stanford336

echen333

July 31, 2025

1 Assignment Overview

1.1 Profiling and Benchmarking

Problem : bechmarking_script

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

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

Solution

(a) We list our measured times on an H100 for the 5 different model sizes in Table 1. The standard deviation of run times in each run is actually pretty small (as seen in the table), though the mean time can differ significantly across runs.

(b) Without any warm-up steps, the first measurement time is an order of magnitude larger than other steps. This is probably because it takes some time to load the large model into memory, CUDA context initialization, and some things under the hood. With 1 and 2 warm-up steps, the difference seem a lot smaller, but there might still be different due to caching and memory-management, e.g. the L1 cache fills up after 2 warm-up runs.

Model Size	Fwd Mean (s)	Fwd Std (s)	Bwd Mean (s)	Bwd Std (s)
Small	0.035	0.005	0.091	0.024
Medium	0.073	0.005	0.198	0.040
Large	0.097	0.010	0.247	0.058
XL	0.132	0.013	0.312	0.060
2.7B	0.079	0.004	0.193	0.027

Table 1: Latency statistics for forward and backward passes across model sizes

Problem : nsys_profile

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

(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? (Hint: look at the "CUDA GPU Kernel Summary" under "Stats Systems View", and filter using NVTX ranges to identify which parts of the model are responsible for which kernels.)

(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?

(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?

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

Solution

After much effort, I could not get this working on gpus from cloud providers. Most (runpod, vast.ai) don't allow you to configure kernel paranoid levels (see here), and the others (lambda) don't provide access to gpu clock control. This is understandable I guess since otherwise you could monitor other user's processes or mess with clock frequency?

Problem : mixed_precision_accumulation

Run the following code and commment on the (accuracy of the) results:

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

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

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

```
print(s)

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

Solution

We get:

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

We see that whether s is float16 vs float32 can lead to a decrease in accuracy. Example 4 also shows that upcasting the summand during summing leads to no difference.

Problem : bechmarking_mixed_precision

(a) Consider the following model. Suppose we are training the model on a GPU and 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 different components?

(b) 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?

(c) Modify your benchmarking script to optionally run the model using mixed precision with BF16. Time the forward and backward passes with and without mixed-precision for each language model size described in §1.1.2. Compare the results of using full vs. mixed precision, and comment on any trends as model size changes. You may find the nullcontext no-op context manager to be useful.

Solution

(a) The respective data types are:

- model parameters: float32
- output of first feed-forward: float16
- output of layer norm: float32
- logits: float16
- loss: float16
- gradients: float32

(b) The division by std is sensitive to mixed precision since we divide by standard deviation, which is very small and can underflow to zero. Changing it to BF16 helps since it can avoid overflow/underflow but still probably too imprecise given less mantissa bits.

(c) On an A100, we get the mean times listed in Table 2. We see that there is perhaps some very small overhead with using autocast for smaller models, but bfloat16 is much faster in both the forward and backward passes for the XL and 2.7B models.

Model Size	**Fwd Mean (s)**	**Fwd Mixed Mean (s)**	**Bwd Mean (s)**	**Bwd Mixed Mean (s)**
Small	0.028	0.031	0.061	0.062
Medium	0.057	0.060	0.124	0.135
Large	0.087	0.090	0.239	0.208
XL	0.149	0.122	0.452	0.292
2.7B	0.221	0.081	0.676	0.223

Table 2: Mean times for forward and backward passes with and without mixed precision.

Problem : memory_profiling

Profile your forward pass, backward pass, and optimizer step of the 2.7B model from Table 1 with context lengths of 128, 256 and 512.

(a) Add an option to your profiling script to run your model through the vmemory 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?

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

(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?

(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 10242).Deliverable: A 1-2 sentence response with your derivation.

(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?

Solution

(a) In Figure 1 and 2, we show the memory timelines of a 2.7B model. One figure is just for the forward pass and the second is for a full training step(forward pass, backward pass, and optimizer step). From the peeks in Figure 1, we can see that the forward pass first loads the model and data and memory then proceeds to slowly climb from storing the activations for each layer. The activations are then freed by the garbage collector.

From Figure 2, we also see the forward pass, but note that in the backwards pass, some activations are freed but we also need to allocate memory for our gradients. Finally, the optimizer step accumulates even more memory as we need 2 extra floats for each parameter.

(b) We show our peak memory usage results in Table 1.

(c) It seems that using mixed precision increases peak memory usage by less than 10% for both the forward pass or the full pass. For example, peak memory usage of a full pass goes from 55.246 GB to 61.998 GB.

(d) It would be of size $\frac{4bsq}{1024^2} = 5.0MB$ for $s = 128$ where b is batch size, s is sequence length, and q is d_model.

(e) The largest allocation is 100 MiB, which correspond to the each of the three weights in our Swiglu.

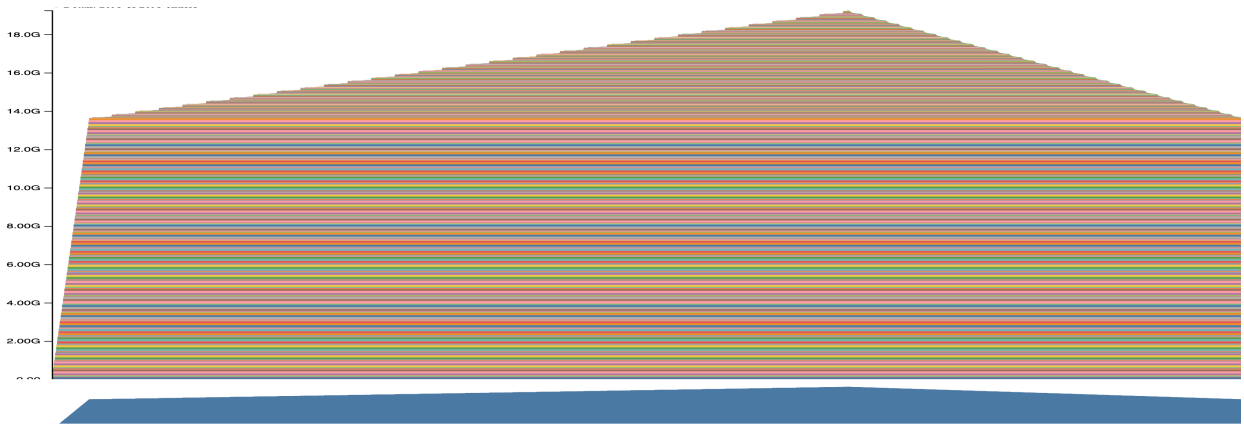


Figure 1: Memory profiling of only forward pass.

Context Length	Forward Pass (GB)	Full Pass (GB)
128	19.412	55.246
256	26.103	55.267
512	42.708	55.306

Table 3: Peak memory of each context length during forward pass and full step.

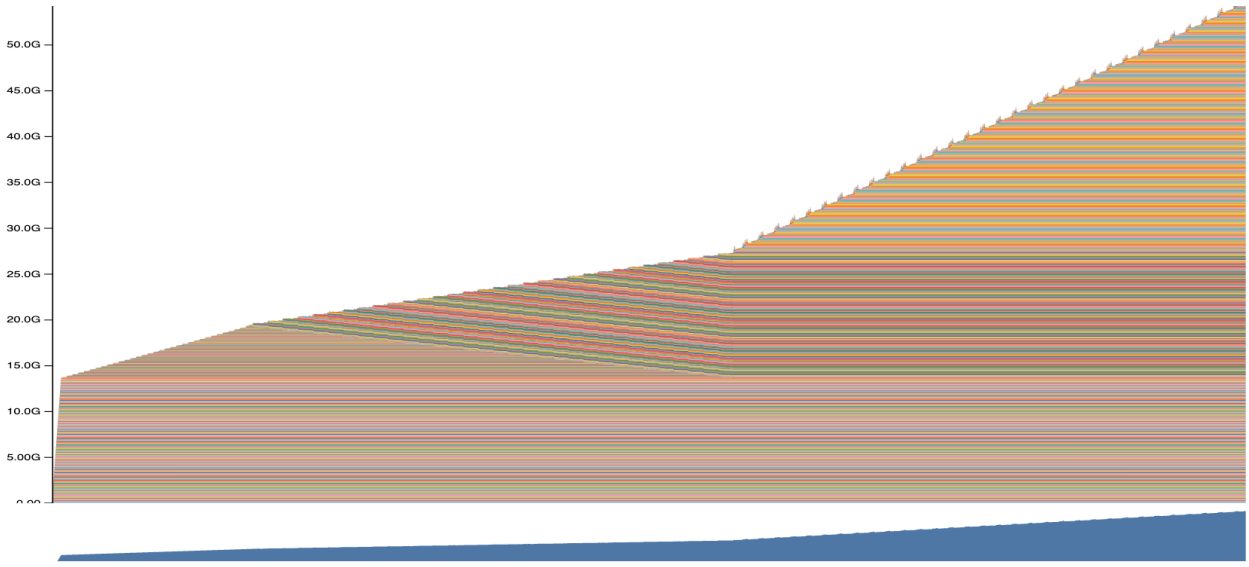


Figure 2: Memory profiling of forward pass, backward pass, and optimizer step.

Problem : pytorch_attention

Benchmark your attention implementation at different scales. Write a script that will:

- Fix the batch size to 8 and don't use multihead attention (i.e. remove the head dimension).
- Iterate through the cartesian product of [16, 32, 64, 128] for the head embedding dimension dmodel, and [256, 1024, 4096, 8192, 16384] for the sequence length.
- Create random inputs Q, K, V for the appropriate size.
- Time 100 forward passes through attention using the inputs.
- Measure how much memory is in use before the backward pass starts, and time 100 backward passes.
- Make sure to warm up, and to call torch.cuda.synchronize() after each forward/backward pass.

Report the timings (or out-of-memory errors) you get for these configurations. At what size do you get out-of-memory errors? Do the accounting for the memory usage of attention in one of the smallest configurations you find that runs out of memory (you can use the equations for memory usage of Transformers from Assignment 1). How does the memory saved for backward change with the sequence length? What would you do to eliminate this memory cost?
Deliverable: A table with your timings, your working out for the memory usage, and a 1-2 paragraph response.

Solution

We list our timings in Table 4. Not sure if this was done correctly but one dot product attention does give out of memory as we allocate at most 17GB. The memory usage is dominated by the self-attention matrix, which is given by $\frac{4bs^2}{1024^3} = 8GB$. But we have to double this since we also store the softmax, so 16GB total. The memory saved for backprop is quadratic in terms of seq length as a doubling in seq length quadruples mem allocated. We can halve the memory used by only storing the self-attention matrix and computing the softmax activation later for backprop.

Table 4: Attention Timings for Different Head Dimensions and Sequence Lengths

head_dim	seq	forward_time (s)	backward_time (s)	mem_allocated (GB)
16	256	0.000150	0.000545	0.038298
16	1024	0.000145	0.001260	0.136414
16	4096	0.000150	0.008449	1.149633
16	8192	0.000148	0.027077	4.379640
16	16384	0.000231	0.101105	17.282106
32	256	0.000143	0.000684	0.072377
32	1024	0.000146	0.001242	0.138511
32	4096	0.000143	0.007719	1.158022
32	8192	0.000142	0.028087	4.396418
32	16384	0.000222	0.106797	17.315660
64	256	0.000143	0.000744	0.073425
64	1024	0.000140	0.001559	0.142705
64	4096	0.000144	0.008877	1.174799
64	8192	0.000149	0.031139	4.429972
64	16384	0.000213	0.165870	17.382769
128	256	0.000151	0.000976	0.075523
128	1024	0.000141	0.002070	0.151094
128	4096	0.000146	0.017478	1.208353
128	8192	0.000150	0.062794	4.497081
128	16384	0.000222	0.240708	17.516987

Problem : torch_compile

(a) Extend your attention benchmarking script to include a compiled version of your PyTorch implementation of attention, and compare its performance to the uncompiled version with the same configuration as the pytorch_attention problem above.
Deliverable: A table comparing your forward and backward pass timings for your compiled attention module with the uncompiled version from the pytorch_attention problem above.

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

Deliverable: A table comparing your vanilla and compiled Transformer model.

Solution

(a) Our results are listed in Table 5.

(b) Our results are listed in Table 6.

Table 5: Compiled Attention Timings for Different Head Dimensions and Sequence Lengths

head_dim	seq	forward_time (s)	backward_time (s)	compiled_fwd_t (s)	compiled_bwd (s)
16	256	0.000151	0.000483	0.000102	0.000439
16	1024	0.000149	0.000947	0.000099	0.000707
16	4096	0.000147	0.004506	0.000105	0.003435
16	8192	0.000153	0.013160	0.000111	0.011679
16	16384	0.000240	0.045683	0.000120	0.042423
32	256	0.000165	0.000575	0.000099	0.000577
32	1024	0.000147	0.000981	0.000098	0.000980
32	4096	0.000146	0.004465	0.000105	0.004464
32	8192	0.000147	0.012589	0.000111	0.012579
32	16384	0.000224	0.048435	0.000124	0.045201
64	256	0.000140	0.000704	0.000103	0.000696
64	1024	0.000150	0.001418	0.000100	0.001420
64	4096	0.000145	0.006087	0.000104	0.006164
64	8192	0.000148	0.017773	0.000104	0.017369
64	16384	0.000221	0.109859	0.000120	0.109745
128	256	0.000142	0.000866	0.000099	0.000855
128	1024	0.000148	0.002173	0.000101	0.002169
128	4096	0.000150	0.012388	0.000099	0.009314
128	8192	0.000149	0.049469	0.000102	0.049545
128	16384	0.000221	0.186847	0.000119	0.189612

Table 6: End-to-end torch.compile Model Timings

size	compiled_fwd	compiled_bwd	fwd	bwd
small	0.003762	0.029213	0.021868	0.057893
med	0.010411	0.067801	0.043957	0.113304
large	0.025404	0.147252	0.066049	0.188688
xlarge	0.051192	0.287173	0.088227	0.325805
2.7B	0.067724	0.439231	0.083521	0.463938

Problem : flash_benchmarking

Write a benchmarking script using ‘triton.testing.do_bench‘ that compares the performance of your (partially) Triton implementation of FlashAttention-2 forward and backward passes with a regular PyTorch implementation (i.e., not using FlashAttention). Specifically, you will report a table that includes latencies for forward, backward, and the end-to-end forward-backward pass, for both your Triton and PyTorch implementations. Randomly generate any necessary inputs before you start benchmarking, and run the benchmark on a single H100. Always use batch size 1 and causal masking. Sweep over the cartesian product of sequence lengths of various powers of 2 from 128 up to 65536, embedding dimension sizes of various powers of 2 from 16 up to size 128, and precisions of torch.bfloat16 and torch.float32.

Solution

We list our results in Table 7. Unfortunately, we get CUDA out of memory on an H100 for sequence length 65536, probably because our triton backwards pass loads the entire $\mathbb{R}^{s \times s}$ self-attention matrix and mask in memory.

Table 7: Pytorch vs Flash Benchmark of forward, backward, and both passes

seq	emb_dim	mixed	py_fwd	trit_fwd	py_bwd	trit_bwd	py_both	trit_both
128	16	False	0.045545	0.066489	0.333232	0.272222	0.558249	0.452566
128	16	True	0.070118	0.062458	0.347733	0.322728	0.636037	0.502674
128	64	False	0.047957	0.065068	0.320361	0.273858	0.539307	0.479513
128	64	True	0.074854	0.064664	0.358402	0.332926	0.640823	0.522657
128	128	False	0.041334	0.069680	0.337906	0.288260	0.636626	0.490000
128	128	True	0.063855	0.073255	0.360612	0.342031	0.642625	0.534417
1024	16	False	0.059099	0.079310	0.342903	0.300703	0.600768	0.504497
1024	16	True	0.070023	0.079621	0.374110	0.349009	0.678567	0.555026
1024	64	False	0.059332	0.105782	0.537888	0.528735	0.829396	0.766735
1024	64	True	0.070420	0.105752	0.577603	0.520420	0.909049	0.768059
1024	128	False	0.068092	0.140362	0.671203	0.592038	0.990126	0.884724
1024	128	True	0.070278	0.140725	0.712403	0.656971	1.057425	0.942089
8192	16	False	1.332101	0.343029	3.450961	2.198964	4.773540	2.527205
8192	16	True	1.345900	0.346638	3.174937	2.236320	4.506846	2.554399
8192	64	False	1.514402	0.607018	4.471091	3.303576	5.951714	3.797207
8192	64	True	1.345400	0.611444	3.792449	2.816552	5.092928	3.382561
8192	128	False	1.845271	1.080108	6.823875	4.437626	8.339348	5.934016
8192	128	True	1.361437	1.085509	5.116917	3.276142	5.961788	4.324386
32768	16	False	19.358335	3.812200	47.867455	27.046816	66.631966	30.673334
32768	16	True	19.833612	3.813579	42.624306	28.237707	61.928768	31.808426
32768	64	False	22.717176	6.846011	61.844608	39.780624	79.440765	42.770973
32768	64	True	19.851099	6.801335	45.909889	29.491104	65.597374	36.143200
32768	128	False	28.331680	13.573898	71.030464	51.605377	99.893982	65.556160
32768	128	True	19.895416	13.555909	53.616833	36.282185	70.206978	45.486832

2 Distributed Data Parallel Training

Problem : distributed_communication_single_node

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, NCCL + GPU.
all-reduce data size: float32 data tensors ranging over 1MB, 10MB, 100MB, 1GB.
Number of processes: 2, 4, or 6 processes.
Resource requirements: Up to 6 GPUs. Each benchmarking run should take less than 5 minutes.

Solution

We list some benchmark runs below in Table 8, run on 8xTesla V100s. We find that array size have little influence on time on NCCL and instead n_procs seems to influence the time a lot more for these array sizes – probably due to communication overhead. We also note the significant difference between all-reduce on nccl vs gloo.

Table 8: End-to-end torch.compile Model Timings

size	n_procs	backend	mean_time
1.000000	2.000000	nccl	0.364509
1.000000	4.000000	nccl	0.578839
1.000000	6.000000	nccl	0.701945
10.000000	2.000000	nccl	0.372378
10.000000	4.000000	nccl	0.589919
10.000000	6.000000	nccl	0.703578
100.000000	2.000000	nccl	0.393590
100.000000	4.000000	nccl	0.613600
100.000000	6.000000	nccl	0.718957
1000.000000	2.000000	nccl	0.424514
1000.000000	4.000000	nccl	0.675741
1000.000000	6.000000	nccl	0.846845
1000.000000	2.000000	gloo	1.468517
1000.000000	4.000000	gloo	2.125806
1000.000000	6.000000	gloo	2.668937

Problem : naive_ddp_benchmarking

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 the single-node setting

(1 node x 2 GPUs) for the XL model size described in §1.1.2.

Solution

We train a BasicsTransformerLM model with the XL model size over 50 steps with a context length on 2 H100s. Because of memory constraints from the model and the optimizer states, we use a context length and a total batch size of 32. We find average time spent communicating gradients per step is 0.0681s and average time per step is 0.846s.

Problem : minimal_ddp_flat_benchmarking

Modify your minimal DDP implementation to communicate a tensor with flattened gradients from all parameters. Compare its performance with the minimal DDP implementation that issues an allreduce for each parameter tensor under the previously-used conditions (1 node x 2 GPUs, XL model size as described in §1.1.2).

Solution

We measured 0.0967s spent communicating the gradients and 0.898s for each model step, on average. This is actually slower than the naive ddp, probably since the time collecting and allocating all the gradients is greater than the extra communication time.

Problem : 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 with our previously-studied settings (the minimal DDP implementation that either issues an all-reduce for each parameter tensor, or a single all-reduce on the concatenation of all parameter tensors) with the same setup: 1 node, 2 GPUs, and the XL model size described in §1.1.2.
Deliverable: The measured time per training iteration when overlapping the backward pass with communication of individual parameter gradients, with 1-2 sentences comparing the results.

(b) Instrument your benchmarking code (using the 1 node, 2 GPUs, XL model size setup) with the Nsight profiler, comparing between the initial DDP implementation and this DDP implementation that overlaps backward computation and 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 initial DDP implementation, and another from this DDP implementation that overlaps compute with communication) that visually show that communication is or isn't overlapped with the backward pass.

Solution

(a) Over 50 steps, we now get that overlapping takes 0.796s per iteration while communication of individual parameters gradients takes 0.776s per iteration.

(b) Yes would be useful to instrument, but yea could not get nsight profiler working on gpu cloud providers.

Problem : ddp_bucketed_benchmarking

(a) Benchmark your bucketed DDP implementation using the same config as the previous experiments (1 node, 2 GPUs, XL model size), varying the maximum bucket size (1, 10, 100, 1000 MB). Compare your results to the previous experiments without bucketing—do the results align with your expectations? If they don't align, why not? You may have to use the PyTorch profiler as necessary to better understand how communication calls are ordered and/or executed. What changes in the experimental setup would you expect to yield results that are aligned with your expectations?
Deliverable: Measured time per training iteration for various bucket sizes. 3-4 sentence commentary about the results, your expectations, and potential reasons for any mismatch.

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

Solution

(a) We get: {Individual: 0.86, Bucket 1MB: 0.973, Bucket 10MB: 0.8596, Bucket 100MB: 0.874, Bucket 1000MB: OOM} This generally aligns with our expectations – buckets too small or too large slow down training iteration time as too small of buckets is just overhead on individual and too large is same as all. I guess I would've expected a little better from using 10MB buckets since it seems to compress together two small layers (norms) and one big layer that are strung together. This may be because our gpu's are more compute bound so we aren't really doing that much communication and also we are only using 2 gpu's so all reduce time is a lot faster.

(b) We get $\text{nb} \times \text{o} + \frac{\text{tot\_size/nb}}{\text{bandwidth}}$ where $\text{nb} \times \text{o}$ represents total communication call overhead and $\frac{\text{tot\_size/nb}}{\text{bandwidth}}$ represents time to one bucket finish. Thus,

$$\begin{aligned}
\text{nb} &= \arg\min_{\text{nb}} \text{nb} \times \text{o} + \frac{\text{tot_size/nb}}{\text{bandwidth}} \\
&\implies \text{o} - \frac{\text{tot_size}}{\text{bandwidth}} \frac{1}{\text{nb}^2} = 0 \\
&\implies \text{nb} = \sqrt{\frac{\text{tot_size}}{\text{bandwith} \times \text{o}}}.
\end{aligned}$$

Problem : communication_accounting

Consider a new model config, XXL, with d_model=16384, d_ff=53248, and num_blocks=126. Because for very large models, the vast majority of FLOPs are in the feedforward networks, we make some simplifying assumptions. First, we omit attention, input embeddings, and output linear layers. Then, we assume that each FFN is simply two linear layers (ignoring the activation function), where the first has input size d_model and output size d_ff, and the second has input size d_ff and output size d_model. Your model consists of num_blocks blocks of these two linear layers. Don't do any activation checkpointing, and keep your activations and gradient communications in BF16, while your accumulated gradients, master weights and optimizer state should be in FP32.

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

(b) Now assume your master weights, optimizer state, gradients and half of your activations (in practice every second layer) are sharded across NFSDP devices. Write an expression for how much memory this would take per device. What value does NFSDP need to be for the total memory cost to be less than 1 v5p TPU (95GB per device)?
Deliverable: Your calculations and a one-sentence response.

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

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

Solution

(a) We get num_params = d_model $*$ d_ff $*$ 2 $*$ num_blocks $= 2.2 \cdot 10^{11}$ and storing the gradients, weights, and optimizer states takes $4 * 4 *$ num_params $= 3276$ GiB in total. The backward in BF16 would only store the activations per layer: $2 *$ num_blocks $*$ (d_ff + d_model) $= 0.016$ GiB for batch=1. Thus, it would take 41 H100s.

(b) Storing half the activations would take num_blocks $\times$ d_model $\times$ batch $=$ $(0.002 *$ batch) GiB. Sharded across n devices, each device would need

$$\frac{3276}{n} + (0.002 * \text{batch}) < 95 \implies \text{n} > 34 \text{ tpu}'\text{s}.$$

(c) We attempt to equate total compute time per layer with total communication time per

layer. Considering only the forward pass, the compute time is

$$\frac{\text{total\_FLOPs}}{\text{C}} = \frac{2\text{d\_model} \cdot \text{d\_ff} \cdot \text{B}}{\text{C}} = 3.8 * 10^{-6}\text{B}.$$

From the book, the total memory time consists of sharing activations (TP) and model weights (FSDP) is:

$$\text{T}_{\text{FSDP}} = \frac{2 \cdot 2 \cdot \text{D} \cdot \text{F}}{\text{Y} \cdot \text{W}_{\text{ici}} \cdot \text{M}_{\text{x}}}$$
$$\text{T}_{\text{MP}} = \frac{2 \cdot 2 \cdot \text{B} \cdot \text{D}}{\text{X} \cdot \text{W}_{\text{ici}} \cdot \text{M}_{\text{Y}}}.$$

Then,

$$\text{T}_{\text{comms}} = \text{T}_{\text{FSDP}} + \text{T}_{\text{MP}} = 4 \cdot \frac{\text{D}}{\text{W}_{\text{ici}}} \left(\frac{\text{F}}{2\text{Y}} + \frac{\text{B}}{\text{X}} \right) > 3.8 * 10^{-6}\ \text{B} \implies \text{B} < 641.$$

(d) One idea is to increase the context length such that total tokens processed per pass is constant.

3 Optimizer State Sharding

Problem

(a) Create a script to profile the peak memory usage when training language models with and without optimizer state sharding. Using the standard configuration (1 node, 2 GPUs, XL model size), 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.).
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.

(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 the standard configuration (1 node, 2 GPUs, XL model size).
Deliverable: 2-3 sentence response with your timings.

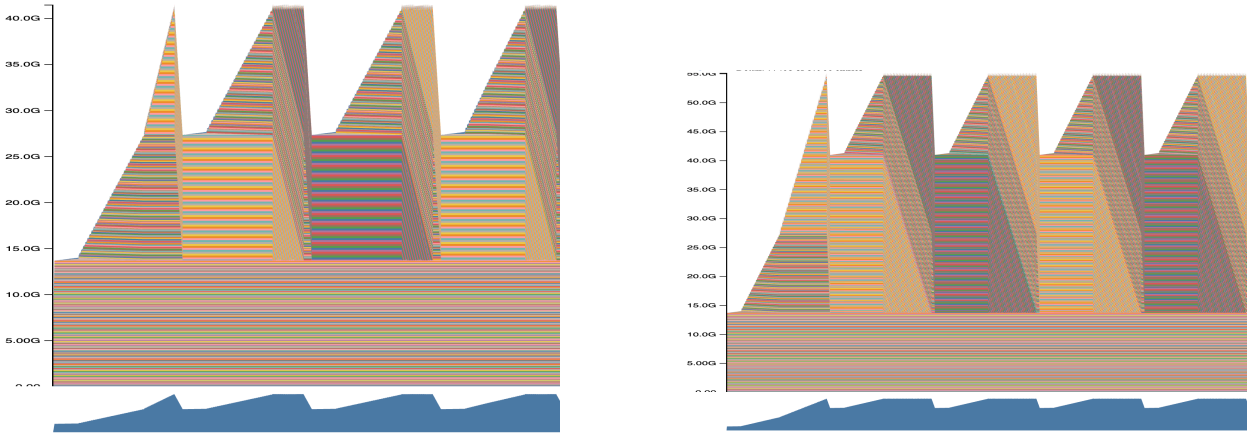
(c) How does our approach to optimizer state sharding differ from ZeRO stage 1 (described as ZeRODP Pos in Rajbhandari et al., 2020)?
Deliverable: 2-3 sentence summary of any differences, especially those related to memory and communication volume.

Solution

(a) Our model profiles are shown below. With optimizer sharding, we get a peak memory usage of 42GB, or 4 * total parameter size (model params(1), gradients (1), optimizer state (1)) plus a bit for storing activations. Without sharding, we get 55GB peak memory usage, with an extra 13gb for the other half of optimizer states.

(b) Over 100 epochs, we find that with sharding, we get 0.249s per epoch versus 0.261s per epoch without sharding. Implementation matters here since the speedup is probably from not having to step through all the parameters and instead broadcasting them asynchronously. It would be slower if async_op = False.

(c) We used a broadcast instead of an all-gather, which could give some better memory efficiency and speedups. Otherwise it seems pretty similar since we also keep $4\Psi + \frac{K\Psi}{n_d}$ floats.



(a) With optimizer sharding

(b) Without optimizer sharding

Figure 3: Memory profiles