CS 336: Language Modelling from Scratch

Assignment 2

May 3, 2024

SUNet ID: mattreed
Name: Matt Reed

1 Benchmarking

(a) Write a script to perform basic end-to-end benchmarking of the forward and backward passes in your model. Specifically, your script should support the following: • Given hyperparameters (e.g., number of layers), initialize a model. • Generate a random batch of data. • Run w warm-up steps (before starting measuring time), then time the execution of n steps (either only forward, or both forward and backward passes, depending on an argument). For timing, you can use the Python timeit module (e.g., either using the timeit function, or using timeit.default_timer(), which gives you the system's highest resolution clock, thus a better default for benchmarking than time.time()). • Call torch.cuda.synchronize() after each step. Deliverable: A script that will initialize a basics Transformer model with the given hyperparameters, create a random batch of data, and time forward and backward passes.

(b) Time the forward and backward passes for the model sizes described in §2.1.2. Use 1 warmup step and compute the average and standard deviation of timings over 5 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? Deliverable: A 1-2 sentence response with your timings.

Model Size	Forward Pass (s)	Backward Pass (s)	Total (s)
Small	0.0188	0.0340	0.0529
Medium	0.045	0.0852	0.131
Large	0.0964	0.191	0.287
Extra Large	0.185	0.359	0.545
2.7B	0.257	0.534	0.791

The standard deviation is exceptionally small (1000 ppm at the most).

(c) One caveat of benchmarking is not performing the warm-up step. Repeat your analysis without the warm-up step. How does this affect your results? Why do you think this happens? Deliverable: A 2-3 sentence response.

Model Size	Forward Pass (s)	Backward Pass (s)	Total (s)
Small	0.084	0.029	0.118
Medium	0.115	0.085	0.200
Large	0.176	0.191	0.367
Extra Large	0.247	0.360	0.607
2.7B	0.319	0.535	0.855

The standard deviation is still very small for the backward pass, but the forward pass now has a standard deviation of closer to 30 ms. It is even more enlightening to see that is is always the first forward pass that takes an order of magnitude longer than the rest.

2 function_call_table

(a) What is the total time spent on your forward pass, as measured by the PyTorch profiler? Does it match what we had measured before with the Python standard library? (Hint: use the "CPU Total" and "GPU Total" columns, and look for the row corresponding to your record_function block) Deliverable: A 1-2 sentence response.

For the forward pass, the CPU total was 183.275ms and the GPU total was 169.738ms. This matches roughly what we recorded before (0.247) since the GPU can run at the same time as the CPU, so we wouldn't need to necessarily add these two times together.

(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? Deliverable: A 1-2 sentence response

The Kernel that takes the most GPU time for the forward pass is sm80_xmma_gemm_f32f32_f32f32_f32_tn_n_tilesize128x12.. and sm80_xmma_gemm_f32f32_f32f32_f32_tn_n_tilesize64x64x.. with 439.034ms and 256.109ms respectively which takes 81.88% of the GPU time. For forward and backward passes, The time is not split between a lot more kernels, with these two still at the top but only at 17% and 15%. Others include sm80_xmma_gemm_f32f32_f32f32_f32_tn_n_tilesize64x64x.., void cut-

lass::Kernel¡cutlass_80_simt_sgemm_128x128_8.
sm80_xmma_gemm_f32f32_f32f32_f32_nn_n_tilesize64x128..

(c) Although the vast majority of FLOPs take place in matrix multiplications, you will notice that several other kernels still take a non-trivial amount of the overall runtime. What other kernels besides matrix multiplies do you see accounting for non-trivial CUDA runtime in the forward pass? Deliverable: A 1-2 sentence response.

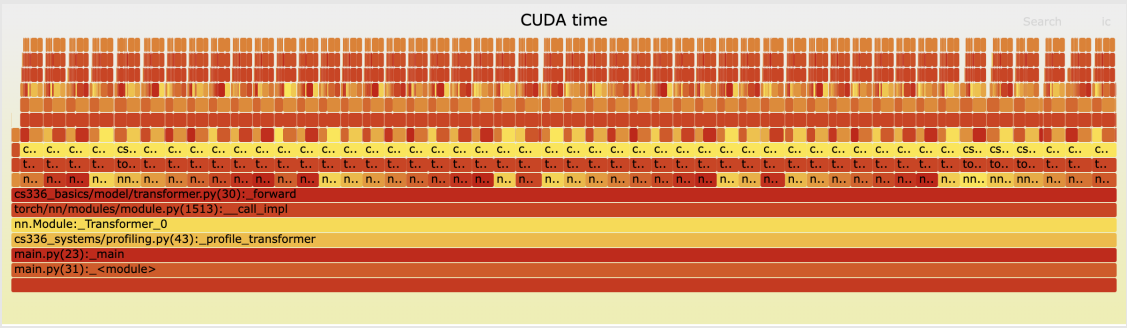
aten::copy_ (2.39%), aten::div (3.12%), aten::mul (3.06%), and aten::bmm (1.90%) all take non trivial amounts of time. Their respective kernels are maybe sm80_xmma_gemm_f32f32_f32f32_f32_nn_n_tilesize64x64x..., void at::native::vectorized_elementwise_kernel¡4, at..., void at::native::elementwise_kernel¡128, 2, at::nati...

(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? Deliverable: A 1-2 sentence response

Matrix Multiplies drops from 82% to 71%, aten::mul rises from 3.06% to 8.40%. aten::add grows from 1.7% to 3.3%. Everything else is in the same ballpark.

3 flame_graph

(a) Run with the PyTorch profiler with a XL-shaped language model's (§2.1.2) forward pass and generate a flame graph. Then, answer the questions below. (a) Include a snapshot of your flame graph here. You should see a cyclic pattern in your graph – where does that come from?



The cyclic nature is because of the multiple layers of transformer blocks and multiple linear layers within each of those.

(b) What fraction of time does your model seem to spend on all RMSNorm layers? (Hint: you can zoom in and analyze one Transformer block, then multiply by the number of layers.)

106 + 82 * 48 / 354629 us. = 0.025.
About 2.5% of the time is spent on RMS norms.

(c) What fraction of the time is spent in softmax (inside the attention operation)?

248 * 48 / 354629 = 0.033.
About 3.3% of the time is spent on softmax operations.

(d) Does the graph match your prior expectations of how runtime is distributed in your Transformer? Did you see any surprises? Deliverable: A 2-4 sentence response

To be honest, it is almost exactly what I expected. The vast majority of the runtime is spent on matrix multiplies, and this is further divided based on the relative sizes of the linear layers. The large feed forward layers take much more time than the smaller attention projections although there are more of those.

4 benchmarking_mixed_precision)

(a) Modify your benchmarking script to optionally run the model with mixed precision. Time the forward and backward passes with and without mixed-precision for each language model size described in §2.1.2. Compare the results of using full vs. mixed precision, and comment on any trends as model size changes. You may find the null-context no-op context manager to be useful. Deliverable: A 2-3 sentence response with your timings and commentary

Model Size	Time (s)	Time Mixed (s)
Small	0.0529	0.0341
Medium	0.131	0.0694
Large	0.287	0.122
Extra Large	0.545	0.192
2.7B	0.791	0.194

The mixed model runs far faster, especially for the larger model sizes. This is because GPU's can run much faster with smaller data types.

(b) Suppose we are training the following model on a GPU: ToyModel(). Suppose that the model parameters are originally in FP32. We'd like to use autocasting mixed precision with FP16. What are the data types of: • the model parameters within the autocast context, • the output of the first feed-forward layer (ToyModel.fc1), • the output of layer norm (ToyModel.ln), • the model's predicted logits, • the loss, • and the model's gradients? Deliverable: The data types for each of the components listed above

the model parameters within the autocast context: float32
the output of the first feed-forward layer (ToyModel.fc1): float16
the output of layer norm (ToyModel.ln): float32
the model's predicted logits: float16
the loss: float32
the model's gradients: float32

(c) You should have seen that FP16 mixed precision autocasting treats the layer normalization layer differently than the feed-forward layers. What parts of layer normalization are sensitive to mixed precision? If we use BF16 instead of FP16, do we still need to treat layer normalization differently? Why or why not? Deliverable: A 2-3 sentence response.

Layer normalization is sensitive to mixed precision because it includes a lot of arithmetic sums which can accmulate large rounding errors. It is especially problematic with standard deviation where each rounding error is squared. This can decrease precision and lead to unstable training. BF16 are better than FP16 because of their increased dynamic range, however it is still probably best to use float32 for these steps.

5 Pytorch_layer_norm

(a) Benchmark your RMSNorm implementation against PyTorch's native LayerNorm. For that, write a script that will: (a) Fix the number of rows in the input matrix as 50,000. (b) Iterate through the following values for the size of the last dimension: [1024, 2048, 4096, 7 8192] (c) Create random inputs x and w for the appropriate size. For LayerNorm, also create a random bias term. (d) Time 1,000 forward passes through each normalization layer using those inputs. (e) Make sure to warm up and call

torch.cuda.synchronize() after each forward pass. Report the timings you get for these 3 implementations. Which implementation seems faster — RMSNorm or LayerNorm? Why do you think that's the case? How does this gap seem to evolve as the hidden dimension grows? Deliverable: A table with your timings, and a 2-3 sentence response.

Batch Size	RMS Time (ms)	LayerNorm Time (ms)
1024	0.659	0.193
2048	1.114	0.324
4096	2.176	0.803
8192	4.298	1.649

Clearly the layernorm is much faster, especially for smaller batch sizes. This is probably because Pytorch has made a custom optimized kernel for layernorm since it is such a common operation. It was 3.41 times faster for a batch size of 1024 but only 2.61 times faster for the batch size of 8192.

(b) For each language model size described in §2.1.2, replace RMSNorm in the Transformer model by PyTorch's native LayerNorm. How long does a forward pass (with our previous default parameters for model size, etc) take on average? (Hint: you can modify your Transformer model to take flags in the constructor that indicate which implementation of normalization layer to use. Then, instantiate your Transformer with different flags to measure the effect on the forward pass). Deliverable: A 2-3 sentence response.

Model Size	Forward Pass RMS (s)	Forward Pass LayerNorm (s)
Small	0.019	0.017
Medium	0.045	0.043
Large	0.096	0.092
Extra Large	0.185	0.181
2.7B	0.257	0.251

Using the Layer Norm instead of the RMS Norm saved a little bit of time, but it was honestly pretty negligible compared to the matrix multiples, especially as the model size grew.

6 triton_rmsnorm_forward

7 rmsnorm_forward_benchmarking

(a) Extend your benchmarking script from the problem pytorch_layernorm to also have your Triton RMSNorm implementation as an alternative normalization layer. Use the same configurations from that existing script. Report the timings you get for these 3 implementations. Do you see a speed-up with your Triton kernel for any of those

sizes? What is the smallest size of the last dimension for which you see a speed-up? Deliverable: A table or plot with your measurements, and a one-two sentence response.

Batch Size	RMS Time (ms)	LayerNorm Time (ms)	Triton RMS Time (ms)
1024	0.655	0.194	0.553
2048	1.114	0.324	0.357
4096	2.176	0.803	0.620
8192	4.298	1.649	1.156

Yes, the triton kernel is faster than the pytorch implementation (and even the pytorch layer norm), but only after a certain batch size. At 2048, it has already passed the pytorch RMS norm, but it isn't faster than layernorm until 8192.

(b) Replace your original RMSNorm layers in the Transformer model with (a) your Triton implementation, in addition to (b) with PyTorch's native LayerNorm, and benchmark your forward pass with our standard hyperparameters. Report the timings you get with each normalization layer. Deliverable: A table with forward pass latencies with each of the normalization layer implementations. You can repeat the numbers you got before, just include them for ease of comparison.

Model Size	F Pass RMS (s)	F Pass LayerNorm (s)	F Pass Triton (s)
Small	0.0190	0.0173	0.0151
Medium	0.045	0.043	0.0398
Large	0.096	0.092	0.0879
Extra Large	0.185	0.181	0.183
2.7B	0.257	0.251	0.249

8 rmsnorm_jvp_g

(a) Assume that x is a N × H matrix, and that g is a H-dimensional vector. Derive the JVP for the g vector in RMSNorm (Equation 5). (Hint: start with Equation 5 and take the partial derivative of a given entry in the output vector with respect to a given entry of g). Deliverable: A derivation ending with a simple expression to compute gL when given x, g and RMSNorm(x,g)L.

First take the gradient of the RMSNorm with respect to g.

$$\begin{aligned}\nabla_g \text{RMSNorm}(x, g) &= \nabla_g \frac{x}{\sqrt{\frac{1}{d_{model}} \sum_{i=1}^{d_{model}} x_i^2 + \epsilon}} \odot g \\ &= \sum_{j=0}^{N} \frac{x_j}{\sqrt{\frac{1}{d_{model}} \sum_{i=1}^{d_{model}} x_i^2 + \epsilon}}\end{aligned}$$

The final equation for the gradient with respect to the loss is:

$$\nabla_g L = \sum_{j=0}^{N} \frac{x_j}{\sqrt{\frac{1}{d_{model}} \sum_{i=1}^{d_{model}} x_i^2 + \epsilon}} \odot \nabla_{\text{RMSNorm}} L$$

9 rmsnorm_jvp_x

(a) Assume that x is a N × H matrix, and that g is a H-dimensional vector. Derive the JVP for the x input matrix in RMSNorm (Equation 5). (Hint: start with Equation 5 and take the partial derivative of a given entry RMSNorm(x, g)k in the output vector with respect to a given xij). Deliverable: A derivation ending with a simple expression to compute xL when given x, g and RMSNorm(x,g)L.

First take the gradient of the RMSNorm with respect to x.

$$\begin{aligned}\nabla_x \text{RMSNorm}(x, g) &= \nabla_x \frac{x}{\sqrt{\frac{1}{d_{model}} \sum_{i=1}^{d_{model}} x_i^2 + \epsilon}} \odot g \\ &= \frac{1}{\text{RMS}(x)} \nabla_x (x \odot g) + (x \odot g) \nabla_x \frac{1}{\text{RMS}(x)} \\ &= \frac{g}{\text{RMS}(x)} + (x \odot g) \frac{-1}{\text{RMS}(x)^2} \nabla_x \text{RMS}(x) \\ &= \frac{g}{\text{RMS}(x)} + (x \odot g) \frac{-1}{\text{RMS}(x)^2} \sum_i \frac{x_i}{H \times \text{RMS}(x)} \\ &= \frac{g}{\text{RMS}(x)} - x \frac{\sum_i x_i \odot g}{H \text{RMS}(x)^3}\end{aligned}$$

The final equation for the gradient with respect to the loss is:

$$\nabla_x L = \frac{g \odot \nabla_{\text{RMSNorm}} L}{\text{RMS}(x)} - x \frac{\sum_i x_i \odot g \odot \nabla_{\text{RMSNorm}} L_i}{H \times \text{RMS}(x)^3}$$

10 triton_rmsnorm_backward

11 rmsnorm_benchmarking

(a) Extend your benchmarking script for normalization layers to optionally execute a backward pass. To run the backward pass, simply call result.backward(dy), where result is the output of the forward pass, and dy is another random tensor (in addition to the random inputs for the forward pass) of the appropriate shape. Remember to clear the gradients (set tensor.grad = None) of the input tensors before each forward pass; otherwise, the backward pass will also accumulate gradients at each time iteration. Then, show the average time for a combined forward and backward pass for each normalization layer implementation so far. Deliverable: A table with your timings

Batch Size	RMS Time (ms)	LayerNorm Time (ms)	Triton RMS Time (ms)
1024	0.970	0.781	0.980
2048	1.671	0.996	1.072
4096	3.253	1.583	2.00
8192	6.416	2.806	3.999

(b) Swap your PyTorch implementation of RMSNorm with your Triton implementation, and measure end-to-end performance. How long does your forward pass take now? What about your backward pass? What speed-up do you get in each case? (Hint: to make your Triton implementation a drop-in replacement for your previous one, you can write a simple torch.nn.Module subclass that has a g nn.Parameter, and that simply calls your Triton autograd function in the forward pass). Deliverable: Two-three sentences with your timings and analysis.

I already analyzed the forward pass in a previous question, so here are the backwards times for the s, m, l, xl, and 2.7 models respectively:
Without triton: 0.0513, 0.137, 0.309, 0.609, 0.892
With triton: 0.0498, 0.133, 0.301, 0.582, 0.873
It saves about 2% of the time for a forward and backward pass combined.

12 torch_compile

(a) Extend your RMSNorm benchmarking script to include another candidate: a compiled version of your PyTorch implementation of RMSNorm. How does it compare to the existing layers in the forward pass? Deliverable: A table with your timings for the forward pass including your compiled RMSNorm layer.

Last Dimension: 1024
RMSNorm Time: 0.7205798989161849 seconds
LayerNorm Time: 0.2628357317298651 seconds
Triton Time: 0.639177706092596 seconds
Compiled Time: 1.4790748711675406 seconds

Last Dimension: 2048
RMSNorm Time: 1.1831099749542773 seconds
LayerNorm Time: 0.39431128092110157 seconds
Triton Time: 0.4326761863194406 seconds
Compiled Time: 0.9795156619511545 seconds

Last Dimension: 4096
RMSNorm Time: 2.2562566259875894 seconds
LayerNorm Time: 0.8788720788434148 seconds
Triton Time: 0.7016721689142287 seconds
Compiled Time: 0.841821285430342 seconds

Last Dimension: 8192
RMSNorm Time: 4.455199412070215 seconds
LayerNorm Time: 1.717649782076478 seconds
Triton Time: 1.2332354979589581 seconds
Compiled Time: 1.5847675134427845 seconds

The compiled RMSNorm does nearly as good as the Triton implementation, especially at biggest model sizes, but it is not quite as good because it might be choosing suboptimal kernels.

(b) Run your analysis including the backward pass. How does the compiled RMSNorm layer perform? Deliverable: A table with your timings for the combined forward and backward passes including your compiled RMSNorm layer

Last Dimension: 1024
RMSNorm Time: 0.9789539077319205 seconds
LayerNorm Time: 0.7794575430452824 seconds
Triton Time: 1.1265135020948946 seconds
Compiled Time: 2.2337939380668104 seconds

Last Dimension: 2048
RMSNorm Time: 1.6737040029838681 seconds
LayerNorm Time: 0.9967236761003733 seconds
Triton Time: 1.2986896289512515 seconds
Compiled Time: 1.0077345240861177 seconds

Last Dimension: 4096
RMSNorm Time: 3.255894050002098 seconds
LayerNorm Time: 1.5831958851777017 seconds
Triton Time: 2.2505642287433147 seconds
Compiled Time: 1.7227312279865146 seconds

Last Dimension: 8192
RMSNorm Time: 6.566828748211265 seconds
LayerNorm Time: 2.806836602743715 seconds
Triton Time: 4.2724682269617915 seconds
Compiled Time: 2.9690164611674845 seconds

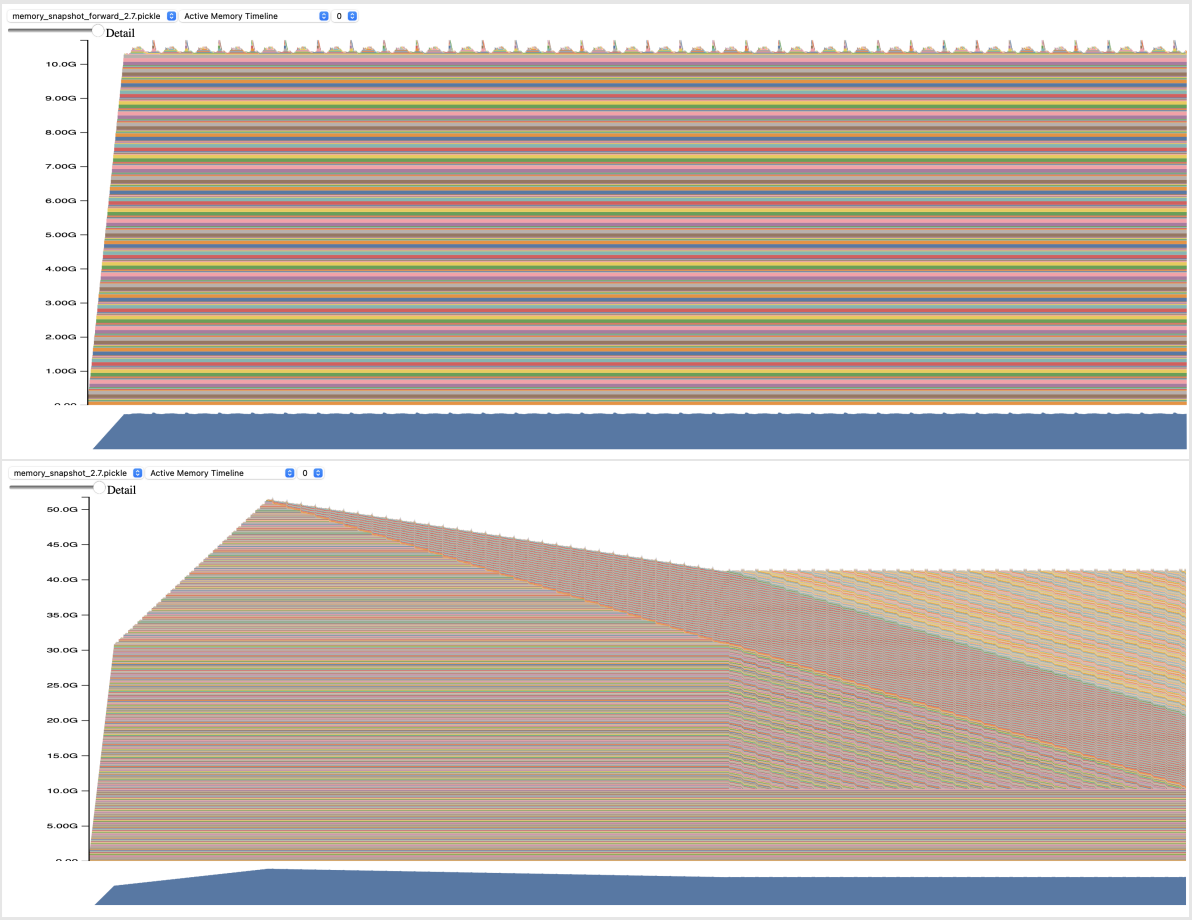
(c) Now, compile your entire Transformer model in your end-to-end benchmarking script. How does the performance of the forward pass change? What about the combined forward and backward passes and optimizer steps? Conduct this analysis for each language model size described in §2.1.2. Deliverable: A table comparing your vanilla and compiled Transformer model

Model Sizes: s, m, l, xl, 2.7
Stock Forward (s): 0.017, 0.0457, 0.0942, 0.1849, 0.257
Compiled Forward Passes (s): 0.0067, 0.0160, 0.0566, 0.134, 0.192
Stock Full Training Step (s): 0.0513, 0.137, 0.309, 0.609, 0.892
Compiled Full Training Step (s): 0.0395, 0.109, 0.263, 0.525, 0.819

13 memory_profiling

(a) Add an option to your profiling script to run your model through the memory profiler. It may be helpful to reuse some of your previous infrastructure (e.g., to activate mixed-precision, load specific model sizes, etc). Then, run your script to get a memory profile of the 2.7B model when either doing inference only (just forward pass) or a full training step. How do your memory timelines look like? Can you tell which stage is running

based on the peaks you see? Deliverable: Two images of the "Active memory timeline" of a 2.7B model, from the memory_viz tool: one for the forward pass, and one for running a full training step (forward and backward passes, then optimizer step), and a 2-3 sentence response.



The memory is super cool to look at because it matches exactly what you'd expect it to look like. First there is the steep uptick in memory which represents allocating space for the model's paramters (and the optimizer state in the second one). Then for the forward pass (note this is at inference time, so this is run with torch.no_grad()), there is only a bunch of small spikes for storing activations and such. The full pass on the other hand has a gradual incline which represents the forward pass, accumulating stored activations. It then shows the backward pass where these activations are released from memory and only the gradients are stored. Then finally the gradients are used to update the parameters and the optimizer state.

(b) What is the peak memory usage of each model size when doing a forward pass? What about when doing a full training step? Deliverable: A table with two numbers per model size.

Model Sizes: s,m,l,xl,2.7
Forward Pass (with no grad) (GB): 0.56, 1.49, 3.23, 6.42, 10.77
Full Pass (GB): 3.95, 10.55, 21.22, 38.63, 51.91

(c) Find the peak memory usage of the 2.7B model when using mixed-precision, for both a forward pass and a full optimizer step. Does mixed-precision significantly affect memory usage? Deliverable: A 2-3 sentence response.

For just the forward pass, the 2.7 Mixed Model uses 15.65 GB. For the full pass it uses 50.04 GB. I'm not sure why the forward pass is higher. If anything I would expect it to be the same (as it is in the full training step) because from my understanding the mixed precision is only used to cast the variables for matmuls, not for affecting the underlying parameters. Maybe that is why the forward pass is higher, because it needs to allocate memory for the lower precision parameters as well.

(d) Consider the 2.7B model. At our reference hyperparameters, what is the size of a tensor of activations in the Transformer residual stream, in single-precision? Give this size in MB (i.e., divide the number of bytes by 1024^2). Deliverable: A 1-2 sentence response with your derivation

Size = batch_size x context_length x d_model x data_size
Size = 16 x 128 x 2560 x 4
Size = 17.97 MB

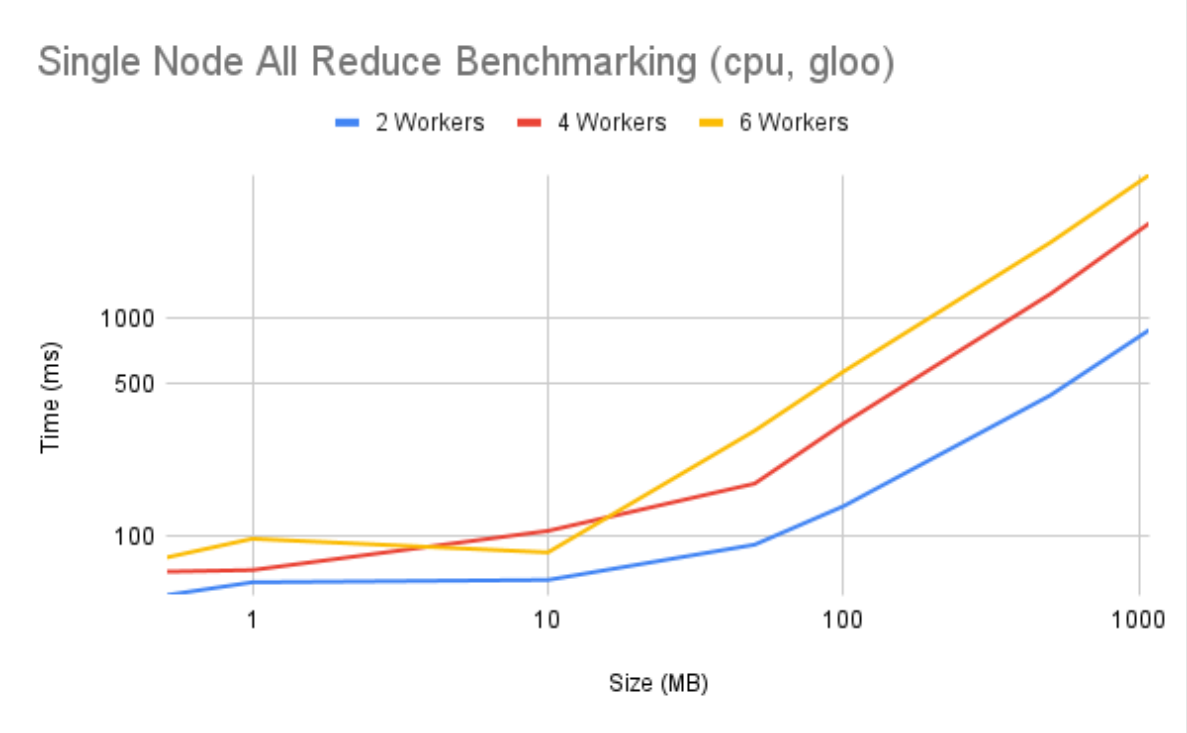
(e) Now look closely at the "Active Memory Timeline" from pytorch.org/memory_viz of a memory snapshot of the 2.7B model doing a forward pass. When you reduce the "Detail" level, the tool hides the smallest allocations to the corresponding level (e.g., putting "Detail" at 10% only shows the 10% largest allocations). What is the size of the largest allocations shown? Looking through the stack trace, can you tell where those allocations come from? Deliverable: A 1-2 sentence response.

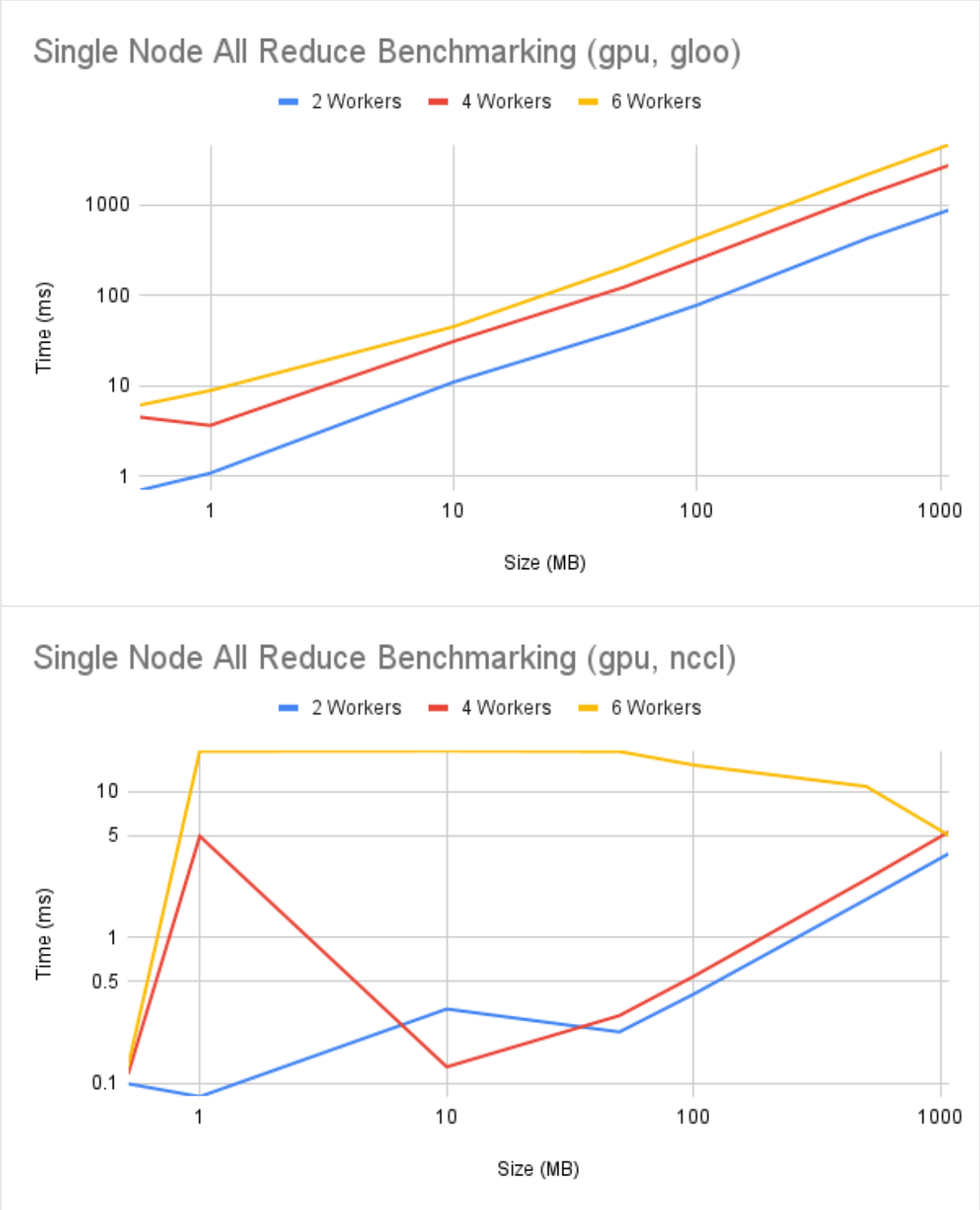
The biggest memory allocations (excluding the incremental 100 MiB blocks allocated for the parameters) are the 80.0MiB allocations from the feedforward layers.

14 distributed_communication_single_node

(a) Write a script to benchmark the runtime of the all-reduce operation in the single-node multi-process setup. The example code above may provide a reasonable starting point.

Experiment with varying the following settings: Backend + device type: Gloo + CPU, Gloo + GPU, NCCL + GPU. all-reduce data size: float32 data tensors ranging 512KB, 1MB, 10MB, 50MB, 100MB, 500MB, 1GB. Number of processes: 2, 4, or 6 processes. Resource requirements: Up to 6 GPUs. Each benchmarking run should take less than 5 minutes. Deliverable: Plot(s) and/or table(s) comparing the various settings, with 2-3 sentences of commentary about your results and thoughts about how the various factors interact.

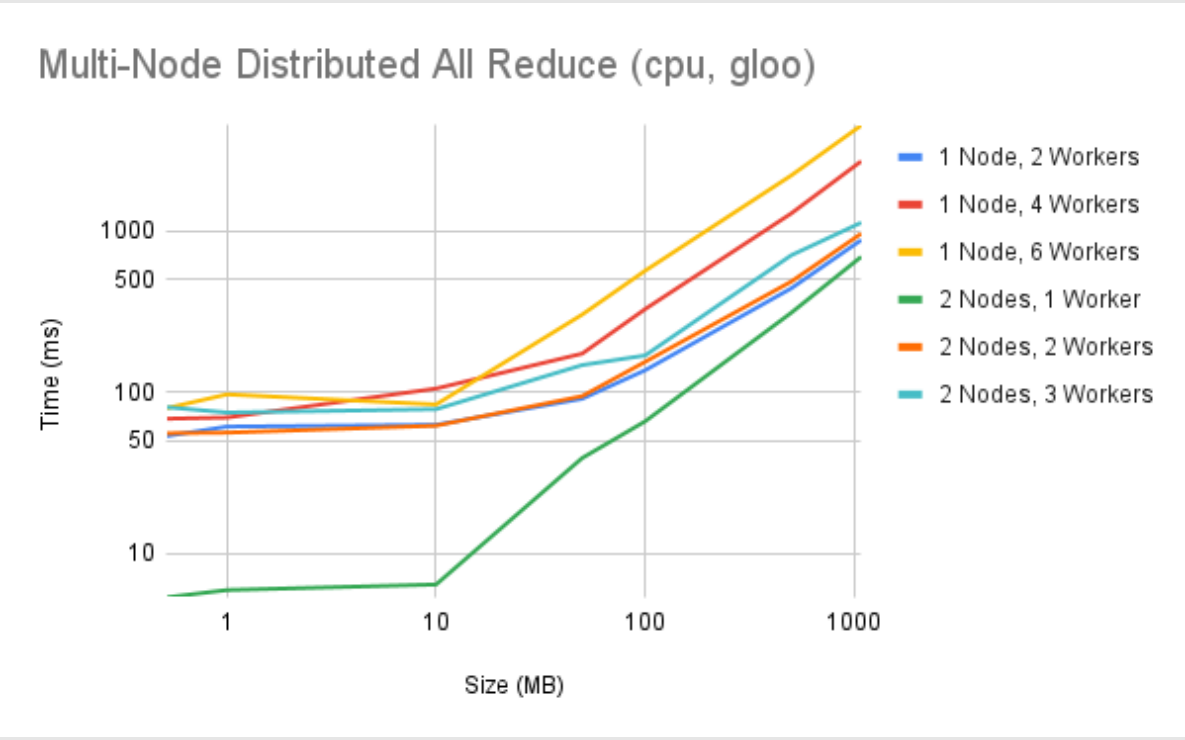


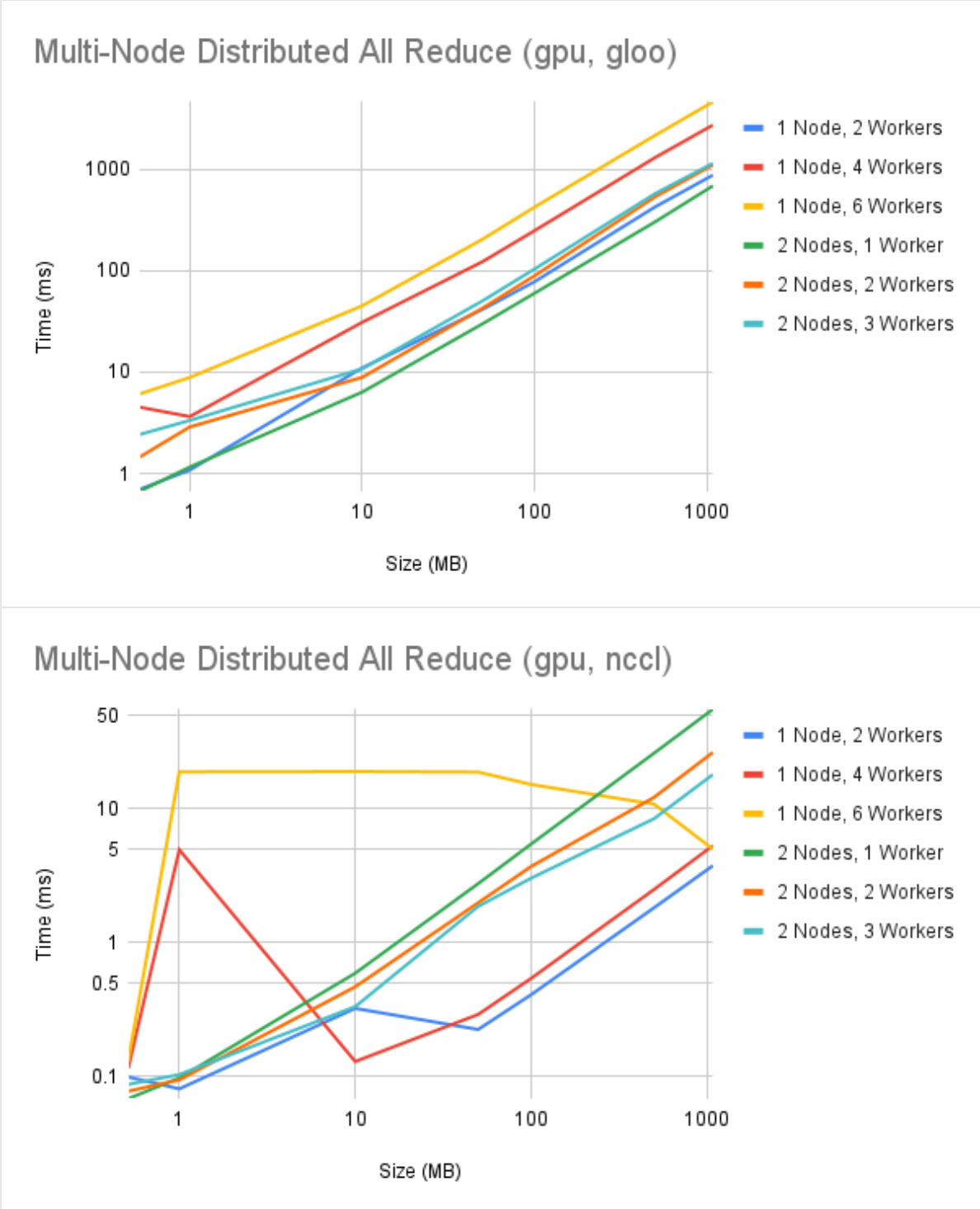


One trend that is immediately obvious is that NCCL is significantly faster (10x). Furthermore, once the model reaches a certain size to get past the overhead of having multiple workers, the time spent scales roughly linearly. There was a lot more overhead with more workers for NCCL and overall less workers generally operated faster.

15 distributed_communication_multi_node

(a) Benchmark the runtime of the all-reduce operation in the multi-node multi-process setting. In particular, vary the following settings: Number of nodes: 1 or 2. Your results from problem distributed_communication_single_node should cover the single-node setting. Backend + device type: Gloo + CPU, Gloo + GPU, NCCL + GPU. Size of data on each device: float32 data tensors ranging 512KB to 1GB. Number of processes per node: For the single-node setting, use 2, 4 or 6 processes (matching problem distributed_communication_single_node). For measuring multi-node latency, use 1, 2, or 4 processes per node. Deliverable: Plot(s) comparing the various settings, with 2-4 sentences of commentary about your results and thoughts about how the various factors interact. In particular, be sure to compare the single- and multi-node settings.





It seems that gloo operates faster when workers are split between nodes than on a single node. On the other hand, NCCL operates faster when workers are on a single node than when they are split between nodes.

16 naive_ddp

17 naive_ddp_benchmarking

(a) In this naïve DDP implementation, parameters are individually all-reduced across ranks after each backward pass. To better understand the overhead of data parallel training, create a script to benchmark your previously-implemented language model when trained with this naïve implementation of DDP. Measure the total time per training step and the proportion of time spent on communicating gradients. Collect measurements in both the single-node setting (1 node x 2 GPUs) and the multi-node setting (2 nodes x 1 GPU). Run your benchmark on GPUs with each of the model sizes described in §2.1.2. Deliverable: A description of your benchmarking setup, along with the measured time per training iteration and time spent communicating gradients for each setting.

I initialized a 2.7B transformer model and benchmarked the average full training step (as well as the individual parts):
With 2 Nodes x 1 GPU each:
Forward: 0.4200414866791107
all reduce: 0.5297914531547576
optimizer: 0.12092821132391691

With 1 Node x 2 GPU each:
Forward: 0.42503071839455514
all reduce: 0.09938307798001915
optimizer: 0.12049294619355351

Clearly, the forward and backward passes are equivalent between the two setups, however, the all reduce takes siginifcantly longer on two nodes than on one. This matches with what we predicted earlier that NCCL takes longer when split between nodes.

18 minimal_ddp_flat_benchmarking

(a) Modify your minimal DDP implementation to communicate a tensor with flattened gradients from all parameters.a Compare its performance with the minimal DDP implementation that issues an allreduce for each parameter tensor under the previously-used conditions (model sizes described in §2.1.2 on 1 node x 2 GPUs and 2 nodes x 1 GPU). Deliverable: The measured time per training iteration and time spent communicating gradients under distributed data parallel training with a single batched all-reduce call. 1-2 sentences comparing the results when batching vs. individually communicating

gradients.

Again, I initialized a 2.7B transformer model and benchmarked the average full training step (as well as the individual parts):
With 2 Nodes x 1 GPU each:
Forward: 0.41875554379075763
all reduce: 0.5043694798368961
optimizer: 0.11989447647938505
With 1 Node x 2 GPU each:
Forward: 0.4197033928707242
all reduce: 0.10176506540738046
optimizer: 0.12018219772726298

There is slight improvement in the 2 Nodes case (presumably because the overhead for NCCL communication calls is higher there), but for the single node case, it is slightly worse.

19 ddp_overlap_individual_parameters

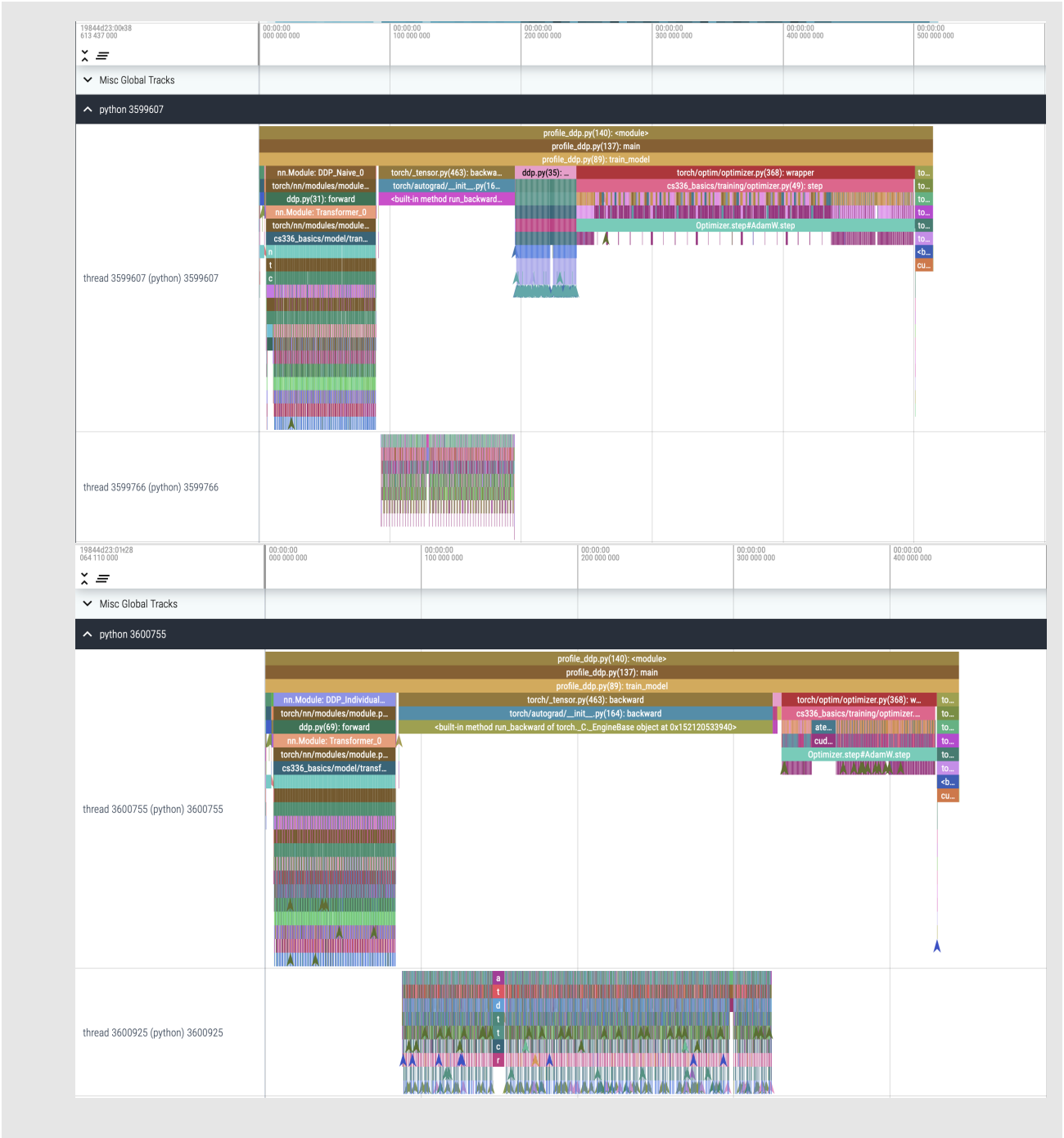
20 ddp_overlap_individual_parameters_benchmarking

(a) Benchmark the performance of your DDP implementation when overlapping backward pass computation with communication of individual parameter gradients. Compare its performance on each of the model sizes described in §2.1.2 with our previously-studied settings (the minimal DDP implementation that either issues an all-reduce for each parameter tensor or a single all-reduce for the concatenation of all parameter tensors). Deliverable: The measured time per training iteration and time spent communicating gradients under distributed data parallel training with a single batched all-reduce call. 1-2 sentences comparing the results when batching vs. individually communicating gradients.

Model	total	Forward	All Reduce	Optimizer
naive s	0.0757	0.0439	0.0236	0.0082
naive m	0.1726	0.0827	0.0716	0.0183
naive l	0.3410	0.1464	0.1552	0.0394
naive xl	0.6329	0.2475	0.3094	0.0760
naive 2.7	0.9558	0.3288	0.5059	0.1211
indiv s	0.0664	0.0579	0.0003	0.0082
indiv m	0.1386	0.1196	0.0006	0.0184
indiv l	0.2703	0.2291	0.0011	0.0400
indiv xl	0.5191	0.4403	0.0014	0.0774
indiv 2.7	0.7931	0.6718	0.0009	0.1204

The overlapping communication calls speeds up model training significantly as shown in the table. These benefits are shown in the stark contrast between all reduce times. Although the forward pass takes longer, since computation and communication is overlapping, the overall time is reduced.

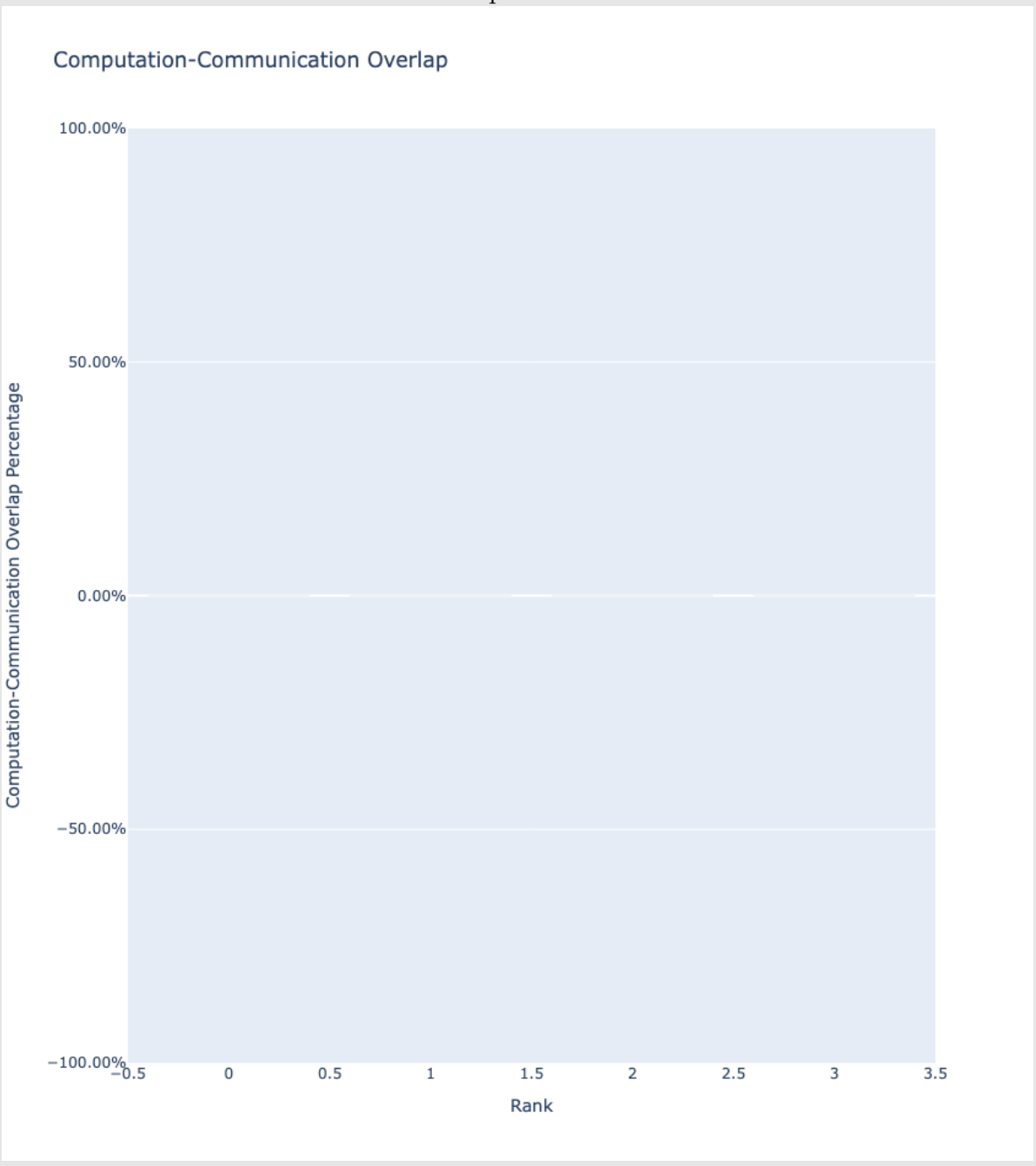
(b) Instrument your benchmarking code with the PyTorch profiler and export a chrome trace when benchmarking your DDP implementation on an XL-sized model, with and without overlapping computation with communication. Visually compare the two traces, and provide a profiler screenshot demonstrating that one implementation overlaps compute with communication while the other doesn't. Deliverable: 2 screenshots (one from the DDP implementation that overlaps compute with communication, and another that doesn't) that visually show that communication is or isn't overlapped with the backward pass.



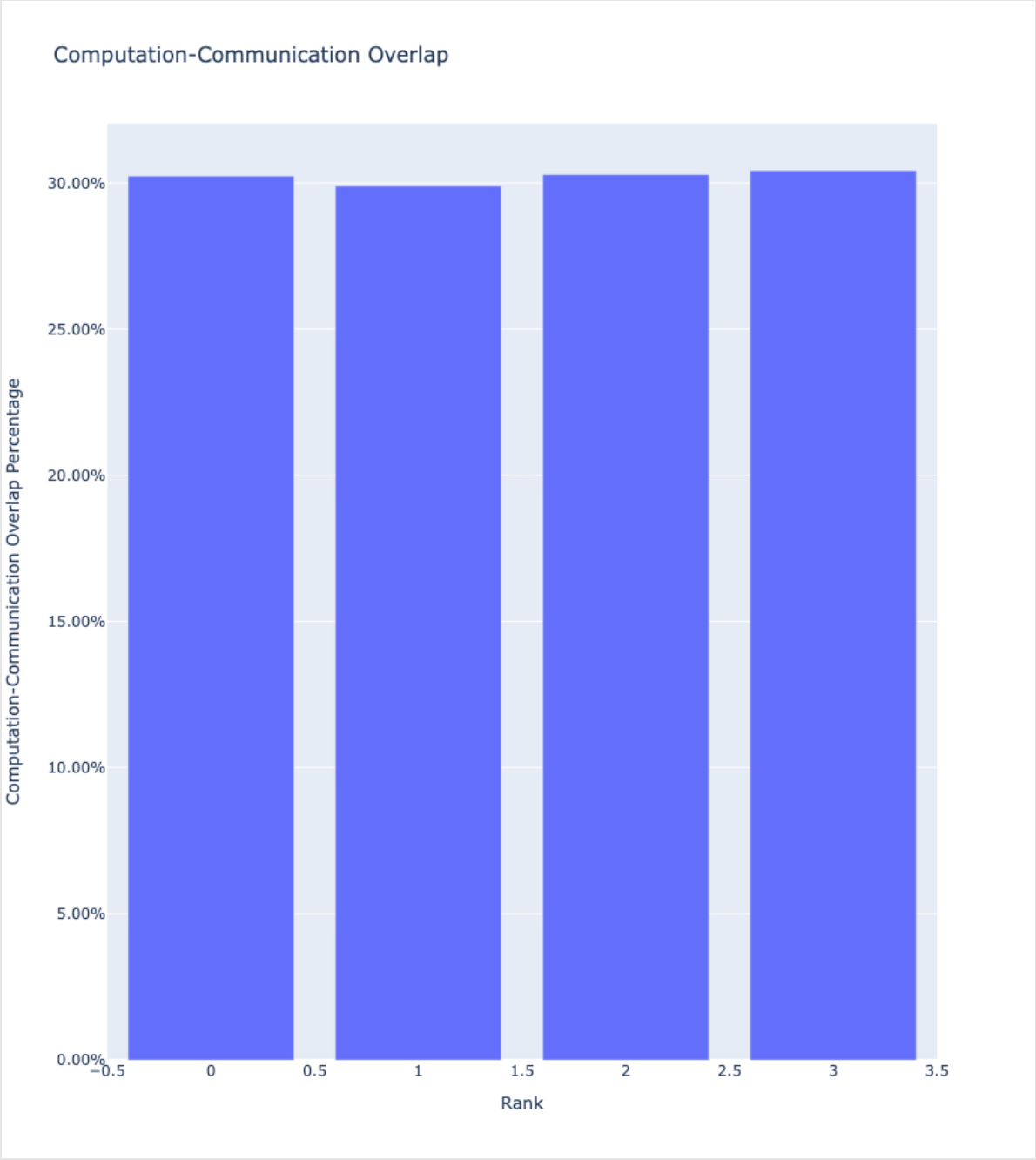
(c) A common way of quantifying the communcation-computation overlap is to divide the time spent in computation while communication by the total amount of time spent communicating. Thus, an overlap of 1.0 indicates that all communication happens while other compute is happening, while an overlap of 0.0 indicates that they are completely disjoint. Use the Holistic Trace Analysis library to measure degree of communication-computation overlap for each model.a As a sanity check, ensure that you get the expected overlap of 0.0 when analyzing the naive DDP implementation. Deliverable: 1-2

sentence response with the measured communication computation overlap ratio.

Naive Implementation:



Overlapped Implementation:



As expected the naive implementation conducts all communication independently so it does not take advantage of parallelism. The overlapped implementation has roughly 30% overlap which is not ideal, but also a big improvment over 0%.

21 ddp_overlap_bucketed

22 ddp_bucketed_benchmarking

(a) Vary the maximum bucket size (5, 10, 50, 100, 500MB) and compare the results with individually communicating parameter gradients. Experiment with each of the model sizes described in §2.1.2 on GPUs with both the NCCL and Gloo backends. Comment on your results—do they 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 would yield results that are aligned with your expectations? Deliverable: Measured time per training iteration for various bucket sizes on each backend. 3-4 sentence commentary about the results, your expectations, and potential reasons for any mismatch.

Using NCCL:

Bucket Size (MB)	Small	Large	2.7B
5	0.06007	0.20583	0.47231
10	0.05814	0.17627	0.46938
50	0.05520	0.17585	0.46978
100	0.05941	0.17407	0.47362
500	0.06203	0.17966	0.48678

Using size Large:

Bucket Size (MB)	GLOO	NCCL
5	1.3893	0.20583
10	1.330	0.17627
50	1.3568	0.17585
100	1.3724	0.17407
500	1.346	0.17966

It is clear that NCCL is much faster than GLOO, but neither benefits from an expanded bucket size by that much. Maybe the communication overhead is not large enough for it to make expanding the bucket size worth it. However,the initial bucketing did show an improvement over the individually shared parameters, so having a bucket size of around 10 MB seems to do pretty good. Furthermore, if we tried some smaller bucket sizes as well, it might be clearer what the trend is.

(b) Write an equation that models the overhead of DDP 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 (nb). From this equation, write an equation for the optimal bucket size that

minimizes DDP overhead. Deliverable: Equation that models DDP overhead, and an equation for the optimal bucket size

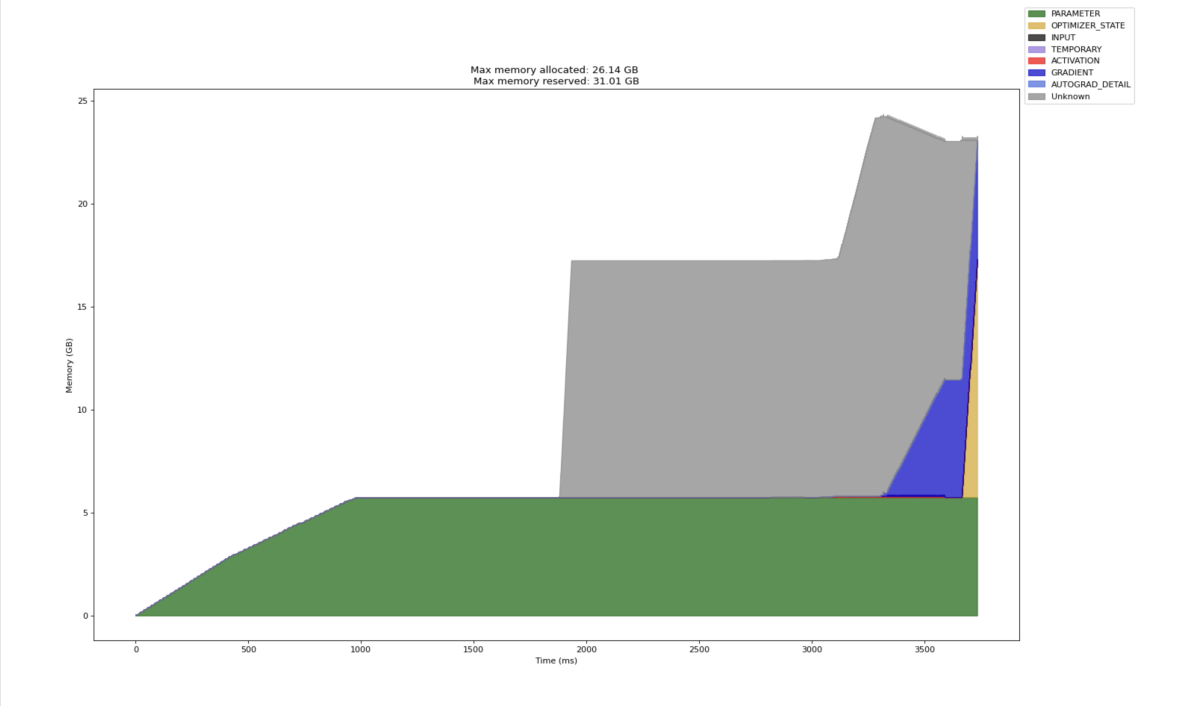
Time per bucket $= \frac{s}{w \times nb} + o$
Total Overhead $= \frac{s}{w} + o \times nb$
Therefore minimizing the DDP overhead would mean maximizing the bucket size.

23 optimizer_state_sharding

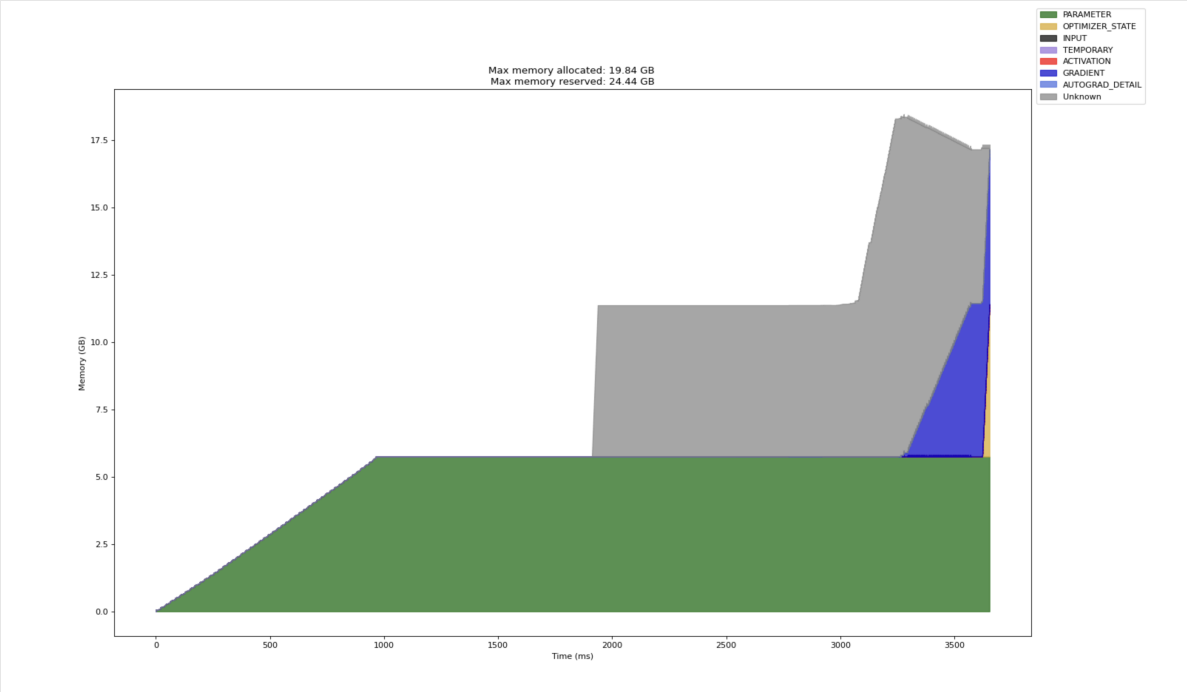
24 optimizer_state_sharding_accounting

(a) Create a script to profile the peak memory usage when training language models with and without optimizer state sharding (use the same model hyperparameters as naive_ddp_benchmarking). 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.). Perform your analysis on an XL-shaped model from §2.1.2. Deliverable: 2-3 sentence response with peak memory usage results and a breakdown of how the memory is divided between different model and optimizer components.

Non-sharded:



Sharded:



I ran this experiemnt with two workers so it is unsurprising that the optimizer state is approximately half as big in the second image. Even naively splitting parameters naively by index seems to do a pretty decent job in this scenario. The gradient storage is identical between the two, however, because that is irrespective of the optimizer sharding.

(b) How does our implementation of optimizer state sharding affect training speed? Measure the time taken per iteration with and without optimizer state sharding for each model size listed in §2.1.2, using both 1 node x 2 GPUs and 2 nodes x 1 GPU. What proportion of a training iteration is spent communicating parameter updates (i.e., the communication overhead of optimizer state sharding)? Deliverable: 2-3 sentence response with your timings.

Model Size	Sharded (1 Node, 2 GPUs)	Non-Sharded (1 Node, 2 GPUs)	Sharded (2 Nodes, 1 GPU)	Non-Sharded (2 Nodes, 1 GPU)
s	0.07117	0.07726	0.09876	0.08223
m	0.12144	0.12473	0.21154	0.16418
l	0.20450	0.21573	0.40238	0.29452
xl	0.35394	0.36822	0.76305	0.53689
2.7	0.46769	0.48816	1.18067	0.81046

For size 2.7B on 2 nodes, 0.4955820 seconds are spent on backward with 0.120947546

seconds with shard, so .37/1.18 or 31% is spent communicating parameters in the optimizer step. Surprisingly, the model actually runs faster with sharding when using 1 Node, perhaps because the communication cost is negligible compared to benefits of parallelism. However, the sharding takes 30-40% longer when using 2 nodes because of the increasing communication costs.

(c) How does our approach to optimizer state sharding differ from ZeRO stage 1 (described as ZeRODP Pos in Rajbhandari et al., 2020)? Deliverable: 2-3 sentence summary of any differences, especially those related to memory and communication volume.

ZeRODP is different than our sharded optimizer because they also distribute parameters and gradients between different nodes, not just the optimizer states.