CS 336: Assignment 2

Brandon Snider

April 29, 2025

Contents

1.1 Profiling and Benchmarking

Problem (`benchmarking_script`): 4 points

a) See `cs336_systems/benchmark.py` and `cs336_systems/benchmark.sh`

b) Benchmarking results (CUDA, **5 warmup steps**, 10 timed steps, varying sequence length):

Model	Forward ($\mu \pm \sigma$)	Backward ($\mu \pm \sigma$)	Total ($\mu \pm \sigma$)
small	15.839 ± 0.974 ms	15.571 ± 0.080 ms	31.411 ± 1.044 ms
medium	30.328 ± 0.161 ms	30.963 ± 0.067 ms	61.291 ± 0.191 ms
large	45.521 ± 0.777 ms	46.152 ± 0.274 ms	91.673 ± 1.043 ms
xl	60.960 ± 0.977 ms	68.650 ± 0.038 ms	129.610 ± 0.997 ms
2.7B	42.300 ± 0.587 ms	86.600 ± 0.066 ms	128.900 ± 0.607 ms

Table 1: Benchmarking Results (sequence length = 128)

Model	Forward ($\mu \pm \sigma$)	Backward ($\mu \pm \sigma$)	Total ($\mu \pm \sigma$)
small	15.356 ± 0.062 ms	16.091 ± 0.098 ms	31.447 ± 0.140 ms
medium	30.226 ± 0.072 ms	31.871 ± 0.100 ms	62.098 ± 0.156 ms
large	45.633 ± 0.211 ms	61.766 ± 0.146 ms	107.399 ± 0.285 ms
xl	62.196 ± 0.671 ms	107.093 ± 0.215 ms	169.289 ± 0.602 ms
2.7B	45.968 ± 0.122 ms	132.788 ± 0.162 ms	178.756 ± 0.274 ms

Table 2: Benchmarking Results (sequence length = 256)

Model	Forward ($\mu \pm \sigma$)	Backward ($\mu \pm \sigma$)	Total ($\mu \pm \sigma$)
small	15.943 ± 0.266 ms	22.940 ± 0.028 ms	38.883 ± 0.277 ms
medium	32.684 ± 0.923 ms	57.238 ± 0.172 ms	89.922 ± 1.089 ms
large	49.374 ± 0.525 ms	116.740 ± 0.074 ms	166.114 ± 0.503 ms
xl	81.372 ± 0.152 ms	200.596 ± 0.230 ms	281.968 ± 0.327 ms
2.7B	89.902 ± 0.273 ms	236.174 ± 0.117 ms	326.076 ± 0.304 ms

Table 3: Benchmarking Results (sequence length = 512)

Model	Forward ($\mu \pm \sigma$)	Backward ($\mu \pm \sigma$)	Total ($\mu \pm \sigma$)
small	23.971 ± 0.019 ms	52.898 ± 0.013 ms	76.869 ± 0.023 ms
medium	62.225 ± 0.308 ms	137.851 ± 0.240 ms	200.076 ± 0.409 ms
large	118.398 ± 0.123 ms	273.568 ± 0.418 ms	391.965 ± 0.399 ms
xl	OOM	OOM	OOM
2.7B	OOM	OOM	OOM

Table 4: Benchmarking Results (sequence length = 1024)

There is little variation across measurements, as seen by the small standard deviations (generally well under 1 ms).

c) Benchmarking results (CUDA, **0 warmup steps**, 10 timed steps, varying sequence length):

Model	Forward ($\mu \pm \sigma$)	Backward ($\mu \pm \sigma$)	Total ($\mu \pm \sigma$)
small	50.044 ± 109.166 ms	24.972 ± 29.328 ms	75.016 ± 138.493 ms
medium	71.781 ± 128.984 ms	40.520 ± 30.044 ms	112.301 ± 159.028 ms
large	84.114 ± 122.191 ms	59.346 ± 37.632 ms	143.460 ± 159.822 ms
xl	101.616 ± 125.648 ms	80.611 ± 38.144 ms	182.228 ± 163.791 ms
2.7B	79.821 ± 122.315 ms	94.699 ± 25.789 ms	174.521 ± 148.104 ms

Table 5: CUDA Benchmarking Results (no warmup, sequence length = 128)

Model	Forward ($\mu \pm \sigma$)	Backward ($\mu \pm \sigma$)	Total ($\mu \pm \sigma$)
small	54.248 ± 122.671 ms	27.946 ± 38.036 ms	82.194 ± 160.707 ms
medium	69.122 ± 121.881 ms	41.867 ± 30.631 ms	110.989 ± 152.511 ms
large	87.265 ± 129.787 ms	73.268 ± 34.331 ms	160.534 ± 164.111 ms
xl	108.917 ± 140.552 ms	116.053 ± 27.636 ms	224.971 ± 168.187 ms
2.7B	85.747 ± 126.751 ms	140.738 ± 26.156 ms	226.485 ± 152.906 ms

Table 6: CUDA Benchmarking Results (no warmup, sequence length = 256)

Model	Forward ($\mu \pm \sigma$)	Backward ($\mu \pm \sigma$)	Total ($\mu \pm \sigma$)
small	54.190 ± 119.762 ms	33.965 ± 35.434 ms	88.154 ± 155.196 ms
medium	71.163 ± 122.827 ms	65.110 ± 26.234 ms	136.273 ± 149.061 ms
large	90.134 ± 126.374 ms	125.395 ± 28.581 ms	215.528 ± 154.953 ms
xl	123.206 ± 130.294 ms	210.528 ± 28.432 ms	333.734 ± 158.725 ms
2.7B	124.921 ± 111.795 ms	244.471 ± 28.501 ms	369.392 ± 140.296 ms

Table 7: CUDA Benchmarking Results (no warmup, sequence length = 512)

Model	Forward ($\mu \pm \sigma$)	Backward ($\mu \pm \sigma$)	Total ($\mu \pm \sigma$)
small	60.625 ± 115.953 ms	61.123 ± 26.407 ms	121.748 ± 142.360 ms
medium	100.052 ± 119.305 ms	146.729 ± 27.650 ms	246.781 ± 146.955 ms
large	155.401 ± 116.441 ms	281.863 ± 25.416 ms	437.264 ± 141.853 ms
xl	OOM	OOM	OOM
2.7B	OOM	OOM	OOM

Table 8: CUDA Benchmarking Results (no warmup, sequence length = 1024)

Without warmup, the standard deviations are much larger. The initial steps incur one-time overheads such as kernel loading and memory allocation for tensors like the parameters and gradients. Once these setup costs are paid and the stead-state throughput is reached, subsequent steps exhibit much less variability.

Benchmarking results (CUDA, **1 warmup step**, 10 timed steps, varying sequence length):

Model	Forward ($\mu \pm \sigma$)	Backward ($\mu \pm \sigma$)	Total ($\mu \pm \sigma$)
small	15.457 ± 0.097 ms	15.801 ± 0.130 ms	31.258 ± 0.212 ms
medium	30.379 ± 0.127 ms	31.172 ± 0.182 ms	61.551 ± 0.298 ms
large	46.200 ± 0.706 ms	47.573 ± 0.253 ms	93.772 ± 0.809 ms
xl	61.188 ± 1.109 ms	69.504 ± 0.060 ms	130.692 ± 1.120 ms
2.7B	41.377 ± 0.642 ms	86.992 ± 0.140 ms	128.370 ± 0.710 ms

Table 9: CUDA Benchmarking Results (1 warmup step, sequence length = 128)

Model	Forward ($\mu \pm \sigma$)	Backward ($\mu \pm \sigma$)	Total ($\mu \pm \sigma$)
small	15.383 ± 0.151 ms	16.121 ± 0.145 ms	31.503 ± 0.286 ms
medium	31.065 ± 0.880 ms	31.977 ± 0.267 ms	63.042 ± 1.110 ms
large	46.086 ± 0.400 ms	62.285 ± 0.047 ms	108.371 ± 0.404 ms
xl	63.262 ± 1.544 ms	108.228 ± 0.181 ms	171.490 ± 1.652 ms
2.7B	45.812 ± 0.222 ms	132.597 ± 0.515 ms	178.410 ± 0.539 ms

Table 10: CUDA Benchmarking Results (1 warmup step, sequence length = 256)

Model	Forward ($\mu \pm \sigma$)	Backward ($\mu \pm \sigma$)	Total ($\mu \pm \sigma$)
small	16.191 ± 0.861 ms	22.714 ± 0.094 ms	38.905 ± 0.945 ms
medium	31.881 ± 1.170 ms	57.215 ± 0.050 ms	89.096 ± 1.204 ms
large	49.574 ± 0.717 ms	116.698 ± 0.148 ms	166.273 ± 0.802 ms
xl	81.754 ± 0.239 ms	201.487 ± 0.385 ms	283.241 ± 0.429 ms
2.7B	89.518 ± 0.112 ms	235.488 ± 0.240 ms	325.006 ± 0.304 ms

Table 11: CUDA Benchmarking Results (1 warmup step, sequence length = 512)

Model	Forward ($\mu \pm \sigma$)	Backward ($\mu \pm \sigma$)	Total ($\mu \pm \sigma$)
small	23.848 ± 0.035 ms	52.626 ± 0.044 ms	76.474 ± 0.054 ms
medium	62.023 ± 0.067 ms	137.619 ± 0.159 ms	199.642 ± 0.175 ms
large	118.616 ± 0.044 ms	273.708 ± 0.310 ms	392.324 ± 0.301 ms
xl	OOM	OOM	OOM
2.7B	OOM	OOM	OOM

Table 12: CUDA Benchmarking Results (1 warmup step, sequence length = 1024)

Even with a single warmup step, the variance is noticeably higher than with five warmup steps. This suggests that one iteration may not be sufficient to complete all initialization processes, such as loading all necessary GPU kernels or stabilizing memory allocation patterns. Subsequent steps might still encounter some initial overheads until a true steady state is reached, which appears to take a few iterations.

Problem (`nsys_profile`): 5 points

a) Mean total forward pass time; all sizes, all sequence lengths:

Model	128	256	512	1024
small	19.018 ms	19.058 ms	19.738 ms	26.163 ms
medium	38.074 ms	38.077 ms	39.196 ms	69.533 ms
large	61.813 ms	58.349 ms	61.301 ms	135.094 ms
xlarge	79.546 ms	76.364 ms	90.199 ms	OOM
2.7b	53.105 ms	62.066 ms	106.359 ms	OOM

Table 13: Forward Pass Total Time (ms) by Model Size and Sequence Length

The timings are quite similar to what was observed with `timeit` (generally within 10%).

b) Kernel that takes the most cumulative time during the forward pass (large model, sequence length = 512):

```
sm90_xmma_gemm_f32f32_tf32f32_f32_tn_n_tilesize128x128x32_warpgroupsize1x1x1_execute_segment
 _k_off_kernel__5x_cublas
```

This is a general matrix-matrix multiplication kernel where the inputs, accumulator, and outputs are all `float32`. The particular kernel is different for different model sizes (different tile sizes, etc.), but it's always a general matrix-matrix multiplication kernel.

Number of instances: 109

This is the same kernel as the one that takes the most cumulative time during the backward pass.

c) In general, the non-matmul kernels that contribute significantly to the forward pass are element-wise tensor operators—pointwise arithmetic, vectorized element-wise computations, reductions, and simple data-movement copies.

A few specific examples (listed in decreasing order of contribution):

```
void at::native::elementwise_kernel<(int)128, (int)2,
  void at::native::gpu_kernel_impl_nocast<
    at::native::BinaryFunctor<float, float, float,
      at::native::binary_internal::MulFunctor<float>>>(
    at::TensorIteratorBase &, const T1 &)::[lambda(int) (instance 1)]>
  (int, T3)
```

```
void at::native::vectorized_elementwise_kernel<(int)4,
  at::native::BinaryFunctor<float, float, float,
    at::native::binary_internal::MulFunctor<float>>,
  std::array<char *, (unsigned long)3>>
  (int, T2, T3)
```

```
void at::native::elementwise_kernel<(int)128, (int)2,
  void at::native::gpu_kernel_impl_nocast<
    at::native::BinaryFunctor<float, float, float,
      at::native::binary_internal::DivFunctor<float>>>(
    at::TensorIteratorBase &, const T1 &)::[lambda(int) (instance 1)]>
  (int, T3)
```

```
void at::native::elementwise_kernel<(int)128, (int)2,
    void at::native::gpu_kernel_impl_nocast<
        at::native::CUDAFunctor_add<float>>(
        at::TensorIteratorBase &, const T1 &)::[lambda(int) (instance 1)]>
    (int, T3)
```

d) With forward-pass inference only, the four GEMM kernels (all the `sm90_xmma_gemm_*` kernels) add up to ~36 % of the work.

During a full training step (forward + backward + AdamW update), those kernels take roughly the same amount of time, but the overall kernel time increases significantly because of the many vectorised element-wise AdamW and reduction kernels (the "vectorized_elementwise_kernel" calls and "reduce_kernel" calls). Consequently, GEMMs now represent only ~19 % of the total. In other words, matrix multiplication's share of runtime is roughly halved, while the element-wise update kernels (mul/add/div/sqrt/fill) and a few extra reductions become the dominant cost.

e) In many cases, the softmax operation takes as long as computing the attention scores and taking the inner products with the value vectors combined (the softmax:matmul ratio within the attention operation varies from ~0.6x to ~1.2x in my experiments).

This is despite a vastly lower FLOP count (on the order of a 10x difference) for the softmax operation, compared to the matmuls.

The softmax operation consumes significantly more wall time per FLOP, yielding poor utilization due to its memory-bound, control-flow-heavy nature.

Problem (`mixed_precision_accumulation`): 1 point

We get the most accurate result (10.0001) with both the accumulator and the summands in `float32` (the first loop). With the accumulator in `float32` and the summands in `float16` (the third and fourth loops), we get close (10.0021). With the accumulator in `float16`, though, we a much less accurate result (9.9531). This is because, as the spacing between representable values in `float16` increases, many of the of the late additions round away and the sum stalls.

Problem (`benchmarking_mixed_precision`): 2 points

a) Model parameters: `float32`
Output of `fc1`: `float16`
Output of `ln`: `float32`
Predicted logits: `float16`
Loss: `float32`
Gradients: `float32`

b) The sensitive parts are the mean and variance reductions to compute the layer normalization statistics, and the reciprocal square root computation. The sensitivity is due to the possibility of overflow when the intermediate values are held in the $\pm$ 65k range of FP16. BF16 matches the dynamic range of FP32. That removes the overflow risk, and makes it possible to run LayerNorm in BF16.

c) Forward pass timings (sequence length = 512):

Model	Mixed (BF16) (ms)	FP32 (ms)
small	22.008	20.015
medium	43.673	56.011
large	67.021	130.569
xl	90.293	249.101
2.7B	84.217	362.488

Table 14: Forward Pass Timings (sequence length = 512)

Backward pass timings (sequence length = 512):

Model	Mixed (BF16) (ms)	FP32 (ms)
small	27.576	41.988
medium	54.394	114.903
large	79.397	259.485
xl	137.134	503.508
2.7B	149.459	716.287

Table 15: Backward Pass Timings (sequence length = 512)

BF16 is fatster than FP16 for all model sizes. At smaller sizes, the difference in forward pass timings is small, though the difference in backward pass timings is still significant. As the size increases, the difference grows dramatically. This makes intuitive sense, given that matmuls come to dominate the wall clock time when running a forward pass in FP32, and those are the operations for which we get the most benefit from running using mixed precision.

Problem (`memory_profiling`): 4 points

a)

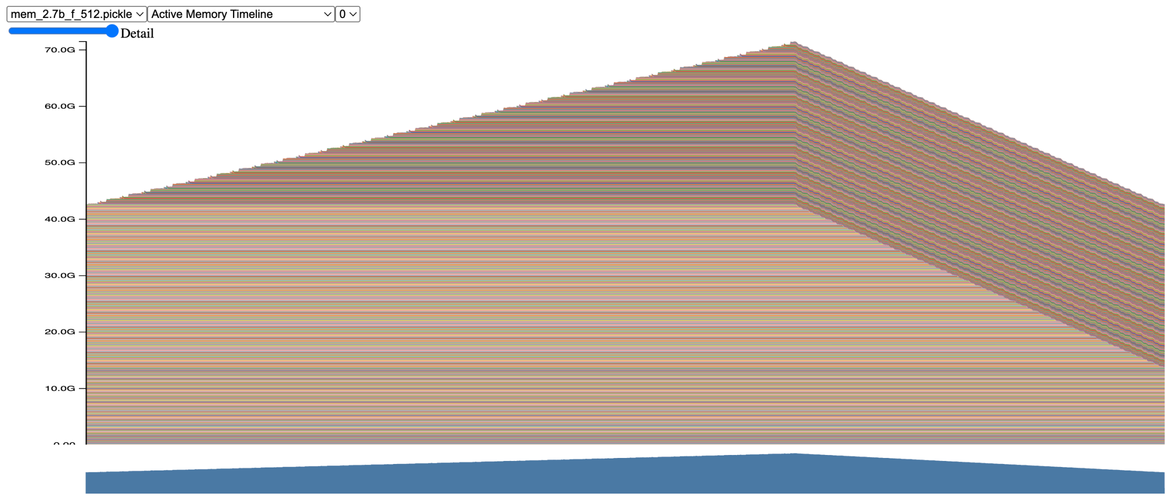

Figure 1: Memory Profile (FP32, 2.7B, forward pass only, 512 sequence length)

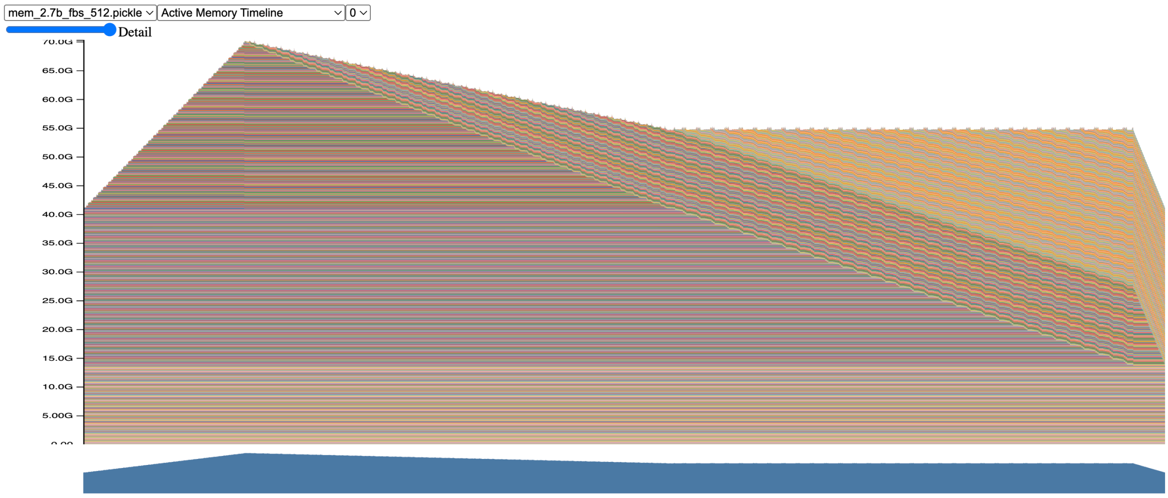

Figure 2: Memory Profile (FP32, 2.7B, full train step, 512 sequence length)

In both memory timelines, active memory rises to a peak of about 70GB during the forward pass, when memory is being allocated for the activations. In the forward-only run, the timeline is almost a perfect triangle, as memory declines sharply back to the weight-only baseline at the end of the forwrd pass. In the full training step, the descent is shallower — during the backward pass, activations are iteratively freed after gradients have been materialised (for which memory is allocated), so memory hovers in the mid-50GB range. It plateaus there while the optimizer updates the parameters, and finally drops to a steady state that is larger than the initial baseline due to the optimizer state. The peak identifies the end of the forward pass, the long sloping shoulder is the backward pass, and the flat tail is the optimizer step.

b) Peak memory usage by sequence length for 2.7b model:

Sequence Length	Forward Pass (GB)	Full Training Step (GB)
128	23.1	51.1
256	35.5	51.2
512	65.1	65.4

Table 16: Peak Memory Usage by Sequence Length for 2.7B Model

c) At shorter sequence lengths, mixed precision does not seem to significantly reduce memory usage (I see very similar numbers for peak memory usage at sequence lengths of 128 and 256, whether running the forward pass only or the full training step). At a sequence length of 512, however, the memory usage for a forward pass dropped from ~66GB without mixed precision to ~54GB with it. The reduction was less dramatic for the full training step, in which memory usage dropped from ~66GB to ~62GB.

d) Answer (sequence length = 512, batch size = 4): 20MB

Derivation:

$$
\begin{aligned}
\text{elements} &= B \times L \times d_{\text{model}} = 4 \times 512 \times 2560 = 5,242,880 \\
\text{bytes} &= \text{elements} \times 4 = 20,971,520 \\
\text{MB} &= \frac{\text{bytes}}{1024^2} = 20
\end{aligned}
\tag{1}
$$

e) At a low detail level, I consistently see allocations of 128MB. These appear to be the attention-probability matrices in the attention blocks in each layer, which would be of size $4 * \frac{4*32*512*512}{1024^2} = 128\ \text{MB}$.

1.2 Optimizing Attention with FlashAttention-2

Problem (`pytorch_attention`): 2 points

a) Timings and memory usage (just before backward pass) of `scaled_dot_product_attention` for different d_{model} and sequence lengths:

Seq. Len	Forward (ms)	Backward (ms)	Memory Usage (MB)
256	0.09	0.45	70
1024	0.22	0.85	103
4096	2.54	8.22	613
8192	9.93	31.72	2232
16384	38.91	125.12	8692

Table 17: Timings for d_model = 16

Seq. Len	Forward (ms)	Backward (ms)	Memory Usage (MB)
256	0.10	0.46	70
1024	0.23	0.84	105
4096	2.61	8.38	621
8192	10.25	32.34	2249
16384	40.16	127.58	8726

Table 18: Timings for d_model = 32

Seq. Len	Forward (ms)	Backward (ms)	Memory Usage (MB)
256	0.10	0.47	71
1024	0.25	0.88	109
4096	2.89	8.94	638
8192	11.46	34.76	2282
16384	45.34	137.91	8793

Table 19: Timings for d_model = 64

Seq. Len	Forward (ms)	Backward (ms)	Memory Usage (MB)
256	0.09	0.46	73
1024	0.29	0.96	118
4096	3.48	10.13	671
8192	13.71	39.28	2350
16384	54.20	155.69	8927

Table 20: Timings for d_model = 128

I did not hit out of memory errors for any of these configurations.

Memory usage of `scaled_dot_product_attention` just before the backward pass, assuming the full $L \times L$ attention matrix is stored, along with inputs Q, K, V, and the output O (Batch size $B = 8$, sequence length $L = 16384$, head dimension $d = 128$, data type `float32`):

$$\begin{aligned}
\mathrm{mem}_{Q,K,V,O} &= 4 \times B \times L \times d \times 4 \\
\mathrm{mem}_{Q,K,V,O} &= 4 \times 8 \times 16384 \times 128 \times 4 = 268,435,456 \quad \text{bytes} \\
\mathrm{mem}_P &= B \times L^2 \times 4 \\
\mathrm{mem}_P &= 8 \times 16384^2 \times 4 = 8,589,934,592 \quad \text{bytes} \\
\mathrm{mem}_{\text{total}} &= \mathrm{mem}_{Q,K,V,O} + \mathrm{mem}_P \\
\mathrm{mem}_{\text{total}} &= 268,435,456 + 8,589,934,592 = 8,858,370,048 \quad \text{bytes} \\
\mathrm{MB} &= \frac{\mathrm{mem}_{\text{total}}}{1024^2} \approx 8448 \quad \mathrm{MB}
\end{aligned} \tag{2}$$

The memory saved for backward changes with the square of the sequence length, as expected.

To eliminate this cost, we need to avoid materializing the full attention probability matrix P, which uses $O(L^2)$ memory. We can do this by tiling/blocking the attention operation. We iterate through blocks of the key (K) and value (V) matrices. For each block of queries (Q), we compute partial attention scores against a block of K. We use these to update online softmax statistics (a running maximum and normalizer) and immediately compute a weighted sum of the corresponding V block. The weighted sum is accumulated into the final output block corresponding to Q. The backward pass then recomputes necessary attention components on-the-fly instead of relying on a stored P from the forward pass.

1.3 Benchmarking JIT-Compiled Attention

Problem (`torch_compile`): 2 points

a) Timings and memory usage for `scaled_dot_product_attention` with and without `torch.compile` ($d_{\text{model}} = 128$, all times in milliseconds):

Seq. Len	Fwd	Fwd Comp	Bwd	Bwd Comp	Mem	Mem Comp
256	0.09	0.09	0.46	0.50	73	73
1024	0.29	0.21	0.96	0.71	118	118
4096	3.48	2.31	10.13	5.73	671	672
8192	13.71	8.97	39.28	21.89	2350	2350
16384	54.20	54.42	155.69	156.22	8927	8927

Table 21: Attention performance with/without JIT compilation (d_model=128)

We see that JIT compilation offers some benefit within a relatively narrow range of sequence lengths. Outside of that range, neither time nor memory usage is significantly affected.

b) Timings for the medium model (batch size = 4, sequence length = 1024) with and without `torch.compile` (all times in milliseconds):

Compiled?	Fwd (fwd-only)	Fwd (full)	Bwd	Opt	Total
No	125.94	126.10	256.67	24.07	406.84
Yes	90.48	90.73	185.06	23.65	299.44

Table 22: Full Model Timings (Medium, B=4, L=1024) with/without JIT Compilation

We see significant improvements in latency for both the forward and backward passes, and no significant change in the optimizer step. In total, JIT compilation shaves ~25% off the total training step time with this configuration.

Problem (`flash_forward`): 15 points

a) See `flash_torch.py`

b) See `flash_triton.py`

c) See updated `flash_triton.py`

Problem (`flash_backward`): 5 points

See `flash_triton.py`

Problem (`flash_benchmarking`): 5 points

a) Comparison of PyTorch SDPA and FlashAttention-2 (Triton) using `bfloat16` (all times in milliseconds):

Seq	D	Py Fwd	Py Bwd	Py Tot	Flash Fwd	Flash Bwd	Flash Tot	× Fwd	× Bwd	× Tot
128	16	0.048	0.173	0.367	0.007	0.108	0.238	6.5×	1.6×	1.5×
128	32	0.048	0.225	0.447	0.008	0.146	0.278	6.0×	1.5×	1.6×
128	64	0.048	0.178	0.361	0.008	0.105	0.217	5.9×	1.7×	1.7×
128	128	0.042	0.168	0.346	0.011	0.092	0.204	3.8×	1.8×	1.7×
256	16	0.049	0.173	0.357	0.009	0.11	0.226	5.6×	1.6×	1.6×
256	32	0.047	0.172	0.359	0.01	0.104	0.216	4.7×	1.6×	1.7×
256	64	0.044	0.167	0.348	0.01	0.098	0.209	4.5×	1.7×	1.7×
256	128	0.045	0.168	0.342	0.015	0.099	0.21	2.9×	1.7×	1.6×
512	16	0.048	0.171	0.349	0.011	0.106	0.22	4.3×	1.6×	1.6×
512	32	0.049	0.170	0.349	0.014	0.107	0.218	3.5×	1.6×	1.6×
512	64	0.05	0.171	0.347	0.014	0.105	0.217	3.6×	1.6×	1.6×
512	128	0.05	0.173	0.350	0.023	0.107	0.219	2.1×	1.6×	1.6×
1024	16	0.062	0.173	0.352	0.017	0.112	0.22	3.6×	1.6×	1.6×
1024	32	0.062	0.174	0.354	0.022	0.109	0.22	2.9×	1.6×	1.6×
1024	64	0.064	0.174	0.354	0.021	0.109	0.223	3.1×	1.6×	1.6×
1024	128	0.062	0.173	0.357	0.04	0.108	0.222	1.6×	1.6×	1.6×
2048	16	0.105	0.202	0.355	0.028	0.109	0.219	3.8×	1.9×	1.6×
2048	32	0.108	0.203	0.354	0.037	0.108	0.22	2.9×	1.9×	1.6×
2048	64	0.107	0.203	0.363	0.035	0.108	0.223	3.0×	1.9×	1.6×
2048	128	0.11	0.209	0.362	0.072	0.11	0.226	1.5×	1.9×	1.6×
4096	16	0.292	0.604	0.904	0.05	0.213	0.277	5.9×	2.8×	3.3×
4096	32	0.293	0.599	0.901	0.068	0.208	0.291	4.3×	2.9×	3.1×
4096	64	0.295	0.600	0.901	0.064	0.21	0.287	4.6×	2.9×	3.1×
4096	128	0.299	0.615	0.923	0.138	0.22	0.372	2.2×	2.8×	2.5×
8192	16	1.057	2.089	3.144	0.094	0.685	0.799	11.3×	3.1×	3.9×
8192	32	1.054	2.096	3.152	0.131	0.685	0.83	8.0×	3.1×	3.8×
8192	64	1.041	2.090	3.147	0.123	0.677	0.82	8.5×	3.1×	3.8×
8192	128	1.068	2.111	3.187	0.257	0.698	0.997	4.2×	3.0×	3.2×
16384	16	3.865	7.878	11.745	0.237	2.567	2.815	16.3×	3.1×	4.2×
16384	32	3.831	7.902	11.766	0.316	2.587	2.905	12.1×	3.1×	4.1×
16384	64	3.836	7.914	11.782	0.308	2.607	2.907	12.4×	3.0×	4.1×
16384	128	3.87	7.939	11.852	0.567	2.711	3.322	6.8×	2.9×	3.6×
32768	16	15.034	31.009	46.081	0.829	10.075	10.96	18.1×	3.1×	4.2×
32768	32	14.971	31.125	46.180	0.965	10.215	11.141	15.5×	3.0×	4.1×

Seq	D	Py Fwd	Py Bwd	Py Tot	Flash Fwd	Flash Bwd	Flash Tot	× Fwd	× Bwd	× Tot
32768	64	15.019	31.069	46.147	1.048	10.423	11.341	14.3×	3.0×	4.1×
32768	128	15.054	31.113	46.263	2.241	10.29	12.553	6.7×	3.0×	3.7×
65536	16	59.954	124.306	184.130	4.463	67.858	72.805	22.1×	N/A	N/A
65536	32	59.982	124.183	184.058	6.553	73.431	79.919	15.4×	N/A	N/A
65536	64	60.204	124.908	185.229	4.038	41.751	44.917	14.9×	N/A	N/A
65536	128	60.603	126.154	186.594	8.884	42.45	50.692	6.8×	N/A	N/A

Table 23: Performance Comparison (BF16): PyTorch vs. FlashAttention-2 (Triton)

Comparison of PyTorch SDPA and FlashAttention-2 (Triton) using `float32` (all times in milliseconds):

Seq	D	Py Fwd	Py Bwd	Py Tot	Flash Fwd	Flash Bwd	Flash Tot	× Fwd	× Bwd	× Tot
128	16	0.052	0.181	0.362	0.009	0.117	0.23	6.1×	1.5×	1.6×
128	32	0.055	0.180	0.361	0.01	0.116	0.228	5.3×	1.6×	1.6×
128	64	0.055	0.182	0.361	0.018	0.117	0.232	3.1×	1.6×	1.6×
128	128	0.047	0.181	0.372	0.029	0.107	0.223	1.6×	1.7×	1.7×
256	16	0.044	0.181	0.372	0.011	0.119	0.232	4.0×	1.5×	1.6×
256	32	0.047	0.184	0.376	0.014	0.12	0.236	3.3×	1.5×	1.6×
256	64	0.048	0.182	0.376	0.027	0.12	0.235	1.8×	1.5×	1.6×
256	128	0.05	0.182	0.375	0.047	0.12	0.23	1.1×	1.5×	1.6×
512	16	0.05	0.179	0.368	0.016	0.118	0.232	3.0×	1.5×	1.6×
512	32	0.052	0.179	0.368	0.022	0.117	0.23	2.3×	1.5×	1.6×
512	64	0.053	0.178	0.369	0.047	0.118	0.229	1.1×	1.5×	1.6×
512	128	0.055	0.179	0.369	0.085	0.118	0.233	0.6×	1.5×	1.6×
1024	16	0.067	0.184	0.378	0.026	0.122	0.232	2.6×	1.5×	1.6×
1024	32	0.071	0.181	0.376	0.038	0.122	0.236	1.9×	1.5×	1.6×
1024	64	0.069	0.177	0.370	0.085	0.117	0.23	0.8×	1.5×	1.6×
1024	128	0.079	0.182	0.380	0.16	0.122	0.266	0.5×	1.5×	1.4×
2048	16	0.124	0.285	0.422	0.046	0.123	0.236	2.7×	2.3×	1.8×
2048	32	0.132	0.287	0.432	0.069	0.136	0.243	1.9×	2.1×	1.8×
2048	64	0.145	0.308	0.467	0.165	0.157	0.338	0.9×	2.0×	1.4×
2048	128	0.16	0.351	0.529	0.313	0.216	0.539	0.5×	1.6×	1.0×
4096	16	0.445	0.958	1.411	0.087	0.329	0.438	5.1×	2.9×	3.2×
4096	32	0.457	0.976	1.443	0.135	0.361	0.513	3.4×	2.7×	2.8×
4096	64	0.497	1.031	1.544	0.325	0.444	0.795	1.5×	2.3×	1.9×

Seq	D	Py Fwd	Py Bwd	Py Tot	Flash Fwd	Flash Bwd	Flash Tot	× Fwd	× Bwd	× Tot
4096	128	0.583	1.197	1.801	0.621	0.663	1.313	0.9×	1.8×	1.4×
8192	16	1.597	3.379	4.980	0.166	1.105	1.307	9.6×	3.1×	3.8×
8192	32	1.632	3.428	5.064	0.259	1.157	1.476	6.3×	3.0×	3.4×
8192	64	1.78	3.687	5.481	0.64	1.527	2.246	2.8×	2.4×	2.4×
8192	128	2.119	4.310	6.449	1.242	2.339	3.621	1.7×	1.8×	1.8×
16384	16	5.999	12.898	18.917	0.33	4.018	4.426	18.2×	3.2×	4.3×
16384	32	6.153	13.083	19.250	0.489	4.29	4.915	12.6×	3.0×	3.9×
16384	64	6.907	14.446	21.373	1.247	6.011	7.383	5.5×	2.4×	2.9×
16384	128	8.109	16.732	24.867	4.336	9.364	13.592	1.9×	1.8×	1.8×
32768	16	23.447	50.809	74.260	1.138	15.695	16.855	20.6×	3.2×	4.4×
32768	32	24.11	51.529	75.673	1.965	16.965	19.087	12.3×	3.0×	4.0×
32768	64	26.787	56.182	83.030	5.108	23.464	28.437	5.2×	2.4×	2.9×
32768	128	32.432	66.594	99.056	19.407	37.15	55.996	1.7×	1.8×	1.8×
65536	16	98.512	OOM	OOM	4.463	67.858	72.805	22.1×	N/A	N/A
65536	32	100.8	OOM	OOM	6.553	73.431	79.919	15.4×	N/A	N/A
65536	64	107.266	OOM	OOM	22.101	95.416	116.402	4.9×	N/A	N/A
65536	128	125.682	OOM	OOM	76.855	142.404	214.972	1.6×	N/A	N/A

Table 24: Performance Comparison (FP32): PyTorch vs. FlashAttention-2 (Triton)

2.1 Single-Node Distributed Communication in PyTorch

Problem (`distributed_communication_single_node`): 5 points

See `cs336_systems/benchmark_all_reduce.py`

Single-node all-reduce latency (mean, ms):

Backend	Device	Tensor Size (MB)	2 Procs	4 Procs	6 Procs
NCCL	GPU	1	0.04	0.05	0.05
NCCL	GPU	10	0.08	0.10	0.12
NCCL	GPU	100	0.40	0.51	0.50
NCCL	GPU	1000	3.14	4.43	4.15
Gloo	CPU	1	0.57	0.76	1.33
Gloo	CPU	10	2.84	15.07	6.11
Gloo	CPU	100	42.70	58.17	64.19
Gloo	CPU	1000	335.85	968.32	1037.66

Table 25: Single-node all-reduce latency (mean, ms)

Commentary:

NCCL on GPUs is vastly faster than Gloo on CPUs (consistently 10-100x). With both backends, latency grows roughly linearly with tensor size. With NCCL on GPUs, latency grows only very mildly (and not even in all cases) with more ranks. With Gloo on CPUs, latency seems to grow more reliably and more quickly with world size.

2.2 A Naïve DDP Implementation

Problem (`naive_ddp`): 5 points

See `cs336_systems/naive_ddp.py`

Problem (`naive_ddp_benchmarking`): 3 points

See `cs336_systems/ddp_benchmarking.py`

In the benchmarking script, I collect measurements for global batch sizes [2, 4, 8, 16, 32] and a sequence length of 128, measuring total time for a single training step and the time spent in communication in each case. As expected, the results show that communication time for gradients is independent of batch size, total time is roughly linear in batch size.

The script includes 5 warmup steps and 5 measurement steps on each rank. Each device synchronizes before each measurement. The results are collected in a list on each rank. The lists are gathered and flattened on rank 0, which averages the results and reports the mean and standard deviation of the measurements.

All training was done in float32 without JIT-compilation. With mixed precision training and JIT compilation, I would expect compute time to come down significantly, exacerbating the communication overhead.

The results are as follows (all times in milliseconds):

Batch Size	Total Time ($\mu \pm \sigma$)	Comm Time ($\mu \pm \sigma$)	Comm Prop.
2	286.09 ± 1.78 ms	43.03 ± 1.31 ms	15.04%
4	309.61 ± 1.85 ms	42.44 ± 1.03 ms	13.71%
8	366.11 ± 1.89 ms	42.73 ± 1.84 ms	11.67%
16	531.74 ± 0.60 ms	42.76 ± 0.97 ms	8.04%
32	827.26 ± 0.56 ms	42.57 ± 0.59 ms	5.15%

Table 26: Naive DDP Benchmark Results (XL Model, 2 GPUs, Seq Len=128)

2.3 Improving Upon the Minimal DDP Implementation

Problem (`minimal_ddp_flat_benchmarking`): 2 points

Results when training the XL model in float32 on 2 GPUs with a sequence length of 128 and batch size of 16:

Single batched all-reduce call:

Avg total time / step : 523.03 ± 0.25 ms
Avg communication time / step : 36.01 ± 0.62 ms
Communication proportion : 6.89%

Individually communicating gradients:

Avg total time / step : 531.74 ± 0.60 ms
Avg comm time / step : 42.76 ± 0.97 ms
Comm proportion : 8.04%

As expected, the single batched all-reduce call is faster than individually communicating gradients, and I would expect the benefit to increase with larger world sizes. However, there is still significant communication overhead, which cannot easily be mitigated without overlapping communication with computation.

Problem (`ddp_overlap_individual_parameters`): 5 points

See `cs336_systems/ddp_overlap_individual.py`

Problem (`ddp_overlap_individual_parameters_benchmarking`): 1 point

a) Total time per training iteration (batch size 16, seq len 128, mean over 5 iterations after 5 warmup iterations, all times in milliseconds):

DDP Implementation	Total Time ($\mu \pm \sigma$)
Naive DDP	531.74 ± 0.60 ms
Flat DDP	523.03 ± 0.25 ms
Overlap-individual DDP	509.30 ± 0.64 ms

Table 27: Performance comparison of DDP implementations

The implementation that overlaps communication with computation improves significantly over both the naive (one all-reduce per parameter) and flat (one all-reduce for all parameters) implementations.

The naive implementation used 42.76ms for communication, on average. We can then estimate that the time per training step not if communication overhead could be completely eliminated would be $531.74 - 42.76 \approx 489$ ms. The overlap-individual implementation used 509.30ms per training step, suggesting ≈ 20 ms of communication overhead — an improvement of ~53% over naive DDP, and of ~44% over flat DDP.

b) Trace of naive DDP (no overlapping):

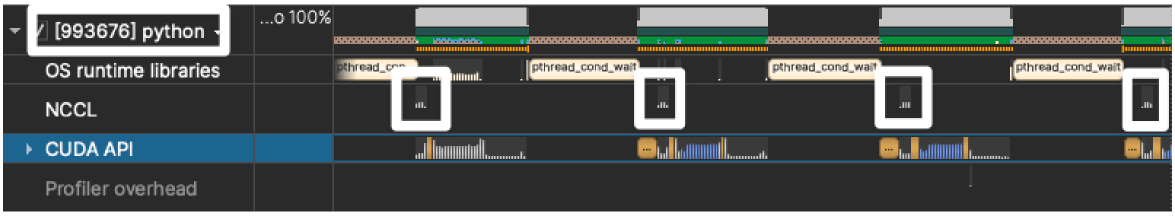

Figure 3: Naive DDP trace

Trace of "overlap-individual" DDP:

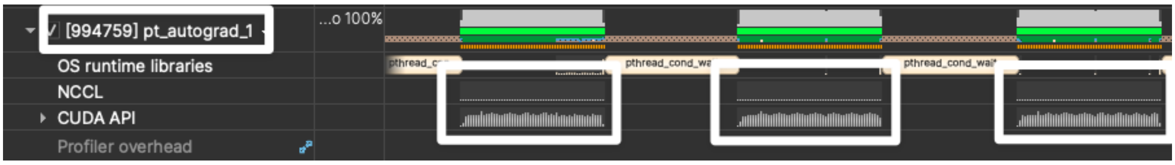

Figure 4: Overlap-individual DDP trace

In the first trace (naive DDP), we see all NCCL launches coming from the main Python thread (where we iterate over parameters and call `dist.all_reduce` on each, after `loss.backward()`). We also see clusters of NCCL tiles separated by long gaps (while the main thread is blocked in `pthread_cond_wait` during backward).

In the second trace (overlap-individual DDP), we see the NCCL row under a thread starting with `pt_autograd_`, which belongs to Python's internal autograd engine and is active only while the backward graph is still being executed. We also see a more spread out distribution of NCCL tiles, as communication is overlapped with computation during the backward pass.

Problem (`ddp_overlap_bucketed`): 8 points

See `cs336_systems/ddp_overlap_bucketed.py`

Problem (`ddp_bucketed_benchmarking`): 3 points

a) Mean total time per training step for the XL model (batch size 16, seq len 128) with varying bucket sizes:

Bucket Size (MB)	Total Time ($\mu \pm \sigma$)
1	510.98 ± 0.33 ms
10	509.02 ± 0.19 ms
100	516.44 ± 0.29 ms
1000	512.88 ± 0.34 ms

Table 28: DDP Overlap Bucketed Performance (XL Model, 2 GPUs, B=16, L=128)

On two H100s, the all-reduce of an entire 8 GB gradient tensor finishes in a small fraction of the time taken by the backward pass of the XL model. Because we build buckets in reverse parameter order, the gradients for the deepest layers are transmitted first and overlap with compute for shallower layers. The only communication that shows up significantly in step time is the "tail" of the final bucket. This is why bucket size makes very little difference in this experimental setup, except that in this specific setup the 100 MB bucket size happens to leave a large bucket outstanding when the backward pass computation completes.

If we increased the world-size or moved to a slower interconnect (e.g. PCIe), the hidden-under-compute advantage would diminish and the expected trend of “larger bucket ⟶ fewer launches ⟶ faster” would appear.

b) Let:

s be the total bytes of parameters (i.e. total bytes of gradients to move at each step)
w be the all-redue algorithm bandwidth
o be the overhead associated with each communication call
n_b be the number of buckets

After assuming that the time to compute gradients for a bucket is equal to the time to commuicate gradients for the bucket, all reductions except the very last one will overlap with compute. What remains is:

$$T_{\text{over}} = \frac{s/n_b}{w} + n_b o = \frac{s}{n_b w} + n_b o \tag{3}$$

This is the data transfer time for the final bucket, plus the overhead for all the communication calls.

To minimize T_{over}:

$$\frac{dT_{\text{over}}}{dn_b} = -\frac{s}{n_b^2 w} + o = 0 \Rightarrow n_b^* = \sqrt{\frac{s}{ow}} \tag{4}$$

Then, with equal-sizes buckets, the optimal bucket size is:

$$B^* = \frac{s}{n_b^*} = \sqrt{swo} \quad \text{(bytes)} \tag{5}$$

2.4 4D Parallelism

Problem (`communication_accounting`): 10 points

a) Calculations:

Let $d_{\text{model}} = 16384$, $d_{\text{ff}} = 53248$, and $N_{\text{blocks}} = 126$. Assume FP32 (4 bytes) for static state (weights, gradients, optimizer state) and BF16 (2 bytes) for activations. Let B be the batch size and L be the sequence length.

Parameters per block:

$$\begin{aligned} P_{\text{block}} &= d_{\text{model}} \times d_{\text{ff}} + d_{\text{ff}} \times d_{\text{model}} \\ &= 16384 \times 53248 + 53248 \times 16384 = 1,744,830,464 \end{aligned} \tag{6}$$

Total parameters:

$$P_{\text{total}} = P_{\text{block}} \times N_{\text{blocks}} = 1,744,830,464 \times 126 = 219,848,638,464 \tag{7}$$

Static Memory (FP32): Weights ($P_{\text{total}} \times 4$), gradients ($P_{\text{total}} \times 4$), and optimizer state ($P_{\text{total}} \times 8$ for AdamW).

$$\begin{aligned} M_{\text{static}} &= P_{\text{total}} \times (4 + 4 + 8) = 219,848,638,464 \times 16 \\ &= 3,517,578,215,424 \quad \text{bytes} \\ &= 3,517,578,215, \frac{424}{1024^3} \quad \text{GB} \approx 3276 \quad \text{GB} \end{aligned} \tag{8}$$

Activation Memory (BF16):

$$\begin{aligned} \text{Elements/sample} &= N_{\text{blocks}} \times (d_{\text{model}} + d_{\text{ff}}) \\ &= 126 \times (16384 + 53248) = 8,773,632 \\ \text{Bytes/sample} &= \text{Elements/sample} \times 2 = 17,547,264 \quad \text{bytes} \\ M_{\text{act}}(B, L) &= \text{Bytes/sample} \times B \times L \\ &= \frac{17,547,264 \times B \times L}{1024^3} \quad \text{GB} \end{aligned} \tag{9}$$

Total Memory Required:

$$M_{\text{total}}(B, L) = M_{\text{static}} + M_{\text{act}}(B, L) = \left(3276 + \frac{17,547,264 \times B \times L}{1024^3}\right) \quad \text{GB} \tag{10}$$

Number of GPUs: Assuming 80 GB per H100 GPU.

$$N_{\text{GPUs}(B,L)} = \left\lceil M_{\text{total}} \frac{B, L}{80} \right\rceil = \left\lceil \frac{3276 + \frac{17,547,264 \times B \times L}{1024^3}}{80} \right\rceil \tag{11}$$

Instantiation for $B = 128$, $L = 1024$:

$$
\begin{aligned}
M_{\text{act}}(128, 1024) &= \frac{17,547,264 \times 128 \times 1024}{1024^3} \quad \text{GB} \\
&= 2,300,034,940, \frac{928}{1024^3} \quad \text{GB} \approx 2142 \quad \text{GB} \\
M_{\text{total}}(128, 1024) &= 3276 \quad \text{GB} + 2142 \quad \text{GB} = 5418 \quad \text{GB} \\
N_{\text{GPUs}(128,1024)} &= \left\lceil \frac{5418}{80} \right\rceil = \lceil 67.725 \rceil = 68
\end{aligned}
\tag{12}
$$

Storing the static state (weights, gradients, optimizer states) in FP32 requires 3276 GB. For a batch size $B = 128$ and sequence length $L = 1024$, the BF16 activations saved for backward require an additional 2142 GB. This totals 5418 GB, necessitating 68 H100 80GB GPUs.

b) Calculations:

Assume Fully Sharded Data Parallel (FSDP) shards the master weights, gradients, and optimizer states ($M_{\text{static}} = 3276$ GB) across N_{fsdp} devices. Assume required activation memory is halved to $M_{(\text{act})'}(B, L) = M_{\text{act}} \frac{B,L}{2}$, and this is also effectively sharded across N_{fsdp} devices.

Total Sharded Memory:

$$
\begin{aligned}
M_{\text{FSDP}}(B, L) &= M_{\text{static}} + M_{(\text{act})'}(B, L) \\
&= M_{\text{static}} + M_{\text{act}} \frac{B, L}{2} \\
&= \left(3276 + \frac{17,547,264 \times B \times L}{2 \times 1024^3} \right) \quad \text{GB}
\end{aligned}
\tag{13}
$$

Memory per device: Let M_{target} be the target memory per GPU (set to 95 GB).

$$
M_{\text{device}}(B, L, N_{\text{fsdp}}) = M_{\text{FSDP}} \frac{B, L}{N_{\text{fsdp}}}
\tag{14}
$$

Required FSDP size (N_{fsdp}) for $M_{\text{device}} \leq M_{\text{target}}$:

$$
\begin{aligned}
M_{\text{FSDP}} \frac{B, L}{N_{\text{fsdp}}} &\leq M_{\text{target}} \\
N_{\text{fsdp}}(B, L) &\geq M_{\text{FSDP}} \frac{B, L}{M_{\text{target}}} \\
N_{\text{fsdp}}(B, L) &= \left\lceil M_{\text{FSDP}} \frac{B, L}{M_{\text{target}}} \right\rceil
\end{aligned}
\tag{15}
$$

Instantiation for $B = 128$, $L = 1024$, $M_{\text{target}} = 95$ GB:

$$
\begin{aligned}
M_{(\mathrm{act})'}(128, 1024) &= M_{\mathrm{act}} \frac{128, 1024}{2} = \frac{2142}{2} = 1071 \quad \mathrm{GB} \\
M_{\mathrm{FSDP}}(128, 1024) &= 3276 \quad \mathrm{GB} + 1071 \quad \mathrm{GB} = 4347 \quad \mathrm{GB} \\
N_{\mathrm{fsdp}}(128, 1024) &= \left\lceil \frac{4347}{95} \right\rceil = \lceil 45.757... \rceil = 46
\end{aligned}
\tag{16}
$$

Under FSDP with activation checkpointing, sharding the static memory (3276 GB) and halved activation memory (1071 GB for B=128, L=1024) across devices requires $N_{\mathrm{fsdp}} \geq 46$ devices to keep the memory per device ($4347 \frac{\mathrm{GB}}{N_{\mathrm{fsdp}}}$) at or below 95 GB.

c) Calculations:

QUANTITY	VALUE	NOTE
Total chips	$N = XY = 16 \times 4 = 64$	given
Mesh factors	$M_X = 2, M_Y = 1$	given
FFN width	$F = d_{\mathrm{ff}} = 53248$	given
Compute rate	$C = 4.6 \times 10^{14}$ FLOP s^-1	given
Bandwidth	$W_{\mathrm{ici}} = 1.8 \times 10^{11}$ byte s^-1	given
ICI arithmetic intensity	$\alpha = \frac{C}{W_{\mathrm{ici}}} = \frac{4.6 \times 10^{14}}{1.8 \times 10^{11}} \approx 2.555 \times 10^3$	TPU Scaling Book

Table 29: Compute-Bound Calculation Inputs (TPU v5p)

According to the section on mixed FDSP + TP from the TPU Scaling Book [1], the compute-bound inequality for mixed FSDP + TP is

$$
\frac{B}{N} > \frac{4\alpha^2}{M_X M_Y F}
\tag{17}
$$

.

Computing the right-hand side:

$$
\frac{4\alpha^2}{M_X M_Y F} = \frac{4\left(2.555 \times 10^3\right)^2}{2 \times 1 \times 53248} \approx 2.46 \times 10^2 \quad \text{tokens}
\tag{18}
$$

Hence:

- Minimum per-device batch size:

$$
\left(\frac{B}{N}\right)_{\min} \approx 2.46 \times 10^2
\tag{19}
$$

  tokens.
- Corresponding global batch size:

$$
B_{\min} = \left(\frac{B}{N}\right)_{\min} \times N \approx 2.46 \times 10^2 \times 64 \approx 1.58 \times 10^4
\tag{20}
$$

  .

One-sentence answer: For the specified $X = 16, Y = 4$ layout on TPU v5p, the model stays compute-bound only when each chip processes $\approx 2.46 \times 10^2$ tokens (global batch $\approx 1.58 \times 10^4$), or more per forward pass.

d) Per the Ultrascale Playbook [2], our global batch size is $\text{gbs} = \text{mbs} \times \text{grad\_add} \times \text{dp}$, where mbs is the micro batch size, grad_add is the gradient accumulation steps, and dp is the data parallelism factor.

   We can then reduce our global batch size by (i) reducing our micro-batch size (ii) taking fewer gradient accumulation steps (iii) reducing our data parallelism factor.

   To reduce our global batch size while maximizing throughput, we don't want to simply reduce our micro-batch size with no other changes. Instead, we may want to reduce our micro-batch size and, for example, reduce activation checkpointing, using the memory we get from the smaller batch size to store more activations and get higher throughput than we otherwise would at the smaller micro-batch size.

   We can also reduce our data parallelism factor in favor of other forms of parallelism, such as pipeline parallelism, in which we partition the model along the depth dimension (i.e. different devices handle different layers). Then, to prevent idle time caused by the dependency of each device on the output of the previous layer (computed on a different device), we can use an algorithm like DualPipe [3], introduced in the DeepSeek-V3 technical report, which both overlaps forward and backward communication-computation phases and reduces pipeline bubbles.

3 Optimizer State Sharding

Problem (`optimizer_state_sharding`): 10 points

See `cs336_systems/optimizer_state_sharding.py`

Problem (`optimizer_state_sharding_accounting`): 5 points

a) **Avg. peak memory usage per device** (MB) for the XL model on 2 GPUs with a batch size of 16 and sequence length of 128:

Optimizer	After Init	Before Step	After Step
Sharded	7804.65	26312.84	23723.84
Unsharded	7804.65	34104.87	31564.80

Table 30: Optimizer State Sharding Memory Usage (XL, 2 GPUs, B=16, Seq Len=128)

The results line up with expectations.

This is a ~1.998B parameter model, so the model weights alone require ~7.62GB of memory. In all cases, the memory usage after model initialization lines up closely with the expected memory usage for the model weights.

The gradients then also require ~7.62GB of memory, and the optimizer states require ~15.24GB of memory in total. The total static memory requirement is then roughly $7.62 + 7.62 + 15.24 = 30.48$ GB. This aligns closely with the "after step" memory usage recorded in the unsharded case.

In the sharded case, we'd expect half of the optimizer states to be stored on each device (on average), so the memory per device is roughly $7.62 + 7.62 + \left(\frac{15.24}{2}\right) = 22.86$ GB. This aligns closely with the "after step" memory usage recorded in the sharded case.

As expected, the difference in peak memory usage from "before step" to "after step" (~2.6GB) does not depend on the optimizer state sharding scheme, because we don't handle activations any differently in the sharded case from the unsharded case.

b) **Mean total time per training step** for the XL model on 2 GPUs, using naive DDP (one all-reduce per parameter tensor) and a sequence length of 128:

Batch Size	Sharded (μ, ms)	Unsharded (μ, ms)
16	510.53	527.62
32	804.91	822.41

Table 31: Optimizer State Sharding Performance (XL, 2 GPUs, Naive DDP, Seq Len=128)

Optimizer state sharding provides a modest speedup over the unsharded implementation with this setup. As expected, the gain from sharding is more pronounced with smaller batch sizes, where the optimizer step represents a larger fraction of total training time. This gain comes from the reduction in both HBM traffic and the number of elementwise operations in the optimizer step, since the optimizer on each rank is only responsible for tracking states and performing updates for its local subset of parameters.

c) Our sharded optimizer keeps the same per-rank memory profile as ZeRO stage 1 (only the FP32 Adam moments are partitioned, while parameters and gradients are fully replicated), but it differs in *how* the updated weights are exchanged.

   We broadcast each tensor individually from its "owner" rank after the local `step`, creating many small messages. ZeRO-DP P_{os} performs a single fused all-gather of the whole parameter shard, so the total bytes moved per iteration is still 2Ψ, but ZeRO incurs far fewer communication calls, and therefore lower latency at scale.

Bibliography

[1] J. Austin *et al.*, “How to Scale Your Model,” 2025.

[2] H. Z. P. N. M. M. L. W. T. W. Nouamane Tazi Ferdinand Mom, “The Ultra-Scale Playbook: Training LLMs on GPU Clusters.” [Online]. Available: https://huggingface.co/spaces/nanotron/ultrascale-playbook

[3] DeepSeek-AI, “DeepSeek-V3 Technical Report.” [Online]. Available: https://arxiv.org/abs/2412.19437

Index of Figures

Index of Tables