CS336 Assignment 2 Writeup

Marcel Rød
Stanford University
roed@stanford.edu

2. Optimizing single-GPU performance

2.1. Profiling and benchmarking

2.1.1. (benchmarking_script)

(a) The script is in `cs336_systems/profile_script.py`, and implements all the requirements.

(b) Using 1 warmup step and 5 measurement steps, I get the following table of results:

Model Size	Pass	Mean	Standard Deviation
small	forward	$1.36 \cdot 10^{-2}$s	$7.42 \cdot 10^{-3}$s
small	backward	$1.51 \cdot 10^{-2}$s	$3.56 \cdot 10^{-5}$s
small	both	$2.41 \cdot 10^{-2}$s	$1.16 \cdot 10^{-4}$s
medium	forward	$4.40 \cdot 10^{-2}$s	$4.86 \cdot 10^{-2}$s
medium	backward	$3.65 \cdot 10^{-2}$s	$4.96 \cdot 10^{-5}$s
medium	both	$5.36 \cdot 10^{-2}$s	$2.24 \cdot 10^{-4}$s
large	forward	$4.84 \cdot 10^{-2}$s	$2.83 \cdot 10^{-2}$s
large	backward	$7.37 \cdot 10^{-2}$s	$3.99 \cdot 10^{-5}$s
large	both	$1.04 \cdot 10^{-1}$s	$7.40 \cdot 10^{-5}$s
xl	forward	$8.16 \cdot 10^{-2}$s	$5.01 \cdot 10^{-2}$s
xl	backward	$1.30 \cdot 10^{-1}$s	$2.89 \cdot 10^{-4}$s
xl	both	$1.80 \cdot 10^{-1}$s	$7.31 \cdot 10^{-4}$s
2.7B	forward	$7.44 \cdot 10^{-2}$s	$3.05 \cdot 10^{-2}$s
2.7B	backward	$1.55 \cdot 10^{-1}$s	$8.51 \cdot 10^{-5}$s
2.7B	both	$2.13 \cdot 10^{-1}$s	$1.66 \cdot 10^{-4}$s

Note that the standard deviation is higher for the first sample for each model size, as even when reconstructing the model and emptying the CUDA cache for every profiling instance, the GPU stays warmed up after the first sample of a given model shape. Still, the standard deviation is fairly small compared to the mean measured runtime.

(c) Now, removing the warmup step and measuring again, we get the following results:

Model Size	Pass	Mean	Standard Deviation
small	forward	$7.98 \cdot 10^{-2}$s	$1.03 \cdot 10^{-1}$s
small	backward	$1.64 \cdot 10^{-2}$s	$2.95 \cdot 10^{-3}$s
small	both	$2.83 \cdot 10^{-2}$s	$8.72 \cdot 10^{-3}$s

Model Size	Pass	Mean	Standard Deviation
medium	forward	$5.86 \cdot 10^{-2}$s	$4.93 \cdot 10^{-2}$s
medium	backward	$3.59 \cdot 10^{-2}$s	$6.54 \cdot 10^{-4}$s
medium	both	$6.04 \cdot 10^{-2}$s	$1.53 \cdot 10^{-2}$s
large	forward	$6.14 \cdot 10^{-2}$s	$3.84 \cdot 10^{-2}$s
large	backward	$7.25 \cdot 10^{-2}$s	$1.44 \cdot 10^{-3}$s
large	both	$1.15 \cdot 10^{-1}$s	$2.43 \cdot 10^{-2}$s
xl	forward	$8.95 \cdot 10^{-2}$s	$4.84 \cdot 10^{-2}$s
xl	backward	$1.28 \cdot 10^{-1}$s	$2.58 \cdot 10^{-3}$s
xl	both	$2.35 \cdot 10^{-1}$s	$1.12 \cdot 10^{-1}$s
2.7B	forward	$8.27 \cdot 10^{-2}$s	$3.00 \cdot 10^{-2}$s
2.7B	backward	$1.53 \cdot 10^{-1}$s	$4.15 \cdot 10^{-3}$s
2.7B	both	$2.47 \cdot 10^{-1}$s	$6.77 \cdot 10^{-2}$s

In this case I can clearly see the standard deviation being much larger than for the first set of measurements. For instance, for the small model, the standard deviation is larger than the mean itself. Not performing at least one warmup step, means that the GPU might have to do some initialization work during the first measurement step, which can lead to a large variance in the measured runtime. The kernels that are used for each operations have to be chosen by the CUDA runtime, and this can take much more time during the first step than during the subsequent ones.

2.1.2. `(function_call_table)`

(a) As measured by the PyTorch profiler, the mean time of the forward pass is roughly 90ms (looking at the CPU time avg, which matches the closest to what I measured in the last task). This is close to the previously measured value for the xl model size, which was 81ms.

(b) Running again with the backward and optimizer steps disabled and sorting by CUDA Total Time, I see that the abstract operation `aten::mm` takes up the most time. This is not a kernel, hoIver, and the kernel that takes the most time is

```
sm90_xmma_gemm_f32f32_tf32f32_f32_tn_n_tilesize256x128x32_warpgroupsize2x1x1_execute_segment_k_off_kernel__5x_cublas
```

with a cumulative GPU time during the forward pass of 38ms per forward pass, so 7.6ms per iteration. The kernel gets called 960 times over 5 loops, meaning I call it 192 times in a single forward pass.

When doing both the forward and the backward passes together, I see the a different matmul kernel,

```
sm90_xmma_gemm_f32f32_tf32f32_f32_nt_n_tilesize128x128x32_warpgroupsize1x1x1_execute_segment_k_off_kernel__5x_cublas
```

take up the most time, with a total of 111ms (22.2ms per iter), and 480 calls in 5 loops, so 96 calls in a single forward-backward-optimizer pass.

(c) The kernels unrelated to matrix multiplication that take up non-trivial amounts of CUDA runtime in our model are kernels related to `aten::mul`, `aten::add_`, `aten::div`, `aten::sum`, `aten::copy_`, `aten::clone`, `aten::reshape`, as well as backward versions of these operations. These ops are negligible in terms of FLOPs, but require relatively a lot of memory bandwidth, which is why they take up non-negligible amounts of time. There are a ton of specialized, specific kernels being listed in the profiling printout, but listing them here would balloon the size of the writeup.

(d) Adding the optimization pass and doing the same analysis, I see that the fraction of CUDA time spent on `aten::mm` is reduced from 30% to 18%, and that elementwise operations like `aten::mul` and `aten::sub_` take up a larger fraction in total than they did before. In fact, the most time-consuming single kernel is now an elementwise multiplication kernel (name too long to add here), taking up 7% of the total time. It is overall still the case that `aten::mm` takes up more time than `aten::mul`, which are the respective abstract operations being used, but since the `aten::mm` operations are split into many individual kernels, they don't take as much proportional time individually.

2.1.3. (flame_graph)

(a) The flame graph is shown below. The cyclic pattern comes from the fact that the model is comprised of several TransformerBlocks, which are all identical in architecture.

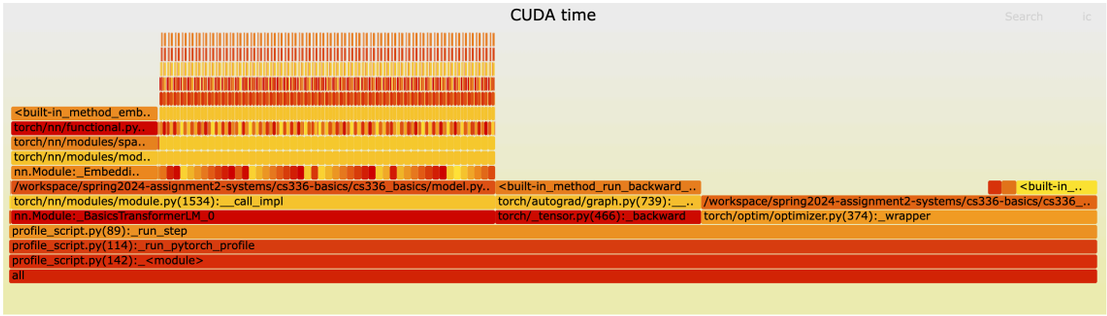


Figure 1: The flamegraph for the forward, backward and optimizer passes of the model.

I also notice the bug described by Gabriel Poesia in the Slack channel for the class, where the profiler isn't able to capture the full stack trace for the backward pass, and instead just shows the higher-level function call.

(b) The model spends $2 \cdot 0.03\%$ of the total time on RMSNorm layers for each TransformerBlock as well as 1 RMSNorm for the final LN. Multiplying this by the number of blocks gets us a total time of $(48 \cdot 2 + 1) \cdot 0.03\% = 2.91\%$ of the total time.

(c) Softmax takes 0.06% per call, and is also called once per block, so it takes up $48 \cdot 0.06\% = 2.88\%$ of the total time as well.

(d) The graph mostly matches my expectations, but I'm a bit surprized by the amount of time it takes to run simple ops like the activation functions. Hopefully this can be fixed by fusing them with the matmul operations, in order to minimize the amount of reading and writing to DRAM that's necessary to perform these ops.

2.1.4. (benchmarking_mixed_precision)

(a) With mixed precision enabled, I get the following results:

Model Size	Pass	Mean	Standard Deviation
small	forward	$1.40 \cdot 10^{-2}$s	$6.17 \cdot 10^{-3}$s
small	backward	$2.06 \cdot 10^{-2}$s	$3.64 \cdot 10^{-5}$s
small	both	$3.06 \cdot 10^{-2}$s	$9.45 \cdot 10^{-5}$s
medium	forward	$2.98 \cdot 10^{-2}$s	$1.51 \cdot 10^{-2}$s
medium	backward	$4.32 \cdot 10^{-2}$s	$7.11 \cdot 10^{-5}$s
medium	both	$6.23 \cdot 10^{-2}$s	$1.31 \cdot 10^{-4}$s
large	forward	$4.64 \cdot 10^{-2}$s	$3.34 \cdot 10^{-2}$s

Model Size	Pass	Mean	Standard Deviation
large	backward	$8.22 \cdot 10^{-2}$s	$6.88 \cdot 10^{-5}$s
large	both	$1.11 \cdot 10^{-1}$s	$3.24 \cdot 10^{-4}$s
xl	forward	$7.36 \cdot 10^{-2}$s	$6.12 \cdot 10^{-2}$s
xl	backward	$1.38 \cdot 10^{-1}$s	$2.07 \cdot 10^{-4}$s
xl	both	$1.91 \cdot 10^{-1}$s	$9.48 \cdot 10^{-3}$s
2.7B	forward	$6.63 \cdot 10^{-2}$s	$3.68 \cdot 10^{-2}$s
2.7B	backward	$1.55 \cdot 10^{-1}$s	$2.55 \cdot 10^{-4}$s
2.7B	both	$1.97 \cdot 10^{-1}$s	$1.66 \cdot 10^{-4}$s

The mean times are somewhat shorter for the larger models, but the difference is not as large as one would expect. At small sizes, the model is slightly slower than the single precision model. This is likely due to overheads and the cost of elementwise operations that haven't yet been fused.

(b) Using the `ToyModel` class in an automatic mixed precision context, we have the following:
- Model parameters are FP32
- The output of `fc1` is FP16
- The layer norm outputs are in FP32
- The predicted logits are FP16
- The loss is in FP16
- The model gradients are in FP16, but are later aggregated in FP32

(c) Since the layer normalization does accumulations (a mean and often a stddev estimation) over large dimensions, it is useful to have it in a higher precision than FP16. This is less important when using BF16, however, since the range of values that can be represented is closer to that of FP32. There is still a benefit to using FP32 in this case, however, and it's known that specifically for layer normalization the difference between FP/BF16 and FP32 is enough to affect overall model stability.

2.1.5. `(pytorch_layernorm)`

(a) The produced timings are as follows:

Layer	N_{cols}	Mean Time	Relative Delta
LayerNorm	1024	$2.011 \cdot 10^{-4}$s	1 x
LayerNorm	2048	$3.306 \cdot 10^{-4}$s	1 x
LayerNorm	4096	$8.140 \cdot 10^{-4}$s	1 x
LayerNorm	8192	$1.667 \cdot 10^{-3}$s	1 x
RMSNorm	1024	$6.444 \cdot 10^{-4}$s	3.2 x
RMSNorm	2048	$1.116 \cdot 10^{-3}$s	3.4 x
RMSNorm	4096	$2.174 \cdot 10^{-3}$s	2.67 x
RMSNorm	8192	$4.288 \cdot 10^{-3}$s	2.57 x

They show that the PyTorch LayerNorm is between 3 and 4 times faster than the RMSNorm implementation. The difference gets somewhat smaller as the number of columns increases, but even at the largest size, the LayerNorm is still around 2.5 times faster. This is algorithmically surprising, since the RMSNorm should be doing less work than the LayerNorm, but it is likely due to the fact that the PyTorch implementation is highly optimized and uses a fused kernel.

(b) Swapping the RMSNorm for PyTorch's LayerNorm implementation, we get the following times:

Model Size	Pass	Mean	Standard Deviation
small	forward	$1.19 \cdot 10^{-2}$s	$6.85 \cdot 10^{-3}$s
small	backward	$1.32 \cdot 10^{-2}$s	$1.78 \cdot 10^{-5}$s
small	both	$2.08 \cdot 10^{-2}$s	$5.01 \cdot 10^{-5}$s
medium	forward	$3.81 \cdot 10^{-2}$s	$4.27 \cdot 10^{-2}$s
medium	backward	$3.22 \cdot 10^{-2}$s	$8.57 \cdot 10^{-5}$s
medium	both	$4.72 \cdot 10^{-2}$s	$1.44 \cdot 10^{-4}$s
large	forward	$4.54 \cdot 10^{-2}$s	$2.58 \cdot 10^{-2}$s
large	backward	$6.66 \cdot 10^{-2}$s	$1.76 \cdot 10^{-4}$s
large	both	$9.47 \cdot 10^{-2}$s	$1.08 \cdot 10^{-4}$s
xl	forward	$7.43 \cdot 10^{-2}$s	$4.19 \cdot 10^{-2}$s
xl	backward	$1.19 \cdot 10^{-1}$s	$1.87 \cdot 10^{-4}$s
xl	both	$1.66 \cdot 10^{-1}$s	$3.34 \cdot 10^{-4}$s
2.7B	forward	$6.77 \cdot 10^{-2}$s	$2.36 \cdot 10^{-2}$s
2.7B	backward	$1.45 \cdot 10^{-1}$s	$1.94 \cdot 10^{-4}$s
2.7B	both	$2.00 \cdot 10^{-1}$s	$2.46 \cdot 10^{-4}$s

Note that the times are shorter across the board, with the smaller models showing marginally better results, because their performance is more sensitive to the overhead of the RMSNorm implementation.

2.2. Writing a fused RMSNorm kernel

2.2.1. `(rmsnorm_forward_benchmarking)`

(a) I compare the Triton RMSNorm implementation to the PyTorch LayerNorm and non-native RMSNorm implementations below:

Layer	N_{cols}	Mean Time
LayerNorm	1024	$2.48 \cdot 10^{-4}$ s
LayerNorm	2048	$3.30 \cdot 10^{-4}$ s
LayerNorm	4096	$8.14 \cdot 10^{-4}$ s
LayerNorm	8192	$1.67 \cdot 10^{-3}$ s
RMSNorm	1024	$6.45 \cdot 10^{-4}$ s
RMSNorm	2048	$1.12 \cdot 10^{-3}$ s
RMSNorm	4096	$2.17 \cdot 10^{-3}$ s
RMSNorm	8192	$4.29 \cdot 10^{-3}$ s
RMSNormTriton	1024	$7.96 \cdot 10^{-4}$ s
RMSNormTriton	2048	$5.85 \cdot 10^{-4}$ s
RMSNormTriton	4096	$8.55 \cdot 10^{-4}$ s
RMSNormTriton	8192	$1.79 \cdot 10^{-3}$ s

A speedup is visible already from the size of 2048, however the LayerNorm is still faster than the Triton RMSNorm implementation at all sizes.

(b) The full comparison for all types of norms follows below. Note that I added more warmup steps to further decrease the variance in the measurements, and this can be seen from the lower standard deviations.

Model Size	Pass	Norm	Mean	Standard Deviation
small	forward	RMSNorm	$8.75 \cdot 10^{-3}$s	$1.22 \cdot 10^{-4}$s
small	forward	LayerNorm	$7.42 \cdot 10^{-3}$s	$3.01 \cdot 10^{-5}$s
small	forward	RMSNormTriton	$1.10 \cdot 10^{-2}$s	$5.50 \cdot 10^{-5}$s
medium	forward	RMSNorm	$1.69 \cdot 10^{-2}$s	$1.31 \cdot 10^{-4}$s
medium	forward	LayerNorm	$1.47 \cdot 10^{-2}$s	$9.44 \cdot 10^{-5}$s
medium	forward	RMSNormTriton	$2.15 \cdot 10^{-2}$s	$3.61 \cdot 10^{-5}$s
large	forward	RMSNorm	$3.06 \cdot 10^{-2}$s	$4.30 \cdot 10^{-5}$s
large	forward	LayerNorm	$2.90 \cdot 10^{-2}$s	$1.39 \cdot 10^{-4}$s
large	forward	RMSNormTriton	$3.39 \cdot 10^{-2}$s	$1.27 \cdot 10^{-4}$s
xl	forward	RMSNorm	$5.03 \cdot 10^{-2}$s	$1.12 \cdot 10^{-4}$s
xl	forward	LayerNorm	$4.81 \cdot 10^{-2}$s	$3.48 \cdot 10^{-5}$s
xl	forward	RMSNormTriton	$4.73 \cdot 10^{-2}$s	$6.73 \cdot 10^{-5}$s
2.7B	forward	RMSNorm	$5.91 \cdot 10^{-2}$s	$5.84 \cdot 10^{-5}$s
2.7B	forward	LayerNorm	$5.60 \cdot 10^{-2}$s	$1.29 \cdot 10^{-4}$s
2.7B	forward	RMSNormTriton	$5.58 \cdot 10^{-2}$s	$7.68 \cdot 10^{-5}$s

For smaller models, the Triton RMSNorm is slower than the PyTorch LayerNorm, but for larger models, the Triton RMSNorm is faster, even edging out the native LayerNorm implementation. The transition happens at size XL, which has an inner dimension of 1600, where the Triton RMSNorm is faster than the PyTorch non-native RMSNorm. The model might be slower at smaller sizes due to the overhead of doing a Triton call. This could potentially be mitigated by fusing more operations together.

2.2.2. `(rmsnorm_jvp_g)`

(a) Given $\nabla_{\text{rms}} L$, we want to find $\nabla_g L$

$$\text{rms}(x, g) = \frac{x}{\sqrt{\frac{1}{d_{\text{model}}} \sum_{i=1}^{d_{\text{model}}} x_i^2 + \varepsilon}} \odot g$$

we use the hadamard product rule to get

$$\nabla_g \text{ rms}(x, g) = \frac{x}{\sqrt{\frac{1}{d_{\text{model}}} \sum_{i=1}^{d_{\text{model}}} x_i^2 + \varepsilon}}$$

$$\nabla_g L = \nabla_g \text{rms}(x, g) \nabla_{\text{rms}} L = \frac{x \odot \nabla_{\text{rms}} L}{\sqrt{\frac{1}{d_{\text{model}}} \sum_{i=1}^{d_{\text{model}}} x_i^2 + \varepsilon}}$$

(b) Implemented in the code.

2.2.3. `(rmsnorm_jvp_x)`

(a) Given $\nabla_{\text{rms}} L$, we want to find $\nabla_x L$. Using the chain rule, we get

$$\nabla_x L = \left(\frac{\partial R_i}{\partial x_j} \frac{\partial L}{\partial R_i} \right)_j$$

Define $\text{RMS} = \sqrt{\frac{1}{d_{\text{model}}} \sum_{i=1}^{d_{\text{model}}} x_i^2 + \varepsilon}$. Then

$$\begin{aligned}
\frac{\partial R}{\partial x_j} &= \left(\frac{\delta_{ij}}{\text{RMS}} - \frac{\frac{1}{d_{\text{model}}} x_i x_j}{\text{RMS}^3} \right) \odot g_i \\
&= \frac{1}{\text{RMS}} \left(\delta_{ij} g_i - \left(\frac{1}{d_{\text{model}}} \right) \frac{x_i x_j}{\text{RMS}^2} g_i \right)
\end{aligned}$$

And combining with $\frac{\partial L}{\partial R_i}$ we get the full expression

$$\begin{aligned}
\nabla_x L &= \frac{1}{\text{RMS}} \sum_i \left(\delta_{ij} g_i - \frac{1}{d_{\text{model}}} x_i x_j \frac{1}{\text{RMS}^2} g_i \right) \frac{\partial L}{\partial R_i} \\
&= \frac{1}{\text{RMS}} \left(g_j \frac{\partial R}{\partial x_j} - \frac{1}{d_{\text{model}}} \sum_i x_i g_i \frac{\partial R}{\partial x_i} \right)_j
\end{aligned}$$

Which gives us the correct result when implemented.

2.2.4. (rmsnorm_benchmarking)

(a) The following table shows combined forward and backward pass timings for each of the layers:

Layer	N_{cols}	Mean Time
LayerNorm	1024	$2.480 \cdot 10^{-4}$ s
LayerNorm	2048	$3.302 \cdot 10^{-4}$ s
LayerNorm	4096	$8.141 \cdot 10^{-4}$ s
LayerNorm	8192	$1.667 \cdot 10^{-3}$ s
RMSNorm	1024	$6.453 \cdot 10^{-4}$ s
RMSNorm	2048	$1.116 \cdot 10^{-3}$ s
RMSNorm	4096	$2.174 \cdot 10^{-3}$ s
RMSNorm	8192	$4.287 \cdot 10^{-3}$ s
RMSNormTriton	1024	$7.959 \cdot 10^{-4}$ s
RMSNormTriton	2048	$5.846 \cdot 10^{-4}$ s
RMSNormTriton	4096	$8.545 \cdot 10^{-4}$ s
RMSNormTriton	8192	$1.791 \cdot 10^{-3}$ s

(b) We measure all three implementations for the forward and backward pass, giving the following results:

Model Size	Pass	Norm	Mean	Standard Deviation
small	forward	RMSNorm	$8.68 \cdot 10^{-3}$s	$1.23 \cdot 10^{-4}$s
small	backward	RMSNorm	$1.51 \cdot 10^{-2}$s	$2.41 \cdot 10^{-5}$s
small	forward	LayerNorm	$7.40 \cdot 10^{-3}$s	$3.16 \cdot 10^{-5}$s
small	backward	LayerNorm	$1.33 \cdot 10^{-2}$s	$1.58 \cdot 10^{-5}$s
small	forward	RMSNormTriton	$1.12 \cdot 10^{-2}$s	$3.12 \cdot 10^{-5}$s

Model Size	Pass	Norm	Mean	Standard Deviation
small	backward	RMSNormTriton	$1.73 \cdot 10^{-2}$s	$1.43 \cdot 10^{-4}$s
medium	forward	RMSNorm	$1.71 \cdot 10^{-2}$s	$2.66 \cdot 10^{-4}$s
medium	backward	RMSNorm	$3.65 \cdot 10^{-2}$s	$5.19 \cdot 10^{-5}$s
medium	forward	LayerNorm	$1.48 \cdot 10^{-2}$s	$2.98 \cdot 10^{-5}$s
medium	backward	LayerNorm	$3.24 \cdot 10^{-2}$s	$5.58 \cdot 10^{-5}$s
medium	forward	RMSNormTriton	$2.15 \cdot 10^{-2}$s	$4.88 \cdot 10^{-5}$s
medium	backward	RMSNormTriton	$3.35 \cdot 10^{-2}$s	$7.30 \cdot 10^{-5}$s
large	forward	RMSNorm	$3.08 \cdot 10^{-2}$s	$7.90 \cdot 10^{-5}$s
large	backward	RMSNorm	$7.35 \cdot 10^{-2}$s	$1.18 \cdot 10^{-4}$s
large	forward	LayerNorm	$2.91 \cdot 10^{-2}$s	$2.83 \cdot 10^{-5}$s
large	backward	LayerNorm	$6.67 \cdot 10^{-2}$s	$6.40 \cdot 10^{-5}$s
large	forward	RMSNormTriton	$3.35 \cdot 10^{-2}$s	$2.70 \cdot 10^{-4}$s
large	backward	RMSNormTriton	$6.76 \cdot 10^{-2}$s	$2.66 \cdot 10^{-4}$s
xl	forward	RMSNorm	$5.07 \cdot 10^{-2}$s	$5.95 \cdot 10^{-5}$s
xl	backward	RMSNorm	$1.30 \cdot 10^{-1}$s	$3.37 \cdot 10^{-4}$s
xl	forward	LayerNorm	$4.82 \cdot 10^{-2}$s	$1.53 \cdot 10^{-4}$s
xl	backward	LayerNorm	$1.19 \cdot 10^{-1}$s	$2.37 \cdot 10^{-4}$s
xl	forward	RMSNormTriton	$4.76 \cdot 10^{-2}$s	$3.01 \cdot 10^{-4}$s
xl	backward	RMSNormTriton	$1.20 \cdot 10^{-1}$s	$6.82 \cdot 10^{-4}$s
2.7B	forward	RMSNorm	$5.91 \cdot 10^{-2}$s	$1.01 \cdot 10^{-4}$s
2.7B	backward	RMSNorm	$1.55 \cdot 10^{-1}$s	$1.06 \cdot 10^{-4}$s
2.7B	forward	LayerNorm	$5.60 \cdot 10^{-2}$s	$1.47 \cdot 10^{-5}$s
2.7B	backward	LayerNorm	$1.45 \cdot 10^{-1}$s	$2.34 \cdot 10^{-4}$s
2.7B	forward	RMSNormTriton	$5.58 \cdot 10^{-2}$s	$1.54 \cdot 10^{-4}$s
2.7B	backward	RMSNormTriton	$1.45 \cdot 10^{-1}$s	$1.41 \cdot 10^{-4}$s

For the largest size, the forward pass of the triton kernel is slightly faster than both the RMSNorm and the LayerNorm implementations, at 55.8ms. The backward pass is faster than the PyTorch RMSNorm implementation, and exactly as fast as the LayerNorm implementation, at 145ms.

2.3. PyTorch JIT compiler

2.3.1. `(torch_compile)`

(a) Only the forward pass:

Layer	N_{cols}	Mean Time
LayerNorm	1024	$2.257 \cdot 10^{-4}$s
LayerNorm	2048	$3.332 \cdot 10^{-4}$s
LayerNorm	4096	$8.170 \cdot 10^{-4}$s
LayerNorm	8192	$1.669 \cdot 10^{-3}$s
RMSNorm	1024	$6.464 \cdot 10^{-4}$s

Layer	N_{cols}	Mean Time
RMSNorm	2048	$1.117 \cdot 10^{-3}$s
RMSNorm	4096	$2.174 \cdot 10^{-3}$s
RMSNorm	8192	$4.288 \cdot 10^{-3}$s
RMSNormTriton	1024	$7.705 \cdot 10^{-4}$s
RMSNormTriton	2048	$5.929 \cdot 10^{-4}$s
RMSNormTriton	4096	$9.490 \cdot 10^{-4}$s
RMSNormTriton	8192	$1.437 \cdot 10^{-3}$s
RMSNorm Compiled	1024	$1.562 \cdot 10^{-3}$s
RMSNorm Compiled	2048	$9.417 \cdot 10^{-4}$s
RMSNorm Compiled	4096	$1.532 \cdot 10^{-3}$s
RMSNorm Compiled	8192	$1.925 \cdot 10^{-3}$s

We see that the compiled RMSNorm implementation is slower than the Triton implementation for all sizes. At size 1024, the compiled RMSNorm It is faster than the PyTorch implementation except at 1024

(b) With the backward pass included:

Layer	N_{cols}	Mean Time
LayerNorm	1024	$1.054 \cdot 10^{-3}$s
LayerNorm	2048	$1.017 \cdot 10^{-3}$s
LayerNorm	4096	$1.609 \cdot 10^{-3}$s
LayerNorm	8192	$2.837 \cdot 10^{-3}$s
RMSNorm	1024	$9.603 \cdot 10^{-4}$s
RMSNorm	2048	$1.678 \cdot 10^{-3}$s
RMSNorm	4096	$3.263 \cdot 10^{-3}$s
RMSNorm	8192	$6.426 \cdot 10^{-3}$s
RMSNormTriton	1024	$1.618 \cdot 10^{-3}$s
RMSNormTriton	2048	$1.867 \cdot 10^{-3}$s
RMSNormTriton	4096	$2.523 \cdot 10^{-3}$s
RMSNormTriton	8192	$5.071 \cdot 10^{-3}$s
RMSNorm Compiled	1024	$3.224 \cdot 10^{-3}$s
RMSNorm Compiled	2048	$2.405 \cdot 10^{-3}$s
RMSNorm Compiled	4096	$3.525 \cdot 10^{-3}$s
RMSNorm Compiled	8192	$4.443 \cdot 10^{-3}$s

When including the backward pass, it seems like Triton RMSNorm is faster than the compiled norm until size 8192, where the compiled norm is faster. I think this has to do with the parallel summation required for the weight gradients in the backward pass. Getting this right requires some tuning, which `torch.compile` does well for large sizes. My bespoke implementation seems to be better optimized for smaller dimension inputs, however. Note that I'm using PyTorch 2.3.0, which has significantly better performance with `torch.compile` than 2.2.

(c) A comparison of the vanilla and compiled Transformer model follows:

Size	Pass	Compiled?	Mean Time	Standard Deviation
small	forward	No	$8.76 \cdot 10^{-3}$s	$4.16 \cdot 10^{-5}$s
small	forward	Yes	$5.14 \cdot 10^{-3}$s	$7.71 \cdot 10^{-5}$s
small	both	No	$2.36 \cdot 10^{-2}$s	$5.41 \cdot 10^{-5}$s
small	both	Yes	$1.40 \cdot 10^{-2}$s	$1.13 \cdot 10^{-5}$s
medium	forward	No	$1.81 \cdot 10^{-2}$s	$8.43 \cdot 10^{-5}$s
medium	forward	Yes	$1.02 \cdot 10^{-2}$s	$9.83 \cdot 10^{-5}$s
medium	both	No	$5.67 \cdot 10^{-2}$s	$6.45 \cdot 10^{-4}$s
medium	both	Yes	$3.19 \cdot 10^{-2}$s	$1.17 \cdot 10^{-4}$s
large	forward	No	$3.06 \cdot 10^{-2}$s	$9.01 \cdot 10^{-5}$s
large	forward	Yes	$3.33 \cdot 10^{-2}$s	$1.16 \cdot 10^{-4}$s
large	both	No	$1.04 \cdot 10^{-1}$s	$4.00 \cdot 10^{-4}$s
large	both	Yes	$7.95 \cdot 10^{-2}$s	$9.34 \cdot 10^{-5}$s
xl	forward	No	$5.05 \cdot 10^{-2}$s	$3.26 \cdot 10^{-5}$s
xl	forward	Yes	$3.04 \cdot 10^{-2}$s	$9.45 \cdot 10^{-5}$s
xl	both	No	$1.79 \cdot 10^{-1}$s	$2.44 \cdot 10^{-4}$s
xl	both	Yes	$1.14 \cdot 10^{-1}$s	$1.28 \cdot 10^{-4}$s
2.7B	forward	No	$5.89 \cdot 10^{-2}$s	$1.02 \cdot 10^{-4}$s
2.7B	forward	Yes	$4.09 \cdot 10^{-2}$s	$1.75 \cdot 10^{-4}$s
2.7B	both	No	$2.13 \cdot 10^{-1}$s	$8.93 \cdot 10^{-5}$s
2.7B	both	Yes	$1.55 \cdot 10^{-1}$s	$1.28 \cdot 10^{-3}$s

We see that the compiled model is faster than the vanilla model for most sizes and passes, in fact, all except for the forward pass for the "large" model are faster.

2.4. Profiling memory

2.4.1. `(memory_profiling)`

(a) The forward pass only looks very flat, and we can see the memory usage pattern repeat for each layer in the forward pass.

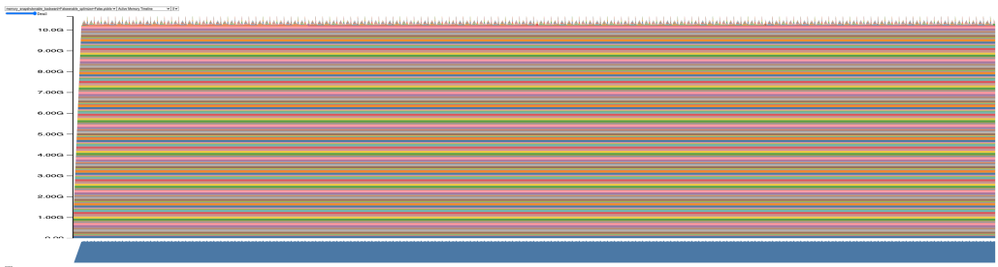


Figure 2: The forward pass with `torch.no_grad()` enabled.

For the full step, we see clear spikes in memory usage:

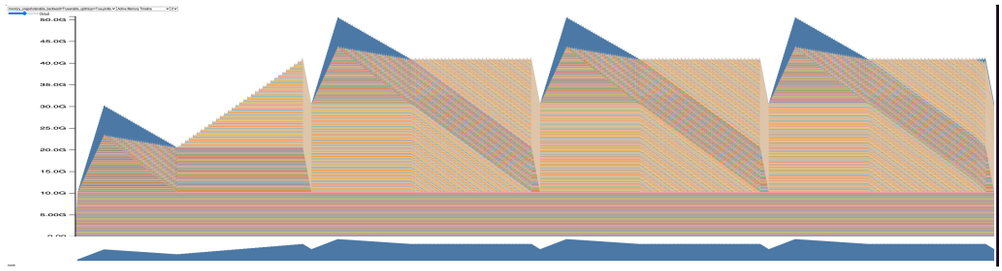


Figure 3: The full step. We see the initial spike from the forward pass, followed by a decrease in memory usage as the backward pass is happening. Then the optimizer step keeps the memory usage roughly constant. Finally there's a dip in memory usage when the gradients are set to `None`. This happens once for the warmup step, and 3 times for the profiled steps. The first iteration is a special case because we have to allocate memory for the optimizer state, and we see this in the ramp in this phase.

(b) We use the `torch.cuda.max_memory_allocated` function and reset appropriately to measure the memory usage of the forward and backward passes.

Size	Pass	Memory
small	forward	698.99 MiB
small	full step	2.81 GiB
medium	forward	1.53 GiB
medium	full step	7.19 GiB
large	forward	3.12 GiB
large	full step	13.72 GiB
xl	forward	6.05 GiB
xl	full step	23.88 GiB
2.7B	forward	10.07 GiB
2.7B	full step	38.56 GiB

It seems like it takes around 4 times as much peak memory to do the full step compared to just the forward pass. Notice also that the peak memory usage is lower than the spikes we saw in the previous plots. I think this is because there are cached files in memory.

(c) With mixed precision, we get the following:

Size	Pass	Memory
2.7B	forward	14.63 GiB
2.7B	full step	43.30 GiB

Clearly, mixed precision takes more memory than running at full precision, which is a bit surprising. The memory usage is up by almost 50% for the forward pass and around 20% on the full step, both of which are quite significant.

(d) The the residual stream takes up

$$\text{batch\_size} \cdot \text{context\_length} \cdot \text{d\_model} \cdot \frac{4\ \text{B}}{2^{20}\frac{\text{B}}{\text{MiB}}},$$

which comes out to exactly 20 MiB for the 2.7B model.

(e) The largest individual tensors are 100 MiB. Since the model was initialized on CPU then moved to GPU, the stacktrace shows the `.to()` call as the source of the memory allocation. Considering the size of these tensors, it is obvious that they belong to the feedforward linear layers, which are $4 \cdot$ d_model $\cdot$ d_ff bytes, in our case $4 \cdot 2560 \cdot (2560 \cdot 4)$ B $= 100$ MiB

3. Distributed data parallel training

3.1. Single-node distributed communication in PyTorch

3.1.1. `(distributed_communication_single_node)`

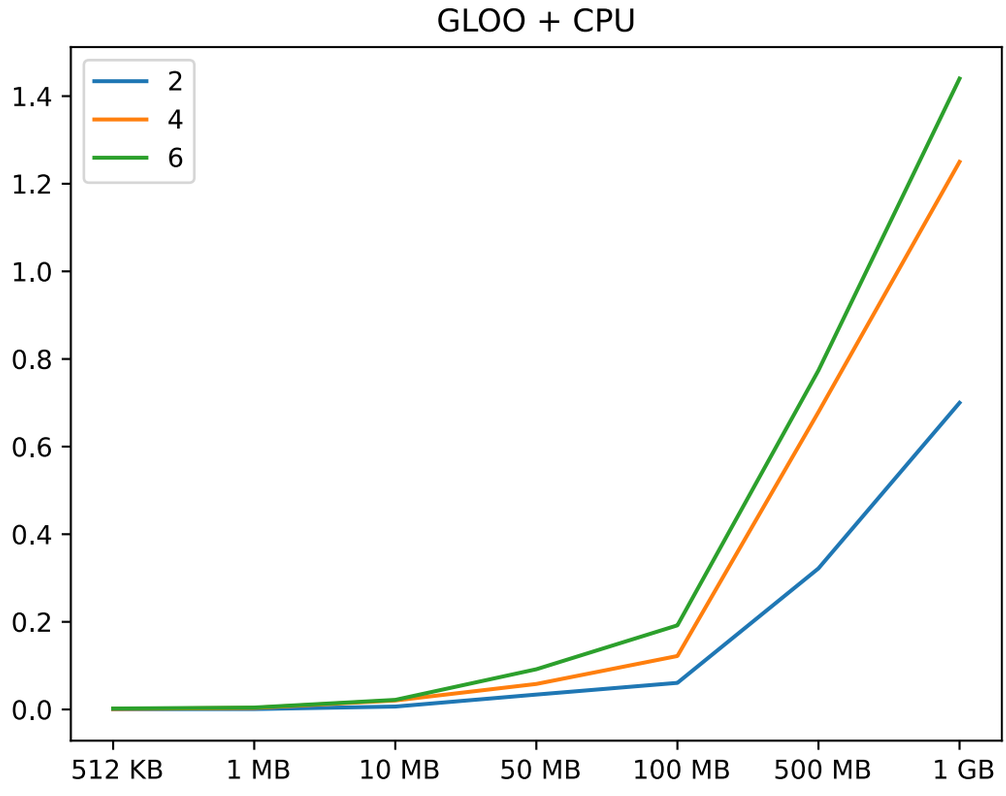
The script does the required all-reduces, and makes sure to synchronize the processes before the allreduce operation.

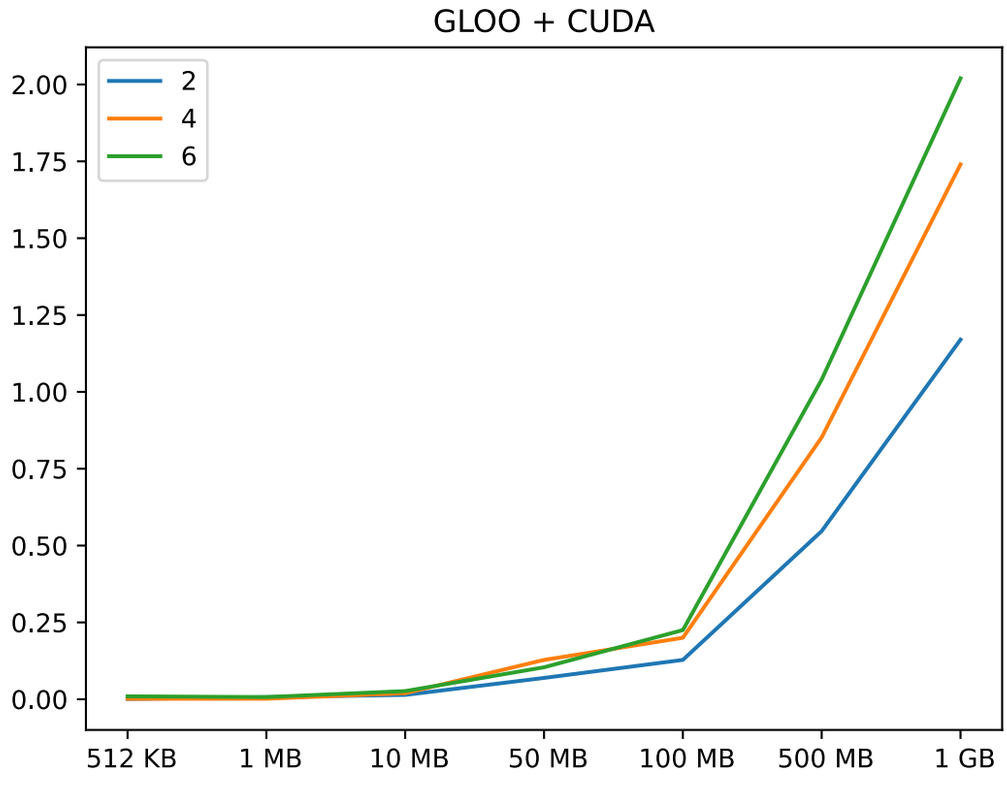
Backend + Device	Data size	Processes	Mean Time
GLOO + CPU	512 KB	2	$8.33 \cdot 10^{-4}$s
GLOO + CPU	512 KB	4	$2.52 \cdot 10^{-3}$s
GLOO + CPU	512 KB	6	$8.45 \cdot 10^{-3}$s
GLOO + CPU	1 MB	2	$1.56 \cdot 10^{-3}$s
GLOO + CPU	1 MB	4	$9.45 \cdot 10^{-3}$s
GLOO + CPU	1 MB	6	$1.86 \cdot 10^{-2}$s
GLOO + CPU	10 MB	2	$6.09 \cdot 10^{-3}$s
GLOO + CPU	10 MB	4	$3.46 \cdot 10^{-2}$s
GLOO + CPU	10 MB	6	$2.71 \cdot 10^{-2}$s
GLOO + CPU	50 MB	2	$3.81 \cdot 10^{-2}$s
GLOO + CPU	50 MB	4	$6.96 \cdot 10^{-2}$s
GLOO + CPU	50 MB	6	$8.59 \cdot 10^{-2}$s
GLOO + CPU	100 MB	2	$6.72 \cdot 10^{-2}$s
GLOO + CPU	100 MB	4	$1.34 \cdot 10^{-1}$s
GLOO + CPU	100 MB	6	$1.69 \cdot 10^{-1}$s
GLOO + CPU	500 MB	2	$2.42 \cdot 10^{-1}$s
GLOO + CPU	500 MB	4	$6.37 \cdot 10^{-1}$s
GLOO + CPU	500 MB	6	$8.14 \cdot 10^{-1}$s
GLOO + CPU	1 GB	2	$7.62 \cdot 10^{-1}$s
GLOO + CPU	1 GB	4	1.29 s
GLOO + CPU	1 GB	6	1.68 s
GLOO + CUDA	512 KB	2	$2.74 \cdot 10^{-1}$s
GLOO + CUDA	512 KB	4	$4.56 \cdot 10^{-1}$s
GLOO + CUDA	512 KB	6	$8.41 \cdot 10^{-1}$s
GLOO + CUDA	1 MB	2	$1.88 \cdot 10^{-1}$s
GLOO + CUDA	1 MB	4	$4.62 \cdot 10^{-1}$s
GLOO + CUDA	1 MB	6	$7.82 \cdot 10^{-1}$s
GLOO + CUDA	10 MB	2	$1.97 \cdot 10^{-1}$s

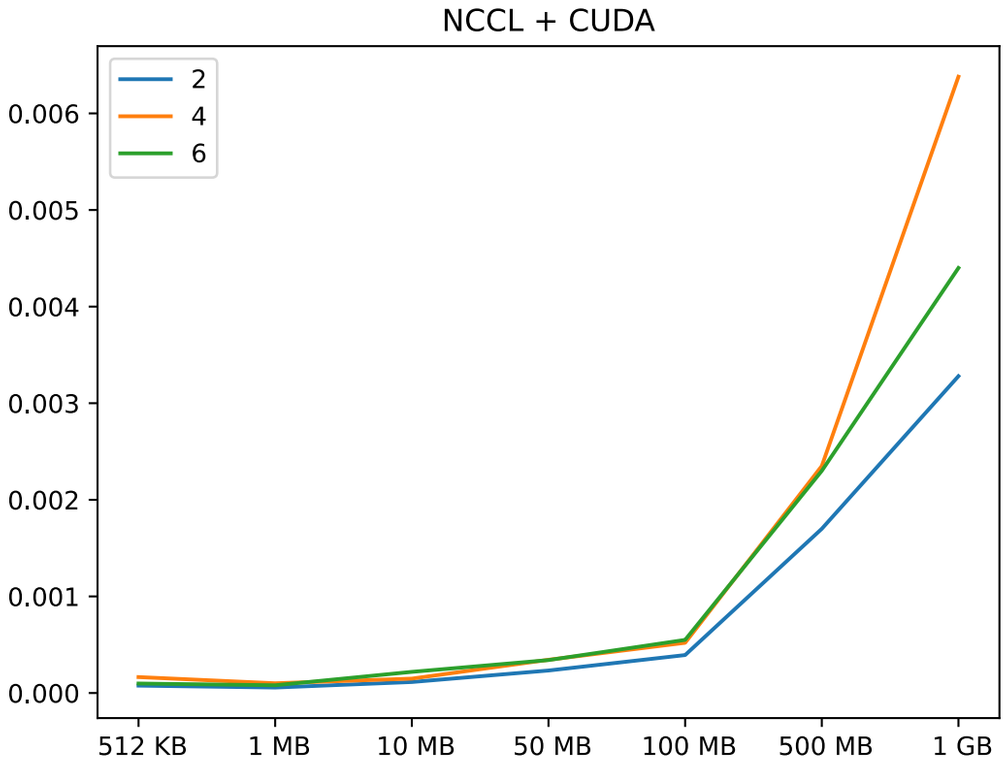
Backend + Device	Data size	Processes	Mean Time
GLOO + CUDA	10 MB	4	$5.62 \cdot 10^{-1}$s
GLOO + CUDA	10 MB	6	$7.70 \cdot 10^{-1}$s
GLOO + CUDA	50 MB	2	$2.34 \cdot 10^{-1}$s
GLOO + CUDA	50 MB	4	$6.10 \cdot 10^{-1}$s
GLOO + CUDA	50 MB	6	$8.91 \cdot 10^{-1}$s
GLOO + CUDA	100 MB	2	$3.71 \cdot 10^{-1}$s
GLOO + CUDA	100 MB	4	$6.68 \cdot 10^{-1}$s
GLOO + CUDA	100 MB	6	1.10 s
GLOO + CUDA	500 MB	2	$6.80 \cdot 10^{-1}$s
GLOO + CUDA	500 MB	4	1.35 s
GLOO + CUDA	500 MB	6	1.80 s
GLOO + CUDA	1 GB	2	1.40 s
GLOO + CUDA	1 GB	4	2.29 s
GLOO + CUDA	1 GB	6	3.02 s
NCCL + CUDA	512 KB	2	$2.38 \cdot 10^{-4}$s
NCCL + CUDA	512 KB	4	$2.18 \cdot 10^{-4}$s
NCCL + CUDA	512 KB	6	$2.25 \cdot 10^{-4}$s
NCCL + CUDA	1 MB	4	$2.58 \cdot 10^{-4}$s
NCCL + CUDA	1 MB	6	$1.39 \cdot 10^{-3}$s
NCCL + CUDA	10 MB	2	$2.49 \cdot 10^{-4}$s
NCCL + CUDA	10 MB	4	$2.39 \cdot 10^{-4}$s
NCCL + CUDA	10 MB	6	$1.25 \cdot 10^{-3}$s
NCCL + CUDA	50 MB	2	$2.20 \cdot 10^{-4}$s
NCCL + CUDA	50 MB	4	$2.31 \cdot 10^{-4}$s
NCCL + CUDA	50 MB	6	$2.53 \cdot 10^{-3}$s
NCCL + CUDA	100 MB	2	$2.22 \cdot 10^{-4}$s
NCCL + CUDA	100 MB	4	$2.35 \cdot 10^{-4}$s
NCCL + CUDA	100 MB	6	$6.46 \cdot 10^{-3}$s
NCCL + CUDA	500 MB	2	$2.41 \cdot 10^{-4}$s
NCCL + CUDA	500 MB	4	$2.30 \cdot 10^{-4}$s
NCCL + CUDA	500 MB	6	$6.15 \cdot 10^{-4}$s
NCCL + CUDA	1 GB	2	$2.57 \cdot 10^{-4}$s
NCCL + CUDA	1 GB	4	$2.42 \cdot 10^{-4}$s
NCCL + CUDA	1 GB	6	$1.48 \cdot 10^{-3}$s

Clearly GLOO with CPU and GPU are fairly similar in performance, and we notice that the reductions take a lot of time compared to NCCL. Somehow the GLOO CUDA performance is worse than the GLOO CPU performance, likely because there there is no data movement across devices on CPU, since one process can access the memory of another process. Adding more processes increases the time taken for the allreduce in an

expected fashion. NCCL runs incredibly fast, and scales well even to 6 devices at 1GB. The plot below presents these results in an overview.







For GLOO we see that increasing from 2 to 4 devices increases the time taken by a much larger amount than increasing from 4 to 6 devices. This holds for both CPU and CUDA. In the NCCL case, we see an interesting effect where communicating between 4 GPUs seems to take longer than for 6 GPUs in the 1GB case. This might be the effect of noise in the measurements, and might have been improved by running more samples or being more careful with synchronization and warmups. It's also possible that the kernels used for NCCL communication are more optimized for the 6 GPU case than the 4 GPU case. This might have something to do with the topology of the interconnect on the machine, since GPUs 0-3 have differing CPU affinity and potentially NIC connections compared to GPUs 4-5 (or 4-7).

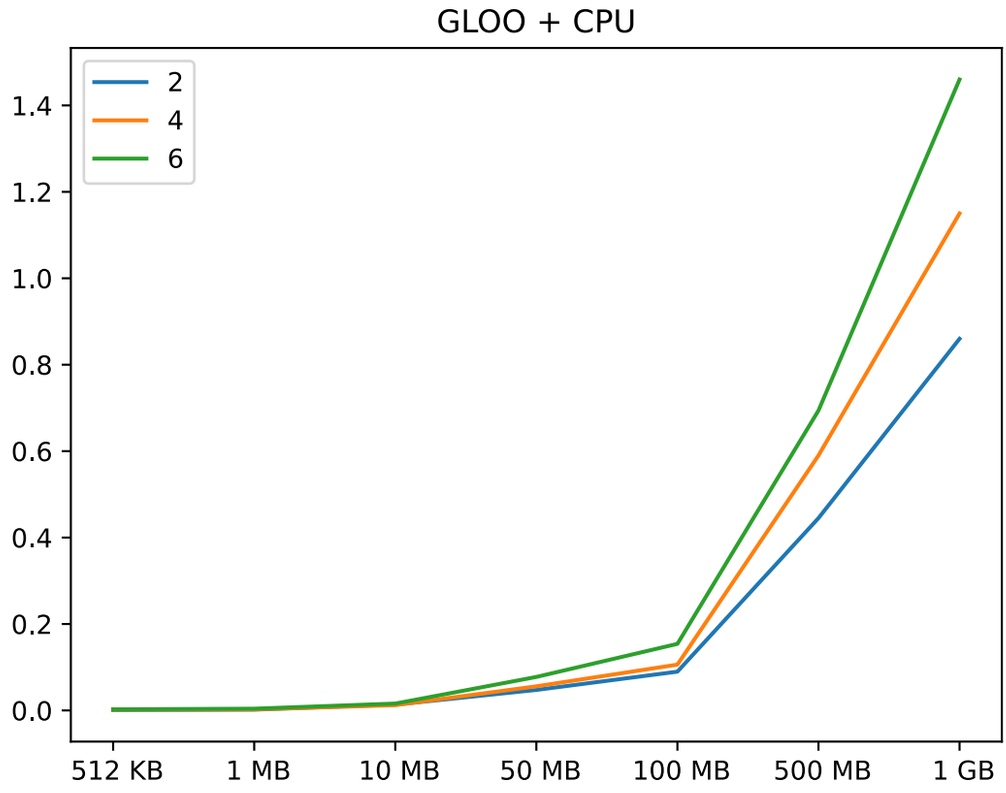
3.2. Multi-node Distributed Communication in PyTorch

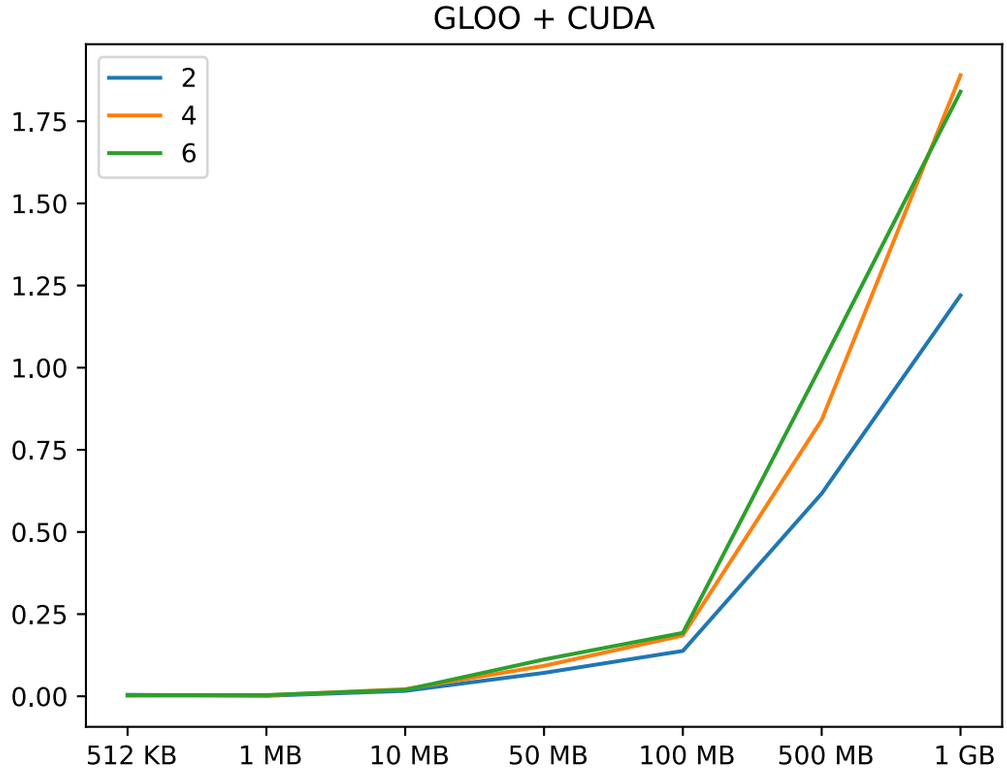
3.2.1. `(distributed_communication_multi_node)`

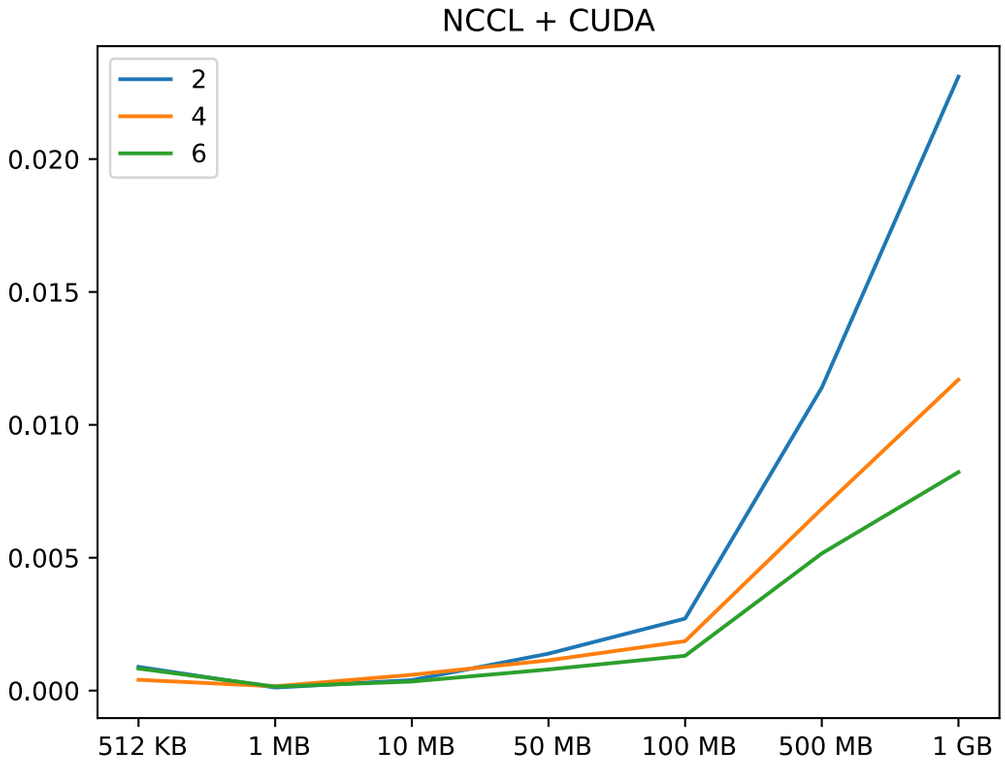
Performing the same benchmark, but splitting the workers across two nodes, we get the following results:

Backend + Device	Data size	Processes	Mean Time
Test			

Plotting these in the same way:


GLOO + CPU
2
4
6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
512 KB
1 MB
10 MB
50 MB
100 MB
500 MB
1 GB


GLOO + CUDA
2
4
6
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
512 KB
1 MB
10 MB
50 MB
100 MB
500 MB
1 GB



Again we see dramatic differences between GLOO and NCCL. The GLOO CUDA performance is around 2 orders of magnitude slower than the NCCL CUDA run. In this plot we also see similar strange effects for the NCCL run, where the 6 GPU case is the fastest of the three. The reasons stated in the previous tasks could also apply here. In particular, I think there are optimizations for >4 (usually 8) GPUs that have not been implemented for the 2 and 4 GPU cases, and are kicking in during our testing.

3.3. A naïve implementation of distributed data parallel training

3.3.1. `(naive_ddp)`

The DDP training script is in `ddp.py`, found in the function `ddp_train`. Relating to the later tasks it can take a parameter to check that the results agree with the non-ddp training, and another to flatten the parameters to batch the allreduce.

3.3.2. `(naive_ddp_benchmarking)`

My benchmarking setup initializes the same model for each rank, and makes sure that each rank gets the right part of the batch we are working on. Then, 5 steps are ran, with only the final step being timed. We make sure to synchronize CUDA devices around timing points, and check that the results are correct outside of timing. Result follow below:

Size	Single Node	Two Nodes
small	$4.14 \cdot 10^{-2}$s	$4.24 \cdot 10^{-2}$s
medium	$7.57 \cdot 10^{-2}$s	$1.02 \cdot 10^{-1}$s
large	$1.27 \cdot 10^{-1}$s	$1.94 \cdot 10^{-1}$s
xl	$2.01 \cdot 10^{-1}$s	$3.34 \cdot 10^{-1}$s
2.7B	$2.24 \cdot 10^{-1}$s	$4.48 \cdot 10^{-1}$s

3.4. Improving upon the minimal DDP implementation

3.4.1. `(minimal_ddp_flat_benchmarking)`

Size	Single Node	Two Nodes
small	$3.69 \cdot 10^{-2}$	$3.94 \cdot 10^{-2}$ s
medium	$7.09 \cdot 10^{-2}$	$9.40 \cdot 10^{-2}$ s
large	$1.22 \cdot 10^{-1}$	$1.78 \cdot 10^{-1}$ s
xl	$1.95 \cdot 10^{-1}$	$3.01 \cdot 10^{-1}$ s
2.7B	$2.29 \cdot 10^{-1}$	$4.18 \cdot 10^{-1}$ s

Flattening seems to have a slight effect, giving us better performance, especially in the two-node case. This is likely happening because of the reduce number of communication calls, even though the total amount of data is the same.

3.4.2. `(minimal_ddp_benchmarking)`

The original script was given an argument to add this batching functionality, and the results of retiming are as follows:

Size	Single Node	Two Nodes
small	$3.66 \cdot 10^{-2}$s	$3.82 \cdot 10^{-2}$s
medium	$7.10 \cdot 10^{-2}$s	$9.27 \cdot 10^{-1}$s
large	$1.24 \cdot 10^{-1}$s	$1.78 \cdot 10^{-1}$s
xl	$1.97 \cdot 10^{-1}$s	$3.09 \cdot 10^{-1}$s
2.7B	$2.30 \cdot 10^{-1}$s	$4.13 \cdot 10^{-1}$s

3.4.3. `(ddp_overlap_individual_parameters_benchmarking)`

(a) Benchmarking the DDP implementation with overlapping backwards pass yields this table of results:

Size	Single Node	Two Nodes
small	$4.55 \cdot 10^{-2}$s	$4.91 \cdot 10^{-2}$s
medium	$9.03 \cdot 10^{-2}$s	$8.71 \cdot 10^{-2}$s
large	$1.39 \cdot 10^{-1}$s	$1.54 \cdot 10^{-1}$s
xl	$2.16 \cdot 10^{-1}$s	$2.81 \cdot 10^{-1}$s
2.7B	$2.19 \cdot 10^{-1}$s	$4.14 \cdot 10^{-1}$s

(b)

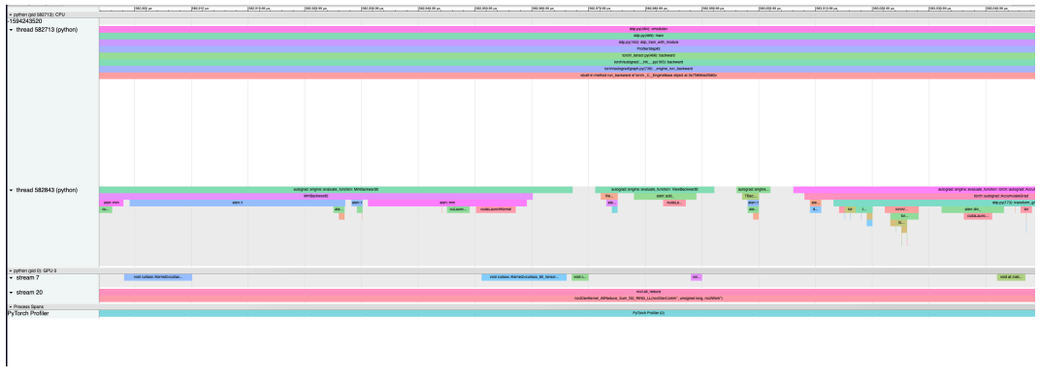


Figure 10: Overlapping communication and computation in the backward pass. The two GPU streams overlap.

Figure 11: Nonoverlapping communication and computation in the backward pass. The two streams on GPU are clearly separated.

(c) Running the Holistic Trace Analysis, we get a dataframe showing the overlap percentage for each GPU.

For the benchmark, rank 0 has overlap 12.54%, and rank 1 has 9.10%. Using naïve DDP shows us an overlap of 0% for both ranks as expected.

3.4.4. `(ddp_bucketed_benchmarking)`

(a) Running the bucketed DDP implementation with different bucket sizes and backends gives us the following:

Backend	Model Size	Bucket Size	Time
GLOO	small	5	323 ms
GLOO	small	10	335 ms
GLOO	small	50	306 ms
GLOO	small	100	319 ms
GLOO	small	500	338 ms
GLOO	medium	5	1.21 s
GLOO	medium	10	1.16 s
GLOO	medium	50	1.12 s
GLOO	medium	100	1.06 s
GLOO	medium	500	1.06 s
GLOO	large	5	2.39 s
GLOO	large	10	2.59 s

Backend	Model Size	Bucket Size	Time
GLOO	large	50	2.17 s
GLOO	large	100	2.59 s
GLOO	large	500	2.42 s
NCCL	small	5	36.1 ms
NCCL	small	10	36.6 ms
NCCL	small	50	45.1 ms
NCCL	small	100	44.9 ms
NCCL	small	500	37.1 ms
NCCL	medium	5	72.5 ms
NCCL	medium	10	71.7 ms
NCCL	medium	50	72.0 ms
NCCL	medium	100	72.3 ms
NCCL	medium	500	72.3 ms
NCCL	large	5	128 ms
NCCL	large	10	128 ms
NCCL	large	50	130 ms
NCCL	large	100	128 ms
NCCL	large	500	128 ms

It seems like the bucket size matters somewhat for GLOO, but has a negligible effect for NCCL. The bucket size of 50 seems to perform the best with GLOO. The small difference between these times might be due to the relatively short amount of time spent on communication compared to the computation time.

(b) Let s be the size of the model parameters, w be the bandwidth of the all-reduce algorithm, o the overhead of each communication call and n_b the number of buckets. Then each bucket will have size $\frac{s}{n_b}$, and the total time taken to communicate this bucket will be $o + \frac{s}{n_b}w$. The time to compute all buckets will be $\frac{s}{w}$, but $\frac{n_b-1}{n_b}$ of these will overlap with communication. This gives us a total time of

$$t = n_b\left(o + \frac{s}{n_b w}\right) - \frac{n_b - 1}{n_b}\frac{s}{w} = n_b o + \frac{s}{n_b w}$$

Minimizing, we get that

$$\frac{\partial}{\partial n_b}t = o - \frac{s}{n_b^2 w} = 0 \Rightarrow n_b = \sqrt{\frac{s}{wo}} \Rightarrow \frac{s}{n_b} = \sqrt{ows}.$$

3.4.5. (optimizer_state_sharding_accounting)

(a) The script can be found the `ddp.py` file. It measures the peak memory usage of each rank (this usage is similar across the two ranks for 2 GPUs, in part due to the way I decided on allocating the memory to different ranks). Without or without sharding, the peak memory usage after model initialization is 5.69 GiB. Right before the optimizer step, the memory usage is 11.57 GiB both with and without memory sharding. After the optimizer step, the peak memory usage is 23.01 GiB without sharding, and 17.38 GiB for rank 0 and 17.26 GiB for rank 1 with sharding. Breaking this down, we see that the gradients are 5.55 GiB and the parameters take up another 5.55 GiB. The optimizer state takes up 11.11 GiB without optimizer sharding, but only 5.49/5.61 GiB with sharding over two GPUs.

(b) On 1 node x 2 GPUs we see the following times for each model size:

Size	Not Sharded	Sharded
small	46.5 ms	45.0 ms
medium	90.1 ms	86.3 ms
large	137 ms	141 ms
xl	202 ms	198 ms
2.7B	203 ms	209 ms

and for 2 nodes x 1 GPU, the table looks like:

Size	Not Sharded	Sharded
small	40.6 ms	40.7 ms
medium	7.93 ms	84.6 ms
large	123 ms	159 ms
xl	192 ms	288 ms
2.7B	197 ms	344 ms

The overhead of using a sharded optimizer is close to zero for all model sizes when running on a single machine with two GPUs. This is as expected since ZeRO stage 1 is known to be a free memory optimization. However, we notice that the difference between sharding and not sharding the optimizer state is quite pronounced for the 2 nodes x 1 GPU case. This might be because the sharding is suboptimal, or that gradients aren't ready in the right order for the shards to be transmitted between devices. With some more optimization of this implementation we should be able to match the runtime performance of the non-sharded optimizer state.

(c) The key differences between my implementation of optimizer state sharding and the ZeRO stage 1 approach are:

- The granularity of sharding is per-parameter tensor in my approach, while the ZeRO stage 1 approach potentially might shard per parameter entry.
- The communication volume is the same between the two approaches, but my implementation uses broadcasts instead of all-gather operations. This means my implementation performs a number of broadcasts equal to the number of parameters, while the ZeRO stage 1 approach can do a single all-gather operation. Since my implementation is asychronous, I wouldn't expect this to matter too much.