Convolution filters with High Level Synthesis tools

Stefano Mattoccia

http://vision.disi.unibo.it/~smatt

Dipartimento di Informatica, University of Bologna

DEEP LEARNING ON-CHIP

September 20-22, 2017 – Politecnico di Torino, Torino (Italy)

Deep-learning on FPGA

Training deep-networks (Tensorflow, Torch, etc)

Model conversion
C/C++ (HLS) -> RTL

XILINX
VIVADO
High Level
Sybthesis

Inference on FPGA

HLS tools: allow automatic synthesis of behavioral C/C++ code into RTL

Why HLS tools vs HDL (Hardware Description Language)?

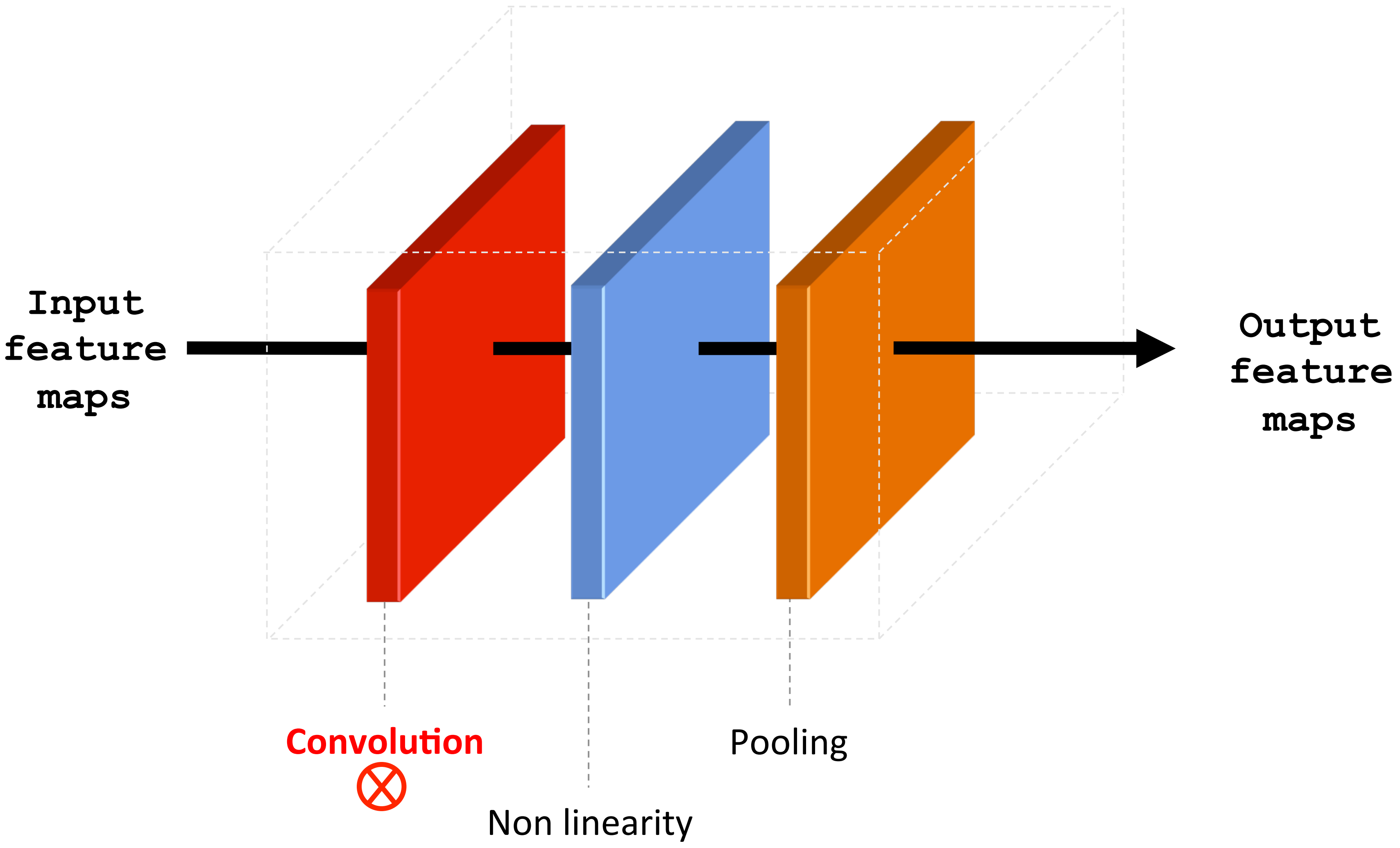
- accelerate the design process
- more similar to a conventional software design flow
- suited for high-level algorithms (e.g. Computer vision)
- more attractive for CS students (and researchers too)

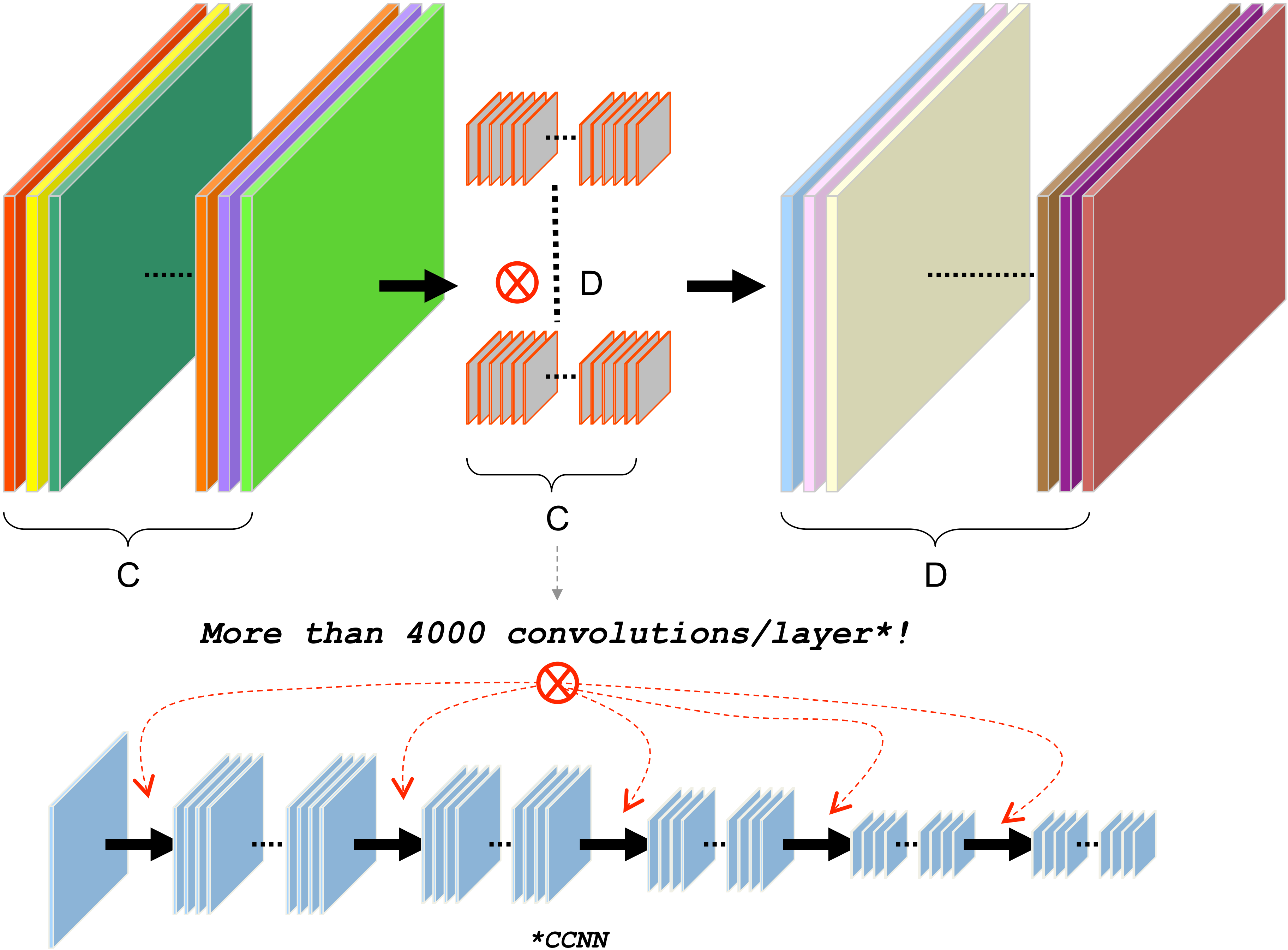
Shared with HDL:

- require a deep knowledge of the target architecture
- require a deep analysis of the problem

Xilinx Vivado HLS, designed by Prof. Cong's group at UCLA

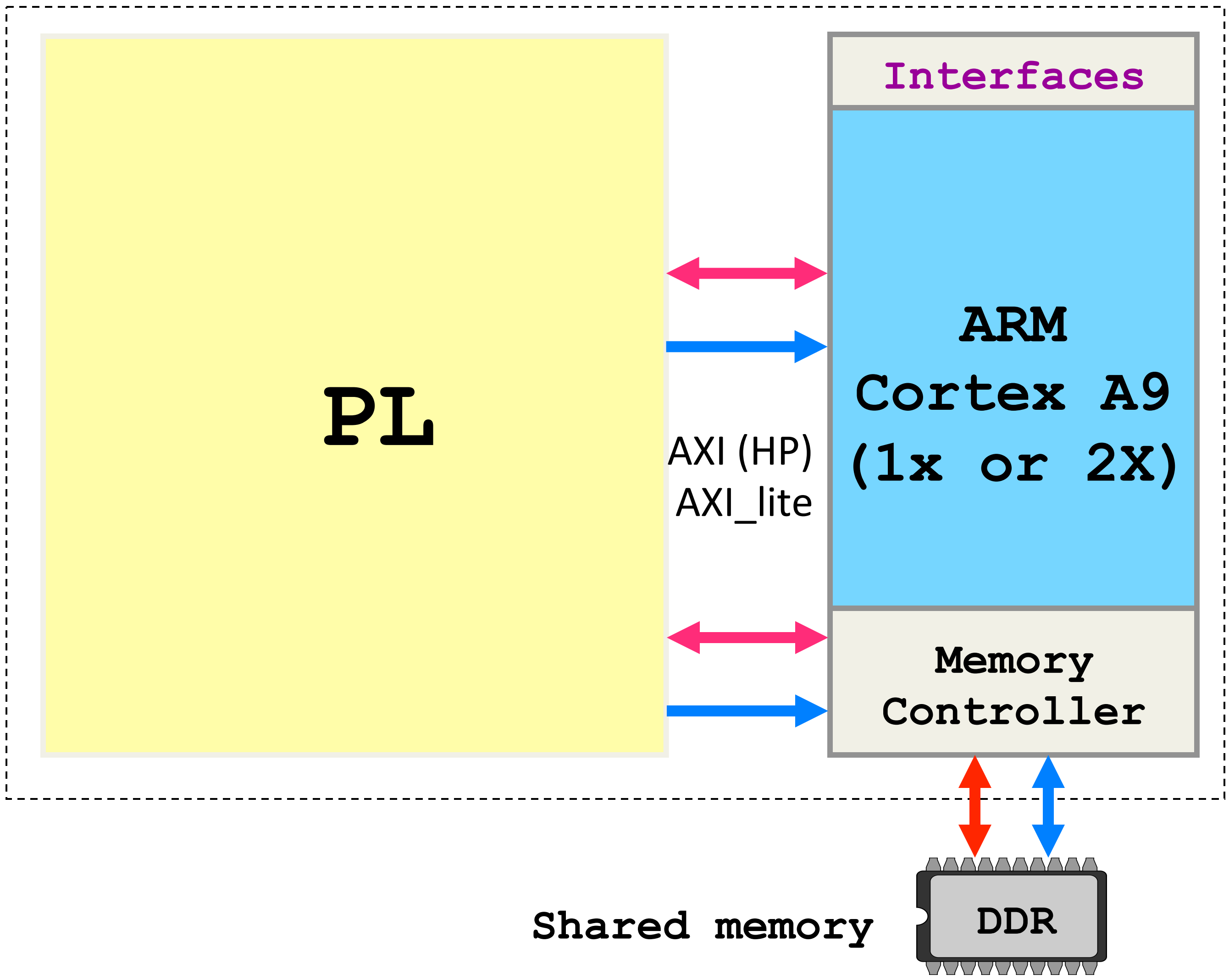
- **A typical CNN is made of multiple layers**
- **Early ones are most demanding (convolution filters)**





D
C
C
D
More than 4000 convolutions/layer*!
*CCNN

Zynq architecture



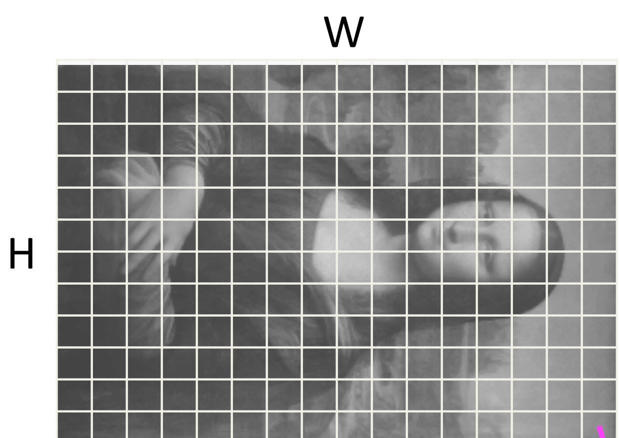
Zynq 7020 (7100)

Look-Up Tables (LUTs)	:	53,200	(277,400)	≈5X
Flip-Flops	:	106,400	(554,800)	≈5X
BRAM (dual port,36Kb)	:	140	(755)	≈5X
DSP	:	220	(2020)	≈9X

About 600 KB
A monochrome VGA (640x480) image is ≈300 KB

Imaging sensors

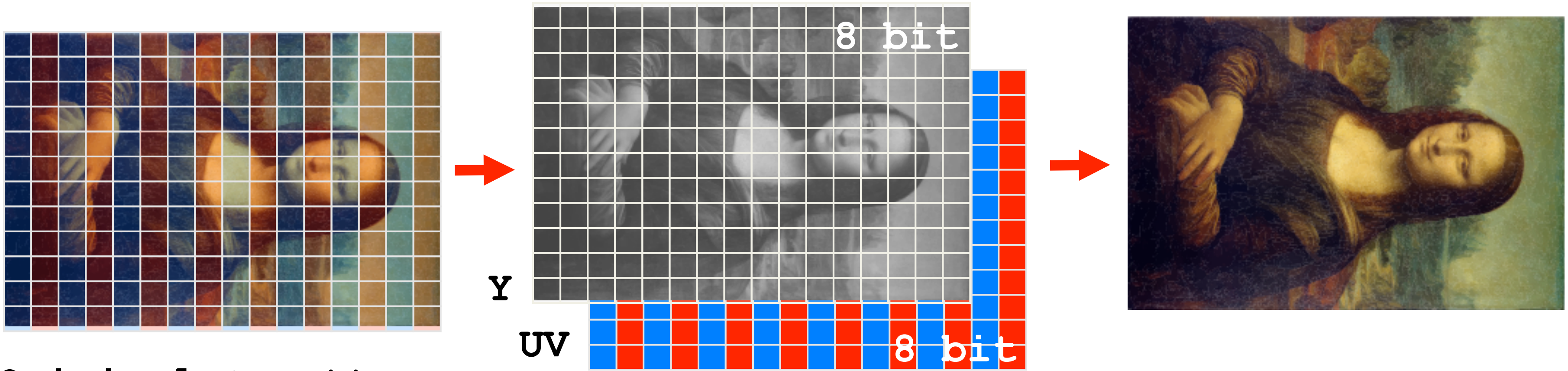
- **Continuos video stream (30+ fps)**
- **Few syncronization signals (often embedded in the stream)**
- **Programmable (resolution, frame rate, etc)**
- **Image encoding: monochrome (below) or color (next slide)**



Monochrome image

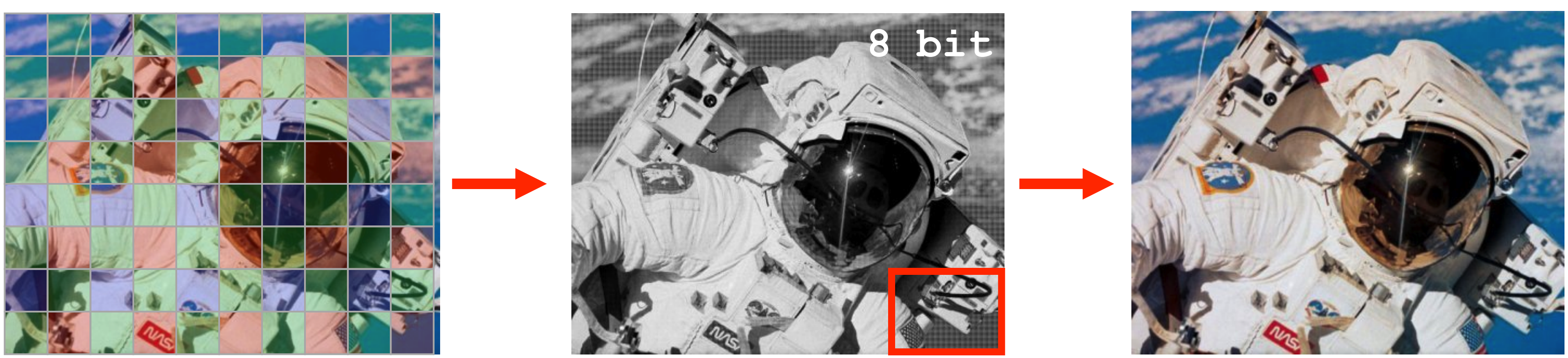
Each pixel is encoded with 8+ bits

Color: YUV (4:4:2)

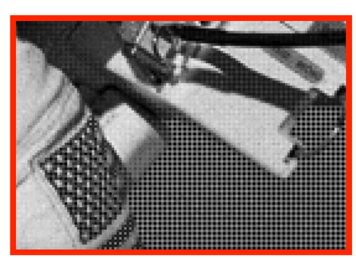


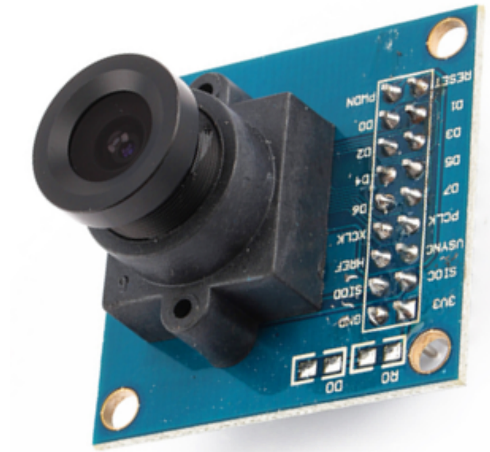
Original + pattern

Color: Bayer



Original + pattern



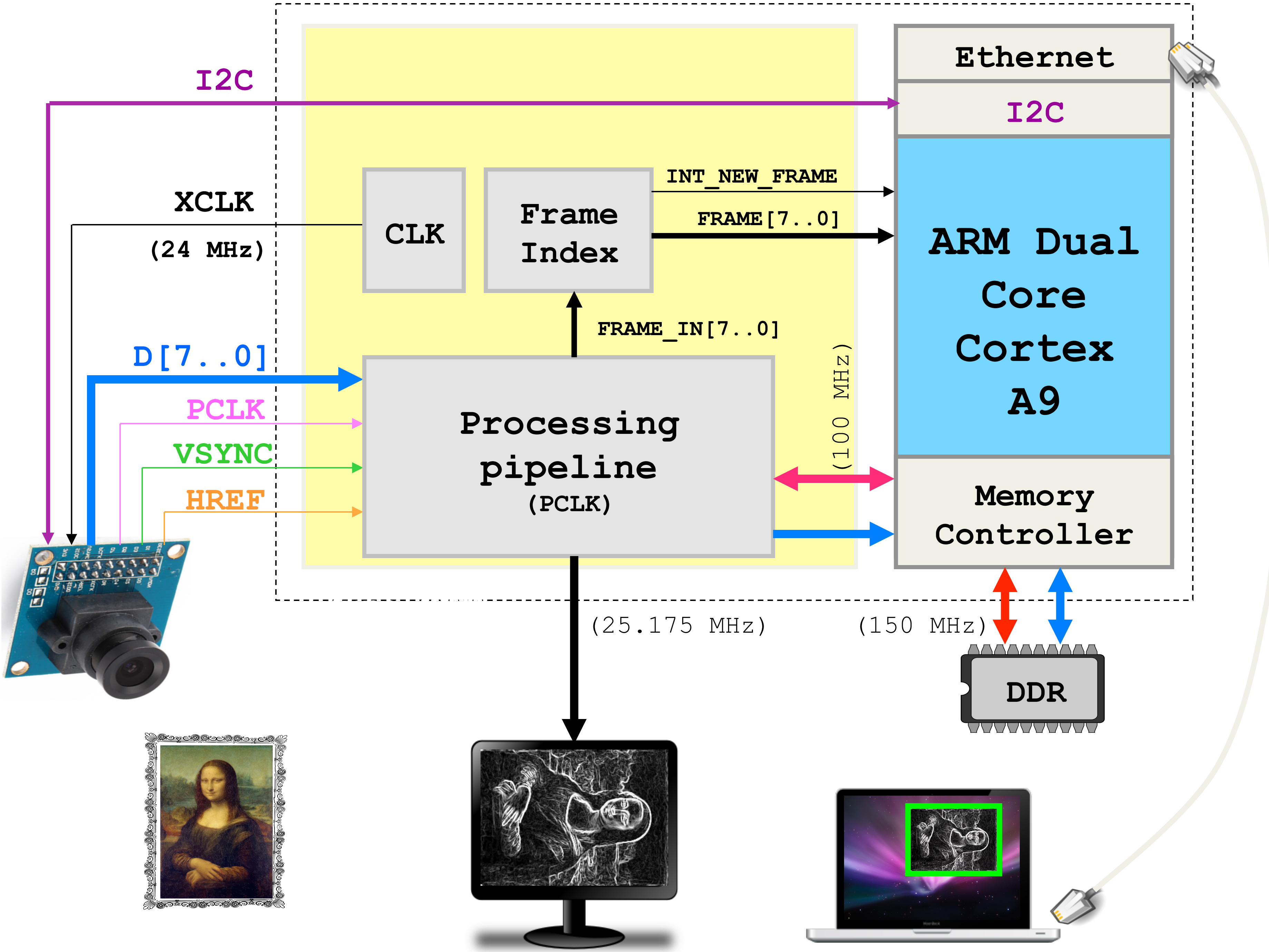


Digital imaging sensor manufactured by Omnivision:

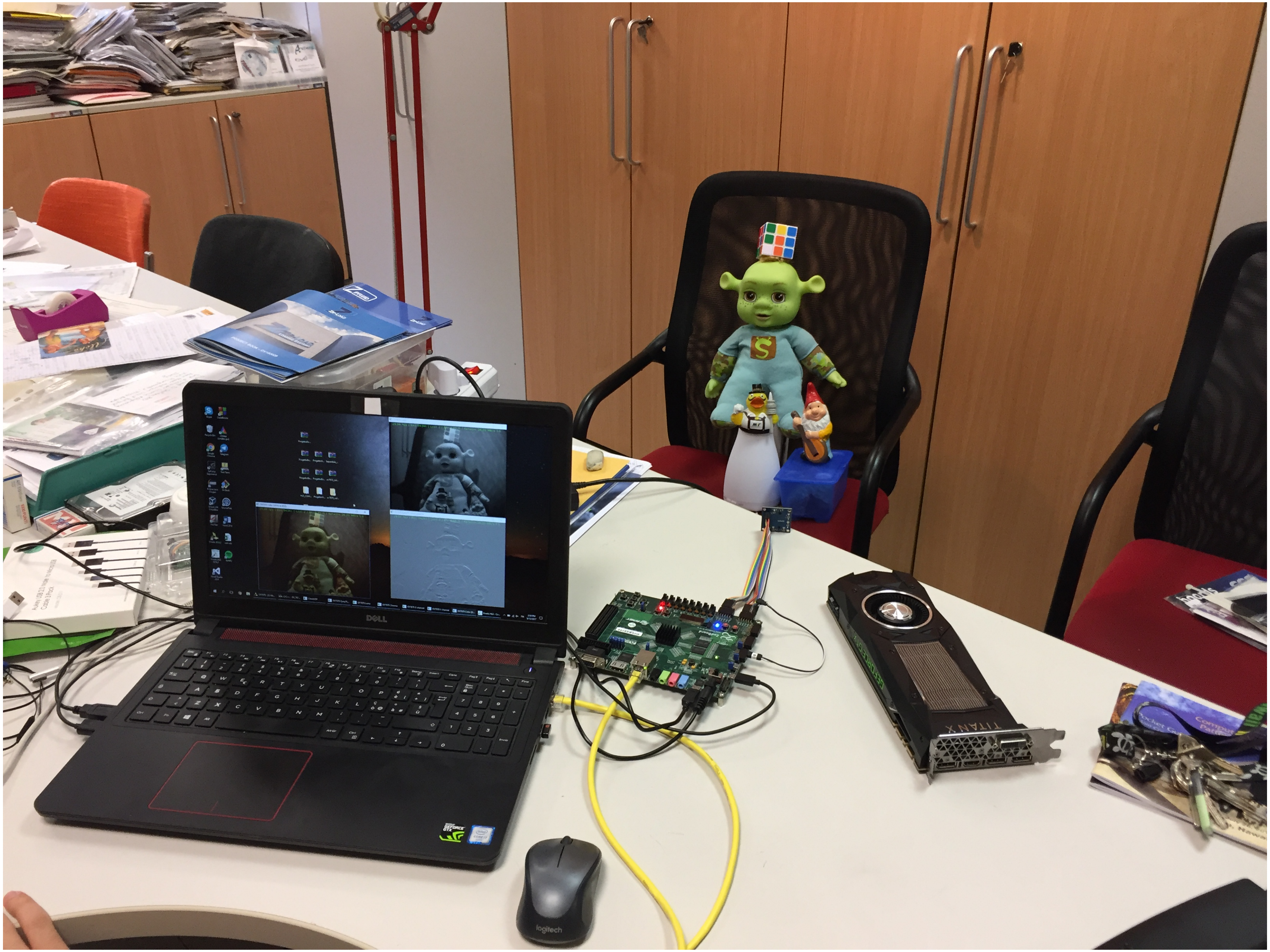
- Resolution : 640x480 (VGA), 320x240 (QVGA), etc
- Frame rate : 30 fps
- Color format : RGB, YUV (4:2:2) and YCbCr (4:2:2)
- Scan mode : rolling shutter
- Output : parallel (16 bit for YCbCr)
- Programming : I2C
- Cost : 5$

Custom embedded camera with HLS

- Entirely designed with HLS tools (HDL free)
- Agnostic to imaging sensors (tested with OV and Aptina)
- Video stream 30+ fps to a remote OpenCV PC client (UDP)
- Configurable from a remote PC client (TCP)
- VGA output for debugging
- OS: standalone (Linux in progress)
- Applications: stereo, deep learning (inference), etc

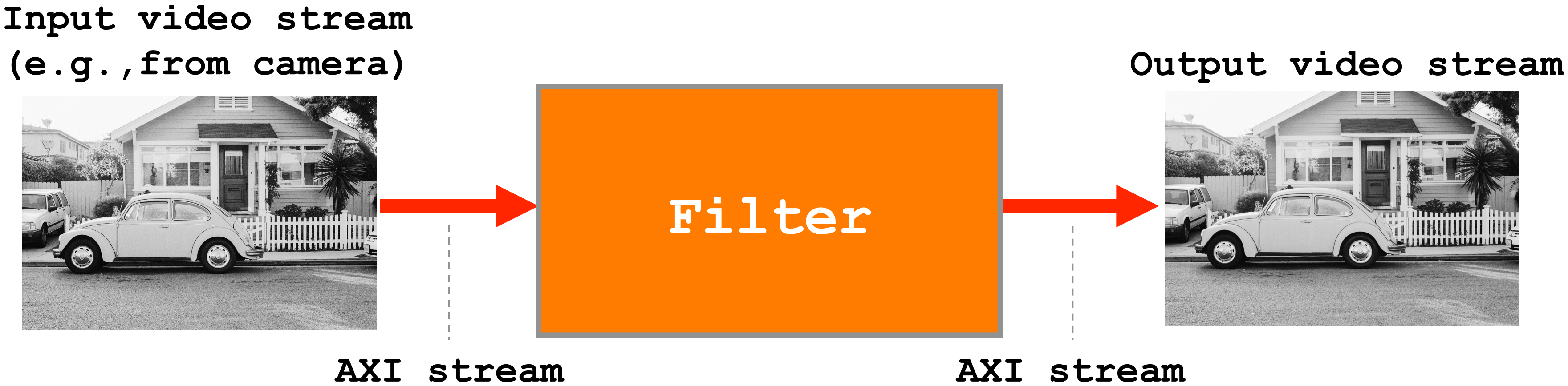


I2C
XCLK
(24 MHz)
D[7..0]
PCLK
VSYNC
HREF
CLK
Frame Index
INT_NEW_FRAME
FRAME[7..0]
FRAME_IN[7..0]
Processing pipeline
(PCLK)
(100 MHz)
Ethernet
I2C
ARM Dual Core Cortex A9
Memory Controller
(25.175 MHz)
(150 MHz)
DDR



https://www.youtube.com/watch?v=EG3NYqMJvZI

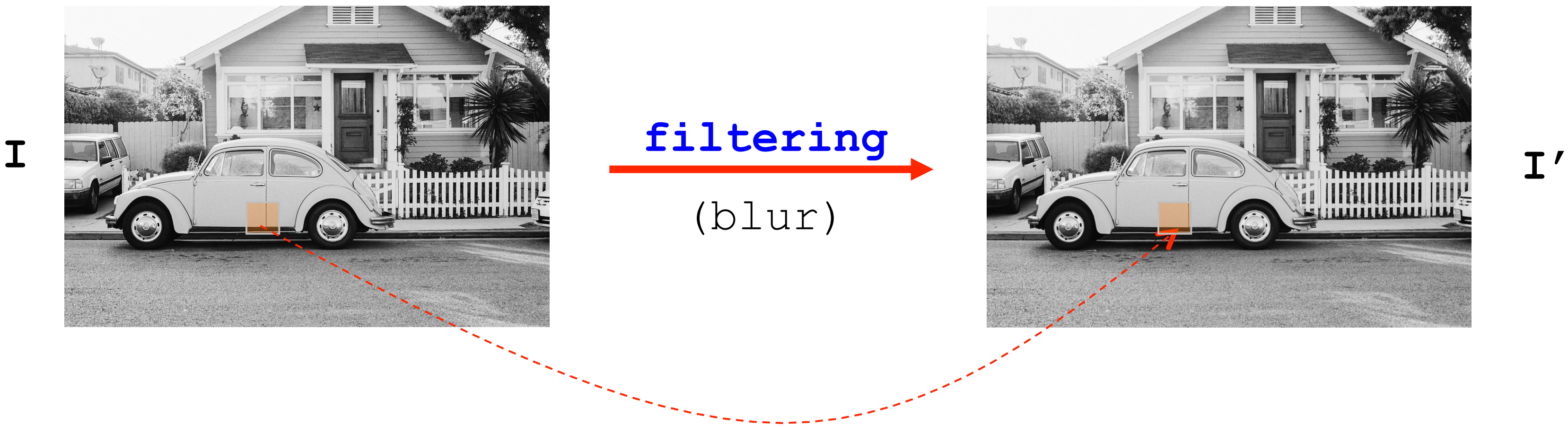
AXI stream



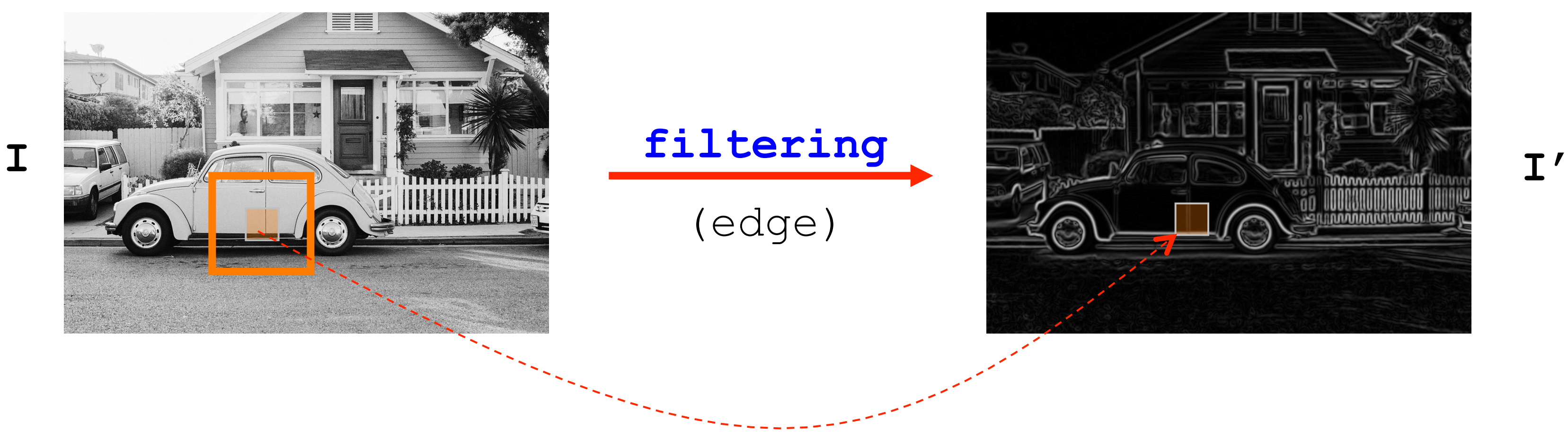

```
#include "ap_int.h"
typedef ap_uint<8> pixel;

void Filter(pixel input_img[640*480], pixel output_img[640*480])
{
#pragma HLS INTERFACE axis port=out_img
#pragma HLS INTERFACE axis port=in_img
. . .
}
```

Image filtering and convolution

- Given an input image I, filtering aims at replacing it with a more meaningful representation I′



- Often (e.g., CNN), I′[x,y] is obtained by processing a patch (<<I) centered in I[x,y]



- Often I'[x,y] is a linear combination, according to *kernel coefficients/weights*, of pixels within a patch
- This operation is known as *convolution* (operator $\otimes$)

Kernel K (N × M)

1	2	1
2	4	2
1	2	1

K $\otimes$ **I**

I (H × W)

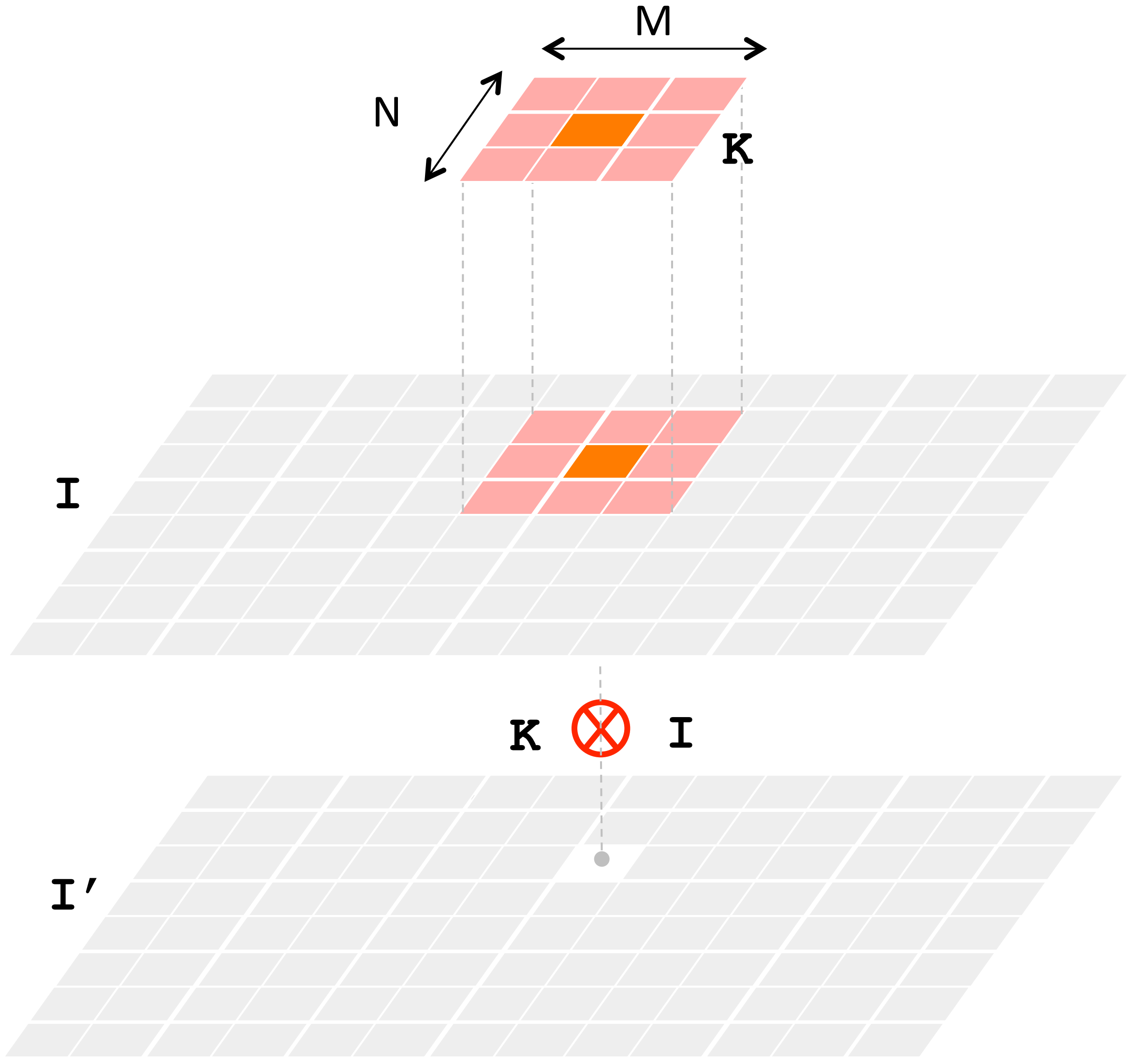
3	1	5	7	2	2	2	8	7	5	6	7
5	2	7	7	8	8	9	6	6	5	9	8
3	4	6	8	0	1	2	5	6	6	7	7
2	3	0	8	0	2	2	3	5	7	8	9
1	9	4	7	1	1	4	3	4	5	7	2
2	4	5	6	7	8	7	9	9	0	2	3

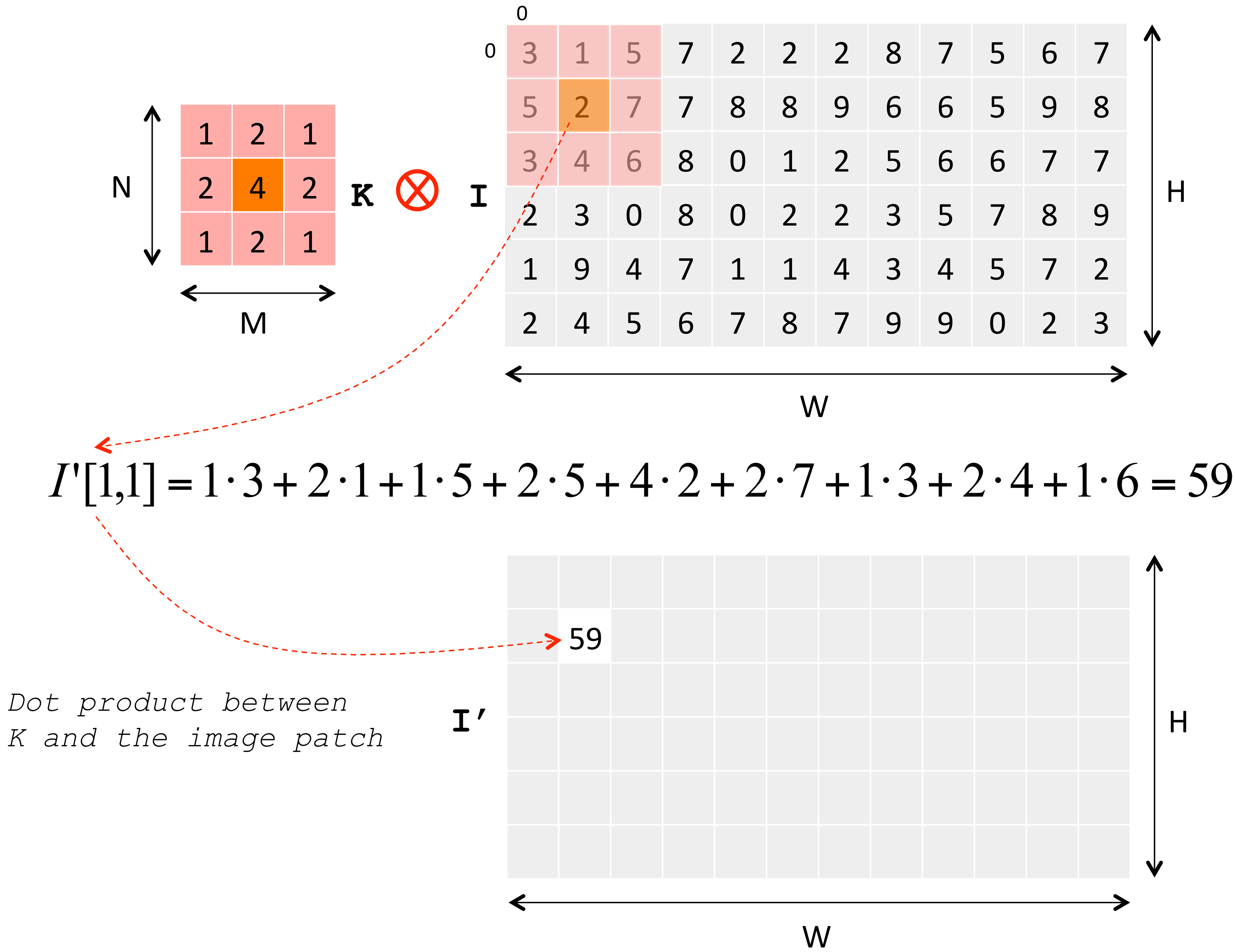
$$I'[x,y] = K[x,y] \otimes I[x,y] = \sum_{|i|<M/2, |j|<N/2} I[x-i, y-i] \cdot K[i,j]$$

$$I'[x,y] = K[x,y] \otimes I[x,y] = \sum_{i,j} I[x-i, y-i] \cdot K[i,j]$$

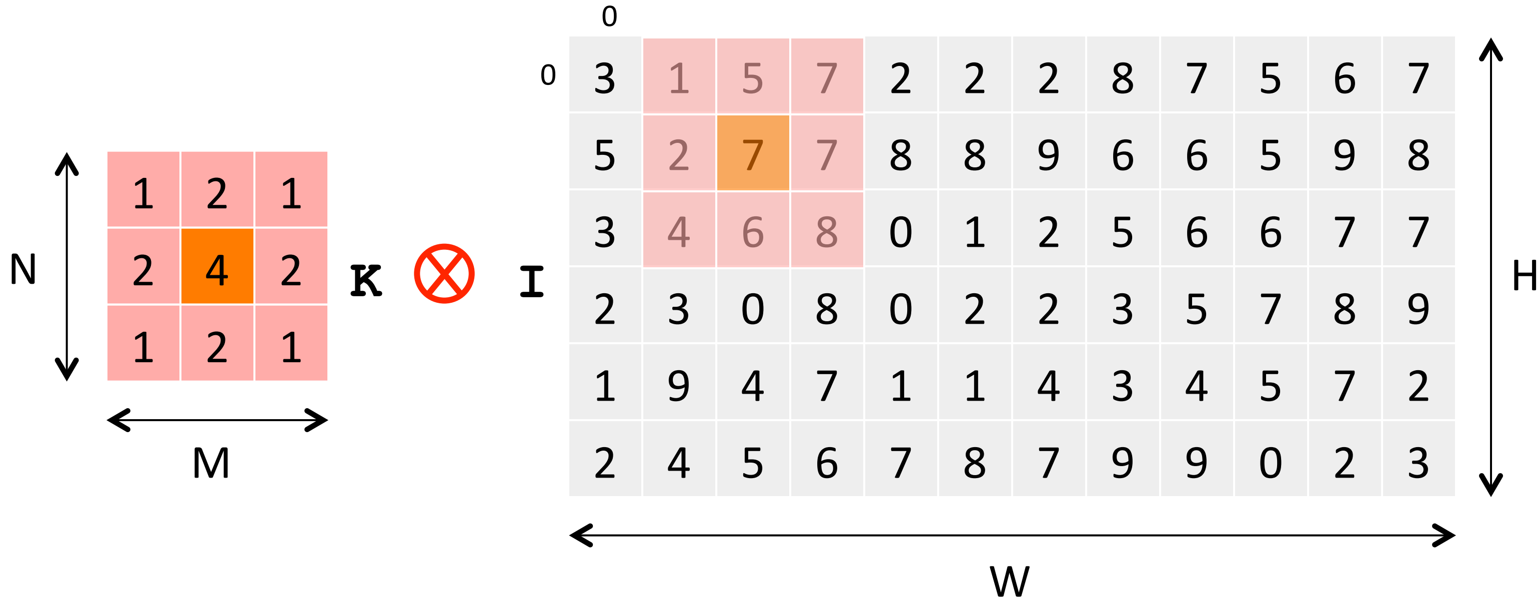
\ is the integer division

The output image I′, convolution between K and I, is obtained by *sliding* the kernel window K over all the input image I

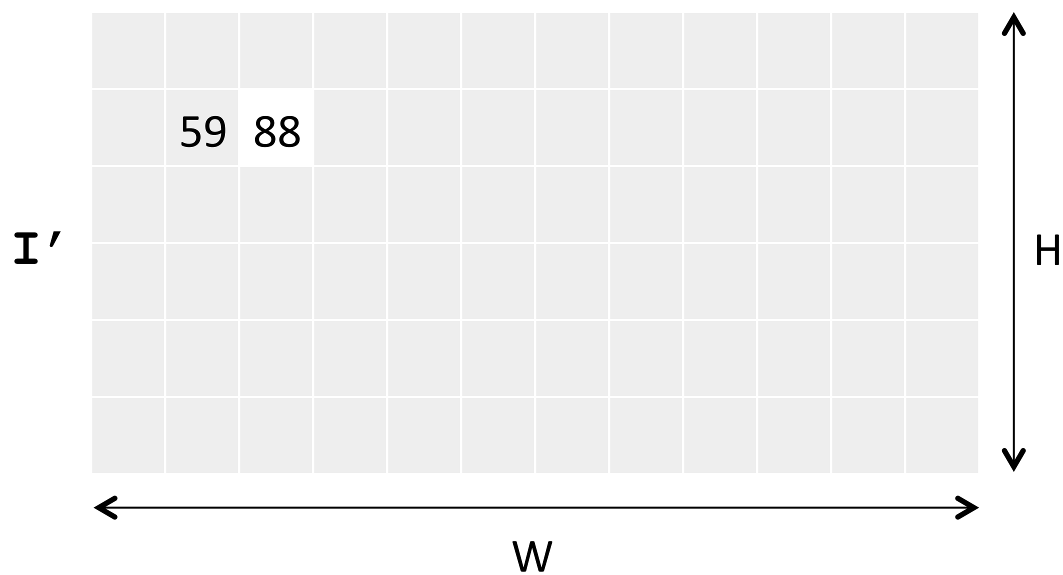


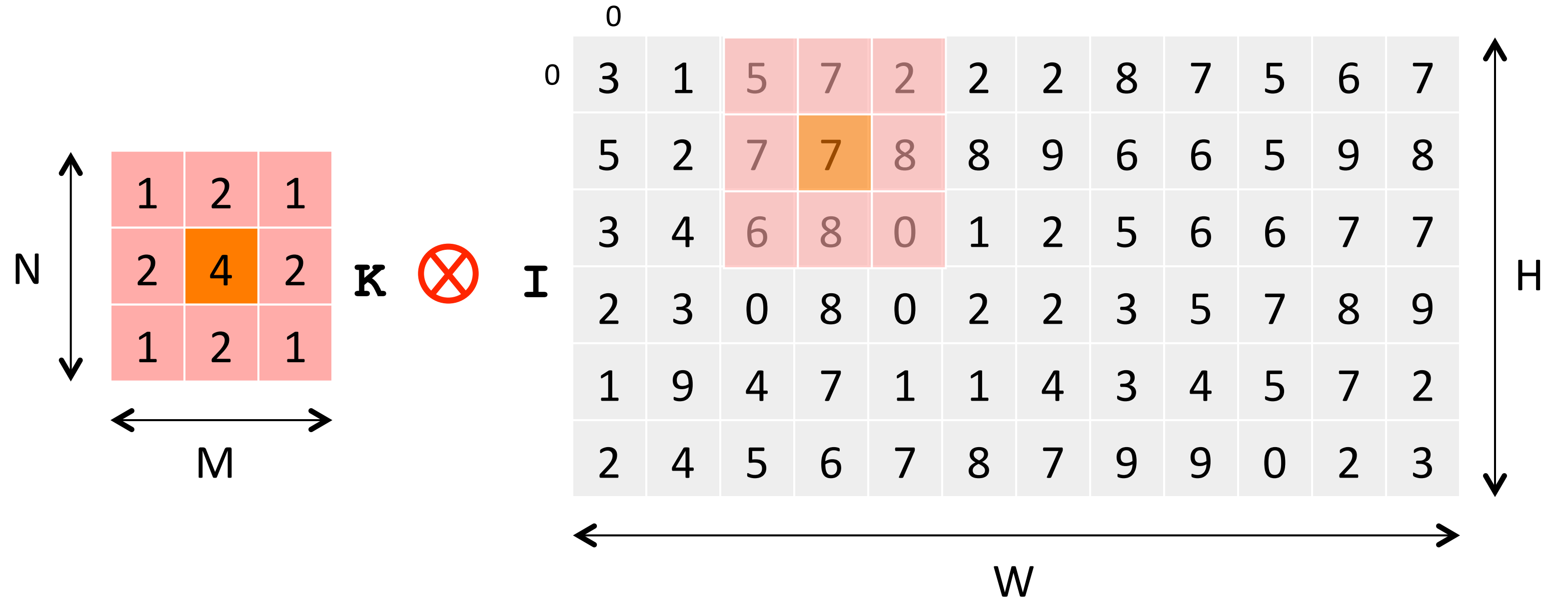


$$I'[1,1] = 1\cdot 3 + 2\cdot 1 + 1\cdot 5 + 2\cdot 5 + 4\cdot 2 + 2\cdot 7 + 1\cdot 3 + 2\cdot 4 + 1\cdot 6 = 59$$

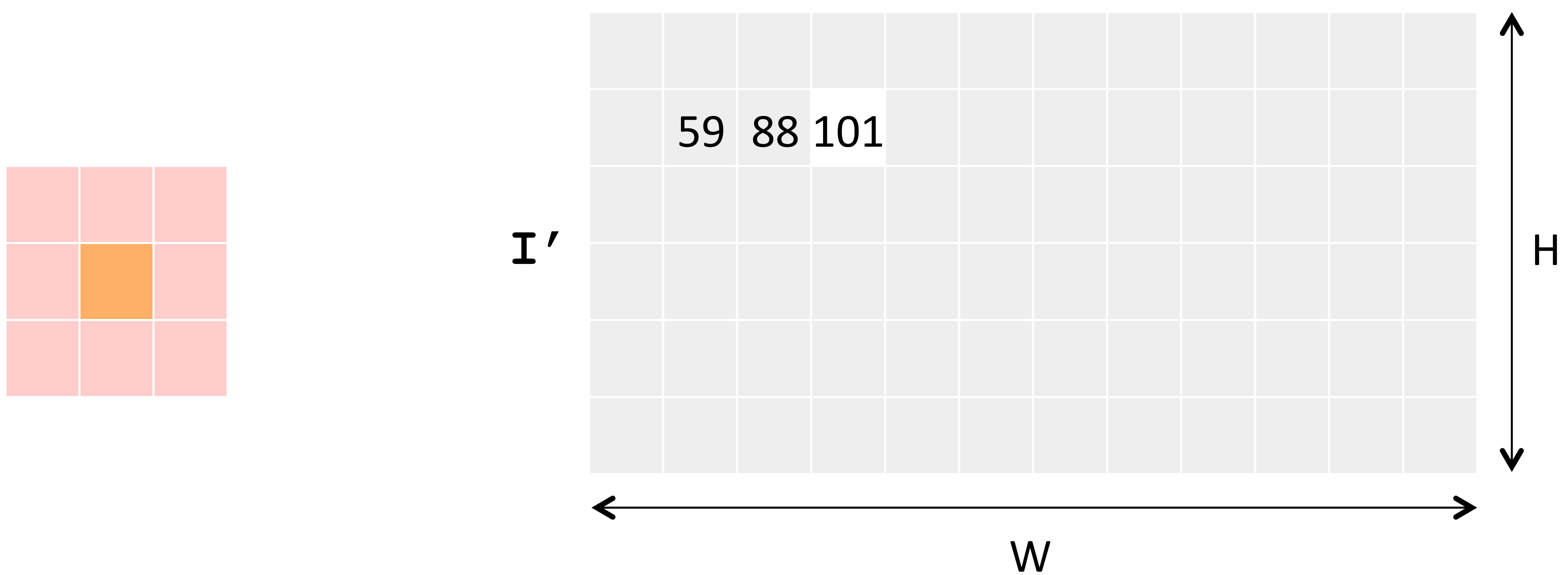


$$I'[1,2] = 1\cdot 1 + 2\cdot 5 + 1\cdot 7 + 2\cdot 2 + 4\cdot 7 + 2\cdot 7 + 1\cdot 4 + 2\cdot 6 + 1\cdot 8 = 88$$





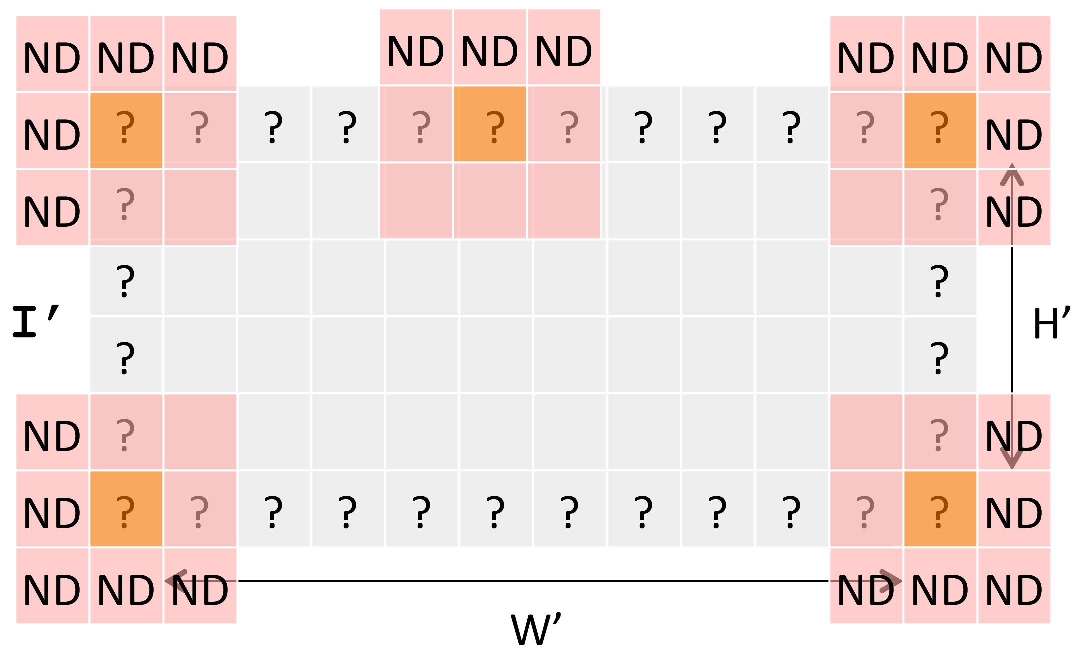
$$I'[1,3] = 1\cdot 5 + 2\cdot 7 + 1\cdot 2 + 2\cdot 7 + 4\cdot 7 + 2\cdot 8 + 1\cdot 6 + 2\cdot 8 + 1\cdot 0 = 101$$



Handling border effects

Image I′ is meaningful only for a subset of I points

ND = not defined



The actual size of I′ is:

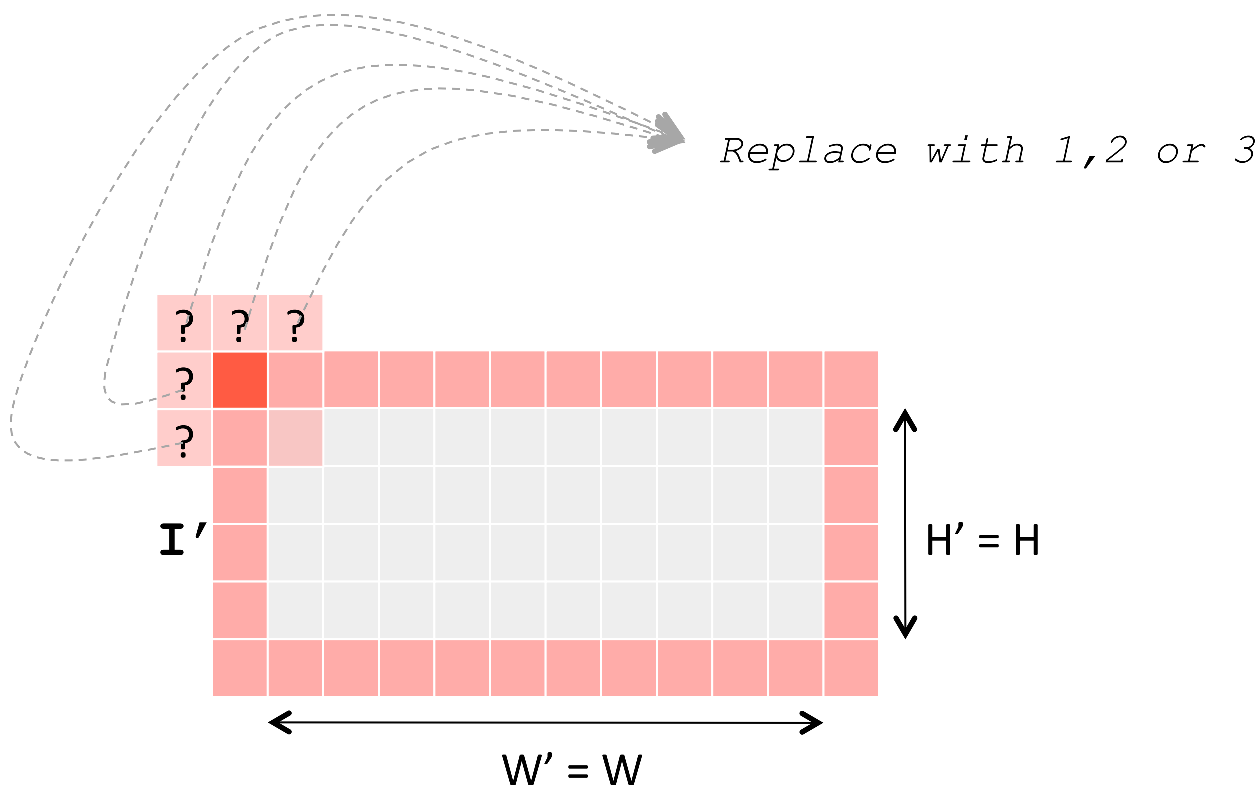
W′ = (W – 2*(M/2))x(H – 2*(N/2)) / stands for integer division

How to get rid of this problem?

- **Apply the filter on a subset o I points (smaller I')**

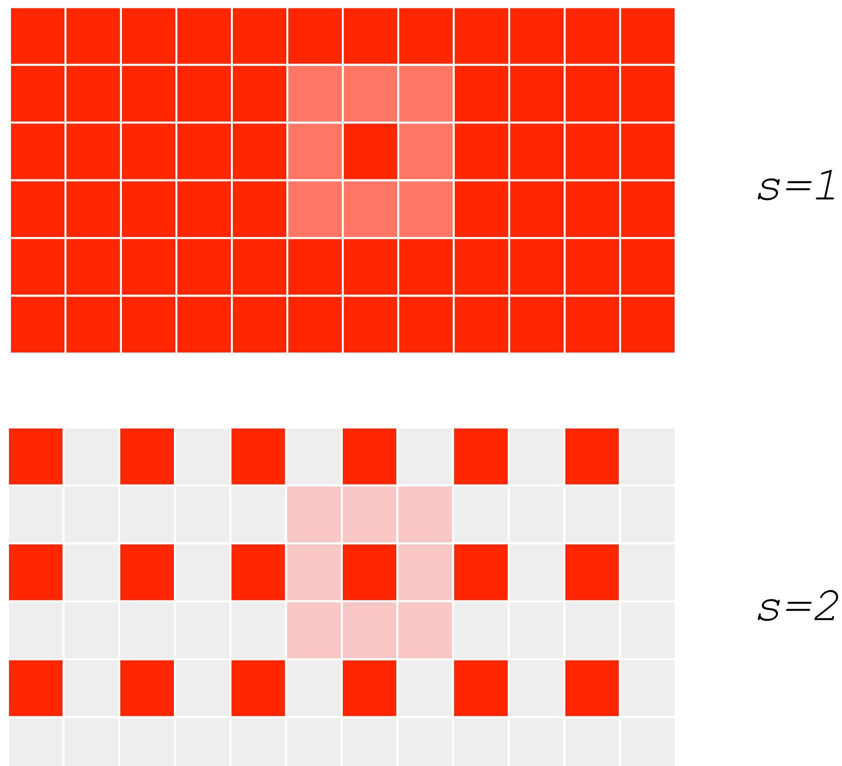
- ***R*eplace missing pixels with artificial ones (padding)**

  1. ***closest*** **pixel available**
  2. **a constant value (e.g. 0)**
  3. **randomly/not initialized values**



Stride

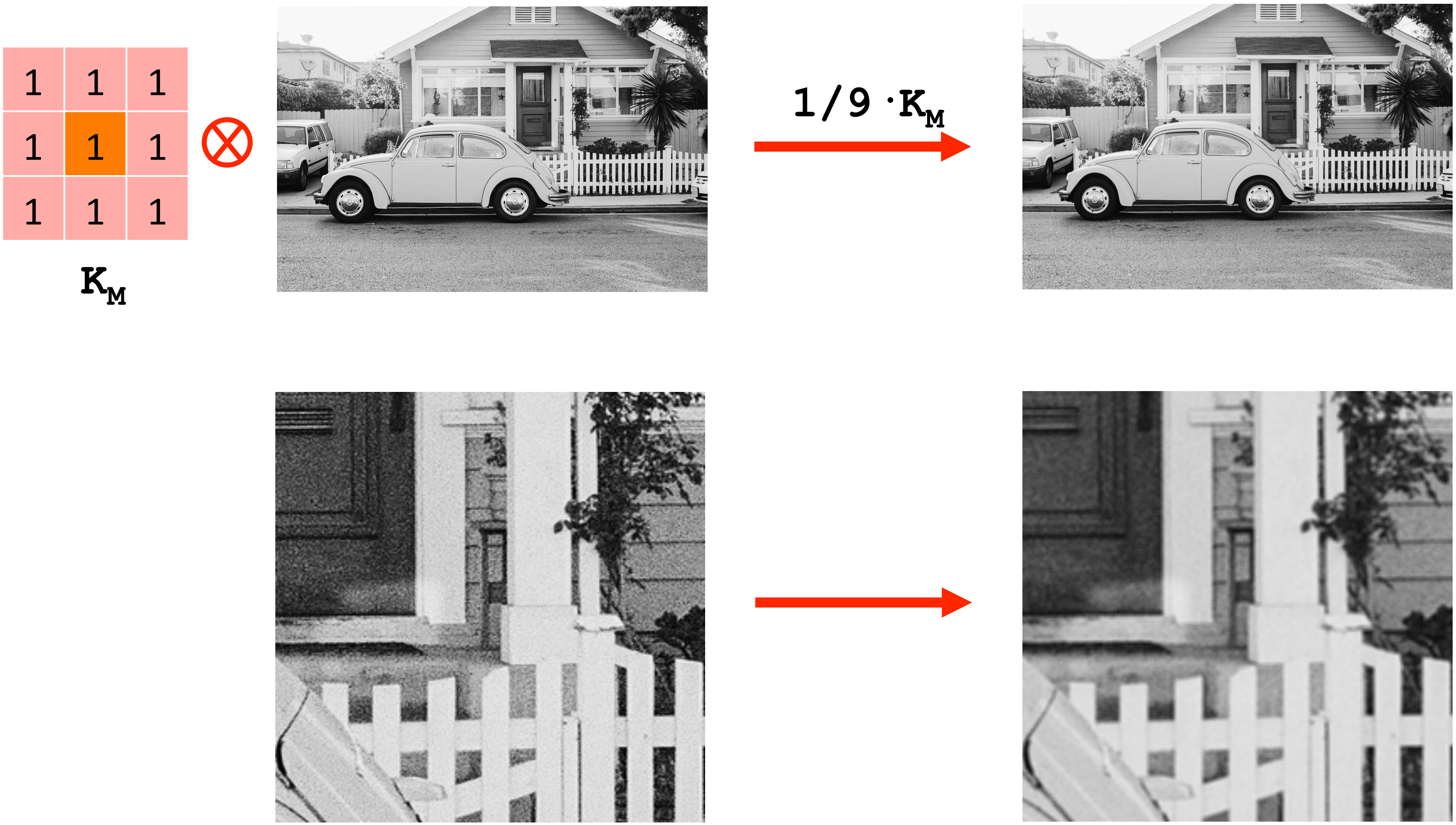
In CNNs the convolution is often applied yo a subsampled regular grid according to *stride* parameter s



Increasing the stride:

- **reduces the number of computation (by a factor s^2)**
- **reduces the size of the feature map (by the same factor)**

Mean filter



Gaussian filter

1	2	1
2	4	2
1	2	1

K_G

$\otimes$

$1/16 \cdot K_G$ →

→

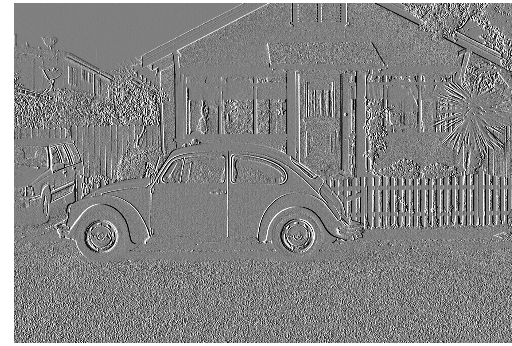
Sobel (horizontal) filter

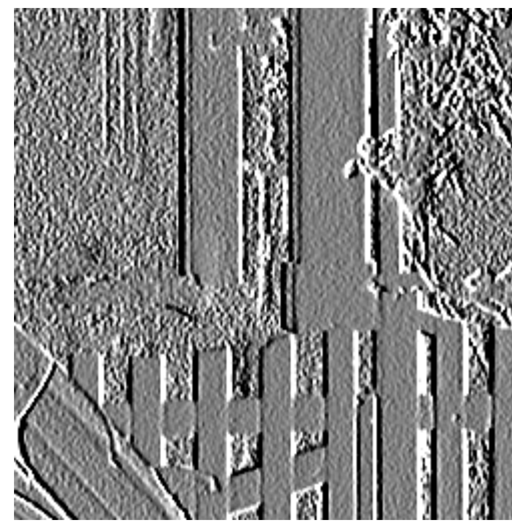
1	0	-1
2	0	-2
1	0	-1

$\otimes$

K_{SH}

$K_{SH} + 128$



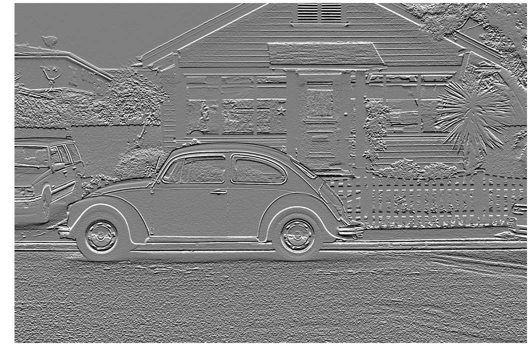


Sobel (vertical) filter

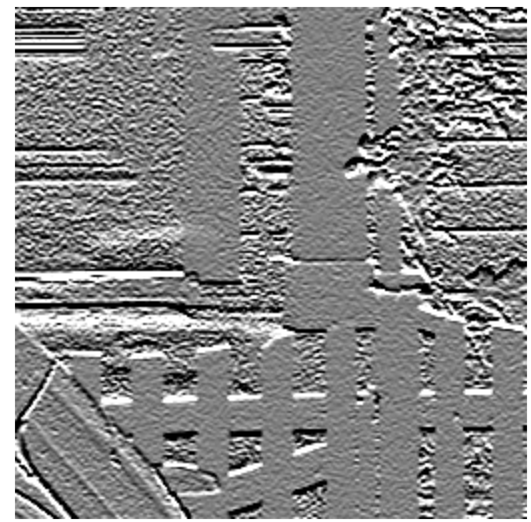
1	2	1
0	0	0
-1	-2	-1

K_{SV} $\otimes$

$K_{SV} + 128$ →



→



Convolution filter with HLS

Can be replaced with a stream (C++)

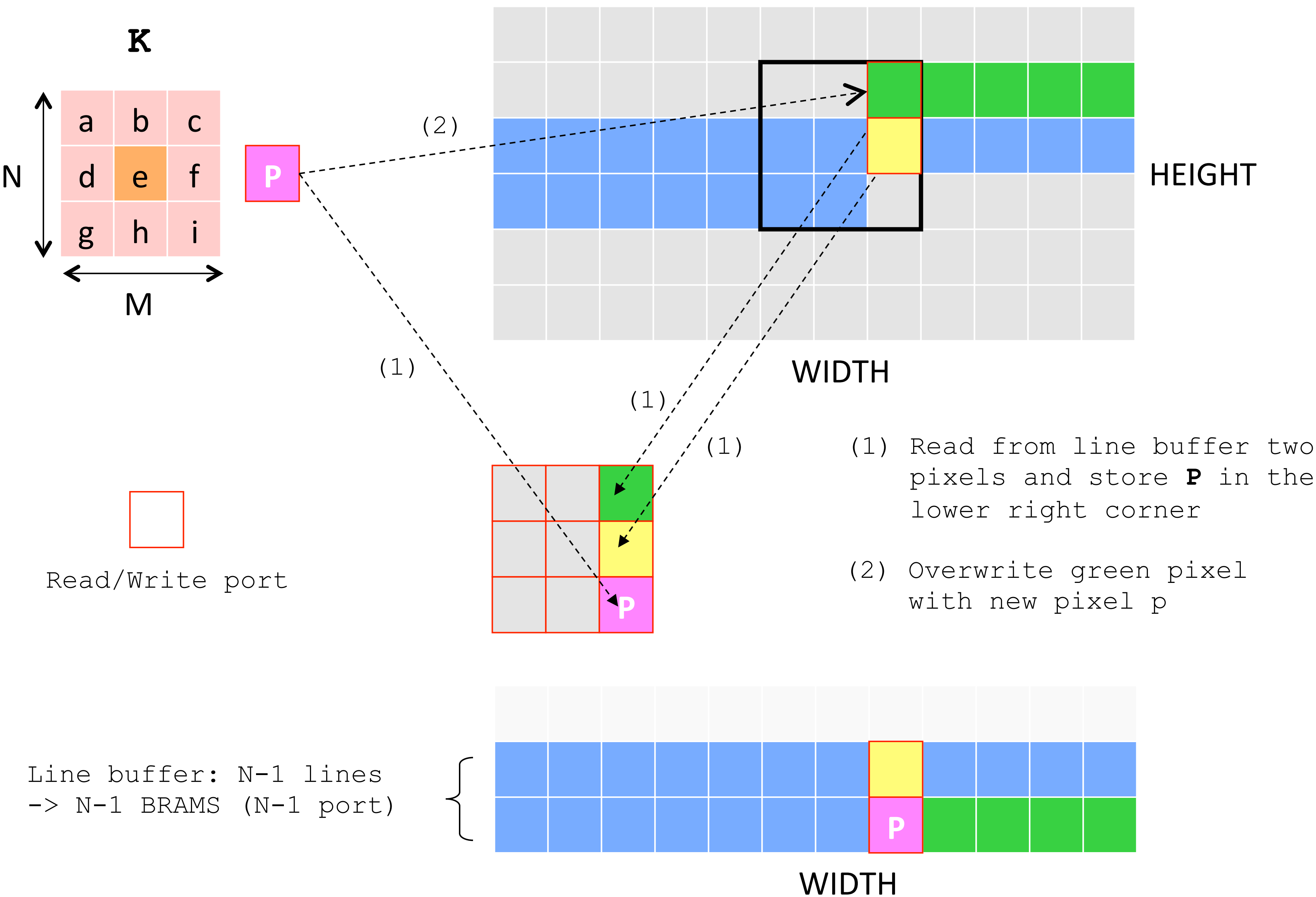
```
#include "ap_int.h"
typedef ap_uint<8> pixel;

void Filter(pixel input_img[640*480], pixel output_img[640*480])
{
#pragma HLS INTERFACE axis port=out_img
#pragma HLS INTERFACE axis port=in_img
        . . .
Loop_row: for (int row = 0; row < HEIGHT + (N-1)/2; row++)
        Loop_col: for int col = 0; col < WIDTH + (M-1)/2; col++)
        { #pragma HLS PIPELINE II=1

                // filter code here

        }
}
```

- **#pragma HLS_PIPELINE:** the sythesis tool is driven to perform inner loop computations in pipeline

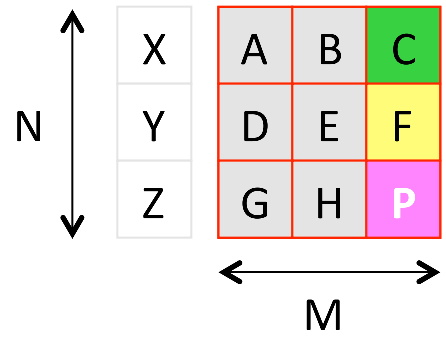
- Parameter **II=1** means that we desire to perform one iteration per clock

- Aims at improving throughput (not latency)

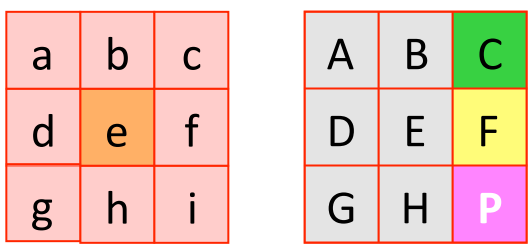
Convolution and data structures: line buffer



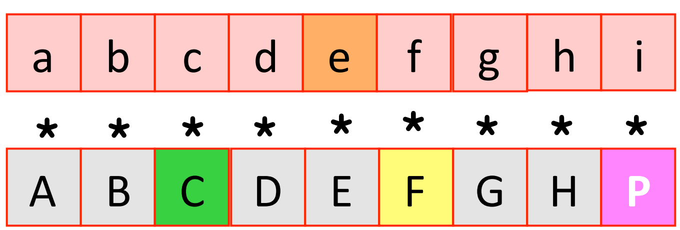
Convolution and data structures: image patch



(3) The image patch is a shift register updated with the new N pixels (rightmost column)

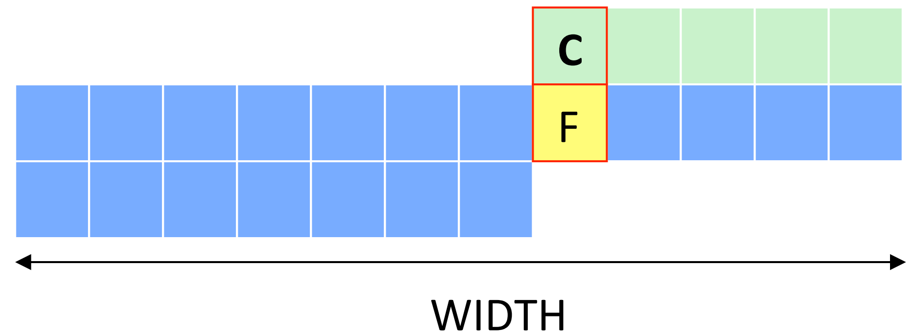


(4) Dot product: MxN parallel read (K and patch).
For both data structures MxN read ports

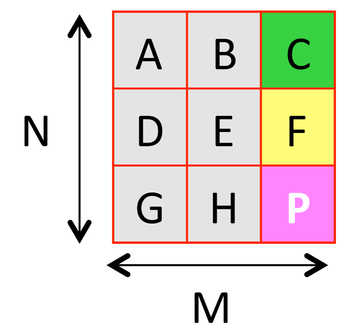


a*A + b*B + c*C + d*D + e*E + f*F + g*G + h*H + i*P

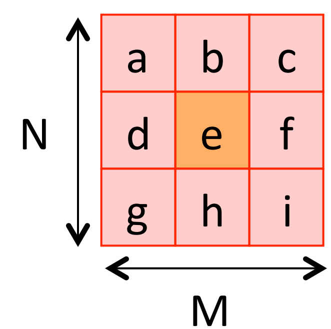
```
// line buffer
static pixel line_buffer[N - 1][IMAGE_WIDTH];
#pragma HLS ARRAY_PARTITION variable=line_buffer complete dim=1
```




```
// processing window
static pixel window[N][M];
#pragma HLS ARRAY_PARTITION variable=window complete dim=0
```

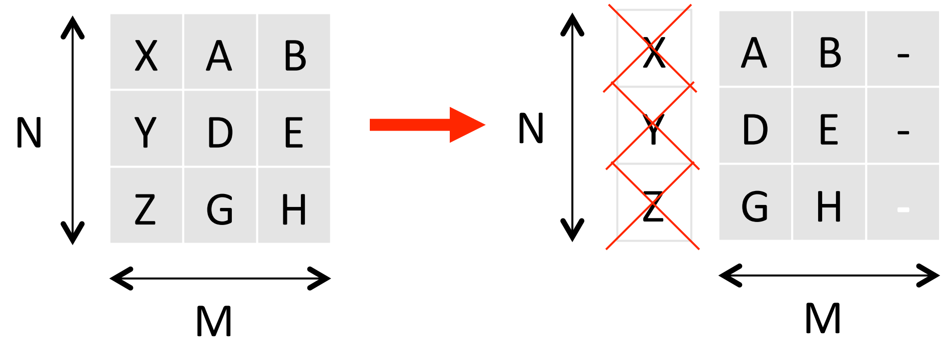



```
// kernel_config
static s_int kernel[N][M];
#pragma HLS ARRAY_PARTITION variable=kernel complete dim=0
```

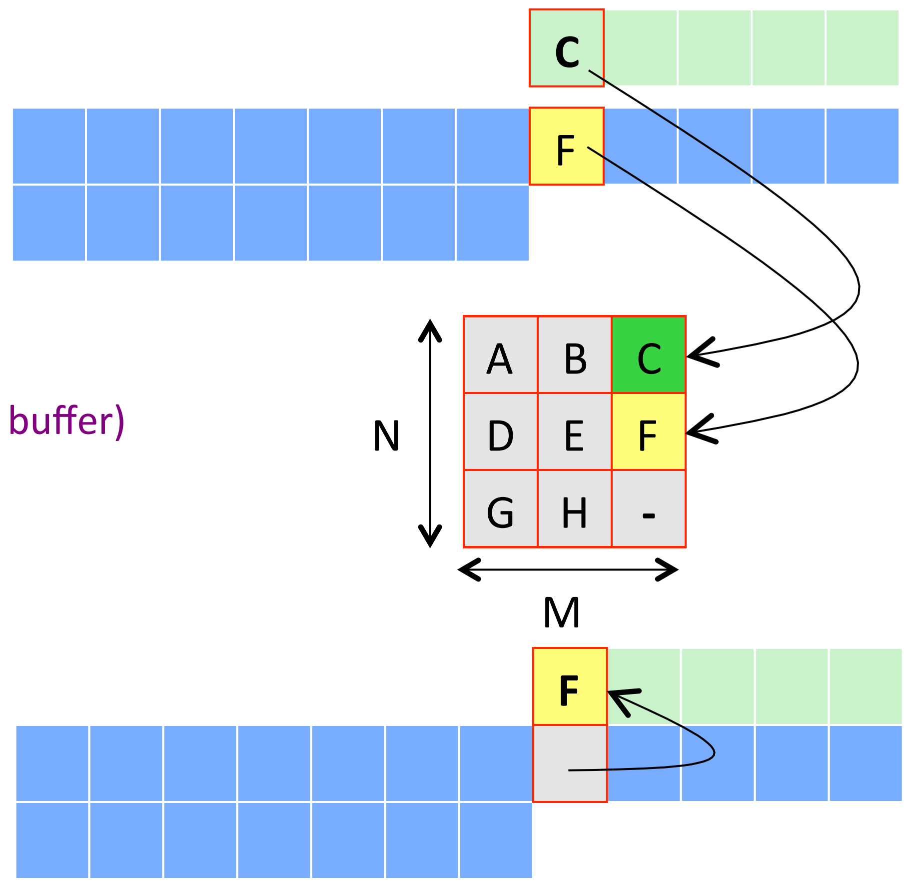


Updating data structures with HLS 1/2

```
//shift processing window columns
for (int ii = 0; ii < N; ii++)
for (int jj = 0; jj < M - 1; jj++)
            window[ii][jj] = window[ii][jj+1];
```



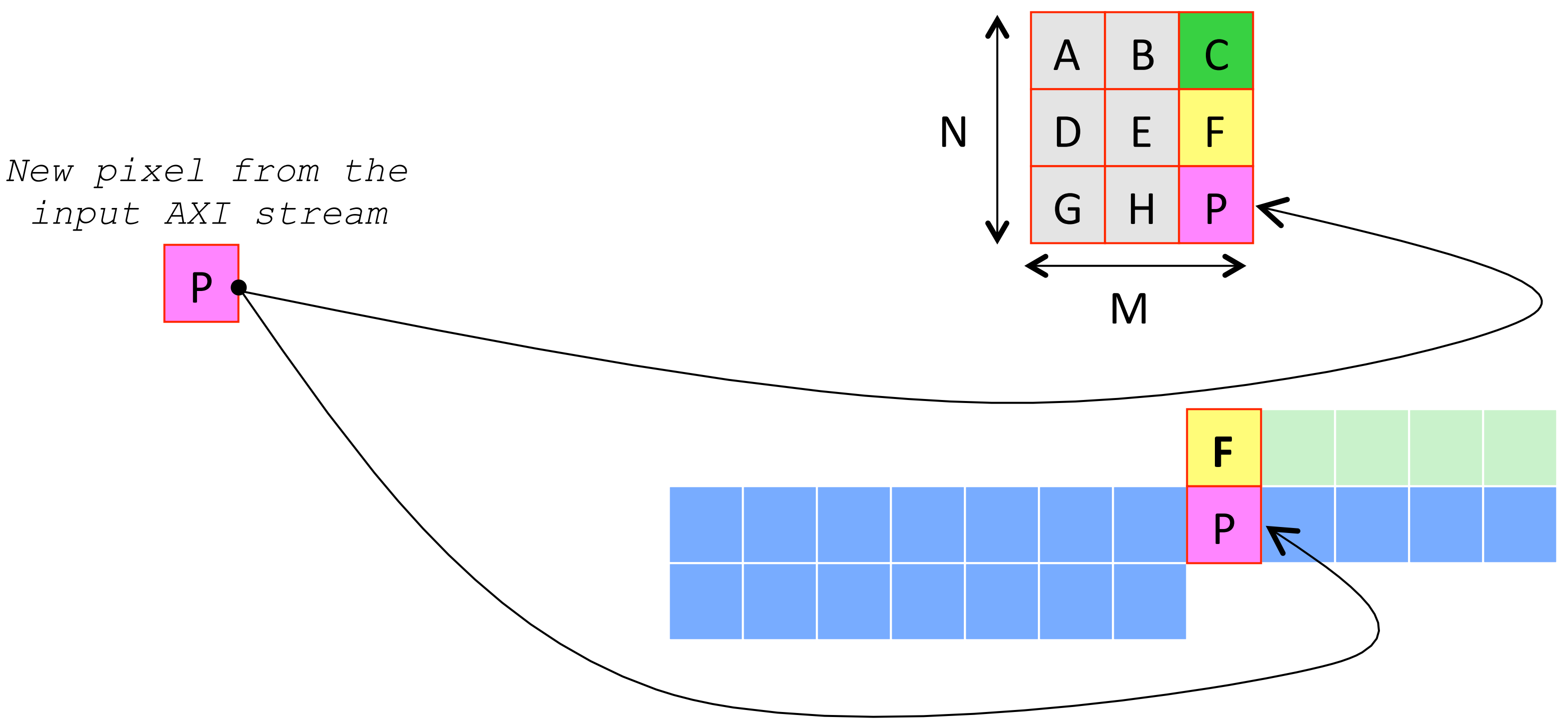

```
// copy N - 1 values from line_buffer
// to processing window and update/shift
// line buffer columns

if (col < WIDTH)  // to avoid out of bound access (line buffer)
for (int ii = 0; ii < N - 1; ii++) {
        window[ii][M- 1] = line_buffer[ii][col];

    if (ii < N - 2)
        line_buffer[ii][col] = line_buffer[ii + 1][col];
}
```



Updating data structures with HLS 2/2

Read a new pixel from input AXI stream

```
// read new input pixel, update processing window and line buffer

if (col < WIDTH && row < HEIGHT)
{
        pixel in_temp = in_img[row * WIDTH + col];
        window[KERNEL_HEIGHT - 1][KERNEL_WIDTH - 1] = in_temp;
        line_buffer[KERNEL_HEIGHT - 2][col] = in_temp;
}
```

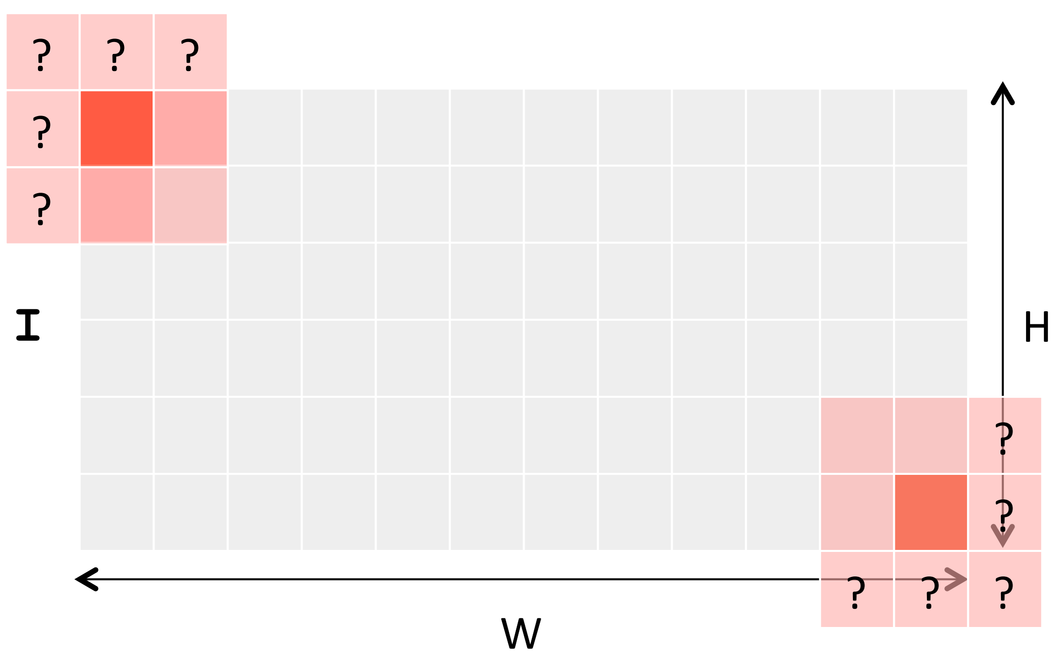


Dot product and output update with HLS 1/2

```
// compute output value
if (row >= (N-1)/2 && col >= (M-1)/2)
{
    pixel out = pixel_weighted_average(kernel, kern_sum, kern_off, window);
    out_img[(row - (N-1)/2) * IMG_WIDTH + (col - (N-1)/2)] = out;
}
```

Write a new pixel into the output AXI stream

'If' and 'out_img' indexes enable to *handle* (as better as possibile) border effects



Dot product and output update with HLS 2/2

```
pixel pixel_weighted_average(s_int kernel[N][M],
                             s_int kern_sum,
                             s_int kern_off,
                             pixel window[N][N])
{
#pragma HLS INLINE

        ap_int<MAC_BITS> out_temp = 0;
        ap_int<MUL_BITS> temp = 0;

#pragma HLS RESOURCE variable=temp core=Mul_LUT

        // dot product
        Edge_i:  for (int i = 0; i < N; i++)
                Edge_j: for (int j = 0; j < M; j++){
                        temp = window[i][j] * kernel[i][j];
                        out_temp = out_temp + temp;
                }

        // update output, normalize and add offset
        return ((out_temp / kern_sum) + kern_off)(7,0);
}
```

Computational complexity

For each point:

- **MxN multiplications**
- **MxN-1 additions**

Overall: ≈ WxHxMxN multiplications and additions

Complexity grows according to MxN

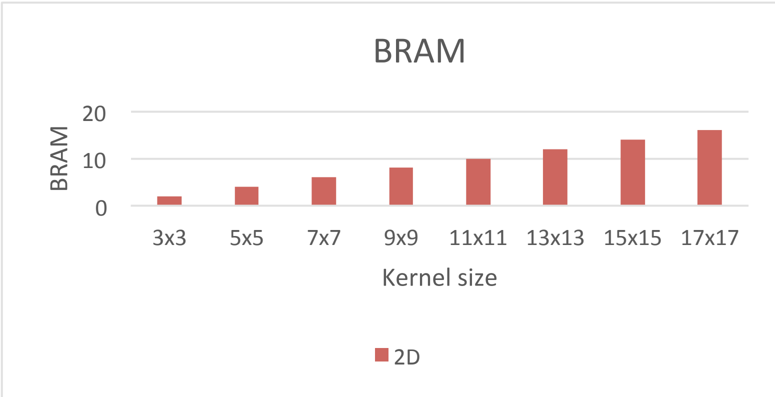
With VGA images and M=N=3:

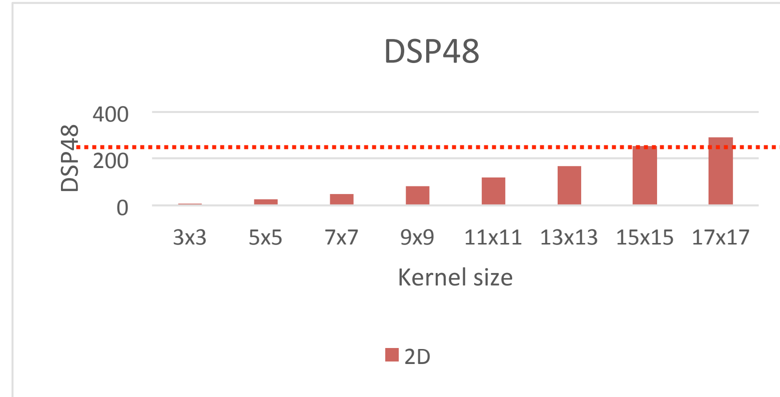
640x480x9x9 ≈ 2.7 M operations

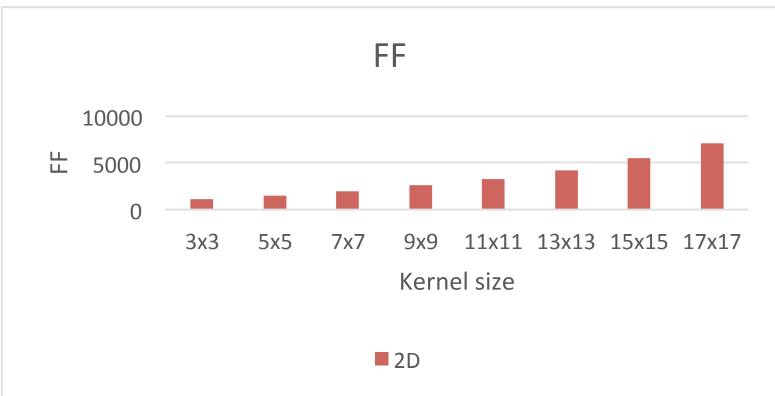
With VGA images and M=N=9:

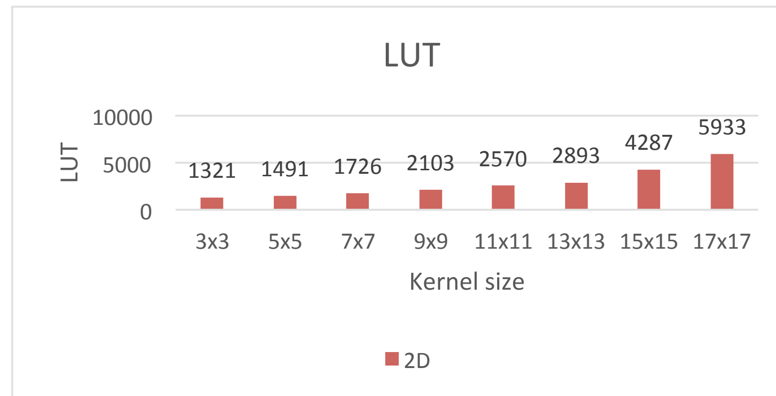
640x480x9x9 ≈ 25 M operations

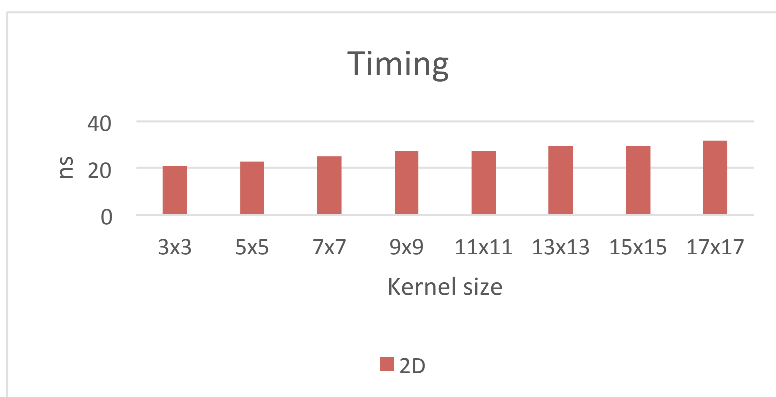
Implementation report





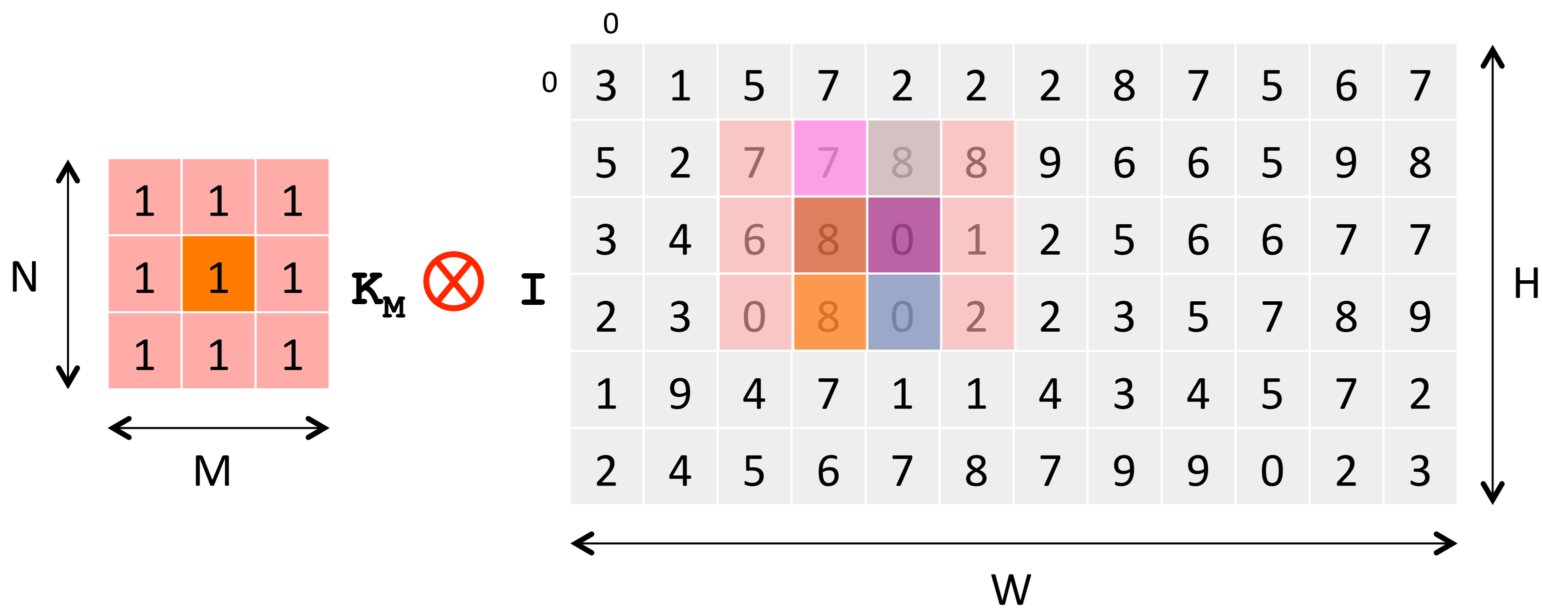






How to reduce the number of computations?

Let's consider the **mean filter K_M**: it can be efficiently computed exploiting redundancy within overlapping regions



1	1	1
1	1	1
1	1	1

3	1	5	7	2	2	2	8	7	5	6	7
5	2	7	7	8	8	9	6	6	5	9	8
3	4	6	8	0	1	2	5	6	6	7	7
2	3	0	8	0	2	2	3	5	7	8	9
1	9	4	7	1	1	4	3	4	5	7	2
2	4	5	6	7	8	7	9	9	0	2	3

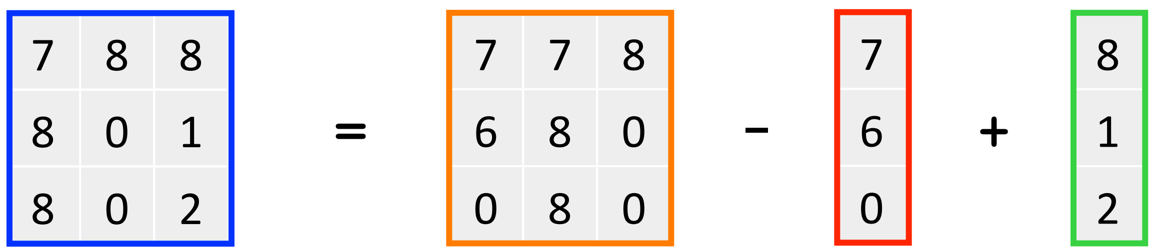
$$I'[2,3] = 1\cdot 7 + 1\cdot 7 + 1\cdot 8 + 1\cdot 6 + 1\cdot 8 + 1\cdot 0 + 1\cdot 0 + 1\cdot 8 + 1\cdot 0 = 44$$

$$I'[2,4] = 1\cdot 7 + 1\cdot 8 + 1\cdot 8 + 1\cdot 8 + 1\cdot 0 + 1\cdot 1 + 1\cdot 8 + 1\cdot 0 + 1\cdot 2 = 42$$

Box-filtering

I

3	1	5	7	2	2	2	8	7	5	6	7
5	2	7	7	8	8	9	6	6	5	9	8
3	4	6	8	0	1	2	5	6	6	7	7
2	3	0	8	0	2	2	3	5	7	8	9
1	9	4	7	1	1	4	3	4	5	7	2
2	4	5	6	7	8	7	9	9	0	2	3



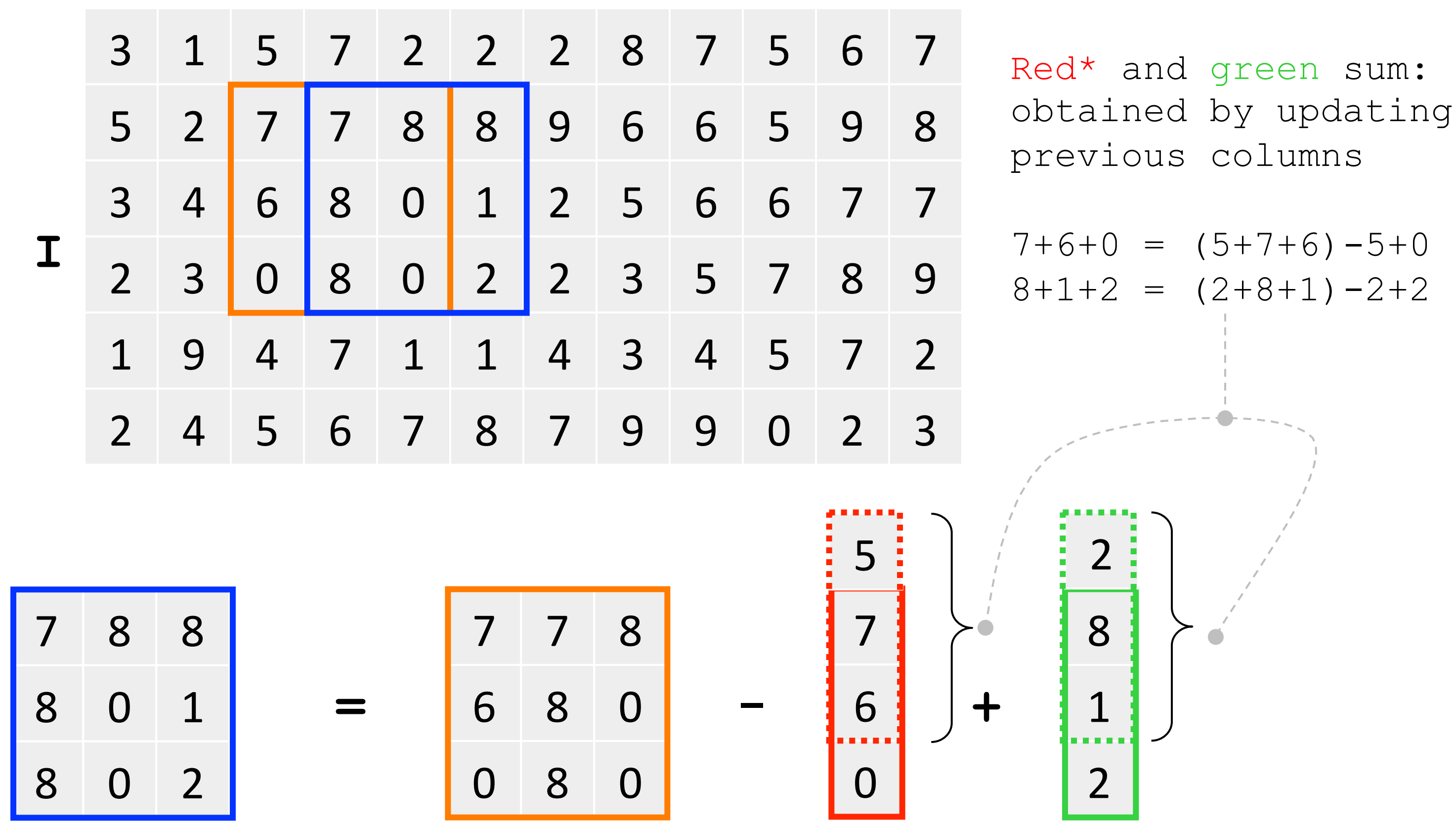
Next point: I'(2,3)=?

I'(2,3)=44 already computed

The number of operations is now **6** (**vs 9** of the 2D version)

Buffering: previous I' (a single value) along the scanline

Can we do better?



Next point: I’(2,3)=?

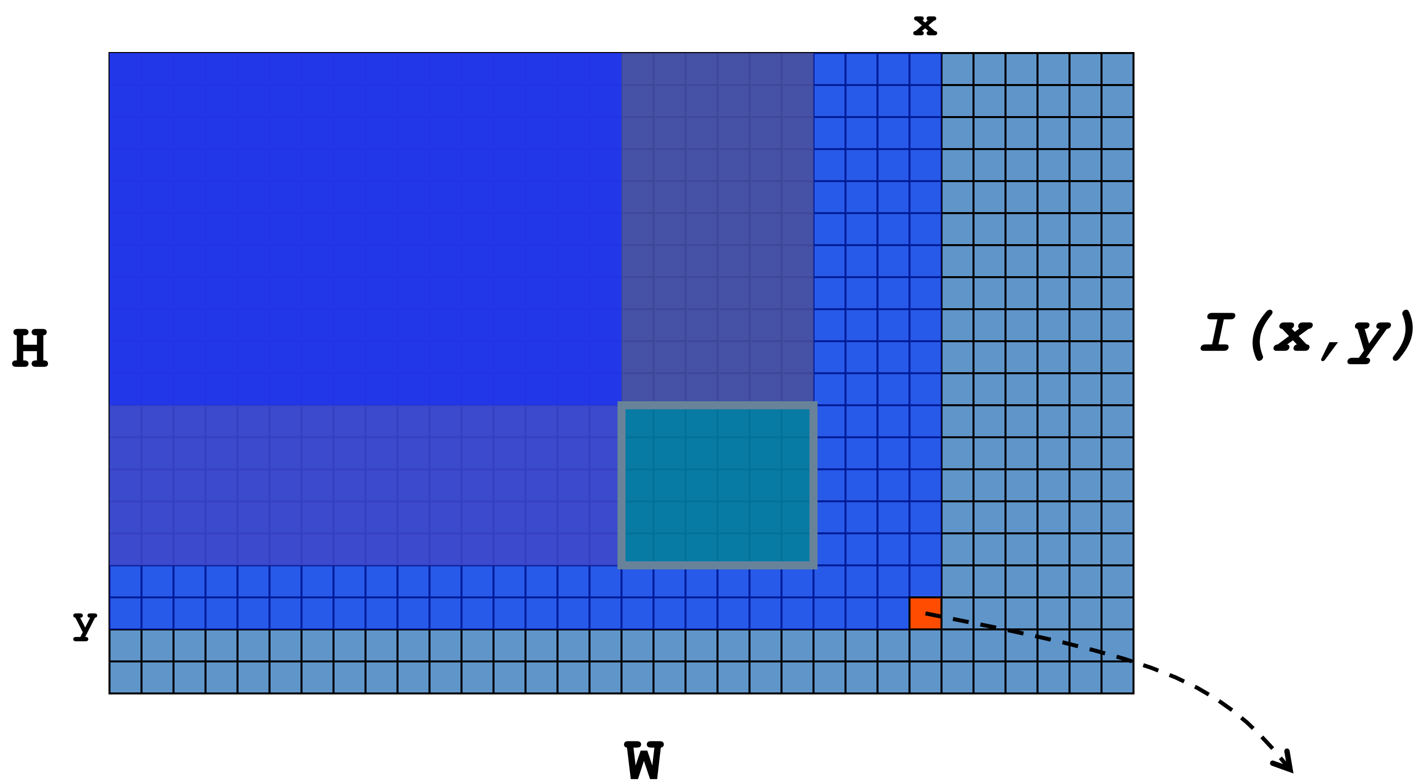
I’(2,3)=44 already computed

The number of operations is now **4** and **constant** (**vs 9** of 2D)

Buffering: ≈ previous I’ (a single value) along the scanline plus previous sum of columns (1 row of size W)

* Already computed (with the same updating strategy) at I’[2,1]

Integral Images (aka Summmed Area Table)



$$S(x,y) = \sum_{i<x,\, j<y} I(i,j)$$

$$S^2(x,y) = \sum_{i<x,\, j<y} I^2(i,j)$$

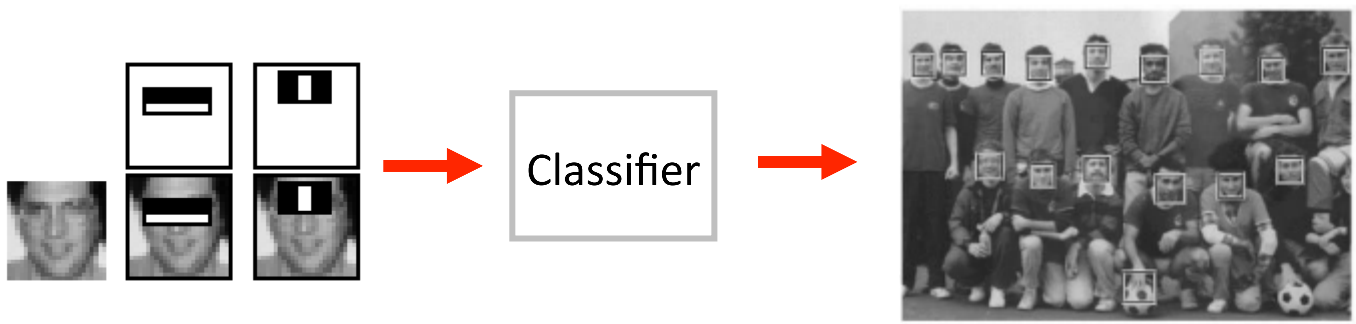
[Crow], Summed-area tables for texture mapping, Computer Graphics, 18(3):207–212, 1984

[Viola and Jones], Rapid object detection using a boosted cascade of simple features, CVPR 2001

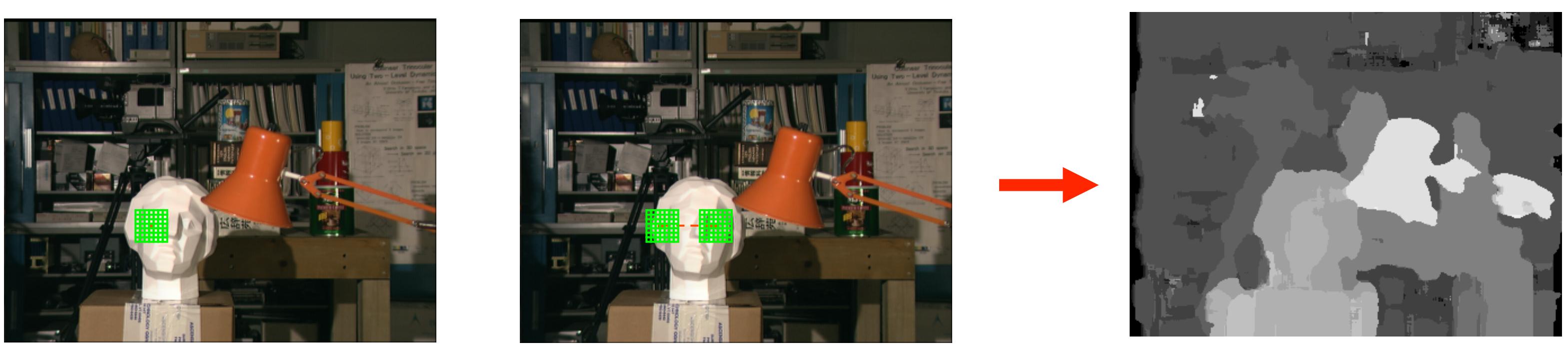
Unfortunately these constant time techniques do not apply to generic convolution kernels

Nevertheless, are both very popular for many computer vision tasks:

- Face detection [Viola-Jones]: integral images



- Stereo vision (Block Matching): box-filtering



Reference (R) Target (T) Disparity map

[Viola and Jones], Rapid object detection using a boosted cascade of simple features, CVPR 2001

How to reduce the number of computations of a generic convolution filter?

- Moving to the frequency domain: **Fourier transform**
  - equivalent results
  - efficient only for *larger* kernels

- Using approximation strategies such as **separable filters**
  - not always equivalent results
  - acceptable approximation adopting appropriate strategies
  - simple and efficient implementation on embedded systems including FPGAs

Separable filters (rank 1)

- A filter is separable if its kernel **K** can be obtained as the product of two vectors **a** e **b**
- In this case: $\mathbf{I'} = \mathbf{K} \otimes \mathbf{I} = \mathbf{a} \otimes \mathbf{b} \otimes \mathbf{I}$
- This constraint holds if the kernel has rank 1

Example: Mean, Gaussian and the Sobel filters are separable:

$$\mathbf{K_M} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} = \underbrace{\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}}_{\mathbf{a}} \underbrace{\begin{bmatrix} 1 & 1 & 1 \end{bmatrix}}_{\mathbf{b}}$$

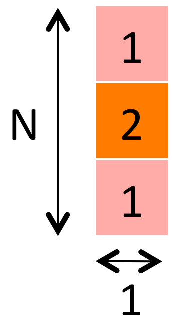
$$\mathbf{K_G} = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 4 & 1 \end{bmatrix} = \underbrace{\begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}}_{\mathbf{a}} \underbrace{\begin{bmatrix} 1 & 2 & 1 \end{bmatrix}}_{\mathbf{b}}$$

$$\mathbf{K_{SH}} = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} = \underbrace{\begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}}_{\mathbf{a}} \underbrace{\begin{bmatrix} 1 & 0 & -1 \end{bmatrix}}_{\mathbf{b}}$$

$$\mathbf{K_{SV}} = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} = \underbrace{\begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}}_{\mathbf{a}} \underbrace{\begin{bmatrix} 1 & 2 & 1 \end{bmatrix}}_{\mathbf{b}}$$

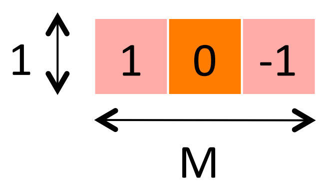
- For separable filters the 2D convolution can be replaced with two distinct 1D convolutions (associtive property):

1) Convolve the image I with 1D kernel **a** (or **b**, commutative)

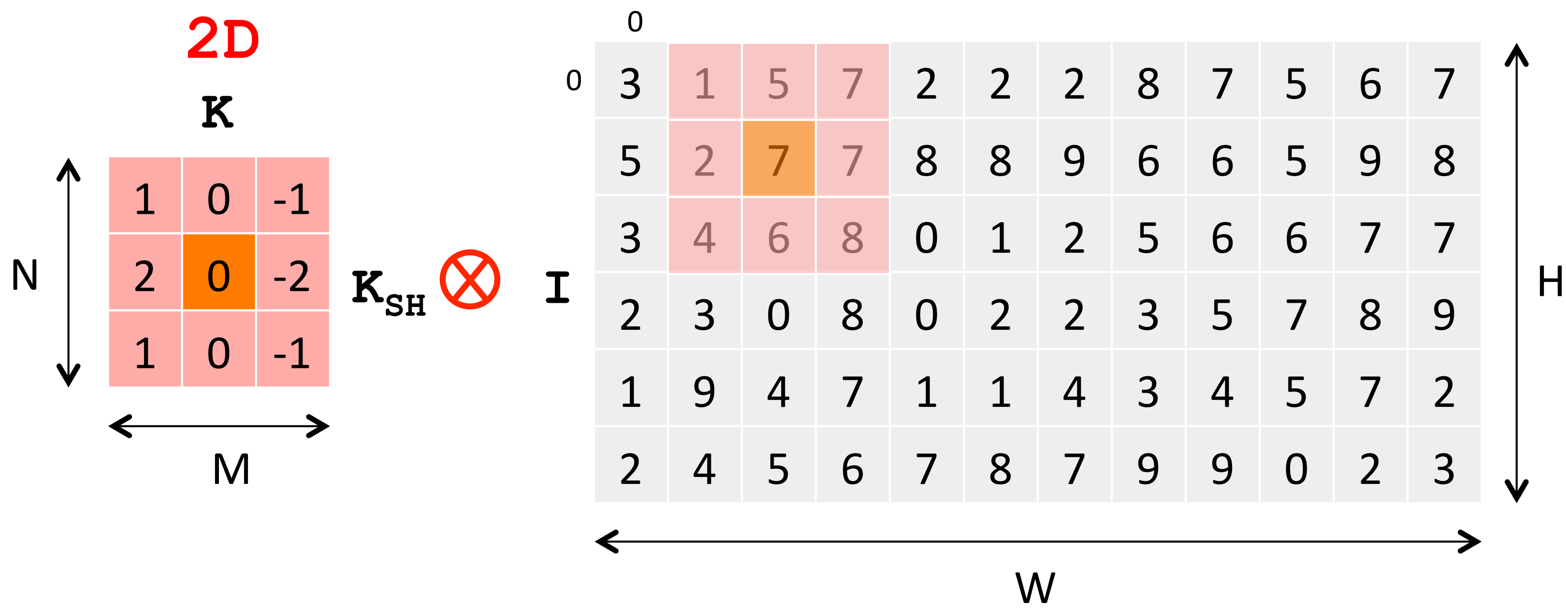


$$\mathbf{I}_{\mathbf{TEMP}} = \mathbf{a} \otimes \mathbf{I}$$

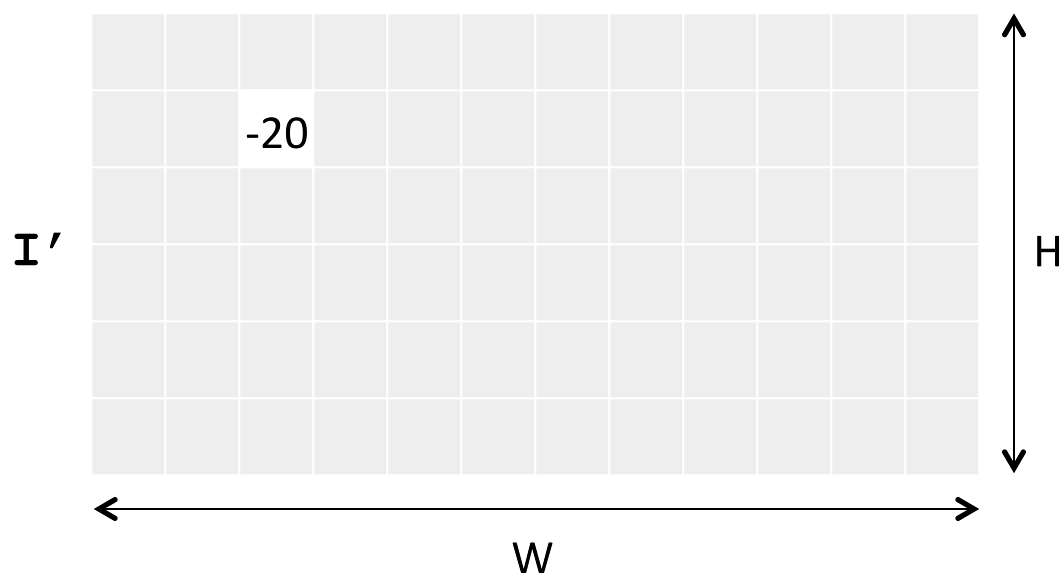
2) Convolve the result with 1D kernel **b** (or **a**, commutative)



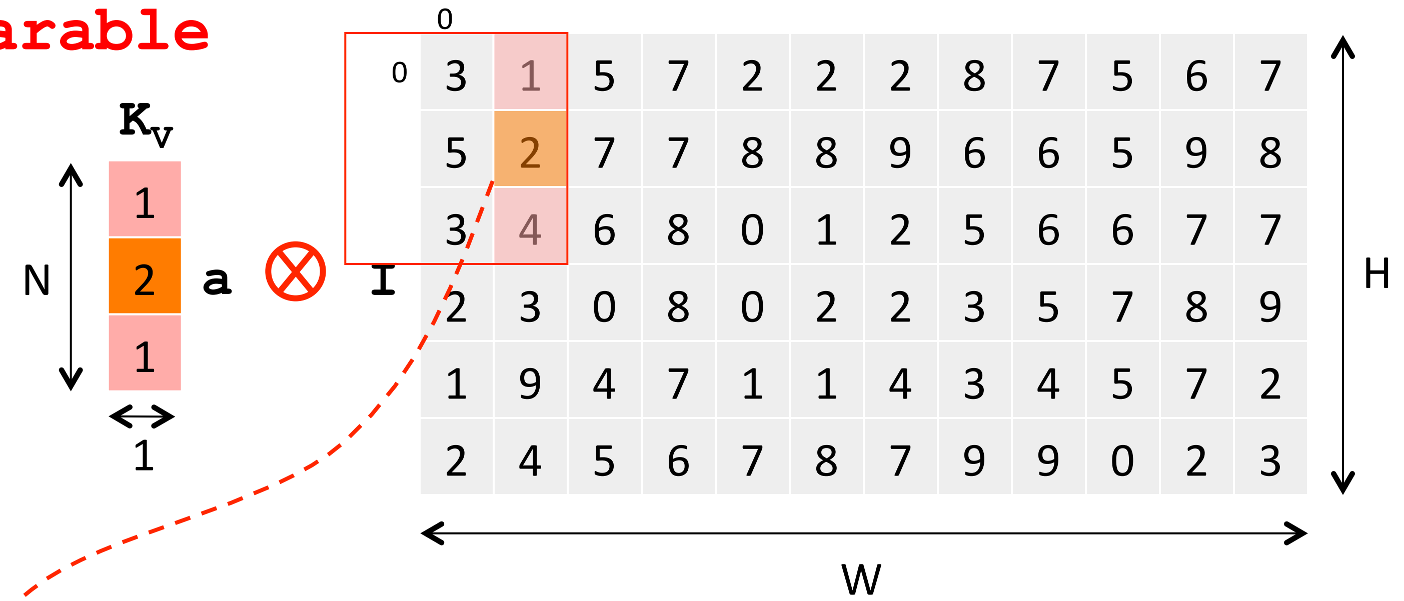
$$\mathbf{I}' = \mathbf{b} \otimes \mathbf{I}_{\mathbf{TEMP}}$$



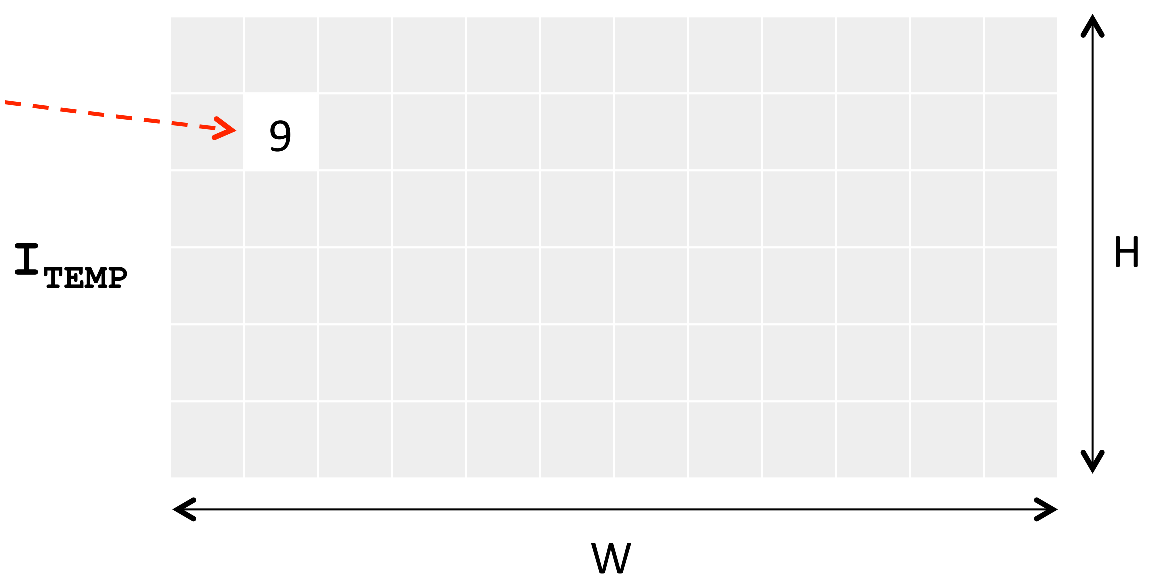
$$I'[1,2] = 1\cdot 1 + 0\cdot 5 - 1\cdot 7 + 2\cdot 2 + 0\cdot 7 - 2\cdot 7 + 1\cdot 4 + 0\cdot 6 - 1\cdot 8 = -20$$

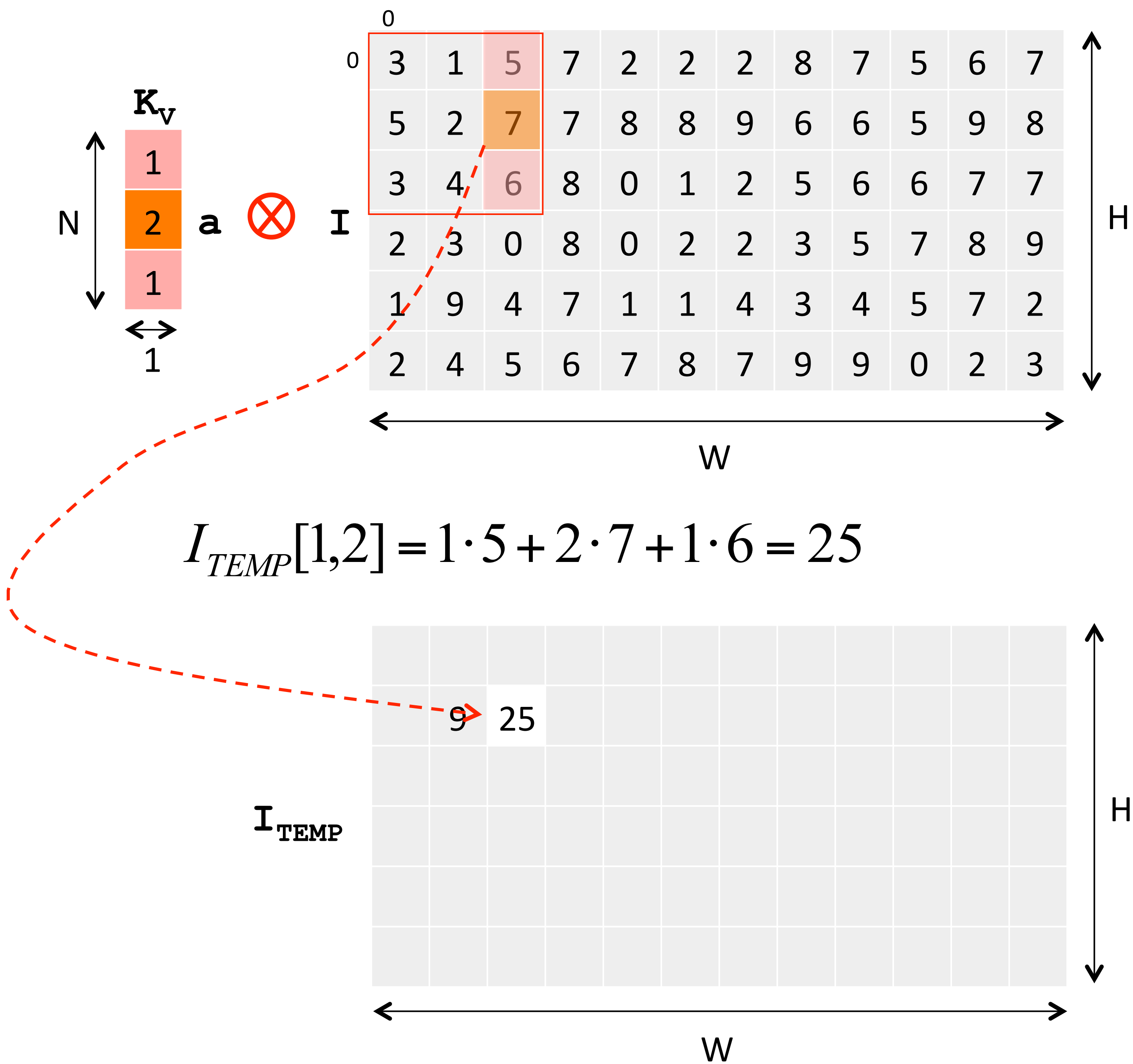


Separable

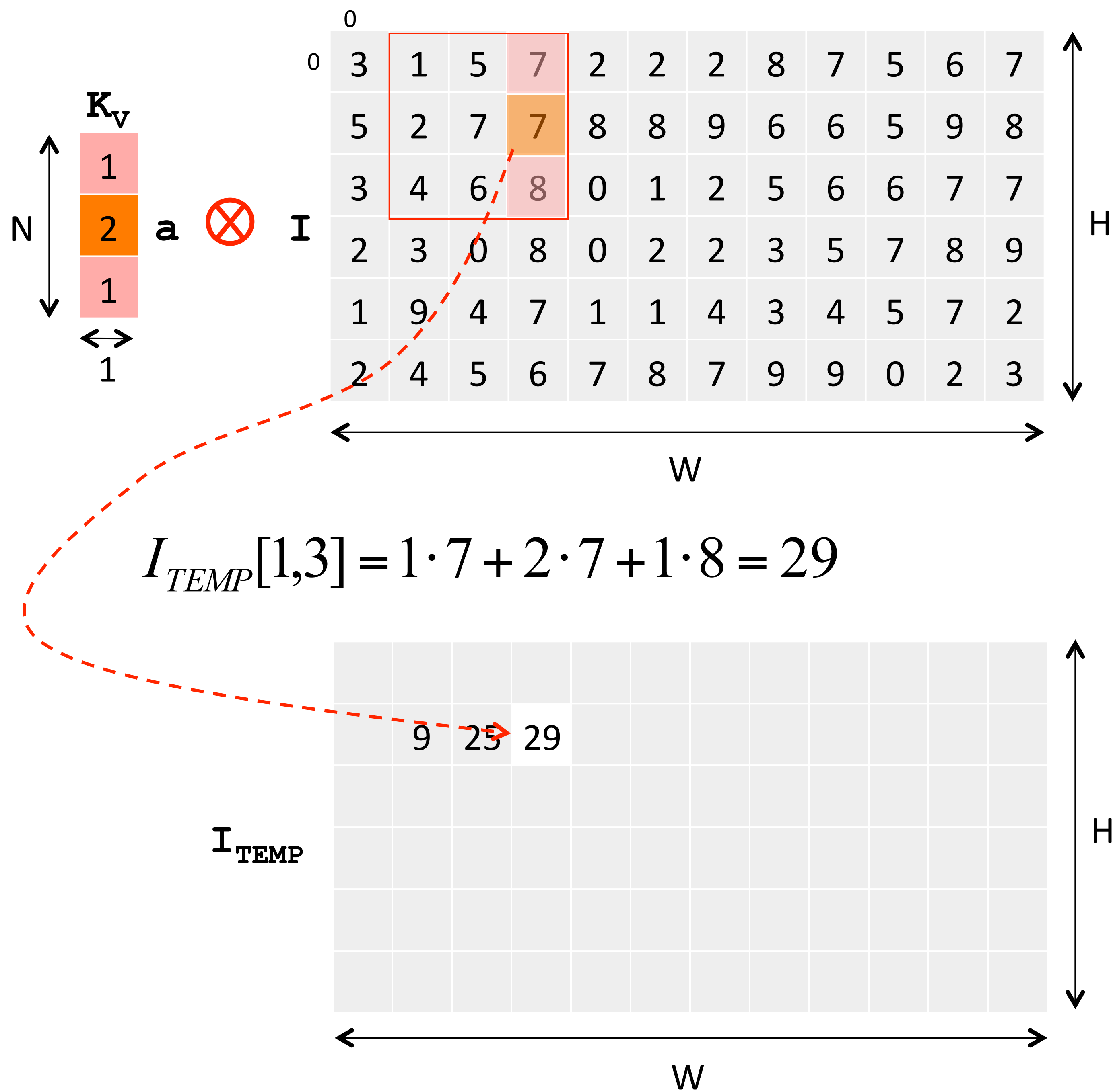


$$I_{TEMP}[1,1] = 1 \cdot 1 + 2 \cdot 2 + 1 \cdot 4 = 9$$

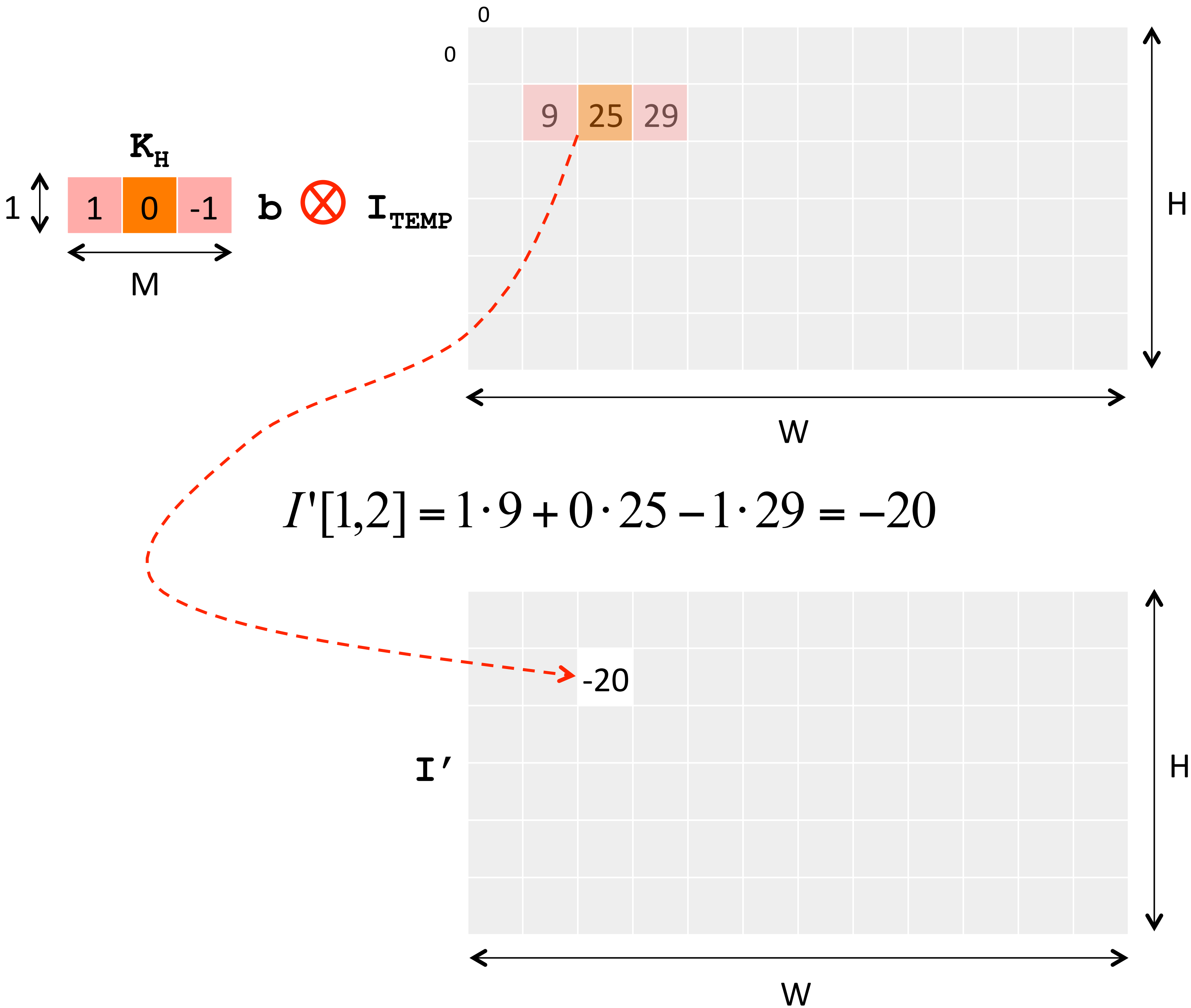




$$I_{TEMP}[1,2] = 1\cdot 5 + 2\cdot 7 + 1\cdot 6 = 25$$



K_V
N
1
2
1
1
a
I
0
0
3 1 5 7 2 2 2 8 7 5 6 7
5 2 7 7 8 8 9 6 6 5 9 8
3 4 6 8 0 1 2 5 6 6 7 7
2 3 0 8 0 2 2 3 5 7 8 9
1 9 4 7 1 1 4 3 4 5 7 2
2 4 5 6 7 8 7 9 9 0 2 3
H
W
$I_{TEMP}[1,3] = 1 \cdot 7 + 2 \cdot 7 + 1 \cdot 8 = 29$
9 25 29
I_TEMP
H
W



$$I'[1,2] = 1 \cdot 9 + 0 \cdot 25 - 1 \cdot 29 = -20$$

Separable vs 2D: complexity and buffering

- Same results, less computations:

  Conventional ≈ WxHx**MxN**

  Separable ≈ WxHx(**M+N**)

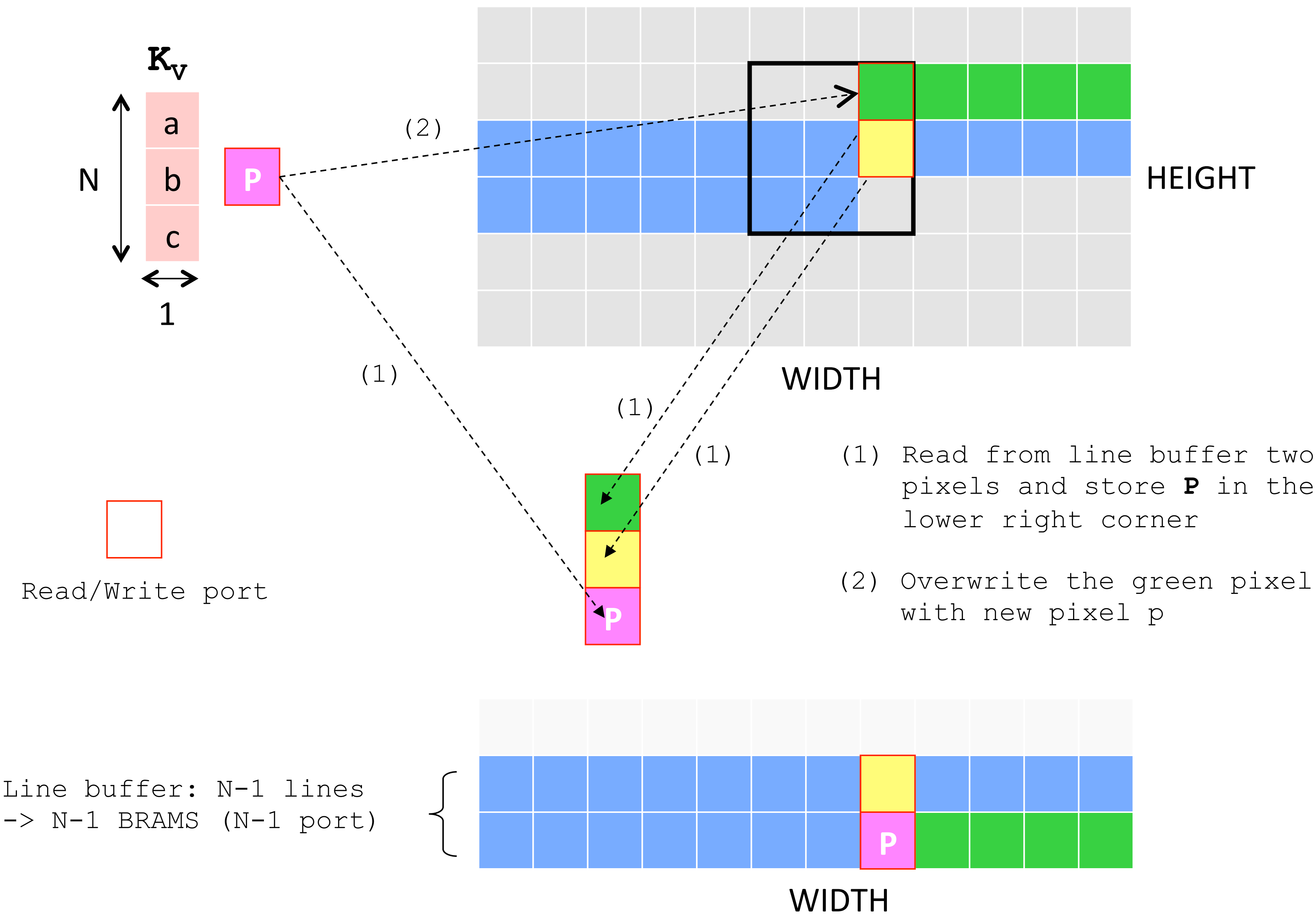
Complexity grows according to M + N

- Same buffering (i.e., N-1 image lines)

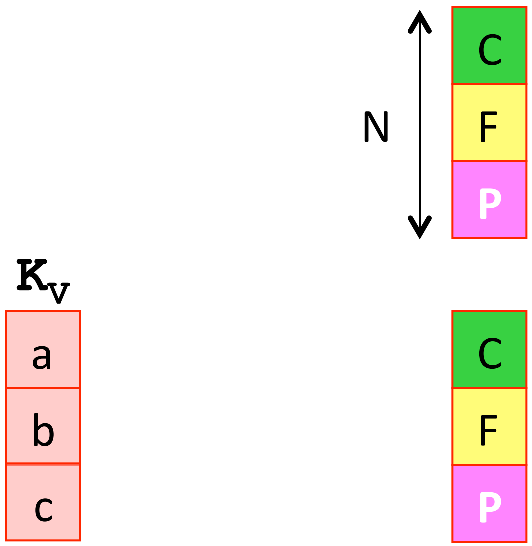
↓

Separable filters are particularly suited for hardware implementation (with negligible modifications wrt 2D)

Convolution and data structures: line buffer

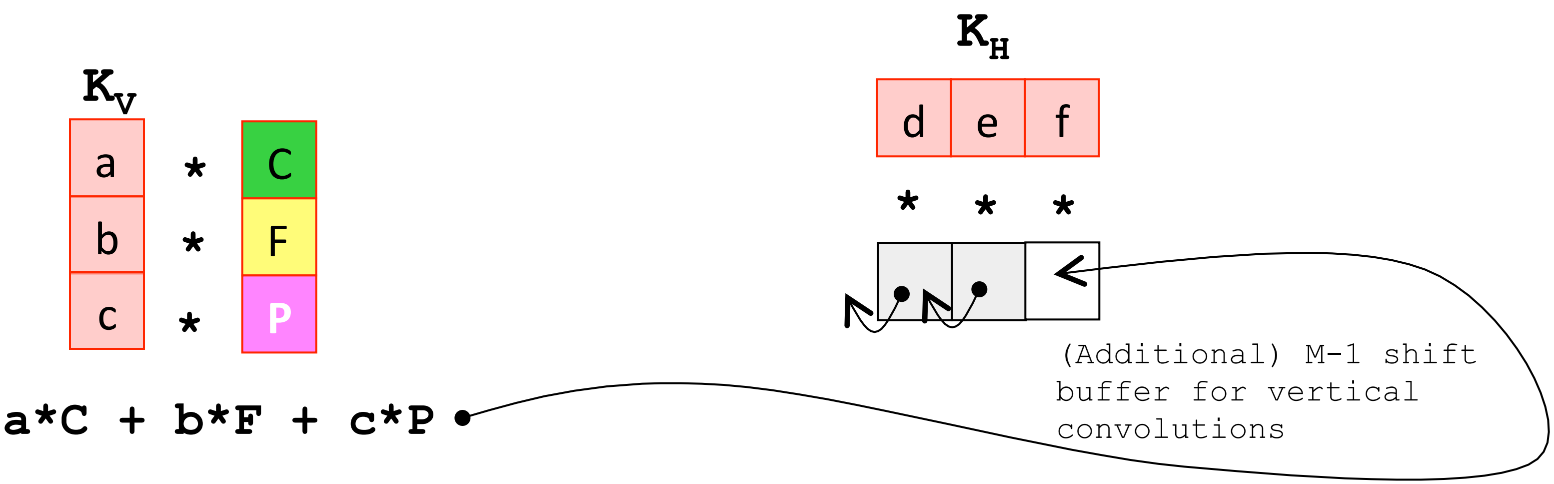


Convolution and data structures: image patch

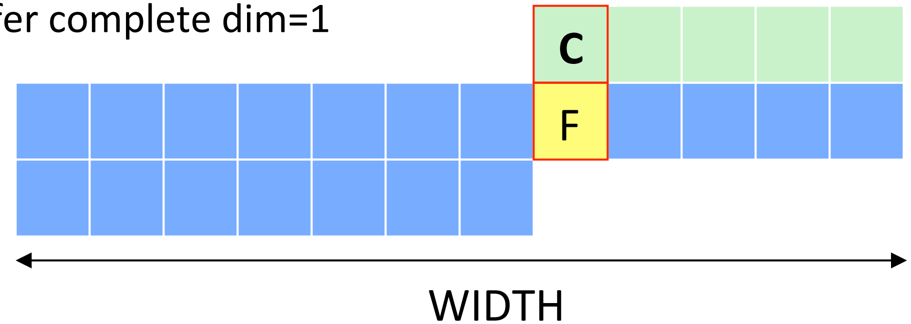


(3) The image patch is a colum vector

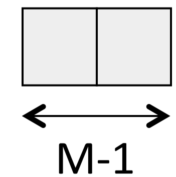
(4) Dot product: N parallel read (K and patch). For both data structures N read ports



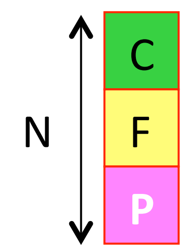
```
// line buffer
static pixel line_buffer[N - 1][IMAGE_WIDTH];
#pragma HLS ARRAY_PARTITION variable=line_buffer complete dim=1
```



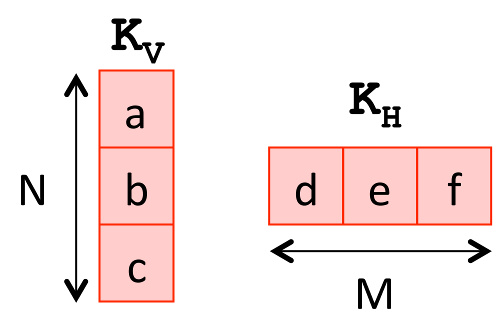

```
// shift register (vertical convolutions)
static accumulated_pixel convolution_buffer[M-1];
#pragma HLS ARRAY_PARTITION variable=convolution_buffer complete dim=0
```



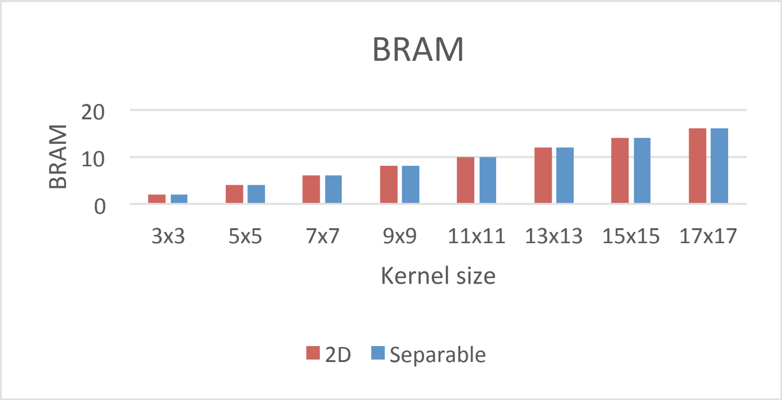

```
// processing window
static pixel window[N];
#pragma HLS ARRAY_PARTITION variable=window complete dim=0
```

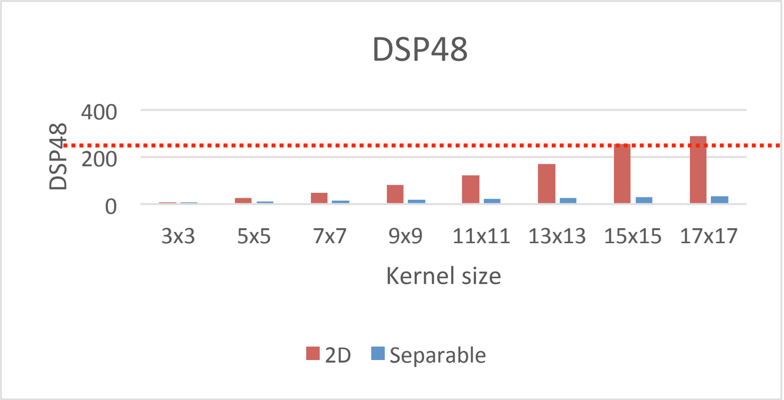


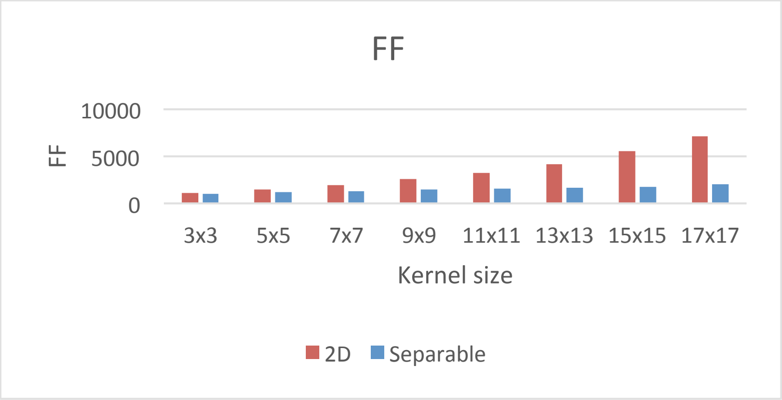

```
// vertical and horizontal kernels
static s_int kernel_v[N];
static s_int kernel_h[N];
#pragma HLS ARRAY_PARTITION variable=kernel_v complete dim=0
#pragma HLS ARRAY_PARTITION variable=kernel_h complete dim=0
```

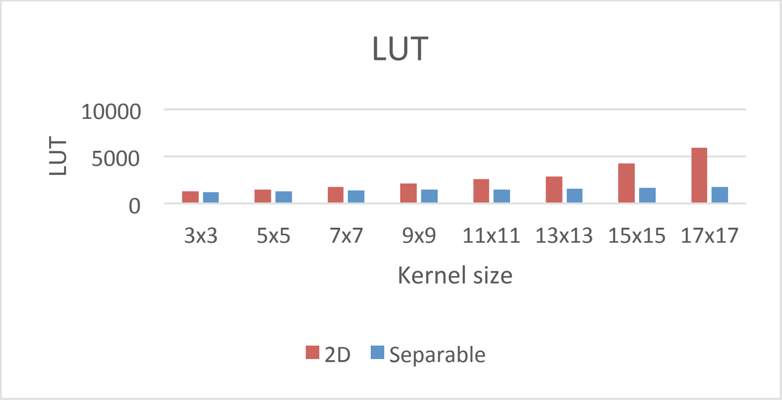


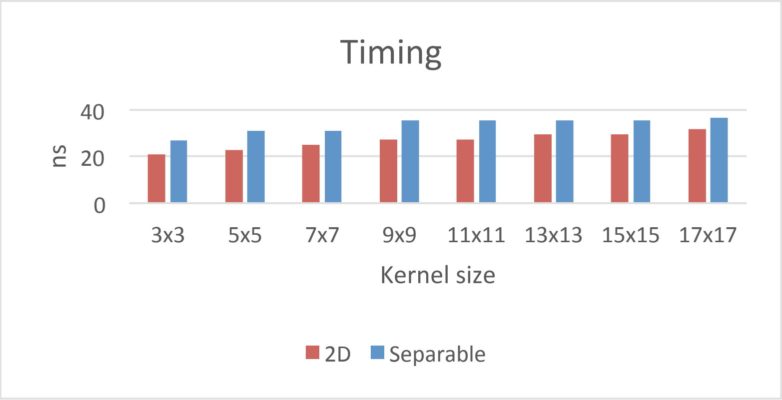
Implementation report: 2D vs separable











How to obtain the coefficients?

- If rank(K) is 1 the filter is (fully) separable:
  - Factorize K as USV' with SVD decomposition
  - Extract the first columns of U and V
  - Scale these two vectors according to the unique non singular value σ (upper left element of S)

Matlab code:

```
[U,S,V] = svd(K)
 a = sqrt(S(1,1))*U(:,1)
 b = sqrt(S(1,1))*V(:,1)'
```

- A *low-rank filter* can be obtained by combining two or more fully separable filters (again, using SVD)

$$\mathbf{K_{SH}} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \xrightarrow{\text{svd}} \mathbf{K_{SH}} = \mathbf{USV'}$$

$$\mathbf{U} = \begin{bmatrix} -0.4082 & 0.9129 & 0 \\ -0.8165 & -0.3651 & -0.4472 \\ -0.4082 & -0.1826 & 0.8944 \end{bmatrix} \rightarrow \mathbf{a} = \begin{bmatrix} -0.4082 \\ -0.8165 \\ -0.4082 \end{bmatrix}$$

$$\mathbf{S} = \begin{bmatrix} 3.4641 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \rightarrow \sigma = 3.4641$$

$$\mathbf{V} = \begin{bmatrix} 0.7071 & 0.7071 & 0 \\ 0 & 0 & 1 \\ -0.7071 & 0.7071 & 0 \end{bmatrix} \rightarrow \mathbf{b} = \begin{bmatrix} -0.7071 & 0 & -0.7071 \end{bmatrix}$$

$$\mathbf{K_{SH}} = \sigma\,\mathbf{ab}$$

$$K_G = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \xrightarrow{\text{svd}} K_G = USV'$$

$$U = \begin{bmatrix} -0.4082 & 0.9129 & 0 \\ -0.8165 & -0.3651 & -0.4472 \\ -0.4082 & -0.1826 & 0.8944 \end{bmatrix} \rightarrow a = \begin{bmatrix} -1 \\ -2 \\ -1 \end{bmatrix}$$

$$S = \begin{bmatrix} 6 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \rightarrow \sigma = 6$$

$$V = \begin{bmatrix} -0.4082 & -0.9129 & 0 \\ -0.8165 & 0.3651 & -0.4472 \\ -0.4082 & 0.1826 & 0.8994 \end{bmatrix} \rightarrow b = \begin{bmatrix} -1 & -2 & -0 \end{bmatrix}$$

$$K_G = \sigma\, ab$$

Can we replace 2D convs. with separable ones?

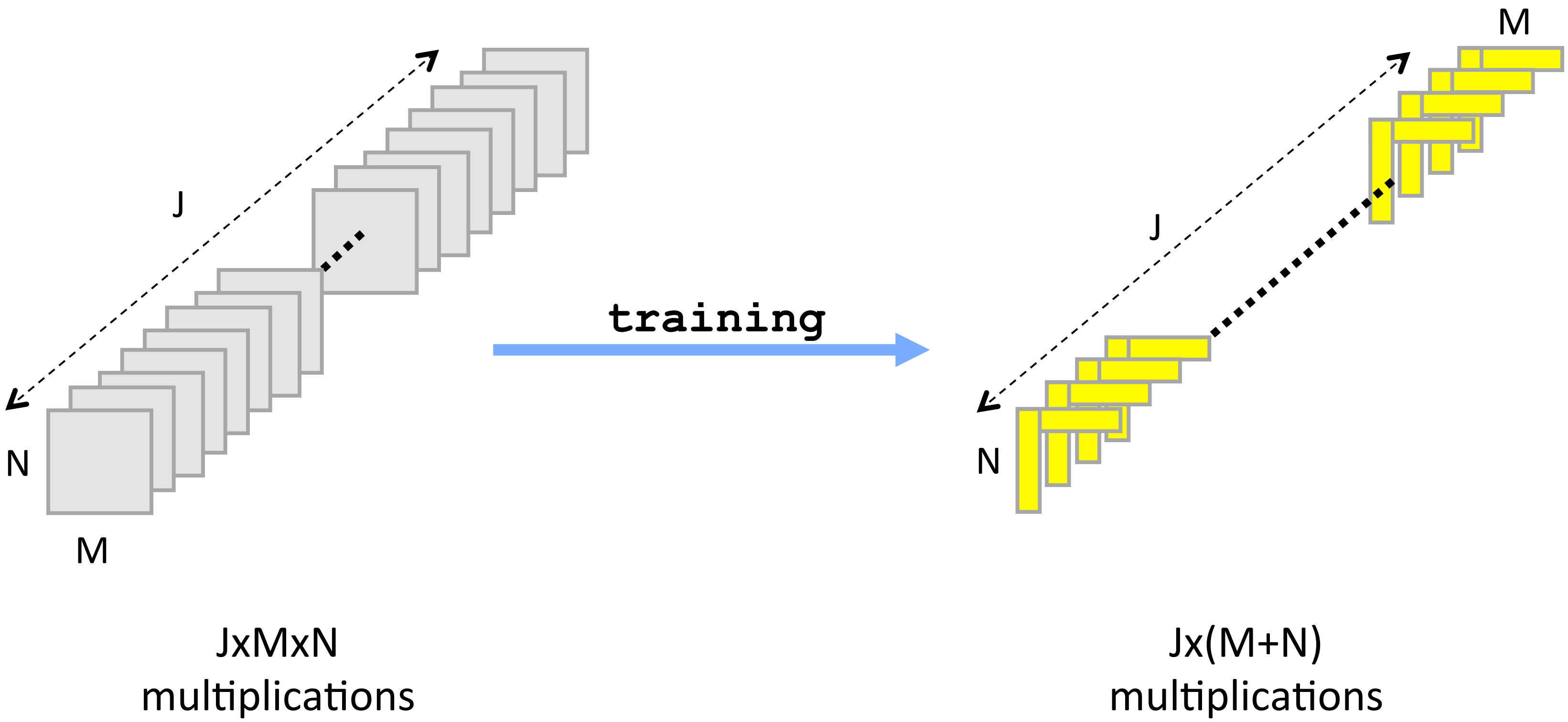
- Unfortunately not all filters are separable
- Convolution filters in CNNs are not separable
- How to take advantage of separability with deep networks without degrading performance?

[**SEP] proposed two techniques:**

1. **Enforcing separability in the training phase**
2. **Approximating (after training) 2D filters with a linear combination of fewer separable filters**

[SEP] Sironi, Tekin, Rigamonti, Lepetit, Fua, "Learning Separable Filters", PAMI 2015

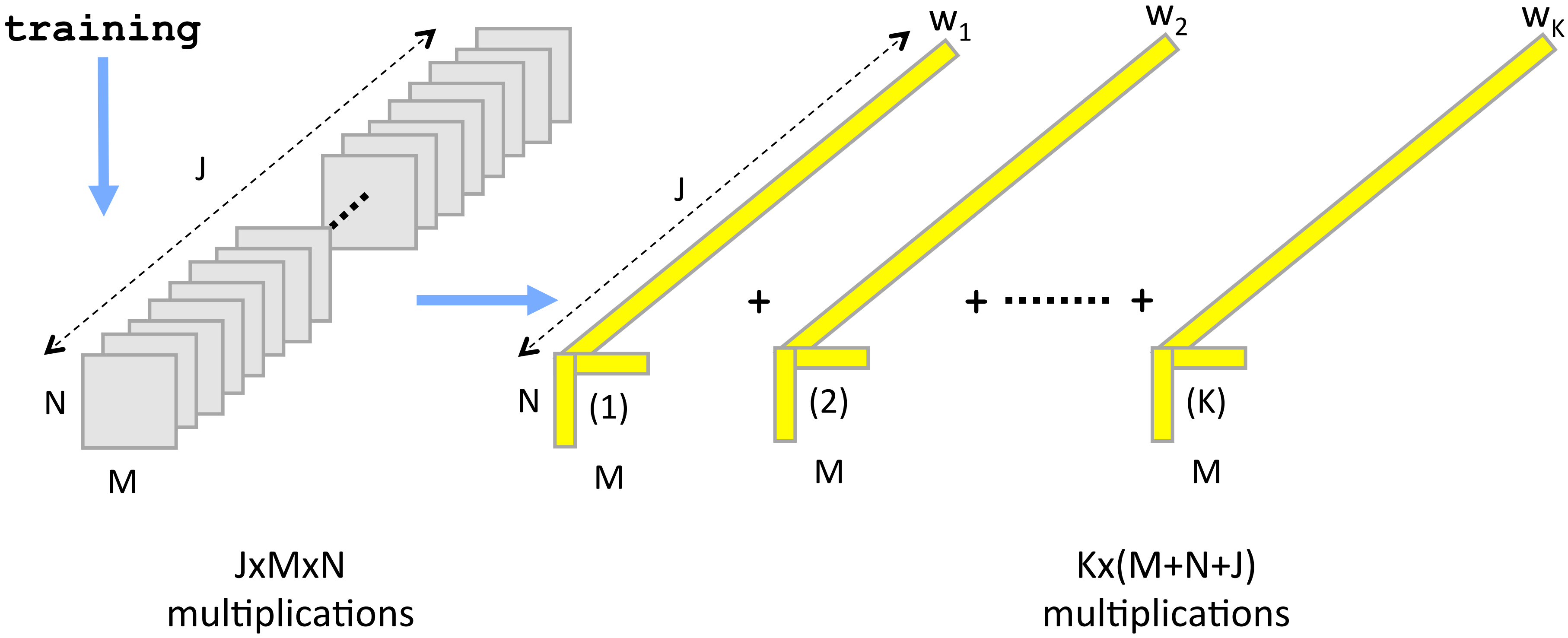
Enforcing separability in the training phase



Enforcing a soft-constraint during the training phase enables to replace each filter of the bank with its separable version of rank 1:

- **efficient for inference: Jx(MxN) vs Jx(M+N)**
- **worse results wrt original network**

Linear (weighted)combination of separable filters



Extending the previous method to the whole bank enables to replace it with a 3D separable version:

- **less efficient than previous technique**
- **better results**
- **standard training**

Example in the 2D domain

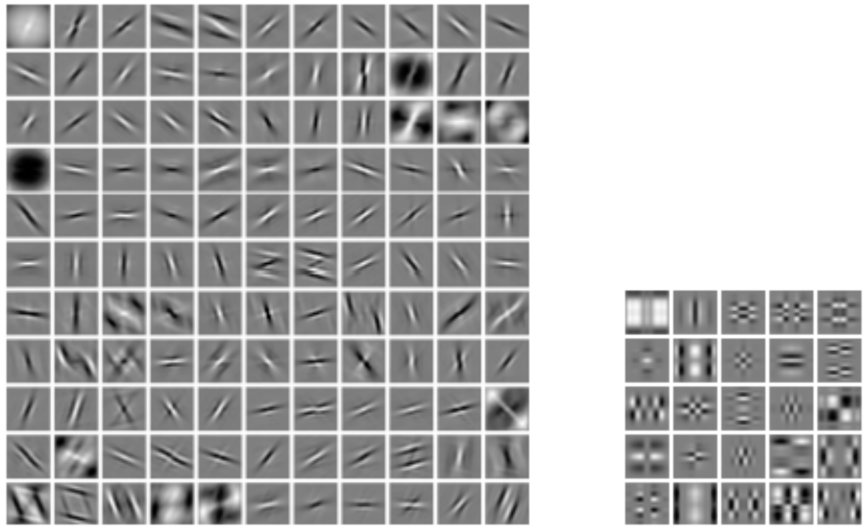


Image from [SEP]

Each of the 121 2D filters on the left can be approximated by a linear combination of the 25 separable filters on the right (see [SEP] for details)

Conclusions

- **Implementations of image filters with HLS tools is almost equivalent to a conventions software design flow**

- **Separable filters significantly reduces the amount of hardware resources (in particular DSPs)**

- **Demanding convolutional layers of CNNs can be approximated with separable filters**

- **Other issues concerned with CNNs and FPGA not discussed:**

  - **floating-point vs fixed-point computation**

  - **data transfer (FPGA <-> DDR)**

Acknowledgements*

Federico Bertoli

Luca Bonfigioli

Paolo Di Febbo

Alessandro Maragno

Alessio Mingozzi

Matteo Poggi

Marco Rossini

Nicola Severini

Fabio Tosi

** alphabetical order*