LayerNorm

Layer normalization (LayerNorm) normalizes the outputs of a layer. This helps prevent issues like overflow or underflow, keep the model stable, and ensures smooth training. To keep this blog concise, we will focus only on the forward pass.

In pytorch, the layernorm module is defined as: [torch.nn.LayerNorm](). The operation can be expressed as:

$$y = \frac{x - E[x]}{\sqrt{Var[x] + \epsilon}} * \gamma + \beta$$

Where:

- x = input tensor
- γ = learnable weights
- β = learnable bias
- ϵ = epsilon, small number for numerical stability.

The mean and variance are computed per input sample, across the normalized shape:

$$E[x] = \frac{1}{d}\sum_{i=1}^{d} x_i,\ Var[x] = \frac{1}{d}\sum_{i=1}^{d} (x_i - E[x])^2$$

Where d is the number of features in the normalized shape.

Example:
Consider a $2x3$ tensor:

$$X = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

The normalized shape is the last dimension=3.

Compute the mean:

$$\text{Mean of row 1} = \frac{1+2+3}{3} = 2$$
$$\text{Mean of row 2} = \frac{4+5+6}{3} = 5$$

Compute the variance:

$$\text{variance of row 1} = \frac{(1-2)^2+(2-2)^2+(3-2)^2}{3} = \frac{1+0+1}{3} \approx 0.6667$$

$$\text{variance of row 2} = \frac{(4-5)^2+(5-5)^2+(6-5)^2}{3} = \frac{1+0+1}{3} \approx 0.6667$$

Compute LayerNorm output

Assume γ=1, β=0 and $\epsilon = 10^{-5}$ then:

$$\text{Row 1} = \frac{[1,2,3]-2}{0.6667+10^{-5}} \approx [-1.2247, 0, 1.2247]$$

$$\text{Row 2} = \frac{[4,5,6]-5}{0.6667+10^{-5}} \approx [-1.2247, 0, 1.2247]$$

Resulting LayerNorm output:

$$\begin{pmatrix} -1.2247 & 0 & 1.2247 \\ -1.2247 & 0 & 1.2247 \end{pmatrix}$$

LayerNorm is considered a Memory bound because it involves extensive memory access[1]. In other words, its performance is limited more by reading and writing data than by computation. We load all the input values, store the intermediate values- mean and variance for each sample, and write the normalized results back to memory.

To quantify this, we can measure the effective memory bandwidth in gigabytes per second(GB/s) the formula used is:

$$GB/s = \frac{2x \text{ number of elements } x \text{ size of each element in bytes}}{\text{latency in seconds}} x10^{-6}$$

See the code: here

- The factor 2 is a simplified estimate to account for the main memory traffic, reading the input tensor and writing the output tensor.
- LayerNorm also reads and writes intermediate values like mean and variance, but these are much smaller than the full tensor, so they are usually not included in the formula.
- Multiplying by 10^{-6} - converts bytes per second to gigabytes per second.

Even though this is an approximation, measuring GB/s this way gives a good estimate of memory-bound performance, because the bulk of memory movement comes from the input and output tensors.

Triton (Tutorial Version):

We took the triton LayerNorm tutorial and modified it to:

- Support bfloat16 inputs
- Only test the forward pass
- Add autotuning to find the best configuration for the optimal result.

How it works:
We reshape the input over the normalized shape , collapsing all leading dimensions into rows. In the code:

$$x\_arg = x.reshape(-1, x.shape[-1]).$$

Then, for each row, we compute the mean, variance and normalize the values applying the learnable weight and bias. The kernel processes the row in blocks to improve memory efficiency.

See the code here.

TorchInductor:

Just like with Matmul, we used torch.compile for layernorm- wrapping the pytorch function in a compiled version so we could benchmark it in default and max-autotune modes.

The kernels implementation can be found here.

Kernel LLM

We used our own LayerMorm-specific prompt, with the same flags that we used in matmul. KernelLLM successfully generated Triton kernels, but compilation initially failed. We corrected the same issues as before,including the grid and tensor data type by switching to bfloat16. When we validated it against eager mode for correctness to bfloat16, it failed. To fix it, we converted the x_tile, tmp7 and the t1.sum to float32. That prevents the accumulator overflow by changing it to float16. We generalized the kernels for different shapes, but they are likely to fail for non-power-of-2 shapes.

A significant amount of time was spent trying to optimize LayerNorm kernels, including autotuning and making them general for different shapes. However, these attempts often resulted in segmentation faults, requiring deep investigation into memory handling and kernel internals. Ultimately, we decided to stop pursuing LayerNorm optimization, as the effort outweighed the benefit for this project.

The finalized LayerNorm kernels we used can be found here.

Makora Generate

We reused the same prompt from KernelLLM, again Mako handled prompt validation, kernel generation, compilation and benchmarking automatically, making it very easy to use.

The finalized kernel we used can be found here.

Helion

Helion's "tile-first" programming model also works well for reductions, where you want explicit control over what gets fused into one kernel. The outer hl.tile(m) in our LayerNorm[x] defines a row-wise kernel: for each row tile we compute mean/variance in FP32, then normalize and cast back to the input dtype, all within one compiled kernel body.

Note that TritonBench passes (x, weight, bias, eps) for LayerNorm, but our Helion kernel is non-affine (we ignore weight/bias). That matches the benchmark operator setup where weight=1 and bias=0, so the comparison stays meaningful.