

CS 6023 - GPU Programming

Convolution Operation

14/09/2018

Setting and Agenda

- So far we have looked at vector addition, matrix multiplication, histogram computation
- We will look at another popular operation: Convolution

Acknowledgement: Nvidia teaching kit

Standard continuous time convolution

- Convolution system with input

$$x \quad (x(t) = 0, \quad t < 0)$$

transfer function $h(t)$

and output y is given as

$$y(t) = \int_0^t h(\tau)x(t - \tau)d\tau = \int_0^t h(t - \tau)x(\tau)d\tau$$

- For discrete time convolution

$$y[n] = \sum_{m=0}^n h[m]x[n - m] = \sum_{m=0}^n h[n - m]x[m]$$

Applications of convolution

- EE/ME: Modelling linear time invariant (LTI) systems
- Probability: computing the probability distributions of sum of two independent random variables
- Acoustics/Signal processing: filters on input signals
- Image processing: Edge detection, blurring, sharpening, etc.
 - => Deep learning

Small modifications

- We had

$$y[n] = \sum_{m=0}^n h[m]x[n-m] = \sum_{m=0}^n h[n-m]x[m]$$

- Changes:
 - Assume finite support of kernel function / transfer function, typically odd number so say $(2k + 1)$
 - Center kernel around the point at which we are computing output
 - Flip kernel function

$$y[n] = \sum_{m=n-k}^{n+k} h'[m]x[m]$$

Example



- Kernel size is 5, input size is 6
- Calculate $Y[2]$
- $Y[2] = X[0] H[0] + X[1] H[1] + X[2] H[2] + X[3] H[3] + X[4] H[4]$

Example

X	X[0]	X[1]	X[2]	X[3]	X[4]	X[5]	X[6]
	1	2	3	4	5	6	7

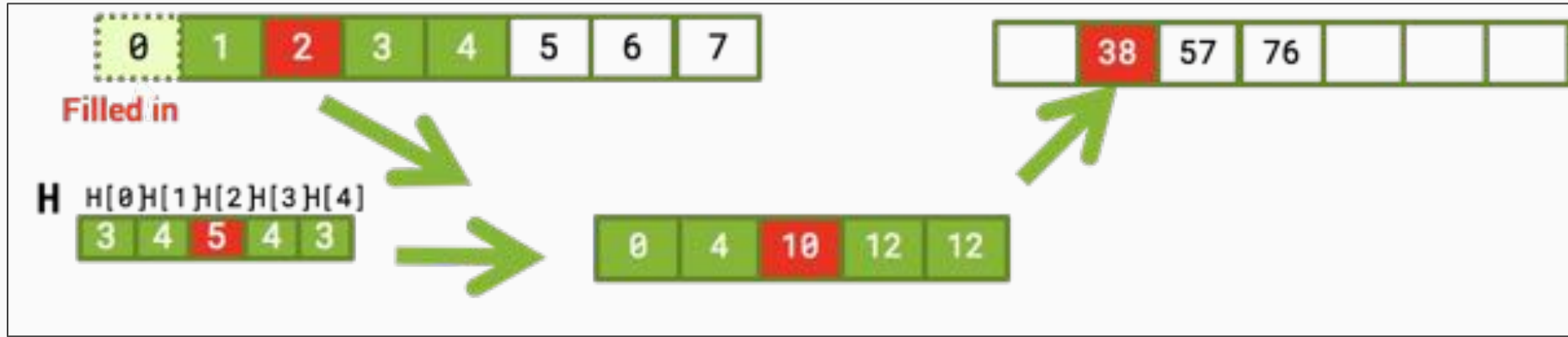
Y	Y[0]	Y[1]	Y[2]	Y[3]	Y[4]	Y[5]	Y[6]
			57	76			

H	H[0]	H[1]	H[2]	H[3]	H[4]
	3	4	5	4	3

6	12	20	20	18
---	----	----	----	----

- Kernel size is 5, input size is 6
- Calculate $Y[3]$
- $Y[3] = X[1] H[0] + X[2] H[1] + X[3] H[2] + X[4] H[3] + X[5] H[4]$

Example - what happens in the boundary



- Kernel size is 5, input size is 6
- Calculate $Y[1]$
- $Y[1] = 0 \cdot H[0] + X[0] \cdot H[1] + X[1] \cdot H[2] + X[2] \cdot H[3] + X[3] \cdot H[4]$

How can we parallelize this?

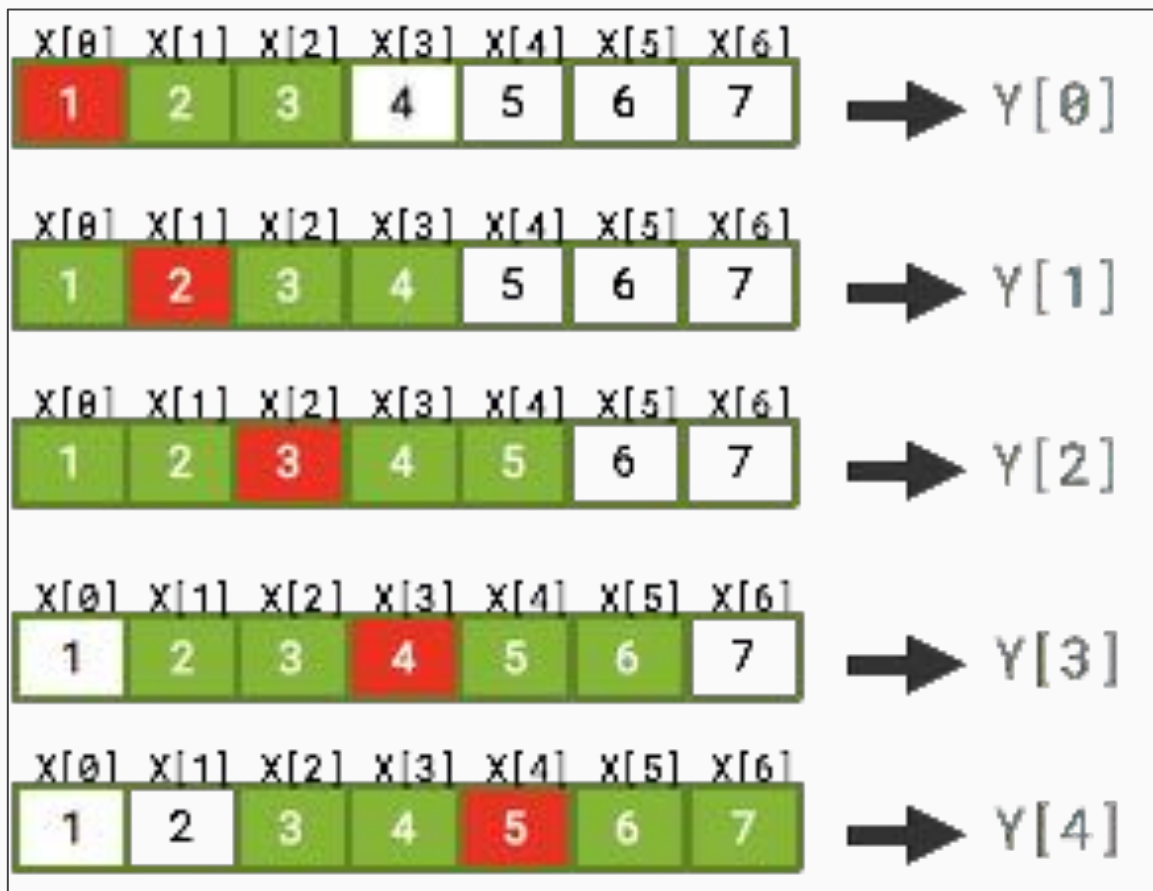
Basic version - Each thread computes one output index

```
__global__ void convolution_1D_basic_kernel(float *X, float *H,  
      float *Y, int Kernel_Width, int Width)  
{  
    int i = blockIdx.x*blockDim.x + threadIdx.x;  
    float Yvalue = 0;  
    int X_start_point = i - (Kernel_Width/2);  
  
    for (int j = 0; j < Kernel_Width; j++) {  
        if (X_start_point + j >= 0 && X_start_point + j < Width) {  
            Yvalue += X[X_start_point + j]*H[j];  
        }  
    }  
  
    Y[i] = Yvalue;  
}
```

Basic version - Each thread computes one output index

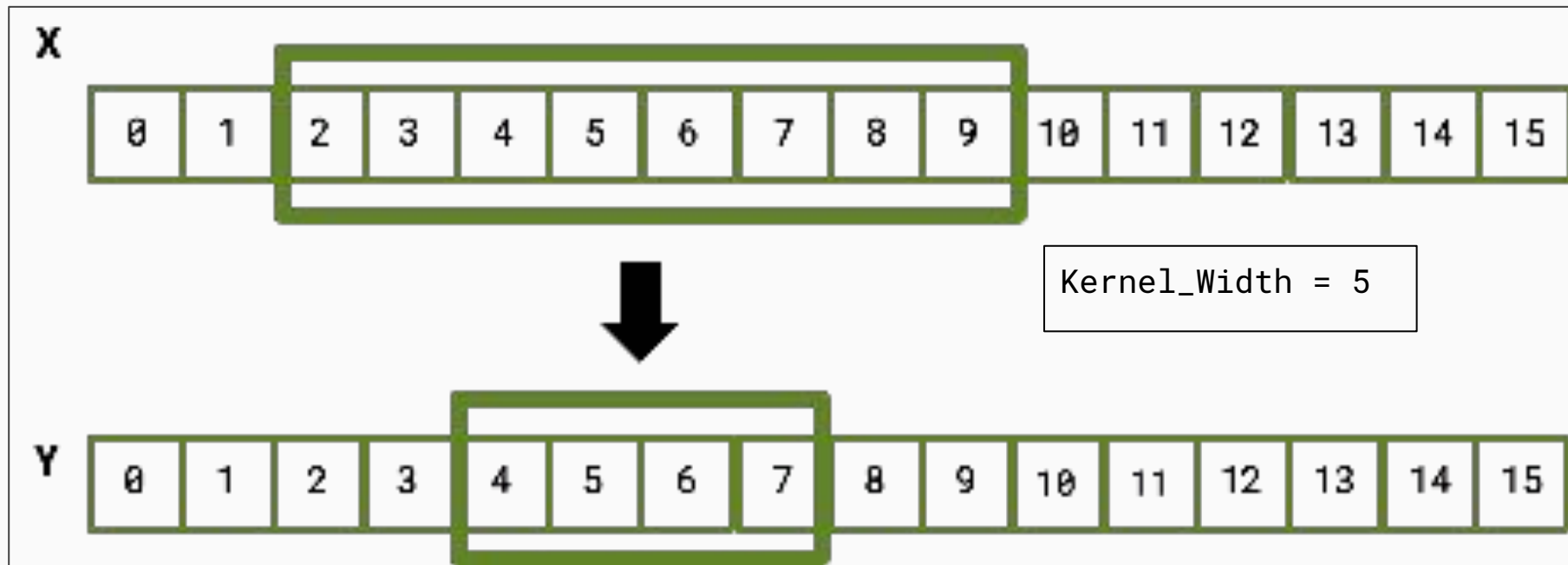
```
__global__ void convolution_1D_basic_kernel(float *X, float *H,  
    float *Y, int Kernel_Width, int Width)  
{  
    int i = blockIdx.x*blockDim.x + threadIdx.x;  
    float Yvalue = 0;  
    int X_start_point = i - (Kernel_Width/2);  
  
    for (int j = 0; j < Kernel_Width; j++) {  
        if (X_start_point + j >= 0 && X_start_point + j < Width) {  
            Yvalue += X[X_start_point + j]*H[j];  
        }  
    }  
  
    Y[i] = Yvalue;  
}
```

What can we do to improve performance further?

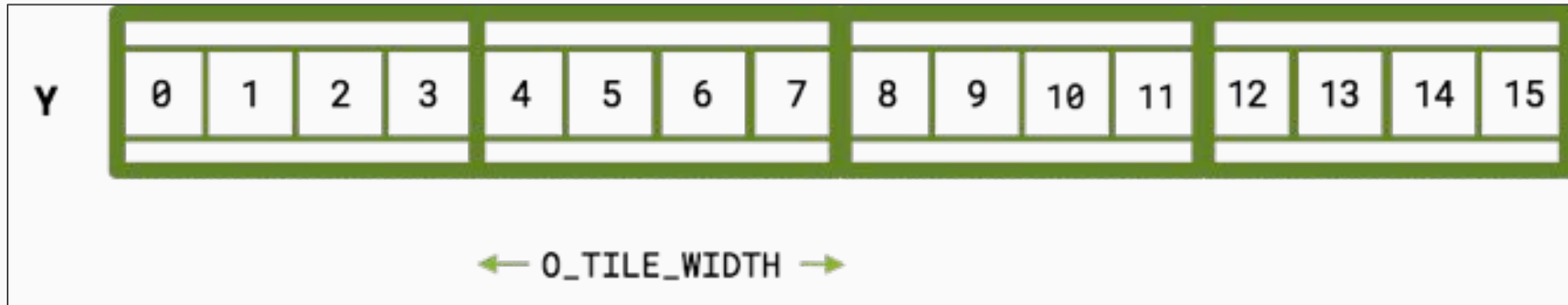


A block's access pattern

- Suppose each block to calculate T consecutive values of the output
- It needs to access $T + \text{Kernel_Width} - 1$ number of input elements, again we are assuming an odd sized `Kernel_Width`

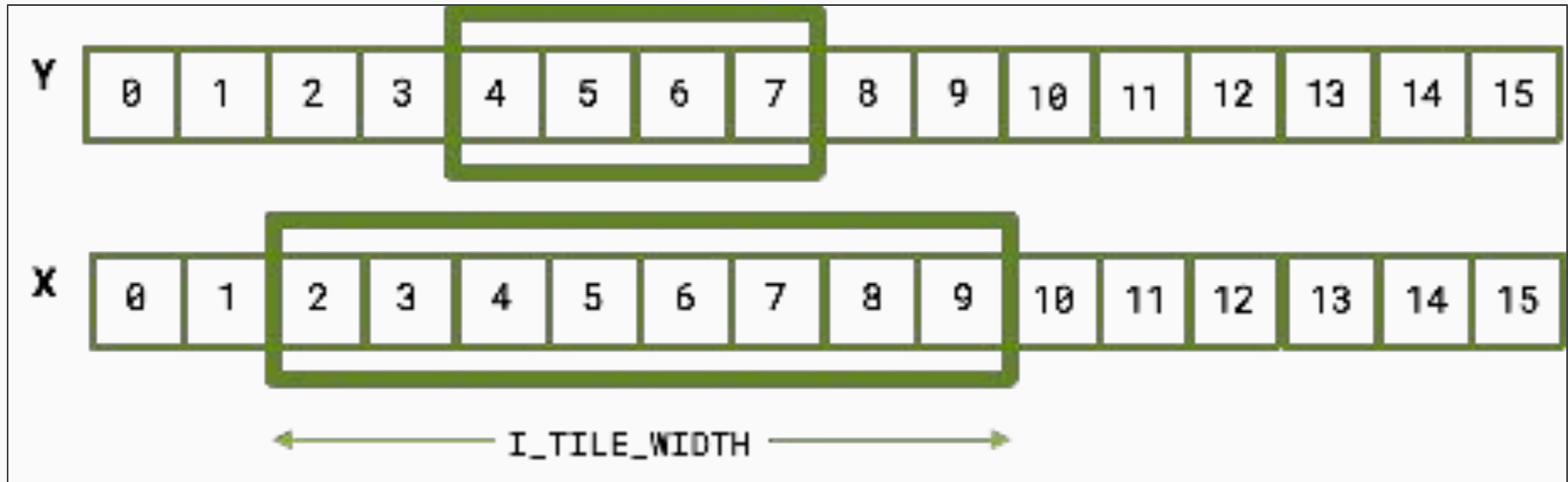


We are defining “Output” Tiles Too



- In this case `O_TILE_WIDTH = 4`
- Typically, `O_TILE_WIDTH` is significantly larger than `KERNEL_WIDTH`

Two kinds of tiles



- How do we size the blocks? What does each thread in a block do?

Two block design options

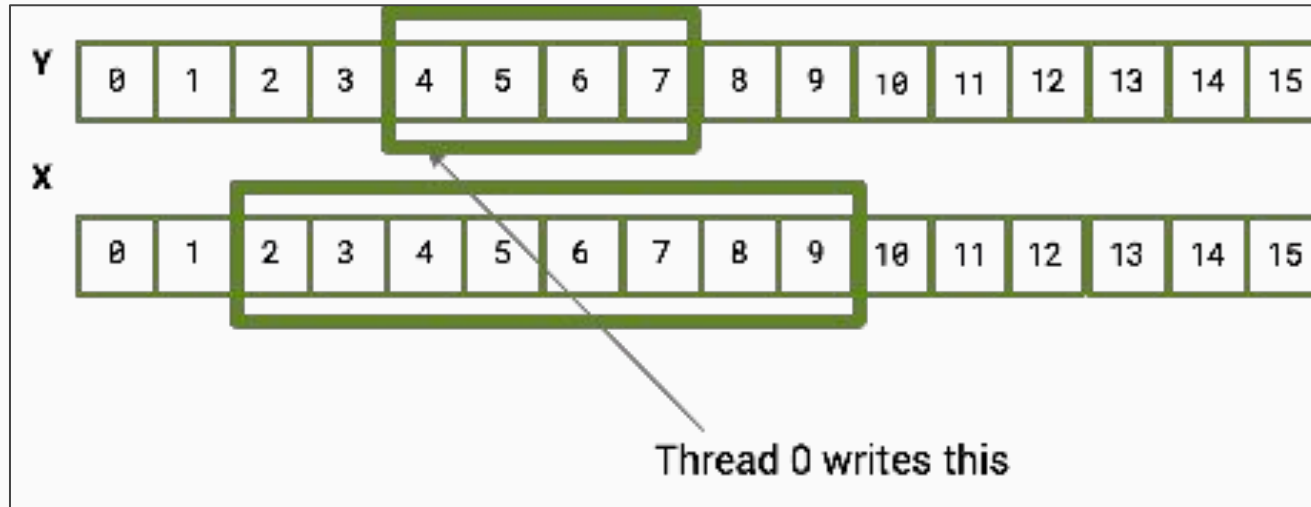
- Option 1: Threads per block = `O_TILE_SIZE`
- Option 2: Threads per block = `I_TILE_SIZE`

Two block design options

- Option 1: Threads per block = `O_TILE_SIZE`
 - Each thread can calculate Y value for one index
 - But how do we divide loading of X amongst threads
 - Some threads have to load more than 1 values
=> Control divergence
- Option 2: Threads per block = `I_TILE_SIZE`
 - Each thread can load X value for one index
 - But how do we divide computing different Y values amongst threads
 - Some threads have to compute more than 1 values
=> Control divergence

Two block design options

Option 1: Threads per block = 0_TILE_SIZE



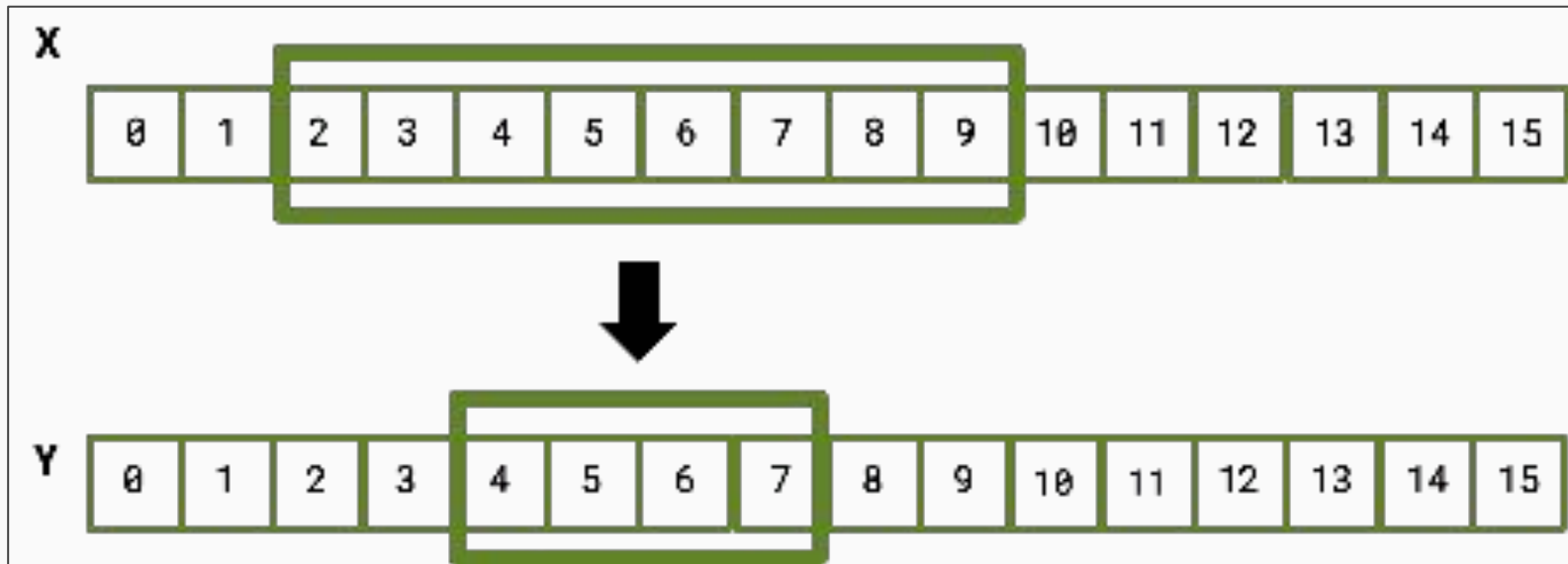
How will you distribute the loads?

Pseudocode

- Distribute loading of X amongst threads
- Load input tile into shared memory
- Synchronize
- Compute each output value
 - Do we use shared memory?
 - Do we use atomics?

Computational intensity with tiling

- Consider an output tile of size O_Tile_Width and kernel of size $Kernel_Width$
- What is the fraction of memory loads (only X) avoided by tiling?

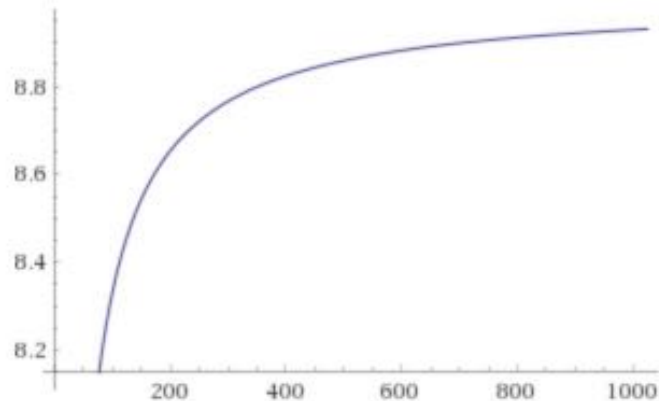
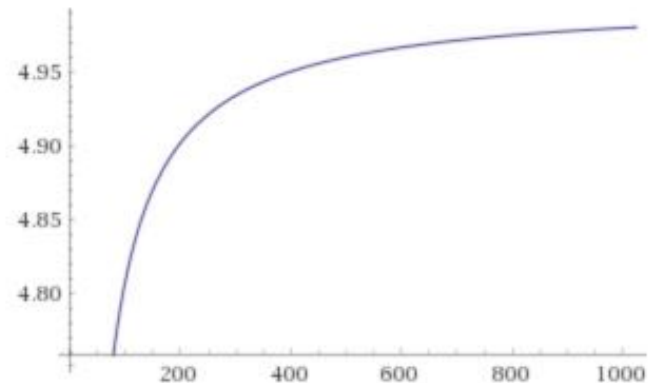


Computational intensity with tiling

- Consider an output tile of size O_Tile_Width and kernel of size $Kernel_Width$
- What is the fraction of memory loads avoided by tiling?
- Total number of loads are $O_Tile_Width + Kernel_Width - 1$
- Had we not used shared memory, each output element of Y would have required $Kernel_Width$ number of reads
- Bandwidth reduction =
$$\frac{O_Tile_Width * Kernel_Width}{O_Tile_Width + Kernel_Width - 1}$$

Bandwidth reduction

O_Tile_Width	16	32	64	128	256
Kernel_Width = 5	4.0	4.4	4.7	4.8	4.9
Kernel_Width = 9	6.0	7.2	8.0	8.5	8.7



Can we do anything else?

- Amongst all the inputs we have across algorithms, what is different about the kernel input for convolution?

Can we do anything else?

- Amongst all the inputs we have across algorithms, what is different about the kernel input for convolution?

It is read a very large number of times and never changed.

One option: read it to shared memory.

But ends up consuming shared memory capacity (duplicated per block)

Can we do anything else?

- CUDA devices provide constant memory which can be cached
 - Cached values are accessible by threads in all blocks
 - Moderately high memory bandwidth
- However, to qualify for caching, we need to provide compiler explicit qualifier of `const __restrict__`

```
__global__ void convolution_1D_basic_kernel(float *X,  
    const float __restrict__ *H,  
    float *Y, int Kernel_Width, int Width)
```

Restrict qualifier

- `restrict` qualifier is available in other languages (including C/C++)
- Required to avoid performance hit from **pointer aliasing**
- Without the `restrict` qualifier, the compiler may assume pointers could point to the same memory and thus reload values multiple times
- Example: In the code below, compiler may reload `c[i]` because it is not sure if `c` and `a` point to the same address

```
void example(float *a, float *b, float *c, int i) {  
    a[i] = a[i] + c[i];  
    b[i] = b[i] + c[i];  
}
```

2d convolutions

- Input X is two-dimensional
- Kernel H is two-dimensional
- Output Y is also thus two-dimensional
- Very common in image processing

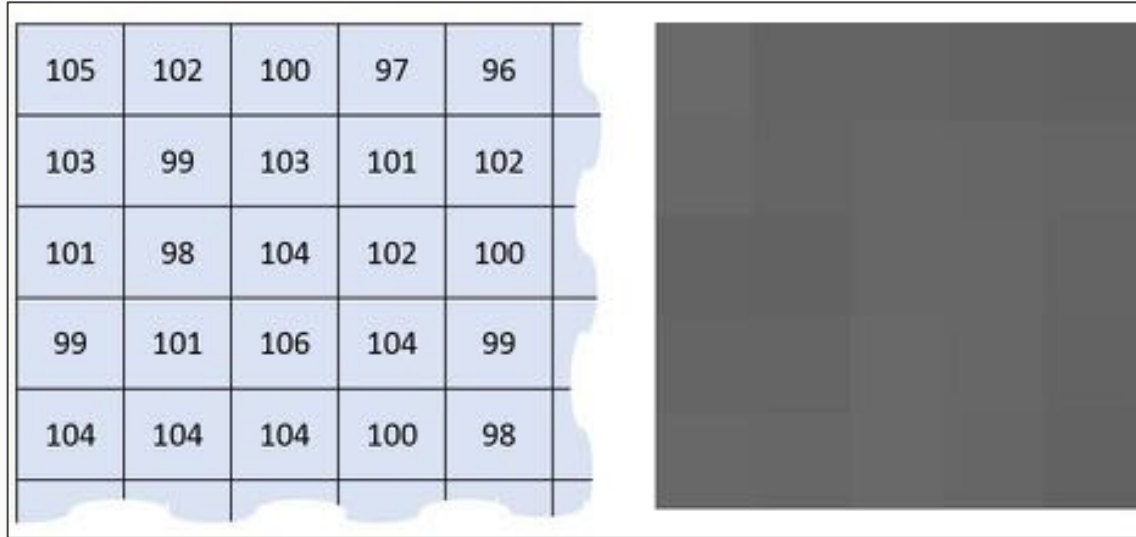
2d convolutions



Consider a 2d matrix
representing a black and
white image

Each point represents the
grayness of that point (0 -
255)

2d convolutions



Consider a 2d matrix
representing a black and
white image

Each point represents the
grayness of that point (0 -
255)

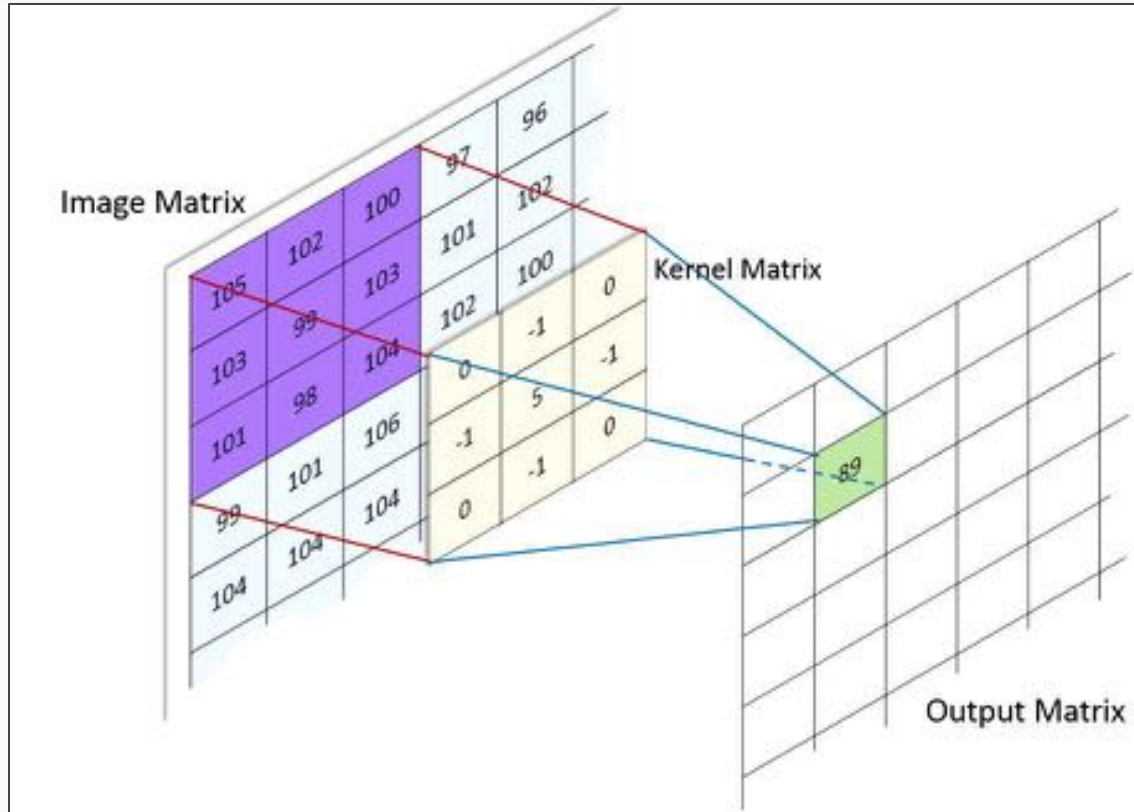
2d convolutions

$$\textit{Kernel} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

For 2d convolution, the convolutional kernel is 2d

This specific kernel implements sharpening

2d convolutions



At each point in the output matrix, kernel is centered and multiplied pointwise with the input matrix and added

2d convolutions

105	102	100	97	96
103	99	103	101	102
101	98	104	102	100
99	101	106	104	99
104	104	104	100	98

Image Matrix

0	-1	0
-1	5	-1
0	-1	0

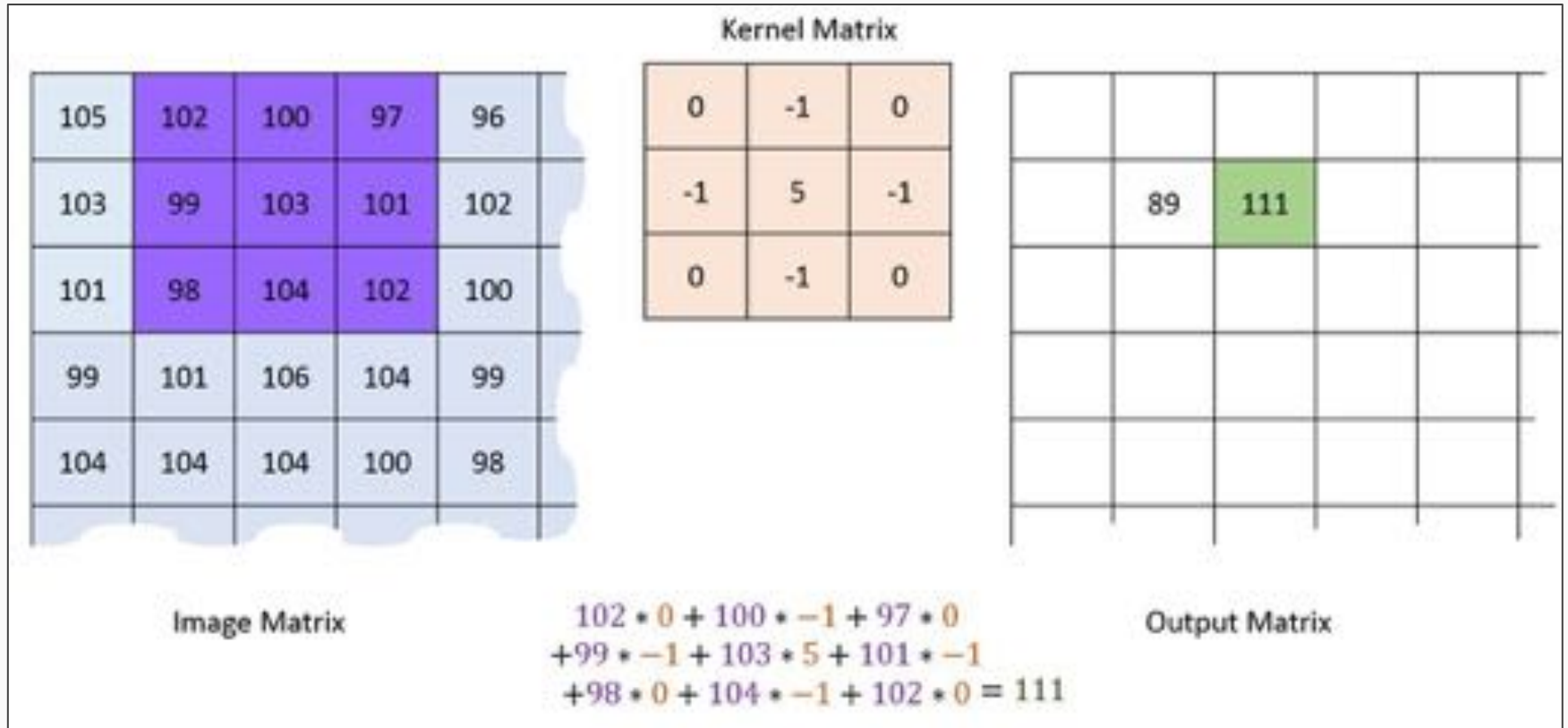
Kernel Matrix

	89			

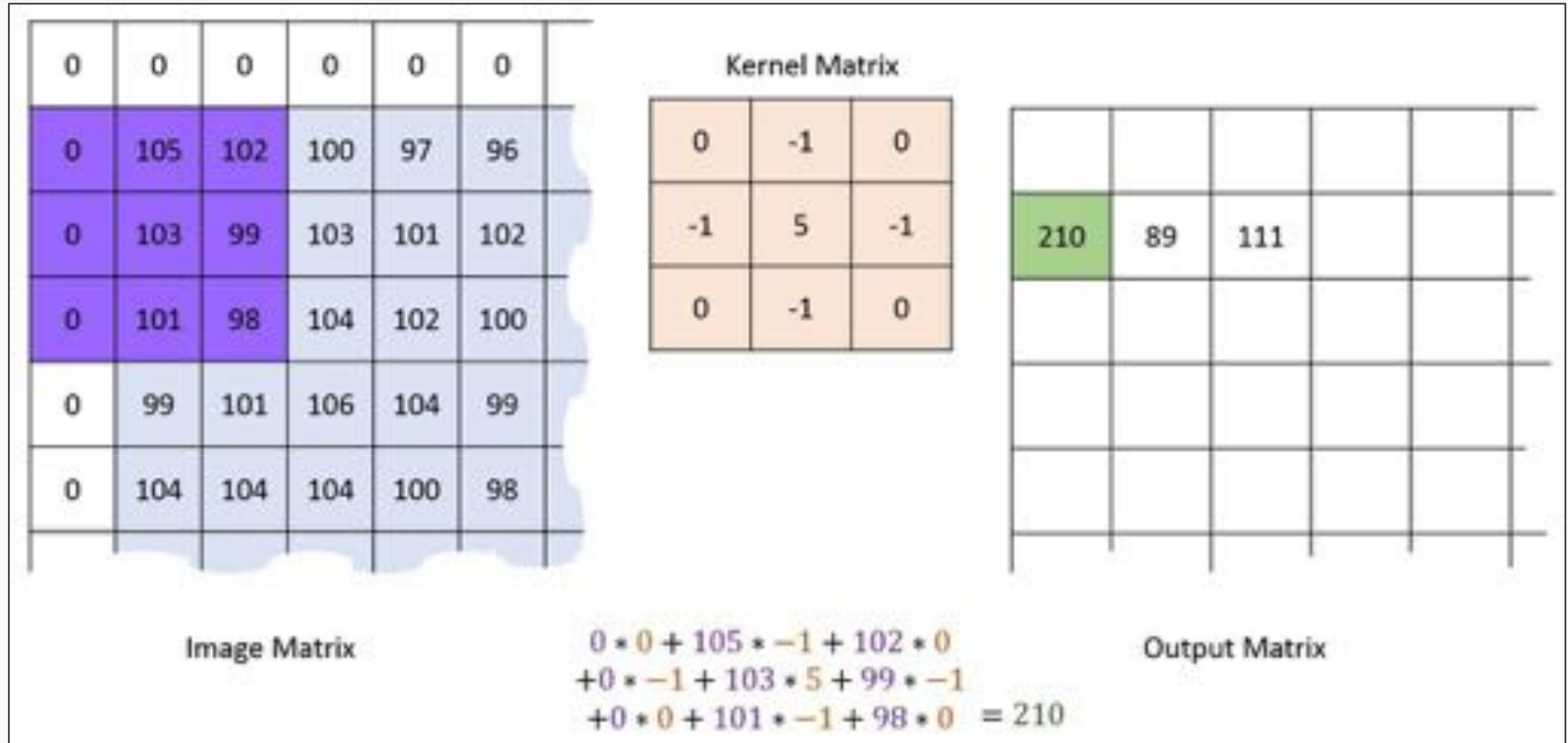
Output Matrix

$$\begin{aligned} &105 * 0 + 102 * -1 + 100 * 0 \\ &+ 103 * -1 + 99 * 5 + 103 * -1 \\ &+ 101 * 0 + 98 * -1 + 104 * 0 = 89 \end{aligned}$$

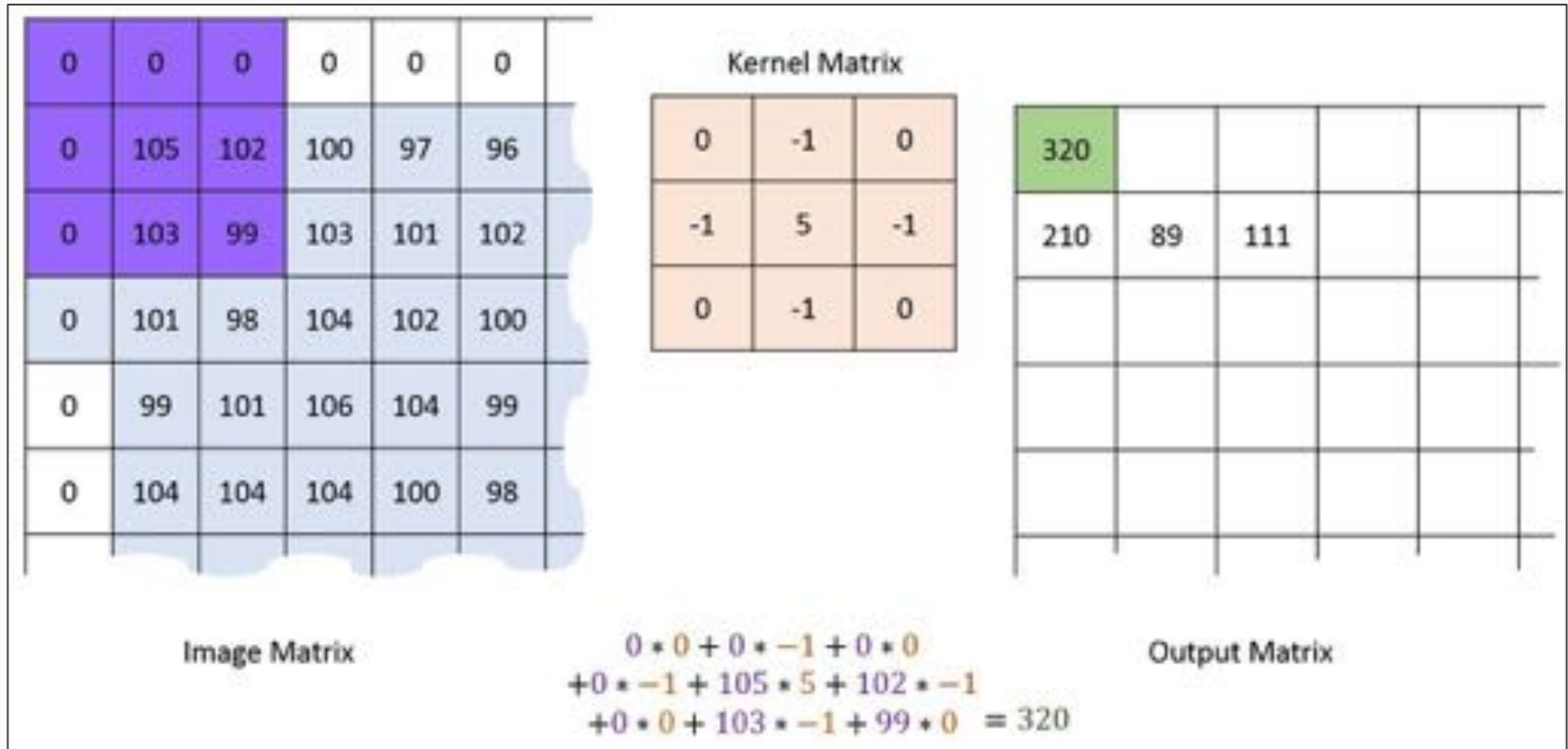
2d convolutions



2d convolutions



2d convolutions



2d convolutions - Result of sharpen kernel








Original



After convolution

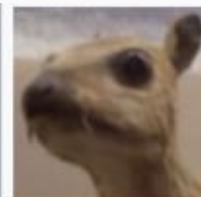
Other kernels - [https://en.wikipedia.org/wiki/Kernel_\(image_processing\)](https://en.wikipedia.org/wiki/Kernel_(image_processing))

Operation	Kernel	Image result
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	

Box blur

(normalized)

$$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$



Gaussian blur 3 x 3

(approximation)

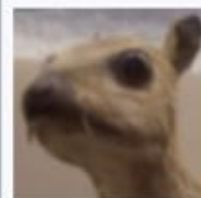
$$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$



Gaussian blur 5 x 5

(approximation)

$$\frac{1}{256} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$



Unsharp masking 5 x 5

Based on Gaussian blur
with amount as 1 and
threshold as 0
(with no image mask)

$$\frac{-1}{256} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & -476 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$



How to parallelize

- Like in 1d convolution, we should use shared memory
- What are the options for tile sizing
- Show that with tiling:
 - not only operational intensity increases
 - but access patterns become coalesced
- Assume output stationary with tile-size = `0_TILE_SIZE`

Memory access pattern of a thread

Kernel H. Dim = 3×3



Input X. Dim = 1000×1000



Input tile Dim = 16×16

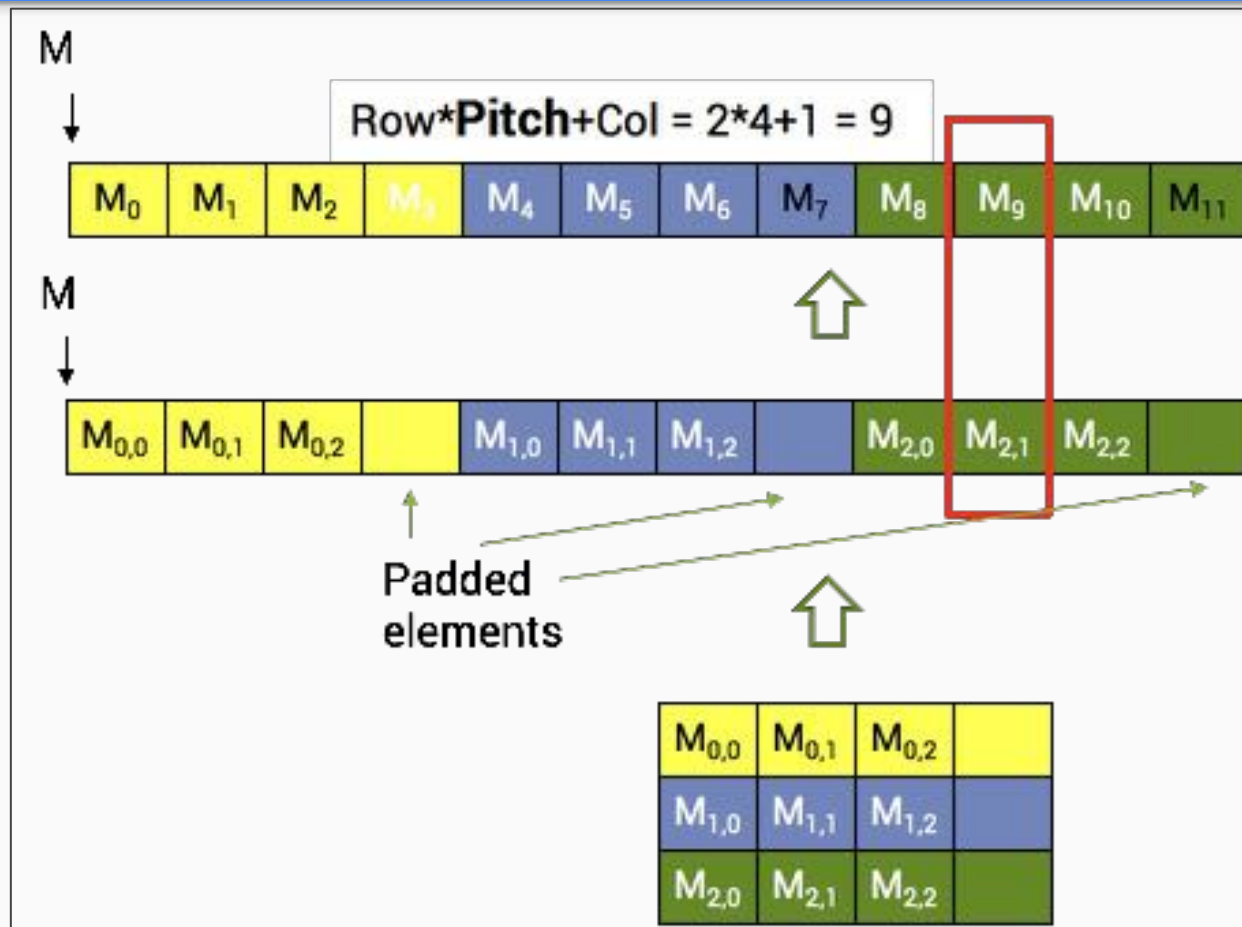
Input Y. Dim = 1000×1000



Output tile Dim = 15×15

One more trick with matrix loading

- Pad matrices by zeros to ensure their sizes are multiples of DRAM bursts
- Applies in other algos also
- Sometimes automatically done by compiler



Computational intensity with tiling for 2d convolution

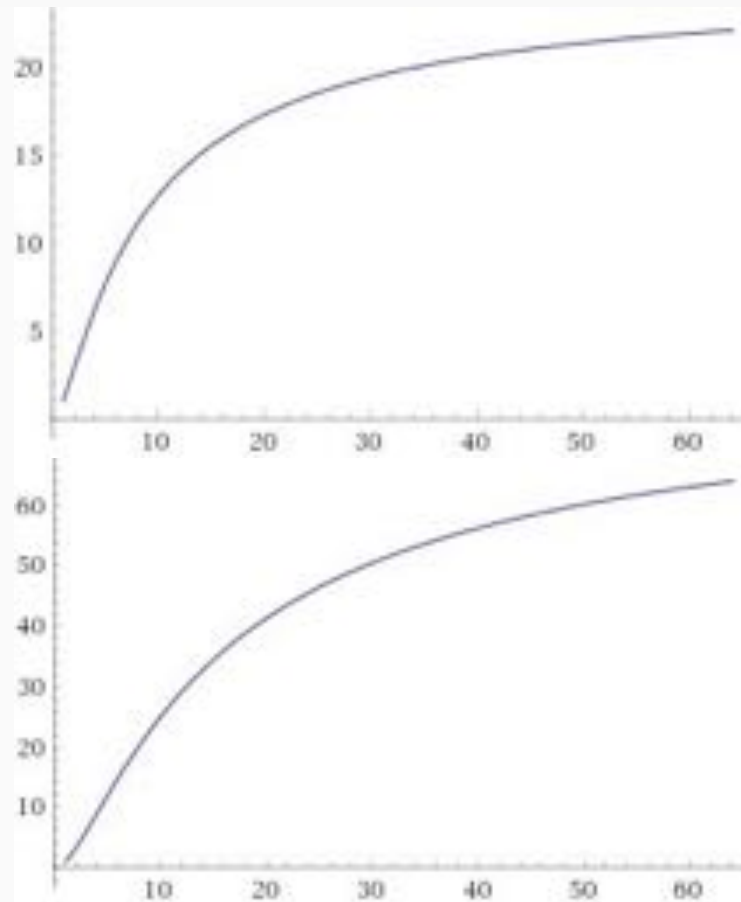
- Consider an output tile of size $O_Tile_Width \times O_Tile_Width$ and kernel of size $Kernel_Width \times Kernel_Width$
- What is the fraction of memory loads avoided by tiling?

Computational intensity with tiling for 2d convolution

- Consider an output tile of size $O_Tile_Width \times O_Tile_Width$ and kernel of size $Kernel_Width \times Kernel_Width$
- What is the fraction of memory loads avoided by tiling?
- Total number of loads are $(O_Tile_Width + Kernel_Width - 1)^2$
- Had we not used shared memory, each output element of Y would have required $(Kernel_Width)^2$ number of reads
- Bandwidth reduction =
$$\frac{(O_Tile_Width * Kernel_Width)^2}{(O_Tile_Width + Kernel_Width - 1)^2}$$

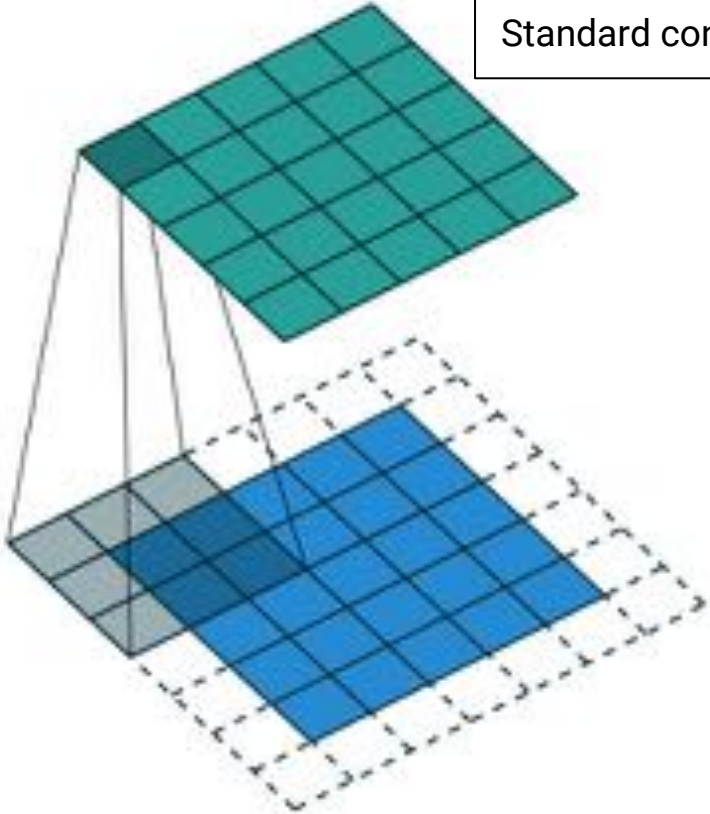
Bandwidth reduction

O_Tile_Width	16	32	64	128
Kernel_Width = 5	11.1	16	19.7	22.1
Kernel_Width = 9	20.3	36	51.8	64

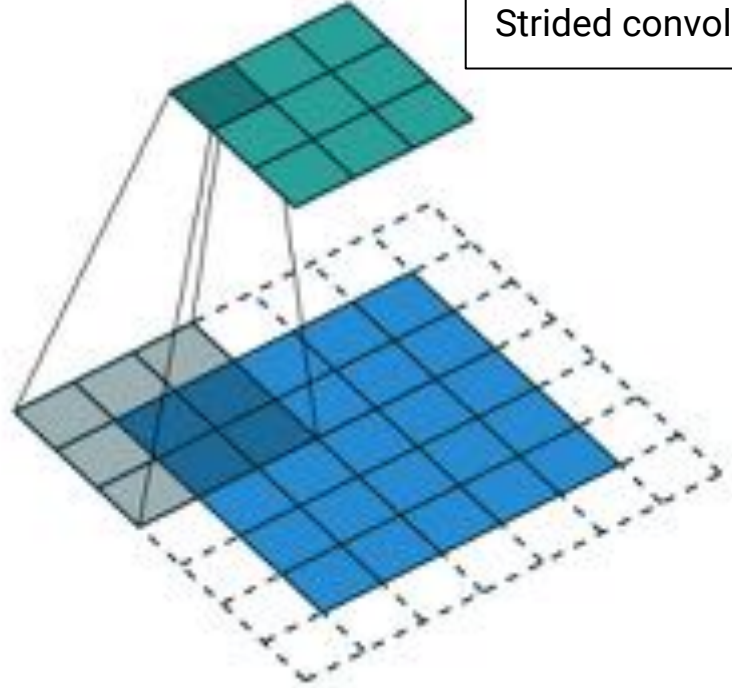


Strided convolutions

Standard convolution



Strided convolution



Multi-channel convolution: 3 channel RGB

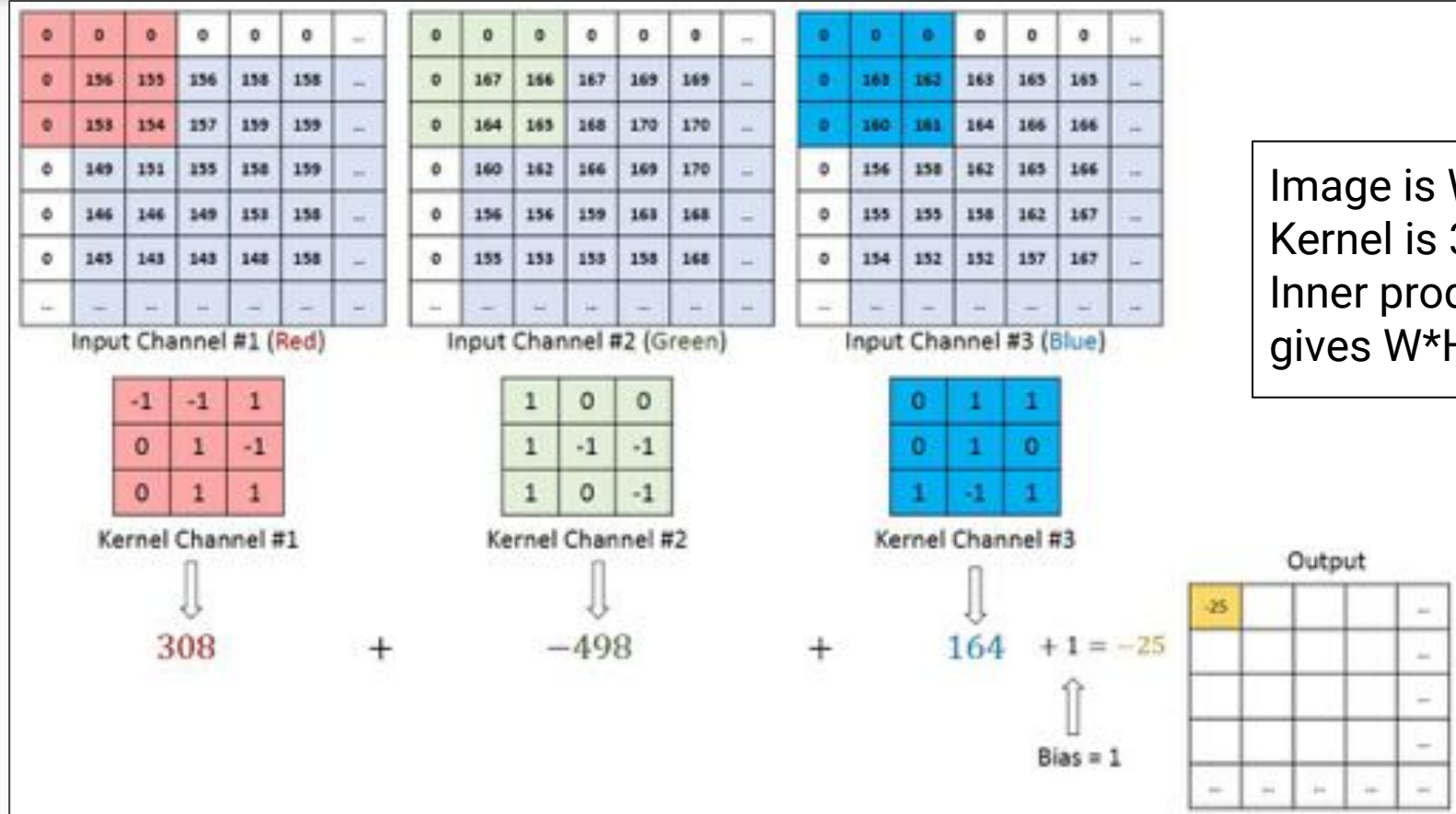
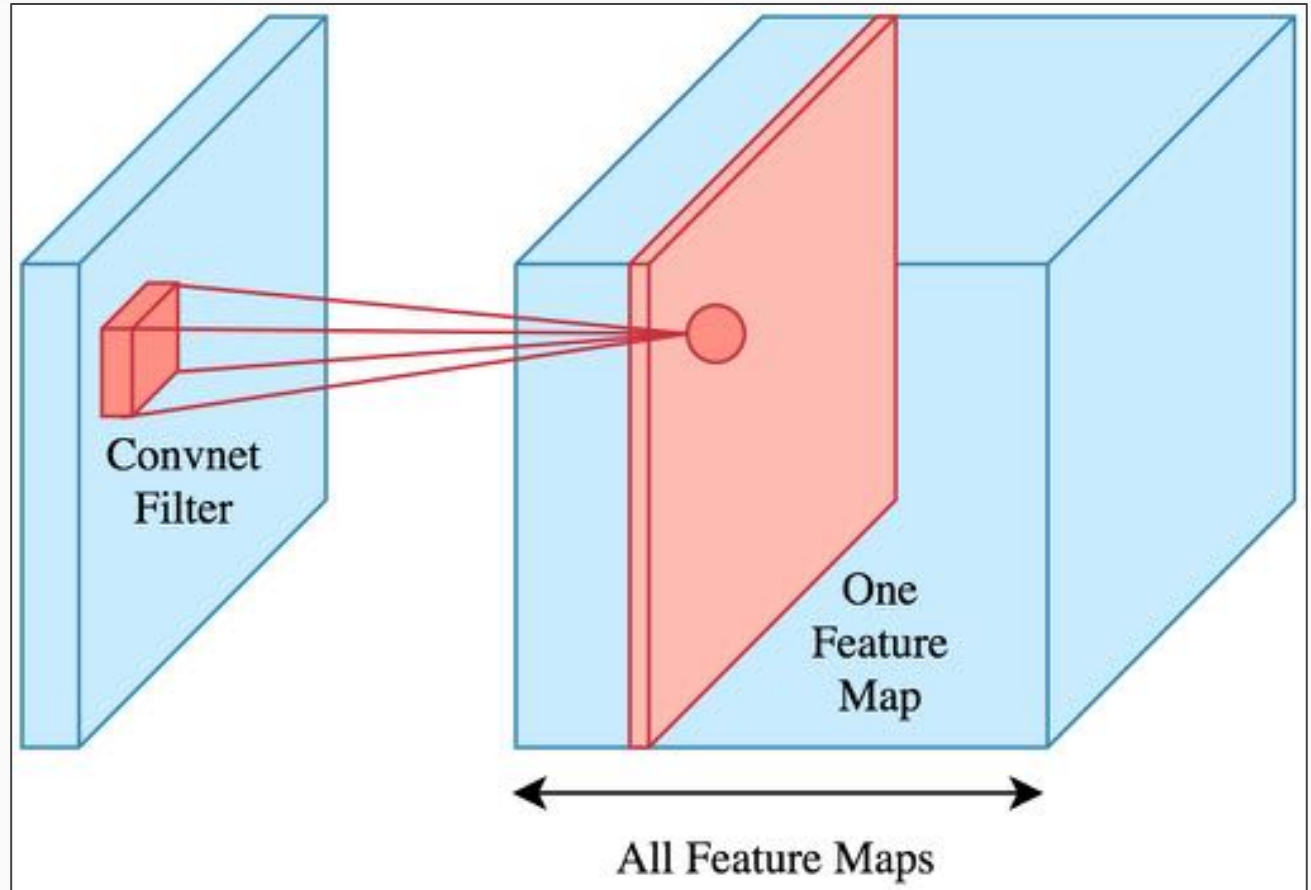


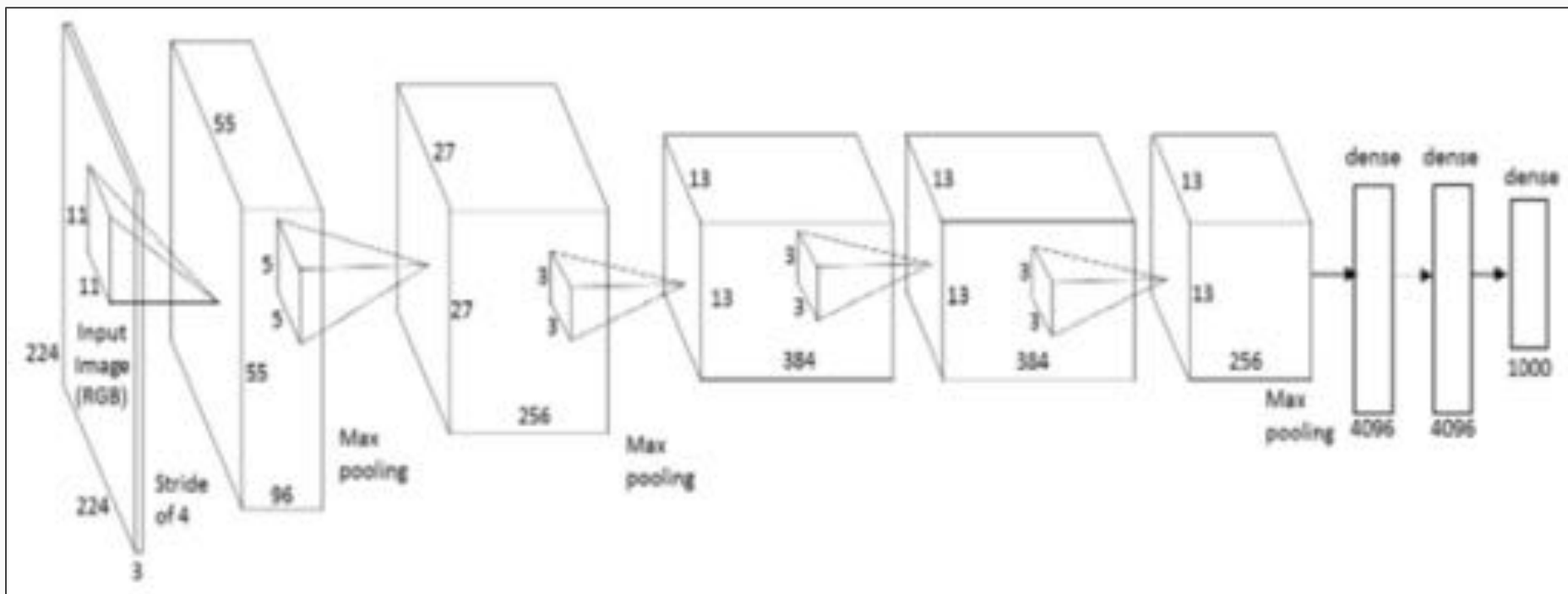
Image is $W \times H \times 3$
Kernel is $3 \times 3 \times 3$
Inner product
gives $W \times H \times 1$

Many channel convolution

In deep neural networks, channels are obtained by stacking the results of convolution operation on many kernels



A Deep Learning Model

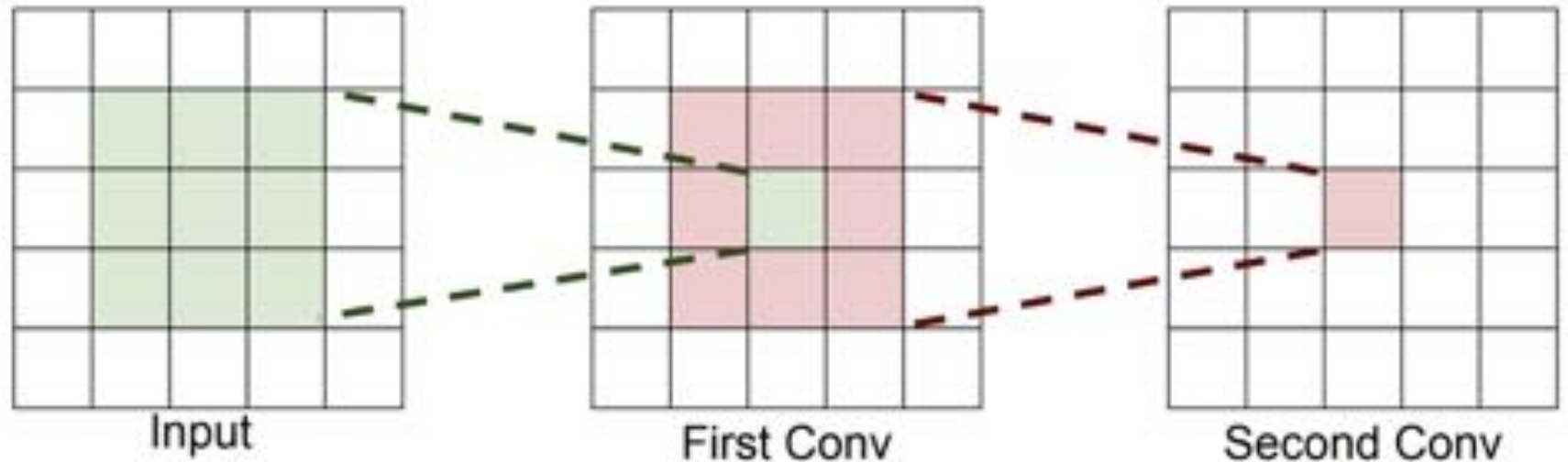


Why deep neural networks?

- There are several reasons, not all understood mathematically
- But from convolution pov, we ask the question:
 - How big of a region in the input does a neuron on the 2nd conv layer see?

Why deep neural networks?

- There are several reasons, not all understood mathematically
- But from convolution pov, we ask the question:
 - How big of a region in the input does a neuron on the 2nd conv layer see?

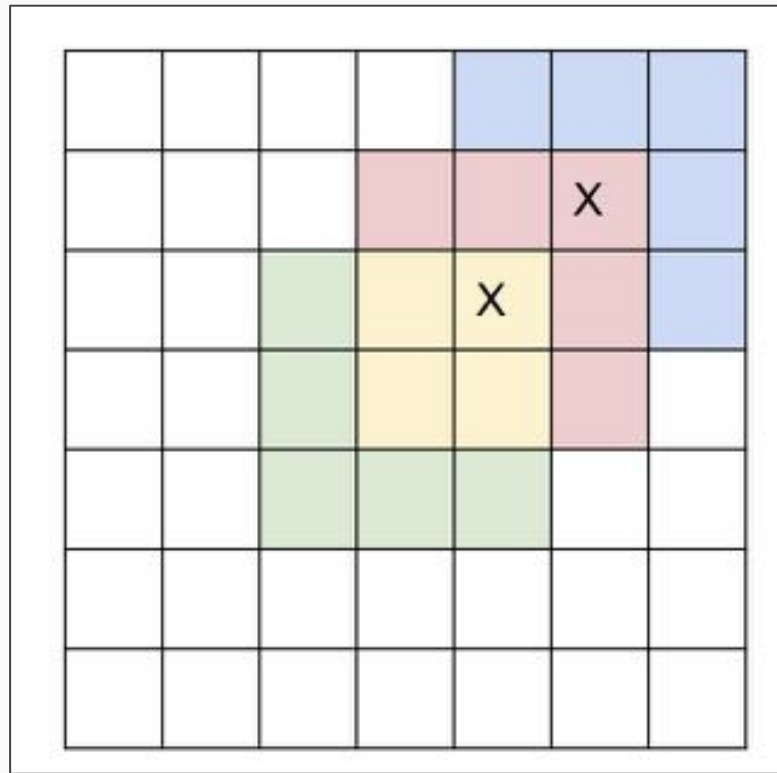


Why deep neural networks?

- If we stack 3x3 filters (or kernels)
what is the effective area seen by
the 3rd layer?

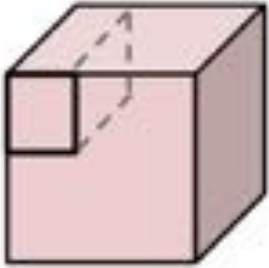
Why deep neural networks?

- If we stack 3x3 filters (or kernels) what is the effective area seen by the 3rd layer?
- Answer: 7x7
- Effect: Instead of $K^2 \times 49$ parameters we have $3 \times K^2 \times 9$ parameters.
=> Fewer parameters and more non-linearity

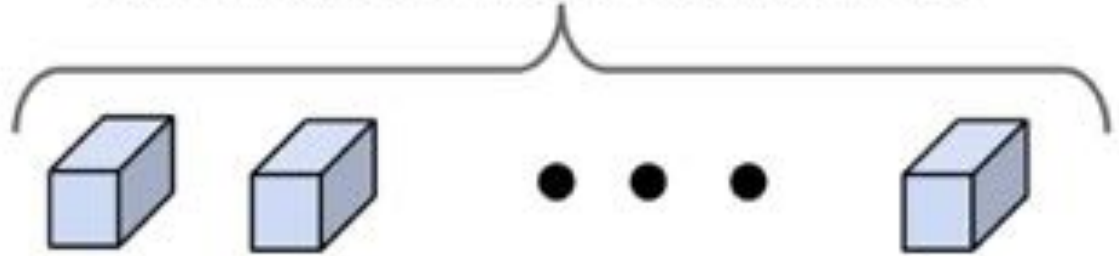


Implementing convolution as a matrix multiplication

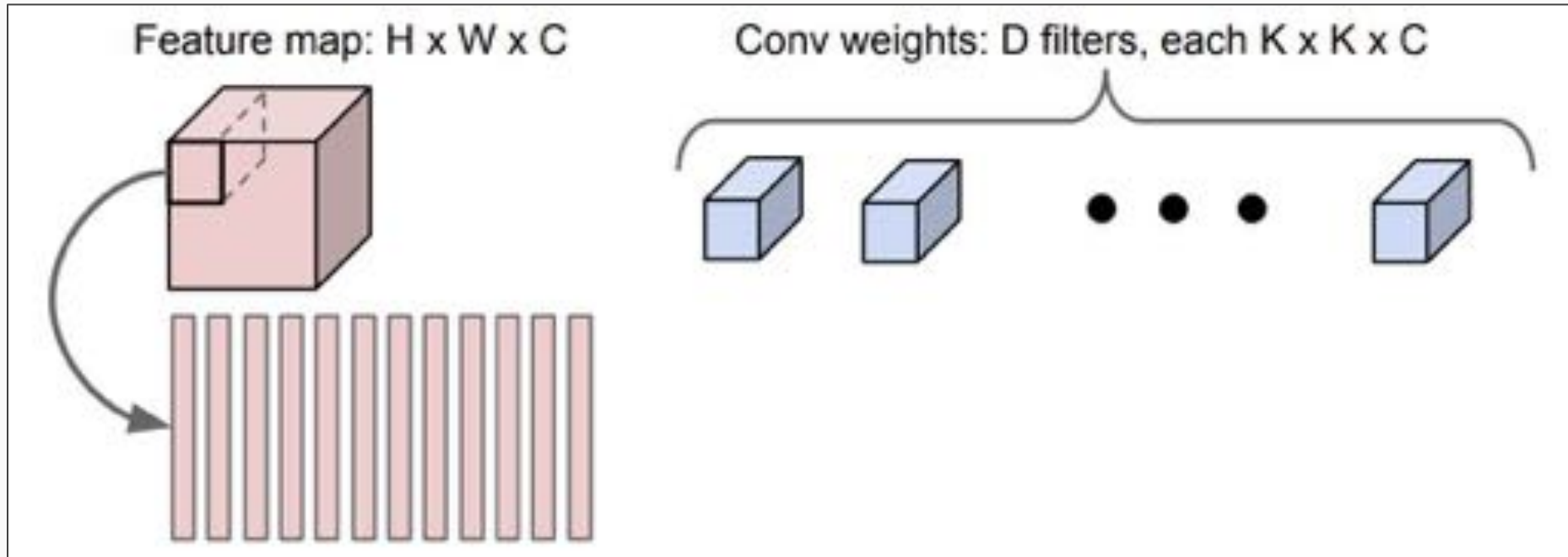
Feature map: $H \times W \times C$



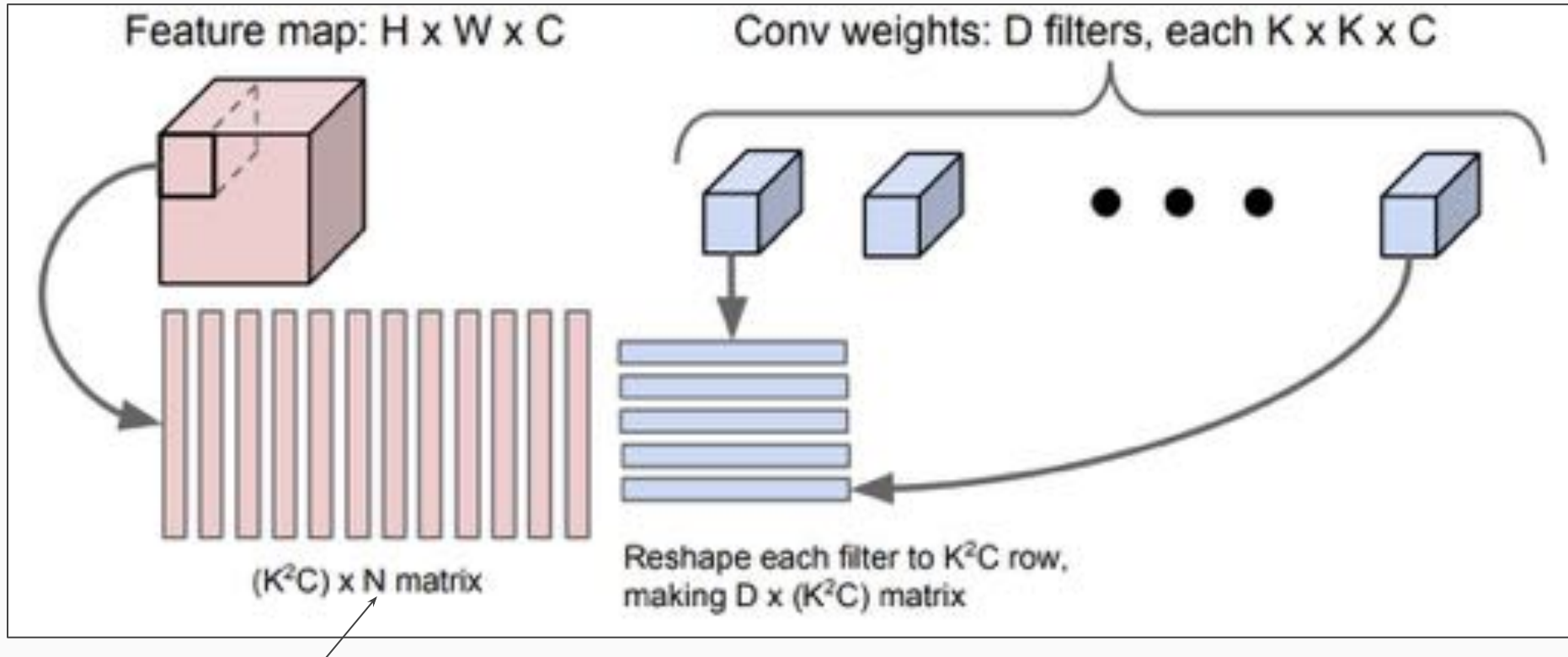
Conv weights: D filters, each $K \times K \times C$



Implementing convolution as a matrix multiplication



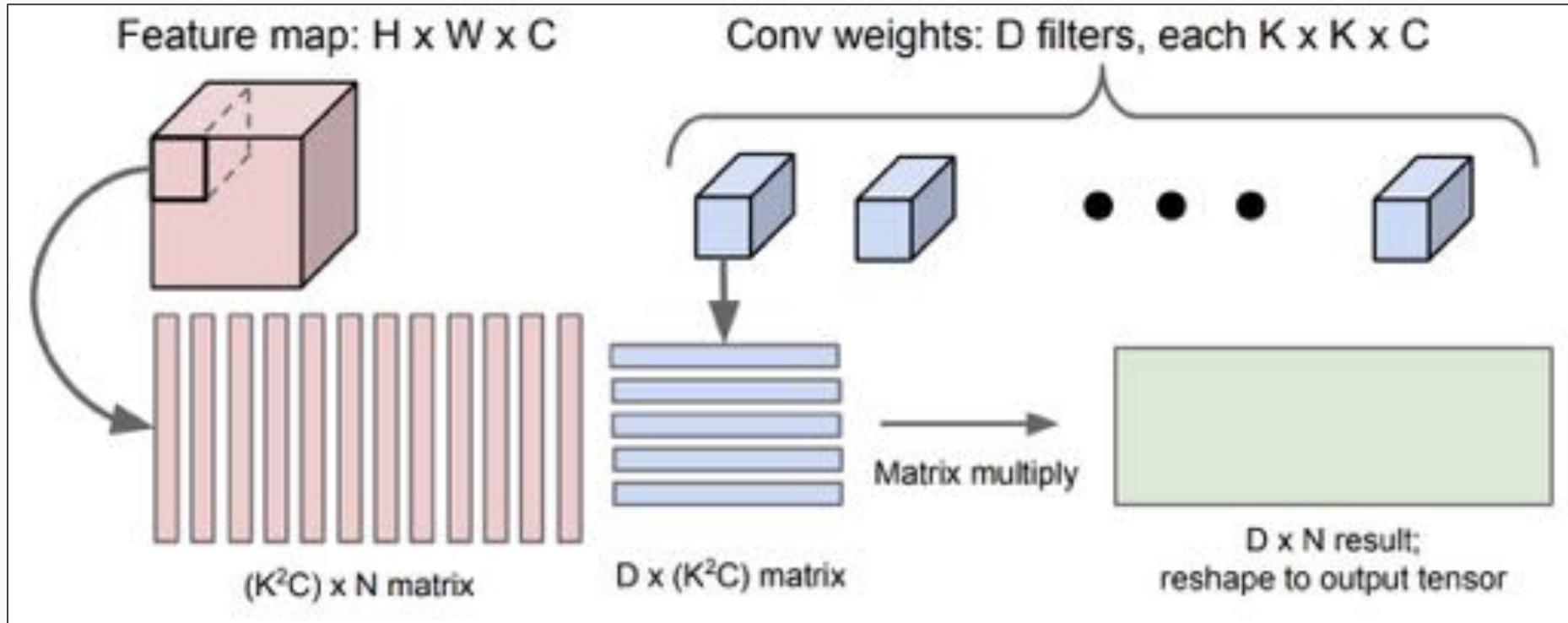
Implementing convolution as a matrix multiplication

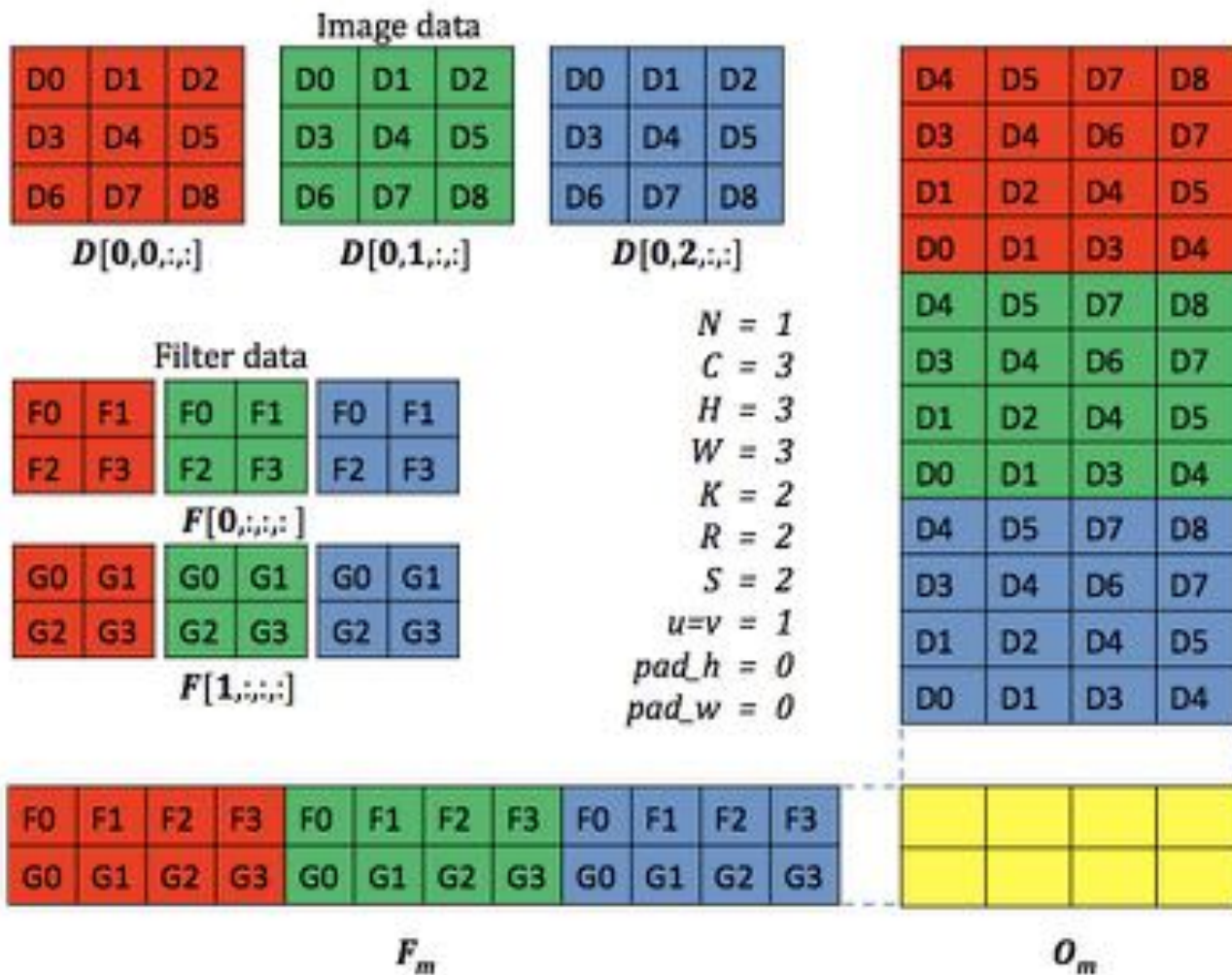


N is the number of K^2 slices in $H \times W$

http://cs231n.stanford.edu/slides/2016/winter1516_lecture11.pdf

Implementing convolution as a matrix multiplication





“cuDNN: Efficient Primitives for Deep Learning”,
<https://arxiv.org/pdf/1410.0759.pdf>

Implementing convolution as a matrix multiplication

- Pros due to implementation as matrix multiplication
 - Efficient CUDA GEMM
 - Similar to the tiling algorithm we read
 - Important difference: Do the fetching of tiles and computation on tiles in parallel
 - Hides latency even further
- Cons:
 - ?

Implementing convolution as a matrix multiplication

- Pros due to efficient implementation of matrix multiplication
 - CUDA GEMM
 - Similar to the tiling algorithm we read
 - Important difference: Do the fetching of tiles and computation on tiles in parallel
 - Hides latency even further
- Cons:
 - Duplication of data
 - Transformation step: Eg. `im2col` on CUDA from Caffe

Another approach - Using Fast Fourier Transform

- Convolution theorem

- Convolutions in the spatial domain are equivalent to point-wise multiplications in Fourier transforms

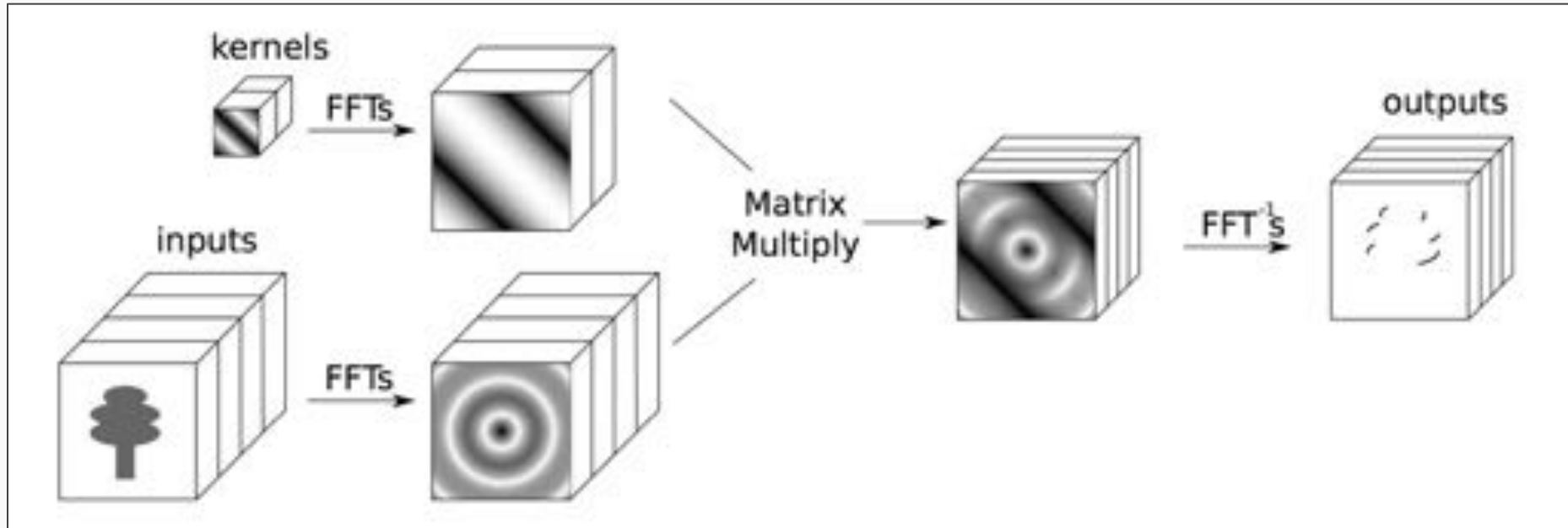
$$f * g = \mathcal{F}^{-1}(\mathcal{F}(f) \cdot \mathcal{F}(g))$$

- For the case of finite sequences, we can take discrete fourier and inverse fourier transforms

- Complexity

- Given an image of size $N \times N$ and kernel of size $K \times K$, the standard method uses $(N - K + 1)^2 \times K^2$ multiplications = $O(N^2 K^2)$ for typical K
- With FFT: Complexity of each FFT is $O(2N^2 \log N)$ and point-wise multiplication takes $O(N^2)$. Thus overall $O(N^2 \log N)$.
- FFT complexity independent of K because, fourier transform taken on the same support for both kernel and image => Does not benefit from small K

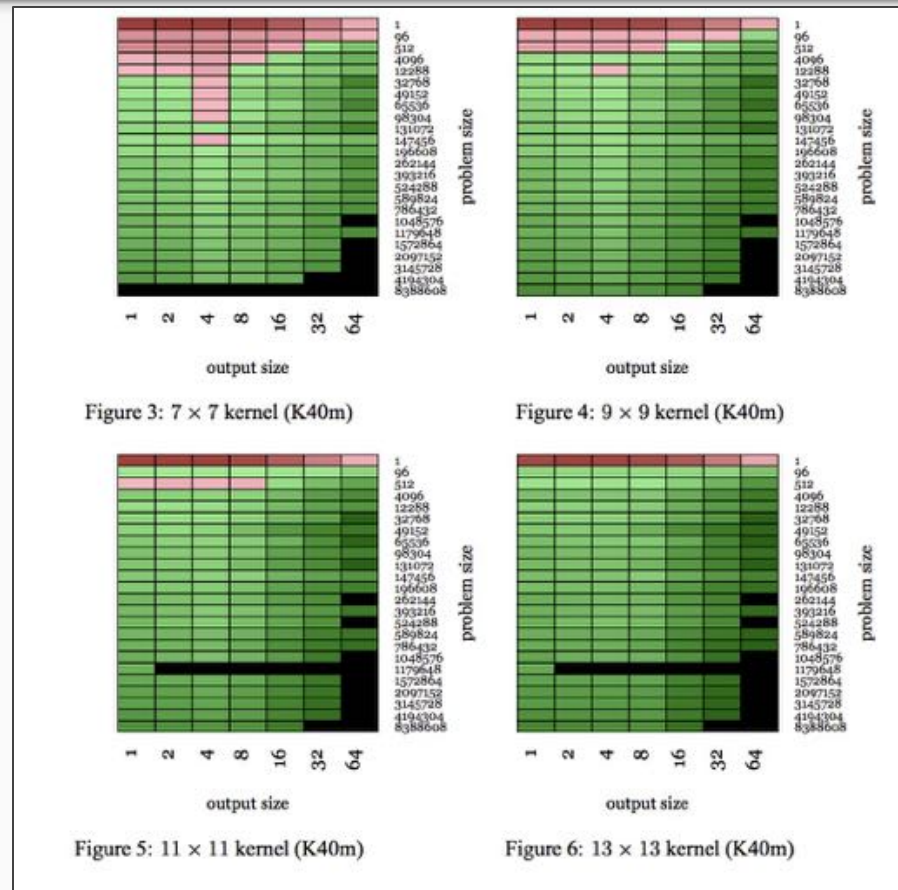
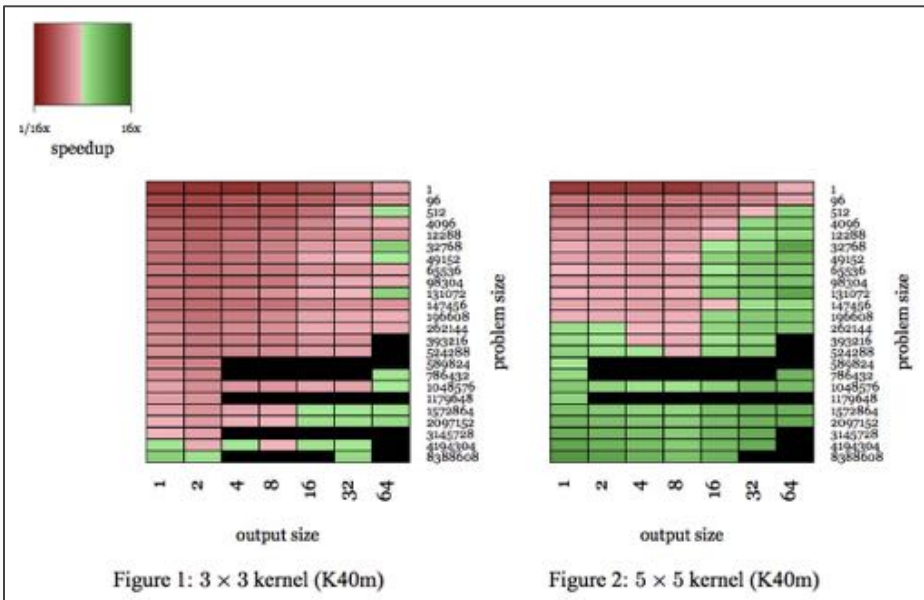
Convolutions with FFT



“Fast Training of Convolutional Networks through FFTs”,
<https://arxiv.org/pdf/1312.5851.pdf>

FFT performance vs standard convolution

Larger improvement for larger
kernel sizes. Does not work for 3x3.
Does not work for stride length > 1



Algorithmic approaches to convolution

- Similar to Strassen's Algorithm for matrix multiplication
 - Standard matrix multiplication of $N \times N$ matrices takes $O(N^3)$ ops
- Consider a simple multiplication problem
 - $P = (a + ib) \times (c + id) = (ac - bd) + i(bc + ad)$
 - 4 multiplications
 - Can we do better
- Define $m_1 = (a + b)(c - d)$, $m_2 = bc$, $m_3 = ad$
 - Then $P = (m_1 - m_2 + m_3) + i(m_2 + m_3)$
 - 3 multiplications

Strassen's algorithm for matrix multiplication

$$\mathbf{C} = \mathbf{A}\mathbf{B} \quad \mathbf{A}, \mathbf{B}, \mathbf{C} \in R^{2^n \times 2^n}$$

$$\mathbf{C}_{1,1} = \mathbf{A}_{1,1}\mathbf{B}_{1,1} + \mathbf{A}_{1,2}\mathbf{B}_{2,1}$$

$$\mathbf{C}_{1,2} = \mathbf{A}_{1,1}\mathbf{B}_{1,2} + \mathbf{A}_{1,2}\mathbf{B}_{2,2}$$

$$\mathbf{C}_{2,1} = \mathbf{A}_{2,1}\mathbf{B}_{1,1} + \mathbf{A}_{2,2}\mathbf{B}_{2,1}$$

$$\mathbf{C}_{2,2} = \mathbf{A}_{2,1}\mathbf{B}_{1,2} + \mathbf{A}_{2,2}\mathbf{B}_{2,2}$$

Standard algorithm
8 multiplications and 2 additions

$$\mathbf{M}_1 := (\mathbf{A}_{1,1} + \mathbf{A}_{2,2})(\mathbf{B}_{1,1} + \mathbf{B}_{2,2})$$

$$\mathbf{M}_2 := (\mathbf{A}_{2,1} + \mathbf{A}_{2,2})\mathbf{B}_{1,1}$$

$$\mathbf{M}_3 := \mathbf{A}_{1,1}(\mathbf{B}_{1,2} - \mathbf{B}_{2,2})$$

$$\mathbf{M}_4 := \mathbf{A}_{2,2}(\mathbf{B}_{2,1} - \mathbf{B}_{1,1})$$

$$\mathbf{M}_5 := (\mathbf{A}_{1,1} + \mathbf{A}_{1,2})\mathbf{B}_{2,2}$$

$$\mathbf{M}_6 := (\mathbf{A}_{2,1} - \mathbf{A}_{1,1})(\mathbf{B}_{1,1} + \mathbf{B}_{1,2})$$

$$\mathbf{M}_7 := (\mathbf{A}_{1,2} - \mathbf{A}_{2,2})(\mathbf{B}_{2,1} + \mathbf{B}_{2,2})$$

$$\mathbf{C}_{1,1} = \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7$$

$$\mathbf{C}_{1,2} = \mathbf{M}_3 + \mathbf{M}_5$$

$$\mathbf{C}_{2,1} = \mathbf{M}_2 + \mathbf{M}_4$$

$$\mathbf{C}_{2,2} = \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6$$

Strassen algorithm
7 multiplications and 18
additions/subtractions

Matrix multiplication algorithms

Complexity of matrix multiplication of $N \times N$ matrices is $O(N^\omega)$

Year	$\omega <$	
< 1969	3	
1969	2.81	Strassen
1978	2.79	Pan
1979	2.78	Bini et al
1981	2.55	Schonhage
1982	2.50	Pan; Romani; CW
1987	2.48	Strassen
1987	2.38	CW

Figure 1: Historical improvements in bounding ω

Similarly, for convolution operation - Winograd algorithm

Takes 6 multiplications in the standard approach

$$F(2, 3) = \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{bmatrix}$$

$$\begin{aligned} m_1 &= (d_0 - d_2)g_0 & m_2 &= (d_1 + d_2)\frac{g_0 + g_1 + g_2}{2} \\ m_4 &= (d_1 - d_3)g_2 & m_3 &= (d_2 - d_1)\frac{g_0 - g_1 + g_2}{2} \end{aligned}$$

With Winograd algo, we need 4 multiplications + 2 multiplications with constants

Applying similar fast algorithms for convolution

N is the batch size (number of images
simultaneously convolved)

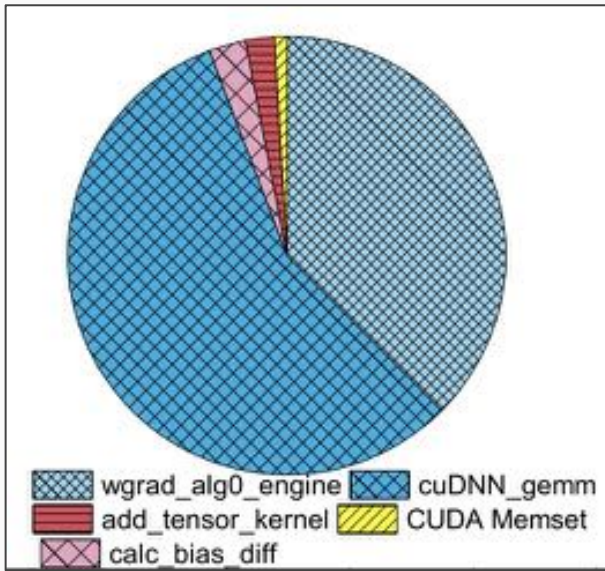
N	cuDNN		F(2x2,3x3)		Speedup
	msec	TFLOPS	msec	TFLOPS	
1	12.52	3.12	5.53	7.03	2.26X
2	20.36	3.83	9.89	7.89	2.06X
4	104.70	1.49	17.72	8.81	5.91X
8	241.21	1.29	33.11	9.43	7.28X
16	203.09	3.07	65.79	9.49	3.09X
32	237.05	5.27	132.36	9.43	1.79X
64	394.05	6.34	266.48	9.37	1.48X

Table 5. cuDNN versus $F(2 \times 2, 3 \times 3)$ performance on VGG Network E with fp32 data. Throughput is measured in Effective TFLOPS, the ratio of direct algorithm GFLOPs to run time.

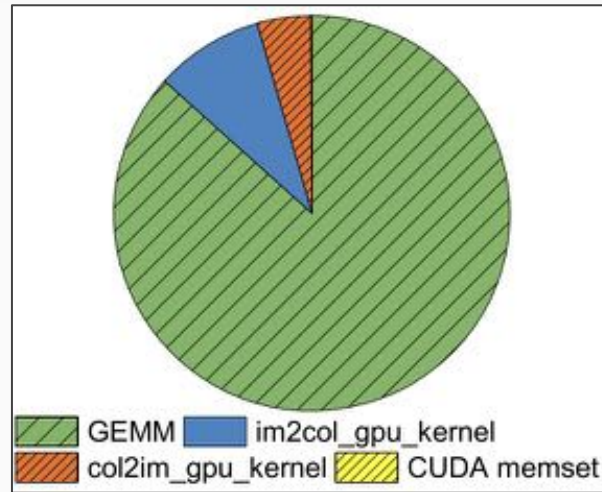
N	cuDNN		F(2x2,3x3)		Speedup
	msec	TFLOPS	msec	TFLOPS	
1	14.58	2.68	5.53	7.06	2.64X
2	20.94	3.73	9.83	7.94	2.13X
4	104.19	1.50	17.50	8.92	5.95X
8	241.87	1.29	32.61	9.57	7.42X
16	204.01	3.06	62.93	9.92	3.24X
32	236.13	5.29	123.12	10.14	1.92X
64	395.93	6.31	242.98	10.28	1.63X

Table 6. cuDNN versus $F(2 \times 2, 3 \times 3)$ performance on VGG Network E with fp16 data.

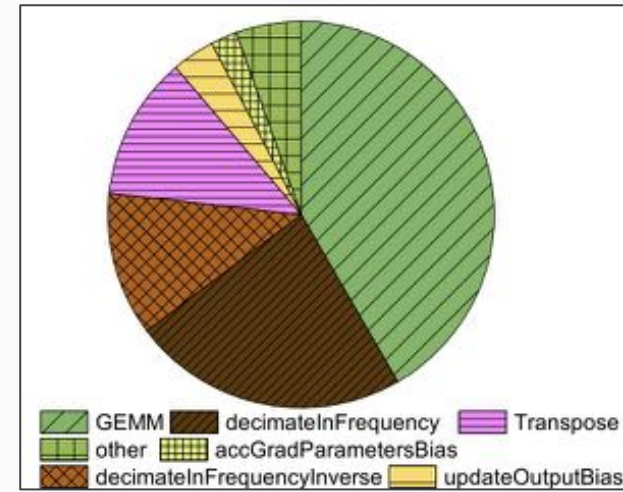
Comparison of different approaches



cuDNN with Winograd



Caffe with GEMM/im2col



fbfft with FFT inner product

Runtime breakdowns of convolutional layers in different implementations

Dynamic Shared Memory

Static Shared Memory

```
__global__ void staticReverse(int *d, int n)
{
    __shared__ int s[64];
    int t = threadIdx.x;
    int tr = n-t-1;
    s[t] = d[t];
    __syncthreads();
    d[t] = s[tr];
}

....

cudaMemcpy(d_d, a, n*sizeof(int), cudaMemcpyHostToDevice);
staticReverse<<<1,n>>>(d_d, n);
cudaMemcpy(d, d_d, n*sizeof(int), cudaMemcpyDeviceToHost);
```

Dynamic Shared Memory

```
__global__ void dynamicReverse(int *d, int n)
{
    extern __shared__ int s[];
    int t = threadIdx.x;
    int tr = n-t-1;
    s[t] = d[t];
    __syncthreads();
    d[t] = s[tr];
}

...

cudaMemcpy(d_d, a, n*sizeof(int), cudaMemcpyHostToDevice);
dynamicReverse<<<1,n,n*sizeof(int)>>>(d_d, n);
cudaMemcpy(d, d_d, n * sizeof(int), cudaMemcpyDeviceToHost);
```

Dynamic Shared Memory

```
__global__ void dynamicReverse(int *d, int n)
{
    extern __shared__ int s[];
    int t = threadIdx.x;
    int tr = n-t-1;
    s[t] = d[t];
    __syncthreads();
    d[t] = s[tr];
}
```

1. Declare the shared array with extern qualifier
2. Declare the shared array as unsized
3. The invocation of the kernel has a third parameter that gives the size of the shared memory per block in bytes

...

```
cudaMemcpy(d_d, a, n*sizeof(int), cudaMemcpyHostToDevice);
dynamicReverse<<<1, n, n*sizeof(int)>>>(d_d, n);
cudaMemcpy(d, d_d, n * sizeof(int), cudaMemcpyDeviceToHost);
```