CS336 Assignment 2 (systems): Systems and Parallelism

Version 0.0.1

Spring 2024

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. Fused RMSNorm Triton kernel
3. Distributed data parallel training
4. Optimizer state sharding

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

`github.com/stanford-cs336/spring2024-assignment2-systems`

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

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

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

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

2 Optimizing single-GPU performance

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

2.1 Profiling and benchmarking

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

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

2.1.1 Setup - importing your `basics` Transformer model

Let's start by making sure that you can load your 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'll copy the contents on that package here (in the `cs336-basics/`) folder and pip install the package within it (named `cs336_basics`) and the package that will contain our assignment 2 code (named `cs336_systems`). Now, after you set up the `conda` environment for this assignment (e.g., with the default name of `cs336_systems`), you can do:

```
spring2024-assignment2-systems$ conda activate cs336_systems
(cs336_systems) pip install -e ./cs336-basics/ ./cs336-systems/'[test]'
```

You can test that you can import your model with:

```
(cs336_systems) ~$ python
Python 3.10.14 (main, Mar 21 2024, 16:24:04) [GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import cs336_basics
>>>
```

Any module you wrote in assignment 1 should now be available (e.g., if you had `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, we'll use a context length of 128, a vocabulary size of 10,000, and a batch size of 16.

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 (swapping layers, changing precision, enabling/disabling Tensor Cores, 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 and backward passes. Since we will only be measuring speed and memory, we will use random weights and data.

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

Table 1: Specifications of different model sizes

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`, this function calls returns control to your code without waiting for the matrix multiplication to finish. In this way, the CPU can continue running while the GPU computes the matrix multiplication. On the other hand, this means that naïvely measuring how long the `torch.matmul` call takes to return does not tell us how long the GPU takes to actually run the matrix multiplication. In PyTorch, we can call `torch.cuda.synchronize()` to wait for all GPU kernels to complete, allowing us to get more accurate measurements of CUDA kernel runtime. With this in mind, let's write our basic profiling infrastructure.

Problem (`benchmarking_script`): 4 points

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

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

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

(b) Time the forward and backward passes for the model sizes described in §2.1.2. Use 1 warmup step and compute the average and standard deviation of timings over 5 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 step. Repeat your analysis without the warm-up step. How does this affect your results? Why do you think this happens?

Deliverable: A 2-3 sentence response.

2.1.4 PyTorch 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).

PyTorch ships with a profiler that we can use under `torch.profiler`. In this part of the assignment, you will use the PyTorch profiler to analyze the runtime of your Transformer model. We will assume the following imports:

```
from torch.profiler import profile, record_function, ProfilerActivity
```

The `record_function` environment will be used to to aggregate metrics for all operations in a given subtask (such as the forward pass), which we'll be able to see later. For example, you can wrap your forward and backward passes, as well as the optimizer step, as follows:

```
def run_step(model, inputs, optimizer, enable_backward, ...):
    with record_function('forward_pass'):
        model.forward(inputs, ...)

    if enable_backward:
        with record_function('backward_pass'):
            loss = ...; loss.backward()
        with record_function('optimizer'):
            optimizer.step()
```

To start the PyTorch profiler, we can wrap our code block that calls a function such as `run_step` in a `profile` environment (imported above). We will have the profiler record the stack traces to enable us to generate a flame graph later:

```
# Remember to do a warm-up step first. Then:
with profile(
        activities=[
            torch.profiler.ProfilerActivity.CPU,
            torch.profiler.ProfilerActivity.CUDA,
        ], experimental_config=torch._C._profiler._ExperimentalConfig(verbose=True),
        record_shapes=True,
        profile_memory=False,
        with_stack=True,
) as prof:
    for _ in range(n_steps):
        run_step(model, ...)
        prof.step()

prof.export_stacks("lm_profiler_stacks.txt", "self_cuda_time_total")
print(prof.key_averages().table(sort_by="cpu_time_total", row_limit=50))
```

This will print a table of functions with their performance statistics, ordered by cumulative wallclock time spent in them. You will also see CUDA kernels, typically coming from PyTorch's internal `ATen`[1] library ("A Tensor library"). In this table, the "Self" columns (like "Self CUDA") indicate how much time is spent

[1] `https://github.com/pytorch/pytorch/tree/main/aten`

in the body of that particular function, whereas the "total" columns (like "CUDA total") includes both that function and any function that it calls (this, "total" accumulates time across the stack trace, whereas "self" does not).

Problem (`function_call_table`): 3 points

Add an option to your script to profile a `XL`-shaped language model (§2.1.2) using the PyTorch profiler, as explained above. Then, profile your forward pass, backward pass, and optimizer step and look at the table of wallclock time spent in each function.

(a) What is the total time spent on your forward pass, as measured by the PyTorch profiler? Does it match what we had measured before with the Python standard library? (Hint: use the "CPU Total" and "GPU Total" columns, and look for the row corresponding to your `record_function` block)

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?

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.

Besides the table, we will also analyze a *flame graph* that shows a timeline of function calls of the program, as well as how long was spent on each. We will use this tool to generate an SVG file from the PyTorch-exported stack traces. Running the code above should produce a non-empty `lm_profiler_stacks.txt`. Given that file, run the following to generate a flame graph from the stack traces:

```
git clone https://github.com/brendangregg/FlameGraph
cd FlameGraph
./flamegraph.pl --title "CUDA time" --countname "us." lm_profiler_stacks.txt > lm-flame-graph.svg
```

The flame graph will help us identify the performance of each of our layers. You can open `lm-flame-graph.svg` in any program that supports SVGs, like most Web browsers.

Problem (`flame_graph`): 2 points

Run with the PyTorch profiler with a `XL`-shaped language model's (§2.1.2) forward pass and generate a flame graph. Then, answer the questions below.

(a) Include a snapshot of your flame graph here. You should see a cyclic pattern in your graph –

where does that come from?

Deliverable: An image and a one sentence response.

(b) What fraction of time does your model seem to spend on all RMSNorm layers? (Hint: you can zoom in and analyze one Transformer block, then multiply by the number of layers.)

Deliverable: A one sentence response.

(c) What fraction of the time is spent in softmax (inside the attention operation)?

Deliverable: A one sentence response.

(d) Does the graph match your prior expectations of how runtime is distributed in your Transformer? Did you see any surprises?

Deliverable: A 2-4 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 A100 spec sheet says that its maximum throughput with FP32 is 19.5 TFLOP/second, while its maximum throughput with FP16 (half-precision floats) or BF16 (brain floats) is significantly higher at 312 TFLOP/second. As a result, using lower-precision datatypes should help us speed up training and inference.

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, consider the following accumulation when performed in FP16 vs. FP32:

```
>>> (torch.tensor(100, dtype=torch.float16) + torch.tensor(0.01, dtype=torch.float16)).item()
100.0

>>> (torch.tensor(100, dtype=torch.float32) + torch.tensor(0.01, dtype=torch.float32)).item()
100.01000213623047
```

Problem (`benchmarking_mixed_precision`): 1 point

(a) Modify your benchmarking script to optionally run the model with mixed precision. Time the forward and backward passes with and without mixed-precision for each language model size described in §2.1.2. Compare the results of using full vs. mixed precision, and comment on any trends as model size changes. You may find the `nullcontext` no-op context manager to be useful.

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

(b) Suppose we are training the following model on a GPU:

```
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 that the model parameters are originally in FP32. We'd like to use autocasting mixed precision with FP16. What are the data types of:

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

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

(c) 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.

2.1.6 RMSNorm

Your profiling likely suggests that there is an opportunity for optimization in your normalization layers. Recall that one of the motivations for RMSNorm is that it requires fewer operations compared to full LayerNorm, and thus can be more efficient in principle. Let's see if that holds when comparing your implementation of RMSNorm with PyTorch's native `torch.nn.LayerNorm`.

Problem (`pytorch_layernorm`): 2 points

(a) Benchmark your RMSNorm implementation against PyTorch's native LayerNorm. For that, write a script that will:

(a) Fix the number of rows in the input matrix as 50,000.

(b) Iterate through the following values for the size of the last dimension: `[1024, 2048, 4096,`

8192]

(c) Create random inputs x and w for the appropriate size. For LayerNorm, also create a random bias term.

(d) Time 1,000 forward passes through each normalization layer using those inputs.

(e) Make sure to warm up and call `torch.cuda.synchronize()` after each forward pass.

Report the timings you get for these 3 implementations. Which implementation seems faster — RMSNorm or LayerNorm? Why do you think that's the case? How does this gap seem to evolve as the hidden dimension grows?

Deliverable: A table with your timings, and a 2-3 sentence response.

(b) For each language model size described in §2.1.2, replace RMSNorm in the Transformer model by PyTorch's native LayerNorm. How long does a forward pass (with our previous default parameters for model size, etc) take on average?

(Hint: you can modify your Transformer model to take flags in the constructor that indicate which implementation of normalization layer to use. Then, instantiate your Transformer with different flags to measure the effect on the forward pass).

Deliverable: A 2-3 sentence response.

2.2 Writing a fused RMSNorm kernel

We'll now try to close the gap between our RMSNorm layer and PyTorch's LayerNorm. For that, we will write a Triton implementation of RMSNorm, allowing us to have more control over how the computation executes on the GPU.

2.2.1 Example - Weighted sum

To review what you'll need to know about Triton and how it interoperates with PyTorch, let's look at a simple example first: writing a kernel for a "weighted sum" operation: 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 2D shape [N, H], and weight has 1D shape [H]
    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 single row of x, and write a scalar output to the output tensor. Instead of taking *tensors* as arguments, we take *pointers* to their first elements, as well as a *stride* for x that tells us how to move between rows. We can use the stride to load a tensor corresponding to the row 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 row 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:

```
import triton
import triton.language as tl

def weighted_sum_fwd(
```

```
        x_ptr : tl.pointer_type,
        weight_ptr : tl.pointer_type,
        x_row_stride : tl.uint32,
        output_ptr : tl.pointer_type,
        H : tl.uint32,
        BLOCK_SIZE: tl.constexpr):
    # Each instance will compute the weighted sum of a row of x.
    row_idx = tl.program_id(0)
    # Pointer to the first entry of the row this instance sums up.
    row_start_ptr = x_ptr + row_idx * x_row_stride
    offsets = tl.arange(0, BLOCK_SIZE)
    # Pointers to the entries we'll sum up.
    x_ptrs = row_start_ptr + offsets
    weight_ptrs = weight_ptr + offsets
    # Load the data from x given the pointers to its entries,
    # using a mask since BLOCK_SIZE may be > H.
    mask = offsets < H
    row = tl.load(x_ptrs, mask=mask, other=0)
    weight = tl.load(weight_ptrs, mask=mask, other=0)
    output = tl.sum(row * weight)
    # Write back output (a single scalar per instance).
    output_ptr = output_ptr + row_idx
    tl.store(output_ptr, output)
```

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

```
class WeightedSumFunc(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x, weight):
        # Remember x and weight for the backward pass, when we
        # only receive the gradient wrt. the output tensor, and
        # need to compute the gradients wrt. x and weight.
        ctx.save_for_backward(x, weight)

        H, output_dims = x.shape[-1], x.shape[:-1]

        assert len(weight.shape) == 1 and weight.shape[0] == H, "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.BLOCK_SIZE = triton.next_power_of_2(H)
        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[(n_rows, )](
            x, weight, x.stride(0), y, H,
            num_warps=16, BLOCK_SIZE=ctx.BLOCK_SIZE)
        return y
```

Backward pass Since we are defining our own kernel, we will also need to write our own backward function. This will involve writing down the Jacobian-vector product for the backward step by hand, and then turning that into a Triton kernel.

In the forward pass, we were given the inputs to our layer, and needed to compute its outputs. In the backward pass, recall that we will be given the gradients of the objective with respect to our outputs, and need to compute the gradient with respect to each of our inputs. In our case, our operation has as inputs a matrix $x : \mathbb{R}^{n \times h}$ and a weight vector $w : \mathbb{R}^h$. For short, let's call our operation $f(x, w)$, whose range is $\mathbb{R}^n$. Then, assuming we are given $\nabla_{f(x,w)} L$ – the gradient of loss L with respect to the output of our layer, our job is now to compute $\nabla_x L$ and $\nabla_w L$. By convention, we see each of these gradients as tensors of the same shape as their corresponding input tensors (thus, we'll have $\nabla_x L : \mathbb{R}^{n \times h}$, $\nabla_w L : \mathbb{R}^h$ and receive $\nabla_{f(x,w)} L : \mathbb{R}^h$). In the multi-dimensional case, the chain rule tells us that each such gradient is given by the Jacobian-vector product (JVP):

$$\nabla_w L = J_w(x, w)^\top \nabla_{f(x,w)} L \tag{1}$$

$$\nabla_x L = J_x(x, w)^\top \nabla_{f(x,w)} L \tag{2}$$

Here, J is the Jacobian: the matrix of partial derivatives of all output dimensions with respect to each input dimension — with respect to the entries in w in the case of J_w, and the entries of x in J_x.[2] To operationalize the backward pass, we avoid computing J explicitly, and directly try to find an expression for the JVP that we might be able to evaluate more efficiently. Thus, since $f(x, w)$ has n elements, we have:

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

$$(\nabla_x L)_{ij} = \sum_{k=1}^{n} \frac{\partial f(x, w)_k}{\partial w_i} (\nabla_{f(x,w)} L)_i = w_j \cdot (\nabla_{f(x,w)} L)_i \tag{4}$$

This gives a simple formula for computing the backward pass. To obtain the backward step with respect to x, we apply Eq 4 and take the outer product of w and $\nabla_{f(x,w)}$. To compute the backward step with respect to w (i.e. $(\nabla_w L)_j$), we must multiply our input gradient by the corresponding output row.

Our kernel for the backward pass will start by computing $\nabla_x L$, which is simpler:

```
@triton.jit
def weighted_sum_backward(
        grad_output_ptr : tl.pointer_type,
        grad_x_ptr : tl.pointer_type,
        partial_grad_weight_ptr : tl.pointer_type,
        x_ptr : tl.pointer_type,
        weight_ptr : tl.pointer_type,
        x_row_stride : tl.uint32,
        H : tl.uint32,
        BLOCK_SIZE: tl.constexpr):
    row_idx = tl.program_id(0)
    row_start_ptr = x_ptr + row_idx * x_row_stride
    offsets = tl.arange(0, BLOCK_SIZE)
    x_ptrs = row_start_ptr + offsets
    grad_output_ptrs = weight_ptr + offsets
    mask = offsets < H
```

[2] Since x is a matrix, $J_x(x, w)$ would technically be a 3d tensor. To simplify exposition, we can imagine x here as a vector of dimension $n \times h$.

```
    weight = tl.load(weight_ptr + offsets, mask=mask, other=0)
    # Gradient with respect to the output of our operation at row_idx.
    grad_output = tl.load(grad_output_ptr + row_idx) # (scalar)
    # Compute gradient with respect to the current row of x.
    grad_x_row = grad_output * weight  # (See Eq 4)
    # Move grad_x_ptr to the right output row and write the gradient.
    grad_x_ptr = grad_x_ptr + row_idx * x_row_stride
    tl.store(grad_x_ptr + offsets, grad_x_row, mask=mask)
```

All operations up to here should be familiar from the forward pass. Now, for computing ∇_w, we have a different situation. Each kernel instance is responsible for one row 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_buffer` contains an $N \times H$ matrix, where each row is the contribution of the corresponding row of x to ∇_w. We will later obtain the full ∇_w using `torch.sum` to sum up these rows[3]. Using that assumption, the last bit of the backward kernel will be simple:

```
    # Now compute partial gradient with respect to the weight vector.
    # We will write one row to partial_grad_weight_ptr, and later
    # accumulate these rows to compute the gradient w.r.t. the weight vector.
    partial_grad_weight_ptr = partial_grad_weight_ptr + row_idx * x_row_stride + offsets
    row = tl.load(row_start_ptr + offsets, mask=mask, other=0)
    grad_weight_row = row * grad_output  # (See Eq 3)
    tl.store(partial_grad_weight_ptr, grad_weight_row, mask=mask)
```

We can now complete our PyTorch autograd function:

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

    @staticmethod
    def backward(ctx, grad_out):
        x, weight = ctx.saved_tensors
        N, H = x.shape
        # Allocate output tensors.
        partial_grad_weight = torch.empty_like(x)
        grad_x = torch.empty_like(x)

        weighted_sum_backward[(N, )](
            grad_out, grad_x, partial_grad_weight,
            x, weight, x.stride(0), H,
            num_warps=16, BLOCK_SIZE=ctx.BLOCK_SIZE)
        return grad_x, partial_grad_weight.sum(axis=0)
```

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>)
```

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

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.

2.2.2 RMSNorm forward pass

You will now replace your previous RMSNorm implementation in PyTorch by a faster Triton implementation. Recall that the RMSNorm layer has learnable parameters $g : \mathbb{R}^{\texttt{d\_model}}$, and the forward pass performs the following operation on a vector $x : \mathbb{R}^{\texttt{d\_model}}$:

$$\text{RMSNorm}(x, g) = \frac{x}{\sqrt{\frac{1}{d_{\text{model}}} \sum_{i=1}^{d_{\text{model}}} x_i^2 + \varepsilon}} \odot g \tag{5}$$

To get started, you will first implement the forward pass. While we have been talking about x as a matrix, to be compatible with its use in the Transformer layers, your implementation should support arbitrary tensors, and normalize over the last dimension. When using your Triton kernel, you can use `torch.view` to see the tensor as a matrix, then call the kernel and use `torch.view` again to restore the desired shape.

Problem (`triton_rmsnorm_forward`): 6 points

(a) Write a PyTorch `autograd.Function` that implements the RMSNorm forward pass. Your function should take two parameters: `x` (a *tensor* where the last dimension has shape H) and `weight` (a vector of dimension H). Remember that the implementation of the `forward` method 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`. For your first version, you can just implement Equation 5 in PyTorch.

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

(b) Write a Triton kernel for the forward pass of RMSNorm. Then, write another subclass of `torch.autograd.Function` that calls this (fused) kernel in the forward pass, instead of computing the result in PyTorch.

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

Let's now compare the performance of your Triton implementation of RMSNorm against both your PyTorch implementation, as well as with PyTorch's LayerNorm.

Problem (`rmsnorm_forward_benchmarking`): 2 points

(a) Extend your benchmarking script from the problem `pytorch_layernorm` to also have your Triton RMSNorm implementation as an alternative normalization layer. Use the same configurations from that existing script.

Report the timings you get for these 3 implementations. Do you see a speed-up with your Triton

kernel for any of those sizes? What is the smallest size of the last dimension for which you see a speed-up?

Deliverable: A table or plot with your measurements, and a one-two sentence response.

(b) Replace your original RMSNorm layers in the Transformer model with (a) your Triton implementation, in addition to (b) with PyTorch's native LayerNorm, and benchmark your forward pass with our standard hyperparameters. Report the timings you get with each normalization layer.

Deliverable: A table with forward pass latencies with each of the normalization layer implementations. You can repeat the numbers you got before, just include them for ease of comparison.

2.2.3 Backward pass

For the backward pass, we will implement a specialized kernel that will be equivalent to what PyTorch's automatic differentiation engine (autograd) computes in the backward pass. Again, the ability to fuse operations will allow our Triton kernel to be faster. Moreover, we will leverage a second technique for performance: *recomputation*. Since memory operations are expensive, it is often faster to recompute intermediate results in the backward pass, rather than storing them into DRAM during the forward pass and loading them back. We will apply recomputation to the normalization layer.

Before implementing the backward pass, we first have to work out the Jacobian-vector products for the RMSNorm layer.

Problem (`rmsnorm_jvp_g`): 4 points

(a) Assume that x is a $N \times H$ matrix, and that g is a H-dimensional vector. Derive the JVP for the g vector in RMSNorm (Equation 5).

(Hint: start with Equation 5 and take the partial derivative of a given entry in the output vector with respect to a given entry of g).

Deliverable: A derivation ending with a simple expression to compute $\nabla_g L$ when given x, g and $\nabla_{\text{RMSNorm}(x,g)} L$.

(b) Write a PyTorch function that uses your expression to compute $\nabla_g L$. It should take 3 tensor parameters: $\nabla_g L$, x and g. Note that x might have any shape, as long as the last dimension matches the dimensionality of g.

Deliverable: A function to compute $\nabla_g L$.
Implement the adapter `[adapters.rmsnorm_backward_g_pytorch]` to call your function. Then, make sure it passes the test `pytest -k test_rmsnorm_backward_g_pytorch`

Now, we do the same procedure for the x parameter of RMSNorm.

Problem (`rmsnorm_jvp_x`): 8 points

(a) Assume that x is a $N \times H$ matrix, and that g is a H-dimensional vector. Derive the JVP for the x input matrix in RMSNorm (Equation 5).

(Hint: start with Equation 5 and take the partial derivative of a given entry $\text{RMSNorm}(x, g)_k$ in the output vector with respect to a given x_{ij}).

Deliverable: A derivation ending with a simple expression to compute $\nabla_x L$ when given x, g

and $\nabla_{\text{RMSNorm}(x,g)} L$.

(b) Write a PyTorch function that uses your expression to compute $\nabla_x L$. It should again take 3 tensor parameters: $\nabla_g L$, x and g. As before, x might have any shape, as long as the last dimension matches the dimensionality of g.

Deliverable: A function to compute $\nabla_x L$.
Implement the adapter `[adapters.rmsnorm_backward_x_pytorch]` to call your function. Then, make sure it passes the test `pytest -k test_rmsnorm_backward_x_pytorch`

(c) Complete the backward pass of your `autograd.Function` subclass that computes RMSNorm in PyTorch. Remember that you need to save x and w to the context in the forward pass to be able to use them in `backward`.

Deliverable: A complete `autograd.Function` subclass for RMSNorm, with both the forward and backward passes. This class should already be hooked into the adapter `[adapters.rmsnorm_autograd_function_pytorch]` from the previous problem. After completing the backward pass, make sure it passes the test
`pytest -k test_rmsnorm_autograd_pytorch_forward_backward`

After writing a PyTorch implementation of the backward pass that matches PyTorch's autograd engine, you will now make it as efficient as possible. For that, you will write an efficient fused Triton kernel to do the backward pass. When computing ∇_w, you will notice the same issue we had in our WeightedSum example: we need to accumulate the gradient across rows of x, even though our kernel will run one instance for each row. For this problem, you can use the same simple solution as before: you can allocate a buffer for partial gradients that your kernel writes to, and then obtain ∇_w using `torch.sum`.

Problem (`triton_rmsnorm_backward`): 4 points

(a) Write a Triton kernel to perform the backward pass of RMSNorm. Your kernel should take pointers to x, g, $\nabla_{\text{RMSNorm}(x,g)} L$, as well as to the output tensors ∇_x and the partial gradients of ∇_g. Then, complete your `autograd.Function` subclass that uses Triton kernels by implementing the `backward` method.

Deliverable: A complete `autograd.Function` sucblass that uses Triton kernels to implement RMSNorm. Your subclass should already be hooked in the adapter `[adapters.rmsnorm_autograd_function_triton]`. Make sure it now passes the test `pytest -k test_rmsnorm_autograd_triton_forward_backward`

Let's now again benchmark the combined forward and backward passes of your RMSNorm implementation using Triton against the alternatives we've been looking at (your PyTorch implementation, and PyTorch's native LayerNorm).

To execute the backward pass, we typically call `loss.backward()` with respect to some scalar `loss` value. However, we can also do `some_tensor.backward(grad)`, where we manually provide PyTorch with the gradient `grad` with respect to the tensor `some_tensor`. Let's now benchmark your forward and backward kernels together.

Problem (`rmsnorm_benchmarking`): 2 points

(a) Extend your benchmarking script for normalization layers to optionally execute a backward pass. To run the backward pass, simply call `result.backward(dy)`, where `result` is the output of

the forward pass, and `dy` is another random tensor (in addition to the random inputs for the forward pass) of the appropriate shape. Remember to clear the gradients (set `tensor.grad = None`) of the input tensors before each forward pass; otherwise, the backward pass will also accumulate gradients at each time iteration. Then, show the average time for a combined forward and backward pass for each normalization layer implementation so far.

Deliverable: A table with your timings.

(b) Swap your PyTorch implementation of RMSNorm with your Triton implementation, and measure end-to-end performance. How long does your forward pass take now? What about your backward pass? What speed-up do you get in each case?

(Hint: to make your Triton implementation a drop-in replacement for your previous one, you can write a simple `torch.nn.Module` subclass that has a `g nn.Parameter`, and that simply calls your Triton autograd function in the forward pass).

Deliverable: Two-three sentences with your timings and analysis.

2.3 PyTorch JIT compiler

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. In particular, it will try to automatically generate fused Triton kernels by dynamically analyzing your computation graph. The interface to use the PyTorch compiler is very simple. For instance, if we wanted to apply it to a single layer of our model, we can use:

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

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

Problem (`torch_compile`): 2 points

(a) Extend your RMSNorm benchmarking script to include another candidate: a compiled version of your PyTorch implementation of RMSNorm. How does it compare to the existing layers in the forward pass?

Deliverable: A table with your timings for the forward pass including your compiled RMSNorm layer.

(b) Run your analysis including the backward pass. How does the compiled RMSNorm layer perform?

Deliverable: A table with your timings for the combined forward and backward passes including your compiled RMSNorm layer.

(c) 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? Conduct this analysis for each language model size described in §2.1.2.

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

2.4 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 use:

```python
from torch.profiler import profile

# Start recording memory history.
torch.cuda.memory._record_memory_history(max_entries=1000000)
n_steps = 3

with profile(
    activities=[
        torch.profiler.ProfilerActivity.CPU,
        torch.profiler.ProfilerActivity.CUDA,
    ],
    schedule=torch.profiler.schedule(wait=0, warmup=0, active=1, repeat=n_steps),
    experimental_config=torch._C._profiler._ExperimentalConfig(verbose=True),
    record_shapes=True,
    profile_memory=True,
    with_stack=True,
) as prof:
    for _ in range(n_steps):
        # run model on a batch of data...
        prof.step()
    # Save a graphical timeline of memory usage.
    prof.export_memory_timeline("timeline.html", device=device)

# 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 two files:

`timeline.html` will be a regular HTML file containing a visual timeline of memory allocations. PyTorch tries to heuristically classify each allocation into a few pre-defined categories (like model weights, optimizer state, activations, etc), but this not always works with custom code. You can open this file in a Web browser to see the timeline.

`memory_snapshot.pickle` will be a pickle file that you can load into the following online tool: `https://pytorch.org/memory_viz` . This tool can also show a memory usage timeline (without the categoration), but also lets you see 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`): 8 points

(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

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

(b) What is the peak memory usage of each model size when doing a forward pass? What about when doing a full optimizer step?

Deliverable: A table with two numbers per model size.

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

Deliverable: A 2-3 sentence response.

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

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

(e) Now look closely at the "Active Memory Timeline" from `pytorch.org/memory_viz` of a memory snapshot of the `2.7B` model doing a forward pass. When you reduce the "Detail" level, the tool hides the smallest allocations to the corresponding level (e.g., putting "Detail" at 10% only shows the 10% largest allocations). What is the size of the largest allocations shown? Looking through the stack trace, can you tell where those allocations come from?

Deliverable: A 1-2 sentence response.

3 Distributed data parallel training

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

3.1 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.[4]

```
$ 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 must accept this integer rank as its first positional argument. In addition, we pass in the `world_size`, which refers to the total number of worker processes.

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

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

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

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

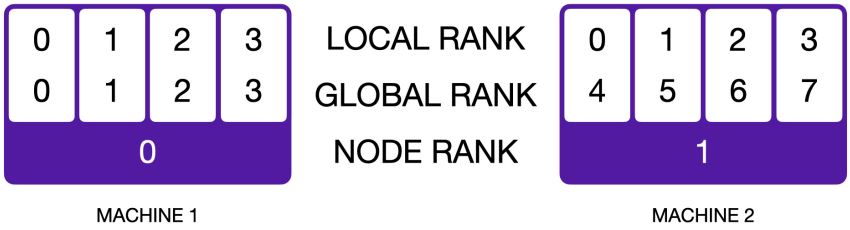

Figure 1: A schematic representation of a distributed application running on 2 nodes with a world size of 8. Each worker process is identified by a global rank (from 0 to 7) and a local rank (from 0 to 3). Figure taken from lightning.ai/docs/fabric/stable/advanced/distributed_communication.html

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. See Figure 1 for a visual representation.

node: a machine on the network.

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

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

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

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

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

3.1.1 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.

- 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.

Problem (`distributed_communication_single_node`): 3 points

Write a script to benchmark the runtime of the all-reduce operation in the single-node multi-process setup. The example code above may provide a reasonable starting point. Experiment with varying the following settings:

Backend + device type: Gloo + CPU, Gloo + GPU, NCCL + GPU.

all-reduce data size: float32 data tensors ranging 512KB, 1MB, 10MB, 50MB, 100MB, 500MB, 1GB.

Number of processes: 2, 4, or 6 processes.

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.

3.2 Multi-node Distributed Communication in PyTorch

Thus far, we've covered distributed communication in the single-node multi-process setting. In practice, we want to train models on multiple nodes, which accordingly requires communication across machines. Let's modify our example above to work in a multi-node setup to better understand how it works. We'll be using Slurm to manage resources and run scripts across multiple nodes.[5]

First, we'll write a Slurm sbatch job script. Using the SBATCH flags, we'll define a job that spans two nodes, with four tasks per node. We allocate 2 CPUs to each of those tasks and 4G of memory to each node. Thus, our world size is 8 (4 tasks per node, 2 nodes), with each node having a local world size of 4 (4 tasks per node). In our job script, we'll activate a conda environment and export the environment variables `MASTER_ADDR` and `MASTER_PORT`. `MASTER_ADDR` is set to the hostname of the first node, and `MASTER_PORT` is assigned based off of the job's numeric ID (to prevent cases where the port might be already taken on the master). Finally, the `srun` command will execute the given command for each task—in this case, it'll execute the given command 8 times (4 tasks per node, 2 nodes).

```
#!/bin/bash
#SBATCH --cpus-per-task=8
#SBATCH --ntasks-per-node=4
#SBATCH --nodes=2
#SBATCH --mem=8G
#SBATCH --time=00:02:00

# Activate conda environment
eval "$(conda shell.bash hook)"
# Change conda environment name, if necessary
conda activate cs336_systems

# Get a unique port for this job based on the job ID
export MASTER_PORT=$(expr 10000 + $(echo -n $Slurm_JOBID | tail -c 4))
```

[5] If you don't have access to a Slurm cluster, you may find the `pdsh` parallel shell to be useful for issuing commands to multiple machines at once. Alternatively, opening multiple SSH sessions and manually running commands is often sufficient for small-scale experimentation.

```
export MASTER_ADDR=$(scontrol show hostnames "$Slurm_JOB_NODELIST" | head -n 1)
echo "MASTER_PORT: ${MASTER_PORT}"
echo "MASTER_ADDR: ${MASTER_ADDR}"

 # Execute command for each task
 srun python multinode_distributed_hello_world.py
```

In our sbatch job script above, we execute the Python script `multinode_distributed_hello_world.py`, which we show below. This script initializes a process group, constructs a data tensor filled with 1, and then uses the all-reduce operation to sum the data tensors across all ranks. After the all-reduce, each rank holds the summed tensors across all ranks (in this case, a tensor filled with 8, since our world size is 8). For demonstration purposes, each rank also logs the world size, its rank, the local world size, and its local rank—these values are set via environment variables created by Slurm for each task.

```
import os
from datetime import timedelta

import torch
import torch.distributed as dist

def setup():
    # These variables are set via srun
    rank = int(os.environ["Slurm_PROCID"])
    local_rank = int(os.environ["Slurm_LOCALID"])
    world_size = int(os.environ["Slurm_NTASKS"])
    local_world_size = int(os.environ["Slurm_NTASKS_PER_NODE"])
    # MASTER_ADDR and MASTER_PORT should have been set in our sbatch script,
    # so we make sure that's the case.
    assert os.environ["MASTER_ADDR"]
    assert os.environ["MASTER_PORT"]
    # Default timeout is 30 minutes. Reducing the timeout here, so the job fails quicker if there's
    # a communication problem between nodes.
    timeout = timedelta(seconds=60)
    dist.init_process_group("gloo", rank=rank, world_size=world_size, timeout=timeout)
    return rank, world_size, local_rank, local_world_size

def multinode_distributed_demo():
    rank, world_size, local_rank, local_world_size = setup()
    print(
        f"World size: {world_size}, global rank: {rank}, "
        f"local rank: {local_rank}, local world size: {local_world_size}"
    )
    data = torch.ones(5)
    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__":
    multinode_distributed_demo()
```

You might notice that we're not using `mp.spawn` to create processes in this script, since the Slurm `srun` command handles running our script `world_size` times. When we run the script and check the Slurm job

logs, we see each rank reporting its relevant attributes, and that the all-reduce was successful between the two nodes:

```
$ sbatch --partition=batch-cpu multinode_distributed_hello_world.sbatch
Submitted batch job 1915
$ cat Slurm-1915.out
MASTER_PORT: 11915
MASTER_ADDR: <hostname redacted>-01
World size: 8, global rank: 0, local rank: 0, local world size: 4
rank 0 data (before all-reduce): tensor([1., 1., 1., 1., 1.])
rank 0 data (after all-reduce): tensor([8., 8., 8., 8., 8.])
World size: 8, global rank: 1, local rank: 0, local world size: 4
rank 1 data (before all-reduce): tensor([1., 1., 1., 1., 1.])
rank 1 data (after all-reduce): tensor([8., 8., 8., 8., 8.])
World size: 8, global rank: 6, local rank: 3, local world size: 4
rank 6 data (before all-reduce): tensor([1., 1., 1., 1., 1.])
rank 6 data (after all-reduce): tensor([8., 8., 8., 8., 8.])
World size: 8, global rank: 4, local rank: 2, local world size: 4
rank 4 data (before all-reduce): tensor([1., 1., 1., 1., 1.])
rank 4 data (after all-reduce): tensor([8., 8., 8., 8., 8.])
World size: 8, global rank: 2, local rank: 1, local world size: 4
rank 2 data (before all-reduce): tensor([1., 1., 1., 1., 1.])
rank 2 data (after all-reduce): tensor([8., 8., 8., 8., 8.])
World size: 8, global rank: 5, local rank: 2, local world size: 4
rank 5 data (before all-reduce): tensor([1., 1., 1., 1., 1.])
rank 5 data (after all-reduce): tensor([8., 8., 8., 8., 8.])
World size: 8, global rank: 7, local rank: 3, local world size: 4
rank 7 data (before all-reduce): tensor([1., 1., 1., 1., 1.])
rank 7 data (after all-reduce): tensor([8., 8., 8., 8., 8.])
World size: 8, global rank: 3, local rank: 1, local world size: 4
rank 3 data (before all-reduce): tensor([1., 1., 1., 1., 1.])
rank 3 data (after all-reduce): tensor([8., 8., 8., 8., 8.])
```

When running multi-node multi-GPU Slurm jobs, use the `--gpus-per-node=N` sbatch flag to control the number of GPUs allocated to each node (this should match `--ntasks-per-node`). This will expose `N` GPUs to each task—from there, make sure to properly set the CUDA devices (e.g., by explicitly using the device string `f"cuda:{local_rank}"` or with `torch.cuda.set_device(local_rank)`) to ensure that each rank is using a separate GPU. In particular, do *not* use the `--gpus-per-task` flag, since Slurm will isolate each task to a single GPU. This will decrease NCCL performance, since it needs to see all the GPUs on a node to make use of NVLink connections between GPUs.

Problem (`distributed_communication_multi_node`): 3 points

Benchmark the runtime of the all-reduce operation in the multi-node multi-process setting. In particular, vary the following settings:

Number of nodes: 1 or 2. Your results from problem `distributed_communication_single_node` should cover the single-node setting.

Backend + device type: Gloo + CPU, Gloo + GPU, NCCL + GPU.

Size of data on each device: float32 data tensors ranging 512KB to 1GB.

Number of processes per node: For the single-node setting, use 2, 4 or 6 processes (matching problem `distributed_communication_single_node`). For measuring multi-node latency, use 1, 2, or 4 processes per node.

Deliverable: Plot(s) comparing the various settings, with 2-4 sentences of commentary about your results and thoughts about how the various factors interact. In particular, be sure to compare the single- and multi-node settings.

3.3 A naïve implementation of distributed data parallel training

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

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

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

1. Given a batch with n examples, the batch is sharded and each device receives n/d disjoint examples (where d is the number of devices used for data parallel training). n should divide d, since the training time is bottlenecked by the slowest process.
2. Each device uses its local copy of the model parameters to run a forward pass on its n/d examples and a backward pass to calculate the gradients. Note that at this point, each device holds the gradients computed from the n/d examples it received.
3. We then use the all-reduce collective communication operation to average the gradients across the different devices, so each device holds the gradients averaged across all n examples.
4. Next, each device runs an optimizer step to update its copy of the parameters—from the optimizer's perspective, it is simply optimizing a local model. The parameters and optimizer states will stay in sync on all of the different devices since they all start from the same initial model and optimizer state and use the same averaged gradients for each iteration. At this point, we've completed a single training iteration and can repeat the process.

Problem (`naive_ddp`): 3 points

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

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

Problem (`naive_ddp_benchmarking`): 3 points

In this naïve DDP implementation, parameters are individually all-reduced across ranks after each backward pass. To better understand the overhead of data parallel training, create a script to benchmark your previously-implemented language model when trained with this naïve implementation of

DDP. Measure the total time per training step and the proportion of time spent on communicating gradients. Collect measurements in both the single-node setting (1 node x 2 GPUs) and the multi-node setting (2 nodes x 1 GPU). Run your benchmark on GPUs with each of the model sizes described in §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.

3.4 Improving upon the minimal DDP implementation

The minimal DDP implementation that we saw in §3.3 has a couple of key limitations:

1. It conducts a separate all-reduce operation for every parameter tensor. The miniature language model that we worked with in problem `minimal_ddp_benchmarking` has 27 parameter tensors, so we'll need 27 all-reduce calls. 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.

3.4.1 Reducing the number of communication calls

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

Problem (`minimal_ddp_flat_benchmarking`): 1 point

Modify your minimal DDP implementation to communicate a tensor with flattened gradients from all parameters.[a] Compare its performance with the minimal DDP implementation that issues an all-reduce for each parameter tensor under the previously-used conditions (model sizes described in §2.1.2 on 1 node x 2 GPUs and 2 nodes x 1 GPU).

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.

[a] It might be helpful to use `torch._utils._flatten_dense_tensors` and `torch._utils._unflatten_dense_tensors`.

3.4.2 Overlapping computation with communication of individual parameter gradients

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

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

Backward hooks: To automatically call a function on a parameter after its gradient has been accumulated in the backward pass, you can use the `register_post_accumulate_grad_hook` function.[6]

Asynchronous communcation: 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 completed. 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 completed. To wait for the operation to finish, you can call `handle.wait()` on the returned communcation 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 after each call.
for tensor in tensor:
    dist.all_reduce(tensor, async_op=False)

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

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

# Ensure that all-reduce calls finished.
for handle in handles:
    handle.wait()
handles.clear()
```

Problem (`ddp_overlap_individual_parameters`): 10 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.

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

`def finish_gradient_synchronization(self):` When called, wait for asynchronous communication calls to complete.

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 all of the gradient communication has finished:

```
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_individual_parameters]` and `[adapters.ddp_individual_parameters_on_after_backward]` (the latter is optional, depending on your implementation you may not need it).

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

Problem (`ddp_overlap_individual_parameters_benchmarking`): 3 points

(a) Benchmark the performance of your DDP implementation when overlapping backward pass computation with communication of individual parameter gradients. Compare its performance on each of the model sizes described in §2.1.2 with our previously-studied settings (the minimal DDP implementation that either issues an all-reduce for each parameter tensor or a single all-reduce for the concatenation of all parameter tensors).

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.

(b) Instrument your benchmarking code with the PyTorch profiler and export a chrome trace when benchmarking your DDP implementation on an `XL`-sized model, with and without overlapping computation with communication. Visually compare the two traces, and provide a profiler screenshot demonstrating that one implementation overlaps compute with communication while the other doesn't.

Deliverable: 2 screenshots (one from the DDP implementation that overlaps compute with communication, and another that doesn't) that visually show that communication is or isn't overlapped with the backward pass.

(c) A common way of quantifying the communcation-computation overlap is to divide the time spent in computation while communication by the total amount of time spent communicating. Thus,

an overlap of 1.0 indicates that all communication happens while other compute is happening, while an overlap of 0.0 indicates that they are completely disjoint.

Use the Holistic Trace Analysis library to measure degree of communication-computation overlap for each model.[a] As a sanity check, ensure that you get the expected overlap of 0.0 when analyzing the naive DDP implementation.

Deliverable: 1-2 sentence response with the measured communication computation overlap ratio.

[a] See hta.readthedocs.io/en/latest/source/features/comm_comp_overlap.html for an example. If HTA wasn't automatically installed when you set up the assignment repo, install it with `pip install git+https://github.com/nelson-liu/HolisticTraceAnalysis.git@comm_kernel_fix` (this fork contains a bug-fix for recent versions of NCCL)

3.4.3 Overlapping computation with communication of bucketed parameter gradients

In the previous section (§3.4.2), we overlapped backprop computation with the communication of individual parameter gradients. However, we previously observed that batching communication calls can improve performance, especially when we have many parameter tensors (as is typical in deep Transformer models). Our previous attempt at batching sent all of the gradients at once, which requires waiting for the backward pass to finish. In this section, we'll try to get the best of both worlds by organizing our parameters into buckets (reducing the number of total communication calls) and all-reducing buckets when each of their constituent tensors are ready (enabling us to overlap communication with computation).

Problem (`ddp_overlap_bucketed`): 10 points

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

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

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

`def finish_gradient_synchronization(self):` When called, wait for asynchronous communication calls to complete.

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

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

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

Problem (`ddp_bucketed_benchmarking`): 3 points

(a) Vary the maximum bucket size (5, 10, 50, 100, 500MB) and compare the results with individually communicating parameter gradients. Experiment with each of the model sizes described in §2.1.2 on GPUs with both the NCCL and Gloo backends. Comment on your results—do they align with your expectations? If they don't align, why not? You may have to use the PyTorch profiler as necessary to better understand how communication calls are ordered and/or executed. What changes in the experimental setup would you expect would yield results that are aligned with your expectations?

Deliverable: Measured time per training iteration for various bucket sizes on each backend. 3-4 sentence commentary about the results, your expectations, and potential reasons for any mismatch.

(b) Write an equation that models the overhead of DDP as a function of the total size (bytes) of the model parameters (s), the all-reduce algorithm bandwidth (w, computed as the size of each rank's data divided by the time it takes to finish the all-reduce), the overhead (seconds) associated with each communication call (o), and the number of buckets (n_b). From this equation, write an equation for the optimal bucket size that minimizes DDP overhead.

Deliverable: Equation that models DDP overhead, and an equation for the optimal bucket size.

4 Optimizer State Sharding

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

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

Problem (`optimizer_state_sharding`): 30 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 differ-

ent parts of the model); these parameters will be sharded across all 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 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 `pytest tests/test_sharded_optimizer.py`. We recommend running the tests multiple times (e.g., 5) to ensure that it passes reliably.

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

Problem (`optimizer_state_sharding_accounting`): 5 points

(a) Create a script to profile the peak memory usage when training language models with and without optimizer state sharding (use the same model hyperparameters as `naive_ddp_benchmarking`). 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.). Perform your analysis on an `XL`-shaped model from §2.1.2.

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 each model size listed in §2.1.2, using both 1 node x 2 GPUs and 2 nodes x 1 GPU. What proportion of a training iteration is spent communicating parameter updates (i.e., the communication overhead of optimizer state sharding)?

Deliverable: 2-3 sentence response with your timings.

(c) How does our approach to optimizer state sharding differ from ZeRO stage 1 (described as ZeRO-DP P_{os} in Rajbhandari et al., 2020)?

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

5 Epilogue

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

References

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