CS336 Assignment 2 (systems): Systems and Parallelism

Version 1.0.4

Spring 2025

1 Assignment Overview

In this assignment, you will gain some hands-on experience with improving single-GPU training speed and scaling training to multiple GPUs.

What you will implement.

1. Benchmarking and profiling harness
2. Flash Attention 2 Triton kernel
3. Distributed data parallel training
4. Optimizer state sharding

What the code looks like. All the assignment code as well as this writeup are available on GitHub at:

github.com/stanford-cs336/assignment2-systems

Please `git clone` the repository. If there are any updates, we will notify you and you can `git pull` to get the latest.

1. `cs336-basics/`: In this assignment, you'll be profiling some of the components that we built in assignment 1. This folder contains the staff solution code for assignment 1, so you will find a `cs336-basics/pyproject.toml` and a `cs336-basics/cs336_basics/*` module in here. If you want to use your own implementation of the model, you can modify the `pyproject.toml` file in the base directory to point to your own package.
2. `/`: The cs336-systems base directory. We created an empty module named `cs336_systems`. Note that there's no code in here, so you should be able to do whatever you want from scratch.
3. `tests/*.py`: This contains all the tests that you must pass. These tests invoke the hooks defined in `tests/adapters.py`. You'll implement the adapters to connect your code to the tests. Writing more tests and/or modifying the test code can be helpful for debugging your code, but your implementation is expected to pass the original provided test suite.
4. `README.md`: This file contains more details about the expected directory structure, as well as some basic instructions on setting up your environment.

How to submit. You will submit the following files to Gradescope:

- `writeup.pdf`: Answer all the written questions. Please typeset your responses.
- `code.zip`: Contains all the code you've written.

Run the script in `test_and_make_submission.sh` to create the `code.zip` file.

In the first part of the assignment, we will look into how to optimize the performance of our Transformer model to make the most efficient use of the GPU. We will profile our model to understand where it spends time and memory during the forward and backward passes, then optimize the self-attention operation with custom GPU kernels, making it faster than a straightforward PyTorch implementation. In the subsequent parts of the assignment, we will leverage multiple GPUs.

1.1 Profiling and Benchmarking

Before implementing any optimization, it is helpful to first profile our program to understand where it spends resources (e.g., time and memory). Otherwise, we risk optimizing parts of the model that don't account for significant time or memory, and therefore not seeing measurable end-to-end improvements.

We will implement three performance evaluation paths: (a) a simple, end-to-end benchmarking using the Python standard library to time our forward and backward passes, (b) profile compute with the NVIDIA Nsight Systems tool to understand how that time is distributed across operations on both the CPU and GPU, and (c) profile memory usage.

1.1.1 Setup - Importing your `Basics` Transformer Model

Let's start by making sure that you can load the model from the previous assignment. In the previous assignment, we set up our model in a Python package, so that it could be easily imported later. We have added the staff implementation of the model in the `./cs336-basics` folder, and have pointed to it in the `pyproject.toml` file. By calling `uv run [command]` as usual, `uv` will automatically locate this local `cs336-basics` package. If you would like to use your own implementation of the model, you can modify the `pyproject.toml` file to point to your own package.

You can test that you can import your model with:

```
~$ uv run python
Using CPython 3.12.10
Creating virtual environment at: /path/to/uv/env/dir
      Built cs336-systems @ file:///path/to/systems/dir
      Built cs336-basics @ file:///path/to/basics/dir
Installed 85 packages in 711ms
Python 3.12.10 (main, Apr  9 2025, 04:03:51) [Clang 20.1.0 ] on linux
...
>>> import cs336_basics
>>>
```

The relevant modules from assignment 1 should now be available (e.g., for `model.py`, you can import it with `import cs336_basics.model`).

1.1.2 Model Sizing

Throughout this assignment, we will be benchmarking and profiling models to better understand their performance. To get a sense of how things change at scale, we will work with and refer to the following model configurations. For all models, we'll use a vocabulary size of 10,000 and a batch size of 4, with varying context lengths. This assignment (and later ones) will require a lot of results to be presented in tables. We strongly recommend that you automate constructing tables for your writeup in code, since formatting tables in LaTeX or Markdown can be very tedious. See `pandas.DataFrame.to_latex()` and `pandas.DataFrame.to_markdown()` or write your own function to generate them from your preferred tabular representation.

Size	d_model	d_ff	num_layers	num_heads
small	768	3072	12	12
medium	1024	4096	24	16
large	1280	5120	36	20
xl	1600	6400	48	25
2.7B	2560	10240	32	32

Table 1: Specifications of different model sizes

1.1.3 End-to-End Benchmarking

We will now implement a simple performance evaluation script. We will be testing many variations of our model (changing precision, swapping layers, etc.), so **it will pay off to have your script enable these variations via command-line arguments** to make them easy to run later on. We also **highly recommend running sweeps over benchmarking hyperparameters, such as model size, context length, etc., using sbatch or submitit on Slurm** for quick iteration.

To start off, let's do the simplest possible profiling of our model by timing the forward and backward passes. Since we will only be measuring speed and memory, we will use random weights and data.

Measuring performance is subtle — some common traps can cause us to not measure what we want. For benchmarking GPU code, one caveat is that CUDA calls are *asynchronous*. When you call a CUDA kernel, such as when you invoke `torch.matmul`, the function call returns control to your code without waiting for the matrix multiplication to finish. In this way, the CPU can continue running while the GPU computes the matrix multiplication. On the other hand, this means that naïvely measuring how long the `torch.matmul` call takes to return does not tell us how long the GPU takes to actually run the matrix multiplication In PyTorch, we can call `torch.cuda.synchronize()` to wait for all GPU kernels to complete, allowing us to get more accurate measurements of CUDA kernel runtime. With this in mind, let's write our basic profiling infrastructure.

Problem (`benchmarking_script`): 4 points

(a) Write a script to perform basic end-to-end benchmarking of the forward and backward passes in your model. Specifically, your script should support the following:

Done

- Given hyperparameters (e.g., number of layers), initialize a model.
- Generate a random batch of data.
- Run w warm-up steps (before you start measuring time), then time the execution of n steps (either only forward, or both forward and backward passes, depending on an argument). For timing, you can use the Python `timeit` module (e.g., either using the `timeit` function, or using `timeit.default_timer()`, which gives you the system's highest resolution clock, thus a better default for benchmarking than `time.time()`).
- Call `torch.cuda.synchronize()` after each step.

Deliverable: A script that will initialize a `basics` Transformer model with the given hyperparameters, create a random batch of data, and time forward and backward passes.

(b) Time the forward and backward passes for the model sizes described in §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) For the Small model, the forward pass averages 0.0533 s with a standard deviation of 0.00066 s, while the backward pass averages 0.1076 s with a standard deviation of 0.00162 s.
For the Medium model, the forward pass takes 0.1604 s ± 0.00257 s, and the backward pass takes 0.3221 s ± 0.00263 s.
For the Large model, the forward pass averages 0.3309 s ± 0.00943 s, and the backward pass averages 0.6763 s ± 0.00581 s.
Overall, runtime increases monotonically with model size. Across all configurations, the backward pass consistently takes about 2× longer than the forward pass due to gradient computation. The standard deviations are relatively small, indicating stable and consistent measurements

Deliverable: A 1-2 sentence response with your timings.

(c) 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?

Deliverable: A 2-3 sentence response.

Done

1.1.4 Nsight Systems Profiler

End-to-end benchmarking does not tell us where our model spends time and memory during forward and backward passes, and so does not expose specific optimization opportunities. To know how much time our program spends in each component (e.g., function), we can use a *profiler*. An execution profiler instruments the code by inserting guards when functions begin and finish running, and thus can give detailed execution statistics at the function level (such as number of calls, how long they take on average, cumulative time spent on this function, etc).

Standard Python profilers (e.g., `CProfile`) are not able to profile CUDA kernels since these kernels are executed asynchronously on the GPU. Fortunately, NVIDIA ships a profiler that we can use via the CLI `nsys`, which we have already installed for you. In this part of the assignment, you will use `nsys` to analyze the runtime of your Transformer model. Using `nsys` is straightforward: we can simply run your Python script from the previous section with `nsys profile` prepended. For example, you can profile a script `benchmark.py` and write the output to a file `result.nsys.rep` with:

```
~$ uv run nsys profile -o result python benchmark.py
```

You can then view the profile on your local machine with the NVIDIA Nsight Systems desktop application. Selecting a particular CUDA API call (on the CPU) in the `CUDA API` row of the profile will highlight all corresponding kernel executions (on the GPU) in the `CUDA HW` row.

We encourage you to experiment with various command-line options for `nsys profile` to get a sense of what it can do. Notably, you can get Python backtraces for each CUDA API call with `--python-backtrace=cuda`, though this may introduce overhead. You can also annotate your code with NVTX ranges, which will appear as blocks in the `NVTX` row of the profile capturing all CUDA API calls and associated kernel executions. In particular, you should use NVTX ranges to **ignore the warm-up steps in your benchmarking script** (by applying a filter on the `NVTX` row in the profile). You can also isolate which kernels are responsible for the forward and backward passes of your model, and you can even isolate which kernels are responsible for different parts of a self-attention layer by annotating your implementation as follows:

```python
...
import torch.cuda.nvtx as nvtx

@nvtx.range("scaled dot product attention")
def annotated_scaled_dot_product_attention(
    ... # Q, K, V, mask
)
    ...
    with nvtx.range("computing attention scores"):
        ... # compute attention scores between Q and K

    with nvtx.range("computing softmax")
        ... # compute softmax of attention scores

    with nvtx.range("final matmul")
        ... # compute output projection
```

For the Small model, using no warm-up steps results in a forward pass of 0.0617 s ± 0.0253 s and a backward pass of 0.1126 s ± 0.0163 s, showing noticeably higher variance. With one warm-up step, variance is reduced (0.0532 s ± 0.00071 s forward, 0.1077 s ± 0.00251 s backward), but measurements are still less stable than with five warm-up steps, which yield 0.0533 s ± 0.00066 s forward and 0.1076 s ± 0.00162 s backward.

Warm-up steps help stabilize benchmarking by ensuring that kernel compilation, memory allocation, and caching effects are settled before timing begins, leading to more consistent and reliable measurements.

```
    return ...
```

You can swap your original implementation with the annotated version in your benchmarking script via:

```
cs336_basics.model.scaled_dot_product_attention = annotated_scaled_dot_product_attention
```

Finally, you can use the `--pytorch` command-line option with `nsys` to automatically annotate calls to the PyTorch C++ API with NVTX ranges.

Problem (`nsys_profile`): 5 points

Profile your forward pass, backward pass, and optimizer step using `nsys` with each of the model sizes described in Table 1 and context lengths of 128, 256, 512 and 1024 (you may run out of memory with some of these context lengths for the larger models, in which case just note it in your report).

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

Deliverable: A 1-2 sentence response.

(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.)

Deliverable: A 1-2 sentence response.

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

Deliverable: A 1-2 sentence response.

(d) Profile running one complete training step with your implementation of AdamW (i.e., the forward pass, computing the loss and running a backward pass, and finally an optimizer step, as you'd do during training). How does the fraction of time spent on matrix multiplication change, compared to doing inference (forward pass only)? How about other kernels?

Deliverable: A 1-2 sentence response.

(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?

Deliverable: A 1-2 sentence response.

1.1.5 Mixed Precision

Up to this point in the assignment, we've been running with FP32 precision—all model parameters and activations have the `torch.float32` datatype. However, modern NVIDIA GPUs contain specialized GPU cores (Tensor Cores) for accelerating matrix multiplies at lower precisions. For example, the NVIDIA A100 spec sheet says that its maximum throughput with FP32 is 19.5 TFLOP/second, while its maximum throughput with FP16 (half-precision floats) or BF16 (brain floats) is significantly higher at 312 TFLOP/second. As a result, using lower-precision datatypes should help us speed up training and inference.

a) almost identical
b)forward: volta sgemm 128x64, invoked 37 times, forward + backward: same kernel
c) (vectorized) element wise kernels
d) forward only ~46% matmul, complete training step 35.9%
e) attention scores 0.6ms vs softmax 1.4 ms while FLOPS are 2*T^2D + T^2 for attention scores and only T^2 * 5 for softmax. So softmax „looses“ time elsewhere i.e. memory overhead

However, naïvely casting our model into a lower-precision format may come with reduced model accuracy. For example, many gradient values in practice are often too small to be representable in FP16, and thus become zero when naïvely training with FP16 precision. To combat this, it's common to use loss scaling when training with FP16—the loss is simply multiplied by a scaling factor, increasing gradient magnitudes so they don't flush to zero. Furthermore, FP16 has a lower dynamic range than FP32, which can lead to overflows that manifest as a NaN loss. Full bfloat16 training is generally more stable (since BF16 has the same dynamic range as FP32), but can still affect final model performance compared to FP32.

To take advantage of the speedups from lower-precision datatypes, it's common to use *mixed-precision* training. In PyTorch, this is implemented with the `torch.autocast` context manager. In this case, certain operations (e.g., matrix multiplies) are performed in lower-precision datatypes, while other operations that require the full dynamic range of FP32 (e.g., accumulations and reductions) are kept as-is. For example, the following code will automatically identify which operations to perform in lower-precision during the forward pass and cast these operations to the specified data type:

```
model : torch.nn.Module = ... # e.g. your Transformer model
dtype : torch.dtype = ... # e.g. torch.float16
x : torch.Tensor = ... # input data

with torch.autocast(device="cuda",dtype=dtype):
    y = model(x)
```

As alluded to above, it is generally a good idea to keep accumulations in higher precision even if the tensors themselves being accumulated have been downcasted. The following exercise will help build your intuition as to why this is the case.

Problem (`mixed_precision_accumulation`): 1 point

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)
```

Accumulating in float32 and calculating in float16 comes close to full precision whereas using fp16 for everything is considerably less precise:
tensor(10.0001)
tensor(9.9531, dtype=torch.float16)
tensor(10.0021)
tensor(10.0021)

Deliverable: A 2-3 sentence response.

We will now apply mixed precision first to a toy model for intuition and then to our benchmarking script.

a) Model Parameters: torch.float32
fc1: torch.float16
ln: torch.float32
Loss: torch.float32
Gradients: torch.float32

Problem (benchmarking_mixed_precision): 2 points

(a) Consider the following model:

```
class ToyModel(nn.Module):
    def __init__(self, in_features: int, out_features: int):
        super().__init__()
        self.fc1 = nn.Linear(in_features, 10, bias=False)
        self.ln = nn.LayerNorm(10)
        self.fc2 = nn.Linear(10, out_features, bias=False)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.ln(x)
        x = self.fc2(x)
        return x
```

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

b) calculating variance (square root of small values) and dividing by std
bf16 has the same dynamic range of fp32 so we dont need to downcast

Deliverable: The data types for each of the components listed above.

(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?

Deliverable: A 2-3 sentence response.

(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.

c) with fp16 ~ 2-2.5x as fast

Deliverable: A 2-3 sentence response with your timings and commentary.

1.1.6 Profiling Memory

So far, we have been looking at compute performance. We'll now shift our attention to *memory*, another major resource in language model training and inference. PyTorch also ships with a powerful memory profiler, which can keep track of allocations over time.

To use the memory profiler, you can modify your benchmarking script as follows:

a) Base usage from model weights. Linear growth during forward, then further growth at the beginning of backward going into linear decay during backward and small bump for optimizer

b) 10GB forward, 11.5GB peak for full training

c) identical memory usage

d) Size: (batch_size * d_model * seq_len * 4 * 1024**-3 = 1MB

```
... # warm-up phase in your benchmarking script

# Start recording memory history.
torch.cuda.memory._record_memory_history(max_entries=1000000)

... # what you want to profile in your benchmarking script

# Save a pickle file to be loaded by PyTorch's online tool.
torch.cuda.memory._dump_snapshot("memory_snapshot.pickle")

# Stop recording history.
torch.cuda.memory._record_memory_history(enabled=None)
```

e) 16MB from attention scores (4 * 4 * 16 * 256 * 256
39.1MB from output projection (4 * 4 * 256 * 10000)

This will output a file `memory_snapshot.pickle` that you can load into the following online tool: `https://pytorch.org/memory_viz` . This tool will let you see the overall memory usage timeline as well as each individual allocation that was made, with its size and a stack trace leading to the code where it originates. To use this tool, you should open the link above in a Web browser, and then drag and drop your Pickle file onto the page.

You will now use the PyTorch profiler to analyze the memory usage of your model.

Problem (`memory_profiling`): 4 points Using model size medium with ctx= 256

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

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

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

Deliverable: A table with two numbers per context length.

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

Deliverable: A 2-3 sentence response.

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

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

(e) Now look closely at the "Active Memory Timeline" from `pytorch.org/memory_viz` of a memory snapshot of the `2.7B` model doing a forward pass. When you reduce the "Detail" level, the tool hides the smallest allocations to the corresponding level (e.g., putting "Detail" at 10% only shows

the 10% largest allocations). What is the size of the largest allocations shown? Looking through the stack trace, can you tell where those allocations come from?

Deliverable: A 1-2 sentence response.

1.2 Optimizing Attention with FlashAttention-2

1.2.1 Benchmarking PyTorch Attention

Your profiling likely suggests that there is an opportunity for optimization, both in terms of memory and compute, in your attention layers. At a high level, the attention operation consists of a matrix multiplication followed by softmax, then another matrix multiplication:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\text{mask}\left(\frac{Q^\top K}{\sqrt{d_k}}\right)\right)V \tag{1}$$

The naïve attention implementation needs to save attention score matrices of shape `seq_len` × `seq_len` for each batch/head element, which can grow very large with long sequence lengths, causing out-of-memory errors for any tasks with long inputs or outputs. We will implement an attention kernel following the FlashAttention-2 paper, which computes attention by tiles and avoids ever explicitly materializing the `seq_len` × `seq_len` attention score matrices, enabling scaling to much longer sequence lengths.

Problem (`pytorch_attention`): 2 points

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

(a) Fix the batch size to 8 and don't use multihead attention (i.e. remove the head dimension).

(b) Iterate through the cartesian product of `[16, 32, 64, 128]` for the head embedding dimension d_{model}, and `[256, 1024, 4096, 8192, 16384]` for the sequence length.

(c) Create random inputs Q, K, V for the appropriate size.

(d) Time 100 forward passes through attention using the inputs.

(e) Measure how much memory is in use before the backward pass starts, and time 100 backward passes.

(f) 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.

1.3 Benchmarking JIT-Compiled Attention

Since version 2.0, PyTorch also ships with a powerful just-in-time compiler that automatically tries to apply a number of optimizations to PyTorch functions: see `https://pytorch.org/tutorials/intermediate/torch_compile_tutorial.html` for an intro. In particular, it will try to automatically generate fused Triton kernels by dynamically analyzing your computation graph. The interface to use the PyTorch compiler is very simple. For instance, if we wanted to apply it to a single layer of our model, we can use:

For table see next page

calculation: First OOM for d_model 16 and seq_len 16384: Memory usage = B * ((2*T^2 + 6 * T * d = 2GB * batch_size = 2GB * 8 =16 GB

Memory saved for backward grows ~ O(T^1.5) with sequence length -> we could avoid storing activations and recompute them

a) Compiled table:

16	256	7.274215e-01 +/- 6.626808e-02	9.412828e-01 +/- 4.998593e-02	280.9
16	1024	1.287666e+00 +/- 4.958367e-02	2.504747e+00 +/- 6.799218e-02	402.8
16	4096	1.664725e+01 +/- 3.037822e-01	3.557076e+01 +/- 7.413018e-01	2074.8
16	8192	8.339741e+01 +/- 2.244741e+00	1.593205e+02 +/- 4.194668e+00	8229.3
32	256	9.414172e-01 +/- 1.290850e-01	9.998211e-01 +/- 1.329374e-01	68.5
32	1024	1.314799e+00 +/- 6.196355e-02	2.549766e+00 +/- 2.230574e-01	149.4
32	4096	1.917104e+01 +/- 1.293181e+00	4.053830e+01 +/- 3.113575e+00	2084.8
32	8192	8.742662e+01 +/- 2.115810e+00	1.645602e+02 +/- 3.871498e+00	8249.3
64	256	1.210241e+00 +/- 3.537526e-01	1.185162e+00 +/- 2.848081e-01	120.8
64	1024	1.460849e+00 +/- 8.433635e-02	2.753413e+00 +/- 1.407415e-01	154.4
64	4096	1.971589e+01 +/- 9.029424e-01	4.149131e+01 +/- 1.932130e+00	2104.8
64	8192	1.049356e+02 +/- 2.216332e+00	1.807818e+02 +/- 2.488787e+00	8289.3
128	256	1.202674e+00 +/- 1.315892e-01	1.235932e+00 +/- 2.253136e-01	225.2
128	1024	1.949209e+00 +/- 7.109514e-02	3.859927e+00 +/- 1.317384e-01	164.4
128	4096	2.923703e+01 +/- 8.150190e-01	6.043564e+01 +/- 1.987309e+00	2144.8
128	8192	1.537970e+02 +/- 1.966000e+00	2.555709e+02 +/- 2.927000e+00	8369.3

```
layer = SomePyTorchModule(...)
compiled_layer = torch.compile(layer)
```

Now, `compiled_layer` functionally behaves just like `layer` (e.g., with its forward and backward passes). We can also compile our entire PyTorch model with `torch.compile(model)`, or even a Python function that calls PyTorch operations.

Problem (`torch_compile`): 2 points

(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.

Given the scaling behaviors we've seen with respect to the sequence length, we need significant improvements to handle large sequences. Even with `torch.compile`, the current implementation suffers from very poor memory access patterns at long sequence length. For that, we will write a Triton implementation of FlashAttention-2, where we'll have significantly more control over how memory is accessed and when to compute what.

1.3.1 Example - Weighted Sum

To introduce what you'll need to know about Triton and how it interoperates with PyTorch, we will work through an example kernel for a "weighted sum" operation. For further resources on getting up to speed with Triton, see Triton's tutorials. We note that these tutorials do not use the new, convenient block pointer abstraction, which we will walk through below.

Given an input matrix X, we'll multiply its entries by a column-wise weight vector w, and sum each row, giving us the matrix-vector product of X and w. We are going to work through the forward pass of this operation first, and then write the Triton kernel for the backward pass.

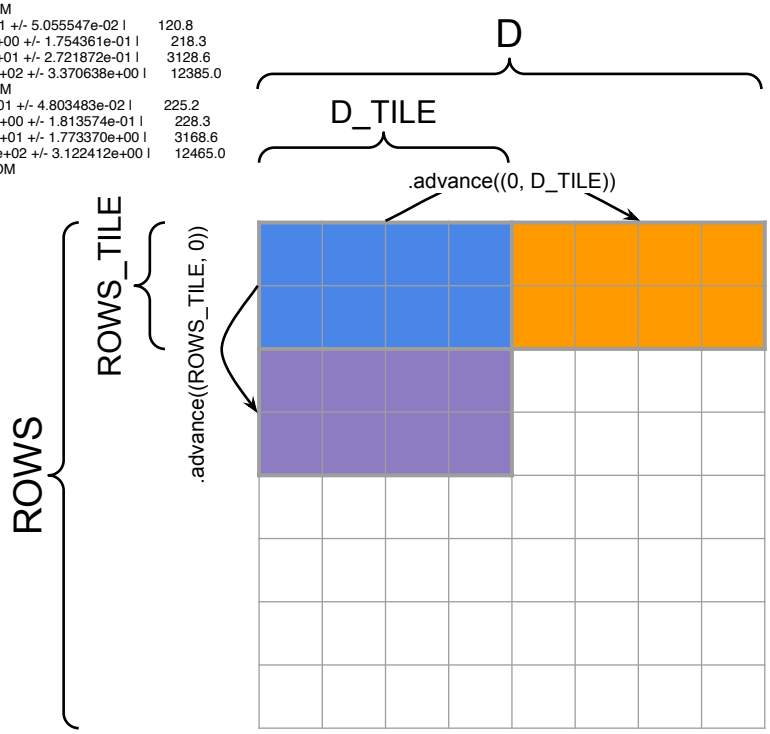
Forward pass The forward pass of our kernel is just the following broadcasted inner product.

```
def weighted_sum(x, weight):
    # Here, assume that x has n-dim shape [..., D], and weight has 1D shape [D]
    return (weight * x).sum(axis=-1)
```

When writing our Triton kernel, we'll have each program instance (potentially running in parallel) compute the weighted sum of a *tile* of rows of x, and write the corresponding scalar outputs to the output tensor. In Triton, a program instance is a block of threads all running the same program, and these *thread blocks* can be run in parallel on the GPU. Instead of taking *tensors* as arguments, we take *pointers* to their first elements, as well as *strides* for each tensor that tell us how to move along axes.

We can use the strides to load a tensor corresponding to the tile of rows of x that we're summing in the running instance, using the program ID to divide up the work (i.e., instance i will process the i-th tile of rows of x). The main difference between the forward pass in Triton and PyTorch in this simple case is the need to do pointer arithmetic and explicit loads/stores. We will use the block pointer abstraction with

d_model	seq_len	Forward (s)	Backward (s)	Memory (MB)
16	256	4.520544e-01 +/- 3.725332e-02	8.930301e-01 +/- 3.522104e-02	28.9
16	1024	2.350687e+00 +/- 3.858239e-01	5.230948e+00 +/- 6.525002e-01	210.8
16	4096	3.314916e+01 +/- 8.142632e-02	7.343781e+01 +/- 1.972224e-01	3098.6
16	8192	1.341644e+02 +/- 3.493349e-01	2.974604e+02 +/- 7.107816e-01	12325.0
16	16384	OOM	OOM	OOM
32	256	4.751640e-01 +/- 6.635296e-02	9.294005e-01 +/- 1.445823e-01	68.5
32	1024	2.305622e+00 +/- 4.967861e-02	5.058343e+00 +/- 8.289172e-02	213.3
32	4096	3.308548e+01 +/- 7.123218e-02	7.380177e+01 +/- 1.930061e-01	3108.6
32	8192	1.393855e+02 +/- 1.522672e+00	3.060375e+02 +/- 2.218230e+00	12345.0
32	16384	OOM	OOM	OOM
64	256	4.563087e-01 +/- 3.761615e-02	9.111882e-01 +/- 5.055547e-02	120.8
64	1024	2.430178e+00 +/- 8.853210e-02	5.404156e+00 +/- 1.754361e-01	218.3
64	4096	3.529663e+01 +/- 1.418731e-01	7.817337e+01 +/- 2.721872e-01	3128.6
64	8192	1.622170e+02 +/- 3.348134e+00	3.276242e+02 +/- 3.370638e+00	12385.0
64	16384	OOM	OOM	OOM
128	256	4.753839e-01 +/- 6.058706e-02	9.161891e-01 +/- 4.803483e-02	225.2
128	1024	2.903987e+00 +/- 9.227478e-02	6.436815e+00 +/- 1.813574e-01	228.3
128	4096	4.290203e+01 +/- 6.939158e-01	9.379558e+01 +/- 1.773370e+00	3168.6
128	8192	2.143595e+02 +/- 4.260375e+00	4.005383e+02 +/- 3.122412e+00	12465.0
128	16384	OOM	OOM	OOM



b)
Uncompiled:
Forward pass:
Average: 2.715384e-01 s
Std: 1.841693e-03 s

Backward pass:
Average: 5.758112e-01 s
Std: 3.721271e-03 s

Optimizer step:
Average: 2.259893e-01 s
Std: 5.540635e-05 s

Compiled

Forward pass:
Average: 2.329428e-01 s
Std: 1.966675e-03 s

Backward pass:
Average: 4.740946e-01 s
Std: 1.174464e-02 s

Optimizer step:
Average: 2.257507e-01 s
Std: 7.991698e-04 s

Figure 1: Tiling and advancing block pointers in the weighted sum kernel example (Section 1.3.1).

`tl.make_block_ptr` to greatly simplify the pointer arithmetic, although this means we need to do some setup to prepare the block pointers.

Refer to Figure 1 for a schematic of tiling and how block pointers are advanced. The weighted sum function from above looks like the following:

```python
import triton
import triton.language as tl

@triton.jit
def weighted_sum_fwd(
    x_ptr, weight_ptr,  # Input pointers
    output_ptr,  # Output pointer
    x_stride_row, x_stride_dim,  # Strides tell us how to move one element in each axis of a tensor
    weight_stride_dim,  # Likely 1
    output_stride_row,  # Likely 1
    ROWS, D,
    ROWS_TILE_SIZE: tl.constexpr, D_TILE_SIZE: tl.constexpr,  # Tile shapes must be known at compile time
):
    # Each instance will compute the weighted sum of a tile of rows of x.
    # `tl.program_id` gives us a way to check which thread block we're running in
    row_tile_idx = tl.program_id(0)

    # Block pointers give us a way to select from an ND region of memory
    #  and move our selection around.
    # The block pointer must know:
    # - The pointer to the first element of the tensor
    # - The overall shape of the tensor to handle out-of-bounds access
```

```
    # - The strides of each dimension to use the memory layout properly
    # - The ND coordinates of the starting block, i.e., "offsets"
    # - The block shape to use load/store at a time
    # - The order of the dimensions in memory from major to minor
    #    axes (= np.argsort(strides)) for optimizations, especially useful on H100

    x_block_ptr = tl.make_block_ptr(
        x_ptr,
        shape=(ROWS, D,),
        strides=(x_row_stride, x_stride_dim),
        offsets=(row_tile_idx * ROWS_TILE_SIZE, 0),
        block_shape=(ROWS_TILE_SIZE, D_TILE_SIZE),
        order=(1, 0),
    )

    weight_block_ptr = tl.make_block_ptr(
        weight_ptr,
        shape=(D,),
        strides=(weight_stride_dim,),
        offsets=(0,),
        block_shape=(D_TILE_SIZE,),
        order=(0,),
    )

    output_block_ptr = tl.make_block_ptr(
        output_ptr,
        shape=(ROWS,),
        strides=(output_stride_row,),
        offsets=(row_tile_idx * ROWS_TILE_SIZE,),
        block_shape=(ROWS_TILE_SIZE,),
        order=(0,),
    )

    # Initialize a buffer to write to
    output = tl.zeros((ROWS_TILE_SIZE,), dtype=tl.float32)

    for i in range(tl.cdiv(D, D_TILE_SIZE)):
        # Load the current block pointer
        # Since ROWS_TILE_SIZE might not divide ROWS, and D_TILE_SIZE might not divide D,
        # we need boundary checks for both dimensions
        row = tl.load(x_block_ptr, boundary_check=(0, 1), padding_option="zero")  # (ROWS_TILE_SIZE, D_TILE_SIZE)
        weight = tl.load(weight_block_ptr, boundary_check=(0,), padding_option="zero")  # (D_TILE_SIZE,)

        # Compute the weighted sum of the row.
        output += tl.sum(row * weight[None, :], axis=1)

        # Move the pointers to the next tile.
        # These are (rows, columns) coordinate deltas
        x_block_ptr = x_block_ptr.advance((0, D_TILE_SIZE))  # Move by D_TILE_SIZE in the last dimension
        weight_block_ptr = weight_block_ptr.advance((D_TILE_SIZE,))  # Move by D_TILE_SIZE

    # Write output to the output block pointer (a single scalar per row).
    # Since ROWS_TILE_SIZE might not divide ROWS, we need boundary checks
    tl.store(output_block_ptr, output, boundary_check=(0,))
```

Let's now wrap this kernel in a PyTorch Autograd function, that will interoperate with PyTorch (i.e., take `Tensors` as inputs, output a `Tensor`, and later also work with the autograd engine during the backward pass):

```
class WeightedSumFunc(torch.autograd.Function):
@staticmethod
    def forward(ctx, x, weight):
        # Cache x and weight to be used in the backward pass, when we
        # only receive the gradient wrt. the output tensor, and
```

```
        # need to compute the gradients wrt. x and weight.
        D, output_dims = x.shape[-1], x.shape[:-1]

        # Reshape input tensor to 2D
        input_shape = x.shape
        x = rearrange(x, "... d -> (...) d")

        ctx.save_for_backward(x, weight)

        assert len(weight.shape) == 1 and weight.shape[0] == D, "Dimension mismatch"
        assert x.is_cuda and weight.is_cuda, "Expected CUDA tensors"
        assert x.is_contiguous(), "Our pointer arithmetic will assume contiguous x"

        ctx.D_TILE_SIZE = triton.next_power_of_2(D) // 16  # Roughly 16 loops through the embedding dimension
        ctx.ROWS_TILE_SIZE = 16  # Each thread processes 16 batch elements at a time
        ctx.input_shape = input_shape

        # Need to initialize empty result tensor. Note that these elements are not necessarily 0!
        y = torch.empty(output_dims, device=x.device)

        # Launch our kernel with n instances in our 1D grid.
        n_rows = y.numel()
        weighted_sum_fwd[(cdiv(n_rows, ctx.ROWS_TILE_SIZE),)](
            x, weight,
            y,
            x.stride(0), x.stride(1),
            weight.stride(0),
            y.stride(0),
            ROWS=n_rows, D=D,
            ROWS_TILE_SIZE=ctx.ROWS_TILE_SIZE, D_TILE_SIZE=ctx.D_TILE_SIZE,
        )

        return y.view(input_shape[:-1])
```

Notice that when we invoke the Triton kernel with `weighted_sum_fwd[(cdiv(n_rows, ctx.ROWS_TILE_SIZE),)]`, we define a so-called "launch grid" of thread blocks by passing the tuple `(cdiv(n_rows, ctx.ROWS_TILE_SIZE),)`. Then, we can access the thread block index with `tl.program_id(0)` in our kernel.

Backward pass Since we are defining our own kernel, we will also need to write our own backward function.

In the forward pass, we were given the inputs to our layer, and needed to compute its outputs. In the backward pass, recall that we will be given the gradients of the objective with respect to our outputs, and need to compute the gradient with respect to each of our inputs. In our case, our operation has as inputs a matrix $x : \mathbb{R}^{n \times h}$ and a weight vector $w : \mathbb{R}^h$. For short, let's call our operation $f(x, w)$, whose range is $\mathbb{R}^n$. Then, assuming we are given $\nabla_{f(x,w)}\mathcal{L}$, the gradient of loss $\mathcal{L}$ with respect to the output of our layer, we can apply the multivariate chain rule to obtain the following expressions for the gradients with respect to x and w:

$$(\nabla_x \mathcal{L})_{ij} = \sum_{k=1}^{n} \frac{\partial f(x,w)_k}{\partial x_{ij}} (\nabla_{f(x,w)}\mathcal{L})_k = w_j \cdot (\nabla_{f(x,w)}\mathcal{L})_i \tag{2}$$

$$(\nabla_w \mathcal{L})_j = \sum_{i=1}^{n} \frac{\partial f(x,w)_i}{\partial w_j} (\nabla_{f(x,w)}\mathcal{L})_i = \sum_{i=1}^{n} x_{ij} \cdot (\nabla_{f(x,w)}\mathcal{L})_i \tag{3}$$

This gives a simple formula for computing the backward pass. To obtain the backward step with respect to x, we apply Eq 2 and take the outer product of w and $\nabla_{f(x,w)}$. To compute the backward step with respect to w (i.e. $(\nabla_w \mathcal{L})_j$), we must multiply our input gradient by the corresponding output row.

Our kernel for the backward pass will start by defining all the block pointers and then computing $\nabla_x \mathcal{L}$:

```python
@triton.jit
def weighted_sum_backward(
    x_ptr, weight_ptr,  # Input
    grad_output_ptr,  # Grad input
    grad_x_ptr, partial_grad_weight_ptr,  # Grad outputs
    stride_xr, stride_xd,
    stride_wd,
    stride_gr,
    stride_gxr, stride_gxd,
    stride_gwb, stride_gwd,
    NUM_ROWS, D,
    ROWS_TILE_SIZE: tl.constexpr, D_TILE_SIZE: tl.constexpr,
):
    row_tile_idx = tl.program_id(0)
    n_row_tiles = tl.num_programs(0)

    # Inputs
    grad_output_block_ptr = tl.make_block_ptr(
        grad_output_ptr,
        shape=(NUM_ROWS,), strides=(stride_gr,),
        offsets=(row_tile_idx * ROWS_TILE_SIZE,),
        block_shape=(ROWS_TILE_SIZE,),
        order=(0,),
    )

    x_block_ptr = tl.make_block_ptr(
        x_ptr,
        shape=(NUM_ROWS, D,), strides=(stride_xr, stride_xd),
        offsets=(row_tile_idx * ROWS_TILE_SIZE, 0),
        block_shape=(ROWS_TILE_SIZE, D_TILE_SIZE),
        order=(1, 0),
    )

    weight_block_ptr = tl.make_block_ptr(
        weight_ptr,
        shape=(D,), strides=(stride_wd,),
        offsets=(0,), block_shape=(D_TILE_SIZE,),
        order=(0,),
    )

    grad_x_block_ptr = tl.make_block_ptr(
        grad_x_ptr,
        shape=(NUM_ROWS, D,), strides=(stride_gxr, stride_gxd),
        offsets=(row_tile_idx * ROWS_TILE_SIZE, 0),
        block_shape=(ROWS_TILE_SIZE, D_TILE_SIZE),
        order=(1, 0),
    )

    partial_grad_weight_block_ptr = tl.make_block_ptr(
        partial_grad_weight_ptr,
        shape=(n_row_tiles, D,), strides=(stride_gwb, stride_gwd),
        offsets=(row_tile_idx, 0),
        block_shape=(1, D_TILE_SIZE),
        order=(1, 0),
    )

    for i in range(tl.cdiv(D, D_TILE_SIZE)):
        grad_output = tl.load(grad_output_block_ptr, boundary_check=(0,), padding_option="zero")  # (ROWS_TILE_SIZE,)

        # Outer product for grad_x
        weight = tl.load(weight_block_ptr, boundary_check=(0,), padding_option="zero")  # (D_TILE_SIZE,)
        grad_x_row = grad_output[:, None] * weight[None, :]
        tl.store(grad_x_block_ptr, grad_x_row, boundary_check=(0, 1))

        # Reduce as many rows as possible for the grad_weight result
```

```
        row = tl.load(x_block_ptr, boundary_check=(0, 1), padding_option="zero")  # (ROWS_TILE_SIZE, D_TILE_SIZE)
        grad_weight_row = tl.sum(row * grad_output[:, None], axis=0, keep_dims=True)
        tl.store(partial_grad_weight_block_ptr, grad_weight_row, boundary_check=(1,))  # Never out of bounds for dim 0

        # Move the pointers to the next tile along D
        x_block_ptr = x_block_ptr.advance((0, D_TILE_SIZE))
        weight_block_ptr = weight_block_ptr.advance((D_TILE_SIZE,))
        partial_grad_weight_block_ptr = partial_grad_weight_block_ptr.advance((0, D_TILE_SIZE))
        grad_x_block_ptr = grad_x_block_ptr.advance((0, D_TILE_SIZE))
```

Computing the gradient ∇_x is simple, and we write the result to the appropriate tile of the output tensor. However, computing ∇_w is a bit more challenging. Each kernel instance is responsible for one row tile of x, but we now need to sum *across* rows of x. Instead of doing this sum directly in our backward pass, we will assume that `partial_grad_weight_ptr` contains an `n_row_tiles` $\times H$ matrix, where the first dimension is only reduced within a row tile from x. We reduce within the current row tile before writing to this tensor. Outside of the kernel, we reduce ∇_w using `torch.sum` to sum up the results from each row tile[1]. The final part of the `autograd.Function` is then relatively simple:

```
class WeightedSumFunc(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x, weight):
        # ... (defined earlier)

    @staticmethod
    def backward(ctx, grad_out):
        x, weight = ctx.saved_tensors
        ROWS_TILE_SIZE, D_TILE_SIZE = ctx.ROWS_TILE_SIZE, ctx.D_TILE_SIZE  # These don't have to be the same
        n_rows, D = x.shape

        # Our strategy is for each thread block to first write to a partial buffer,
        # then we reduce over this buffer to get the final gradient.
        partial_grad_weight = torch.empty((cdiv(n_rows, ROWS_TILE_SIZE), D), device=x.device, dtype=x.dtype)
        grad_x = torch.empty_like(x)

        weighted_sum_backward[(cdiv(n_rows, ROWS_TILE_SIZE),)](
            x, weight,
            grad_out,
            grad_x, partial_grad_weight,
            x.stride(0), x.stride(1),
            weight.stride(0),
            grad_out.stride(0),
            grad_x.stride(0), grad_x.stride(1),
            partial_grad_weight.stride(0), partial_grad_weight.stride(1),
            NUM_ROWS=n_rows, D=D,
            ROWS_TILE_SIZE=ROWS_TILE_SIZE, D_TILE_SIZE=D_TILE_SIZE,
        )
        grad_weight = partial_grad_weight.sum(axis=0)
        return grad_x, grad_weight
```

Finally, we can now obtain a function that works much like those implemented in `torch.nn.functional`:

```
f_weightedsum = WeightedSumFunc.apply
```

Now, calling `f_weightedsum` on two PyTorch tensors x and w will give a tensor such as the following:

```
tensor([ 90.8563, -93.6815, -80.8884,  ..., 103.4840, -21.4634, -24.0192],
       device='cuda:0', grad_fn=<WeightedSumFuncBackward>)
```

Note the `grad_fn` attached to the tensor — this shows that PyTorch knows what to call in the backward pass when this tensor appears in the computation graph. This completes our Triton implementation of the weighted sum operation.

[1] Or, of course, we could write our own kernel for that.

1.3.2 FlashAttention-2 Forward Pass

You will replace your PyTorch attention implementation with a significantly improved Triton implementation following FlashAttention-2 [Dao, 2023]. FlashAttention-2 employs some tricks to compute the forward pass in tiles, which allows for efficient memory access patterns and avoids the need to materialize the full attention matrix on global memory.

Before jumping into this section, we highly recommend reading atleast the original FlashAttention paper [Dao et al., 2022], which will give you intuition for the core technique that enables efficient attention with FlashAttention: computing the softmax in an online fashion across tiles (a technique proposed in [Milakov and Gimelshein, 2018]). We also recommend checking out He [2022] for some more intuition on how GPUs actually execute PyTorch code.

Understanding inefficiencies in vanilla attention. Recall that the forward pass for attention (ignoring masking for now) can be written as:

$$\mathbf{S} = \mathbf{Q}\mathbf{K}^\top/\sqrt{d} \tag{4}$$

$$\mathbf{P}_{ij} = \text{softmax}_j(\mathbf{S})_{ij} \tag{5}$$

$$\mathbf{O} = \mathbf{P}\mathbf{V} \tag{6}$$

The standard backward pass is

$$\mathbf{dV} = \mathbf{P}^\top\mathbf{dO} \tag{7}$$

$$\mathbf{dP} = \mathbf{dO}\mathbf{V}^\top \tag{8}$$

$$\mathbf{dS}_i = \text{dsoftmax}(\mathbf{dP}_i) = \left(\text{diag}(\mathbf{P}_i) - \mathbf{P}_i\mathbf{P}_i^\top\right)\mathbf{dP}_i \tag{9}$$

$$\mathbf{dQ} = \mathbf{dS}\mathbf{K}/\sqrt{d} \tag{10}$$

$$\mathbf{dK} = \mathbf{dS}^\top\mathbf{Q}/\sqrt{d}, \tag{11}$$

As we can see, the backward pass depends on some very large activations from the forward pass. For example, computing $\mathbf{dV}$ in (7) requires $\mathbf{P}$, which are the attention scores of shape `(batch_size, n_heads, seq_len, seq_len)`—the size of this activation matrix depends *quadratically* on the sequence length, explaining the memory issues we encountered above when benchmarking attention at large sequence lengths. During both the forward and backward pass of vanilla attention, we pay significant memory IO costs to transfer $\mathbf{P}$ and other large activations between on-chip SRAM and GPU HBM. There are *several* such transfers made in standard implementations: for example, a standard backward pass implementation would read $\mathbf{P}$ from HBM in the computations of both (7) and (9).

The main goal of FlashAttention is to avoid reading and writing the attention matrix to and from HBM, to reduce IO and peak memory costs. We accomplish this using three techniques: tiling, recomputation, and operator fusion.

Tiling. To avoid reading and writing the attention matrix to and from HBM, we compute the softmax reduction without access to the whole input. Specifically, we restructure the attention computation to split the input into tiles and make several passes over input tiles, thus incrementally performing the softmax reduction.

Recomputation. We avoid storing the large intermediate attention matrices of shape `(batch_size, n_heads, seq_len, seq_len)` in HBM. Instead, we will save certain "activation checkpoints" in HBM and then recompute part of the forward pass during the backward pass, to get the other activations we need for computing gradients. FlashAttention-2 also stores the logsumexp of the attention scores, L, which will be

used to simplify the backward pass computation. The expression for L is:

$$L_i = \log\left(\sum_j \exp\left(\mathbf{S}_{ij}\right)\right) \tag{12}$$

In our final kernel we will compute this in an online manner, but the final result should be the same. With tiling and recomputation together, our memory IO and peak usage no longer depend on `sequence_length`^{2} and therefore we may use larger sequence lengths.

Operator fusion. Lastly, we avoid repeated memory IO for the attention matrix and other intermediate activations by performing all our operations in a single kernel—this is referred to as operator or kernel fusion. We will write a single Triton kernel for the forward pass that performs all the operations involved in attention with limited data transfer between HBM and SRAM. Operator fusion is partly enabled by recomputation, since we can avoid the usual memory IO we would pay to store every intermediate activation to HBM.

For more intuition on these techniques, check out the FlashAttention papers [Dao et al., 2022, Dao, 2023].

Backward pass with recomputation. Using L, we can do the appropriate recomputation and compute the backward pass efficiently. Before we start the backward pass, we pre-compute into global memory the value $D = \text{rowsum}(\mathbf{O} \circ \mathbf{dO})$ (where $\circ$ is element-wise multiplication), which is equal to $\text{rowsum}(\mathbf{P} \circ \mathbf{dP})$ since $\mathbf{PdP}^\top = \mathbf{P}(\mathbf{dOV}^\top)^\top = (\mathbf{PV})\mathbf{dO}^\top = \mathbf{OdO}^\top$ (and $\text{rowsum}(\mathbf{A} \circ \mathbf{B}) = \text{diag}(\mathbf{AB}^\top)$ for any matrices $\mathbf{A}$ and $\mathbf{B}$). With the L and D vectors, the backward pass can be computed without softmax. The full calculation for the backward pass is now:

$$\mathbf{S} = \mathbf{QK}^\top/\sqrt{d} \tag{13}$$
$$\mathbf{P}_{ij} = \exp\left(\mathbf{S}_{ij} - L_i\right) \tag{14}$$
$$\mathbf{dV} = \mathbf{P}^\top\mathbf{dO} \tag{15}$$
$$\mathbf{dP} = \mathbf{dOV}^\top \tag{16}$$
$$\mathbf{dS}_{ij} = \mathbf{P}_{ij} \circ (\mathbf{dP}_{ij} - D_i) \tag{17}$$
$$\mathbf{dQ} = \mathbf{dSK}/\sqrt{d} \tag{18}$$
$$\mathbf{dK} = \mathbf{dS}^\top\mathbf{Q}/\sqrt{d}, \tag{19}$$

We can see that the sequence of operations does not require us to have stored the attention scores $\mathbf{P}$ in HBM during the forward pass—we recompute them from the activations $\mathbf{Q}$, $\mathbf{K}$, and L in (13) and (14).

Details of the flash attention forward pass. Now that we have a high level idea of the techniques used in FlashAttention-2, we will dive into the details of the FA2 forward pass kernel that you will implement. In order to avoid reading and writing the attention matrix to and from HBM, we wish to use tiling, i.e., computing each tile of the output independently of the others. This requires us to be able to compute tiles of P, ideally tiled in both dimensions (for queries and for keys).

However, when we apply softmax to S, we require entire rows of S to be reduced to compute the softmax denominator, meaning we cannot compute P in tiles directly. FlashAttention-2 solves this problem using *online softmax*. In the following text, we will use subscript index i to denote the current query tile, and superscript index (j) to denote the current key tile. The tiles along the query dimension will be of size B_q, and the key dimension, B_k. We will not tile along the hidden dimension d.

We also keep some row-wise running values, $m_i^{(j)} \in \mathbb{R}^{B_q}$ and $l_i^{(j)} \in \mathbb{R}^{B_q}$. The row-wise $m_i^{(j)}$ value is a running maximum, which is tracked so we can compute softmax in a numerically stable manner (recall this trick from our softmax implementation in Assignment 1). We will update $m_i^{(j)}$ with each new row-wise tile of S (when j increases). Using the running maximum, we can compute the unnormalized softmax values

(numerators) as $\tilde{\mathbf{P}}_i^{(j)} = \exp\left(\mathbf{S}_{ij} - m_i^{(j)}\right)$. $l_i^{(j)}$ is a running proxy for the softmax denominator, and will be updated using the unnormalized softmax values as $l_i^{(j)} = \exp(m_i^{(j-1)})$. When we finally write the output, we will need to finish normalizing it by using $l_i^{(T_k)}$, which is the final value of $l_i^{(j)}$ after processing all key tiles. Algorithm 1 shows the forward pass as it should be implemented on GPU.

Algorithm 1 FlashAttention-2 forward pass

Require: $\mathbf{Q} \in \mathbb{R}^{N_q \times d}$, $\mathbf{K}, \mathbf{V} \in \mathbb{R}^{N_k \times d}$, tile sizes B_q, B_k

Split $\mathbf{Q}$ into $T_q = \left\lceil \frac{N_q}{B_q} \right\rceil$ tiles $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_q}$ of size $B_q \times d$

Split $\mathbf{K}, \mathbf{V}$ into $T_k = \left\lceil \frac{N_k}{B_k} \right\rceil$ tiles $\mathbf{K}^{(1)}, \dots, \mathbf{K}^{(T_k)}$ and $\mathbf{V}^{(1)}, \dots, \mathbf{V}^{(T_k)}$ of size $B_k \times d$

for $i = 1, \dots, T_q$ **do**

  Load $\mathbf{Q}_i$ from global memory

  Initialize $\mathbf{O}_i^{(0)} = \mathbf{0} \in \mathbb{R}^{B_q \times d}, l_i^{(0)} = 0 \in \mathbb{R}^{B_q}, m_i^{(0)} = -\infty \in \mathbb{R}^{B_q}$

  for $j = 1, \dots, T_k$ **do**

    Load $\mathbf{K}^{(j)}, \mathbf{V}^{(j)}$ from global memory

    Compute tile of pre-softmax attention scores $\mathbf{S}_i^{(j)} = \frac{\mathbf{Q}_i (\mathbf{K}^{(j)})^\top}{\sqrt{d}} \in \mathbb{R}^{B_q \times B_k}$

    Compute $m_i^{(j)} = \max\left(m_i^{(j-1)}, \text{rowmax}\left(\mathbf{S}_i^{(j)}\right)\right) \in \mathbb{R}^{B_q}$

    Compute $\tilde{\mathbf{P}}_i^{(j)} = \exp\left(\mathbf{S}_i^{(j)} - m_i^{(j)}\right) \in \mathbb{R}^{B_q \times B_k}$

    Compute $l_i^{(j)} = \exp\left(m_i^{(j-1)} - m_i^{(j)}\right) l_i^{(j-1)} + \text{rowsum}\left(\tilde{\mathbf{P}}_i^{(j)}\right) \in \mathbb{R}^{B_q}$

    Compute $\mathbf{O}_i^{(j)} = \text{diag}\left(\exp\left(m_i^{(j-1)} - m_i^{(j)}\right)\right) \mathbf{O}_i^{(j-1)} + \tilde{\mathbf{P}}_i^{(j)} \mathbf{V}^{(j)}$

  end for

  Compute $\mathbf{O}_i = \text{diag}\left(l_i^{(T_k)}\right)^{-1} \mathbf{O}_i^{(T_k)}$

  Compute $L_i = m_i^{(T_k)} + \log\left(l_i^{(T_k)}\right)$

  Write $\mathbf{O}_i$ to global memory as the i-th tile of $\mathbf{O}$.

  Write L_i to global memory as the i-th tile of L.

end for

Return the output $\mathbf{O}$ and the logsumexp L.

Before we get into implementing the forward pass in Triton, we collect here a few general tips and tricks for writing Triton kernels.

Triton Tips and Tricks

- You can use print statements in Triton with `tl.device_print` to debug: https://triton-lang.org/main/python-api/generated/triton.language.device_print.html. There is a setting `TRITON_INTERPRET=1` to run the Triton interpreter on CPU, though we have found it buggy.
- When defining block pointers, make sure they have the correct offsets, and that block offsets are multiplied by the appropriate tile sizes.
- The launch grid of thread blocks is set with

  ```
  kernel_fn[(launch_grid_d1, launch_grid_d2, ...)](...arguments...)
  ```

  in the methods of the `torch.autograd.Function` subclass, as we saw in the weighted sum example.

- Perform matrix multiplications with `tl.dot`.
- To advance a block pointer, use `*_block_ptr = *_block_ptr.advance(...)`

Problem (`flash_forward`): 15 points

Done

(a) Write a pure PyTorch (no Triton) `autograd.Function` that implements the FlashAttention-2 forward pass. This will be a lot slower than the regular PyTorch implementation, but will help you debug your Triton kernel.

Your implementation should take input **Q**, **K**, and **V** as well as a flag `is_causal` and produce the output **O** and the logsumexp value L. You can ignore the `is_causal` flag for this task. The `autograd.Function` forward should then use save L, Q, K, V, O for the backward pass and return O. Remember that the implementation of the `forward` method of `autograd.Function` always takes the context as its first parameter. Any `autograd.Function` class needs to implement a `backward` method, but for now you can make it just raise `NotImplementedError`. If you need something to compare against, you can implement Equation 4 to 6 and 12 in PyTorch and compare your outputs.

The interface is then `def forward(ctx, Q, K, V, is_causal=False)`. Determine your own tile sizes, but make sure they are at least of size 16×16. We will always test your code with dimensions that are clean powers of 2 and at least 16, so you don't need to worry about out-of-bounds accesses.

Deliverable: A `torch.autograd.Function` subclass that implements FlashAttention-2 in the forward pass. To test your code, implement `[adapters.get_flashattention_autograd_function_pytorch]`. Then, run the test with `uv run pytest -k test_flash_forward_pass_pytorch` and make sure your implementation passes it.

(b) Write a Triton kernel for the forward pass of FlashAttention-2 following Algorithm 1. Then, write another subclass of `torch.autograd.Function` that calls this (fused) kernel in the forward pass, instead of computing the result in PyTorch. A few problem-specific tips:

- To debug, we suggest comparing the results of each Triton operation you perform with the tiled PyTorch implementation you wrote in part (a).
- Your launch grid should be set as $(T_q, \texttt{batch\_size})$, meaning each Triton program instance will load only elements from a single batch index, and only read/write to a single query tile of **Q**, **O**, and L.
- The kernel should only have a single loop, which will iterate key tiles $1 \leq j \leq T_k$.
- Advance block pointers at the end of the loop.
- Use the function declaration below (using the block pointer we give you, you should be able to infer the setup of the rest of the pointers):

```
@triton.jit
def flash_fwd_kernel(
    Q_ptr, K_ptr, V_ptr,
    O_ptr, L_ptr,
    stride_qb, stride_qq, stride_qd,
    stride_kb, stride_kk, stride_kd,
    stride_vb, stride_vk, stride_vd,
    stride_ob, stride_oq, stride_od,
    stride_lb, stride_lq,
```

Done

```
    N_QUERIES, N_KEYS,
    scale,
    D: tl.constexpr,
    Q_TILE_SIZE: tl.constexpr,
    K_TILE_SIZE: tl.constexpr,
):
    # Program indices
    query_tile_index = tl.program_id(0)
    batch_index = tl.program_id(1)

    # Offset each pointer with the corresponding batch index
    # multiplied with the batch stride for each tensor
    Q_block_ptr = tl.make_block_ptr(
        Q_ptr + batch_index * stride_qb,
        shape=(N_QUERIES, D),
        strides=(stride_qq, stride_qd),
        offsets=(query_tile_index * Q_TILE_SIZE, 0),
        block_shape=(Q_TILE_SIZE, D),
        order=(1, 0),
    )

    ...
```

where `scale` is $\frac{1}{\sqrt{d}}$ and `Q_TILE_SIZE` and `K_TILE_SIZE` are B_q and B_k respectively. You can tune these later.

These additional guidelines may help you avoid precision issues:

- The on chip buffers $(\mathbf{O}_i, l, m)$ should have `dtype` `tl.float32`. If you're accumulating into an output buffer, use the `acc` argument (`acc = tl.dot(..., acc=acc)`).
- Cast $\tilde{\mathbf{P}}_i^{(j)}$ to the `dtype` of $\mathbf{V}^{(j)}$ before multiplying them, and cast $\mathbf{O}_i$ to the appropriate `dtype` before writing it to global memory. Casting is done with `tensor.to`. You can get the `dtype` of a tensor with `tensor.dtype`, and the dtype of a block pointer/pointer with `*_block_ptr.type.element_ty`.

Deliverable: A `torch.autograd.Function` subclass that implements FlashAttention-2 in the forward pass using your Triton kernel. Implement `[adapters.get_flash_autograd_function_triton]`. Then, run the test with `uv run pytest -k test_flash_forward_pass_triton` and make sure your implementation passes it.

Done

(c) Add a flag as the last argument to your `autograd.Function` implementation for causal masking. This should be a boolean flag that when set to `True` enables an index comparison for causal masking. Your Triton kernel should have a corresponding additional parameter `is_causal: tl.constexpr` (this is a required type annotation). In Triton, construct appropriate index vectors for queries and keys, and compare them to form a square mask of size $B_q \times B_k$. For elements that are masked out, add the constant value of `-1e6` to the corresponding elements of the attention score matrix $\mathbf{S}_i^{(j)}$. Make sure to save the mask flag for backward using `ctx.is_causal = is_causal`.

Deliverable: An additional flag for your `torch.autograd.Function` subclass that implements the FlashAttention-2 forward pass with causal masking using your Triton kernel. Make sure that the flag is optional with default `False` so the previous tests still pass.

Implementing the backward pass with recomputation Notice that unlike the standard backward pass in Eq 7 to 11, we can use recomputation to avoid the softmax operation in the backward pass shown in Eq 13 to 19. This means that we can compute the backward pass using a trivial kernel, and no online tricks are required. Thus, for this part, you can implement backward by calling `torch.compile` on a regular

PyTorch function (not Triton).

Problem (`flash_backward`): 5 points Done

Implement the backward pass for your FlashAttention-2 `autograd.Function` using PyTorch (not Triton) and `torch.compile`. Your implementation should take the **Q**, **K**, **V**, **O**, **dO**, and L tensors as output, and return **dQ**, **dK** and **dV**. Remember to compute and use the D vector. You may follow along the computations of Equations 13 to 19.
Deliverable: To test your implementation, run `uv run pytest -k test_flash_backward`.

Let's now compare the performance of your (partially) Triton implementation of FlashAttention-2 with your PyTorch implementation of regular Attention.

Problem (`flash_benchmarking`): 5 points

Done

(a) 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`. You will likely need to adjust tile sizes depending on the input sizes.

Deliverable: A table of results comparing your implementation of FlashAttention-2 with the PyTorch implementation, using the settings above and reporting forward, backward, and end-to-end latencies.

1.3.3 FlashAttention-2 Leaderboard

Assignment 2's leaderboard will test the speed of your implementation of FlashAttention-2 (including both the forward and backward passes). We challenge you to further improve the performance of your implementation, using any tricks you can come up with. The restrictions are that you cannot change the input/outputs of the function, and you must use Triton (no CUDA, unfortunately). Your inputs will be tested at BF16 with causal masking, and it must pass the same tests as your regular implementation. The implementation must also be your own, and you cannot use pre-existing implementations. Your timing should be measured on H100 on a sample with batch size 1, sequence length 16,384 for queries, keys, and values, and $d_{\text{model}} = 1024$ with 16 heads. We will verify the top 5-10 submissions for correctness and performance. The test we will run to time your implementation is the following:

```
def test_timing_flash_forward_backward():
    n_heads = 16
    d_head = 64
    sequence_length = 16384
    q, k, v = torch.randn(
        3, n_heads, sequence_length, d_head, device='cuda', dtype=torch.bfloat16, requires_grad=True
    )

    flash = torch.compile(FlashAttention2.apply)
```

```
    def flash_forward_backward():
        o = flash(q, k, v, True)
        loss = o.sum()
        loss.backward()

    results = triton.testing.do_bench(flash_forward_backward, rep=10000, warmup=1000)
    print(results)
```

For testing purposes, you can reduce the repetition and warmup time (given in ms) to a something shorter. Some ideas for improvement:

- Tune the tile sizes for your kernel (use Triton autotune for this!)
- Tune additional Triton config parameters
- Implement the backward pass in Triton, not just `torch.compile` (see Section 1.3.4 below)
- Do two passes over your input for the backward pass, one for **dQ** and another for **dK** and **dV** to avoid atomics or synchronization between blocks.
- Stop program instances early when doing causal masking, skipping all tiles that are always all zero
- Separate the non-masked tiles from the tile diagonals, computing the first without ever comparing indices, and the second with a single comparison
- Use TMA (Tensor Memory Accelerator) functionality on H100, following a similar pattern to this tutorial.

Submit your best times to the leaderboard at

`github.com/stanford-cs336/assignment2-systems-leaderboard`

1.3.4 OPTIONAL: Triton backward pass

If you're interested in getting more practice with Triton and/or having a fast leaderboard submission, we provide the tiled FlashAttention-2 backward pass below which you can implement in Triton. Algorithm 2 shows the FlashAttention-2 backward pass as it should be implemented in Triton. A key trick here is to compute **P** twice, once for the backward pass for **dQ** and another time for **dK** and **dV**. This lets us skip synchronization across thread blocks.

Algorithm 2 Tiled FlashAttention-2 backward pass

Require: $\mathbf{Q}, \mathbf{O}, \mathbf{dO} \in \mathbb{R}^{N_q \times d}$, $\mathbf{K}, \mathbf{V} \in \mathbb{R}^{N_k \times d}$, $L \in \mathbb{R}^{N_q}$, tile sizes B_q, B_k

Compute $D = \text{rowsum}(\mathbf{dO} \circ \mathbf{O}) \in \mathbb{R}^{N_q}$

Split $\mathbf{Q}, \mathbf{O}, \mathbf{dO}$ into $T_q = \left\lceil \frac{N_q}{B_q} \right\rceil$ tiles $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_q}$, $\mathbf{O}_1, \dots, \mathbf{O}_{T_q}$, $\mathbf{dO}_1, \dots, \mathbf{dO}_{T_q}$, each of size $B_q \times d$

Split $\mathbf{K}, \mathbf{V}$ into $T_k = \left\lceil \frac{N_k}{B_k} \right\rceil$ tiles $\mathbf{K}^{(1)}, \dots, \mathbf{K}^{(T_k)}$ and $\mathbf{V}^{(1)}, \dots, \mathbf{V}^{(T_k)}$, each of size $B_k \times d$

Split L, D into T_q tiles $L_1, \dots, L_{T_q}$ and $D_1, \dots, D_{T_q}$, each of size B_q

for $j = 1, \dots, T_k$ **do**

  Load $\mathbf{K}^{(j)}, \mathbf{V}^{(j)}$ from global memory

  Initialize $\mathbf{dK}^{(j)} = \mathbf{dV}^{(j)} = \mathbf{0} \in \mathbb{R}^{B_k \times d}$

  for $i = 1, \dots, T_q$ **do**

    Load $\mathbf{Q}_i, \mathbf{O}_i, \mathbf{dO}_i, \mathbf{dQ}_i$ from global memory

    Compute tile of attention scores $\mathbf{S}_i^{(j)} = \frac{\mathbf{Q}_i (\mathbf{K}^{(j)})^\top}{\sqrt{d}} \in R^{B_q \times B_k}$

    Compute attention probabilities $\mathbf{P}_i^{(j)} = \exp\left(\mathbf{S}_i^{(j)} - L_i\right) \in \mathbb{R}^{B_q \times B_k}$

    Compute $\mathbf{dV}^{(j)} \mathrel{+}= (\mathbf{P}_i^{(j)})^\top \mathbf{dO}_i \in \mathbb{R}^{B_k \times d}$

    Compute $\mathbf{dP}_i^{(j)} = \mathbf{dO}_i \mathbf{V}_j^\top \in \mathbb{R}^{B_q \times B_k}$

    Compute $\mathbf{dS}_i^{(j)} = \mathbf{P}_i^{(j)} \circ \left(\mathbf{dP}_i^{(j)} - D_i\right) / \sqrt{d} \in \mathbb{R}^{B_q \times B_k}$

    Load $\mathbf{dQ}_i$ from global memory, then update $\mathbf{dQ}_i \mathrel{+}= \mathbf{dS}_i^{(j)} \mathbf{K}^{(j)} \in \mathbb{R}^{B_q \times d}$, and write back to global memory. Must be atomic for correctness!

    Compute $\mathbf{dK}^{(j)} \mathrel{+}= (\mathbf{dS}_i^{(j)})^\top \mathbf{Q}_i \in \mathbb{R}^{B_k \times d}$.

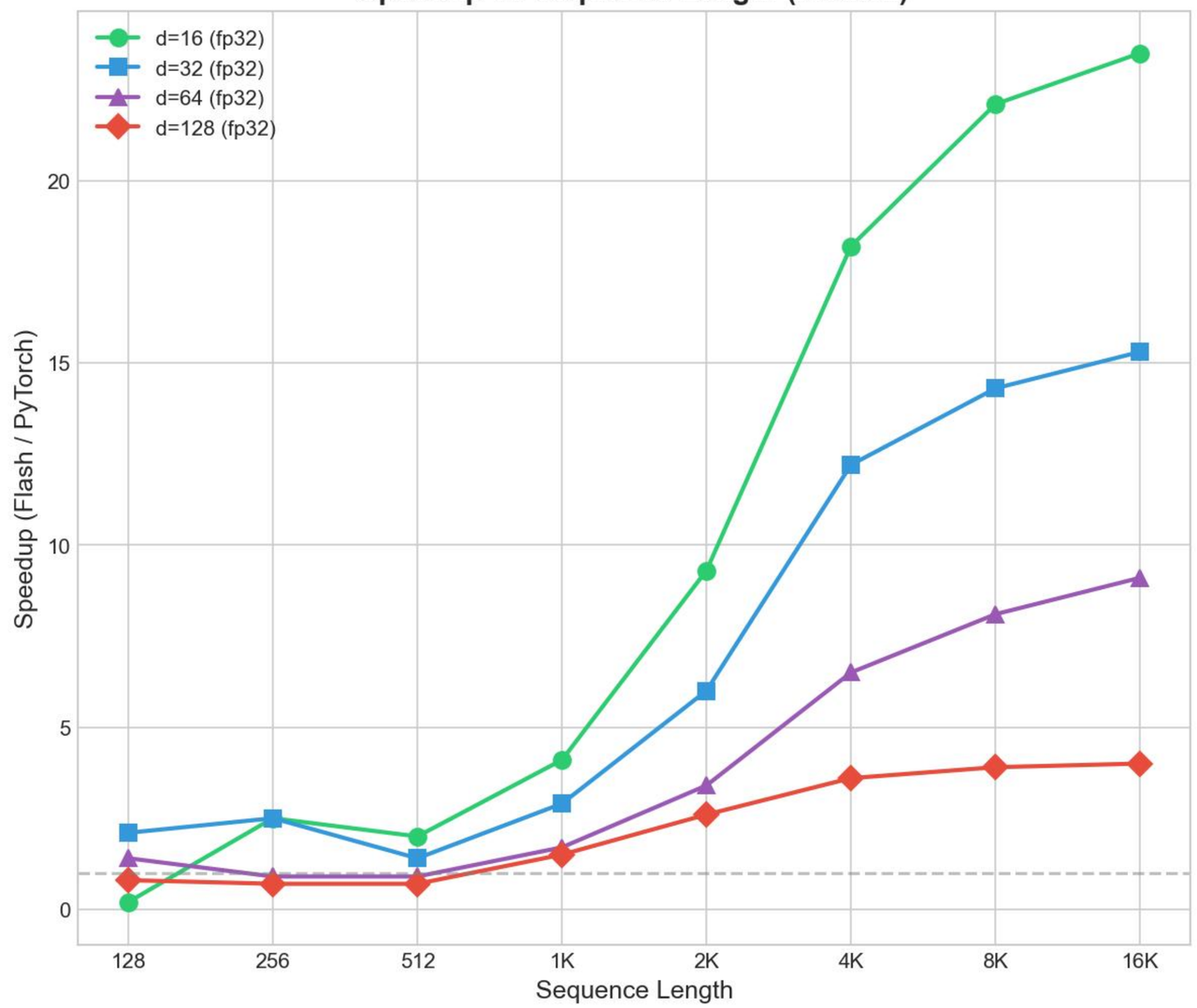
  end for

  Write $\mathbf{dK}^{(j)}$ and $\mathbf{dV}^{(j)}$ to global memory as the j-th tiles of $\mathbf{dK}$ and $\mathbf{dV}$.

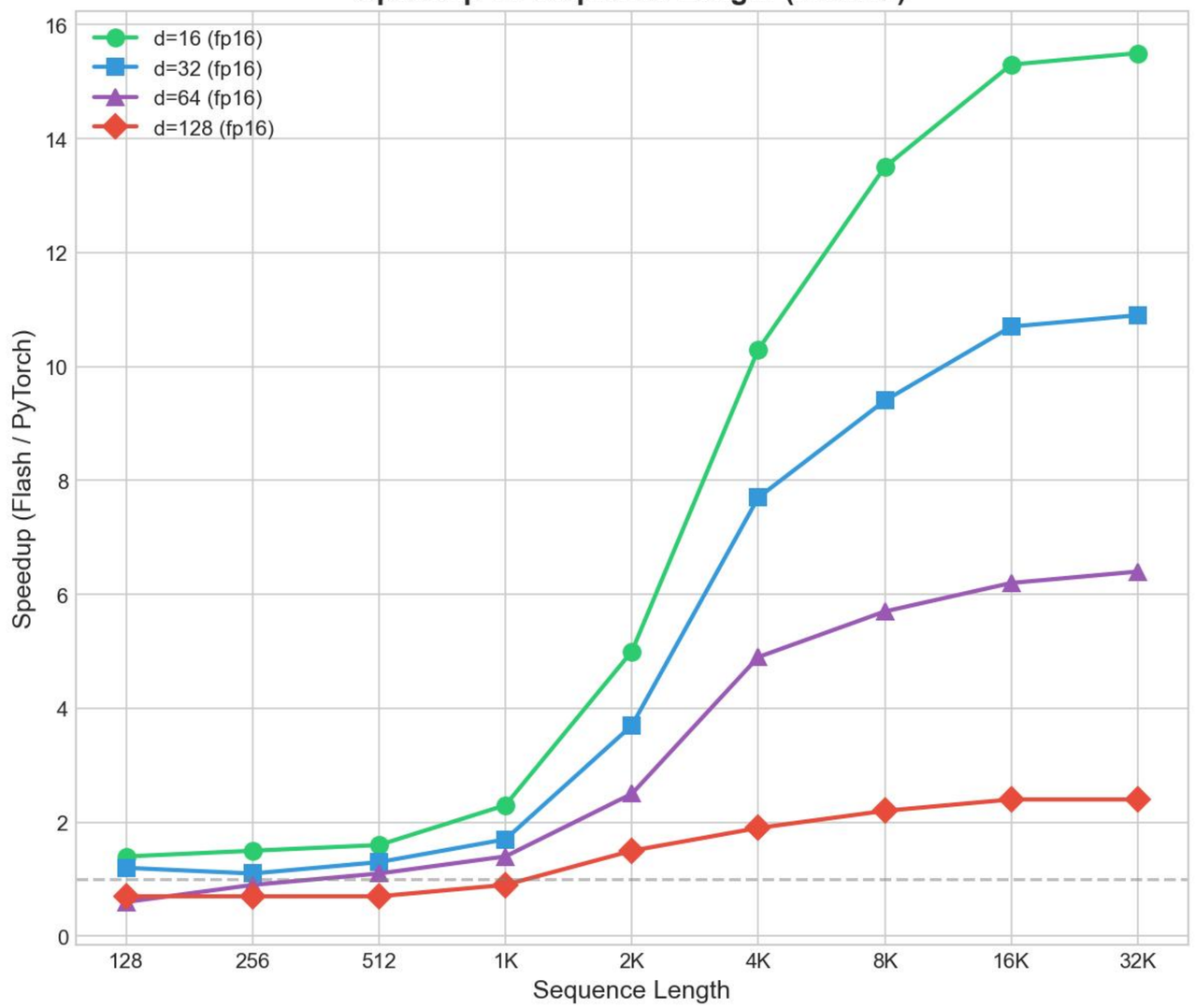
end for

Return $\mathbf{dQ}, \mathbf{dK}, \mathbf{dV}$.

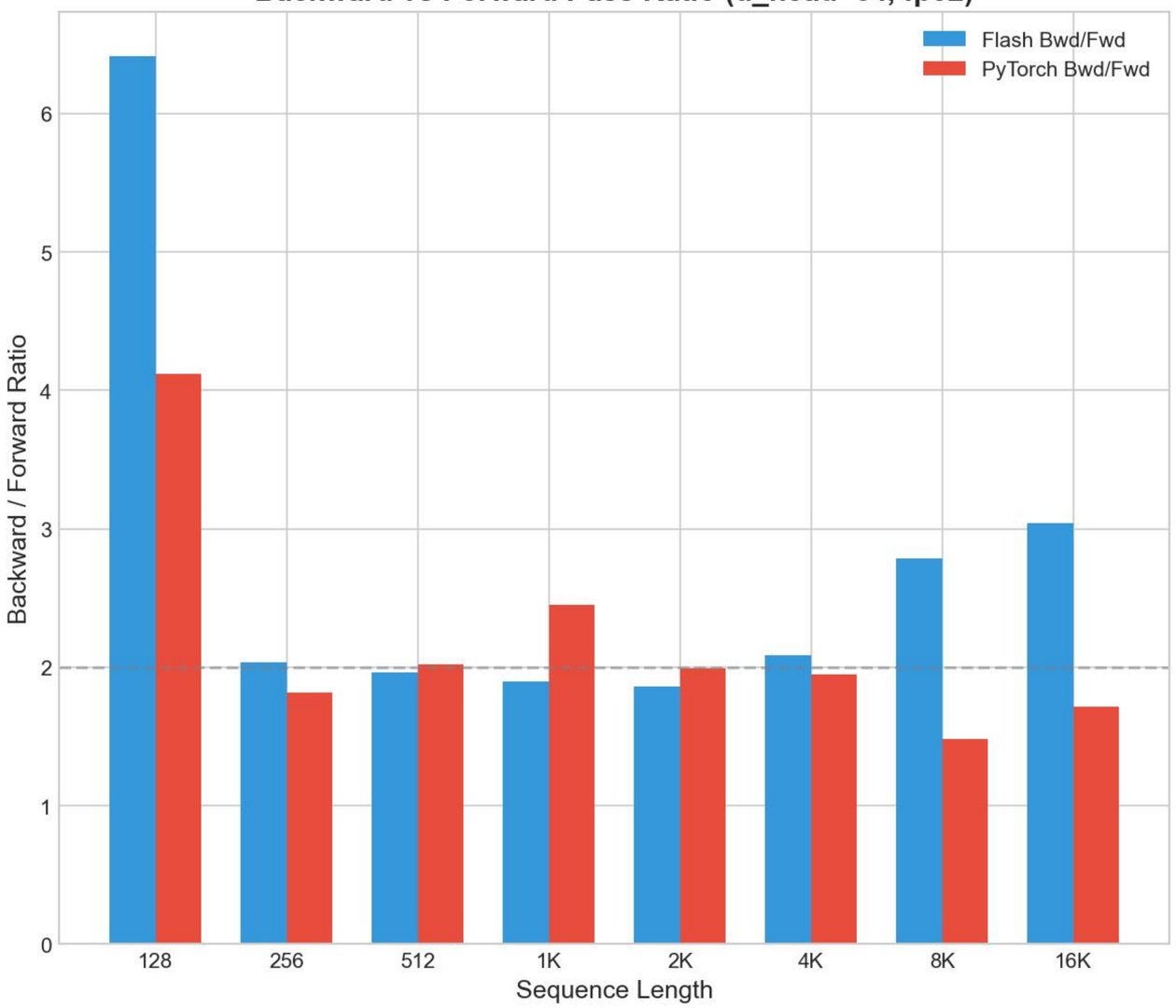
Speedup vs Sequence Length (Float32)



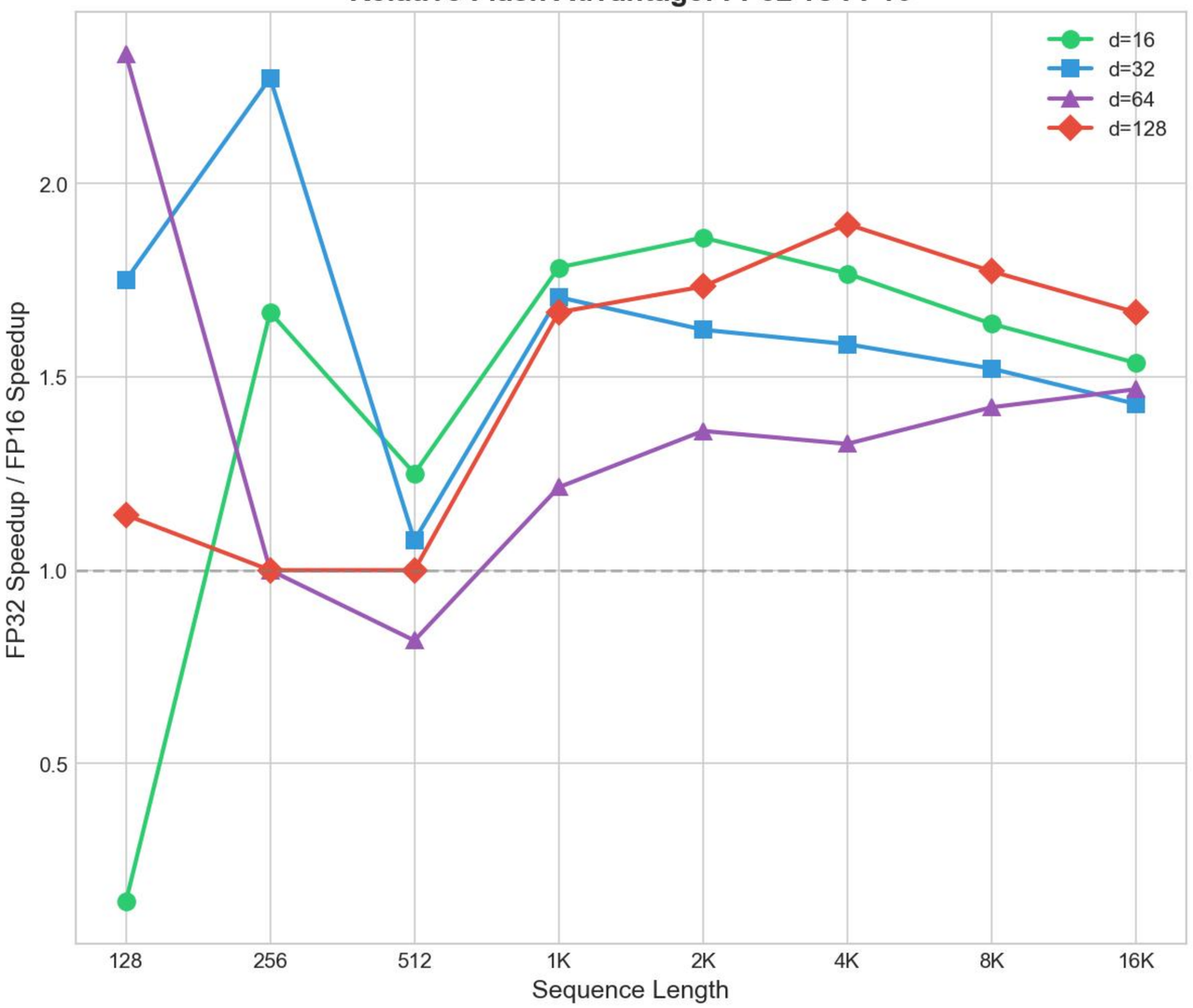
Speedup vs Sequence Length (Float16)

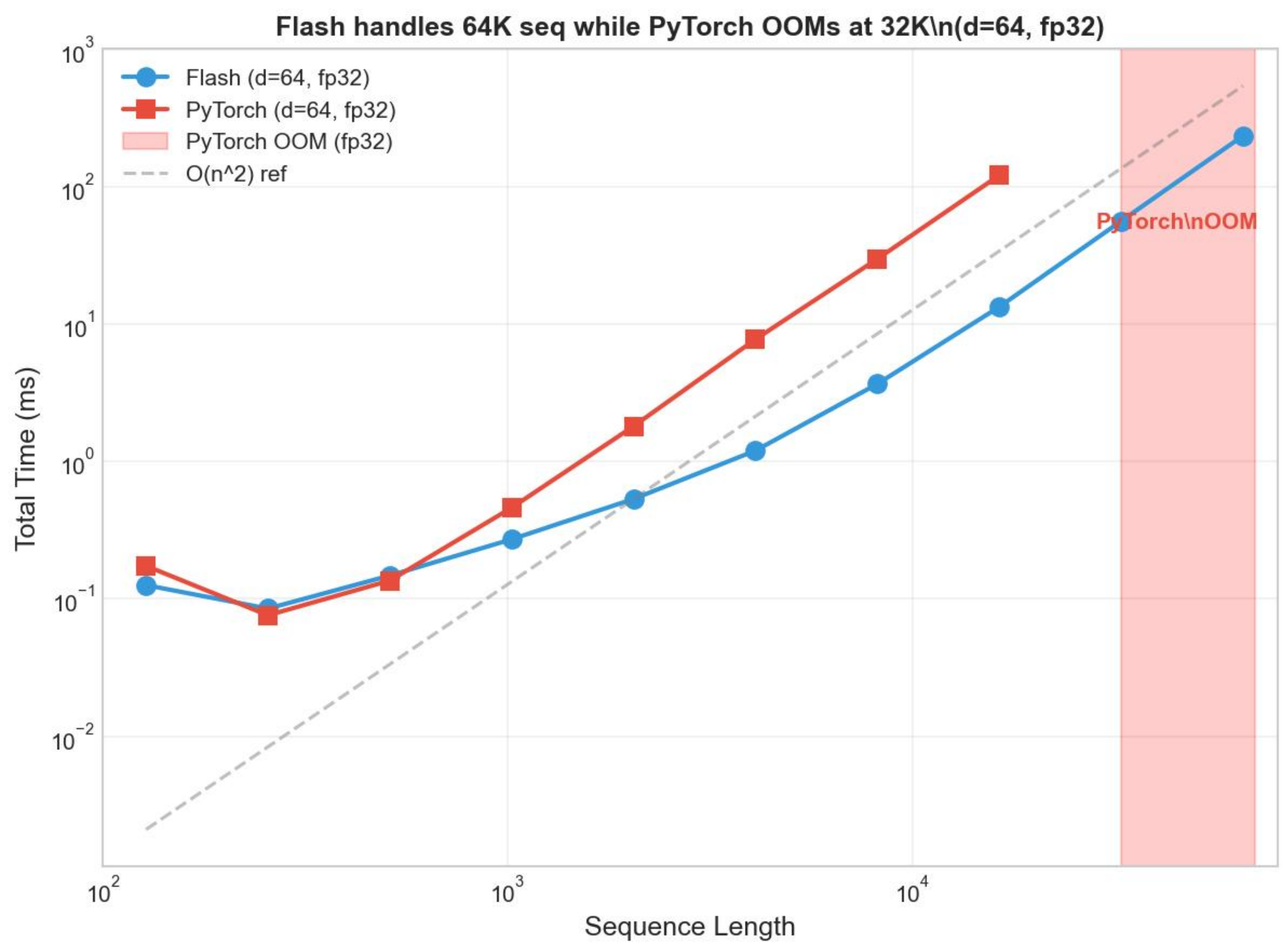


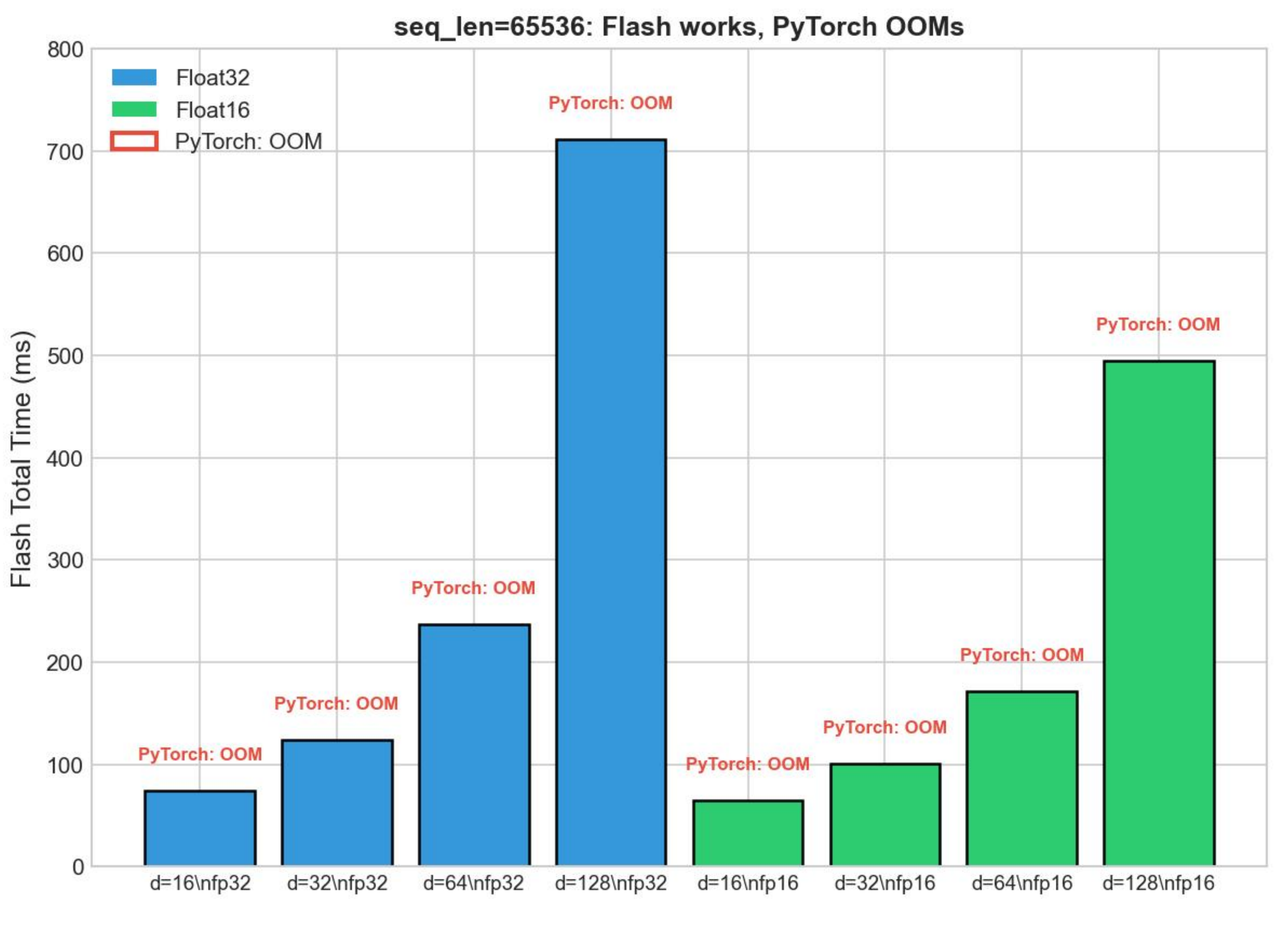
Backward vs Forward Pass Ratio (d_head=64, fp32)



Relative Flash Advantage: FP32 vs FP16




Flash handles 64K seq while PyTorch OOMs at 32K\n(d=64, fp32)
Flash (d=64, fp32)
PyTorch (d=64, fp32)
PyTorch OOM (fp32)
O(n^2) ref
PyTorch\nOOM
Total Time (ms)
Sequence Length


seq_len=65536: Flash works, PyTorch OOMs
Float32
Float16
PyTorch: OOM
Flash Total Time (ms)
d=16\nfp32
d=32\nfp32
d=64\nfp32
d=128\nfp32
d=16\nfp16
d=32\nfp16
d=64\nfp16
d=128\nfp16

2 Distributed Data Parallel Training

In this next part of the assignment, we'll explore methods for using multiple GPUs to train our language models, focusing on data parallelism. We'll start with a primer on distributed communication in PyTorch. Then, we'll study a naive implementation of distributed data parallel training and then implement and benchmark various improvements for improving communication efficiency.

2.1 Single-Node Distributed Communication in PyTorch

Let's start by looking at a simple distributed application in PyTorch, where the goal is to generate four random integer tensors and compute their sum.

In the distributed case below, we will spawn four worker processes, each of which generates a random integer tensor. To sum these tensors across the worker processes, we will call the **all-reduce** collective communication operation, which replaces the original data tensor on each process with the all-reduced result (i.e., the sum).

Now let's take a look at some code.

```python
import os
import torch
import torch.distributed as dist
import torch.multiprocessing as mp

def setup(rank, world_size):
    os.environ["MASTER_ADDR"] = "localhost"
    os.environ["MASTER_PORT"] = "29500"
    dist.init_process_group("gloo", rank=rank, world_size=world_size)

def distributed_demo(rank, world_size):
    setup(rank, world_size)
    data = torch.randint(0, 10, (3,))
    print(f"rank {rank} data (before all-reduce): {data}")
    dist.all_reduce(data, async_op=False)
    print(f"rank {rank} data (after all-reduce): {data}")

if __name__ == "__main__":
    world_size = 4
    mp.spawn(fn=distributed_demo, args=(world_size, ), nprocs=world_size, join=True)
```

After running the script above, we get the output below. As expected, each worker process initially holds different `data` tensors. After the all-reduce operation, which sums the tensors across all of the worker processes, `data` is modified in-place on each of the worker processes to hold the all-reduced result.[2]

```
$ uv run python distributed_hello_world.py
rank 3 data (before all-reduce): tensor([3, 7, 8])
rank 0 data (before all-reduce): tensor([4, 4, 7])
rank 2 data (before all-reduce): tensor([6, 0, 7])
rank 1 data (before all-reduce): tensor([9, 5, 3])
rank 1 data (after all-reduce): tensor([22, 16, 25])
rank 0 data (after all-reduce): tensor([22, 16, 25])
```

[2] If you run this script multiple times, you'll notice that the order of the printed output is not deterministic. Since this application is running in a distributed setting, we cannot control the exact order in which commands are being run—our only guarantee is that after the all-reduce operation is complete, the separate processes will hold bitwise identical result tensors.

```
rank 3 data (after all-reduce): tensor([22, 16, 25])
rank 2 data (after all-reduce): tensor([22, 16, 25])
```

Let's now look back more closely at our script above. The command `mp.spawn` spawns `nprocs` processes that run `fn` with the provided `args`. In addition, the function `fn` is called as `fn(rank, *args)`, where rank is the index of the worker process (a value between 0 and `nprocs-1`). Thus, our `distributed_demo` function must accept this integer rank as its first positional argument. In addition, we pass in the `world_size`, which refers to the total number of worker processes.

Each of these worker processes belong to a *process group*, which is initialized via `dist.init_process_group`. The process group represents multiple worker processes that will coordinate and communicate via a shared master. The master is defined by its IP address and port, and the master runs the process with rank 0. Collective communication operations like all-reduce operate on each process in the process group.

In this case, we initialized our process group with the `"gloo"` backend, but other backends are available. In particular, the `"nccl"` backend will use the NVIDIA NCCL collective communications library, which will generally be more performant for CUDA tensors. However, NCCL can only be used on machines with GPUs, while Gloo can be run on CPU-only machines. A useful rule of thumb is to use NCCL for distributed GPU training, and Gloo for distributed CPU training and/or local development. We used Gloo in this example because it enables local execution and development on CPU-only machines.

When running multi-GPU jobs, make sure that different ranks use different GPUs. One method for doing this is to call `torch.cuda.set_device(rank)`) in the setup function, so that `tensor.to("cuda")` will automatically move it to the specified device. Alternatively, you can explicitly create a per-rank device string (e.g., `device = f"cuda:{rank}"`), and then use this device string as the target device for any data movement (e.g., `tensor.to(f"cuda:{rank}")`).

Terminology. In the rest of the assignment (and various other resources you might see online), you may encounter the following terms in the context of PyTorch distributed communication. Though we will focus on single-node, multi-process distributed training in this assignment, the terminology is useful for understanding distributed training in general. See Figure 2 for a visual representation.

node: a machine on the network.

worker: an instance of a program that's participating in the distributed training. In this assignment, each worker will have a single process, so we'll use *worker*, *process*, and *worker process* interchangeably. However, a worker may use multiple processes (e.g., to load data for training), so these terms are not always equivalent in practice.

world size: The number of total workers in a process group.

global rank: An integer ID (between 0 and `world_size-1`) that uniquely identifies a worker in the process group. For example, for world size of two, one process will have global rank 0 (the master process) and the other process will have rank 1.

local world size: When running applications across different nodes, the local world size is the number of workers running locally on a given node. For example, if we have an application that spawns 4 workers on 2 nodes each, the world size would be 8 and the local world size would be 4. Note that when running on a single node, the local world size of a worker is equivalent to the (global) world size.

local rank: An integer ID (between 0 and `local_world_size-1`) that uniquely identifies the index of a local worker on the machine. For example, if we have an application that spawns 4 processes on 2 nodes each, the each node would have workers with local ranks 0, 1, 2, and 3. Note that when running a single-node multi-process distributed application, the local rank of a process is equivalent to its global rank.

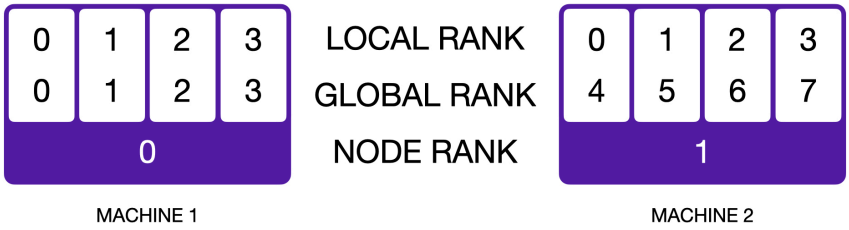


Figure 2: A schematic representation of a distributed application running on 2 nodes with a world size of 8. Each worker process is identified by a global rank (from 0 to 7) and a local rank (from 0 to 3). Figure taken from lightning.ai/docs/fabric/stable/advanced/distributed_communication.html

2.1.1 Best Practices for Benchmarking Distributed Applications

Throughout this portion of the assignment you will be benchmarking distributed applications to better understand the overhead from communication. Here are a few best practices:

- Whenever possible, run benchmarks on the same machine to facilitate controlled comparisons.
- Perform several warm-up steps before timing the operation of interest. This is especially important for NCCL communication calls. 5 iterations of warmup is generally sufficient.
- Call `torch.cuda.synchronize()` to wait for CUDA operations to complete when benchmarking on GPUs. Note that this is necessary even when calling communication operations with `async_op=False`, which returns when the operation is queued on the GPU (as opposed to when the communication actually finishes).[3]
- Timings may vary slightly across different ranks, so it's common to aggregate measurements across ranks to improve estimates. You may find the all-gather collective (specifically the `dist.all_gather_object` function) to be useful for collecting results from all ranks.
- In general, debug locally with Gloo on CPU, and then as required in a given problem, benchmark with NCCL on GPU. Switching between the backends just involves changing the `init_process_group` call and tensor device casts.

Problem (`distributed_communication_single_node`): 5 points

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.

[3] See github.com/pytorch/pytorch/issues/68112#issuecomment-965932386 for more details.

Runtime grows with tensor size and world size. For large tensor sizes where bandwith gets saturated the runtime grows drastically

Deliverable: Plot(s) and/or table(s) comparing the various settings, with 2-3 sentences of commentary about your results and thoughts about how the various factors interact.

2.2 A Naïve Implementation of Distributed Data Parallel Training

Now that we've seen the basics of writing distributed applications in PyTorch, let's build a minimal implementation of distributed data parallel (DDP) training.

Data parallelism splits batches across multiple devices (e.g., GPUs), enabling training on large batch sizes that do not fit on a single device. For example, given four devices that can each handle a maximum batch size of 32, data parallel training would enable an effective batch size of 128.

Here are the steps for naïvely doing distributed data parallel training. Initially, each device constructs a (randomly-initialized) model. We use the broadcast collective communication operation to send the model parameters from rank 0 to all other ranks. At the start of training, each device holds an identical copy of the model parameters and optimizer states (e.g. the accumulated gradient statistics in Adam).

1. Given a batch with n examples, the batch is sharded and each device receives n/d disjoint examples (where d is the number of devices used for data parallel training). n should divide d, since the training time is bottlenecked by the slowest process.

2. Each device uses its local copy of the model parameters to run a forward pass on its n/d examples and a backward pass to calculate the gradients. Note that at this point, each device holds the gradients computed from the n/d examples it received.

3. We then use the all-reduce collective communication operation to average the gradients across the different devices, so each device holds the gradients averaged across all n examples.

4. Next, each device runs an optimizer step to update its copy of the parameters—from the optimizer's perspective, it is simply optimizing a local model. The parameters and optimizer states will stay in sync on all of the different devices since they all start from the same initial model and optimizer state and use the same averaged gradients for each iteration. At this point, we've completed a single training iteration and can repeat the process.

Problem (`naive_ddp`): 5 points Done

Deliverable: Write a script to naively perform distributed data parallel training by all-reducing individual parameter gradients after the backward pass. To verify the correctness of your DDP implementation, use it to train a small toy model on randomly-generated data and verify that its weights match the results from single-process training.[a]

[a] If you're having trouble writing this test, it might be useful to look at `tests/test_ddp_individual_parameters.py`

Problem (`naive_ddp_benchmarking`): 3 points

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.

Deliverable: A description of your benchmarking setup, along with the measured time per training

Avg time per training step with naive DDP: 5.108605e-01 s
Time spent transfering gradients: 1.358289e-01 s . This equals 26.59 % of total time
Avg time per training step on single process: 6.926381e-01 s

iteration and time spent communicating gradients for each setting.

2.3 Improving Upon the Minimal DDP Implementation

The minimal DDP implementation that we saw in §2.2 has a couple of key limitations:

1. It conducts a separate all-reduce operation for every parameter tensor. Each communication call incurs overhead, so it may be advantageous to batch communication calls to minimize this overhead.
2. It waits for the backward pass to finish before communicating gradients. However, the backward pass is incrementally computed. Thus, when a parameter gradient is ready, it can immediately be communicated without waiting for the gradients of the other parameters. This allows us to overlap communication of gradients with computation of the backward pass, reducing the overhead of distributed data parallel training.

In this part of the assignment, we'll address each of these limitations in turn and measure the impact on training speed.

2.3.1 Reducing the Number of Communication Calls

Rather than issuing a communication call for each parameter tensor, let see if we can improve performance by batching the all-reduce. Concretely, we'll take the gradients that we want to all-reduce, concatenate them into a single tensor, and then all-reduce the combined gradients across all ranks. It might be helpful to use `torch._utils._flatten_dense_tensors` and `torch._utils._unflatten_dense_tensors`.

Problem (`minimal_ddp_flat_benchmarking`): 2 points

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 all-reduce for each parameter tensor under the previously-used conditions (1 node x 2 GPUs, XL model size as described in §1.1.2).

Deliverable: The measured time per training iteration and time spent communicating gradients under distributed data parallel training with a single batched all-reduce call. 1-2 sentences comparing the results when batching vs. individually communicating gradients.

2.3.2 Overlapping Computation with Communication of Individual Parameter Gradients

While batching the communication calls might help lower the overhead associated with issuing a large number of small all-reduce operations, all of communication time still directly contributes to the overhead. To resolve this, we can take advantage of the observation that the backward pass incrementally computes gradients for each layer (starting from the loss and moving toward the input)—thus, we can all-reduce parameter gradients as soon as they're ready, reducing the overhead of data parallel training by overlapping computation of the backward pass with communication of gradients.

We'll start by implementing and benchmarking a distributed data parallel wrapper that asynchronously all-reduces individual parameter tensors as they become ready during the backward pass. The following pointers may be useful:

Backward hooks To automatically call a function on a parameter after its gradient has been accumulated in the backward pass, you can use the `register_post_accumulate_grad_hook` function.[4]

[4] See pytorch.org/docs/stable/generated/torch.Tensor.register_post_accumulate_grad_hook.html for more information and usage examples.

Basically identical: Avg time per training step with naive DDP: 5.118098e-01 s
Time spent transfering gradients: 1.467547e-01 s . This equals 28.67 % of total time Avg time per training step on single process: 6.725992e-01 s All weights match!)

Asynchronous communication all PyTorch collective communication operations support synchronous (`async_op=False`) and asynchronous execution (`async_op=True`). Synchronous calls will block until the collective operation is queued on the GPU. This does not mean that the CUDA operation is completed since CUDA operations are asynchronous. That being said, later function calls using the output will behave as expected.[5] In contrast, asynchronous calls will return a distributed request handle—as a result, when the function returns, the collective communication operation is not guaranteed to have been queued on GPU, let alone completed. To wait for the operation to be queued on GPU (and therefore for the output to be usable in later operations), you can call `handle.wait()` on the returned communication handle.

For example, the following two examples all-reduce each tensor in a list of tensors with either a synchronous or an asynchronous call:

```
tensors = [torch.rand(5) for _ in range(10)]

# Synchronous, block until operation is queued on GPU.
for tensor in tensor:
    dist.all_reduce(tensor, async_op=False)

# Asynchronous, return immediately after each call and
# wait on results at the end.
handles = []
for tensor in tensors:
    handle = dist.all_reduce(tensor, async_op=True)
    handles.append(handle)

# ...
# Possibly execute other commands that don't rely on the all_reduce results
# ...

# Ensure that all-reduce calls were queued and
# therefore other operations depending on the
# all-reduce output can be queued.
for handle in handles:
    handle.wait()
handles.clear()
```

Problem (`ddp_overlap_individual_parameters`): 5 points

Implement a Python class to handle distributed data parallel training. The class should wrap an arbitrary PyTorch `nn.Module` and take care of broadcasting the weights before training (so all ranks have the same initial parameters) and issuing communication calls for gradient averaging. We recommend the following public interface:

`def __init__(self, module: torch.nn.Module):` Given an instantiated PyTorch `nn.Module` to be parallelized, construct a DDP container that will handle gradient synchronization across ranks.

`def forward(self, *inputs, **kwargs):` Calls the wrapped module's `forward()` method with the provided positional and keyword arguments.

`def finish_gradient_synchronization(self):` When called, wait for asynchronous communication

[5] In advanced cases, if you are using multiple CUDA streams, you may need explicit synchronization across streams to ensure that the output is ready for later operations. See pytorch.org/docs/stable/notes/cuda.html#cuda-streams.

calls to be queued on GPU.

To use this class to perform distributed training, we'll pass it a module to wrap, and then add a call to `finish_gradient_synchronization()` before we run `optimizer.step()` to ensure that the optimizer step, an operation that depends on the gradients, may be queued:

```
model = ToyModel().to(device)
ddp_model = DDP(model)

for _ in range(train_steps):
    x, y = get_batch()
    logits = ddp_model(x)
    loss = loss_fn(logits, y)
    loss.backward()
    ddp_model.finish_gradient_synchronization()
    optimizer.step()
```

Done

Deliverable: Implement a container class to handle distributed data parallel training. This class should overlap gradient communication and the computation of the backward pass. To test your DDP class, first implement the adapters `[adapters.get_ddp_individual_parameters]` and `[adapters.ddp_individual_parameters_on_after_backward]` (the latter is optional, depending on your implementation you may not need it).

Then, to execute the tests, run `uv run pytest tests/test_ddp_individual_parameters.py`. We recommend running the tests multiple times (e.g., 5) to ensure that it passes reliably.

Problem (`ddp_overlap_individual_parameters_benchmarking`): 1 point

(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.

2.3.3 Overlapping Computation with Communication of Bucketed Parameter Gradients

In the previous section (§2.3.2), we overlapped backprop computation with the communication of individual parameter gradients. However, we previously observed that batching communication calls can improve performance, especially when we have many parameter tensors (as is typical in deep Transformer models). Our previous attempt at batching sent all of the gradients at once, which requires waiting for the backward

pass to finish. In this section, we'll try to get the best of both worlds by organizing our parameters into buckets (reducing the number of total communication calls) and all-reducing buckets when each of their constituent tensors are ready (enabling us to overlap communication with computation).

Problem (`ddp_overlap_bucketed`): 8 points

Implement a Python class to handle distributed data parallel training, using gradient bucketing to improve communication efficiency. The class should wrap an arbitrary input PyTorch `nn.Module` and take care of broadcasting the weights before training (so all ranks have the same initial parameters) and issuing bucketed communication calls for gradient averaging. We recommend the following interface:

`def __init__(self, module: torch.nn.Module, bucket_size_mb: float):` Given an instantiated PyTorch `nn.Module` to be parallelized, construct a DDP container that will handle gradient synchronization across ranks. Gradient synchronization should be bucketed, with each bucket holding at most `bucket_size_mb` of parameters.

`def forward(self, *inputs, **kwargs):` Calls the wrapped module's `forward()` method with the provided positional and keyword arguments.

`def finish_gradient_synchronization(self):` When called, wait for asynchronous communication calls to be queued on GPU.

Beyond the addition of a `bucket_size_mb` initialization parameter, this public interface matches the interface of our previous DDP implementation that individually communicated each parameter. We suggest allocating parameters to buckets using the reverse order of `model.parameters()`, since the gradients will become ready in approximately that order during the backward pass.

Deliverable: Implement a container class to handle distributed data parallel training. This class should overlap gradient communication and the computation of the backward pass. Gradient communication should be bucketed, to reduce the total number of communication calls. To test your implementation, complete `[adapters.get_ddp_bucketed]`, `[adapters.ddp_bucketed_on_after_backward]`, and `[adapters.ddp_bucketed_on_train_batch_start]` (the latter two are optional, depending on your implementation you may not need them).

Then, to execute the tests, run `pytest tests/test_ddp.py`. We recommend running the tests multiple times (e.g., 5) to ensure that it passes reliably.

Problem (`ddp_bucketed_benchmarking`): 3 points

(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 (n_b). 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.

2.4 4D Parallelism

While much more complex in implementation, there are more axes along which we can parallelize our training process. Most commonly we discuss 5 methods of parallelism:

- **Data parallelism (DP)** — Batches of data are split across multiple devices, and each device computes gradients for their own batch. These gradients must somehow be averaged across devices.
- **Fully-Sharded Data Parallelism (FSDP)** — Optimizer states, gradients, and weights are split across devices. If we're using only DP and FSDP, every device needs to gather the weight shards from all other devices before we can perform our forward or backward pass.
- **Tensor Parallelism (TP)** — Activations are sharded across a new dimension, and each device computes the output results for their own shard. With Tensor Parallel we can either shard along the inputs or the outputs the operation we are sharding. Tensor Parallelism can be used effectively together with FSDP if we shard the weights and the activations along corresponding dimensions.
- **Pipeline Parallelism (PP)** — The model is split layerwise into multiple stages, where each stage is run on a different device.
- **Expert Parallelism (EP)** — We separate experts (in Mixture-of-Experts models) onto different devices, and each device computes the output results for their own expert.

Typically, we always combine FSDP and TP, so we can think of them as a single axis of parallelism. This leaves us with 4 axes of parallelism: DP, FSDP/TP, PP, and EP. We will also focus on dense models (not MoEs) and so will not discuss EP further.

When reasoning about distributed training, we often describe our cluster as a *mesh* of devices, where the axes of the mesh are the axes along which we define our parallelism. For instance, if we have 16 GPUs and a model that is much larger than we can fit on a single device, we might be inclined to organize our mesh into a 4×4 grid of GPUs, where the first dimension represents DP, and the second dimension represents combined FSDP and TP.

See the overview in Part 5 of the TPU Scaling Book (Austin et al. [2025]) for more details on how these methods work and how to derive their communication and memory costs (this will be especially helpful to tackle the problem below). For a more detailed pipeline parallel discussion, see The Ultra-Scale Playbook Appendix (Nouamane Tazi [2025]). The rest of this book also has a lot of other information you might find useful.

Problem (`communication_accounting`): 10 points

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?

Deliverable: Your calculations and a one-sentence response.

(b) Now assume your master weights, optimizer state, gradients and half of your activations (in practice every second layer) are sharded across N_{FSDP} devices. Write an expression for how much memory this would take per device. What value does N_{FSDP} 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 $W_{\text{ici}} = 2 \cdot 9 \cdot 10^{10}$ 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?

Deliverable: Your calculations and a one-sentence response.

(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?

Deliverable: A one-paragraph response. Back up your claims with references and/or equations.

3 Optimizer State Sharding

Distributed data parallel training is conceptually simple and often very effective, but requires each rank to hold a distinct copy of the model parameters and optimizer state. This redundancy can come with significant memory costs. For example, the AdamW optimizer maintains two floats per parameter, meaning that it consumes twice as much memory as the model weights. Rajbhandari et al. [2020] describe several methods for reducing this redundancy in data-parallel training by partitioning the (1) optimizer state, (2) gradients, and (3) parameters across ranks, communicating them between workers as necessary.

In this part of the assignment, we'll reduce per-rank memory consumption by implementing a simplified version of optimizer state sharding. Rather than keeping the optimizer states for all parameters, each rank's optimizer instance will only handle a subset of the parameters (approximately 1 / `world_size`). When each rank's optimizer takes an optimizer step, it'll only update the subset of model parameters in its shard. Then, each rank will broadcast its updated parameters to the other ranks to ensure that the model parameters remain synchronized after each optimizer step.

Problem (`optimizer_state_sharding`): 15 points

Implement a Python class to handle optimizer state sharding. The class should wrap an arbitrary input PyTorch `optim.Optimizer` and take care of synchronizing updated parameters after each optimizer step. We recommend the following public interface:

`def __init__(self, params, optimizer_cls: Type[Optimizer], **kwargs: Any):` Initializes the sharded state optimizer. `params` is a collection of parameters to be optimized (or parameter groups, in case the user wants to use different hyperparameters, such as learning rates, for different parts of the model); these parameters will be sharded across all the ranks. The `optimizer_cls` parameter specifies the type of optimizer to be wrapped (e.g., `optim.AdamW`). Finally, any remaining keyword arguments are forwarded to the constructor of the `optimizer_cls`. Make sure to call the `torch.optim.Optimizer` super-class constructor in this method.

`def step(self, closure, **kwargs):` Calls the wrapped optimizer's `step()` method with the provided closure and keyword arguments. After updating the parameters, synchronize with the other ranks.

`def add_param_group(self, param_group: dict[str, Any]):` This method should add a parameter group to the sharded optimizer. This is called during construction of the sharded optimizer by the super-class constructor and may also be called during training (e.g., for gradually unfreezing layers in a model). As a result, this method should handle assigning the model's parameters among the ranks.

Deliverable: Implement a container class to handle optimizer state sharding. To test your sharded optimizer, first implement the adapter `[adapters.get_sharded_optimizer]`. Then, to execute the tests, run `uv run pytest tests/test_sharded_optimizer.py`. We recommend running the tests multiple times (e.g., 5) to ensure that it passes reliably.

Now that we've implemented optimizer state sharding, let's analyze its effect on the peak memory usage during training and its runtime overhead.

Problem (`optimizer_state_sharding_accounting`): 5 points

(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 ZeRO-DP P_{os} in Rajbhandari et al., 2020)?

Deliverable: 2-3 sentence summary of any differences, especially those related to memory and communication volume.

4 Epilogue

Congratulations on finishing the assignment! We hope that you found it interesting and rewarding, and that you learned a bit about how to accelerate language model training by improving single-GPU speed and/or leveraging multiple GPUs.

References

Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning, 2023. URL https://arxiv.org/abs/2307.08691.

Tri Dao, Daniel Y Fu, Stefano Ermon, Atri Rudra, and Christopher Re. Flashattention: Fast and memory-efficient exact attention with IO-awareness. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=H4DqfPSibmx.

Maxim Milakov and Natalia Gimelshein. Online normalizer calculation for softmax, 2018. URL https://arxiv.org/abs/1805.02867.

Horace He. Making deep learning go brrrr from first principles. 2022. URL https://horace.io/brrr_intro.html.

Jacob Austin, Sholto Douglas, Roy Frostig, Anselm Levskaya, Charlie Chen, Sharad Vikram, Federico Lebron, Peter Choy, Vinay Ramasesh, Albert Webson, and Reiner Pope. How to scale your model. 2025. Retrieved from https://jax-ml.github.io/scaling-book/.

Haojun Zhao Phuc Nguyen Mohamed Mekkouri Leandro Werra Thomas Wolf Nouamane Tazi, Ferdinand Mom. The ultra-scale playbook: Training llms on gpu clusters, 2025.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory optimizations toward training trillion parameter models, 2020. arXiv:1910.02054.