Faster Convolution

Kaibo Tang

July 13, 2024

1 Introduction

Usually adopted tricks to speed up training and/or inference typically fall under one of these categories:

1. Using a different precision, e.g., `tf32` or `float16` instead of the usually used `float32`.
2. Using a different implementation, e.g., writing your own convolution using `triton` or implementing convolution as unfolding, matrix multiplication, and folding, etc.
3. Purchasing a better GPU.

For the sake of content, the scope of this note is limited to the first two categories. But interested and economically competent readers might wish to explore the method mentioned in the third category.

2 Method

In this note, I will focus on offering an alternative implementation of `Conv2d`, which takes a lot of inspiration from here.[1][2] For the sake of simplicity, the implementation will not support depth-wise or dilated convolutions. Although only 2D convolution is demonstrated, the same idea can be easily extended to 1D and 3D convolutions.

[1] github.com/pytorch/pytorch/issues/47990
[2] github.com/f-dangel/unfoldNd

2.1 Intuition

For intuition, imaging the situation where we want to apply a 3×3 kernel to a 3×3 image patch, which is essentially a vector inner product between two vectors in $\mathbb{R}^9$. The same strategy can be easily extend to the case where we apply the kernel across the entire image. To do this, we would extract all 3×3 patches and flatten the spatial dimensions. Then, we perform multiplication between this matrix and the weight vector. Furthermore, the same concept can be extend to the case where we have multiple kernels, except that this time we will be dealing with matrix multiplications.

2.2 Implementation

For preparation, we assume the input image is of shape (B, iC, iH, iW) where B denotes the number of images in the mini-batch, iC denotes the input image channels, and iH, iW denotes the spatial shape of the input image. We assume the weight to be of shape (oC, iC, kH, kW), which maps an input image with iC channels to an output image with oC channels using kernels with spatial shape (kH, kW). We assume an optional bias tensor of shape $(oC,)$, which is broadcast and added to the output image channel-wise. Lastly, we consider stride S and padding P.

With the assumptions above, we expect an output image to be of shape (B, oC, oH, oW), where

$$(oH, oW) = \left(\left\lfloor \frac{iH + 2 \times P - kH}{S} \right\rfloor + 1, \left\lfloor \frac{iW + 2 \times P - kW}{S} \right\rfloor + 1 \right). \quad (1)$$

To "extract patches" from an image, which is formally known as unfolding, one can use either the original implementation by PyTorch, or an accelerated implementation offered here.[3] A comparison of the relative performance of the two methods will be provided in the following section. It is noteworthy that the original implementation by PyTorch only support 2D images but `unfoldNd` support input of arbitrary dimensions.

To perform matrix multiplication, we use Einstein summation notation for the sake of clarity.

A brief outline of the implementation is provided in the following pseudo code.

[3] github.com/f-dangel/unfoldNd

Algorithm 1 Alternative implementation of `Conv2d`

Input: $x : (B, iC, iH, iW)$ ▷ Input tensor
Input: $W : (oC, iC, kH, kW)$ ▷ Convolution kernel
Input: $b : (oC,)$ ▷ Bias tensor
Input: $s : (1,), p : (1,)$ ▷ Stride and padding
Output: $y : (B, oC, oH, oW)$ ▷ Output tensor
$x : (B, iC, iH + 2P, iW + 2P) \leftarrow \text{pad}(x, p)$ ▷ Pad x by p pixels
$\tilde{x} : (B, iC \times kH \times kW, oH \times oW) \leftarrow \text{unfold}(x, (kH, kW), s)$
▷ Extract all patches in x and flatten spatial dimensions
$\tilde{W} : (oC, iC \times kH \times kW) \leftarrow \text{flatten}(W)$
$\tilde{y} : (B, oC, oH \times oW) \leftarrow \text{einsum}(\texttt{'bis,oi->bos'}, \tilde{x}, \tilde{W})$
▷ Apply matmul on $\tilde{x}$ and $\tilde{W}$
$y : (B, oC, oH, oW) \leftarrow \text{fold}(\tilde{y}, (oH, oW), (1, 1), s)$ ▷ Fold $\tilde{y}$
$\tilde{b} : (1, oC, 1, 1) \leftarrow \text{reshape}(b)$ ▷ Reshape b for broadcasting
$y : (B, oC, oH, oW) \leftarrow y + \tilde{b}$ ▷ Apply bias

The actual code is provided below and is consistent with the pseudo code provided above.

```python
def conv2d_fast(input: torch.Tensor, weight: torch.Tensor, bias:
    torch.Tensor | None = None, stride: int = 1, padding: int =
    0, use_torch: bool = True):
    """
    Fast 2D convolution using unfold

    Args:
        input (torch.Tensor): input tensor of shape (B, iC, iH,
            iW)
        weight (torch.Tensor): convolution kernel of shape (oC,
            iC, kH, kW)
        bias (torch.Tensor): bias tensor of shape (oC,).
            Default: None
        stride (int): stride of the convolution kernel. Default:
            1
        padding (int): padding of the input tensor. Default: 0
    """

    fold = F.fold if use_torch else foldNd
    unfold = F.unfold if use_torch else unfoldNd
```

```
    if not input.shape[1] == weight.shape[1]:
        raise ValueError(f'Given input channels
        ↪  {input.shape[1]}, expect weight input channels to be
        ↪  {input.shape[1]}, but got {weight.shape[1]}.')

    B, iC, iH, iW = input.shape
    oC, _, kH, kW = weight.shape
    oH, oW = (iH + 2 * padding - kH) // stride + 1, (iW + 2 *
    ↪  padding - kW) // stride + 1

    # pad input
    input = F.pad(input, (padding, padding, padding, padding)) #
    ↪  (B, iC, iH + 2 x P, iW + 2 x P)
    # unfold input
    input_unf = unfold(input, kernel_size=(kH, kW),
    ↪  stride=stride) # (B, iC x kH x kW, oH x oW)
    # reshape weight
    weight = weight.view(oC, -1) # (oC, iC x kH x kW)
    # matmul using einsum
    output_unf = torch.einsum('bis,oi->bos', input_unf, weight)
    ↪  # (B, oC, oH x oW)
    # fold output
    output = fold(output_unf, output_size=(oH, oW),
    ↪  kernel_size=(1, 1), stride=(1, 1)) # (B, oC, oH, oW)
    # apply bias
    if bias is not None:
        output += bias.view(1, -1, 1, 1)

    return output
```

3 Result

In this section, we thoroughly compare the new implementation to other options for accelerating convolution during inference, i.e., with `torch.no_grad()`.

3.1 Experiment setup

A simple experiment is conducted where the input, weight, and bias are given as following:

```
input = torch.rand(1, 3, 1024, 1024).cuda()
weight = torch.rand(16, 3, 3, 3).cuda()
bias = torch.rand(16).cuda()
```

The weight and bias are applied to the input using `F.conv2d` and using the implemented `conv2d_fast` with and without option `use_torch`. For `F.conv2d`, we investigate the effect of allowing CuDNN to use `tf32` for convolution. For `conv2d_fast`, we investigate the effect of allowing CUDA to use `tf32` for matmul.

The experiment is run on a Linux workstation with a 48GB NVIDIA RTX A6000. PyTorch version is 2.3 with CUDA 12.1. To record CUDA time, `torch.autograd.profiler.profile(use_cuda=True)` is used.

A summary of the experiments conducted and the corresponding results are shown below.

Method	`cudnn`	`matmul`	`use_torch`	Time (ms)
`F.conv2d`				121.819
`F.conv2d`	✓			8.000
`conv2d_fast`	✓		✓	49.807
`conv2d_fast`	✓	✓	✓	24.682
`conv2d_fast`	✓			60.777
`conv2d_fast`	✓	✓		22.265

Table 1: Comparison of performance, where `cudnn` denotes setting `torch.backends.cudnn.allow_tf32 = True`, `matmul` denotes setting `torch.backends.cuda.matmul.allow_tf32 = True`, and `use_torch` denotes the corresponding option in `conv2d_fast`. The outputs of all the methods are consistent as verified by `torch.allclose`.

4 Discussion

When none of the other acceleration options are used, `conv2d_fast` offers more than 2× speed up compared to `F.conv2d`.

When using `F.conv2d`, allowing CuDNN to use `tf32` for convolution offers over 15× speedup, which is faster than `conv2d_fast` with no acceleration. When using the alternative implementation `conv2d_fast` with the native `unfold` functional, allowing CUDA to use `tf32` for matmul offers

$\sim 2\times$ speedup. When using the alternative implementation `conv2d_fast` with the generalized `unfoldNd` functional, allowing CUDA to use `tf32` for matmul offers $\sim 2\times$ speedup.

Interestingly, although `conv2d_fast` with the generalized `unfoldNd` functional is slower than the one with native `unfold` without any acceleration, using `tf32` for matmul makes it faster than the one with native `unfold`.

5 Appendix

Interested readers may find the output of the profiler below.

Name	Self CUDA %	CUDA total
aten::conv2d	0.05%	121.819ms
aten::convolution	0.05%	121.761ms
aten::_convolution	0.09%	121.705ms
aten::cudnn_convolution	82.30%	100.253ms
aten::add_	17.48%	21.294ms
aten::reshape	0.03%	44.000us
aten::view	0.01%	13.000us
cudaEventRecord	0.00%	0.000us
cudaStreamIsCapturing	0.00%	0.000us
cudaMalloc	0.00%	0.000us

Self CPU time total: 121.627ms
Self CUDA time total: 121.819ms

Name	Self CUDA %	CUDA total
aten::conv2d	0.44%	8.000ms
aten::convolution	0.47%	7.965ms
aten::_convolution	1.18%	7.927ms
aten::cudnn_convolution	72.09%	5.767ms
aten::add_	25.18%	2.014ms
aten::reshape	0.46%	52.000us
aten::view	0.19%	15.000us
cudaEventRecord	0.00%	0.000us
cudaStreamIsCapturing	0.00%	0.000us

```
               cudaStreamGetPriority         0.00%       0.000us
-----------------------------------  ------------  ------------
Self CPU time total: 7.872ms
Self CUDA time total: 8.000ms
Relative error: 0.00%
```

```
-----------------------------------  ------------  ------------
                                Name    Self CUDA %    CUDA total
-----------------------------------  ------------  ------------
                        aten::copy_        46.26%      23.042ms
                          aten::pad         0.09%      20.789ms
                aten::constant_pad_nd         0.46%      20.743ms
                         aten::col2im        13.26%      15.760ms
                     aten::contiguous         0.09%       9.017ms
                          aten::clone         0.12%       8.974ms
                         aten::einsum         0.73%       7.774ms
                            aten::bmm        13.77%       6.858ms
                         aten::fill_        11.57%       5.765ms
                         aten::im2col         5.08%       3.462ms
-----------------------------------  ------------  ------------
Self CPU time total: 48.565ms
Self CUDA time total: 49.807ms
Relative error: 0.00%
```

```
-----------------------------------  ------------  ------------
                                Name    Self CUDA %    CUDA total
-----------------------------------  ------------  ------------
                         aten::col2im        26.51%      15.401ms
                          aten::copy_        36.89%       9.105ms
                     aten::contiguous         0.13%       8.808ms
                          aten::clone         0.23%       8.777ms
                         aten::einsum         1.21%       3.476ms
                            aten::bmm        11.16%       2.755ms
                         aten::im2col         8.07%       2.571ms
                          aten::add_         8.10%       2.000ms
                          aten::pad          0.16%       1.213ms
                aten::constant_pad_nd         0.75%       1.174ms
-----------------------------------  ------------  ------------
Self CPU time total: 23.856ms
Self CUDA time total: 24.682ms
Relative error: 0.00%
```

Name	Self CUDA %	CUDA total
aten::conv2d	0.13%	38.500ms
aten::convolution	0.14%	38.421ms
aten::_convolution	0.16%	38.335ms
aten::cudnn_convolution	48.57%	29.517ms
aten::scatter_add_	14.96%	9.113ms
aten::_conv_depthwise2d	14.28%	8.721ms
aten::arange	3.64%	4.403ms
aten::einsum	0.49%	3.471ms
aten::bmm	4.53%	2.752ms
aten::eye	0.70%	2.036ms

Self CPU time total: 57.291ms
Self CUDA time total: 60.777ms
Relative error: 0.00%

Name	Self CUDA %	CUDA total
aten::conv2d	0.31%	7.243ms
aten::convolution	0.31%	7.174ms
aten::_convolution	0.32%	7.105ms
aten::_conv_depthwise2d	30.72%	6.874ms
aten::scatter_add_	19.21%	4.297ms
aten::einsum	1.35%	3.472ms
aten::bmm	12.36%	2.751ms
aten::add_	8.95%	1.992ms
aten::eye	1.74%	1.770ms
aten::fill_	5.97%	1.330ms

Self CPU time total: 19.057ms
Self CUDA time total: 22.265ms
Relative error: 0.00%