: $context^2$;
- output: $context \times d_{\text{model}}$.

With batch size 8, context size 16384, and $d_{\text{model}} = 128$, we have:

$\frac{4}{1024^2} \times (8 \times (16384^2 + 4 \times 16384 \times 128)) = 8448$ (MB)

This is close to the 8514 MB peak reported by pytorch. While the inputs and the output are relatively small, with long contexts, most memory is consumed by the intermediate attention matrix. We need to perform the computation differently to avoid storing this huge intermediate result.

1.3 Benchmarking JIT-Compiled Attention

Problem (torch_compile)

(a) **Extend your attention benchmarking script to include a compiled version of your PyTorch implementation of attention, and compare its performance to the uncompiled version with the same configuration as the `pytorch_attention` problem above.**

See the tables above. We do not save on memory, but there are significant savings on running time with larger contexts.

(b) **Now, compile your entire Transformer model in your end-to-end benchmarking script. How does the performance of the forward pass change? What about the combined forward and backward passes and optimizer steps?**

Code:

- `cs336_systems/benchmark_comp.sh`
- `cs336_systems/compiled.ipynb`

Forward inference time in ms (eager / compiled)

Context Model	128	256	512	1024
small	25.0 / 5.9	26.4 / 10.0	26.9 / 19.8	62.3 / 42.6
medium	49.3 / 16.9	49.0 / 33.9	87.2 / 67.0	192.1 / 142.4
large	73.0 / 36.4	91.2 / 77.7	171.6 / 139.7	403.3 / 311.4
xl	146.5 / 69.2	159.1 / 140.2	336.4 / 288.1	778.1 / 618.8
2.7B	128.8 / 121.3	243.8 / 227.8	516.0 / 470.1	1104.8 / 967.1

Forward train time in ms (eager / compiled)

Context Model	128	256	512	1024
small	33.3 / 7.2	33.8 / 10.7	29.5 / 20.3	62.7 / 44.9
medium	60.6 / 17.9	48.8 / 34.1	86.2 / 68.7	192.7 / 144.5
large	92.1 / 36.9	87.4 / 75.6	169.6 / 138.9	417.7 / 320.9
xl	117.2 / 69.0	158.1 / 134.8	343.8 / 286.1	OOM / OOM
2.7B	125.0 / 115.2	240.2 / 222.1	532.8 / 469.2	OOM / OOM

Backward time in ms (eager / compiled)

Context Model	128	256	512	1024
small	67.4 / 19.7	64.1 / 23.5	57.2 / 42.9	127.0 / 89.7
medium	104.3 / 45.4	103.9 / 73.1	170.0 / 138.4	391.4 / 289.1
large	102.6 / 84.7	187.6 / 156.9	359.8 / 295.3	877.7 / 657.6
xl	208.8 / 161.4	349.5 / 303.8	760.9 / 620.0	OOM / OOM
2.7B	276.5 / 255.5	535.2 / 487.0	1233.6 / 1067.9	OOM / OOM

Optimizer time in ms (eager / compiled)

Context Model	128	256	512	1024
small	19.8 / 22.1	19.8 / 21.0	18.4 / 11.7	11.5 / 11.6
medium	35.0 / 46.2	33.8 / 31.7	36.8 / 31.1	32.4 / 31.0
large	72.9 / 74.3	73.3 / 72.5	74.2 / 72.6	74.6 / 75.0
xl	160.2 / 157.9	159.1 / 159.5	159.7 / 159.9	OOM / OOM
2.7B	264.7 / 265.2	264.6 / 264.5	265.4 / 264.6	OOM / OOM

The results are with batch size 4. Compilation reduces the running time of both the forward and backward passes, but less so for larger models and contexts. It has virtually no effect on the optimizer step because it's not compiled.

Problem (flash_forward)

Code:

- `cs336_systems/flash_pytorch.py`
- `cs336_systems/flash_triton.py`

Problem (flash_backward)

Code: `cs336_systems/flash_triton.py`

Problem (flash_benchmarking)

Code:

- `cs336_systems/benchmark_flash.py`
- `cs336_systems/benchmark_flash.sh`
- `cs336_systems/benchmark_flash.ipynb`

The general conclusions are:

- The pytorch implementation is OK for short sequences, but takes increasingly more time with longer ones, eventually getting OOM in the backward pass with sequence length 65536.
- The forward pass of the two implementations of flash attention (compiled torch backward and triton backward passes) is the same, yet I got very different results for the two. I am not sure where this is coming from.
- The Triton backward pass shines where parallelism matters: longer contexts and smaller inner dimensions. The pytorch compiled version is better in the opposite scenarios (shorter contexts / larger inner dimensions).

Forward float32 time in ms (naive pytorch / flash torch back / flash all triton)

Context Dimension	128	256	512	1024
16	42.3 / 84.0 / 112.3	71.8 / 97.6 / 108.3	81.8 / 146.0 / 102.9	82.3 / 111.9 / 110.0
32	61.5 / 82.7 / 106.7	67.4 / 141.2 / 114.0	68.9 / 99.5 / 125.9	80.6 / 103.4 / 144.7
64	71.9 / 84.7 / 102.8	71.0 / 146.1 / 110.1	77.0 / 111.4 / 122.3	80.4 / 152.2 / 119.2
128	74.0 / 82.7 / 108.7	62.9 / 158.6 / 109.3	66.7 / 105.0 / 118.9	90.9 / 112.9 / 117.2

Forward float32 time in ms (naive pytorch / flash torch back / flash all triton)

Context Dimension	2048	4096	8192
16	137.8 / 147.1 / 130.6	592.8 / 118.9 / 126.3	2232.9 / 315.1 / 315.7
32	139.6 / 113.8 / 118.0	592.7 / 153.1 / 164.0	2237.5 / 395.9 / 395.5
64	142.4 / 105.0 / 115.8	596.8 / 162.0 / 165.8	2252.3 / 387.9 / 387.4
128	150.0 / 113.3 / 129.0	611.0 / 225.5 / 224.6	2259.5 / 826.3 / 819.5

Forward float32 time in ms (naive pytorch / flash torch back / flash all triton)

Context Dimension	16384	32768	65536
16	8664.0 / 988.8 / 988.6	34441.4 / 3304.0 / 3332.8	140241.2 / 12921.7 / 14674.7
32	8697.3 / 1348.4 / 1349.2	34712.4 / 4459.0 / 4621.0	141691.0 / 17526.8 / 21094.3
64	8881.4 / 1386.5 / 1410.9	35660.8 / 4557.7 / 4748.0	146422.2 / 18323.9 / 27142.8
128	8863.3 / 2584.2 / 2609.3	36622.9 / 8827.9 / 9358.3	144180.9 / 40416.3 / 40508.5

Forward bfloat16 time in ms (naive pytorch / flash torch back / flash all triton)

Context Dimension	128	256	512	1024
16	45.3 / 85.4 / 87.5	76.8 / 86.8 / 106.1	74.0 / 88.3 / 95.4	74.8 / 101.0 / 125.9
32	69.9 / 82.7 / 113.2	74.4 / 88.4 / 104.3	81.3 / 87.1 / 105.7	82.3 / 88.7 / 109.1
64	77.1 / 87.4 / 108.4	70.6 / 93.4 / 111.7	74.0 / 87.6 / 125.4	83.6 / 91.0 / 117.2
128	77.9 / 84.8 / 107.8	62.1 / 98.1 / 103.6	77.2 / 88.9 / 107.9	80.5 / 103.6 / 120.3

Forward bfloat16 time in ms (naive pytorch / flash torch back / flash all triton)

Context Dimension	2048	4096	8192
16	124.7 / 97.8 / 110.0	382.3 / 95.9 / 124.1	1442.3 / 197.7 / 198.9
32	125.3 / 97.0 / 112.6	382.3 / 95.9 / 113.5	1441.0 / 199.8 / 199.8
64	125.4 / 94.3 / 114.3	382.1 / 99.6 / 122.3	1447.0 / 236.2 / 236.2
128	122.8 / 97.8 / 114.8	384.4 / 165.8 / 169.3	1448.2 / 615.8 / 625.7

Forward bfloat16 time in ms (naive pytorch / flash torch back / flash all triton)

Context Dimension	16384	32768	65536
16	5344.0 / 515.4 / 510.5	20914.6 / 1627.4 / 1608.6	90275.4 / 5450.6 / 5607.5
32	5354.4 / 531.6 / 530.7	21295.4 / 1666.9 / 1667.9	92260.6 / 5760.5 / 6309.2
64	5368.0 / 742.4 / 743.9	22200.0 / 2244.7 / 2286.5	97241.0 / 7802.4 / 9784.7
128	5426.1 / 1780.5 / 1787.5	22855.7 / 5724.3 / 5656.8	102297.4 / 23283.3 / 25387.1

Backward float32 time in s (naive pytorch / flash torch back / flash all triton)

Context Dimension	128	256	512	1024
16	0.9 / 0.5 / 0.5	0.9 / 1.2 / 1.2	1.0 / 1.3 / 1.2	0.9 / 0.4 / 1.3
32	0.9 / 1.2 / 0.5	0.9 / 1.3 / 1.2	0.9 / 0.4 / 1.3	0.9 / 0.4 / 2.2
64	1.0 / 0.6 / 0.5	0.9 / 1.3 / 1.2	0.9 / 0.4 / 1.4	1.0 / 1.2 / 1.2
128	0.9 / 0.3 / 0.5	0.8 / 1.3 / 0.7	0.9 / 0.4 / 1.5	1.0 / 0.4 / 1.1

Backward float32 time in s (naive pytorch / flash torch back / flash all triton)

Context Dimension	2048	4096	8192
16	1.0 / 0.4 / 1.1	1.2 / 0.4 / 1.1	4.6 / 1.4 / 1.3
32	1.0 / 0.4 / 1.2	1.2 / 0.4 / 1.2	4.6 / 1.4 / 1.2
64	1.0 / 0.4 / 1.2	1.9 / 0.5 / 1.2	4.6 / 1.4 / 1.9
128	0.9 / 0.4 / 1.2	2.1 / 0.5 / 1.3	4.6 / 1.5 / 3.8

Backward float32 time in s (naive pytorch / flash torch back / flash all triton)

Context Dimension	16384	32768	65536
16	18.1 / 5.5 / 1.5	72.2 / 21.7 / 5.0	OOM / 87.7 / 21.5
32	18.1 / 5.6 / 2.7	72.5 / 22.1 / 8.9	OOM / 89.5 / 34.7
64	18.3 / 5.8 / 6.2	73.0 / 22.8 / 22.3	OOM / 92.7 / 99.3
128	18.3 / 6.0 / 12.6	72.9 / 24.0 / 46.2	OOM / 91.5 / 190.9

Backward bfloat16 time in s (naive pytorch / flash torch back / flash all triton)

Context Dimension	128	256	512	1024
16	1.0 / 0.5 / 0.6	0.9 / 0.4 / 1.2	0.9 / 1.4 / 0.9	1.0 / 0.5 / 1.3
32	0.9 / 2.0 / 0.6	0.9 / 1.4 / 1.2	0.9 / 0.5 / 1.2	1.0 / 0.5 / 2.1
64	0.9 / 0.7 / 0.6	0.8 / 1.4 / 0.8	1.0 / 0.5 / 1.5	0.9 / 1.3 / 1.2
128	0.9 / 0.5 / 0.6	0.9 / 1.3 / 0.8	1.0 / 0.5 / 1.4	1.0 / 0.5 / 1.2

Backward bfloat16 time in s (naive pytorch / flash torch back / flash all triton)

Context Dimension	2048	4096	8192
16	1.0 / 0.5 / 1.1	1.0 / 0.5 / 1.2	2.8 / 1.3 / 1.0
32	0.9 / 0.5 / 0.9	1.1 / 0.5 / 1.2	2.8 / 1.3 / 1.3
64	0.9 / 0.5 / 1.2	1.0 / 0.6 / 1.3	2.8 / 1.3 / 1.7
128	1.0 / 0.5 / 1.2	2.0 / 0.6 / 1.3	2.8 / 1.4 / 3.0

Backward bfloat16 time in s (naive pytorch / flash torch back / flash all triton)

Context Dimension	16384	32768	65536
16	10.8 / 4.9 / 1.9	42.7 / 19.3 / 6.0	181.4 / 77.5 / 21.2
32	10.8 / 5.0 / 2.7	43.2 / 19.7 / 8.7	182.7 / 78.8 / 34.0
64	10.8 / 5.2 / 5.6	45.0 / 20.2 / 19.9	185.1 / 81.5 / 82.3
128	10.8 / 5.3 / 8.8	46.4 / 21.1 / 29.4	185.7 / 82.0 / 114.9

Forward-backward float32 time in s (naive pytorch / flash torch back / flash all triton)

Context Dimension	128	256	512	1024
16	1.8 / 1.5 / 1.7	1.8 / 3.1 / 2.1	1.8 / 3.2 / 2.1	1.9 / 1.5 / 2.4
32	1.8 / 3.2 / 1.8	1.8 / 3.2 / 1.8	1.8 / 1.5 / 2.1	1.8 / 1.5 / 2.7
64	1.8 / 1.7 / 1.8	1.8 / 3.2 / 2.2	1.9 / 1.5 / 2.5	1.9 / 3.2 / 2.1
128	1.8 / 1.5 / 1.7	1.8 / 3.2 / 1.6	1.8 / 1.6 / 2.4	1.8 / 1.6 / 2.2

Forward-backward float32 time in s (naive pytorch / flash torch back / flash all triton)

Context Dimension	2048	4096	8192
16	1.8 / 1.5 / 2.1	1.9 / 1.6 / 2.0	6.9 / 1.8 / 2.1
32	1.8 / 1.5 / 2.1	1.9 / 1.6 / 2.1	6.9 / 1.9 / 2.2
64	1.9 / 1.6 / 2.1	3.6 / 1.7 / 2.2	6.9 / 1.8 / 2.3
128	1.9 / 1.5 / 2.1	3.7 / 1.6 / 2.1	6.9 / 2.4 / 4.6

Forward-backward float32 time in s (naive pytorch / flash torch back / flash all triton)

Context Dimension	16384	32768	65536
16	26.7 / 6.5 / 2.4	106.5 / 25.1 / 8.4	OOM / 101.4 / 36.5
32	26.8 / 6.9 / 4.1	107.1 / 26.6 / 13.3	OOM / 107.7 / 61.2
64	27.2 / 7.2 / 7.6	108.1 / 27.5 / 26.7	OOM / 111.7 / 131.7
128	27.1 / 8.6 / 15.2	110.4 / 33.4 / 55.5	OOM / 130.4 / 244.8

Forward-backward bfloat16 time in s (naive pytorch / flash torch back / flash all triton)

Context Dimension	128	256	512	1024
16	1.8 / 1.5 / 1.9	1.8 / 1.6 / 2.2	1.8 / 3.4 / 2.0	1.8 / 1.6 / 2.7
32	1.8 / 3.4 / 1.7	1.8 / 3.4 / 2.2	1.8 / 1.6 / 2.2	1.8 / 1.6 / 3.8
64	1.8 / 1.8 / 1.7	1.8 / 3.4 / 2.0	1.8 / 1.6 / 2.7	1.9 / 3.4 / 2.2
128	1.8 / 1.7 / 1.7	1.8 / 3.4 / 1.7	1.8 / 1.6 / 2.6	1.9 / 1.6 / 2.2

Forward-backward bfloat16 time in s (naive pytorch / flash torch back / flash all triton)

Context Dimension	2048	4096	8192
16	1.8 / 1.6 / 2.1	2.0 / 1.7 / 2.1	4.3 / 2.1 / 2.2
32	1.9 / 1.6 / 2.1	2.1 / 1.7 / 2.2	4.3 / 2.0 / 2.2
64	1.9 / 1.7 / 2.2	2.1 / 1.8 / 2.3	4.3 / 2.1 / 2.2
128	2.0 / 1.8 / 2.2	3.7 / 1.8 / 2.2	4.3 / 2.1 / 3.6

Forward-backward bfloat16 time in s (naive pytorch / flash torch back / flash all triton)

Context Dimension	16384	32768	65536
16	16.1 / 5.4 / 2.3	65.4 / 20.9 / 7.5	275.2 / 83.5 / 27.0
32	16.1 / 5.5 / 3.2	65.6 / 21.4 / 10.6	281.6 / 85.5 / 40.2
64	16.2 / 5.9 / 6.3	68.3 / 22.5 / 22.0	286.5 / 89.6 / 93.5
128	16.3 / 7.1 / 10.8	71.1 / 26.8 / 35.0	294.0 / 108.0 / 142.6

Leaderboard

On an H-100, I got 48.3ms with the Triton implementation of both forward and backward passes. I did not perform any further improvements (except for autotuning), so the result is expectedly worse than the top 5 leaderboard times. With triton forward and pytorch backward passes, I got 83.7ms, which roughly corresponds to the 80ms naïve baseline (with `torch.set_float32_matmul_precision("high")`).

2.1 Single-Node Distributed Communication in PyTorch

Problem (distributed_communication_single_node)

Code:

- `cs336_systems/single_node.sh`
- `cs336_systems/single_node.py`
- `cs336_systems/single_node.ipynb`

Using an 8-GPU 40GB A100 SXM4 instance, I got the following results:

Mean latency: GLoo / NCCL (ms)

World	2	4	6
1MB	0.83 / 0.13	2.05 / 0.15	6.58 / 0.17
10MB	11.60 / 0.18	20.73 / 0.26	23.71 / 0.29
100MB	106.69 / 0.94	173.00 / 1.03	178.04 / 1.10
1GB	1088.02 / 6.35	2166.65 / 8.34	2034.02 / 9.09

Comments:

- Using GPUs (NCCL) is orders of magnitude faster than using a CPU (NCCL).

- Time grows significantly (roughly linearly) with tensor size - more rapidly on a CPU.
- Time grows mildly with the world size - again more rapidly on a CPU.

2.2 A Naïve Implementation of Distributed Data Parallel Training

Problem (naive_ddp)

Code:

- `cs336_systems/ddp_benchmark.py`
- `cs336_systems/ddp_benchmark.sh`
- `cs336_systems/ddp_benchmark.ipynb`

Problem (naive_ddp_benchmarking)

All the tests were performed on a 2-GPU H-100 SXM5 instance. The sequence length is fixed at 256 and the batch size varies from 2 to the maximum of 32. Computation is not optimized (eager and at float32). The step would've been more communication-bound with those optimizations. There are 5 warmup steps and 10 measurement steps. For statistics, we take the max over ranks and analyze mean / std over steps.

Naïve DDP

Batch	Step Time ($\mu \pm \sigma$)	Comm Time ($\mu \pm \sigma$)	Comm Frac
2	287.64 ± 1.47 ms	42.17 ± 0.37 ms	14.66%
4	346.32 ± 1.43 ms	42.57 ± 0.37 ms	12.29%
8	499.58 ± 0.79 ms	45.01 ± 0.68 ms	9.01%
16	808.38 ± 0.83 ms	43.47 ± 0.29 ms	5.38%
32	1438.02 ± 0.11 ms	48.39 ± 0.15 ms	3.36%

The total step time increases (roughly linearly) with the batch size. The communication time increases only slightly because larger microbatches make training more computation bound.

2.3 Improving Upon the Minimal DDP Implementation

Code:

- `cs336_systems/ddp_benchmark.py`
- `cs336_systems/ddp_benchmark.sh`
- `cs336_systems/ddp_benchmark.ipynb`

Problem (minimal_ddp_flat_benchmarking)

Flat DDP

Batch	Step Time ($\mu \pm \sigma$)	Comm Time ($\mu \pm \sigma$)	Comm Frac
2	292.12 ± 2.4 ms	48.2 ± 0.41 ms	16.5%
4	344.75 ± 1.58 ms	43.51 ± 0.47 ms	12.62%
8	496.84 ± 0.31 ms	43.44 ± 0.26 ms	8.74%
16	808.65 ± 2.3 ms	46.81 ± 2.48 ms	5.79%

The batched all-reduce call did not help at all; moreover, having to allocate flat gradient tensors, we ran out of memory with the batch size of 32. A 2-GPU H-100 SXM5 instance already handles multiple communication calls very well, but we probably would have seen an improvement if we had used more GPUs or a worse interconnect.

Problem (ddp_overlap_individual_parameters)

Code: `cs336_systems/ddp_overlap_individual.py`

Problem (ddp_overlap_individual_parameters_benchmarking)

(a) **Benchmarking**

Overlap Individual Parameters

Batch	Ind Overlap Step Time	Naïve Step Time	Naïve Comm Time	Comm Improvement
2	275.0 ms	287.6 ms	42.2 ms	30.1%
4	325.1 ms	346.3 ms	42.6 ms	49.8%
8	478.6 ms	499.6 ms	45.0 ms	46.6%
16	788.9 ms	808.4 ms	43.5 ms	44.8%
32	1417.9 ms	1438.0 ms	48.4 ms	41.6%

Comparing with the naïve implementation (since the batched all-reduce did not go well), we see that while the overall improvement is small (we are already mostly computation bound), the improvement in the communication overhead is significant.

(b) **Nsight profiling**

In the naïve implementation, we block for NCCL to complete synchronization.

Block for Gradient Synchronization

Below, we clearly see the overlap of NCCL communication kernels with CUDA compute kernels.

Overlap Backward Pass Computation

Problem (ddp_overlap_bucketed)

Code: `cs336_systems/ddp_overlap_bucketed.py`

Problem (ddp_bucketed_benchmarking)

(a) **Benchmark your bucketed DDP implementation using the same config as the previous experiments (1 node, 2 GPUs, XL model size), varying the maximum bucket size (1, 10, 100, 1000 MB). Compare your results to the previous experiments without bucketing—do the results align with your expectations? If they don't align, why not? You may have to use the PyTorch profiler as necessary to better understand how communication calls are ordered and/or executed. What changes in the experimental setup would you expect to yield results that are aligned with your expectations?**

Overlap Bucketed Parameters

Batch	Naïve	Ind Overlap	1 MB Bucket	10 MB Bucket	100 MB Bucket	1000 MB Bucket
2	287.6 ms	275.0 ms	295.9 ms	292.2 ms	278.2 ms	271.8 ms
4	346.3 ms	325.1 ms	327.0 ms	331.5 ms	339.2 ms	329.4 ms
8	499.6 ms	478.6 ms	476.5 ms	477.2 ms	485.4 ms	482.0 ms
16	808.4 ms	788.9 ms	791.6 ms	790.4 ms	799.9 ms	789.3 ms
32	1438.0 ms	1417.9 ms	1428.2 ms	1416.2 ms	1428.0 ms	1419.5 ms

While bucketed overlapping is almost always better than the naïve solution, it is almost always worse than individual parameter overlapping. As we have seen in the batch vs naïve comparison above, with 2 GPUs and a fast interconnect, having a lot of small communication calls does not add too much overhead. On the other hand, with simple online bucketing, we wait for the bucket to be almost overfilled before synchronizing

it, so the last bucket is synchronized at the very end without overlapping with communication. A better solution would be to use intelligent pre-bucketing: assign parameters to buckets on the first pass, have smaller buckets toward the end, and syncronize each bucket as soon as it gets all the parameters assigned to it.

(b) **Assume that the time it takes to compute the gradients for a bucket is identical to the time it takes to communicate the gradient buckets. Write an equation that models the communication overhead of DDP (i.e., the amount of additional time spent after the backward pass) as a function of the total size (bytes) of the model parameters (s), the all-reduce algorithm bandwidth (w, computed as the size of each rank's data divided by the time it takes to finish the all-reduce), the overhead (seconds) associated with each communication call (o), and the number of buckets (n_b). From this equation, write an equation for the optimal bucket size that minimizes DDP overhead.**

For simplicity, let us make an additional assumption that all the buckets have the same size $\frac{s}{n_b}$. Then the DDP overhead is composed of the overhead per bucket plus the time it takes to communicate the last bucket:

$$\boxed{T = n_b o + \frac{s}{n_b w}}$$

$T \to \infty$ when $n_b \to 0$ or $n_b \to \infty$; therefore the minimimum is achieved when $\frac{dT}{dn_b} = 0$:

$\frac{dT}{dn_b} = 0 \Leftrightarrow o - \frac{s}{n_b^2 w} = 0 \Leftrightarrow n_b = \sqrt{\frac{s}{ow}}$

Therefore, the optimal bucket size is $\frac{s}{\sqrt{\frac{s}{ow}}}$: $\boxed{B = \sqrt{sow}}$.

2.4 4D Parallelism

Problem (communication_accounting)

(a) **How much memory would it take to store the master model weights, accumulated gradients and optimizer states in FP32 on a single device? How much memory is saved for backward (these will be in BF16)? How many H100 80GB GPUs worth of memory is this?**

Number of weights: $W = 2 \times num\_blocks \times d_{\text{model}} \times d_{\text{ff}} = 3276 \times 2^{26}$

Memory for weights, accumulated gradients, and optimizer states:

$M_1 = \frac{4 \times 4 \times W}{2^{30}} = 3276$ (GB)

Activations per token:

$A = num\_blocks \times (d_{\text{model}} + d_{\text{ff}}) = 1071 \times 2^{13}$

For the given batch size (B) and sequence length (L), the memory needed for activations is:

$M_2 = \frac{2 \times BL \times A}{2^{30}} = \frac{1071BL}{2^{16}}$ (GB)

The number of H100 80GB GPUs needed:

$N = \left\lceil \frac{3276 + \frac{1071BL}{2^{16}}}{80} \right\rceil$

For our huge model, if we have 1M tokens/step, e.g. $B = 512$ and $L = 2048$, then

$N = \lceil \frac{3276+17136}{80} \rceil = 256$ (H100 80GB GPUs)

(b) **Now assume your master weights, optimizer state, gradients and half of your activations (in practice every second layer) are sharded across N_{FSDP} devices. Write an expression for how much memory this would take per device. What value does N_{FSDP} need to be for the total memory cost to be less than 1 v5p TPU (95GB per device)?**

Per device, we would need

$M_{\text{device}} = \frac{M_1 + \frac{M_2}{2}}{N_{\text{FSDP}}} = \frac{3276 + \frac{1071BL}{2^{17}}}{N_{\text{FSDP}}}$ (GB)

For $M_{\text{device}} \leq 95$, we would need at least $N_{\text{FSDP}} = \left\lceil \frac{3276 + \frac{1071BL}{2^{17}}}{95} \right\rceil$ devices.

For example, with our earlier choice of $BL = 2^{20}$, we get

$N_{\text{FSDP}} = \lceil \frac{3276+1071 \times 8}{95} \rceil = 125$ (devices).

(c) **Consider only the forward pass. Use the communication bandwidth of $W_{\text{ici}} = 2 \cdot 9 \cdot 10^{10}$ and FLOPS/s of $C = 4.6 \cdot 10^{14}$ for TPU v5p as given in the TPU Scaling Book. Following the notation of the Scaling Book, use $M_X = 2$, $M_Y = 1$ (a 3D mesh), with $X = 16$ being your FSDP dimension, and $Y = 4$ being your TP dimension. At what per-device batch size is this model compute bound? What is the overall batch size in this setting?**

The TPU scaling book gives the following compute-bound inequality:

$\frac{B}{N} > \frac{\alpha^2}{M_X M_Y F}$

Here, $N = XY$ is the total number of chips, $\alpha = \frac{C}{W_{\text{ici}}}$ is the ICI arithmetic intensity, and $F = d_{\text{ff}}$ is the feed-forward dimension. Thus, $\frac{B}{N} > 61.325$. For the microbatch size of 62, the overall batch size is $62 \times 16 \times 4 = 3968$.

(d) **In practice, we want the overall batch size to be as small as possible, and we also always use our compute effectively (in other words we want to never be communication bound). What other tricks can we employ to reduce the batch size of our model but retain high throughput?**

The batch should be large enough to have low noise per step; however, once the noise is low, further increasing the batch size would result in wasteful computation.

eff_batch = microbatch × grad_acc × world_size.

To keep the effective batch small while retaining high throughput:

- Increase the microbatch size and reduce the frequency of gradient synchronization, and / or choose a smaller data-parallel world size.
- Use activation checkpointing, i.e. recompute activations instead of storing them to help us increase the microbatch size without running out of memory.
- Use other types of model parallelism, such as pipeline and expert (if relevant), to help us remain computation bound without increasing the batch size.

3 Optimizer State Sharding

Code:

- `cs336_systems/ddp_benchmark.py`
- `cs336_systems/ddp_benchmark.sh`
- `cs336_systems/ddp_benchmark.ipynb`

Problem (optimizer_state_sharding)

Code: `cs336_systems/sharded_optimizer.py`

Problem (optimizer_state_sharding_accounting)

(a) **Create a script to profile the peak memory usage when training language models with and without optimizer state sharding. Using the standard configuration (1 node, 2 GPUs, XL model size), 34 report the peak memory usage after model initialization, directly before the optimizer step, and directly after the optimizer step. Do the results align with your expectations? Break down the memory usage in each setting (e.g., how much memory for parameters, how much for optimizer states, etc.).**

We are comparing runs without and with optimizer state sharding in the same setup as above using individual parameter overlap. The reported memory peaks are the maxima among both ranks and iterations (excluding the warm-up).

Optimizer Sharding - Peak Memory

Batch	After Init	Before Optim	After Optim
2	7.6 / 7.6 GB	30.6 / 23.0 GB	38.2 / 26.8 GB
4	7.6 / 7.6 GB	30.6 / 24.8 GB	38.1 / 26.7 GB
8	7.6 / 7.6 GB	34.4 / 30.8 GB	38.1 / 26.7 GB
16	7.6 / 7.6 GB	45.3 / 41.5 GB	37.7 / 26.4 GB
32	7.6 / 7.6 GB	68.2 / 64.4 GB	37.6 / 26.5 GB

- Initially, we need memory for the model ($\sim 4 \times 1.44$ GB) plus overhead. Note that gradients and the optimizer state are allocated lazily.
- Before the optimizer step, we additionally have the optimizer state ($\sim 4 \times 1.44$ GB sharded or $\sim 8 \times 1.44$ GB unsharded) and some overlap of activations ($batch\_size \times context\_size \times 4 \times 19.7$ MB) and gradients ($\sim 4 \times 1.44$ GB). The last part is dominated by gradients (batch-size independent) for small batches and by activations (batch-size dependent) for large batches.
- After the optimizer step, we no longer have activations (i.e. are independent of batch size), and so we hold the model weights, the gradients, sharded or unsharded optimizer weights; in addition, temporary allocations for weight updates increase the peak.

(b) **How does our implementation of optimizer state sharding affect training speed?**

Optimizer Sharding

Batch	Not Sharded Opt	Sharded Opt
2	275.0 ms	273.1 ms
4	325.1 ms	327.4 ms
8	478.6 ms	480.9 ms
16	788.9 ms	789.6 ms
32	1417.9 ms	1418.2 ms

Optimizer state sharding by itself has almost no effect on training speed because the optimization step is relatively fast anyway. The main goal is to decrease the peak memory usage, thus allowing larger microbatches.

(c) **How does our approach to optimizer state sharding differ from ZeRO stage 1 (described as ZeRO- DP P_{os} in Rajbhandari et al., 2020)?**

Conceptually, it is the same as ZeRO stage 1; however, the implementation is simplified. They write:

> We perform an all-gather across the data parallel process at the end of each training step to get the fully updated parameters across all data parallel process.

Our approach broadcasts every parameter individually, which increases the communication cost. Doing an all-gather would require all input tensors (one per rank) to have the same shape and the same datatype. This could be achieved by flattening / unflattenting parameters using per-datatype buckets.