Department of Computer Science
IIT Jodhpur

ALGORITHMS FOR BIG DATA PROJECT REPORT

NOVEMBER 2019

HEART OF *DEEP*-LEARNING

ASHUTOSH JATAV (B16CS004)

Introduction:

Machine Learning, especially deep learning has been a core algorithm to be widely used in many fields such as natural language processing, object detection etc. Deep learning algorithm like convolutional neural network and its variants consumes a lot of memory resources. In such a case powerful deep learning algorithm are difficult to apply on mobile memory limited platforms. For this purpose, we have analysed several methods to optimize convolution in convolutional neural network.

Methodology

GEMM:

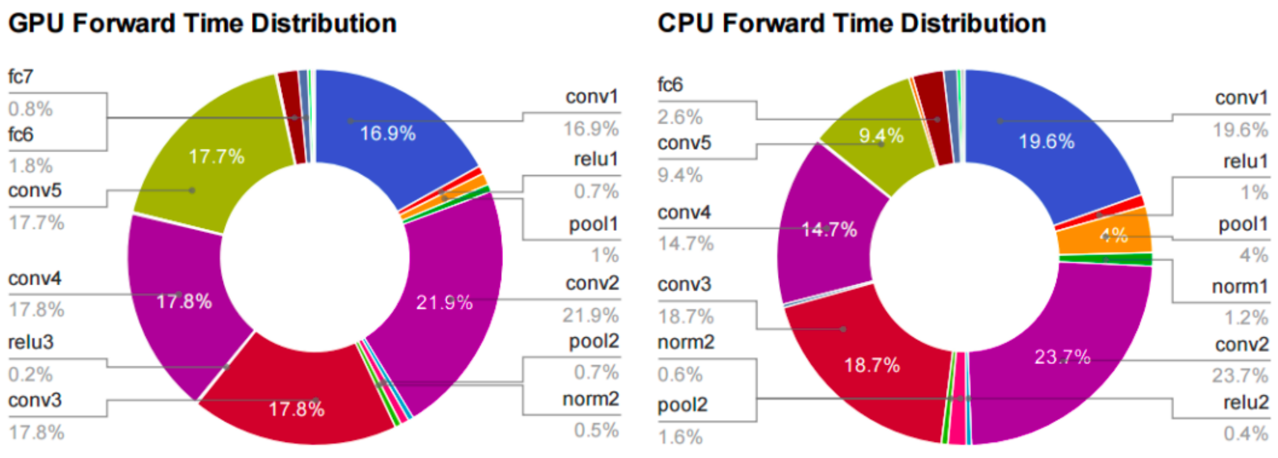


Figure 1: Alex net different layers run time

All the layers that start with fully-connected or convolution are implemented using GEMM and almost all the time is spent on those layers

GEMM: General Matrix to Matrix Multiplication what it does is it multiplies two input matrices together to get an output one.

So, for example single layer in typical network may require multiplication of 256 row and 1152 column matrix by a 1512 row, 192 col matrices to produce a 256 row, 192 columns result. That require 57 million floating point operations and there can be dozens of these layers in a modern architecture. So a network may need billion FLOPs to calculate a single frame

Fully – Connected Layers:

For each output value of an FC layer looks at every value in input layer, multiplies them all by the corresponding weight it has for that input index and sums the result to get its output.

Convolutional Layers:

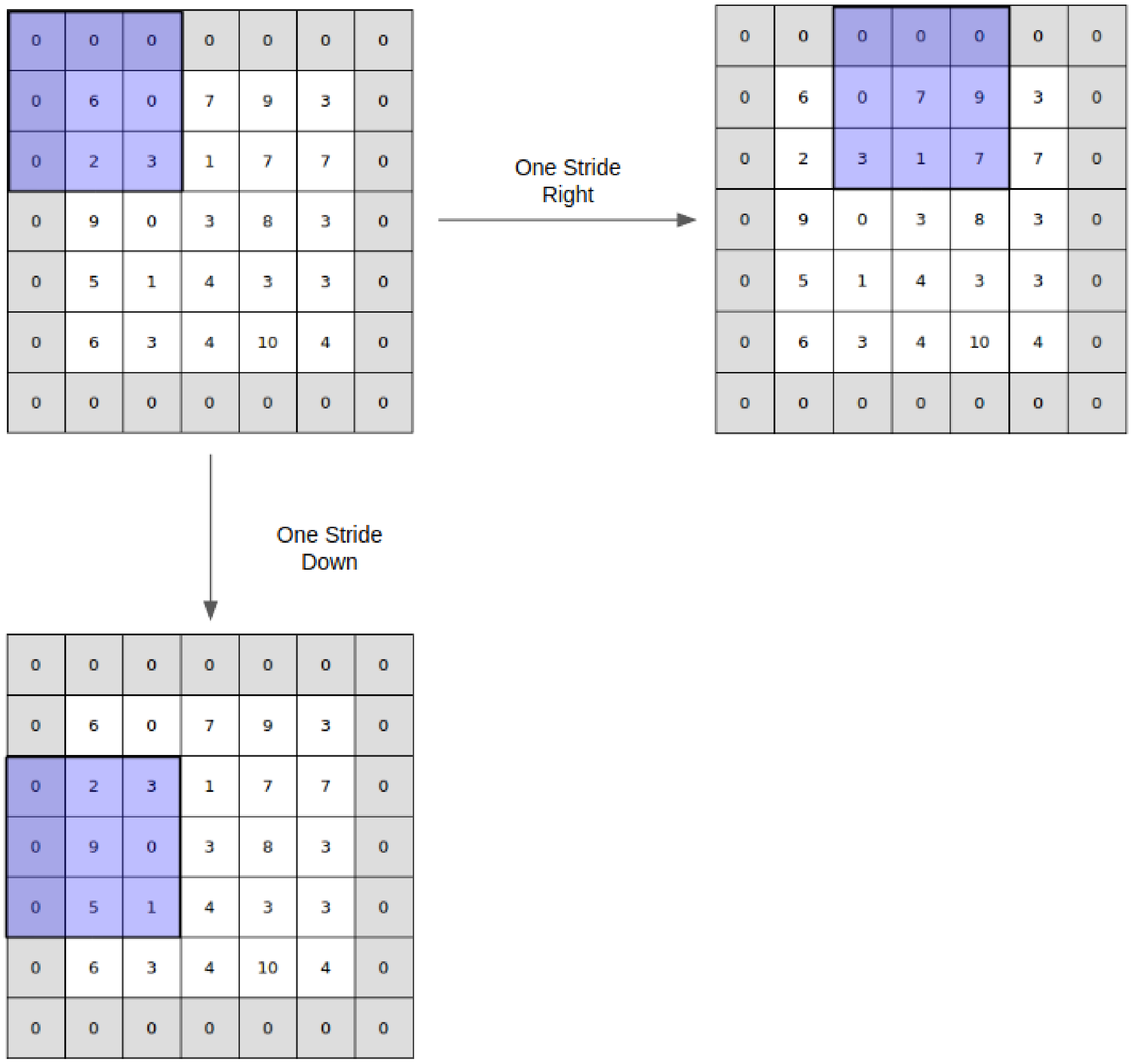
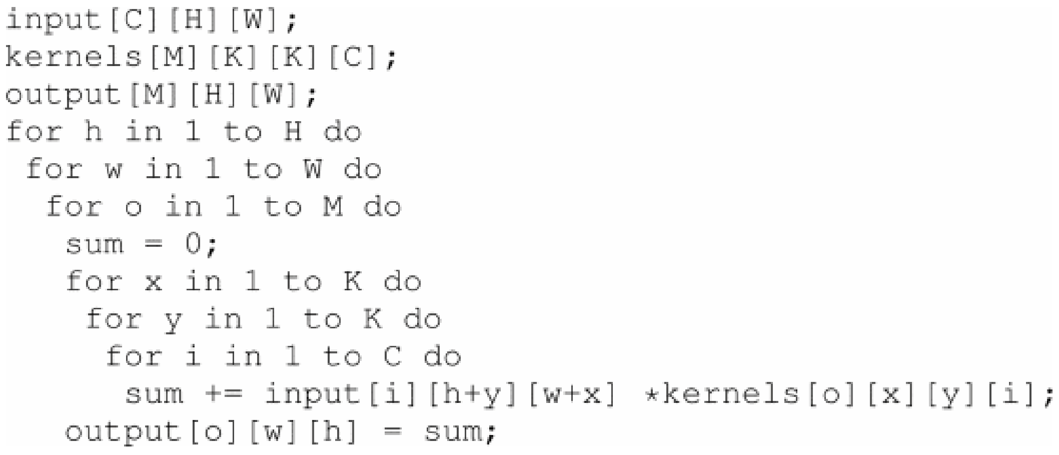


Figure 2: Convolution in Deep learning

Convolutional layer treats its input as a two-dimensional image, with number of channels for each pixel. Convolutional operation produces its output by taking a number of kernels of weights and applying them across the image. Kernel contains a pattern weights when the part of the input image it's looking at has similar pattern it outputs a high value.
CNN uses parameter sharing of weights by neuron in a particular feature map, this works in a way by assuming that if one feature is useful to compute at some spatial position (x,y) then it should also be useful to compute at a different position(x', y') in other words let's say a volume size of [x, y, d] has d depth slice each of [x,y].
neurons in each depth slice are constrain to use the same weights and bias so there would be only d unique weights in this case instead of x*y*d.

As an example, we have an image, or tensor, with the width (**W**), height (**H**), and depth (**D**). We also have a patch called filter sized **F** in height & width and depth **D.** We apply **K** number of filters with that size to the input. The pseudo code as follows:



```
input[C][H][W];
kernels[M][K][K][C];
output[M][H][W];
for h in 1 to H do
 for w in 1 to W do
  for o in 1 to M do
   sum = 0;
   for x in 1 to K do
    for y in 1 to K do
     for i in 1 to C do
      sum += input[i][h+y][w+x] *kernels[o][x][y][i];
   output[o][w][h] = sum;
```

Figure 3: GEMM algorithm

GEMM in Convolution layer:

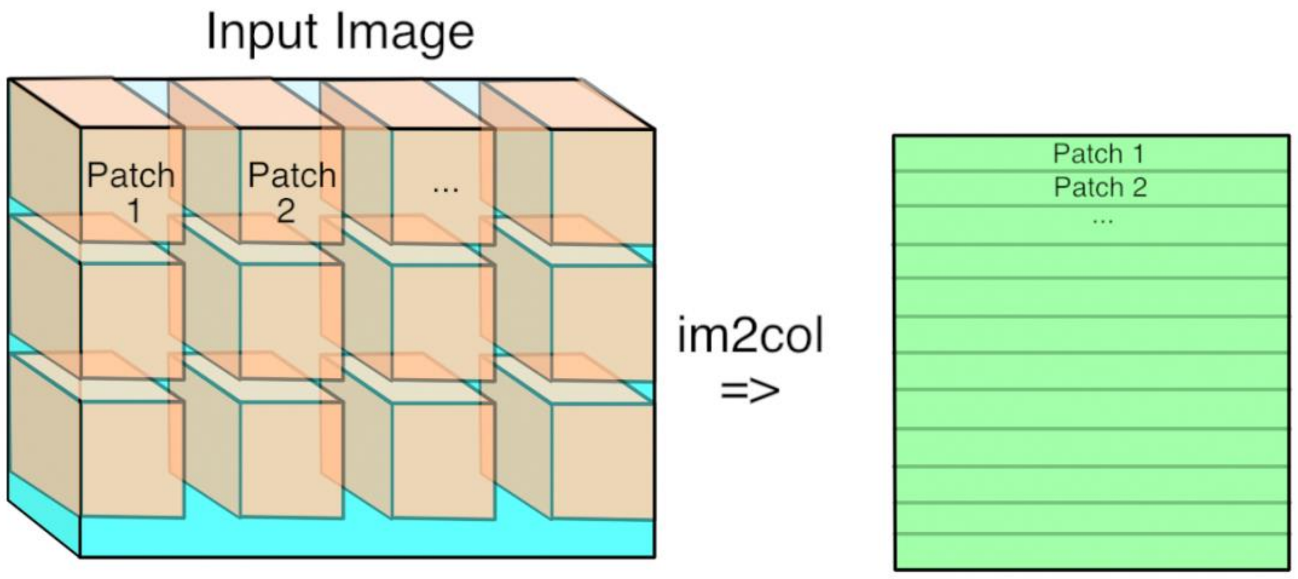


Figure 4: GEMM

1.) Turn the input image 3D array into a 2D array that can be treat like a matrix. Where each kernel is applied is little 3-D cube within the image so we can take each one of these cube of input values and copy them out as a single column into a matrix step is known as image-to-column

The number of strides 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.
If the stride is less than the kernel size then pixels that are included in overlapping kernel sites will be duplicated in the matrix which is inefficient.

Same thing will be done for kernel's weights

k is number of values in each patch and kernel so it's kernel width * kernel height * depth. The resulting matrix is 'Number of patches' columns high, by 'Number of kernel' rows wide. This matrix is actually treated as a 3D array by subsequent operations, by taking the number of kernels dimension as the depth, and then splitting the patches back into rows and columns based on their original position in the input image.

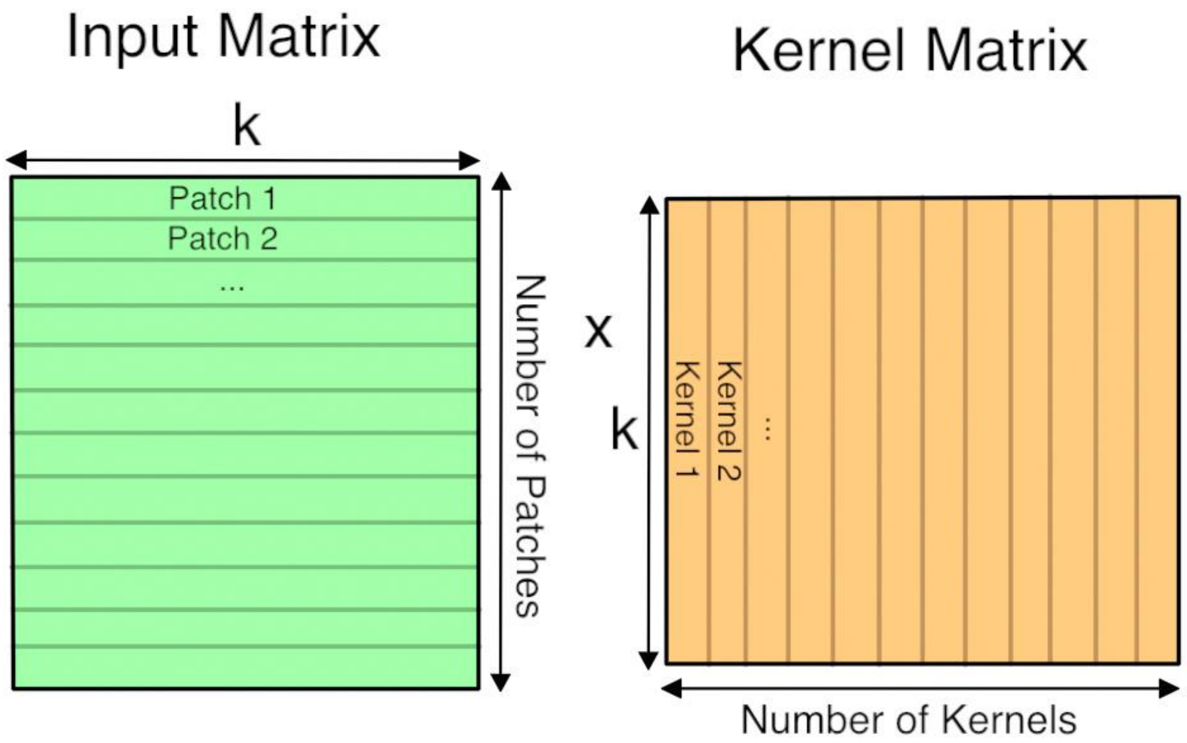


Figure 5: Patch Matrix After GEMM

MCMK Model [6]

The disadvantage of im2col is that it replicates input data to create the input patch-matrix. For convolution with k*k kernel, the input patch matrix can be $k^{2.}$

A GEMM based MCMK algorithm that does not require data replication in the input could be useful for memory-limited embedded system, this MCMK algorithms that eliminate data replication on the input, at the cost of some increase in the size of the output.
Im2col needs $C*H*W*K^2$ space for patch matrix

GEMM-based convolution by sum of scaled matrices

- Scale the input image by each cell of the kernel
- Convolution becomes addition of sub matrices scaled by one element of the kernel, to get the right answer we need to shift the matrix entices before adding

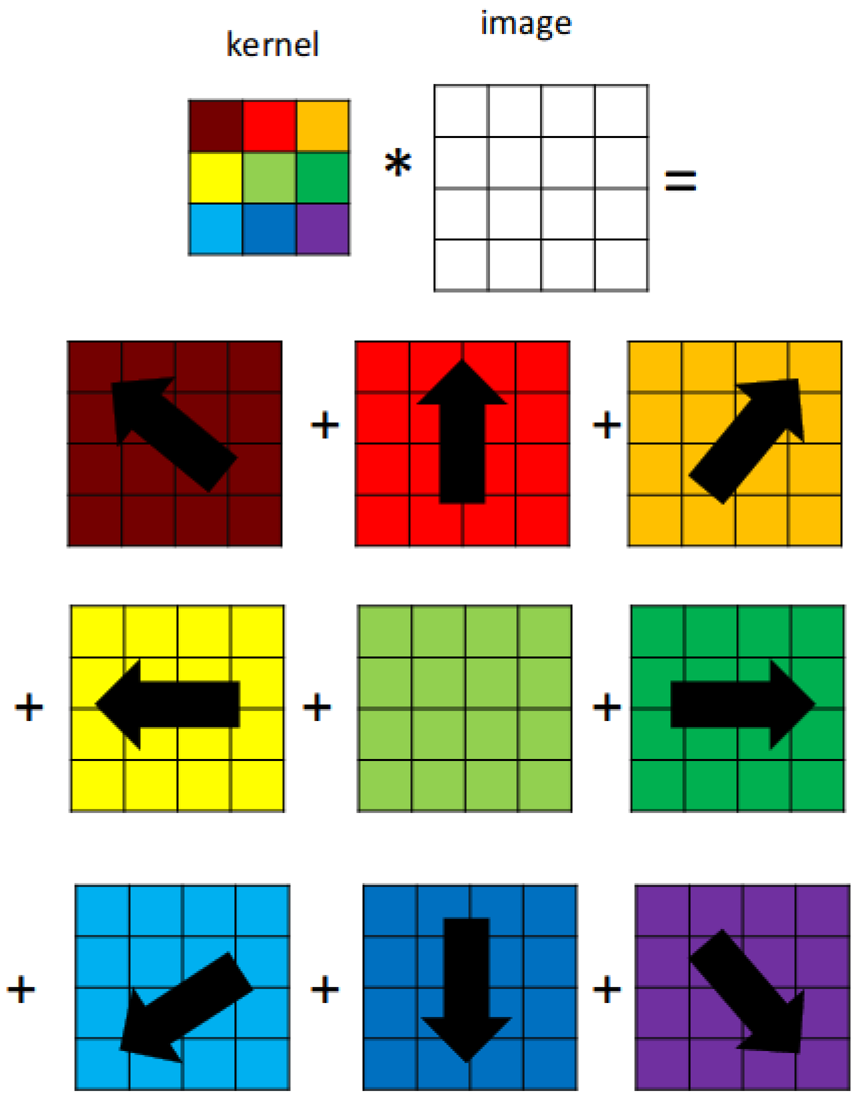


Figure 6: Convolution as sum of scaled matrices

To extend this to multiple channels replace matrix scaling with 1*1 DNN convolution, K*K DNN convolution can be computed as the sum of K^2 1*1 DNN convolution, more GEMM calls required but can be implemented without need for extra patch matrices.

Accumulating Algorisms:

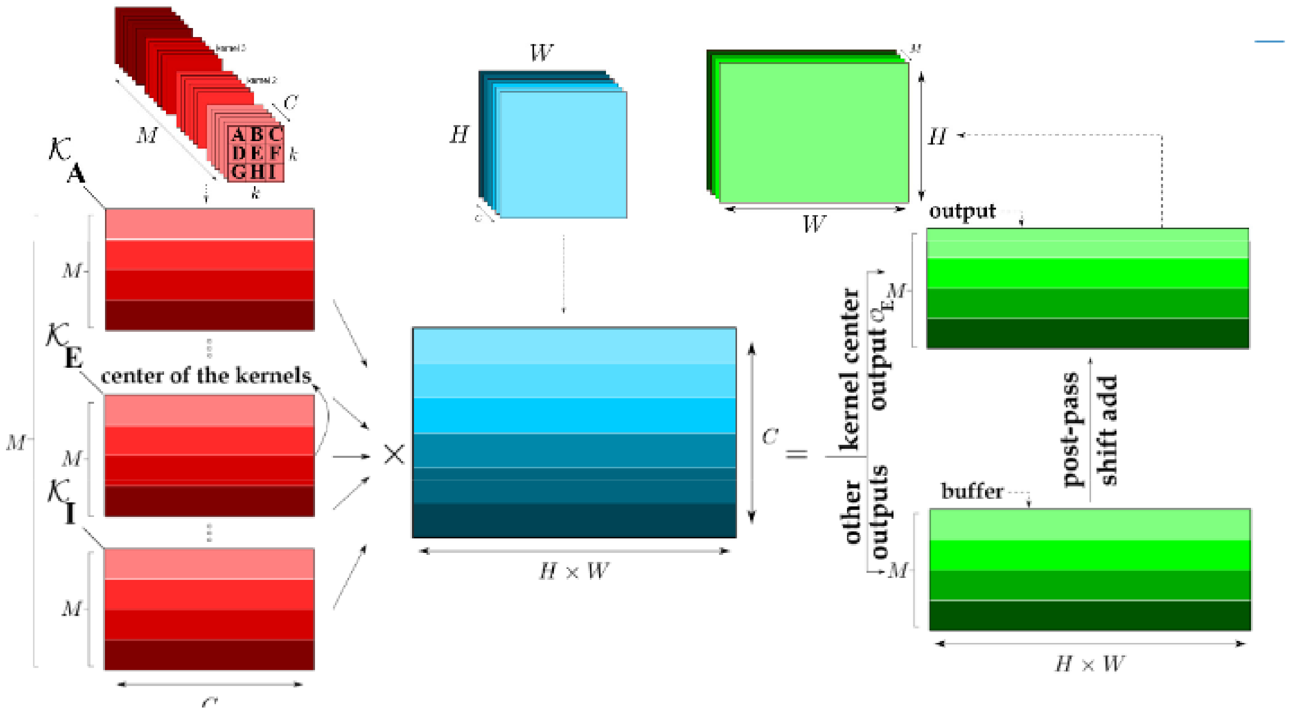


Figure 7: kernel accumulating algorithm

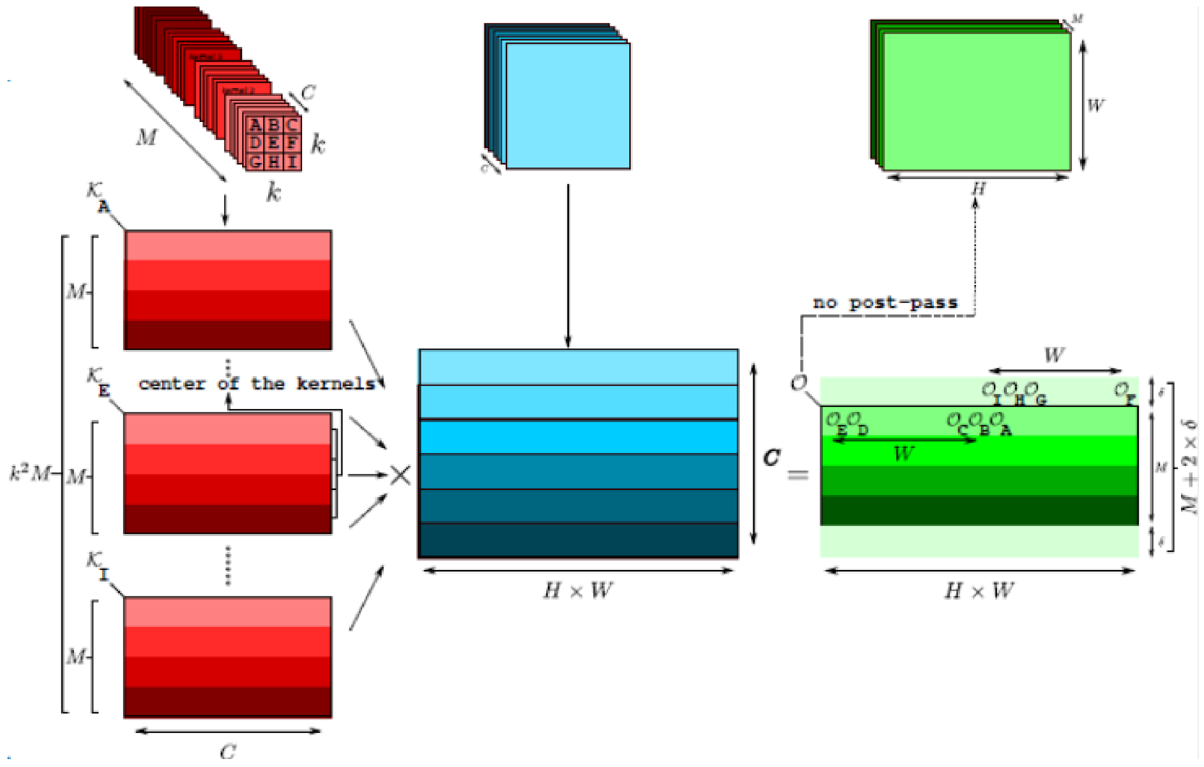
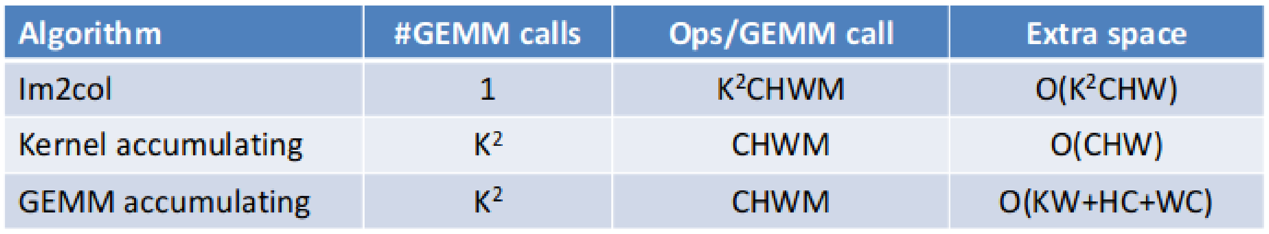


Figure 8: GEMM accumulating algorithm

GEMM Accumulating Algorithm:

Most of the algorithms deal with boundaries as special cases but here boundaries we spill into and over write the next row so lots of wrong values in result matrix, this can be fix by dynamically modify input matrix with carefully-placed zeros or post-pass fix up of values

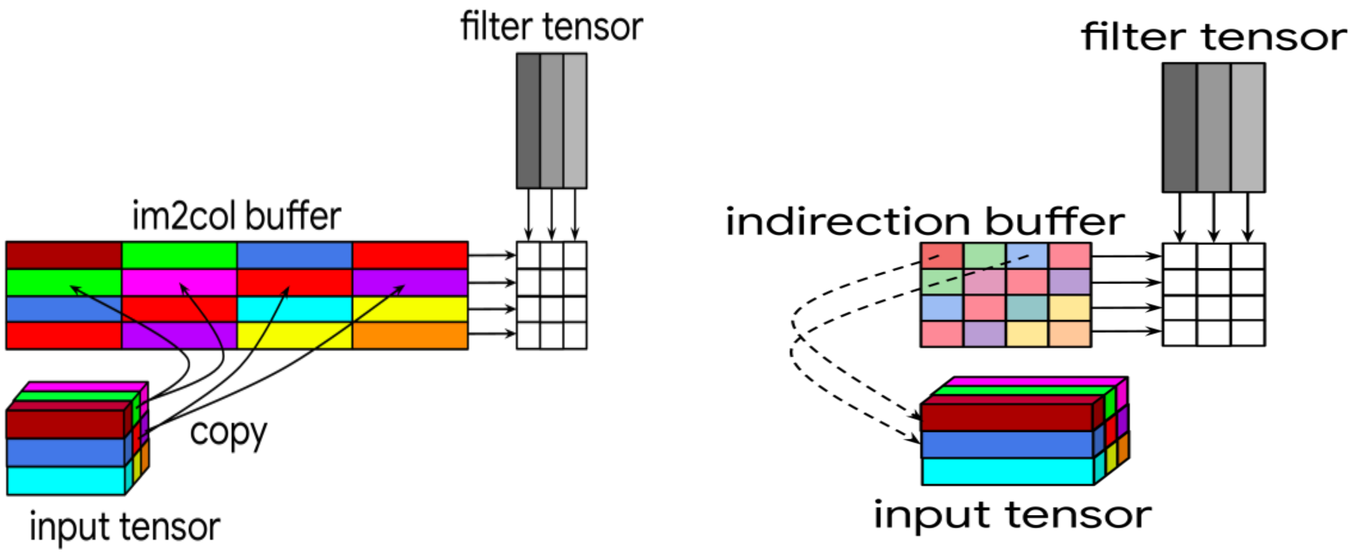
Results:



Algorithm	#GEMM calls	Ops/GEMM call	Extra space
Im2col	1	K^2CHWM	$O(K^2CHW)$
Kernel accumulating	K^2	CHWM	O(CHW)
GEMM accumulating	K^2	CHWM	O(KW+HC+WC)

Indirect Convolution Algorithm [7]

GEMM based algorithms reshuffle data to fir into GEMM interface the Indirect convolution algorithm instead modifies the GEMM primitive to adopt it to the original data layout



In indirect GEMM operation as component of Indirect Convolution algorithm, the indirect buffer contains only pointers to rows of the input tensor and the indirect GEMM operation reads rows of data directly from the input tensor.

An extra loop has been added to load the new pointers in the indirect buffer computes dot products of K elements specified by these pointers with the filter data and accumulates the result of the dot product.

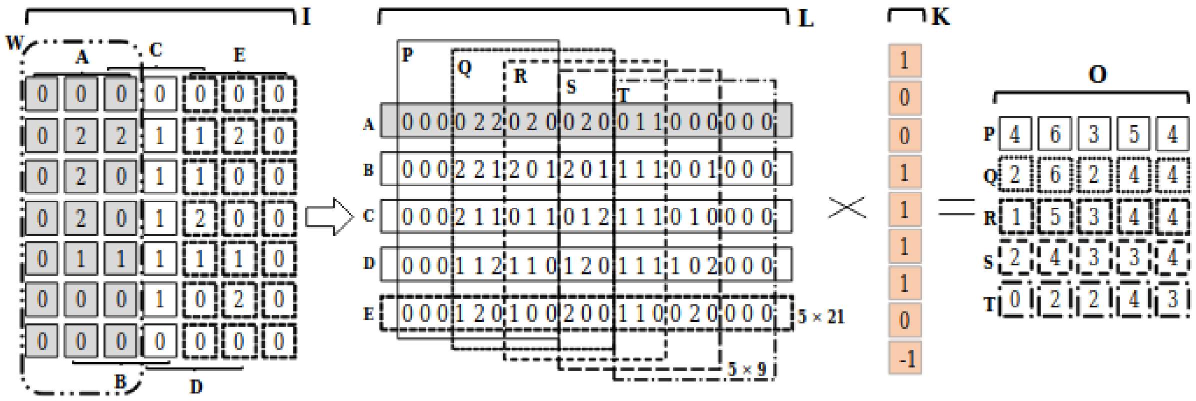
MEMORY EFFICIENT CONVOLUTION FOR DEEP NEURAL NETWORK [8]

im2col has large amount of memory overhead because of significant amount of redundancy in the lowered matrix and the redundancy further increase as size of k decreases.

MEC overcome this by lowering multiple columns at once rather than each single individual sub-matrix w.r.t K. MEC copies sub-matrices W of size i_h*k_w into one row of L

For original im2col algorithm let's say our input is of size 9*9 and kernel size 3*3 then it will create a patch matrix of 25*9

what this algorithm does is



At first partition I A=[0:7, 0:3] then slide W by s(1) to right and create another partition B=[0:7, 1:4] at the end there will be 5 horizontal partitions {A, B, C, D, E} the resulting lowered matrix l has dimensions 5*21 which is 54% smaller than original im2col 25*9.

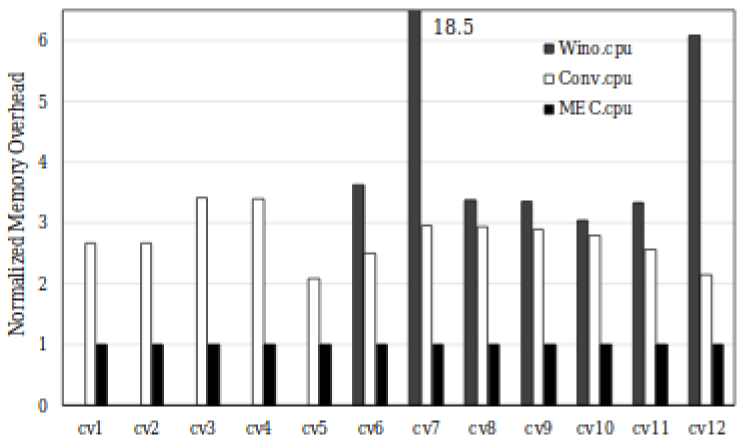
Then MEC multiply lowered matrix with kernel K in a different way than im2col. MEC also creates another set of vertical partition {P, Q, R, S, T} within L where each partition is of size $o_w \times k_h k_w$ which is of size 5*9, by shifting right by $s_h k_w$(3), then each row of output matrix O is product between one of the partitions in {P, Q, R, S, T} and K.

Intuitively MEC eliminates the vertical redundancy in conventional im2col based convolution then it recovers the information by merely shifting the vertical partitions.

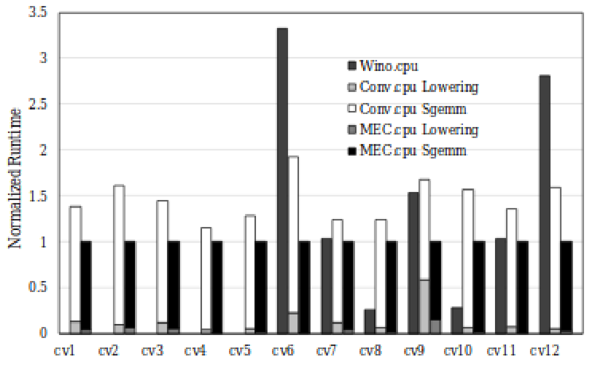
Algorithm 1 $O = VanillaMEC(I, K, s)$

```
1: Allocate $O$ with $o_h o_w$ elements
2: Allocate $L$ with $o_w i_h k_w$ elements
3: Interpret $L$ as $o_w \times i_h \times k_w$ tensor
4: **for** $w \in 0 : o_w, h \in 0 : i_h$ **in parallel do**
5:     $L[w, h, 0 : k_w] = I[h, s_w w : s_w w + k_w]$
6: **end for**
7: Interpret $L$ as $o_w \times i_h k_w$ matrix
8: Interpret $K$ as $k_h k_w \times 1$ matrix
9: Interpret $O$ as $o_h \times o_w$ matrix
10: **for** $h \in 0 : o_h$ **in parallel do**
11:     $O[h, 0 : o_w] =$
        $L[0 : o_w, s_h k_w h : s_h k_w h + k_h k_w] \times K$
12: **end for**
13: Return $O$
```

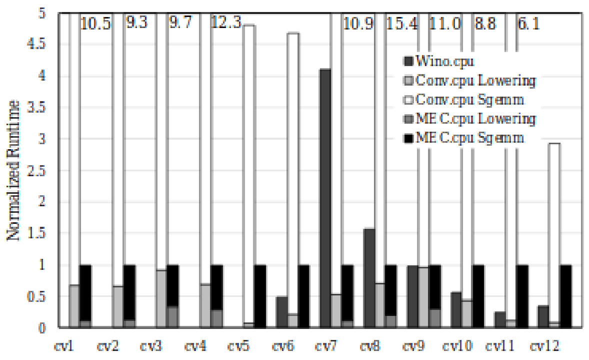
Results:



(b) Memory-overhead on **Mobile and Server-CPU**



(c) Runtime on **Mobile**



(d) Runtime on **Server-CPU**

References

[1] CS231n Convolutional Neural Networks. http://cs231n.github.io/convolutional-networks/, 2018

[2] Convolution in Caffe: a memo. https://github.com/Yangqing/caffe/wiki/Convolution-in-Caffe:-a-memo, 2018

[3] Caffe Installation. http://caffe.berkeleyvision.org/installation.html, 2018

[4] Netlib BLAS. http://www.netlib.org/blas/, 2018

[5] Nvidia cuBLAS. https://developer.nvidia.com/cublas, 2018

[6] https://ieeexplore.ieee.org/document/7995254

[7] https://www.groundai.com/project/the-indirect-convolution-algorithm/1

[8] https://arxiv.org/pdf/1706.06873.pdf