**Note**

Go to the end to download the full example code.

# Layer Normalization

In this tutorial, you will write a high-performance layer normalization kernel that runs faster than the PyTorch implementation.

In doing so, you will learn about:

- Implementing backward pass in Triton.
- Implementing parallel reduction in Triton.

## Motivations

The *LayerNorm* operator was first introduced in [BA2016] as a way to improve the performance of sequential models (e.g., Transformers) or neural networks with small batch size. It takes a vector $x$ as input and produces a vector $y$ of the same shape as output. The normalization is performed by subtracting the mean and dividing by the standard deviation of $x$. After the normalization, a learnable linear transformation with weights $w$ and biases $b$ is applied. The forward pass can be expressed as follows:

$$y = \frac{x - \text{E}[x]}{\sqrt{\text{Var}(x) + \epsilon}} * w + b$$

where $\epsilon$ is a small constant added to the denominator for numerical stability. Let's first take a look at the forward pass implementation.

```python
import torch

import triton
import triton.language as tl

try:
    # This is https://github.com/NVIDIA/apex, NOT the apex on PyPi, so
    # should not be added to extras_require in setup.py.
    import apex
    HAS_APEX = True
except ModuleNotFoundError:
    HAS_APEX = False

DEVICE = triton.runtime.driver.active.get_active_torch_device()


@triton.jit
def _layer_norm_fwd_fused(
    X,  # pointer to the input
    Y,  # pointer to the output
    W,  # pointer to the weights
    B,  # pointer to the biases
    Mean,  # pointer to the mean
    Rstd,  # pointer to the 1/std
    stride,  # how much to increase the pointer when moving by 1 row
    N,  # number of columns in X
    eps,  # epsilon to avoid division by zero
    BLOCK_SIZE: tl.constexpr,
):
    # Map the program id to the row of X and Y it should compute.
    row = tl.program_id(0)
    Y += row * stride
    X += row * stride
    # Compute mean
    mean = 0
    _mean = tl.zeros([BLOCK_SIZE], dtype=tl.float32)
    for off in range(0, N, BLOCK_SIZE):
        cols = off + tl.arange(0, BLOCK_SIZE)
        a = tl.load(X + cols, mask=cols < N, other=0.).to(tl.float32)
        _mean += a
    mean = tl.sum(_mean, axis=0) / N
    # Compute variance
    _var = tl.zeros([BLOCK_SIZE], dtype=tl.float32)
    for off in range(0, N, BLOCK_SIZE):
        cols = off + tl.arange(0, BLOCK_SIZE)
        x = tl.load(X + cols, mask=cols < N, other=0.).to(tl.float32)
        x = tl.where(cols < N, x - mean, 0.)
        _var += x * x
    var = tl.sum(_var, axis=0) / N
    rstd = 1 / tl.sqrt(var + eps)
    # Write mean / rstd
    tl.store(Mean + row, mean)
    tl.store(Rstd + row, rstd)
    # Normalize and apply linear transformation
    for off in range(0, N, BLOCK_SIZE):
        cols = off + tl.arange(0, BLOCK_SIZE)
        mask = cols < N
        w = tl.load(W + cols, mask=mask)
        b = tl.load(B + cols, mask=mask)
        x = tl.load(X + cols, mask=mask, other=0.).to(tl.float32)
        x_hat = (x - mean) * rstd
        y = x_hat * w + b
        # Write output
        tl.store(Y + cols, y, mask=mask)
```

## Backward pass

The backward pass for the layer normalization operator is a bit more involved than the forward pass. Let $\hat{x}$ be the normalized inputs $\frac{x-\text{E}[x]}{\sqrt{\text{Var}(x)+\epsilon}}$ before the linear transformation, the Vector-Jacobian Products (VJP) $\nabla_x$ of $x$ are given by:

$$\nabla_x = \frac{1}{\sigma}\Big(\nabla_y \odot w - \underbrace{\big(\frac{1}{N}\hat{x} \cdot (\nabla_y \odot w)\big)}_{c_1} \odot \hat{x} - \underbrace{\frac{1}{N}\nabla_y \cdot w}_{c_2}\Big)$$

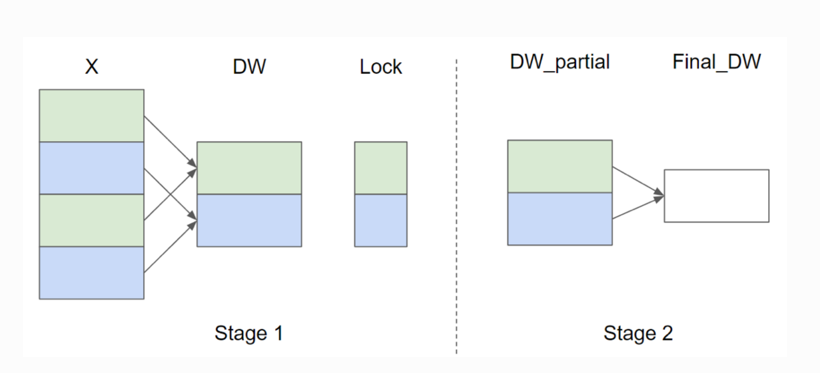
where $\odot$ denotes the element-wise multiplication, $\cdot$ denotes the dot product, and $\sigma$ is the standard deviation. $c_1$ and $c_2$ are intermediate constants that improve the readability of the following implementation.

For the weights $w$ and biases $b$, the VJPs $\nabla_w$ and $\nabla_b$ are more straightforward:

$$\nabla_w = \nabla_y \odot \hat{x} \quad \text{and} \quad \nabla_b = \nabla_y$$

Since the same weights $w$ and biases $b$ are used for all rows in the same batch, their gradients need to sum up. To perform this step efficiently, we use a parallel reduction strategy: each kernel instance accumulates partial $\nabla_w$ and $\nabla_b$ across certain rows into one of $\text{GROUP\_SIZE\_M}$ independent buffers. These buffers stay in the L2 cache and then are further reduced by another function to compute the actual $\nabla_w$ and $\nabla_b$.

Let the number of input rows $M = 4$ and $\text{GROUP\_SIZE\_M} = 2$, here's a diagram of the parallel reduction strategy for $\nabla_w$ ($\nabla_b$ is omitted for brevity):



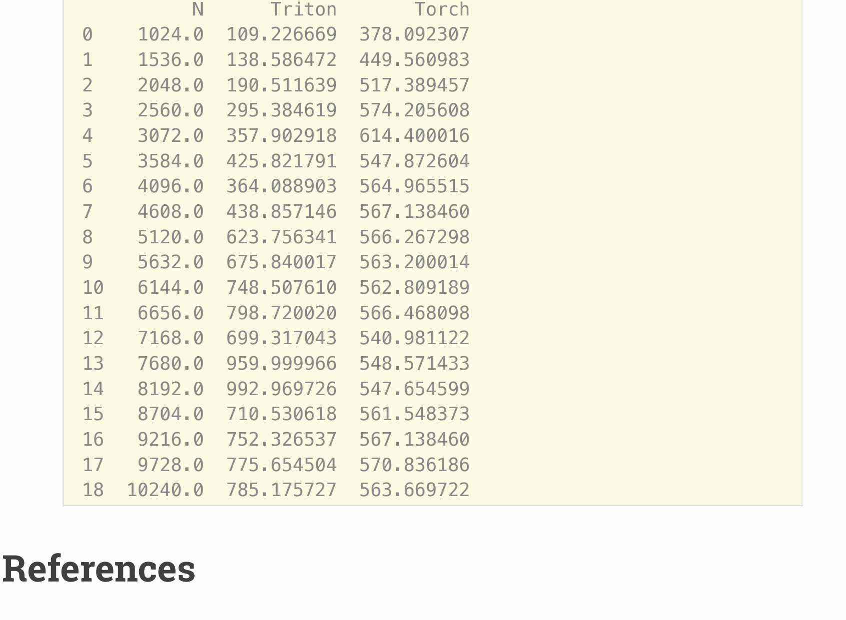

In Stage 1, the rows of X that have the same color share the same buffer and thus a lock is used to ensure that only one kernel instance writes to the buffer at a time. In Stage 2, the buffers are further reduced to compute the final $\nabla_w$ and $\nabla_b$. In the following implementation, Stage 1 is implemented by the function `_layer_norm_bwd_dx_fused` and Stage 2 is implemented by the function `_layer_norm_bwd_dwdb`.

```python
@triton.jit
def _layer_norm_bwd_dx_fused(DX,  # pointer to the input gradient
                             DY,  # pointer to the output gradient
                             DW,  # pointer to the partial sum of weig
                             DB,  # pointer to the partial sum of biase
                             X,  # pointer to the input
                             W,  # pointer to the weights
                             Mean,  # pointer to the mean
                             Rstd,  # pointer to the 1/std
                             Lock,  # pointer to the lock
                             stride,  # how much to increase the pointe
                             N,  # number of columns in X
                             GROUP_SIZE_M: tl.constexpr, BLOCK_SIZE_N:
    # Map the program id to the elements of X, DX, and DY it should co
    row = tl.program_id(0)
    cols = tl.arange(0, BLOCK_SIZE_N)
    mask = cols < N
    X += row * stride
    DY += row * stride
    DX += row * stride
    # Offset locks and weights/biases gradient pointer for parallel re
    lock_id = row % GROUP_SIZE_M
    Lock += lock_id
    Count = Lock + GROUP_SIZE_M
    DW = DW + lock_id * N + cols
    DB = DB + lock_id * N + cols
    # Load data to SRAM
    x = tl.load(X + cols, mask=mask, other=0).to(tl.float32)
    dy = tl.load(DY + cols, mask=mask, other=0).to(tl.float32)
    w = tl.load(W + cols, mask=mask).to(tl.float32)
    mean = tl.load(Mean + row)
    rstd = tl.load(Rstd + row)
    # Compute dx
    xhat = (x - mean) * rstd
    wdy = w * dy
    xhat = tl.where(mask, xhat, 0.)
    wdy = tl.where(mask, wdy, 0.)
    c1 = tl.sum(xhat * wdy, axis=0) / N
    c2 = tl.sum(wdy, axis=0) / N
    dx = (wdy - (xhat * c1 + c2)) * rstd
    # Write dx
    tl.store(DX + cols, dx, mask=mask)
    # Accumulate partial sums for dw/db
    partial_dw = (dy * xhat).to(w.dtype)
    partial_db = (dy).to(w.dtype)
    while tl.atomic_cas(Lock, 0, 1) == 1:
        pass
    count = tl.load(Count)
    # First store doesn't accumulate
    if count == 0:
        tl.atomic_xchg(Count, 1)
    else:
        partial_dw += tl.load(DW, mask=mask)
        partial_db += tl.load(DB, mask=mask)
    tl.store(DW, partial_dw, mask=mask)
    tl.store(DB, partial_db, mask=mask)
    # Release the lock
    tl.atomic_xchg(Lock, 0)


@triton.jit
def _layer_norm_bwd_dwdb(DW,  # pointer to the partial sum of weights g
                         DB,  # pointer to the partial sum of biases gr
                         FINAL_DW,  # pointer to the weights gradient
                         FINAL_DB,  # pointer to the biases gradient
                         M,  # GROUP_SIZE_M
                         N,  # number of columns
                         BLOCK_SIZE_M: tl.constexpr, BLOCK_SIZE_N: tl.c
    # Map the program id to the elements of DW and DB it should compute
    pid = tl.program_id(0)
    cols = pid * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)
    dw = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)
    db = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)
    # Iterate through the rows of DW and DB to sum the partial sums.
    for i in range(0, M, BLOCK_SIZE_M):
        rows = i + tl.arange(0, BLOCK_SIZE_M)
        mask = (rows[:, None] < M) & (cols[None, :] < N)
        offs = rows[:, None] * N + cols[None, :]
        dw += tl.load(DW + offs, mask=mask, other=0.)
        db += tl.load(DB + offs, mask=mask, other=0.)
    # Write the final sum to the output.
    sum_dw = tl.sum(dw, axis=0)
    sum_db = tl.sum(db, axis=0)
    tl.store(FINAL_DW + cols, sum_dw, mask=cols < N)
    tl.store(FINAL_DB + cols, sum_db, mask=cols < N)
```

## Benchmark

We can now compare the performance of our kernel against that of PyTorch. Here we focus on inputs that have Less than 64KB per feature. Specifically, one can set `'mode': 'backward'` to benchmark the backward pass.

```python
class LayerNorm(torch.autograd.Function):

    @staticmethod
    def forward(ctx, x, normalized_shape, weight, bias, eps):
        # allocate output
        y = torch.empty_like(x)
        # reshape input data into 2D tensor
        x_arg = x.reshape(-1, x.shape[-1])
        M, N = x_arg.shape
        mean = torch.empty((M, ), dtype=torch.float32, device=x.device
        rstd = torch.empty((M, ), dtype=torch.float32, device=x.device
        # Less than 64KB per feature: enqueue fused kernel
        MAX_FUSED_SIZE = 65536 // x.element_size()
        BLOCK_SIZE = min(MAX_FUSED_SIZE, triton.next_power_of_2(N))
        if N > BLOCK_SIZE:
            raise RuntimeError("This layer norm doesn't support feature
        # heuristics for number of warps
        num_warps = min(max(BLOCK_SIZE // 256, 1), 8)
        # enqueue kernel
        _layer_norm_fwd_fused[(M, )](  #
            x_arg, y, weight, bias, mean, rstd,  #
            x_arg.stride(0), N, eps,  #
            BLOCK_SIZE=BLOCK_SIZE, num_warps=num_warps, num_ctas=1)
        ctx.save_for_backward(x, weight, bias, mean, rstd)
        ctx.BLOCK_SIZE = BLOCK_SIZE
        ctx.num_warps = num_warps
        ctx.eps = eps
        return y

    @staticmethod
    def backward(ctx, dy):
        x, w, b, m, v = ctx.saved_tensors
        # heuristics for amount of parallel reduction stream for DW/DB
        N = w.shape[0]
        GROUP_SIZE_M = 64
        if N <= 8192: GROUP_SIZE_M = 96
        if N <= 4096: GROUP_SIZE_M = 128
        if N <= 1024: GROUP_SIZE_M = 256
        # allocate output
        locks = torch.zeros(2 * GROUP_SIZE_M, dtype=torch.int32, device
        _dw = torch.zeros((GROUP_SIZE_M, N), dtype=x.dtype, device=w.de
        _db = torch.zeros((GROUP_SIZE_M, N), dtype=x.dtype, device=w.de
        dw = torch.empty((N, ), dtype=w.dtype, device=w.device)
        db = torch.empty((N, ), dtype=w.dtype, device=w.device)
        dx = torch.empty_like(dy)
        # enqueue kernel using forward pass heuristics
        # also compute partial sums for DW and DB
        x_arg = x.reshape(-1, x.shape[-1])
        M, N = x_arg.shape
        _layer_norm_bwd_dx_fused[(M, )](  #
            dx, dy, _dw, _db, x, w, m, v, locks,  #
            x_arg.stride(0), N,  #
            BLOCK_SIZE_N=ctx.BLOCK_SIZE,  #
            GROUP_SIZE_M=GROUP_SIZE_M,  #
            num_warps=ctx.num_warps)
        grid = lambda meta: [triton.cdiv(N, meta['BLOCK_SIZE_N'])]
        # accumulate partial sums in separate kernel
        _layer_norm_bwd_dwdb[grid](
            _dw, _db, dw, db, min(GROUP_SIZE_M, M), N,  #
            BLOCK_SIZE_M=32,  #
            BLOCK_SIZE_N=128, num_ctas=1)
        return dx, None, dw, db, None


layer_norm = LayerNorm.apply


def test_layer_norm(M, N, dtype, eps=1e-5, device=DEVICE):
    # create data
    x_shape = (M, N)
    w_shape = (x_shape[-1], )
    weight = torch.rand(w_shape, dtype=dtype, device=device, requires_g
    bias = torch.rand(w_shape, dtype=dtype, device=device, requires_gra
    x = -2.3 + 0.5 * torch.randn(x_shape, dtype=dtype, device=device)
    dy = .1 * torch.randn_like(x)
    x.requires_grad_(True)
    # forward pass
    y_tri = layer_norm(x, w_shape, weight, bias, eps)
    y_ref = torch.nn.functional.layer_norm(x, w_shape, weight, bias, ep
    # backward pass (triton)
    y_tri.backward(dy, retain_graph=True)
    dx_tri, dw_tri, db_tri = [_.grad.clone() for _ in [x, weight, bias]
    x.grad, weight.grad, bias.grad = None, None, None
    # backward pass (torch)
    y_ref.backward(dy, retain_graph=True)
    dx_ref, dw_ref, db_ref = [_.grad.clone() for _ in [x, weight, bias]
    # compare
    assert torch.allclose(y_tri, y_ref, atol=1e-2, rtol=0)
    assert torch.allclose(dx_tri, dx_ref, atol=1e-2, rtol=0)
    assert torch.allclose(db_tri, db_ref, atol=1e-2, rtol=0)
    assert torch.allclose(dw_tri, dw_ref, atol=1e-2, rtol=0)


@triton.testing.perf_report(
    triton.testing.Benchmark(
        x_names=['N'],
        x_vals=[512 * i for i in range(2, 32)],
        line_arg='provider',
        line_vals=['triton', 'torch'] + (['apex'] if HAS_APEX else []),
        line_names=['Triton', 'Torch'] + (['Apex'] if HAS_APEX else []
        styles=[('blue', '-'), ('green', '-'), ('orange', '-')],
        ylabel='GB/s',
        plot_name='layer-norm-backward',
        args={'M': 4096, 'dtype': torch.float16, 'mode': 'backward'},
    ))
def bench_layer_norm(M, N, dtype, provider, mode='backward', eps=1e-5,
    # create data
    x_shape = (M, N)
    w_shape = (x_shape[-1], )
    weight = torch.rand(w_shape, dtype=dtype, device=device, requires_g
    bias = torch.rand(w_shape, dtype=dtype, device=device, requires_gra
    x = -2.3 + 0.5 * torch.randn(x_shape, dtype=dtype, device=device)
    dy = .1 * torch.randn_like(x)
    x.requires_grad_(True)
    quantiles = [0.5, 0.2, 0.8]

    def y_fwd():

        if provider == "triton":
            return layer_norm(x, w_shape, weight, bias, eps)  # noqa:

        if provider == "torch":
            return torch.nn.functional.layer_norm(x, w_shape, weight,

        if provider == "apex":
            apex_layer_norm = (apex.normalization.FusedLayerNorm(w_sha
            return apex_layer_norm(x)  # noqa: F811, E704

    # forward pass
    if mode == 'forward':
        gbps = lambda ms: 2 * x.numel() * x.element_size() * 1e-9 / (m
        ms, min_ms, max_ms = triton.testing.do_bench(y_fwd, quantiles=
    # backward pass
    if mode == 'backward':
        y = y_fwd()
        gbps = lambda ms: 3 * x.numel() * x.element_size() * 1e-9 / (m
        ms, min_ms, max_ms = triton.testing.do_bench(lambda: y.backwar
                                                     grad_to_none=[x],
    return gbps(ms), gbps(max_ms), gbps(min_ms)


test_layer_norm(1151, 8192, torch.float16)
bench_layer_norm.run(save_path='.', print_data=True)
```




Out:

```
layer-norm-backward:
          N      Triton       Torch
0    1024.0  109.226669  378.092307
1    1536.0  138.586472  449.560983
2    2048.0  190.511639  517.389457
3    2560.0  295.384619  574.205608
4    3072.0  357.902918  614.400016
5    3584.0  425.821791  547.872604
6    4096.0  364.088903  564.965515
7    4608.0  438.857146  567.138460
8    5120.0  623.756341  566.267298
9    5632.0  675.840017  563.200014
10   6144.0  748.507610  562.809189
11   6656.0  798.720020  566.468098
12   7168.0  699.317043  540.981122
13   7680.0  959.999966  548.571433
14   8192.0  992.969726  547.654599
15   8704.0  710.530618  561.548373
16   9216.0  752.326537  567.138460
17   9728.0  775.654504  570.836186
18  10240.0  785.175727  563.669722
```

## References

[BA2016] Jimmy Lei Ba and Jamie Ryan Kiros and Geoffrey E. Hinton, "Layer Normalization", Arxiv 2016

**Total running time of the script:** (0 minutes 29.121 seconds)

Download Jupyter notebook: 05-layer-norm.ipynb

Download Python source code: 05-layer-norm.py

Download zipped: 05-layer-norm.zip