BLAS和GEMM

BLAS简介

BLAS全称是Basic Linear Algebra Subprograms。它规定了一套低级的执行常见线性代数操作的规范。其实现经常针对特殊的机器进行优化，比较著名的BLAS库有ACML, ATLAS, MKL, OpenBLAS。

许多常见的数值软件均采用兼容BLAS规范的实现库来进行线性代数计算，比如Matlab, Numpy, Mathematica。

BLAS提供用于执行基本矢量和矩阵运算的标准构建块的例程。

- 1级BLAS执行标量，矢量和矢量-矢量运算
- 2级BLAS执行矩阵-矢量运算
- 3级BLAS执行矩阵-矩阵运算 (GEMM)

Level 1

This level consists of all the routines described in the original presentation of BLAS(1979), which defined only vector operations on strided arrays:

dot products, vector norms, a generalized vector addition of the form

点积 向量范数 广义向量加法

$$\mathbf{y} \leftarrow \alpha \mathbf{x} + \mathbf{y}$$

(called "axpy") and several other operations.

Level 2

This level contains *matrix-vector operations* including, among other things, a generalized matrix-vector multiplication (gemv):

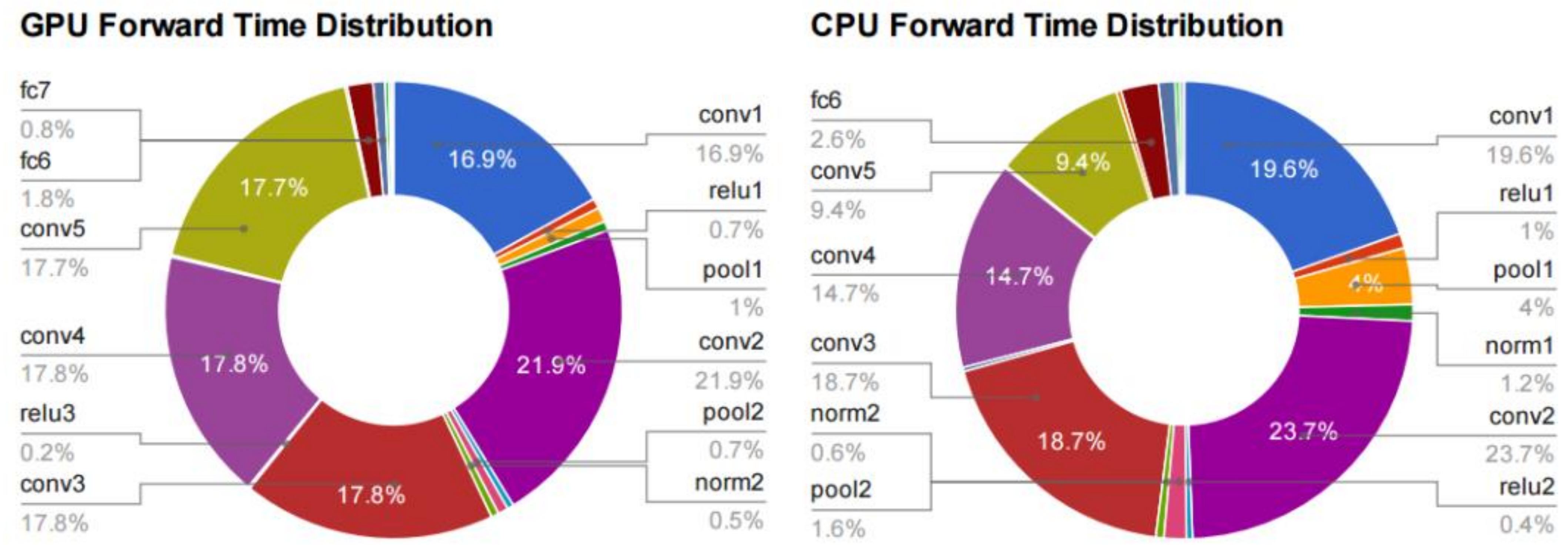
$$\mathbf{y} \leftarrow \alpha \mathbf{A}\mathbf{x} + \beta \mathbf{y}$$

Level 3

This level, formally published in 1990, contains matrix-matrix operations, including a "general matrix multiplication" (gemm), of the form

$$\mathbf{C} \leftarrow \alpha \mathbf{A}\mathbf{B} + \beta \mathbf{C}$$

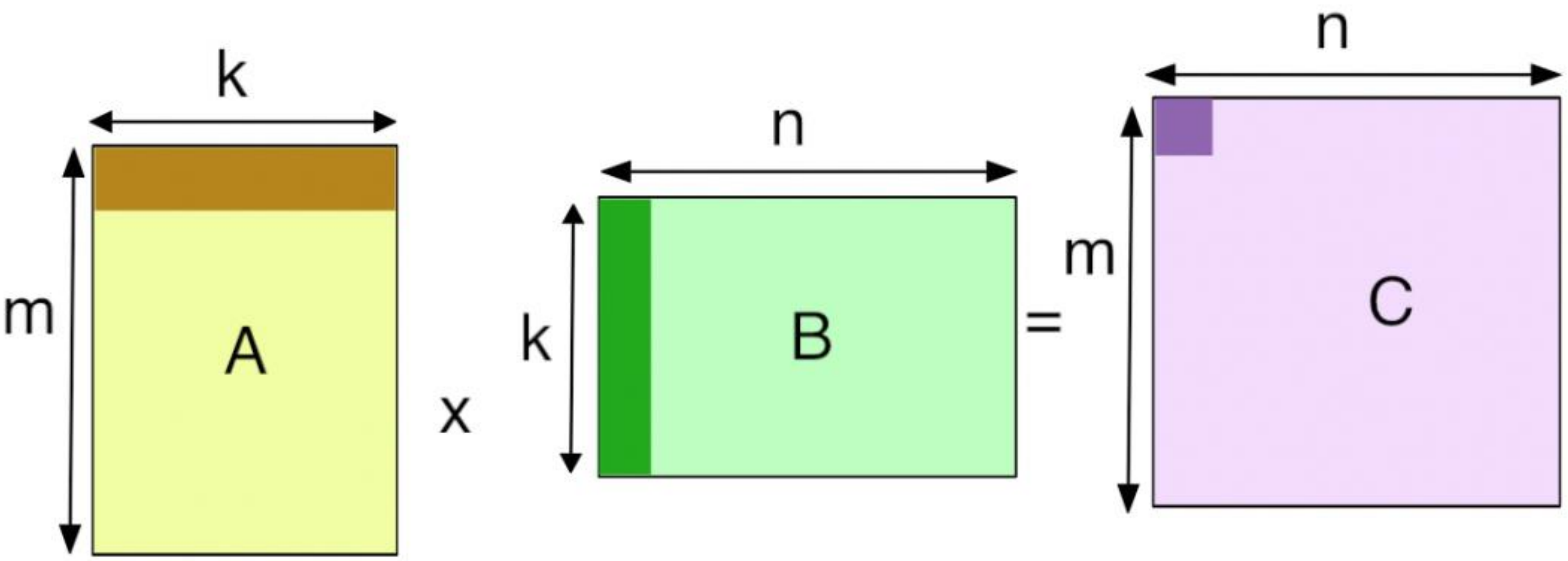
Why GEMM is at the heart of deep learning



使用Alexnet架构进行图像识别的典型深度卷积神经网络的时间划分。

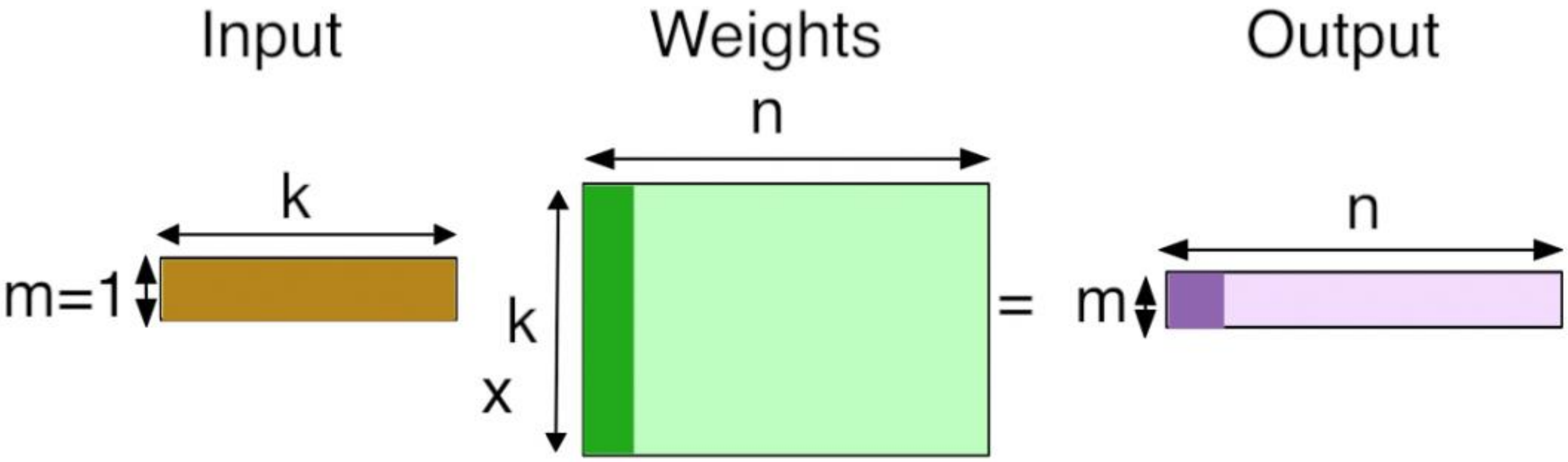
- 所有以fc（用于全连接）或conv（用于卷积）开始的层都是使用GEMM实现。
- 几乎所有时间（95%的GPU和89%的CPU）都花在这些层上。

GEMM (GEneral Matrix to Matrix Multiplication)



例如，网络中的单个层可能需要将256行、1152列矩阵乘以1152行、192列矩阵以产生256行、192列结果。 这需要5700万（256 x 1152 x 192）个浮点运算。

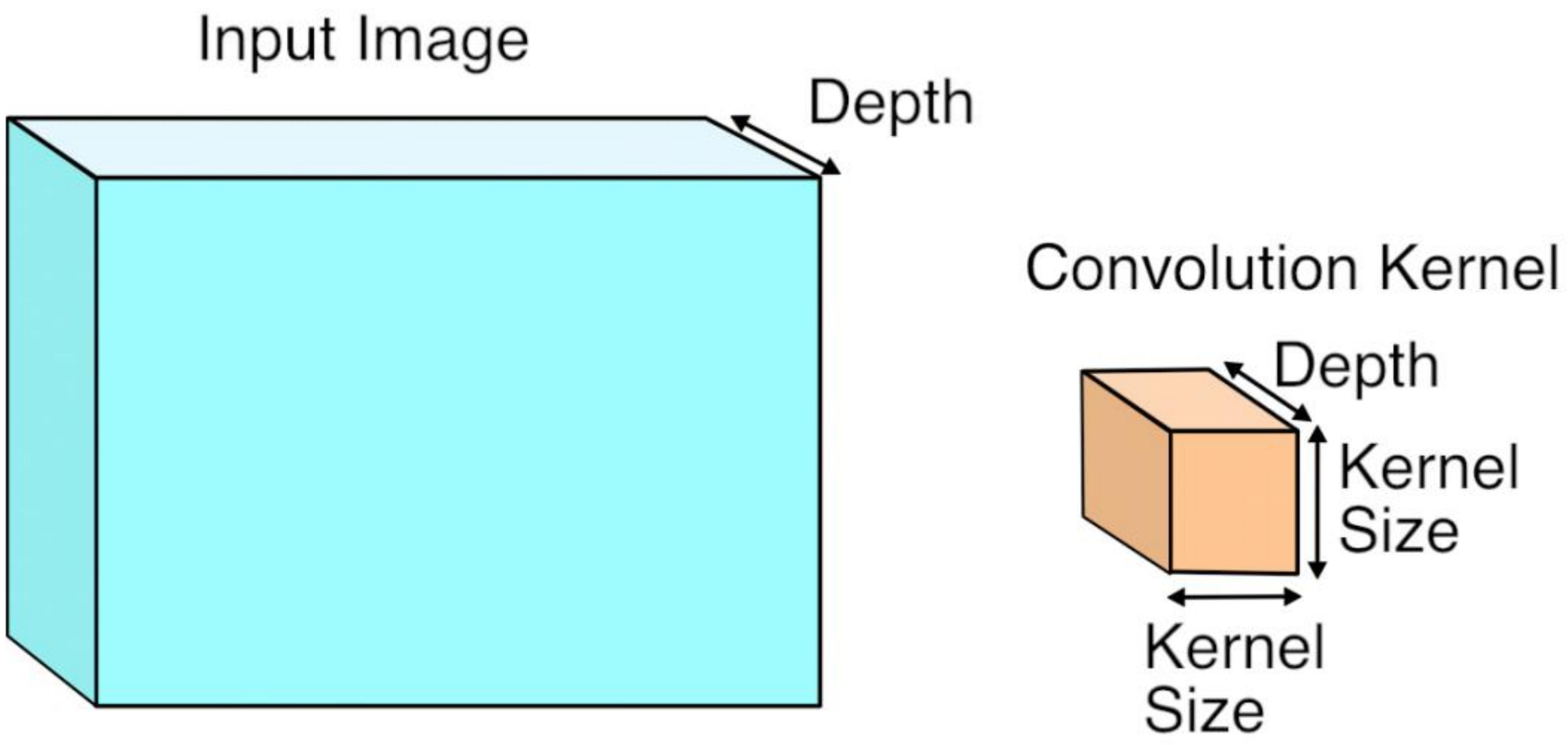
Fully-Connected Layers

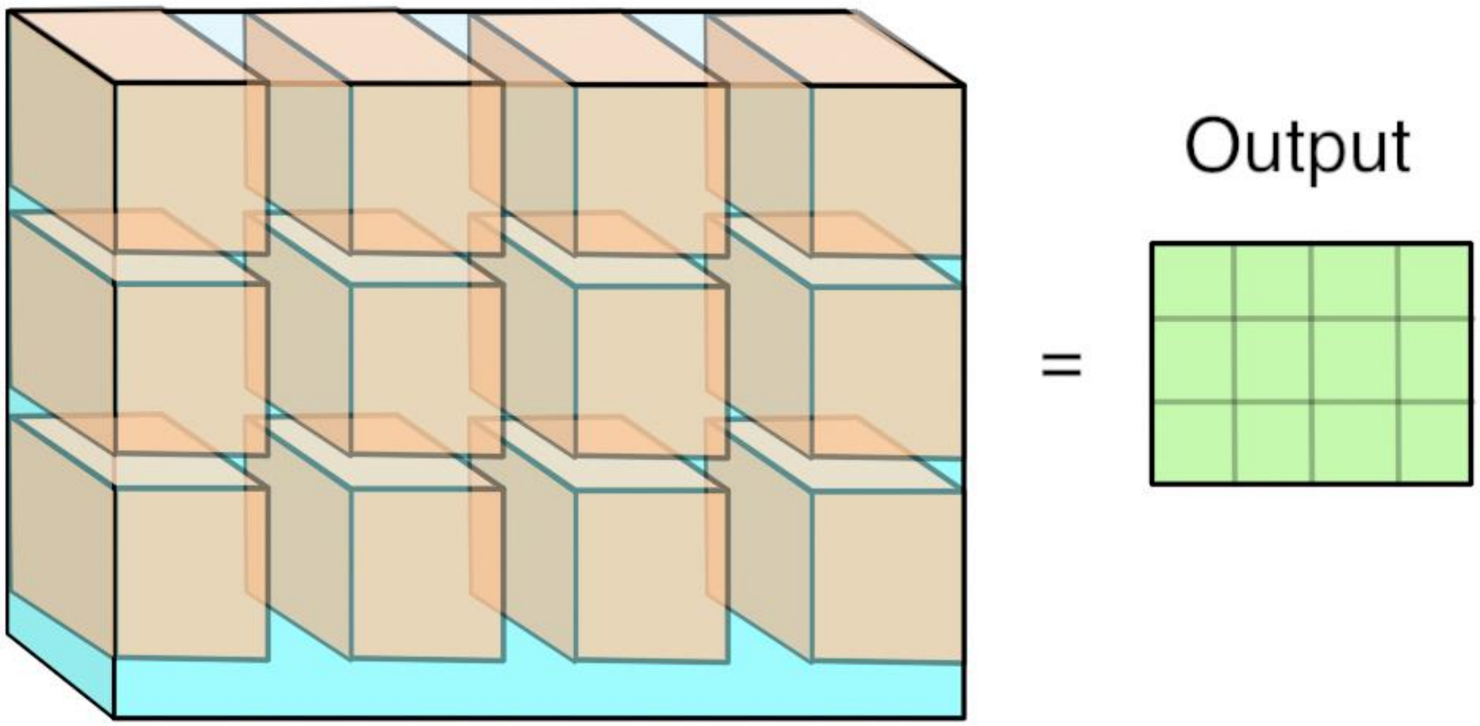


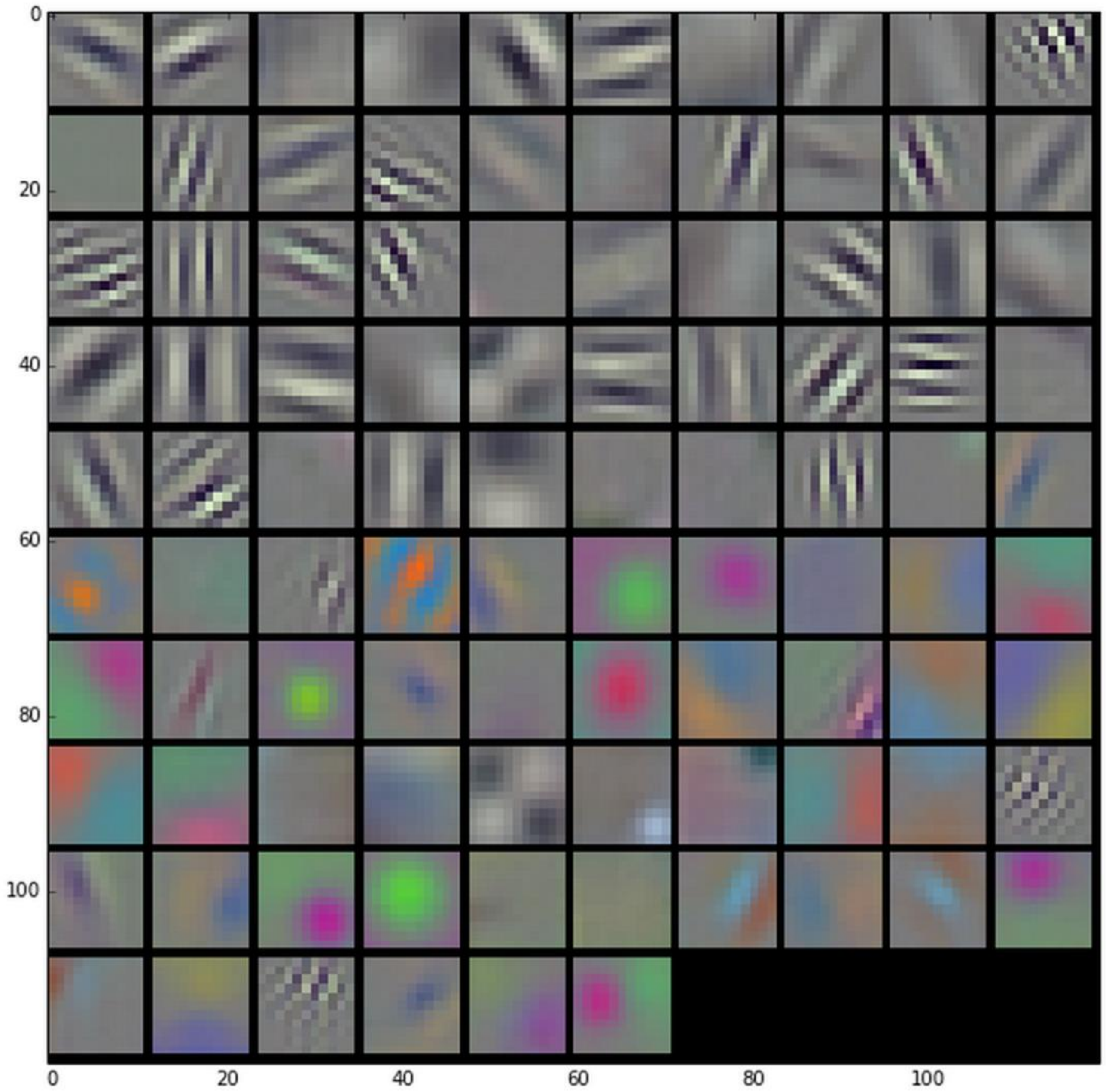
有k个输入值，并且有n个神经元。每个神经元都有自己的每个输入值的学到的权重集。

输出有n个输出值，每个神经元一个，通过计算其权重和输入值的点积来计算。

Convolutional Layers




=
Output

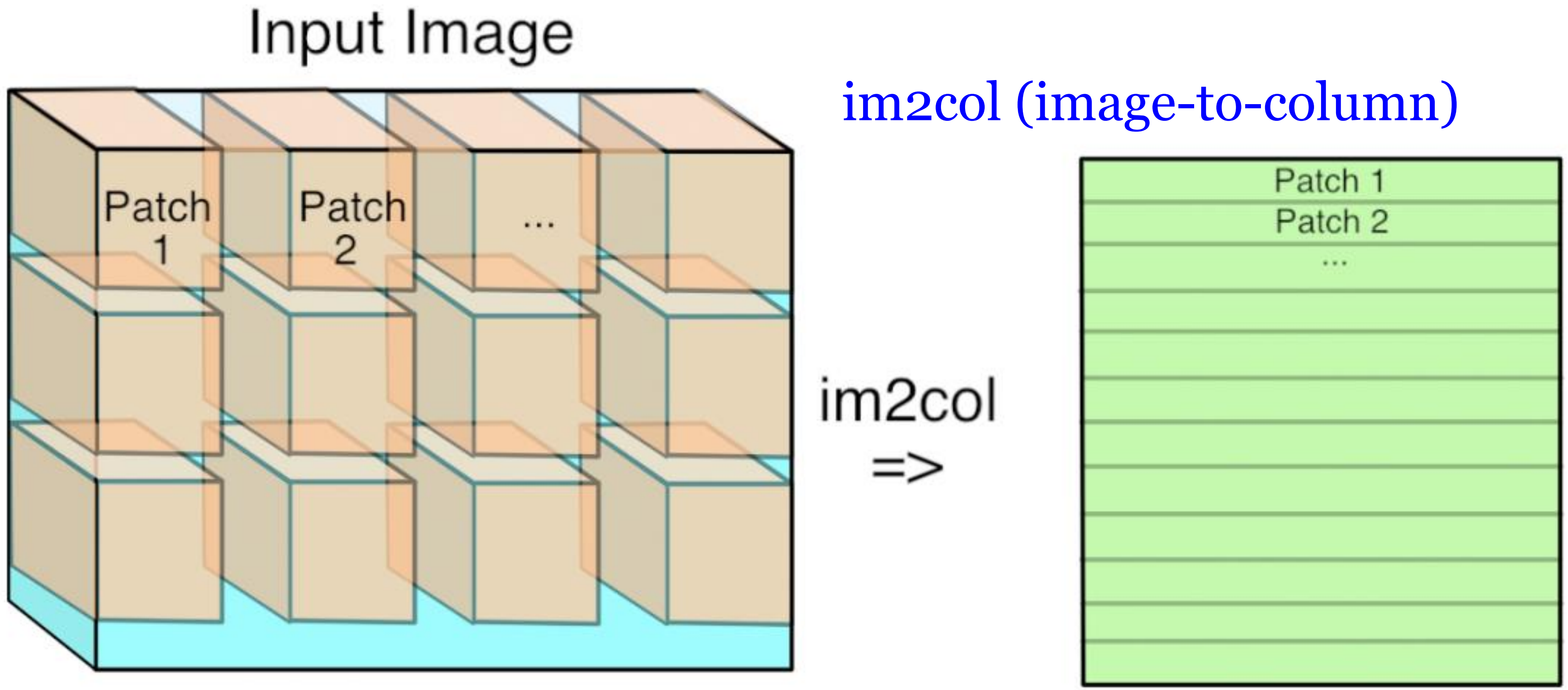

0
20
40
60
80
100
0
20
40
60
80
100

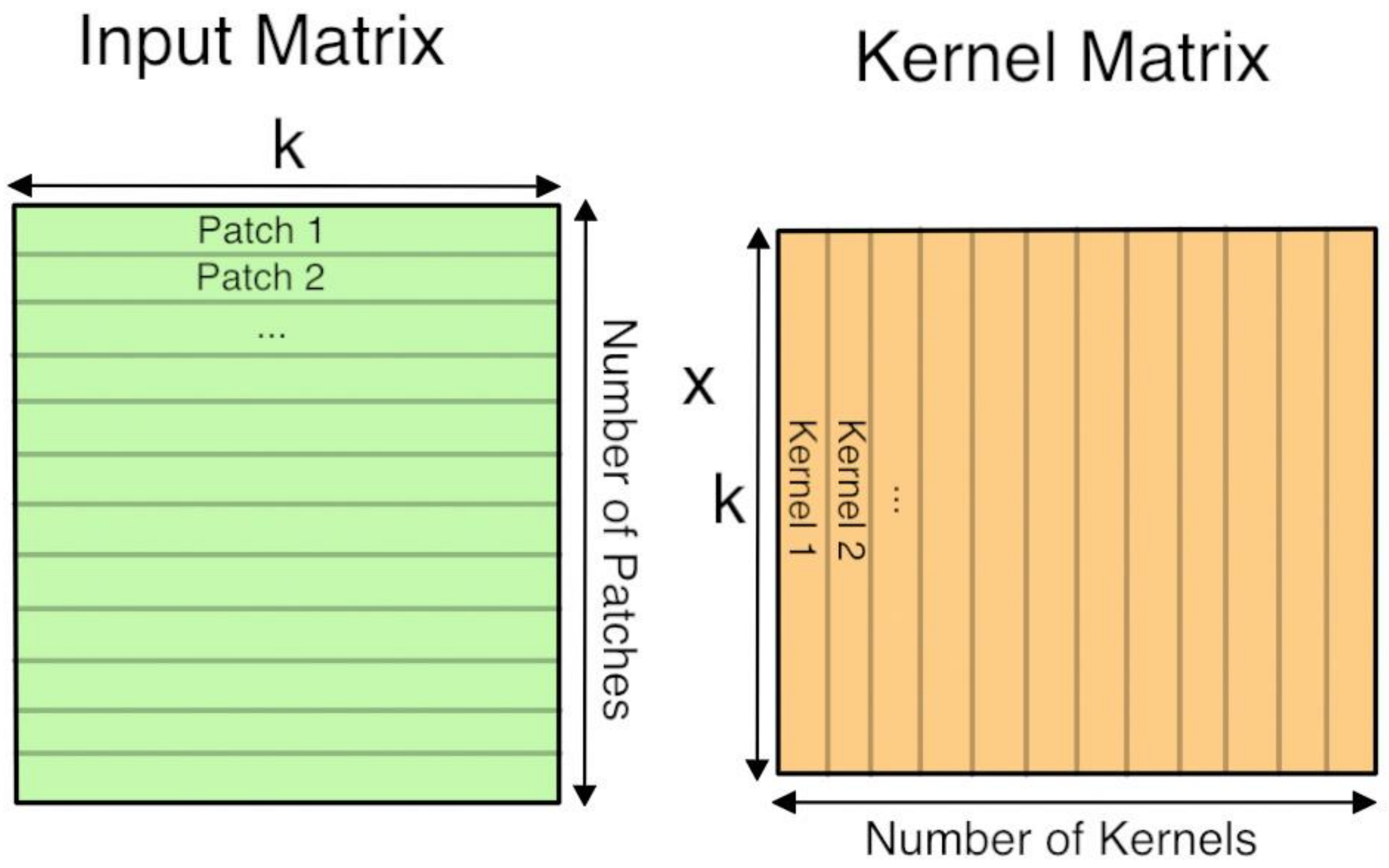
How GEMM works for Convolutions

一个良好的GEMM的实现可以充分利用系统的多级存储结构和程序执行的局部性来充分加速运算。

The first step is to turn the input from an image, which is effectively a 3D array, into a 2D array that we can treat like a matrix.

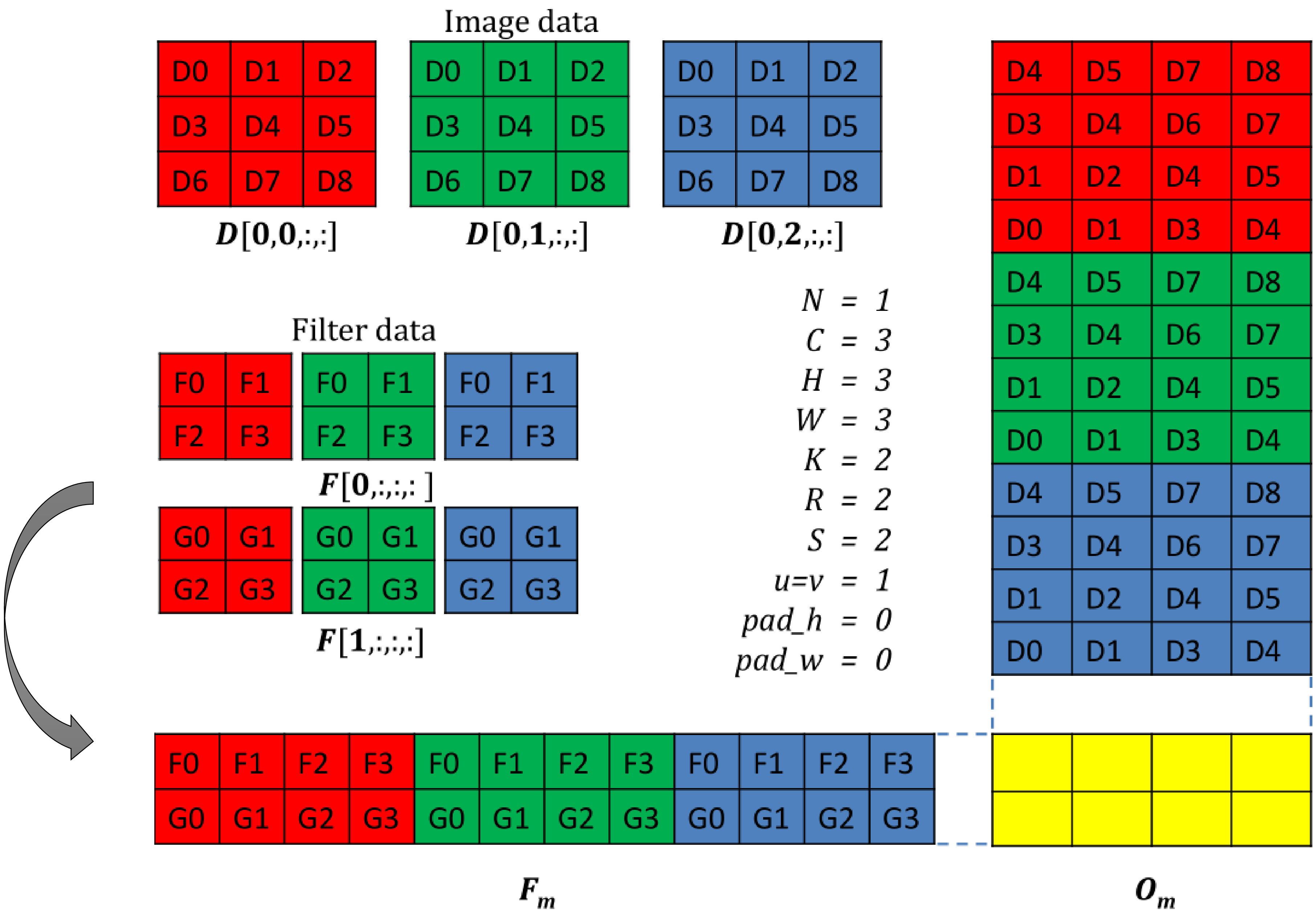
im2col (image-to-column)



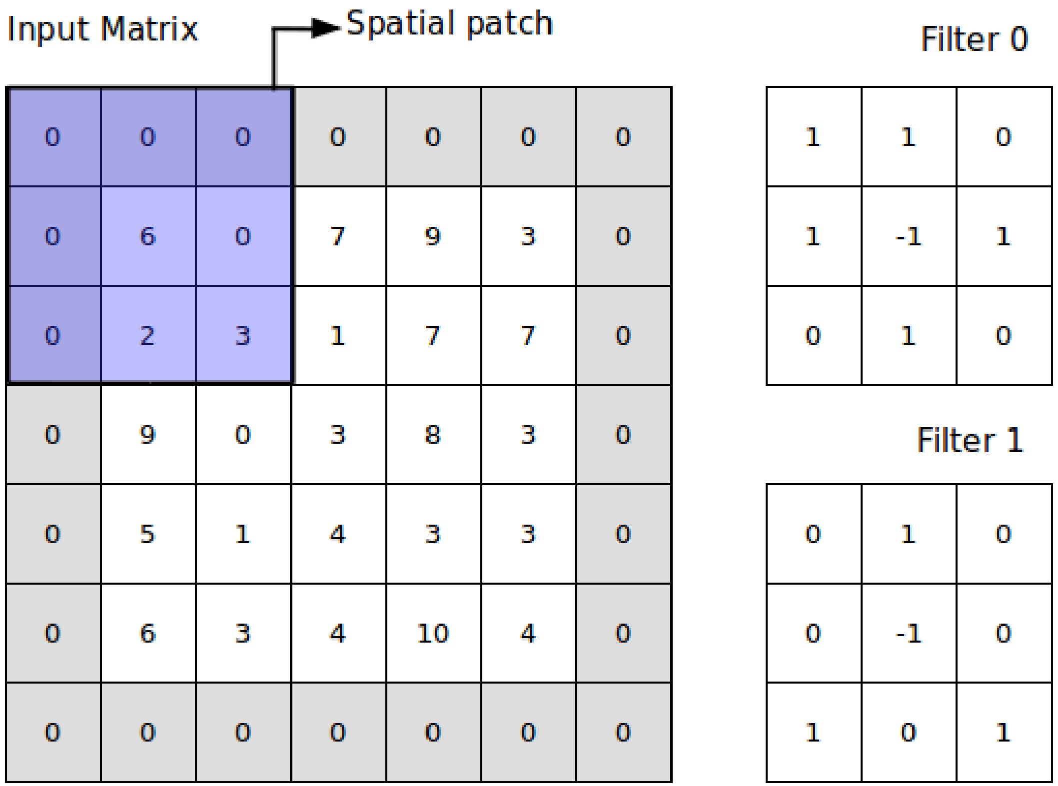


Input Matrix
k
Patch 1
Patch 2
...
Number of Patches
x
Kernel Matrix
k
Kernel 1
Kernel 2
...
Number of Kernels

cuDNN的Im2col

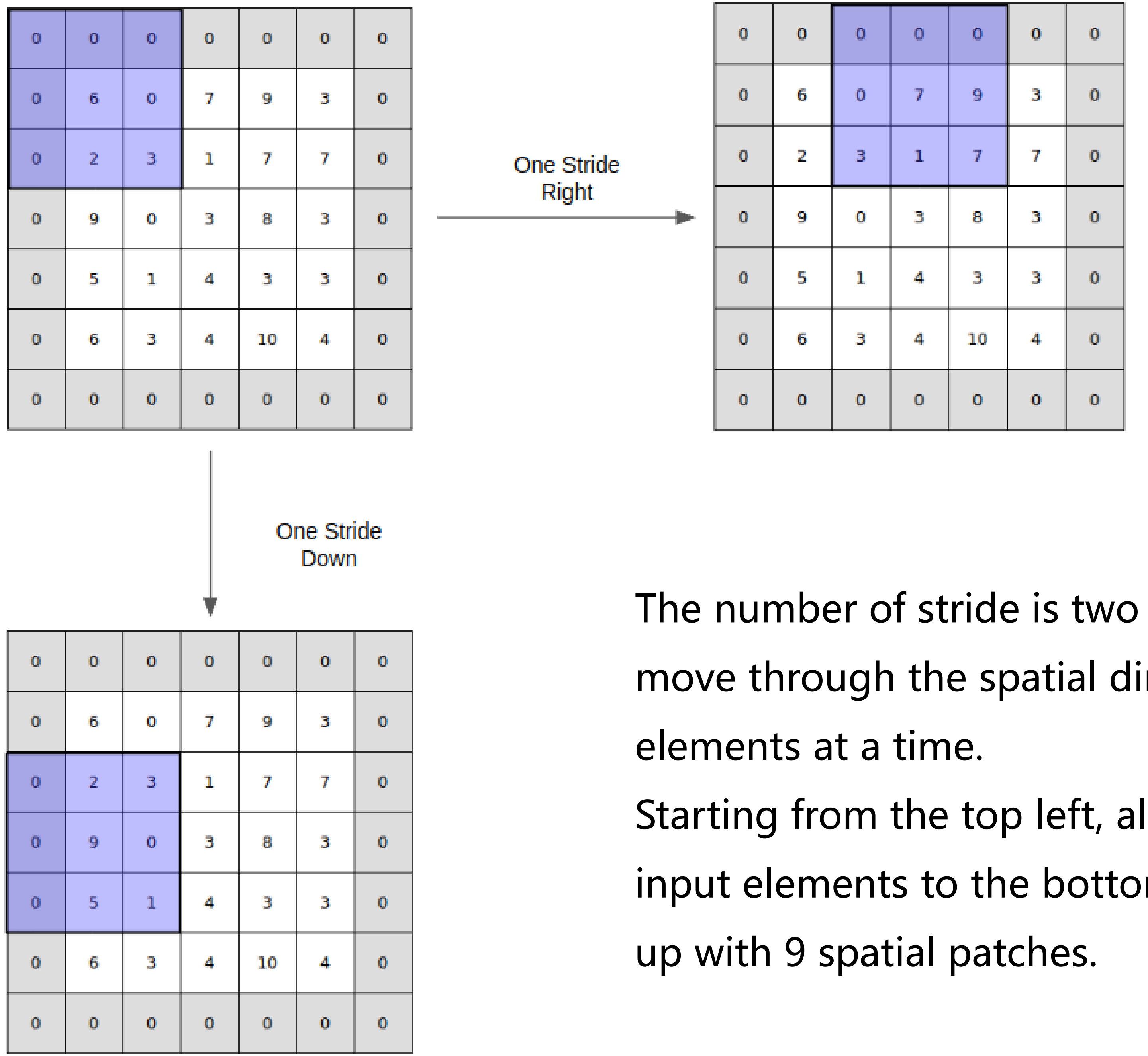


Caffe的im2col



In the illustration above, we have a 5x5 matrix as our input (**Wi** = 5 and **Hi** = 5). We add ***full zero padding*** to our input matrix with an offset of one (**P** = 1), enclosing all the side of our input.

Our filters size are 3x3, thus the value of **F** = 3, and we have two of them, which makes the value of **K** = 2.



The number of stride is two (**S** = 2), which means our filter will move through the spatial dimension of the input patch, two elements at a time.

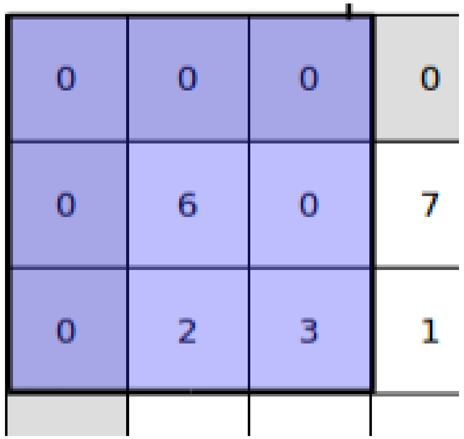
Starting from the top left, all the way until it covered all of the input elements to the bottom right. In our case, we will end up with 9 spatial patches.

Each patch will produce the dot product between the filter with that part of the input—element wise multiplication and sum of all the result, so we end up with a single output element.

Thus, with our parameters (**Wi** = 5, **Hi** = 5, **P** = 1, **F** = 3), the number of output produced will be 9 elements, in the shape of 3x3 matrix (**Wo** = 3 and **Ho** = 3).

As we have 2 filters (**K** = 2), the depth of our final output will also be 2 (**Do** = 2**).**

Topmost left spatial patch on the input



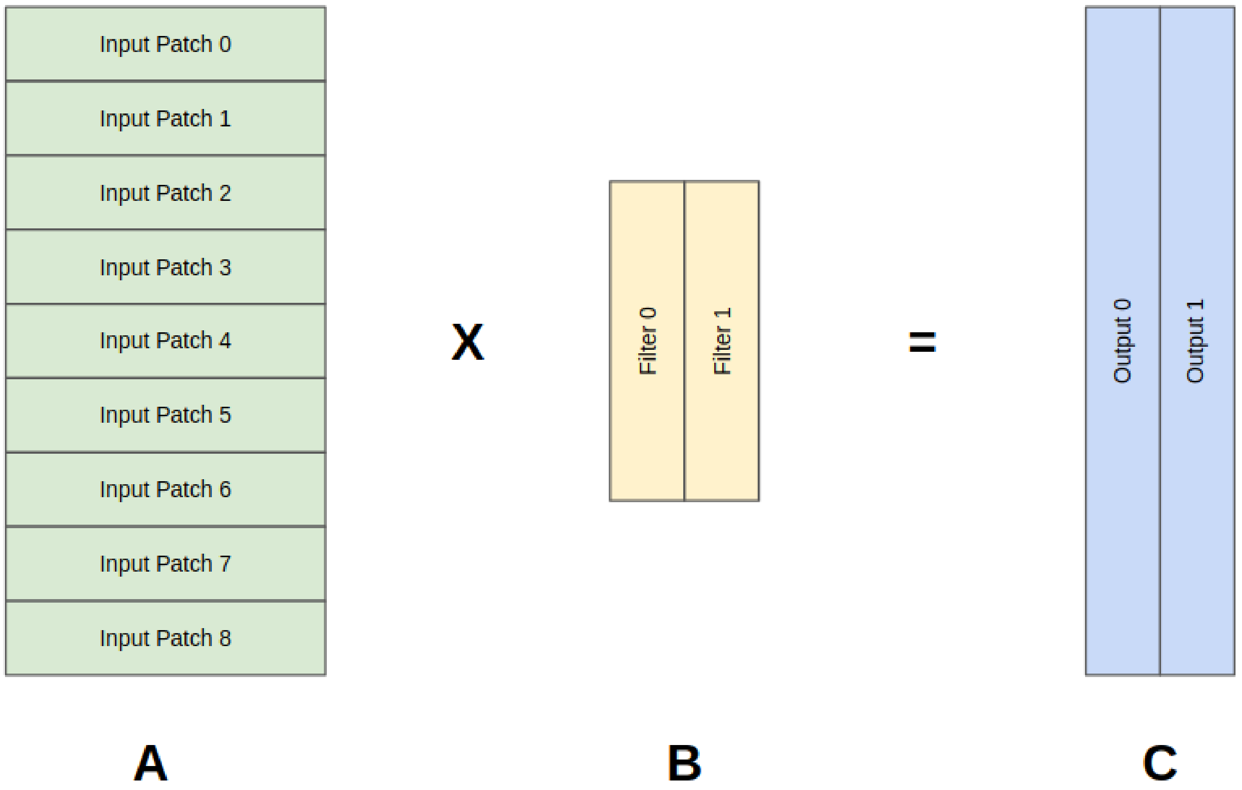
We flatten our input patch into **[0 0 0 0 6 0 0 2 3]**.

This operation is a common operation in image libraries, usually called as **im2col** (image to column).

Based on the input size (5x5), padding size (P=2), filter size (3x3), and the stride (S=2), we already know that there are going to be 9 input patches.

We do the same operation to all 9 of our input patches, then we stack all of the flattened input patches vertically to become a matrix.

GEMM Matrices after im2col Operation on Convolution Matrix



A is a 9x9 matrix and B is a 9x2 matrix, which makes C a 9x2 matrix.

YOLO的GEMM实现原理

假设有输入data_im和卷积核如下:

输入+padding

	1	2	3	4	
	5	6	7	8	
	9	10	11	12	
	13	14	15	16	

height=4
width=4
channels=1//单通道
ksize=3
pad=1
stride=1 //这里假设为1

卷积核

$W_{1,1}$	$W_{1,2}$	$W_{1,3}$
$W_{2,1}$	$W_{2,2}$	$W_{2,3}$
$W_{3,1}$	$W_{3,2}$	$W_{3,3}$

卷积核展开

$W_{1,1}$	$W_{1,2}$	$W_{1,3}$	$W_{2,1}$	$W_{2,2}$	$W_{2,3}$	$W_{3,1}$	$W_{3,2}$	$W_{3,3}$

height_col = (height + 2*pad - ksize) / stride + 1=4
width_col = (width + 2*pad - ksize) / stride + 1=4
channels_col = channels * ksize * ksize=9

卷积核每次划过的像素:

0	0	0
0	1	2
0	5	6

0	0	0
1	2	3
5	6	7

0	0	0
2	3	4
6	7	8

0	0	0
3	4	0
7	8	0

0	1	2
0	5	6
0	9	10

1	2	3
5	6	7
9	10	11

2	3	4
6	7	8
10	11	12

3	4	0
7	8	0
11	12	0

0	5	6
0	9	10
0	13	14

5	6	7
9	10	11
13	14	15

6	7	8
10	11	12
14	15	16

7	8	0
11	12	0
15	16	0

0	9	10
0	13	14
0	0	0

9	10	11
13	14	15
0	0	0

10	11	12
14	15	16
0	0	0

11	12	0
15	16	0
0	0	0

最终得到data_col的第一行所有像素值; c不断自加循环,最后得到data_col像素分布,最终结果如下:

data_col

C=0	0	0	0	0	0	1	2	3	0	5	6	7	0	9	10	11
C=1	0	0	0	0	1	2	3	4	5	6	7	8	9	10	11	12
C=2	0	0	0	0	2	3	4	0	6	7	8	0	10	11	12	0
C=3	0	1	2	3	0	5	6	7	0	9	10	11	0	13	14	15
C=4	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
C=5	2	3	4	0	6	7	8	0	10	11	12	0	14	15	16	0
C=6	0	5	6	7	0	9	10	11	0	13	14	15	0	0	0	0
C=7	5	6	7	8	9	10	11	12	13	14	15	16	0	0	0	0
C=8	6	7	8	0	10	11	12	0	14	15	16	0	0	0	0	0

最终的卷积运算:

由上知: 卷积核的ksize=3,展开后形状为channelsx(ksizexksize), 即1x9,

而data_col形状为channels_colx(height_colxheight_col),即9x16,

所以最终yolo会在通过convolutional_layer.c里的forward_convolutional_layer函数里的gemm函数计算卷积核l.weights和data_col的矩阵乘积,完成卷积操作。

GPU的axpy的实现

```
__global__ void axpy_kernel(int N, float ALPHA, float *X, int OFFX, int INCX,  float *Y, int OFFY, int INCY)
{
    int i = (blockIdx.x + blockIdx.y*gridDim.x) * blockDim.x + threadIdx.x;
    if(i < N) Y[OFFY+i*INCY] += ALPHA*X[OFFX+i*INCX];
}
```

```
extern "C" void axpy_gpu(int N, float ALPHA, float * X, int INCX, float * Y, int INCY)
{
    axpy_gpu_offset(N, ALPHA, X, 0, INCX, Y, 0, INCY);
}
```

```
extern "C" void axpy_gpu_offset(int N, float ALPHA, float * X, int OFFX, int INCX, float * Y, int OFFY, int INCY)
{
    axpy_kernel<<<cuda_gridsize(N), BLOCK>>>(N, ALPHA, X, OFFX, INCX, Y, OFFY, INCY);
    check_error(cudaPeekAtLastError());
}
```

GPU使用cuBLAS库中的cublasSgemm()函数进行矩阵乘法计算

```
void gemm_gpu(int TA, int TB, int M, int N, int K, float ALPHA,
        float *A_gpu, int lda,
        float *B_gpu, int ldb,
        float BETA,
        float *C_gpu, int ldc)
{
    cublasHandle_t handle = blas_handle();
cudaError_t status = cublasSgemm(handle, (TB ? CUBLAS_OP_T : CUBLAS_OP_N),
            (TA ? CUBLAS_OP_T : CUBLAS_OP_N), N, M, K, &ALPHA, B_gpu, ldb, A_gpu, lda, &BETA, C_gpu,
ldc);
                // 检查cublasSgemm运算是否正常（可以看到，darknet中，cuda的每一步操作，基本都要检查一下
运行状态是否正常）

                check_error(status);
}
```

参考CUDA关于cuBLAS库的官方文档，此处cublasSgemm()函数在其中的2.7.1节：cublas<t>gemm()，可以看出，如果不是因为存储方式不同，cublasSgemm()函数的结构与Darknet自己实现的cpu版gemm_cpu()一模一样；因为二者存储格式的不同， 需要交换A_gpu,B_gpu的位置，对应M与N之间，TB与TA间，ldb与lda之间都要相应交换。