CUDA IM2COL Kernels Design

Wayne Anderson

29/03/2024

In multi-channel 2D convolution, IM2COL technique is essential for optimizing the algorithm. IM2COL operator rearranges the original tensor to a matrix, with each row as the element in each convolution area, and each column as the element of the original tensor flatten in height and width dimension.

In mathematical expression, given a source tensor $\boldsymbol{T}$, with width, height and depth of W, H, and D, respectively, on which the convolution process will be processed, the tensor $\boldsymbol{T}$ is shown in equation (1.1).

$$\boldsymbol{T} = \begin{bmatrix} [x_{0,0,0}, x_{0,0,1}, \dots x_{0,0,D-1}] & \cdots & [x_{0,N-1,0}, x_{0,N-1,1}, \dots x_{0,N-1,D-1}] \\ \vdots & \ddots & \vdots \\ [x_{H-1,0,0}, x_{H-1,0,1}, \dots x_{H-1,0,D-1}] & \cdots & [x_{H-1,W-1,0}, x_{H-1,W-1,1}, \dots x_{H-1,W-1,D-1}] \end{bmatrix} \tag{1.1}$$

After the IM2COL process, the tensor will be rearranged as shown in equation (1.2.1 and 1.2.2). Notice that equation 1.2.1 and 1.2.2 only show a part of the whole IM2COL buffer.

$$\begin{bmatrix} x_{0,0,0} & \cdots & x_{0,W-1,0} & x_{1,0,0} & \cdots & x_{H-1,W-1,0} \\ x_{0,0,1} & \cdots & x_{0,W-1,1} & x_{1,0,1} & \cdots & x_{H-1,W-1,1} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_{0,0,D-1} & \cdots & x_{0,W-1,D-1} & x_{1,0,D-1} & \cdots & x_{H-1,W-1,D-1} \end{bmatrix} \tag{1.2.1}$$

$$\begin{bmatrix} x_{0,1,0} & \cdots & x_{0,W-1+\frac{W_K}{2},0} & x_{1,1,0} & \cdots & x_{H-1+\frac{H_K}{2},W-1+\frac{W_K}{2},0} \\ x_{0,0,1} & \cdots & x_{0,W-1+\frac{W_K}{2},1} & x_{1,1,1} & \cdots & x_{H-1+\frac{H_K}{2},W-1+\frac{W_K}{2},1} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_{0,0,D-1} & \cdots & x_{0,W-1+\frac{W_K}{2},D-1} & x_{1,1,D-1} & \cdots & x_{H-1+\frac{H_K}{2},W-1+\frac{W_K}{2},D-1} \end{bmatrix} \tag{1.2.2}$$

Where W_K and H_K are the width and height of the convolution kernel, the height of complete IM2COL buffer should be $W_K * H_K * D$.

For the convolution kernel, $[\boldsymbol{K}]$, which is a tensor array, each tensor of it has the same depth as that of the source tensor. Assuming that the i-th tensor in kernel is represented as $\boldsymbol{K_i}$, and there are N tensors in the kernel:

$$[\boldsymbol{K}] = [\boldsymbol{K}_0, \boldsymbol{K}_1, \dots \boldsymbol{K}_{N-1}] \tag{1.3}$$

The convolution kernel will also be rearranged, as shown in equation (1.4)

$$\begin{bmatrix} \boldsymbol{K}_0 \\ \boldsymbol{K}_1 \\ \vdots \\ \boldsymbol{K}_{N-1} \end{bmatrix} =$$

$$\begin{bmatrix} [\boldsymbol{K_0}(0,0,0), \boldsymbol{K_0}(0,0,1) \dots \boldsymbol{K_0}(0,0,D-1), \dots, \boldsymbol{K_0}(H_K-1, W_K-1, 0), \dots, \boldsymbol{K_0}(H_K-1, W_k-1, D-1)] \\ [\boldsymbol{K_1}(0,0,0), \boldsymbol{K_1}(0,0,1) \dots \boldsymbol{K_1}(0,0,D-1), \dots, \boldsymbol{K_1}(H_K-1, W_K-1, 0), \dots, \boldsymbol{K_1}(H_K-1, W_k-1, D-1)] \\ \vdots \\ [\boldsymbol{K_{N-1}}(0,0,0), \boldsymbol{K_{N-1}}(0,0,1) \dots \boldsymbol{K_{N-1}}(0,0,D-1), \dots, \boldsymbol{K_{N-1}}(H_K-1, W_K-1, 0), \dots, \boldsymbol{K_{N-1}}(H_K-1, W_k-1, D-1)] \end{bmatrix} \tag{1.4}$$

Equation (1.4) shows that each tensor in the kernel is flatten to a row vector. That also shows the implementation technique in DECX. The elements within a tensor in the kernel are frequently accessed during the convolution process for each pixel. Hence, the elements of each tensor is stored adjacently in physical memory, for maximum utilization of CUDA L2 or L1 cache.

In DECX, the design of tensor data structure (logical 3D data structure), of which pitch of depth (dpitch) is aligned to 4. To ensure the coalescing memory access of each warp, which is a group of threads has 32 threads, cases of dpitch = 4, 8, 12, and 16 are implemented by different kernels respectively.

1. The case dpitch = 4 (depth = 0, 1, 2, 3)

The block size is $[128, 2]$, figure 1 shows the address mapping to each thread on each stage when performing an IM2COL operation.

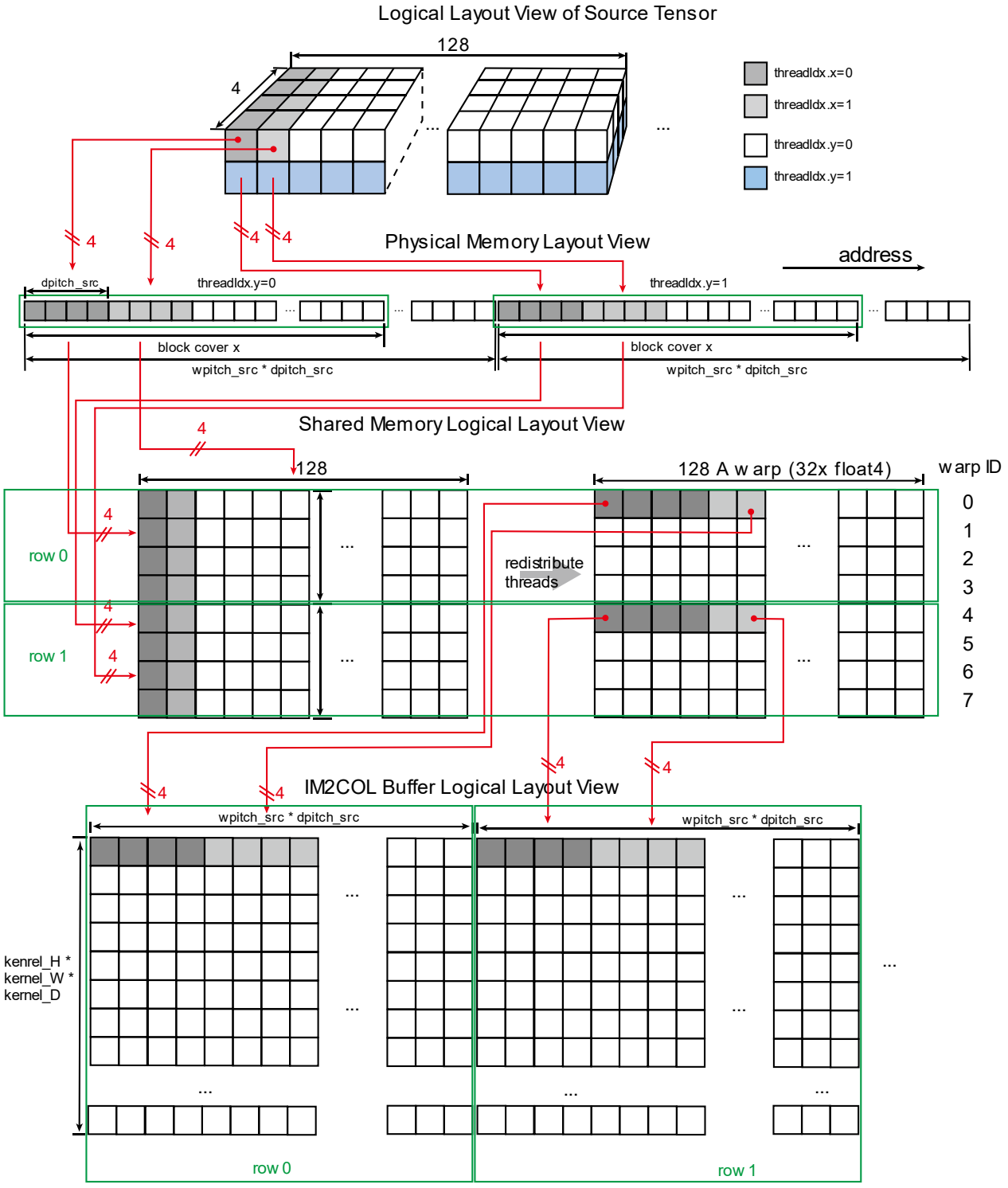

Figure 1 The data flow and thread mapping for dpitch=4 kernel

In this case, each thread loads a float4 from global memory. The thread exactly covers the whole depth dimension of the source tensor, resulting a coalescing memory access. Hundreds of float4 registers are distributed to each thread, storing the result from the loading process.

A shared memory, in type of float, with size of $[8, 128 + 4]$, where 4 is for avoiding bank conflict of shared memory, is

allocated to each block. Each thread then stores the float4 register into shared memory after a transpose. In other words, each thread stores float4::x to the first row of shared memory, then float4::y to the second. Notice that the access pattern is still coalescing in this process, each warp executes $32 \times 4 = 128$ bytes of transaction.

Then the pointer of each row of the shared memory is reinterpreted as float4*. For each row, there are 128 floats need to be stored back to IM2COL buffer. In this case, each row is processed by one warp. Each warp exactly covers 128 floats, which is also in coalescing memory access pattern.

For each thread block, the threads are divided into two groups, each group for each row of the convoluted matrix. The data flow and the address mapping to thread index are described above. Notice that, the shared memory height is mentioned to be 8, which is 4 for each row and 2 rows in total.

2. The case dpitch = 8 (depth = 4, 5, 6, 7)