

CS336 Assignment 2 (systems): Systems and Parallelism

Version 26.1.3

CS336 Staff

Spring 2026

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. Activation checkpointing
3. Flash Attention 2 Triton kernel
4. Distributed data parallel training
5. Optimizer state sharding
6. Fully sharded data parallel training

What the code looks like

The assignment code and this write-up are available on GitHub at:

github.com/stanford-cs336/assignment2-systems

Please clone the repository using Git. 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 `cs336-basics/pyproject.toml` and the `cs336-basics/cs336_basics` package 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 directory contains all the tests 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.

2 Profiling and Benchmarking

In the first part of the assignment, we will explore 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 is possible with regular PyTorch. In the subsequent parts of the assignment, we will leverage multiple GPUs and understand how to train a model across a cluster.

2.1 Profiling

Before implementing any optimizations, 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:

1. Simple end-to-end benchmarking using the Python standard library to time our forward and backward passes
2. Compute profiling with the NVIDIA Nsight Systems tool to understand how that time is distributed across operations on both the CPU and GPU
3. Memory profiling

2.1.1 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 a reference 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 3.13.13
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 78 packages in 168ms
Python 3.13.13 (main, Apr  7 2026, 20:49:46) [Clang 22.1.1 ] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 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`).

2.1.2 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 except in the leaderboard, 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 and plots. We strongly recommend that you automate constructing tables for

your writeup in code, since formatting tables in LaTeX or Typst can be very tedious. See `pandas.DataFrame.to_latex()` and `pandas.DataFrame.to_typst()` 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	2560	10240	32	32
10B	4608	12288	50	36

Table 1: Specifications of different model sizes. These are mostly based on GPT-2 configs.

Use context length 512 unless otherwise specified.

2.1.3 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.

To start off, let's do the simplest possible profiling of our model by timing the forward pass, backward pass, and optimizer step. Since we will only be measuring speed and memory, it's fine to 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 PyTorch function call returns control to your code without waiting for the matrix multiplication to finish. In this way, the CPU can continue running ahead and scheduling new operations while the GPU finishes the matrix multiplication, which is a major performance win. On the other hand, this means that naively 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 scheduled GPU kernels to complete, allowing us to get more accurate measurements of CUDA kernel runtime. The synchronization in this operation refers to synchronizing the CPU runtime with the GPU runtime. With this in mind, let's write our basic profiling infrastructure.

Problem (benchmarking_script): Benchmarking Script (4 points)

- (a) Write a script to perform basic end-to-end benchmarking of the forward pass, backward pass, and optimizer step in your model. Specifically, your script should support the following:
- Given hyperparameters (e.g., number of layers), initialize a model.
 - Generate a random batch of data.
 - Run w warm-up steps (before you start measuring time), then time the execution of n steps (either only forward, forward and backward, or forward and backward with optimizer step, 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-only, forward-and-backward, and full training steps that include the optimizer step.

- (b) Time the forward, backward, and optimizer step for the model sizes described in [Section 2.1.2](#). 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?

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.

2.1.4 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`. We recommend that you get an up-to-date version either from your package manager, or using the installers from [their download page](#). In this part of the assignment, you will use `nsys` to analyze the runtime of your Transformer model.

Using `nsys` is straightforward: run your Python script from the previous section with `nsys profile` prepended. For example, you can run a basic profile for the script `benchmark.py` with:

```
$ uv run nsys profile -- 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.

A more comprehensive profiling run may look like:

```
$ uv run nsys profile --trace=cuda,cudnn,cublas,osrt,nvtx --pytorch=functions-trace,autograd-shapes-nvtx --cudabacktrace=all --python-backtrace=cuda --gpu-metrics-devices=0 -- python benchmark.py
```

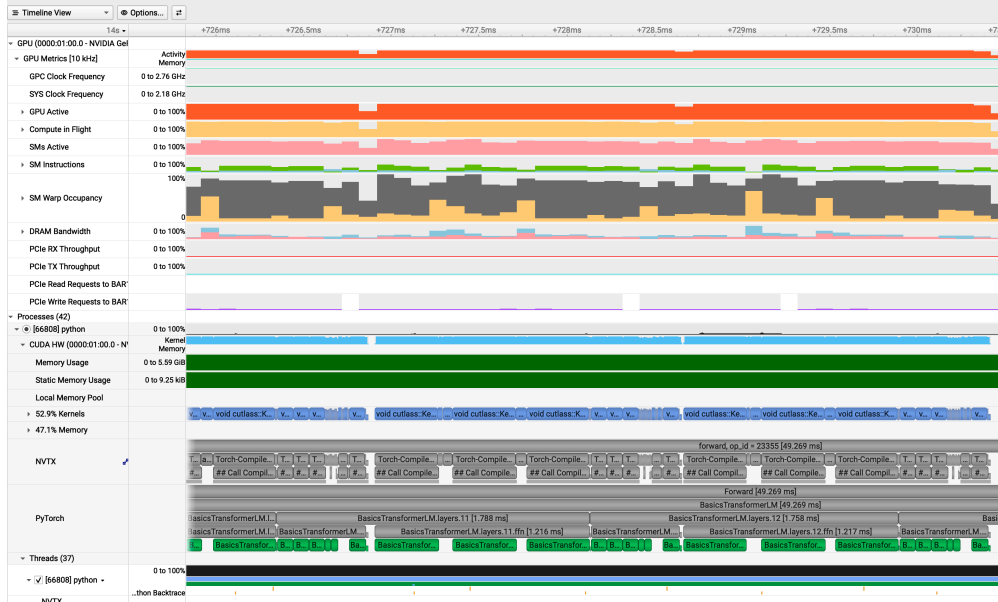


Figure 1: A detailed Nsight Systems trace

In this example, `--trace` specifies which APIs to log, `--pytorch` inserts nvtx labels during module calls and autograd, `--cudabacktrace` and `--python-backtrace` give better backtraces to understand where in your code a given kernel was invoked from, and `--gpu-metrics-devices` specifies which GPU's utilization to measure.

Adding profiling to a run is not free, and will overall slow down your runs. It's often worth only enabling features you're looking for in a given run. Specifically, you might want to remove `--cudabacktrace=all` and `--python-backtrace=cuda` when tracebacks aren't needed, since they have outsize overhead.

We encourage you to experiment with various [command-line options](#) for `nsys profile` to get a sense of what it can do. 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 an `--nvtx-capture` filter on the nvtx label 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:

```
...
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
```

```
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, it's worth noting that `torch.compile` can make it hard to attribute time and resources to specific parts of your code. You will likely have to wrap and strip various parts of your code in `torch.compile` and `nvtx` annotations to correctly attribute time and resource usage to various parts of your source.

Problem (nsys_profile): Nsight Systems Profiling (5 points)

Profile your forward pass, backward pass, and optimizer step using `nsys` with two model sizes from [Table 1](#) of your choice as well as three power-of-two context lengths larger than 128, where the largest available size should be the longest context length you can fit in memory. Pick the combinations you think would be the most interesting to look at. For each profile answer the following questions:

- (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 System 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.

2.1.5 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 B200 spec sheet says that its maximum throughput with FP32 is 80 TFLOPS, while its maximum throughput with FP16 (half-precision floats) or BF16 (bfloat16) is significantly higher at a whopping 2500 TFLOPS. As a result, using lower-precision datatypes should help us speed up training and inference.

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.bfloat16
x : torch.Tensor = ... # input data

with torch.autocast(device_type="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 downcast. The following exercise will help build your intuition as to why this is the case.

Problem (`mixed_precision_accumulation`): Mixed-Precision Accumulation (1 point)

Run the following code and comment 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)

```

Deliverable: A 2-3 sentence response.

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

Problem (benchmarking_mixed_precision): 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?
- the model's gradients?

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 [Section 2.1.2](#). Compare the results of using full precision versus mixed precision, and comment on any trends as model size changes. You may find the `nullcontext` no-op context manager to be useful.

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

2.1.6 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:

```
... # 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)
```

This will output a file `memory_snapshot.pickle` that you can load into the following online tool: 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): Memory Profiling (4 points)

Profile your complete training step of forward pass, backward pass, and optimizer step of the `x1` model from [Table 1](#) with context lengths of 128 and 2048.

- (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 `x1` model when either doing inference only (just forward pass) or a full training step. What 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 an `x1` 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 `x1` model when using mixed-precision, for both a forward pass and a full training step. Does mixed-precision significantly affect memory usage?

Deliverable: A 2-3 sentence response.

- (d) Consider the `x1` model. Given our reference hyperparameters, what is the size of a tensor of activations in the Transformer residual stream, in single-precision? Give this size in MiB (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 `x1` 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.

- (f) Nsight Systems also has flags for memory profiling. You can combine these with the Nsight flags from before to understand what allocations are happening at different steps in your model’s lifespan. Use the PyTorch-provided NVTX labels to determine how much memory is saved for backward (these tensors are often called residuals) by a single `TransformerBlock` in your model. Note the 5 largest contributing operations, and what percentage of the overall memory they contribute.

During the backward pass, all these tensors will be freed, but new gradient tensors are emitted at the same time. Based on your profiles showing how much memory was allocated during the forward pass, and how much memory usage changes for every `TransformerBlock` in the backward pass, calculate how much memory the produced gradient tensors for a `TransformerBlock` take. Does the result match what you expect?

Deliverable: Screenshots from Nsight Systems and a 1-2 paragraph response.

3 Single-GPU Memory

The later parts of this assignment will explore tricks to shard your tensors across multiple GPUs, but there are also tricks that can be applied even to single-GPU training. The most common of these is gradient checkpointing (also known as activation checkpointing).

3.1 Autograd Residuals

Recall that in order to perform a backward pass through your model, we need to save the activations that were produced in the forward pass. While this is obviously the case for some operations, by default it’ll happen for many more than you might expect. The tensors saved for the backward pass are called “residuals”, or simply “saved tensors”.

Let’s build some understanding of what’s being saved in our network. Starting with our unassuming `RMSNorm` function (pure FP32 for simplicity), let’s add some hooks for when tensors are being saved or retrieved by autograd.

```

import torch
from torch import nn

x = torch.randn((4, 512, 2560), requires_grad=True)

class RMSNorm(nn.Module):
    def __init__(
        self,
        hidden_size: int,
        eps: float = 1e-5,
        device=None,
    ):
        super().__init__()
        self.weight = nn.Parameter(torch.ones(hidden_size, device=device))
        self.eps = eps

    def forward(self, x):
        rms = torch.rsqrt(x.pow(2).mean(-1, keepdim=True) + self.eps)
        x = x * rms
        return self.weight * x

def pack_hook(t):
    shape, dtype, grad_fn = t.shape, t.dtype, t.grad_fn
    print(f"Saving residual: {shape=}, {dtype=}, {grad_fn=}")
    return t

def unpack_hook(t):
    shape, dtype, grad_fn = t.shape, t.dtype, t.grad_fn
    print(f>Loading residual: {shape=}, {dtype=}, {grad_fn=}")
    return t

ln = RMSNorm(x.shape[-1])

with torch.autograd.graph.saved_tensors_hooks(pack_hook, unpack_hook):
    y = ln(x)
    y.sum().backward()

```

The output shows a worrying amount of tensors being written out, several of them at full activation size!

```

$ uv run scripts/autograd_experiment.py
Saving residual: shape=torch.Size([4, 512, 2560]), dtype=torch.float32, grad_fn=None
Saving residual: shape=torch.Size([4, 512, 1]), dtype=torch.float32, grad_fn=<RsqrtBackward0
object at 0x7f7dd319b5e0>
Saving residual: shape=torch.Size([4, 512, 1]), dtype=torch.float32, grad_fn=<RsqrtBackward0
object at 0x7f7dd319b5e0>
Saving residual: shape=torch.Size([4, 512, 2560]), dtype=torch.float32, grad_fn=None
Saving residual: shape=torch.Size([4, 512, 2560]), dtype=torch.float32, grad_fn=<MulBackward0
object at 0x7f7dd319b5e0>
Saving residual: shape=torch.Size([2560]), dtype=torch.float32, grad_fn=None

Loading residual: shape=torch.Size([4, 512, 2560]), dtype=torch.float32,
grad_fn=<MulBackward0 object at 0x7f7cf14e6740>

```

```

Loading residual: shape=torch.Size([2560]), dtype=torch.float32, grad_fn=None
Loading residual: shape=torch.Size([4, 512, 1]), dtype=torch.float32, grad_fn=<RsqrtBackward0
object at 0x7f7cf14e6740>
Loading residual: shape=torch.Size([4, 512, 2560]), dtype=torch.float32, grad_fn=None
Loading residual: shape=torch.Size([4, 512, 1]), dtype=torch.float32, grad_fn=<RsqrtBackward0
object at 0x7f7cf14e6740>
Loading residual: shape=torch.Size([4, 512, 2560]), dtype=torch.float32, grad_fn=None

```

3.1.1 Operator Fusion

In this case, it's clear that the granularity of the operations used is too high. We want a single op that takes in the RMSNorm weights and the activation, and spits out the output, as well as for that operation to be unitary in the backward pass. This is one motivation for kernel fusion. Since the RMSNorm is fairly well behaved, we can even automatically fuse it using `torch.compile`.

```

...
ln = torch.compile(RMSNorm(x.shape[-1]))

with torch.autograd.graph.saved_tensors_hooks(pack_hook, unpack_hook):
    y = ln(x)
    y.sum().backward()

```

The new output is significantly better:

```

Saving residual: shape=torch.Size([4, 512, 2560]), dtype=torch.float32, grad_fn=None
Saving residual: shape=torch.Size([2560]), dtype=torch.float32, grad_fn=None
Saving residual: shape=torch.Size([4, 512, 1]), dtype=torch.float32, grad_fn=None
Loading residual: shape=torch.Size([4, 512, 2560]), dtype=torch.float32, grad_fn=None
Loading residual: shape=torch.Size([2560]), dtype=torch.float32, grad_fn=None
Loading residual: shape=torch.Size([4, 512, 1]), dtype=torch.float32, grad_fn=None

```

We only need to save a single full-size activation tensor for the backward pass — namely, the input to the RMSNorm function. Notice also how the order of loading is no longer the reverse of saving, and each residual no longer has a `grad_fn` dependency — PyTorch is treating the entirety of our RMSNorm as a single function.

3.2 Activation Checkpointing

While fusion is undoubtedly useful, it can only get us so far in saving memory. For instance, let's fuse a single TransformerBlock at size xl.

```

import torch
from cs336_basics.model import RotaryEmbedding, TransformerBlock

# num_layers for this model is 32
d_model, d_ff, num_heads, context_length = 2560, 10240, 16, 2048
block = TransformerBlock(d_model=d_model, d_ff=d_ff, num_heads=num_heads,
    positional_encoder=RotaryEmbedding(dim=d_model // num_heads, context_length=context_length))

# Fuse as much torch.compile will allow
block = torch.compile(block, fullgraph=True)
x = torch.randn((4, context_length, d_model), requires_grad=True)

```

```

...
# Now logs the number of bytes saved
total_size_bytes = 0
def pack_hook(t):
    if isinstance(t, torch.nn.Parameter): # Skip logging parameters to avoid double counting
        return t
    global total_size_bytes
    shape, dtype, grad_fn = t.shape, t.dtype, t.grad_fn
    total_size_bytes += t.numel() * t.element_size()
    print(f"Saving residual: {shape=}, {dtype=}, {grad_fn=}")
    return t

...

# Run forward pass, saving for backward
with torch.autograd.graph.saved_tensors_hooks(pack_hook, unpack_hook):
    y = block(x)

print(f"Total size of saved tensors in single TransformerBlock: {total_size_bytes /
(1024**2):.2f} MiB")

```

The script shows us how much memory we're saving for backward:

```

...
Total size of saved tensors in single TransformerBlock: 3651.31 MiB

```

3.6 GiB for every layer. If we do this for all layers, we get 114 GiB of activations, just saved for backward! There's a nontrivial amount of waste in the attention operation's residuals, which we will fix in [Section 4](#), but even with this fix, the memory use will grow linearly with batch size, sequence length and embedding size.

3.2.1 Recomputation

Instead of holding on to every tensor we generate, it's possible to save only periodic checkpoints of our results, and recompute the values in-between. PyTorch has an interface we can call to handle this in a simple fashion. `torch.utils.checkpoint.checkpoint` takes in a function, and arguments to that function. It then modifies the behavior of the function passed by:

1. In the forward pass:
 1. Saving the input values to the function
 2. Suppressing the saving of tensors in the forward pass
2. In the backward pass:
 1. Prepending a recomputation step where the forward pass is recomputed from the previously saved inputs, and values are saved for backward
 2. The backward pass is run and all tensors can be freed

In the simple case of running through 4 transformer blocks, we see that our memory adds up as we would expect.

```

...
def four_blocks(x):
    x = block(x)

```

```

    x = block(x)
    x = block(x)
    x = block(x)
    return x

with torch.autograd.graph.saved_tensors_hooks(pack_hook, unpack_hook):
    y = four_blocks(x)

print(f"Total size of saved tensors in four TransformerBlocks: {total_size_bytes /
(1024**2):.2f} MiB")

```

```
Total size of saved tensors in four TransformerBlocks: 14605.25 MiB
```

But we can employ gradient checkpointing as follows:

```

from torch.utils.checkpoint import checkpoint
def two_blocks(x):
    x = block(x)
    x = block(x)
    return x

def four_blocks_checkpoint(x):
    # checkpoint throws out all the saved tensors until the backward pass
    # when getting to the checkpointed block in the backward pass,
    # it reruns a forward pass to produce the saved tensors,
    # then completes normal backward pass.
    x = checkpoint(two_blocks, x, use_reentrant=False)
    x = checkpoint(two_blocks, x, use_reentrant=False)
    return x

with torch.autograd.graph.saved_tensors_hooks(pack_hook, unpack_hook):
    y = four_blocks_checkpoint(x)

print(f"Total size of saved tensors in four TransformerBlocks with checkpointing:
{total_size_bytes / (1024**2):.2f} MiB")

```

```

Saving residual: shape=torch.Size([0]), dtype=torch.float32, grad_fn=None
Saving residual: shape=torch.Size([4, 2048, 2560]), dtype=torch.float32, grad_fn=None
Saving residual: shape=torch.Size([0]), dtype=torch.float32, grad_fn=None
Saving residual: shape=torch.Size([4, 2048, 2560]), dtype=torch.float32,
grad_fn=<torch.autograd.function.CompiledFunctionBackward object at 0x7aa0657a19d0>
Total size of saved tensors in four TransformerBlocks with checkpointing: 160.00 MiB

```

Keep in mind this hasn't eliminated the memory use. Rather, it's factored our memory use into two categories: the longer term storage we save to prepare for recomputation at the entry point of each checkpoint call (the checkpoint itself), and the short term memory generated in the recomputation pass within the checkpointed block to facilitate a backward pass through it. Since our main consideration is the peak memory usage, we want to balance the memory cost of the saved checkpoints with the memory cost of materializing a full block worth of residuals. The more checkpoints we use, the smaller the amount of materialized memory within a single block, but the more memory we need for the checkpoints themselves.

In our example above, it's clear that the recomputed residuals' memory is dominating (in part because our checkpoints are tiny), so it would be beneficial to shift the balance more toward checkpointed memory. Thus we would want to decrease the scope of a checkpointed block.

Having larger or smaller checkpointed blocks doesn't affect the computational cost of recomputation, but we can continue to reduce the required memory at the cost of compute by using *recursive checkpointing*, meaning we nest `checkpoint` calls inside other `checkpoint` calls.

Problem (gradient_checkpointing): Memory-Optimal Gradient Checkpointing (4 points)

Consider a Transformer with N identical blocks stacked sequentially. Without any checkpointing, all N blocks' worth of residuals are kept alive simultaneously, giving $O(N)$ peak activation memory. We have a free hand to wrap any subset of the forward pass in `checkpoint`, including nesting `checkpoint` calls inside one another.

- (a) What checkpointing strategy minimizes peak activation memory, ignoring the compute cost? Describe how you would arrange the `checkpoint` calls (a code sketch is fine), and give the asymptotic peak activation memory and compute of your strategy as a function of N . Assume the residuals saved by a single block dominate any per-checkpoint bookkeeping.

Deliverable: A 3-5 sentence description of the strategy and its asymptotic peak memory, plus a short code sketch.

- (b) Consider the `x1` model config with batch size 4 and sequence length 2048 as above. If you only have the time/compute budget to run one step of recomputation (meaning you may not nest `checkpoint` calls), what is the best checkpointing strategy to reduce peak memory? Profile your run's peak memory to validate your hypothesis. Compare the peak memory of the next smaller and larger checkpointing block sizes to be sure.

Deliverable: A 3-5 sentence description of your reasoning along with the measured peak memory for your strategy.

4 GPU Kernels

4.1 Optimizing Attention with FlashAttention-2

4.1.1 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:

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

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): PyTorch Attention Benchmarking (2 points)

- (a) Benchmark your attention implementation at different scales. Write a script that will:
- (i) Fix the batch size to 8 and don't use multihead attention (i.e. remove the head dimension).
 - (ii) 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.
 - (iii) Create random inputs Q, K, V for the appropriate size.
 - (iv) Time 100 forward passes through attention using the inputs.
 - (v) Measure how much memory is in use before the backward pass starts, and time 100 backward passes.
 - (vi) Make sure to warm up, and to call `torch.cuda.synchronize()` after each forward/backward pass.

Depending on your GPU, some of these configurations are expected to run out of memory. 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 calculations for the memory usage, and a 1-2 paragraph response.

4.2 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 for using the PyTorch compiler is very simple. For instance, if we wanted to apply it to a single layer of our model, we can use:

```
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): 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.

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

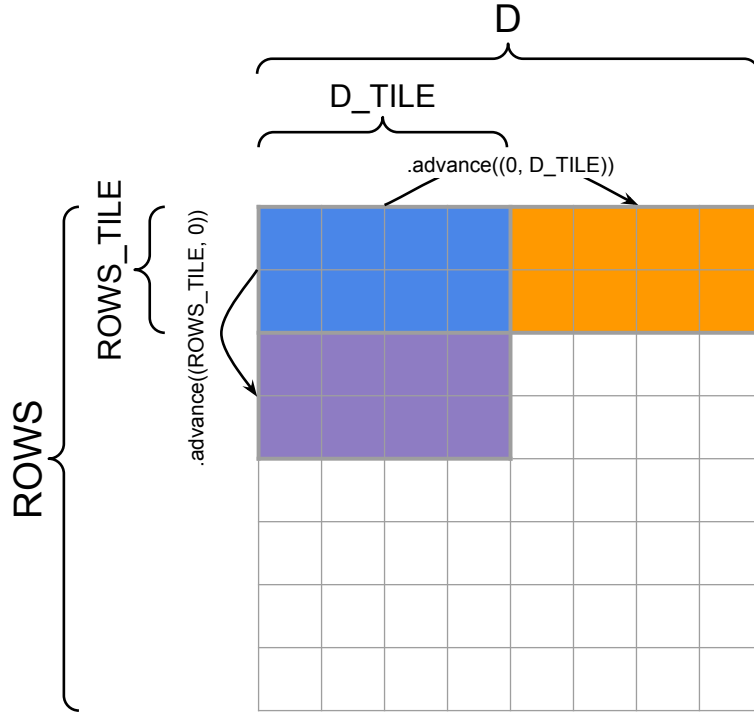


Figure 2: Tiling and advancing block pointers in the weighted sum kernel example (Section 4.2.1).

Refer to Figure 2 for a schematic of tiling and how block pointers are advanced. The weighted sum function from above looks like the following:

```
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
    NUM_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 load/store at a time
    # - The order of the dimensions in memory from major to minor
    #   axes (= np.argsort(strides)) for optimizations, needed for
    #   TMA support on >=Hopper
```

```

x_block_ptr = tl.make_block_ptr(
    x_ptr,
    shape=(NUM_ROWS, D,),
    strides=(x_stride_row, 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=(NUM_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 NUM_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 NUM_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[(triton.cdiv(n_rows, ctx.ROWS_TILE_SIZE),)](
    x, weight,
    y,
    x.stride(0), x.stride(1),
    weight.stride(0),
    y.stride(0),
    NUM_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[(triton.cdiv(n_rows, ctx.ROWS_TILE_SIZE),)]`, we define a so-called “launch grid” of thread blocks by passing the tuple `(triton.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 brevity, 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 the 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 :

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

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

This gives a simple formula for computing the backward pass. To obtain the backward step with respect to x , we apply [Equation 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 $\nabla_x \mathcal{L}$:

```

@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

    # 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 × 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((triton.cdiv(n_rows, ROWS_TILE_SIZE), D), device=x.device, dtype=x.dtype)
        grad_x = torch.empty_like(x)

        weighted_sum_backward[(triton.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.

4.2.2 FlashAttention-2 Forward Pass

You will replace your PyTorch attention implementation with a significantly improved Triton implementation following FlashAttention-2 [T. 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.

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

Before jumping into this section, we highly recommend reading at least the original FlashAttention paper [T. 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 [M. Milakov et al., 2018]). We also recommend checking out [H. He [4]] 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:

$$\mathbf{S} = \mathbf{Q}\mathbf{K}^\top / \sqrt{d} \quad (4)$$

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

$$\mathbf{O} = \mathbf{P}\mathbf{V} \quad (6)$$

The standard backward pass is

$$d\mathbf{V} = \mathbf{P}^\top d\mathbf{O} \quad (7)$$

$$d\mathbf{P} = d\mathbf{O}\mathbf{V}^\top \quad (8)$$

$$d\mathbf{S}_i = d\text{softmax}(\mathbf{dP}_i) = (\text{diag}(\mathbf{P}_i) - \mathbf{P}_i\mathbf{P}_i^\top) d\mathbf{P}_i \quad (9)$$

$$d\mathbf{Q} = d\mathbf{S}\mathbf{K} / \sqrt{d} \quad (10)$$

$$d\mathbf{K} = d\mathbf{S}^\top \mathbf{Q} / \sqrt{d} \quad (11)$$

As we can see, the backward pass depends on some very large activations from the forward pass. For example, computing $d\mathbf{V}$ in Equation 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 Equation 7 and Equation 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:

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

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 [T. Dao et al., 2022; T. 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 precompute the value $D = \text{rowsum}(\mathbf{O} \circ d\mathbf{O})$ in global memory (where $\circ$ is element-wise multiplication), which is equal to $\text{rowsum}(\mathbf{P} \circ d\mathbf{P})$ since $\mathbf{P}d\mathbf{P}^\top = \mathbf{P}(d\mathbf{O}\mathbf{V}^\top)^\top = (\mathbf{P}\mathbf{V})d\mathbf{O}^\top = \mathbf{O}d\mathbf{O}^\top$ (and $\text{rowsum}(\mathbf{A} \circ \mathbf{B}) = \text{diag}(\mathbf{A}\mathbf{B}^\top)$ 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:

$$\mathbf{S} = \mathbf{Q}\mathbf{K}^\top / \sqrt{d} \quad (13)$$

$$P_{ij} = \exp(S_{ij} - L_i) \quad (14)$$

$$d\mathbf{V} = \mathbf{P}^\top d\mathbf{O} \quad (15)$$

$$d\mathbf{P} = d\mathbf{O}\mathbf{V}^\top \quad (16)$$

$$dS_{ij} = P_{ij}(dP_{ij} - D_i) \quad (17)$$

$$d\mathbf{Q} = d\mathbf{S}\mathbf{K} / \sqrt{d} \quad (18)$$

$$d\mathbf{K} = d\mathbf{S}^\top \mathbf{Q} / \sqrt{d} \quad (19)$$

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 Equation 13 and Equation 14.

Details of the FlashAttention 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 $\mathbf{P}$, ideally tiled in both dimensions (for queries and for keys).

However, when we apply softmax to $\mathbf{S}$, we require entire rows of $\mathbf{S}$ to be reduced to compute the softmax denominator, meaning we cannot compute $\mathbf{P}$ 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 those along the key dimension will be of size 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 $\tilde{P}_i^{(j)} = \exp(S_{ij} - m_i^{(j)})$. $l_i^{(j)}$ is a running proxy for the softmax denominator, and will be updated using the unnormalized softmax values as $l_i^{(j)} = \exp(m_i^{(j-1)} - m_i^{(j)}) \cdot l_i^{(j-1)} + \text{rowsum}(\tilde{P}_i^{(j)})$. 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 on the GPU.

Algorithm 1: FlashAttention-2 forward pass

```

1 Require:  $Q \in \mathbb{R}^{N_q \times d}$ ,  $K, V \in \mathbb{R}^{N_k \times d}$ , tile sizes  $B_q, B_k$ 
2 Split  $Q$  into  $T_q = \lceil \frac{N_q}{B_q} \rceil$  tiles  $Q_1, \dots, Q_{T_q}$  of size  $B_q \times d$ 
3 Split  $K, V$  into  $T_k = \lceil \frac{N_k}{B_k} \rceil$  tiles  $K^{(1)}, \dots, K^{(T_k)}$  and  $V^{(1)}, \dots, V^{(T_k)}$  of size  $B_k \times d$ 
4 for  $i = 1, \dots, T_q$  do
5   Load  $Q_i$  from global memory
6   Initialize  $O_i^{(0)} = \mathbf{0} \in \mathbb{R}^{B_q \times d}$ ,  $l_i^{(0)} = 0 \in \mathbb{R}^{B_q}$ ,  $m_i^{(0)} = -\infty \in \mathbb{R}^{B_q}$ 
7   for  $j = 1, \dots, T_k$  do
8     Load  $K^{(j)}, V^{(j)}$  from global memory
9     Compute tile of pre-softmax attention scores  $S_i^{(j)} = \frac{Q_i (K^{(j)})^\top}{\sqrt{d}} \in \mathbb{R}^{B_q \times B_k}$ 
10    Compute  $m_i^{(j)} = \max(m_i^{(j-1)}, \text{rowmax}(S_i^{(j)})) \in \mathbb{R}^{B_q}$ 
11    Compute  $\tilde{P}_i^{(j)} = \exp(S_i^{(j)} - m_i^{(j)}) \in \mathbb{R}^{B_q \times B_k}$ 
12    Compute  $l_i^{(j)} = \exp(m_i^{(j-1)} - m_i^{(j)}) l_i^{(j-1)} + \text{rowsum}(\tilde{P}_i^{(j)}) \in \mathbb{R}^{B_q}$ 
13    Compute  $O_i^{(j)} = \text{diag}(\exp(m_i^{(j-1)} - m_i^{(j)})) O_i^{(j-1)} + \tilde{P}_i^{(j)} V^{(j)}$ 
14  end for
15  Compute  $O_i = \text{diag}(l_i^{(T_k)})^{-1} O_i^{(T_k)}$ 
16  Compute  $L_i = m_i^{(T_k)} + \log(l_i^{(T_k)})$ 
17  Write  $O_i$  to global memory as the  $i$ -th tile of  $O$ .
18  Write  $L_i$  to global memory as the  $i$ -th tile of  $L$ .
19 end for
20 Return the output  $O$  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): FlashAttention-2 Forward Pass (15 points)

- (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 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 Equation 6 and Equation 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 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 $(T_q, \text{batch_size})$, 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,
Q_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,
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 (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 $\tilde{P}_i^{(j)}$ to the `dtype` of $V^{(j)}$ before multiplying them, and cast 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.

- (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 $S_i^{(j)}$. 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 and defaults to `False` so the previous tests still pass.

Implementing the backward pass with recomputation

Notice that unlike the standard backward pass in Equation 7 to Equation 11, we can use recomputation to avoid the softmax operation in the backward pass shown in Equation 13 to Equation 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): FlashAttention-2 Backward Pass (5 points)

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 inputs, and return dQ , dK and dV . Remember to compute and use the D vector. You may follow along the computations of Equation 13 to Equation 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): FlashAttention-2 Benchmarking (5 points)

- (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 B200. 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.

4.2.3 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 dQ and again for dK and dV . This lets us skip synchronization across thread blocks, meaning we can avoid slow atomics.

Algorithm 2: Tiled FlashAttention-2 backward pass

```
1 Require:  $Q, O, dO \in \mathbb{R}^{N_q \times d}$ ,  $K, V \in \mathbb{R}^{N_k \times d}$ ,  $L \in \mathbb{R}^{N_q}$ , tile sizes  $B_q, B_k$ 
2 Compute  $D = \text{rowsum}(O \circ dO) \in \mathbb{R}^{N_q}$ 
3 Split  $Q, O, dO$  into  $T_q = \lceil \frac{N_q}{B_q} \rceil$  tiles  $Q_1, \dots, Q_{T_q}, O_1, \dots, O_{T_q}, dO_1, \dots, dO_{T_q}$ , each of size  $B_q \times d$ 
4 Split  $K, V$  into  $T_k = \lceil \frac{N_k}{B_k} \rceil$  tiles  $K^{(1)}, \dots, K^{(T_k)}$  and  $V^{(1)}, \dots, V^{(T_k)}$ , each of size  $B_k \times d$ 
5 Split  $L, D$  into  $T_q$  tiles  $L_1, \dots, L_{T_q}$  and  $D_1, \dots, D_{T_q}$ , each of size  $B_q$ 
6 for  $j = 1, \dots, T_k$  do
7   Load  $K^{(j)}, V^{(j)}$  from global memory
8   Initialize  $dK_0^{(j)} = dV_0^{(j)} = \mathbf{0} \in \mathbb{R}^{B_k \times d}$ 
9   for  $i = 1, \dots, T_q$  do
10    Load  $Q_i, dO_i$  from global memory
11    Compute tile of attention scores  $S_i^{(j)} = \frac{Q_i(K^{(j)})^\top}{\sqrt{d}} \in \mathbb{R}^{B_q \times B_k}$ 
12    Compute attention probabilities  $P_i^{(j)} = \exp(S_i^{(j)} - L_i) \in \mathbb{R}^{B_q \times B_k}$ 
13    Compute  $dV_i^{(j)} = dV_{i-1}^{(j)} + (P_i^{(j)})^\top dO_i \in \mathbb{R}^{B_k \times d}$ 
14    Compute  $dP_i^{(j)} = dO_i (V^{(j)})^\top \in \mathbb{R}^{B_q \times B_k}$ 
15    Compute  $dS_i^{(j)} = P_i^{(j)} \circ (dP_i^{(j)} - D_i) \in \mathbb{R}^{B_q \times B_k}$ 
16    Compute  $dK_i^{(j)} = dK_{i-1}^{(j)} + (dS_i^{(j)})^\top Q_i / \sqrt{d} \in \mathbb{R}^{B_k \times d}$ 
17  end for
18  Write  $dK_{T_q}^{(j)}$  and  $dV_{T_q}^{(j)}$  to global memory as the  $j$ -th tiles of  $dK$  and  $dV$ .
19 end for
20 for  $i = 1, \dots, T_q$  do
21   Load  $Q_i, dO_i$  from global memory
22   Initialize  $dQ_i^{(0)} = \mathbf{0} \in \mathbb{R}^{B_q \times d}$ 
23   for  $j = 1, \dots, T_k$  do
24    Load  $K^{(j)}, V^{(j)}$  from global memory
25    Compute tile of attention scores  $S_i^{(j)} = \frac{Q_i(K^{(j)})^\top}{\sqrt{d}} \in \mathbb{R}^{B_q \times B_k}$ 
26    Compute attention probabilities  $P_i^{(j)} = \exp(S_i^{(j)} - L_i) \in \mathbb{R}^{B_q \times B_k}$ 
27    Compute  $dP_i^{(j)} = dO_i (V^{(j)})^\top \in \mathbb{R}^{B_q \times B_k}$ 
28    Compute  $dS_i^{(j)} = P_i^{(j)} \circ (dP_i^{(j)} - D_i) \in \mathbb{R}^{B_q \times B_k}$ 
29    Compute  $dQ_i^{(j)} = dQ_i^{(j-1)} + dS_i^{(j)} K^{(j)} / \sqrt{d} \in \mathbb{R}^{B_q \times d}$ 
30  end for
31  Write  $dQ_i^{(T_k)}$  to global memory as the  $i$ -th tile of  $dQ$ .
32 end for
33 Return  $dQ, dK, dV$ .
```

5 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 naïve implementation of distributed data parallel training, then implement and benchmark various improvements to communication efficiency.

5.1 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.

```
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])
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

²If 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.

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 worker process belongs 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. You should always use NCCL for distributed GPU training, and only use Gloo for local development where you don't have a GPU available. 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 3](#) 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, 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.

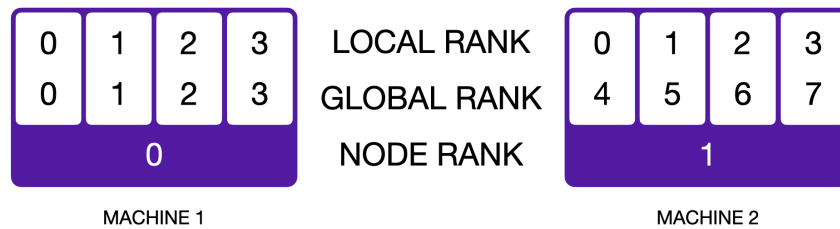


Figure 3: A schematic representation of a distributed application running on 2 nodes with a world size of 8. Each worker process is identified by a global rank (from 0 to 7) and a local rank (from 0 to 3). Figure taken from https://lightning.ai/docs/fabric/stable/advanced/distributed_communication.html

5.1.1 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 should be as simple as modifying your `init_process_group` call and tensor device casts.

Problem (`distributed_communication_single_node`): 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:

all-reduce data size float32 data tensors ranging over 1MB, 10MB, 100MB, 1GB.

Number of GPUs/processes 2, 4, or 6.

Resource requirements: Up to 6 GPUs. Each benchmarking run should take less than 5 minutes.

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

³See <https://github.com/pytorch/pytorch/issues/68112#issuecomment-965932386> for more details.

5.2 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). d should divide n , otherwise some ranks would do more work than others, and the step is bottlenecked by the slowest.
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): Naïve DDP (5 points)

Deliverable: Implement a naïve form of distributed data parallel training that all-reduces individual parameter gradients after the backward pass. To test your implementation, implement `[adapters.get_ddp]` and (optionally) `[adapters.ddp_on_after_backward]`, then run `uv run pytest tests/test_ddp.py`.

Problem (naive_ddp_benchmarking): Naïve DDP Benchmarking (3 points)

In this naïve DDP implementation, parameter gradients 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 [Section 2.1.2](#).

Deliverable: A description of your benchmarking setup, along with the measured time per training iteration and time spent communicating gradients for each setting.

5.3 Improving Upon the Minimal DDP Implementation

The minimal DDP implementation that we saw in [Section 5.2](#) 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.

5.3.1 Reducing the Number of Communication Calls

Rather than issuing a communication call for each parameter tensor, let's 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`): Minimal DDP with Flat Gradients 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 [Section 2.1.2](#)).

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.

5.3.2 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 the 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

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

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 the GPU, let alone completed. To wait for the operation to be queued on the 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 the GPU.
for tensor in tensors:
    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`): DDP with Overlapping 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.
```

⁵In 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 <https://pytorch.org/docs/stable/notes/cuda.html#cuda-streams>.

def finish_gradient_synchronization(self): When called, wait for asynchronous communication calls to finish on the 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, can be safely 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()
```

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]` and `[adapters.ddp_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.py`. We recommend running the tests multiple times (e.g., 5) to ensure that it passes reliably.

Problem (`ddp_overlap_individual_parameters_benchmarking`): DDP Overlapping 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 [Section 2.1.2](#).

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 the initial DDP implementation with this overlapped implementation. 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.

6 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. [S. Rajbhandari et al. \[5\]](#) 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 $1 / \text{world_size}$). 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): 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 they pass 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): 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 S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He [5])?

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

7 Fully-Sharded Data Parallel

With optimizer state sharding and data parallel, we're able to split the optimizer state and activations across our data-parallel axis. However, our model weights remain duplicated — we're storing a full copy of them all on each GPU.

We can solve this by turning our data parallel (DP) axis into a fully-sharded data parallel axis (FSDP). With FSDP, each GPU stores only its own slice of every weight tensor, but has to pull slices from other GPUs to form the full weight tensor using an all-gather to prepare for a forward or backward pass.

To avoid keeping GPU compute waiting around for communication to finish, most FSDP implementations schedule the layer's all-gather in advance of the operation, meaning the relevant weights are ready before they are needed, preventing communication from blocking computation. This keeps weight sharding communication off the critical path, meaning it has no cost as long as communication can keep up with compute and is scheduled well.

Some layers are small enough in memory and compute that the latency overhead of a transfer is not worth it. You should mark these layers not to be sharded by FSDP. In our architecture, this will mostly be the case for norms. This leaves us with the embedding layer and every linear layer.

While it is necessary to *store* master weights in FP32 (any values that are repeatedly accumulated into are sensitive to precision), the weights do not need to be *used* in FP32. In mixed precision, we always convert to the low-precision compute datatype before use, so we may as well convert even before the weight is communicated to save on bandwidth.

Problem (fsdp): Fully-Sharded Data Parallel (15 points)

Implement a Python class for fully-sharded data parallel training. The class should wrap an arbitrary PyTorch `nn.Module` (your full model) and hook into or wrap any `Linear` or `Embedding` layer within it. We recommend the following public interface:

```
def __init__(self, module: torch.nn.Module, compute_dtype: torch.dtype | None = None):
```

Given an instantiated PyTorch `nn.Module` to be parallelized, construct an FSDP module that will handle weight all-gathers and gradient reduce-scatters. Make sure that your hooks or your module wrappers all-gather the weights in time for the forward pass. To limit memory use, only start gathering after the layer two before the current one has completed its forward pass. In the backward pass, your hooks or module wrappers should all-gather to have the weights available for the computation. When the gradients are available, they should be reduce-scattered to the appropriate ranks. Make sure to free the gathered weights after use. When `compute_dtype` is provided, cast the weights to that dtype before communicating or using them for compute, while keeping master weights and optimizer updates in FP32.

```
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 finish on the GPU.

Deliverable: Implement a container class to handle fully sharded data parallel training. Each shard of this container should be compatible with the standard AdamW implementation from assignment 1. To test your FSDP implementation, implement the adapter `[adapters.get_fsdp]`. Run the tests with `uv run pytest tests/test_fsdp.py`. We recommend running the tests multiple times (e.g., 5) to catch any race conditions.

Problem (fsdp_accounting): FSDP Accounting (5 points)

- (a) Given your analysis in [Section 6](#), how much memory do you expect to save from the peak by implementing FSDP? You can ignore the size of the preallocated buffers needed to all-gather weights to each GPU in your calculation.

Deliverable: 2-3 sentence response with your findings.

- (b) Profile the xl model on two GPUs and pay attention to the all-gather of weights. Does the communication finish in time for the forward pass?

Deliverable: 2-3 sentence response with your timings. Include screenshots of Nsight to back up your claims.

8 Analyzing Parallelism Strategies

There are more axes along which we can parallelize our training process. Some common strategies include:

- **Data parallelism (DP)** — Batches of data are split across multiple devices, and each device computes gradients for its own batch. Gradients are then averaged across devices.
- **Fully-Sharded Data Parallelism (FSDP)** — On top of data parallelism, we also split optimizer states, gradients, and weights across devices to reduce memory usage. Devices then need to gather weight shards from other devices during the forward and backward passes.

- **Tensor Parallelism (TP)** — Weight matrices are sharded across the input or output dimension. Devices compute the activations corresponding to their shard, and activations are then reduced or gathered across devices.
- **Pipeline Parallelism (PP)** — The model is split layerwise into multiple stages, where each stage is run on a different device.
- **Expert Parallelism (EP)** — The experts in a Mixture-of-Experts model are split onto different devices, and each device computes the output for its own expert.

In this section, we'll do some basic math in a simplified setting to choose between parallelism strategies and decide how to combine them. To start, we'll focus on DP, FSDP, TP, and their combinations. Our approach will be to calculate the communication cost of each strategy and compare it to the computational cost, which tells us how many devices we can scale to before communication cost becomes the bottleneck.

For a more detailed treatment of TPU/GPU topologies and parallelism strategies, [the TPU Scaling Book](#) (J. Austin *et al.* [6]) is an excellent resource. And for a more detailed pipeline parallel discussion, see [The Ultra-Scale Playbook Appendix](#) (H. Z. P. N. M. M. L. W. T. W. Nouamane Tazi Ferdinand Mom [7]). The rest of these books also have a lot of other information you might find useful.

8.1 Communication Primitives

Our first step will be to understand the communication primitives. In our simplified setting, suppose we have N devices numbered $0, \dots, N-1$, and each pair of devices is connected by a link. We'll also assume each device has W egress (i.e. outgoing) bandwidth; in other words, each device can send data to another device at a rate of W bytes per second. How might we implement gather and reduce?

One common way to implement the all-gather operation is the **ring all-gather**. Recall that in an all-gather, each device i starts with a chunk x_i of size $\frac{S}{N}$, and ends up with the entire $x = [x_0, \dots, x_{N-1}]$ of size S (in bytes). In a ring all-gather, we arrange the devices in a circle. In each step, each device sends its current chunk to the next device to its right, and stores the chunk it received from the device to its left. This process repeats, where each device passes the chunk it just received to the right, and receives a new chunk from the left. After $N-1$ steps, each device has the entire tensor.

In our idealized setting, each device simultaneously transmits a chunk of size $\frac{S}{N}$ in each step, with egress bandwidth W , and there are $N-1$ steps, so the ring all-gather takes $\frac{N-1}{N} \frac{S}{W}$ seconds.

Next, let's analyze the **ring reduce-scatter**. In a reduce-scatter, each device i starts with a full tensor $x^{(i)}$ of size S . We then want to compute the reduction $y = \sum_{i=0}^{N-1} x^{(i)}$, but where each device i ends up with just a chunk y_i of size $\frac{S}{N}$. Like the ring all-gather, we'll start by arranging the devices in a circle. Each device will first divide its tensor $x^{(i)}$ into N chunks $[x_0^{(i)}, \dots, x_{N-1}^{(i)}]$, each of size $\frac{S}{N}$. We'll then pass chunks around just like the ring all-gather, except before passing the chunk on, each device adds its contribution to the chunk (which stores a partial sum). Specifically:

For step $t = 1, \dots, N-1$, device i does the following:

- If $t = 1$, initialize $y \leftarrow x^{(i)}$, which stores the partial sum so far
- Send chunk $y_{(i-t) \bmod N}$ to device $(i+1) \bmod N$
- Receive chunk $z_{(i-t-1) \bmod N}$ from device $(i-1) \bmod N$
- Update your copy of the partial sum: $y_{(i-t-1) \bmod N} \leftarrow y_{(i-t-1) \bmod N} + z_{(i-t-1) \bmod N}$

After $N-1$ steps, device i then has the full sum for chunk y_i , so ring reduce-scatter takes $\frac{N-1}{N} \frac{S}{W}$ seconds, just like ring all-gather.

Finally, let's implement **ring all-reduce**. In an all-reduce, each device i starts with a full tensor $x^{(i)}$ of size S , and ends up with the reduction $y = \sum_{i=0}^{N-1} x^{(i)}$. We'll implement all-reduce as a **ring reduce-scatter** followed by a **ring all-gather**, so the ring all-reduce takes $2 \frac{N-1}{N} \frac{S}{W}$ seconds.

Problem (alternate_ring_all_reduce): Alternate ring all-reduce (1 point)

Instead of implementing all-reduce as a ring reduce-scatter followed by a ring all-gather, let's use the following algorithm:

For step $t = 1, \dots, N - 1$, device i does the following:

- If $t = 1$, initialize $y \leftarrow x^{(i)}$, which stores the partial sum so far
- Send $x^{((i-t+1) \bmod N)}$ to device $(i + 1) \bmod N$
- Receive $x^{((i-t) \bmod N)}$ from device $(i - 1) \bmod N$
- Update your copy of the partial sum: $y \leftarrow y + x^{((i-t) \bmod N)}$

In the same setting as above (W egress bandwidth per device, each $x^{(i)}$ is of size S), how long does this algorithm take?

Deliverable: An answer in terms of S , N , and W , along with a one-sentence justification.

8.2 Analyzing Data Parallel

Given our communication primitives, we are ready to analyze parallelism strategies. We'll analyze the parallelization of a single FFN layer. Recall that given input $\mathbf{x}$, our forward pass is given by the following:

$$\mathbf{x}_1 = \mathbf{x} \mathbf{W}_1 \tag{20}$$

$$\mathbf{x}_2 = \mathbf{x} \mathbf{W}_2 \tag{21}$$

$$\mathbf{z} = f(\mathbf{x}_1) * \mathbf{x}_2 \tag{22}$$

$$\mathbf{y} = \mathbf{z} \mathbf{W}_3, \tag{23}$$

where $\mathbf{x}$ has shape (B, D) , $\mathbf{W}_1$ and $\mathbf{W}_2$ have shape (D, D_{FF}) , and $\mathbf{W}_3$ has shape (D_{FF}, D) . f is our elementwise activation function (e.g. SiLU), and $*$ represents elementwise multiplication.

It will also be useful to explicitly write out the backward pass. Recall that given $d\mathbf{y}$ with shape (B, D) , the backward pass is given by the following:

$$d\mathbf{z} = d\mathbf{y} \mathbf{W}_3^\top \tag{24}$$

$$d\mathbf{x}_2 = d\mathbf{z} * f(\mathbf{x}_1) \tag{25}$$

$$d\mathbf{x}_1 = d\mathbf{z} * f'(\mathbf{x}_1) * \mathbf{x}_2 \tag{26}$$

$$d\mathbf{x} = d\mathbf{x}_1 \mathbf{W}_1^\top + d\mathbf{x}_2 \mathbf{W}_2^\top \tag{27}$$

$$d\mathbf{W}_3 = \mathbf{z}^\top d\mathbf{y} \tag{28}$$

$$d\mathbf{W}_2 = \mathbf{x}^\top d\mathbf{x}_2 \tag{29}$$

$$d\mathbf{W}_1 = \mathbf{x}^\top d\mathbf{x}_1, \tag{30}$$

where $*$ represents elementwise multiplication.

Recall that in data parallelism with N_{DP} devices, we shard our input $\mathbf{x}$ into shards $\mathbf{x}^{(i)}$ of size $(\frac{B}{N_{\text{DP}}}, D)$. The forward pass proceeds as usual without any collectives, producing activations $\mathbf{y}^{(i)}$ of size $(\frac{B}{N_{\text{DP}}}, D)$. In

the backward pass, proceeding as usual with the batch-sharded activations, device i ends up with gradients

$$d\mathbf{W}_3^{(i)} = \mathbf{z}^{(i)\top} d\mathbf{y}^{(i)} \quad (31)$$

$$d\mathbf{W}_2^{(i)} = \mathbf{x}^{(i)\top} d\mathbf{x}_2^{(i)} \quad (32)$$

$$d\mathbf{W}_1^{(i)} = \mathbf{x}^{(i)\top} d\mathbf{x}_1^{(i)}, \quad (33)$$

where instead of summing over all B outer products, we only have the partial sum over the $\frac{B}{N_{\text{DP}}}$ outer products for our shard of the input. Then, we need to do an all-reduce across devices to get the full gradients $d\mathbf{W}_3$, $d\mathbf{W}_2$, and $d\mathbf{W}_1$.

Problem (data_parallel_calcs): Data parallel calculations (3 points)

We now have everything we need to calculate when data parallelism becomes communication bottlenecked. Let C (in FLOP/s) denote the device accelerator speed, and W (in bytes per second) denote each device's egress bandwidth. We can then compute the computation time and communication time. Because computation and communication can be overlapped, we are bottlenecked when communication time becomes larger than computation time. We'll assume that all weights and activations are in FP16 (i.e. two bytes).

- (a) How many FLOPs are required to compute the backward pass, with N_{DP} data parallelism? You can ignore all non-matmul operations. Recall that a matmul $(A, B)(B, C) \rightarrow (A, C)$ takes $2ABC$ flops.

Deliverable: An answer in terms of B , D , D_{FF} , and N_{DP} , along with a one-sentence justification.

- (b) How much communication time is required in the backward pass, with N_{DP} data parallelism?

Deliverable: An answer in terms of a subset of B , D , D_{FF} , N_{DP} , and W , along with a one-sentence justification.

- (c) Fixing the other parameters, how large can N_{DP} become before we're communication bottlenecked?

Deliverable: An inequality with N_{DP} on one side, and an expression in terms of a subset of B , D , D_{FF} , C , and W on the other, along with a one-sentence justification.

8.3 Analyzing Fully Sharded Data Parallel

Next, let's analyze FSDP. Recall that just like DP, FSDP shards the batch dimension of the inputs and activations. In addition, to save on memory, we also shard the optimizer states, gradients, and weights. We can shard the weights along either dimension, producing shards $\mathbf{W}_1^{(i)}$, $\mathbf{W}_2^{(i)}$, and $\mathbf{W}_3^{(i)}$ on device i , each of size $\frac{DD_{\text{FF}}}{N_{\text{FSDP}}}$.

In the forward pass, we will just do the data parallel forward pass on the batch-sharded input. But to do so, recall that we need to first all-gather the weight shards across devices:

$$\mathbf{W}_1 = \text{all-gather} \left(\left\{ \mathbf{W}_1^{(i)} \right\}_{i=0}^{N_{\text{FSDP}}-1} \right) \quad (34)$$

$$\mathbf{W}_2 = \text{all-gather} \left(\left\{ \mathbf{W}_2^{(i)} \right\}_{i=0}^{N_{\text{FSDP}}-1} \right) \quad (35)$$

$$\mathbf{W}_3 = \text{all-gather} \left(\left\{ \mathbf{W}_3^{(i)} \right\}_{i=0}^{N_{\text{FSDP}}-1} \right) \quad (36)$$

$$(\text{do batch-sharded forward pass}) \quad (37)$$

Note that we have some freedom in when we want to do the all-gather: we just need an all-gathered weight before it's used, and we should then discard it to keep memory costs low. For simplicity, in this section we just list the three all-gathers together.

Like the forward, in the backward pass we need to first all-gather the weight shards across devices. We can then do the data parallel backward pass, except we no longer need to do an all-reduce because each device only needs the gradient for its shard. Therefore, we do a reduce-scatter instead:

$$\mathbf{W}_1 = \text{all-gather} \left(\left\{ \mathbf{W}_1^{(i)} \right\}_{i=0}^{N_{\text{FSDP}}-1} \right) \quad (38)$$

$$\mathbf{W}_2 = \text{all-gather} \left(\left\{ \mathbf{W}_2^{(i)} \right\}_{i=0}^{N_{\text{FSDP}}-1} \right) \quad (39)$$

$$\mathbf{W}_3 = \text{all-gather} \left(\left\{ \mathbf{W}_3^{(i)} \right\}_{i=0}^{N_{\text{FSDP}}-1} \right) \quad (40)$$

$$(\text{do batch-sharded backward pass}) \quad (41)$$

$$d\mathbf{W}_1^{(i)} = \text{reduce-scatter} \left(\left\{ d\mathbf{W}_1^{(i)} \right\}_{i=0}^{N_{\text{FSDP}}-1} \right) \quad (42)$$

$$d\mathbf{W}_2^{(i)} = \text{reduce-scatter} \left(\left\{ d\mathbf{W}_2^{(i)} \right\}_{i=0}^{N_{\text{FSDP}}-1} \right) \quad (43)$$

$$d\mathbf{W}_3^{(i)} = \text{reduce-scatter} \left(\left\{ d\mathbf{W}_3^{(i)} \right\}_{i=0}^{N_{\text{FSDP}}-1} \right) \quad (44)$$

Note that the sharding notation (superscript (i)) is overloaded: the inputs to the reduce-scatter are partial sums on the full weights, while the outputs are full sums on the sharded weights.

Problem (fsdp_calcs): Fully sharded data parallel calculations (3 points)

Under the same setting as the data parallel calculations, let's calculate when FSDP becomes communication bottlenecked.

- (a) How many FLOPs are required to compute the backward pass, with N_{FSDP} FSDP? What about the forward pass?

Deliverable: Two answers in terms of B , D , D_{FF} , and N_{FSDP} , along with two one-sentence justifications.

- (b) How much communication time is required in the backward pass, with N_{FSDP} FSDP? What about the forward pass?

Deliverable: Two answers in terms of a subset of B , D , D_{FF} , N_{FSDP} , and W , along with two one-sentence justifications.

- (c) Fixing the other parameters, how large can N_{FSDP} become before the backward pass is communication bottlenecked? What about the forward pass?

Deliverable: Two inequalities with N_{FSDP} on one side, and an expression in terms of a subset of B, D, D_{FF}, C , and W on the other, along with two one-sentence justifications.

8.4 Analyzing Tensor Parallel

In practice, FSDP is often combined with a parallelism strategy called tensor parallelism (TP). In TP, we shard either the input or output dimension of each weight matrix across devices. Input dimension sharding is often called “row parallel,” while output dimension sharding is often called “column parallel.”

Specifically, suppose we want to shard the matmul $\mathbf{x}\mathbf{W}$, for $\mathbf{x}$ with shape (B, D) and $\mathbf{W}$ with shape (D, D_{FF}) . In column parallel we have shards $\mathbf{W}^{(i)}$ of shape $(D, \frac{D_{\text{FF}}}{N_{\text{TP}}})$, and we have

$$\mathbf{x}\mathbf{W} = \text{all-gather} \left(\left\{ \mathbf{x}\mathbf{W}^{(i)} \right\}_{i=0}^{N_{\text{TP}}-1} \right). \quad (45)$$

On the other hand, for row parallel, we have shards $\mathbf{W}^{(i)}$ of shape $(\frac{D}{N_{\text{TP}}}, D_{\text{FF}})$, and each device also narrows the input $\mathbf{x}$ into a slice $\mathbf{x}^{(i)}$ with shape $(B, \frac{D}{N_{\text{TP}}})$ before doing the matmul. Then, we have

$$\mathbf{x}\mathbf{W} = \text{all-reduce} \left(\left\{ \mathbf{x}^{(i)}\mathbf{W}^{(i)} \right\}_{i=0}^{N_{\text{TP}}-1} \right). \quad (46)$$

To parallelize our FFN, we'll use a specific tensor parallel configuration where $\mathbf{W}_1$ and $\mathbf{W}_2$ are column parallel (output-dimension-sharded), while $\mathbf{W}_3$ is row parallel (input-dimension-sharded). Since the row parallel weight only requires a slice of the input, this configuration lets us skip the all-gather after the column parallel weights. This strategy gives the following forward pass, given input $\mathbf{x}$ of size (B, D) :

$$\mathbf{x}_1^{(i)} = \mathbf{x}\mathbf{W}_1^{(i)} \quad (47)$$

$$\mathbf{x}_2^{(i)} = \mathbf{x}\mathbf{W}_2^{(i)} \quad (48)$$

$$\mathbf{z}^{(i)} = f(\mathbf{x}_1^{(i)}) * \mathbf{x}_2^{(i)} \quad (49)$$

$$\mathbf{y}^{(i)} = \mathbf{z}^{(i)}\mathbf{W}_3^{(i)} \quad (50)$$

$$\mathbf{y} = \text{all-reduce} \left(\left\{ \mathbf{y}^{(i)} \right\}_{i=0}^{N_{\text{TP}}-1} \right), \quad (51)$$

where $\mathbf{W}_1^{(i)}$ and $\mathbf{W}_2^{(i)}$ have shape $(D, \frac{D_{\text{FF}}}{N_{\text{TP}}})$, and $\mathbf{W}_3^{(i)}$ has shape $(\frac{D_{\text{FF}}}{N_{\text{TP}}}, D)$.

Problem (tp_calcs): Tensor parallel calculations (4 points)

Under the same setting as the DP and FSDP calculations, let's calculate when TP becomes communication bottlenecked.

- (a) Given input $\mathbf{dy}$ of size (B, D) write out the backward pass of the tensor parallel strategy described above (where $\mathbf{W}_1^{(i)}$ and $\mathbf{W}_2^{(i)}$ have shape $(D, \frac{D_{\text{FF}}}{N_{\text{TP}}})$, and $\mathbf{W}_3^{(i)}$ has shape $(\frac{D_{\text{FF}}}{N_{\text{TP}}}, D)$).

Deliverable: A series of equations describing the backward pass, in terms of $\mathbf{dy}$, sharded weights $(\mathbf{W}_1^{(i)}, \mathbf{W}_2^{(i)}, \mathbf{W}_3^{(i)})$, activations saved from the forward pass $(\mathbf{x}, \mathbf{x}_1^{(i)}, \mathbf{x}_2^{(i)}, \mathbf{z}^{(i)}, \mathbf{y}^{(i)})$, communication primitives, and any intermediate variables you'd like to define. The equations should produce each device's gradients $d\mathbf{W}_1^{(i)}, d\mathbf{W}_2^{(i)}, d\mathbf{W}_3^{(i)}$, and the backward pass output $d\mathbf{x}$. Feel free to reference the non-sharded backward pass in [Section 8.2](#) and modify it.

- (b) How many FLOPs are required to compute the forward pass, with N_{TP} TP? What about the backward pass?

Deliverable: Two answers in terms of B , D , D_{FF} , and N_{TP} , along with two one-sentence justifications.

- (c) How much communication time is required in the forward pass, with N_{TP} TP? What about the backward pass?

Deliverable: Two answers in terms of a subset of B , D , D_{FF} , N_{TP} , and W , along with two one-sentence justifications.

- (d) Fixing the other parameters, how large can N_{TP} become before the backward pass is communication bottlenecked? What about the forward pass?

Deliverable: Two inequalities with N_{TP} on one side, and an expression in terms of a subset of B , D , D_{FF} , C , and W on the other, along with two one-sentence justifications.

8.5 2D Parallelism (FSDP + TP)

We’re finally ready to combine parallelism strategies! In this section, we’ll look at how to optimally combine FSDP and TP. Hint: in the previous sections, you should have found that your batch size and model size parameters limit how many devices you can scale to, where making everything larger allows you to keep scaling devices without being communication bottlenecked. Unfortunately, scaling batch size past a certain point starts to degrade performance because gradients noise shrinks significantly, losing the implicit regularization properties of SGD; this point is often called the “critical batch size.” And scaling laws often tell us how large we want the model to be (that will be your job in the next assignment!).

In this section, we’ll consider a simplified setting where someone comes to you with all of the problem parameters (batch size, model size, bandwidth, accelerator speed). Your job will be to choose a configuration of FSDP and TP that scales to as many devices as possible, while remaining compute-bound rather than communication-bound.

Let’s first walk through the mechanics of combining FSDP with TP. Each device will have a TP rank $i = 0, \dots, N_{\text{TP}} - 1$ and an FSDP rank $j = 0, \dots, N_{\text{FSDP}} - 1$, forming a 2D grid with $N = N_{\text{TP}} N_{\text{FSDP}}$ devices total. Following TP, we’ll first shard $\mathbf{W}_1$ and $\mathbf{W}_2$ along the output dimension, and $\mathbf{W}_3$ along the input dimension. As a result, we’ll have to insert a TP-style all-reduce on the activations. Next, applying FSDP, we’ll shard the batch dimension of the inputs, and we’ll also further shard each weight matrix along whichever dimension wasn’t sharded by TP. Then, we’ll have to insert FSDP-style all-gathers on the weights before doing our TP-style forward/backward passes, and reduce-scatters on the weight gradients after our TP-style backward pass.

The result is that each device (i, j) holds the weight shards $\mathbf{W}_1^{(i,j)}$ and $\mathbf{W}_2^{(i,j)}$ with shape $\left(\frac{D}{N_{\text{FSDP}}}, \frac{D_{\text{FF}}}{N_{\text{TP}}}\right)$, and $\mathbf{W}_3^{(i,j)}$ with shape $\left(\frac{D_{\text{FF}}}{N_{\text{TP}}}, \frac{D}{N_{\text{FSDP}}}\right)$. We can then write out the forward pass as the following, given batch-sharded input $\mathbf{x}^{(j)}$ of size $\left(\frac{B}{N_{\text{FSDP}}}, D\right)$:

$$\mathbf{W}_1^{(i)} = \text{all-gather} \left(\left\{ \mathbf{W}_1^{(i,j)} \right\}_{j=0}^{N_{\text{FSDP}}-1} \right) \quad (52)$$

$$\mathbf{W}_2^{(i)} = \text{all-gather} \left(\left\{ \mathbf{W}_2^{(i,j)} \right\}_{j=0}^{N_{\text{FSDP}}-1} \right) \quad (53)$$

$$\mathbf{W}_3^{(i)} = \text{all-gather} \left(\left\{ \mathbf{W}_3^{(i,j)} \right\}_{j=0}^{N_{\text{FSDP}}-1} \right) \quad (54)$$

$$\mathbf{x}_1^{(i,j)} = \mathbf{x}^{(j)} \mathbf{W}_1^{(i)} \quad (55)$$

$$\mathbf{x}_2^{(i,j)} = \mathbf{x}^{(j)} \mathbf{W}_2^{(i)} \quad (56)$$

$$\mathbf{z}^{(i,j)} = f\left(\mathbf{x}_1^{(i,j)}\right) * \mathbf{x}_2^{(i,j)} \quad (57)$$

$$\mathbf{y}^{(i,j)} = \mathbf{z}^{(i,j)} \mathbf{W}_3^{(i)} \quad (58)$$

$$\mathbf{y}^{(j)} = \text{all-reduce} \left(\left\{ \mathbf{y}^{(i,j)} \right\}_{i=0}^{N_{\text{TP}}-1} \right), \quad (59)$$

ending up with batch-sharded output $\mathbf{y}^{(j)}$ of size $\left(\frac{B}{N_{\text{FSDP}}}, D\right)$. We'll omit the backward pass for brevity in this section, and just focus on the forward pass. But at this point, you should have all the information you need to write it out yourself.

Problem (fsdp_tp_calcs): 2D parallelism calculations (6 points)

Under the same setting as the calculations so far, let's calculate when 2D parallelism becomes bottlenecked.

- (a) How many FLOPs are required to compute the forward pass, with N_{FSDP} FSDP + N_{TP} TP?

Deliverable: An answer in terms of B , D , D_{FF} , N_{FSDP} , and N_{TP} , along with a one-sentence justification.

- (b) How much communication time is required in the forward pass, with N_{FSDP} FSDP + N_{TP} TP? Assume that the communication along each axis can be overlapped (in other words, the collectives along the FSDP axis can be overlapped with the collectives along the TP axis).

Deliverable: An answer in terms of a subset of B , D , D_{FF} , N_{FSDP} , N_{TP} , and W , along with a one-sentence justification. Hint: The answer should be expressed as a max between two quantities (the FSDP and TP collective costs), since the two can be overlapped.

- (c) Under the optimal setting of N_{TP} and N_{FSDP} , how large can $N = N_{\text{TP}}N_{\text{FSDP}}$ become before the forward pass is communication bottlenecked?

Deliverable: An inequality with N on one side, and an expression in terms of a subset of B , D , D_{FF} , C , and W on the other, along with a few sentences and equations as justification.

- (d) Now suppose the FSDP-axis and TP-axis collectives cannot be overlapped because they share the same network resources. Under the optimal setting of N_{TP} and N_{FSDP} , how large can $N = N_{\text{TP}}N_{\text{FSDP}}$ become before the forward pass is communication bottlenecked? Don't worry about truncating N_{TP} and N_{FSDP} to be integers.

Deliverable: An inequality with N on one side, and an expression in terms of a subset of B , D , D_{FF} , C , and W on the other, along with a few sentences and equations as justification.

9 Leaderboard

Assignment 2's leaderboard will test the speed of a full training step for an 8B model. We challenge you to benchmark your code and to optimize memory and runtime, using any tricks you can come up with. The key restrictions are that you cannot change the input/output behavior of the model. Your implementation will be tested against the model in the `cs336_basics` directory. Your inputs will be tested at BF16 with causal masking, and they must pass the same tests as your regular implementation. The implementation must also be your own, and you cannot use or copy pre-existing implementations. Your timing should be measured on two B200 GPUs using a sample with batch size 2 and sequence length 32,768. It is intentionally difficult to fit the model in memory. See the code below for the full config. We

will verify the top 5-10 submissions for correctness and performance. The test we will run to time your implementation is the following:

```
class Config:
    ctx_len = 32768
    vocab_size = 151936
    d_model = 4096
    d_ff = 11008
    num_layers = 34
    num_heads = 32
    torch_dtype = torch.bfloat16
    is_causal = True
    batch_size = 2
cfg = Config()

def test_timing_forward_backward():
    labels, targets = torch.randint(high=cfg.vocab_size, size=(2, cfg.batch_size,
    cfg.ctx_len))

    model = BasicsTransformerLM(Config())
    optimizer = AdamW(model.parameters())

    def train_step():
        optimizer.zero_grad(set_to_none=True)
        res = model(labels)
        loss = cross_entropy(res, targets).sum()
        loss.backward()
        optimizer.step()

    timing_results = triton.testing.do_bench(train_step, rep=30_000, warmup=10_000)
    print(timing_results)
```

For testing purposes, you can reduce the repetition and warmup time (given in ms) to something shorter.

Some ideas for improvement and for making sure your model fits:

- Tune the tile sizes for your kernel (use Triton autotune for this!)
- Tune additional Triton/torch.compile config parameters
- Implement fused AdamW
- The base implementation is materializing the full logits ([batch, seq_len, vocab_size]). Write a kernel that fuses the LM head and your cross-entropy loss. You can also have it compute the backward pass immediately in a fused manner
- Improve FlashAttention
 - Implement the backward pass in Triton, not just torch.compile (see [Section 4.2.3](#))
 - 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 tiles that are guaranteed to be 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 architectures later than Hopper, following a similar pattern to our tutorial
- Use activation checkpointing to trade runtime speed for memory savings only if you need it

Problem (leaderboard): Leaderboard: fastest training step (10 points)

The benchmark will be run at batch size 2 on two B200 GPUs. Your submission will be evaluated on wall-clock time for a complete training step: forward pass, loss, backward pass, and AdamW update.

From an empty PyTorch/Triton cache, your benchmarking run must complete within 10 minutes, so be careful with overly aggressive `torch.compile` and Triton autotuning.

Deliverable: Your best wall-clock time for a full forward-and-backward training step with AdamW.

We expect leaderboard submissions to beat the naïve baseline of 10 seconds.

Submit your result to the leaderboard here:

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

Bibliography

- [1] T. Dao, “FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning.” [Online]. Available: <https://arxiv.org/abs/2307.08691>
- [2] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Re, “FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness,” in *Advances in Neural Information Processing Systems*, A. H. Oh, A. Agarwal, D. Belgrave, and K. Cho, Eds., 2022. [Online]. Available: <https://openreview.net/forum?id=H4DqfPSibmx>
- [3] M. Milakov and N. Gimelshein, “Online normalizer calculation for softmax.” [Online]. Available: <https://arxiv.org/abs/1805.02867>
- [4] H. He, “Making Deep Learning Go Brrrr From First Principles,” 2022, [Online]. Available: https://horace.io/brrr_intro.html
- [5] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, “ZeRO: Memory Optimizations Toward Training Trillion Parameter Models.” 2020.
- [6] J. Austin *et al.*, “How to Scale Your Model,” 2025.
- [7] H. Z. P. N. M. M. L. W. T. W. Nouamane Tazi Ferdinand Mom, “The Ultra-Scale Playbook: Training LLMs on GPU Clusters.” 2025.