Lei Mao

Artificial Intelligence
Machine Learning
Computer Science

Santa Clara, California

CATALOGUE

CUDA Tensor Layouts for Convolution

06-04-2023 06-04-2023 BLOG 13 MINUTES READ (ABOUT 1958

Introduction

There are commonly two layouts for the activation tensors involved in the convolution operations in neural networks, NCHW, NHWC, and NC/xHWx.

In general, the performance of convolution using NHWC is much faster than using NCHW. The NC/xHWx layout is an variant of NHWC that is prepared for NVIDIA Tensor Core operations.

In this blog post, I would like to discuss how to perform convolution on GPU and why NHWC and NC/xHWx activation tensor layouts are much more favored than the NCHW activation tensor layout for convolutional neural network inference.

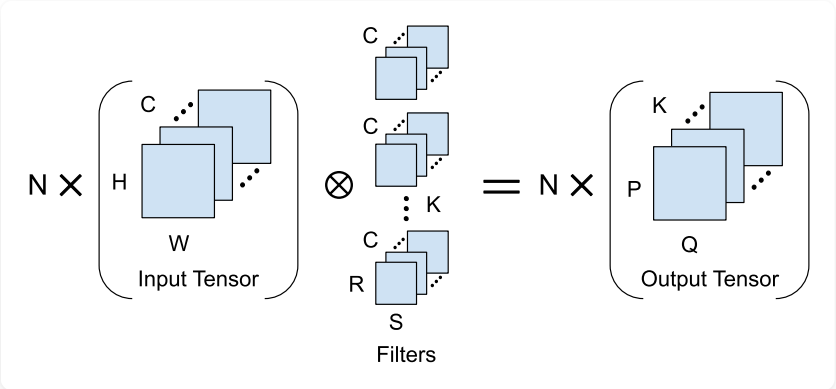
Convolution

The description of convolution in neural networks can be found in the documentation of many deep learning frameworks, such as PyTorch.

Convolution Dimensions

The 2D convolution operation in neural networks consists of an input activation tensor, a filter tensor, an optional bias tensor, and an output activation tensor. We will ignore the bias tensor in this article since it is usually simple to deal with.

The input and output activation tensors and the filter tensor are all 4D tensors that consist of four dimensions. We use N to describe the batch dimension for the input and output tensors. We use C, H, W to describe the number of channels, the spatial height, and the spatial width of the input activation tensor. In order to distinguish the output activation tensor from the input activation tensor, we use K, P, Q to describe the number of channels, the spatial height, and the spatial width of the output activation tensor instead. The filter height and width are described using R and S, respectively.



Convolution Dimension Naming Conventions

Implicit GEMM for Convolution

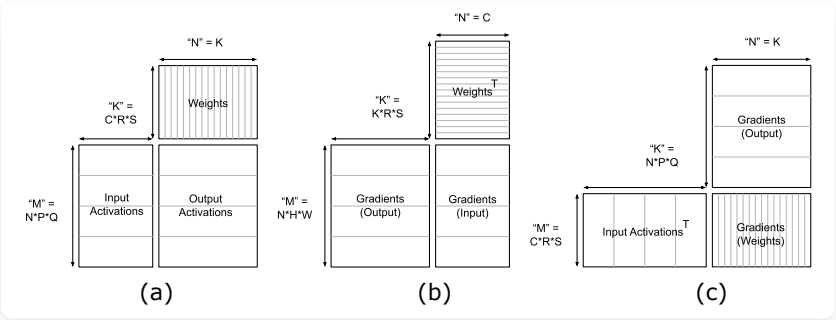
In my previous article "Fast Fourier Transform for Convolution", I described how to perform convolution using the asymptotically faster fast Fourier transform. But this technique is still not the most common way of performing convolution nowadays on GPU and it is out of the scope of this article.

In my previous article "Convolution and Transposed Convolution as Matrix Multiplication", I described how to perform convolution using matrix multiplication in which the activation tensors are dense but the filter tensor is sparse.

On GPUs, convolution is usually performed using a method called implicit GEMM. GEMM stands for general matrix multiplication. The difference between the implicit GEMM method that I am about to describe and the method I

described in the article "Convolution and Transposed Convolution as Matrix Multiplication" is that all the matrices used in the implicit GEMM method are dense matrices.

The implicit GEMM method for convolution can be described using the following figure. We will focus on the forward propagation (a) only as the gradient updates (b and c) usually do not happen in the neural network inference.



The Virtual Matrices Used in the Implicit GEMM Methods for Convolution

Theoretically, if we transpose, expand and reshape the input activation from a 4D tensor of shape (N, C, H, W) to a 2D tensor of shape (NPQ, CRS), transpose and reshape the weight tensor from 4D (K, C, S, R) to 2D (CRS, K), multiply the two tensors, the 2D output tensor is a tensor of shape (NPQ, K) and can be further transposed to the 4D output activation tensor of shape (N, K, P, Q). For example, suppose $N = 1, C = K = 1, H = W = 3, R = S = 2$, $P = Q = 2$ (the convolution stride is 1 and the padding is "valid"). Because there is only one input channel, the only spatial feature in the input activation tensor is a matrix and its values can be assumed to be

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

The reconstructed input activation matrix will be of shape $(4, 4)$ and its values are

$$\begin{bmatrix} 1 & 2 & 4 & 5 \\ 2 & 3 & 5 & 6 \\ 4 & 5 & 7 & 8 \\ 5 & 6 & 8 & 9 \end{bmatrix}$$

The problem of this theoretical formulation is that a new matrices always need to be constructed during inference because the output activation tensor, which will usually be used as the input activation tensor for the next convolution layer, is not of the reconstructed format. Even though the theoretical ration between the number of values in the reconstructed input activation matrix and the number of values in the original input activation tensor is $\frac{PQRS}{HW}$ which sometimes can be 1, constructing such a new matrix is not a no-op and will introduce overhead in computing, not to mention consuming additional memory when this ratio is high. Therefore, in practice, this reconstructed input activation matrix is never constructed in the implicit GEMM method for convolution. The values are read from the input activation tensor of its original layout instead.

NVIDIA Tensor Core

NVIDIA Tensor Core performs small matrix multiplications to accelerate GEMM with extremely high throughput. For example, NVIDIA Tensor Core could perform 16×16×16 GEMM, 16x16 and 16x16 matrix multiplication (and accumulation) for half precision floating point data on a warp basis. Fundamentally, the mathematical motivation of Tensor Core GEMM acceleration has been described in my previous article CUDA Matrix Multiplication, although not explicitly at that time.

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{1,1}^{d \times d} & \mathbf{A}_{1,2}^{d \times d} & \cdots & \mathbf{A}_{1,n/d}^{d \times d} \\ \mathbf{A}_{2,1}^{d \times d} & \mathbf{A}_{2,2}^{d \times d} & \cdots & \mathbf{A}_{2,n/d}^{d \times d} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{A}_{m/d,1}^{d \times d} & \mathbf{A}_{m/d,2}^{d \times d} & \cdots & \mathbf{A}_{m/d,n/d}^{d \times d} \end{bmatrix}$$

$$\mathbf{B} = \begin{bmatrix} \mathbf{B}_{1,1}^{d \times d} & \mathbf{B}_{1,2}^{d \times d} & \cdots & \mathbf{B}_{1,p/d}^{d \times d} \\ \mathbf{B}_{2,1}^{d \times d} & \mathbf{B}_{2,2}^{d \times d} & \cdots & \mathbf{B}_{2,p/d}^{d \times d} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{B}_{n/d,1}^{d \times d} & \mathbf{B}_{n/d,2}^{d \times d} & \cdots & \mathbf{B}_{n/d,p/d}^{d \times d} \end{bmatrix}$$

$$\mathbf{C} = \begin{bmatrix} \mathbf{C}_{1,1}^{d\times d} & \mathbf{C}_{1,2}^{d\times d} & \cdots & \mathbf{C}_{1,p/d}^{d\times d} \\ \mathbf{C}_{2,1}^{d\times d} & \mathbf{C}_{2,2}^{d\times d} & \cdots & \mathbf{C}_{2,p/d}^{d\times d} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{C}_{m/d,1}^{d\times d} & \mathbf{C}_{m/d,2}^{d\times d} & \cdots & \mathbf{C}_{m/d,p/d}^{d\times d} \end{bmatrix}$$

$$\mathbf{C}_{i,j}^{d\times d} = \sum_{k=1}^{n/d} \mathbf{A}_{i,k}^{d\times d} \mathbf{B}_{k,j}^{d\times d}$$

Basically, by decomposing the large matrix multiplication into smaller matrix multiplications and accumulation and caching the small matrices, we could make GEMM extremely math bound. Specifically, the small matrices $\mathbf{A}_{i,k}^{d\times d}$ and $\mathbf{B}_{k,j}^{d\times d}$ are cached in the registers of a warp, and each warp computes a $\mathbf{C}_{i,j}^{d\times d}$ using Tensor Core by iterating the small matrice multiplication and accumulation $\frac{n}{d}$ times.

Tensor Layouts

Now that we have some basic idea of how convolution is performed on GPU via implicit GEMM. Let's check the impact of different activation layouts on the performance of convolution.

NCHW

In the NCHW layout, C is not the fastest dimension. This means, even without assuming the implementation, getting the entire channels from the input activations for implicit GEMM (CRS) needs to stride lots of times, which significantly reduced the valid memory throughput on GPU. For example, suppose the input activation tensor has $N = 1$, $C = 256$, and $H = W = 128$, to get an entire channel from the spatial indices $(12, 35)$ for 1x1 convolution, the slicing operation we will perform for the first sample is $X[1, :, 12, 35]$. Under the hood, getting an entire channel of size $C = 256$ needs to stride $C = 256$ times of size $HW = 16384$ and this invalids the coalesced reading of the data from the DRAM on GPU.

Therefore, the NCHW layout is not favored for the implicit GEMM for convolution.

NHWC

In the NHWC layout, C becomes the fastest dimension. Unlike NCHW slicing for the C dimension, the NHWC slicing for the C dimension can be fully coalesced from the DRAM.

Therefore, the NHWC layout is favored over the NCHW layout for the implicit GEMM for convolution.

NC/xHWx

To take the advantage of NVIDIA Tensor Core, the "virtual" reconstructed input activation matrix needs to be divided in a way that is compatible with the Tensor Core GEMM. This requires the "virtual" reconstructed input activation matrix to be padded (with zeros) so that CSR could be divided by the small matrix dimension requirements from Tensor Core operations. The NHWC layout provides no such guarantee therefore applying the NHWC tensor for Tensor Core GEMM requires padding during the runtime and therefore is a little bit cumbersome to use with Tensor Core.

The NC/xHWx layout is always padded to x elements for the fastest (C) dimension, where x is usually the Tensor Core GEMM dimension requirement. Therefore, it is immediately ready to be used with Tensor Core.

One might ask, is there a NHWC variant layout whose C dimension is not divided like the NC/xHWx layout but padded to x elements according to the Tensor Core GEMM dimension requirement. The answer is yes and using that layout can be very performant for the implicit GEMM method for convolution as well. My educative guess for the reason why NC/xHWx is slightly more often seen than the padded NHWC layout is that the indexing and slicing for the NC/xHWx layout might be more natural in the implementation than that for the padded NHWC layout. For example, using the padded NHWC layout, the indexing and slicing of the input activation tensor would be like this.

$$
\begin{aligned}
&X[1, 0:4, 0:4, 0:16] \\
&X[1, 0:4, 0:4, 16:32] \\
&X[1, 0:4, 0:4, 32:48]
\end{aligned}
$$

Using the NC/16HW16 layout instead, the indexing and slicing of the input activation tensor to get the equivalent matrices would be like this.

$$X[1, 0, 0:4, 0:4, :]$$
$$X[1, 1, 0:4, 0:4, :]$$
$$X[1, 2, 0:4, 0:4, :]$$

References

- NVIDIA Convolutional Layers User's Guide
- TensorRT Tensor Formats
- cuDNN Tensor Layout Formats
- NVIDIA Tensor Core Programming
- Convolution and Transposed Convolution as Matrix Multiplication
- Using Tensor Cores in CUDA Fortran
- CUDA Device Memory Access
- Coalesced Access to Global Memory
- Programming Tensor Cores in CUDA 9
- Simple Tensor Core GEMM

ACCELERATED COMPUTING, CUDA

我小学时的计算机课 PIS 炒股失败回归直播

Comments