Problem (benchmarking_script): 4 points

Answer:
Benchmarking is done on A100
a) See benchmarking_script.py
b)
Results are sum of 10 steps
Warm up steps = 5

Size	Mean forward only (millisecond)	STD forward only	Mean forward and backward (millisecond)	STD Forward and backward
Small	377	8	818	5
medium	759	5	2144	8
large	1420	0.7	4377	1.2
xl	2821	1.5	8893	1.7
2.7B	4169	0.4	13212	0.6

c) Warm up steps = 0

Size	Mean forward only (millisecond)	STD forward only	Mean forward and backward (millisecond)	STD Forward and backward
Small	578	270	1063	332
medium	1015	266	2432	314
large	1652	327	4623	359
xl	3034	299	9121	316
2.7B	4358	266	13444	316

(STD is very high, no surprise)

Warm up steps = 1

Size	Mean forward	STD forward	Mean forward	STD

	only (millisecond)	only	and backward (millisecond)	Forward and backward
Small	377	3.8	823	4.7
medium	811	37.9	2185	45
large	1419	1.3	4377	3.3
xl	2819	1.1	8905	0.6
2.7B	4169	0.5	13215	0.6

(the standard deviation is surprisingly high when size = medium, when warm up steps = 1)

Problem (nsys_profile): 5 points

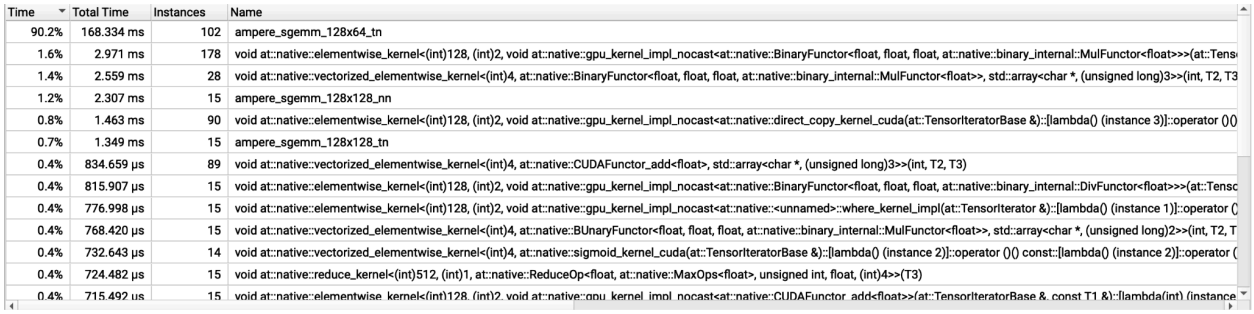
a) I've only profiled context_length = 256, which is what I used in previous experiment

Size	Forward pass, 10 steps (in milliseconds)
Small	470
medium	950
large	1480
xl	1937
2.7B	1891

(Interestingly, forward pass time of small & medium size model is somewhat larger than measured with Python standard library; large size model is about the same; xl and 2.7B size model forward pass time is significantly less than measured with Python standard library. I actually don't have a good explanation for that)

b) On the 2.7B model, the CUDA kernel takes the most cumulative GPU time during the forward pass is "ampere_sgemm_128x64_tn", it was called 103 times. In the backward pass, the kernel takes the most GPU time is "ampere_sgemm_128x128_nt"

c)



Time	Total Time	Instances	Name
90.2%	168.334 ms	102	ampere_sgemm_128x64_tn
1.6%	2.971 ms	178	void at::native::elementwise_kernel<(int)128, (int)2, void at::native::gpu_kernel_impl_nocast<at::native::BinaryFunctor<float, float, float, at::native::binary_internal::MulFunctor<float>>>(at::Tens
1.4%	2.559 ms	28	void at::native::vectorized_elementwise_kernel<(int)4, at::native::BinaryFunctor<float, float, float, at::native::binary_internal::MulFunctor<float>>, std::array<char *, (unsigned long)3>>(int, T2, T3
1.2%	2.307 ms	15	ampere_sgemm_128x128_nn
0.8%	1.463 ms	90	void at::native::elementwise_kernel<(int)128, (int)2, void at::native::gpu_kernel_impl_nocast<at::native::direct_copy_kernel_cuda(at::TensorIteratorBase &)::[lambda() (instance 3)]::operator ()()
0.7%	1.349 ms	15	ampere_sgemm_128x128_tn
0.4%	834.659 µs	89	void at::native::vectorized_elementwise_kernel<(int)4, at::native::CUDAFunctor_add<float>, std::array<char *, (unsigned long)3>>(int, T2, T3)
0.4%	815.907 µs	15	void at::native::elementwise_kernel<(int)128, (int)2, void at::native::gpu_kernel_impl_nocast<at::native::BinaryFunctor<float, float, float, at::native::binary_internal::DivFunctor<float>>>(at::Tensc
0.4%	776.998 µs	15	void at::native::elementwise_kernel<(int)128, (int)2, void at::native::gpu_kernel_impl_nocast<at::native::<unnamed>::where_kernel_impl(at::TensorIterator &)::[lambda() (instance 1)]::operator ()
0.4%	768.420 µs	15	void at::native::vectorized_elementwise_kernel<(int)4, at::native::BUnaryFunctor<float, float, float, at::native::binary_internal::MulFunctor<float>>, std::array<char *, (unsigned long)2>>(int, T2, T
0.4%	732.643 µs	14	void at::native::vectorized_elementwise_kernel<(int)4, at::native::sigmoid_kernel_cuda(at::TensorIteratorBase &)::[lambda() (instance 2)]::operator ()() const::[lambda() (instance 2)]::operator (
0.4%	724.482 µs	15	void at::native::reduce_kernel<(int)512, (int)1, at::native::ReduceOp<float, at::native::MaxOps<float>, unsigned int, float, (int)4>>(T3)
0.4%	715.492 µs	15	void at::native::elementwise_kernel<(int)128, (int)2, void at::native::gpu_kernel_impl_nocast<at::native::CUDAFunctor_add<float>>(at::TensorIteratorBase &, const T1 &)::[lambda(int) (instance

d) Tested on small size model:
For forward pass only (inference), matrix multiplication accounts for 78% total time spent; vs 63% for the entire training step. Increase percentage of vectorized element wise kernel & element wise kernels in the training step.

e) On a small size model, the runtime for softmax within the self-attention layer of your model during a forward pass is 1.98 milliseconds, vs 10.66 milliseconds for matrix multiplications, the ratio is approximately 1:5; while the ratio for FLOPs is 1: 6 x 2 x d_model x d_model = 1: 7M

Problem (mixed_precision_accumulation): 1 point

s = torch.tensor(0,dtype=torch.float32)
for i in range(1000):
  s += torch.tensor(0.01,dtype=torch.float32)
print(s)
Result: tensor(10.0001). I'm actually surprised that float32 can't handle the precision of 4 decimal digits

s = torch.tensor(0,dtype=torch.float16)
for i in range(1000):
  s += torch.tensor(0.01,dtype=torch.float16)
print(s)
Result: `tensor(9.9531, dtype=torch.float16). Not surprising`

```
s = torch.tensor(0,dtype=torch.float32)
for i in range(1000):
    s += torch.tensor(0.01,dtype=torch.float16)
print(s)
Result: tensor(10.0021)

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)
Result: tensor(10.0021)
```

Problem (benchmarking_mixed_precision): 2 points

a)

• the model parameters within the autocast context,
    FP32
• the output of the first feed-forward layer (ToyModel.fc1),
    FP16
• the output of layer norm (ToyModel.ln),
    FP32
• the model's predicted logits,
    FP16
• the loss,
    FP32
• and the model's gradients?
    FP32

Explanation from ChatGPT: Why keep the loss and grads in fp32?
Reductions accumulate large sums; fp16's 1 ULP is too coarse and can swamp small gradient signals. Casting them to fp32 (or bf16 on supported HW) preserves training stability while still giving you the big speed & memory gains from fp16 activations and mat-mul kernels.

b)

The very large sum during layer norm could cause it to overflow, or very high variance could cause it underflow
Using bfloat16 won't change things in my experiment, autocast still use the output of fp32.

c)

A100 / MI300X

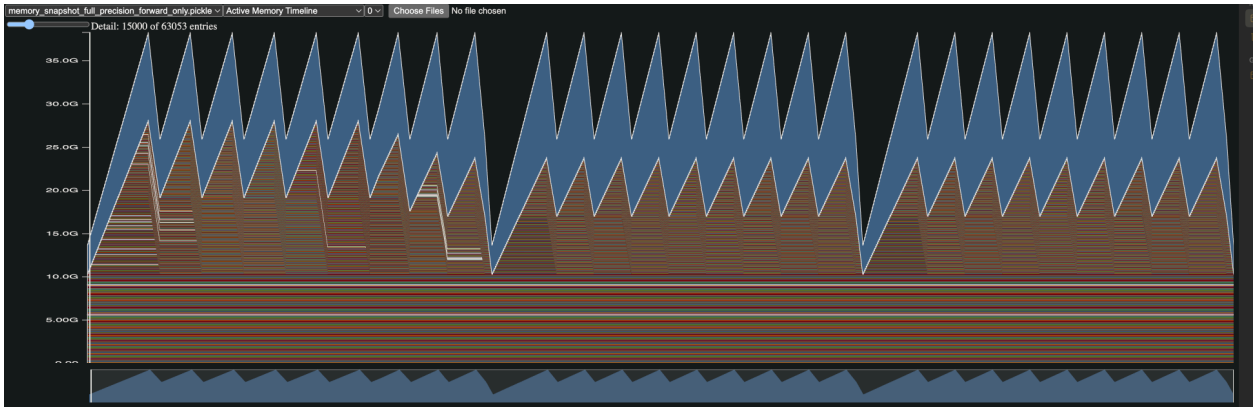
Size	Full precision (millisecond)	Mixed precision
Small	752 / 377	855 / 432
medium	2003 / 824	1727 / 847
large	4334 / 1667	2626 / 1368
xl	8868 / 3424	3538 / 2189
2.7B	13206 / 4965	Out of memory / 2570

Problem (memory_profiling): 4 points

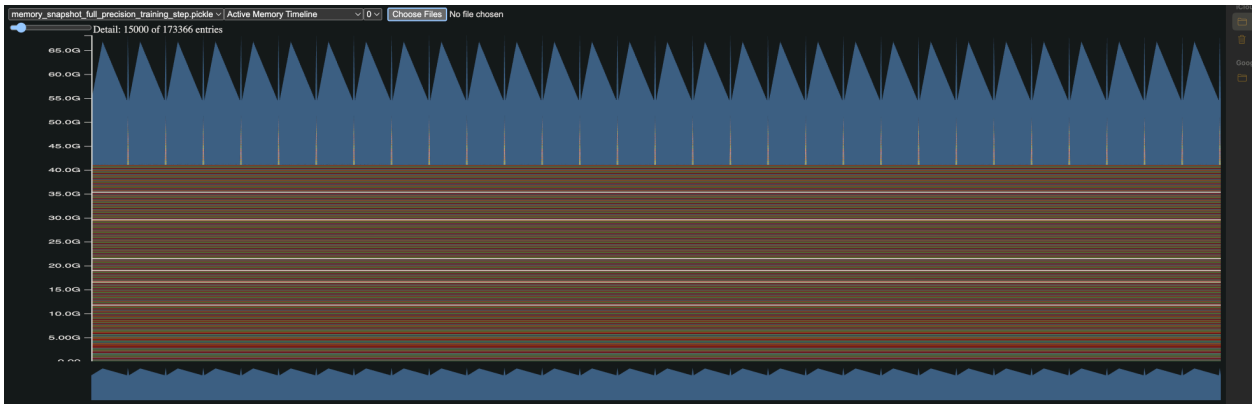
I'm only profiling context_length = 256

a)

Full precision, forward only



Full precision, full training step



(The first peak is backward pass, the second skinny peak is optimizer pass)

b) Memory usage peaks at 37GB for forward only; 67GB for full training step

c) Mixed precision: memory usage peaks at 43GB for forward only; 70 GB for full training step. Doing `torch.autocast(device_type=get_device().type, dtype=dtype)` increases memory usage, as pytorch keeps a copy of the original data. Use `with torch.amp.autocast(model, dtype=torch.bfloat16)` to reduce memory footprint (suggested by ChatGPT)

d) size = batch_size * context_length * d_model * 4 bytes (single precision) = 2.5MB

e) the size of the largest allocations shown is 100MB, but it doesn't show stacktrace. With 40MB we can see the stacktrace, and it's coming from linear layer, specifically from SiLU

Problem (pytorch_attention): 2 points

See the results in the screenshot below, forward & backward are measured on 100 passes

```
d_model: 16, seq_length: 256, forward_total_time: 0.048 sec, backward_total_time: 0.090 sec, forward_max_memory: 22.4 MB, backward_max_memory: 18.2 MB
d_model: 16, seq_length: 1024, forward_total_time: 0.084 sec, backward_total_time: 0.171 sec, forward_max_memory: 89.9 MB, backward_max_memory: 22.7 MB
d_model: 16, seq_length: 4096, forward_total_time: 1.009 sec, backward_total_time: 1.865 sec, forward_max_memory: 1124.7 MB, backward_max_memory: 50.6 MB
d_model: 16, seq_length: 8192, forward_total_time: 3.787 sec, backward_total_time: 6.869 sec, forward_max_memory: 4413.5 MB, backward_max_memory: 117.7 MB
d_model: 16, seq_length: 16384, exception: CUDA out of memory
d_model: 32, seq_length: 256, forward_total_time: 0.046 sec, backward_total_time: 0.083 sec, forward_max_memory: 23.4 MB, backward_max_memory: 19.2 MB
d_model: 32, seq_length: 1024, forward_total_time: 0.078 sec, backward_total_time: 0.167 sec, forward_max_memory: 94.5 MB, backward_max_memory: 27.3 MB
d_model: 32, seq_length: 4096, forward_total_time: 1.040 sec, backward_total_time: 1.889 sec, forward_max_memory: 1141.5 MB, backward_max_memory: 67.4 MB
d_model: 32, seq_length: 8192, forward_total_time: 3.892 sec, backward_total_time: 6.961 sec, forward_max_memory: 4447.0 MB, backward_max_memory: 151.3 MB
d_model: 32, seq_length: 16384, exception: CUDA out of memory
d_model: 64, seq_length: 256, forward_total_time: 0.047 sec, backward_total_time: 0.087 sec, forward_max_memory: 419.3 MB, backward_max_memory: 415.0 MB
d_model: 64, seq_length: 1024, forward_total_time: 0.083 sec, backward_total_time: 0.169 sec, forward_max_memory: 103.7 MB, backward_max_memory: 36.5 MB
d_model: 64, seq_length: 4096, forward_total_time: 1.096 sec, backward_total_time: 1.938 sec, forward_max_memory: 1175.1 MB, backward_max_memory: 100.9 MB
d_model: 64, seq_length: 8192, forward_total_time: 4.108 sec, backward_total_time: 7.145 sec, forward_max_memory: 4514.1 MB, backward_max_memory: 218.4 MB
d_model: 64, seq_length: 16384, exception: CUDA out of memory
d_model: 128, seq_length: 256, forward_total_time: 0.045 sec, backward_total_time: 0.081 sec, forward_max_memory: 29.7 MB, backward_max_memory: 25.5 MB
d_model: 128, seq_length: 1024, forward_total_time: 0.090 sec, backward_total_time: 0.176 sec, forward_max_memory: 122.1 MB, backward_max_memory: 54.9 MB
d_model: 128, seq_length: 4096, forward_total_time: 1.199 sec, backward_total_time: 2.030 sec, forward_max_memory: 1242.2 MB, backward_max_memory: 168.0 MB
d_model: 128, seq_length: 8192, forward_total_time: 4.526 sec, backward_total_time: 7.508 sec, forward_max_memory: 4648.3 MB, backward_max_memory: 352.6 MB
d_model: 128, seq_length: 16384, exception: CUDA out of memory
```

Problem (torch_compile): 2 points

a) scaled dot product attention with torch compile. Interestingly, although there seems to be no memory usage difference for forward/backward passes, there is no CUDA out of memory error in this run, which mean it somehow improve the peak memory performance.

```
/usr/local/lib/python3.11/dist-packages/torch/_inductor/compile_fx.py:194: UserWarning: TensorFloat32 tensor cores for float32 matrix multiplication available but
  warnings.warn(
d_model: 16, seq_length: 256, forward_total_time: 0.034 sec, backward_total_time: 0.056 sec, forward_max_memory: 22.4 MB, backward_max_memory: 18.2 MB
d_model: 16, seq_length: 1024, forward_total_time: 0.057 sec, backward_total_time: 0.113 sec, forward_max_memory: 89.5 MB, backward_max_memory: 22.3 MB
d_model: 16, seq_length: 4096, forward_total_time: 0.539 sec, backward_total_time: 1.086 sec, forward_max_memory: 1124.9 MB, backward_max_memory: 50.6 MB
d_model: 16, seq_length: 8192, forward_total_time: 2.209 sec, backward_total_time: 3.813 sec, forward_max_memory: 4413.7 MB, backward_max_memory: 117.7 MB
d_model: 16, seq_length: 16384, forward_total_time: 7.640 sec, backward_total_time: 15.245 sec, forward_max_memory: 17534.6 MB, backward_max_memory: 344.2 MB
d_model: 32, seq_length: 256, forward_total_time: 0.044 sec, backward_total_time: 0.057 sec, forward_max_memory: 23.4 MB, backward_max_memory: 19.2 MB
d_model: 32, seq_length: 1024, forward_total_time: 0.061 sec, backward_total_time: 0.116 sec, forward_max_memory: 94.6 MB, backward_max_memory: 27.3 MB
d_model: 32, seq_length: 4096, forward_total_time: 0.667 sec, backward_total_time: 1.155 sec, forward_max_memory: 1141.6 MB, backward_max_memory: 67.4 MB
d_model: 32, seq_length: 8192, forward_total_time: 2.404 sec, backward_total_time: 4.047 sec, forward_max_memory: 4447.3 MB, backward_max_memory: 151.3 MB
d_model: 32, seq_length: 16384, forward_total_time: 8.223 sec, backward_total_time: 15.760 sec, forward_max_memory: 17601.7 MB, backward_max_memory: 402.9 MB
d_model: 64, seq_length: 256, forward_total_time: 0.046 sec, backward_total_time: 0.057 sec, forward_max_memory: 410.9 MB, backward_max_memory: 406.7 MB
d_model: 64, seq_length: 1024, forward_total_time: 0.086 sec, backward_total_time: 0.125 sec, forward_max_memory: 103.7 MB, backward_max_memory: 36.5 MB
d_model: 64, seq_length: 4096, forward_total_time: 0.673 sec, backward_total_time: 1.180 sec, forward_max_memory: 1175.2 MB, backward_max_memory: 100.9 MB
d_model: 64, seq_length: 8192, forward_total_time: 2.287 sec, backward_total_time: 4.124 sec, forward_max_memory: 4514.4 MB, backward_max_memory: 218.4 MB
d_model: 64, seq_length: 16384, forward_total_time: 9.111 sec, backward_total_time: 16.480 sec, forward_max_memory: 17735.9 MB, backward_max_memory: 520.4 MB
d_model: 128, seq_length: 256, forward_total_time: 0.046 sec, backward_total_time: 0.058 sec, forward_max_memory: 532.0 MB, backward_max_memory: 527.8 MB
d_model: 128, seq_length: 1024, forward_total_time: 0.092 sec, backward_total_time: 0.131 sec, forward_max_memory: 122.1 MB, backward_max_memory: 54.9 MB
d_model: 128, seq_length: 4096, forward_total_time: 0.773 sec, backward_total_time: 1.269 sec, forward_max_memory: 1242.3 MB, backward_max_memory: 168.0 MB
d_model: 128, seq_length: 8192, forward_total_time: 2.705 sec, backward_total_time: 4.483 sec, forward_max_memory: 4648.6 MB, backward_max_memory: 352.6 MB
d_model: 128, seq_length: 16384, forward_total_time: 10.711 sec, backward_total_time: 17.877 sec, forward_max_memory: 18004.3 MB, backward_max_memory: 755.2 MB
```

b)
Results are sum of 10 steps
Warm up steps = 5

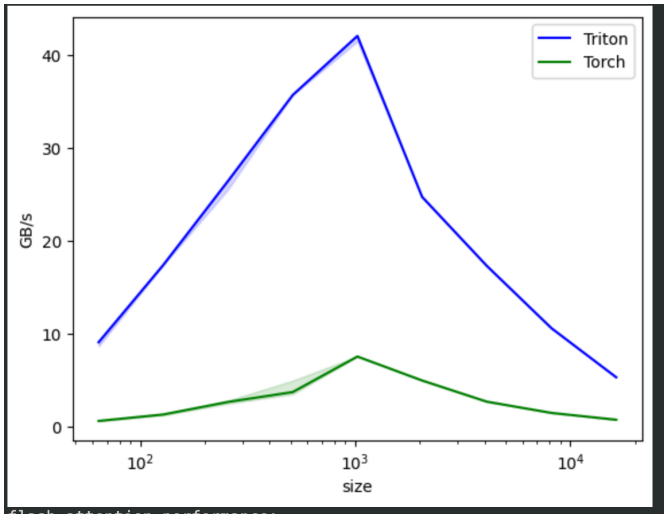
Size	Mean forward only (millisecond), vanilla	Mean forward and backward (millisecond), vanilla	Mean forward only (millisecond), compiled	Mean forward and backward (millisecond), compiled
Small	377	818	178	558
medium	759	2144	551	1732
large	1420	4377	1236	3863

xl	2821	8893	2529	8060
2.7B	4169	13212	3917	12440

Problem (flash_forward): 15 points

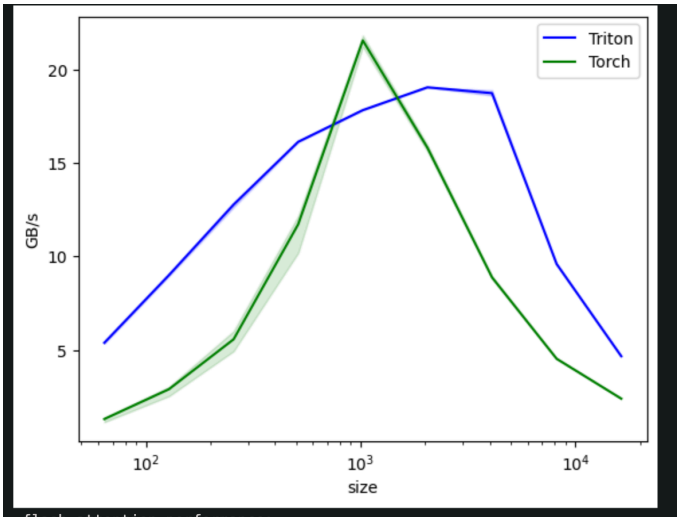
Done. See flash_attention.py. Side note: benchmarking my triton impl of forward pass vs standard pytorch impl on A100 and MI300X

A100:




```
flash-attention-performance:
      size     Triton     Torch
0     64.0   9.142857  0.676056
1    128.0  17.454545  1.373882
2    256.0  26.482758  2.723404
3    512.0  35.720930  3.783251
4   1024.0  42.082190  7.603961
5   2048.0  24.774194  5.027823
6   4096.0  17.355933  2.750840
7   8192.0  10.648180  1.540960
8  16384.0   5.373272  0.808075
```

MI300X




```
flash-attention-performance:
       size      Triton      Torch
0      64.0    5.389030   1.311594
1     128.0    9.006528   2.906102
2     256.0   12.770900   5.571979
3     512.0   16.131773  11.721695
4    1024.0   17.824741  21.550215
5    2048.0   19.044818  15.816303
6    4096.0   18.729449   8.883377
7    8192.0    9.598561   4.533137
8   16384.0    4.668761   2.400649
```

Triton code actually performs better on A100 than MI300X

Problem (flash_benchmarking): 5 points

See benchmark above

FlashAttention-2 Leaderboard

a single MI300X (no H100 available) with batch size 1, sequence length 16384, d_model=1024 and num_heads=16.

tile_size=16 (default),results=75.4004898071289 ms
tile_size=32,results=71.679931640625 ms
tile_size=64,results=64.42243957519531 ms
tile_size=128,results=58.10773849487305 ms

Got `BackendCompilerFailed: backend='inductor' raised:
OutOfResources: out of resource: shared memory, Required: 98304, Hardware limit:
65536. Reducing block sizes or `num_stages` may help.` if tile_size = 256

Problem (distributed_communication_single_node): 5 points

Gloo backend:
1.0 MB, world_size=2, 0.74 seconds
1.0 MB, world_size=4, 1.33 seconds
1.0 MB, world_size=6, 1.51 seconds
10.0 MB, world_size=2, 0.77 seconds
10.0 MB, world_size=4, 1.39 seconds
10.0 MB, world_size=6, 1.67 seconds
100.0 MB, world_size=2, 1.06 seconds
100.0 MB, world_size=4, 1.59 seconds
100.0 MB, world_size=6, 1.31 seconds
1000.0 MB, world_size=2, 3.84 seconds
1000.0 MB, world_size=4, 4.42 seconds
1000.0 MB, world_size=6, 5.48 seconds

NCCL backend (AMD MI300X 8 GPU):
1.0 MB, world_size=2, 11.62 seconds
1.0 MB, world_size=4, 12.51 seconds
1.0 MB, world_size=6, 12.21 seconds
10.0 MB, world_size=2, 11.12 seconds
10.0 MB, world_size=4, 11.79 seconds
10.0 MB, world_size=6, 12.51 seconds
100.0 MB, world_size=2, 11.12 seconds
100.0 MB, world_size=4, 11.97 seconds
100.0 MB, world_size=6, 12.83 seconds
1000.0 MB, world_size=2, 12.50 seconds
1000.0 MB, world_size=4, 12.91 seconds
1000.0 MB, world_size=6, 13.81 seconds
1000.0 MB, world_size=8, 14.95 seconds
10000.0 MB, world_size=8, 26.81 seconds
50000.0 MB, world_size=8, 77.35 seconds
75000.0 MB, world_size=8, 113.34 seconds
100000.0 MB, world_size=8, 147.82 seconds
150000.0 MB, world_size=8, 216.45 seconds

NCCL backend (NVIDIA V100 8 GPU):
1.0 MB, world_size=2, 2.99 seconds
1.0 MB, world_size=4, 3.40 seconds
1.0 MB, world_size=6, 3.75 seconds
10.0 MB, world_size=2, 3.61 seconds
10.0 MB, world_size=4, 3.75 seconds
10.0 MB, world_size=6, 3.87 seconds

100.0 MB, world_size=2, 3.14 seconds
100.0 MB, world_size=4, 3.65 seconds
100.0 MB, world_size=6, 4.32 seconds
1000.0 MB, world_size=2, 5.07 seconds
1000.0 MB, world_size=4, 6.27 seconds
1000.0 MB, world_size=6, 5.97 seconds
5000.0 MB, world_size=2, 12.75 seconds
5000.0 MB, world_size=4, 14.08 seconds
5000.0 MB, world_size=6, 14.59 seconds
10000.0 MB, world_size=2, 23.71 seconds
10000.0 MB, world_size=4, 24.21 seconds
10000.0 MB, world_size=6, 24.54 seconds
15000.0 MB, world_size=2, 32.70 seconds
15000.0 MB, world_size=4, 34.12 seconds
15000.0 MB, world_size=6, 35.50 seconds

NCCL backend (NVIDIA H100 SXM5 8 GPU):
1000.0 MB, world_size=8, 10.04 seconds
10000.0 MB, world_size=8, 24.26 seconds
50000.0 MB, world_size=8, 104.40 seconds
75000.0 MB, world_size=8, 153.98 seconds

Observation for NCCL backend: the overhead is pretty high >10 seconds (or the communication between AMD GPU is slow); increase world_size (when it's <= 8) doesn't increase the overall processing time by much

Actually MI300X outperforms H100 SXM5 :O That's surprising. The results are somewhat different than what is shown in
https://semianalysis.com/2024/12/22/mi300x-vs-h100-vs-h200-benchmark-part-1-training/