



ECE408/CS483/CSE408

Applied Parallel Programming

Lecture 12: Convolutional Neural Networks

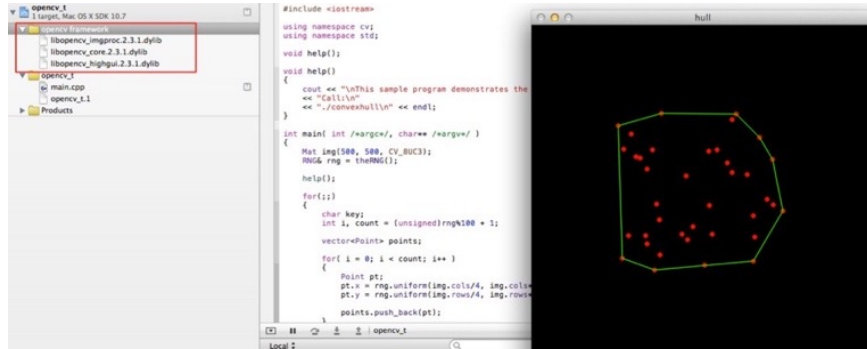
Course Reminders

- **Lecture for Tuesday Oct 3 will be a pre-recorded video lecture. No need to come to class!**
- Lab 4 is due on Friday
- Midterm 1 is on Tuesday, October 10th
- Project Milestone 1: Baseline CPU implementation is due Friday October 13th
 - Project details will be provided by end of this week

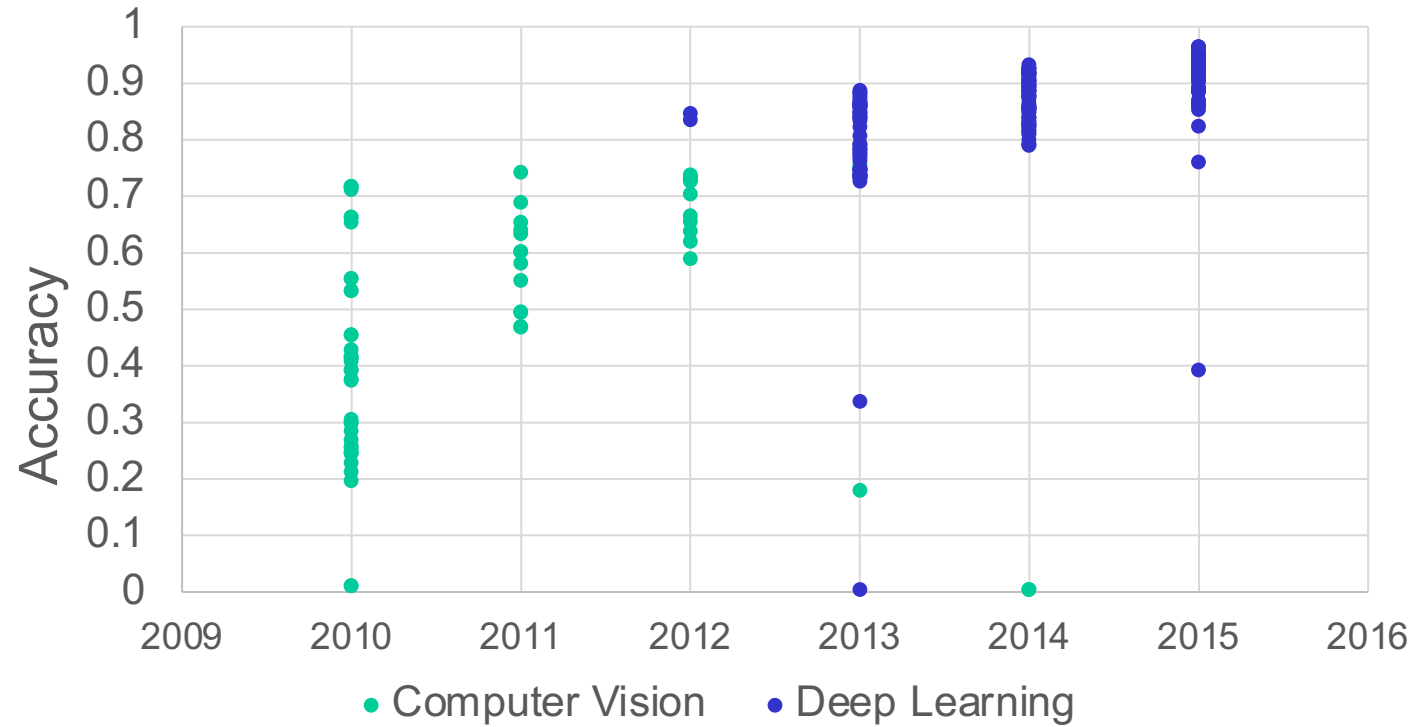
Deep Learning in Computer Vision



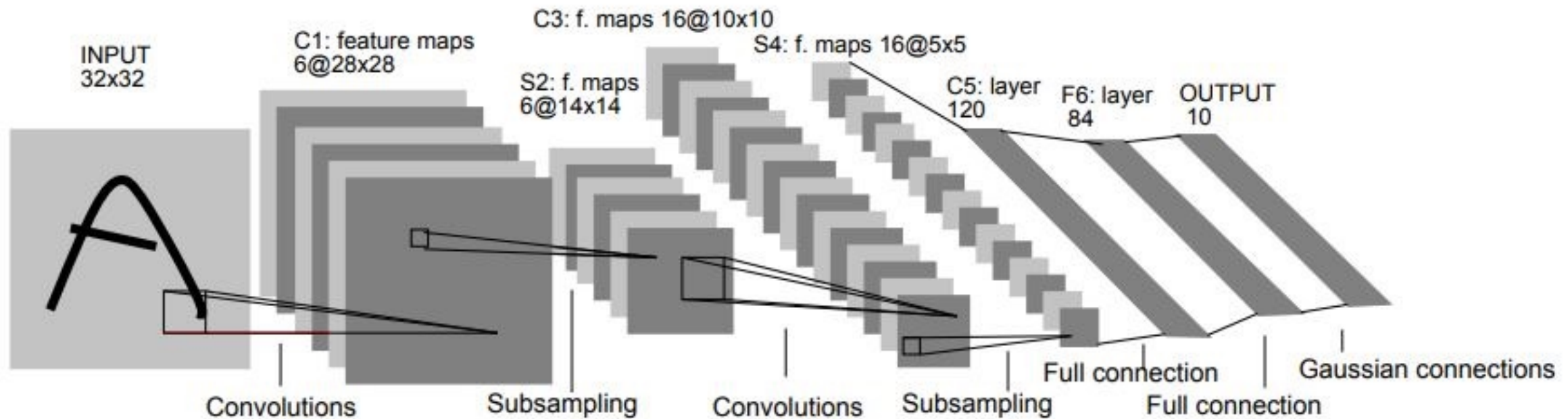
2012 Large Scale Image Recognition Challenge



U of Toronto team used GPUs and trained on 1.2M images in their 2012 winning entry.



LeNet-5: CNN for hand-written digit recognition



Anatomy of a Convolution Layer

Input

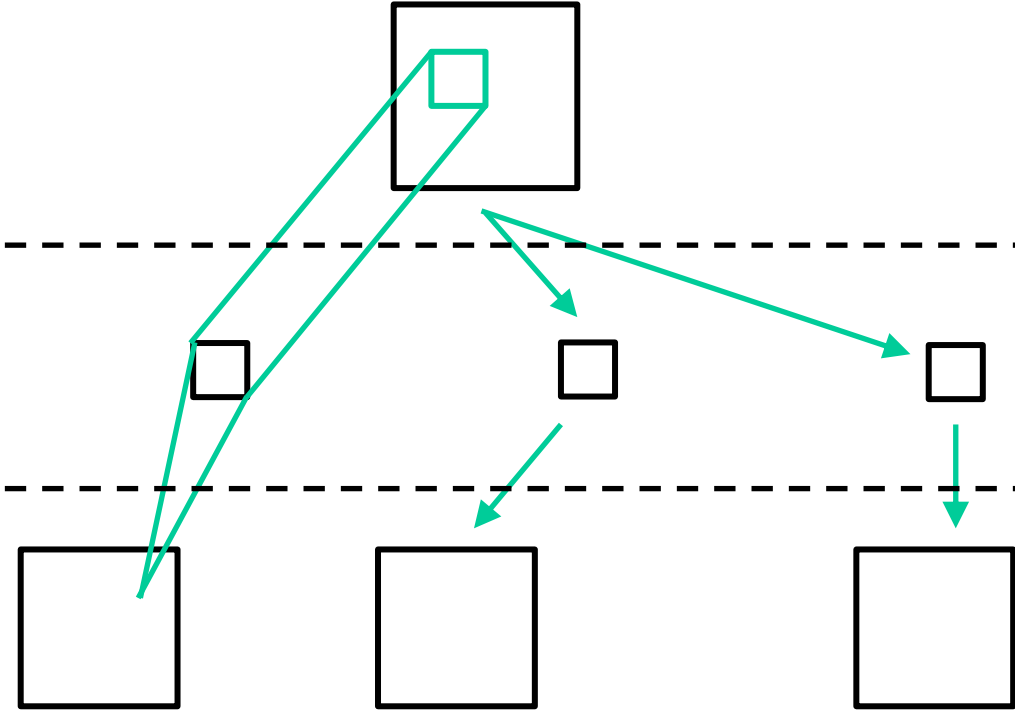
- A inputs each $N_1 \times N_2$

Convolution Layer

- B convolution kernels each $K_1 \times K_2$

Output (total of B)

- A × B outputs each $(N_1 - K_1 + 1) \times (N_2 - K_2 + 1)$



Notion of a Channel in Input Layer

Some Set of Inputs are Related

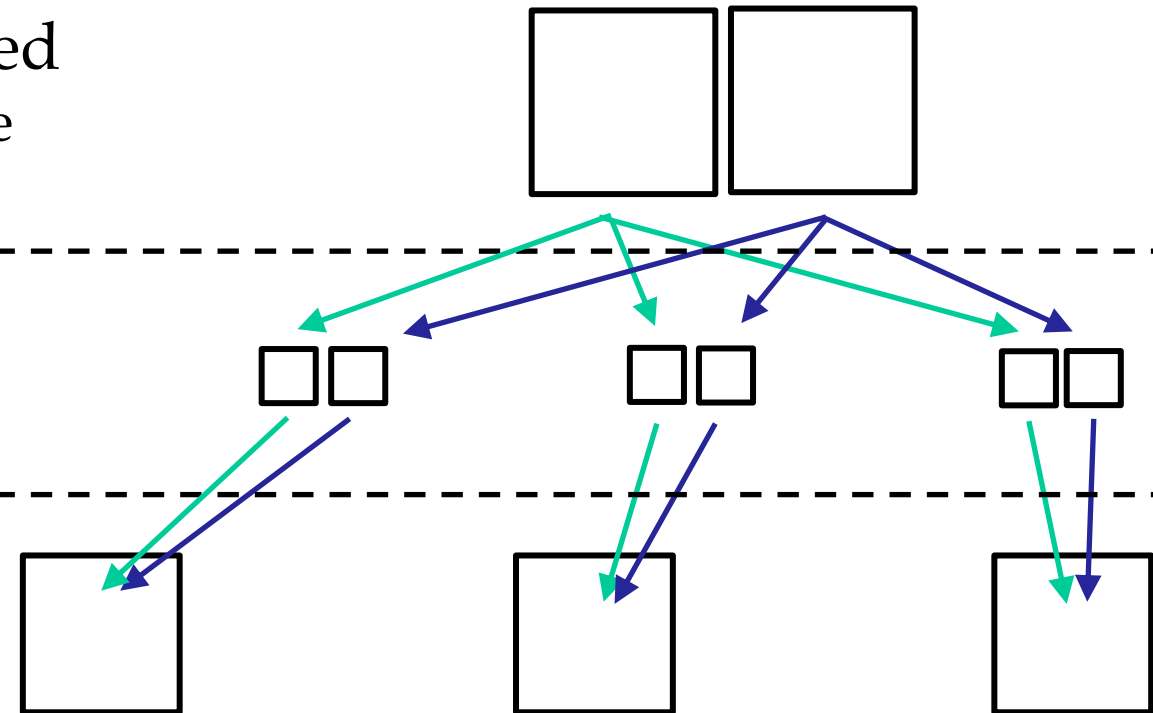
- For example: Red, Green, Blue

Convolution Layer

- Different kernels per channel

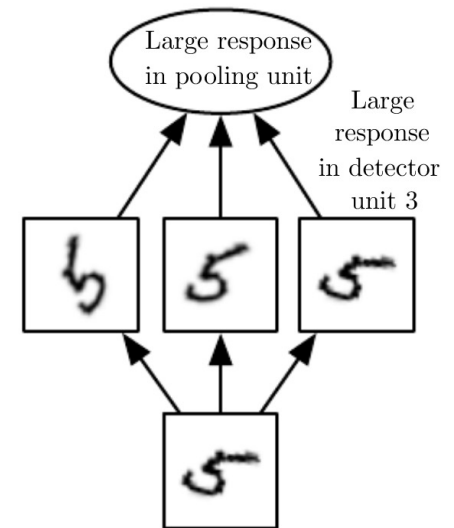
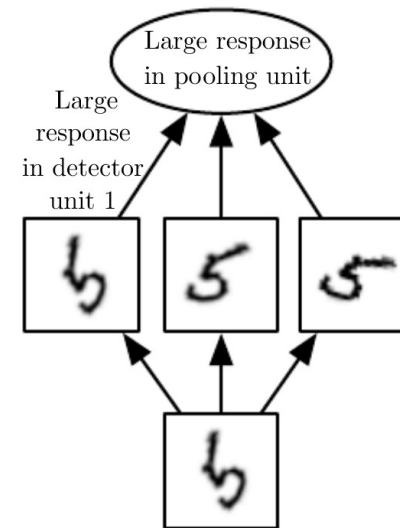
Output

- Channels combine per output

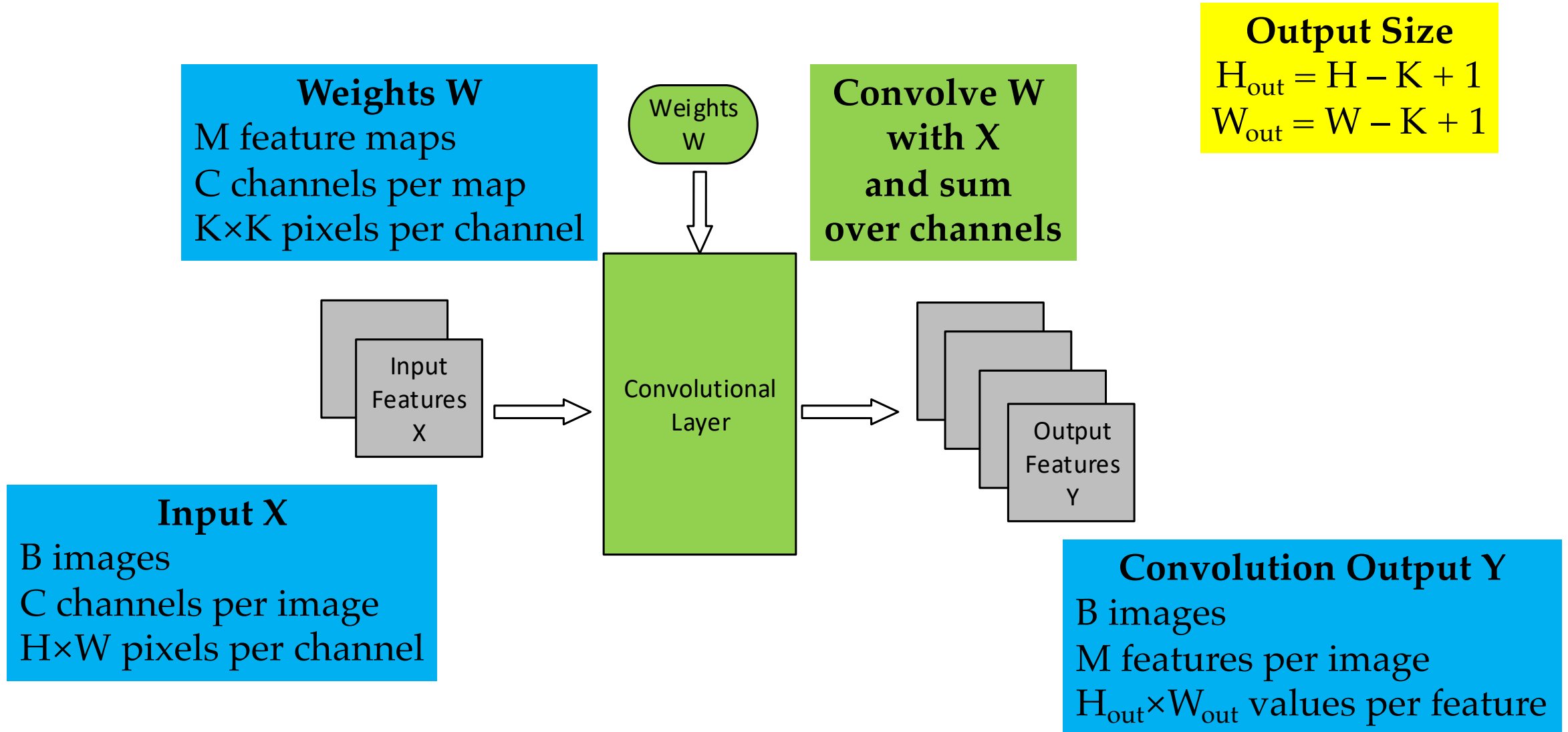


Pooling (Subsampling)

- Subsampling layer
 - Sometimes with bias and non-linearity built in
- Common types
 - max, average, L^2 norm, weighted average
- Helps make representation invariant to size scaling and small translations in the input



Forward Propagation



Outputs Typically Truncate Input

X

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

Compute only
this part of Y.

Y

		321				

W

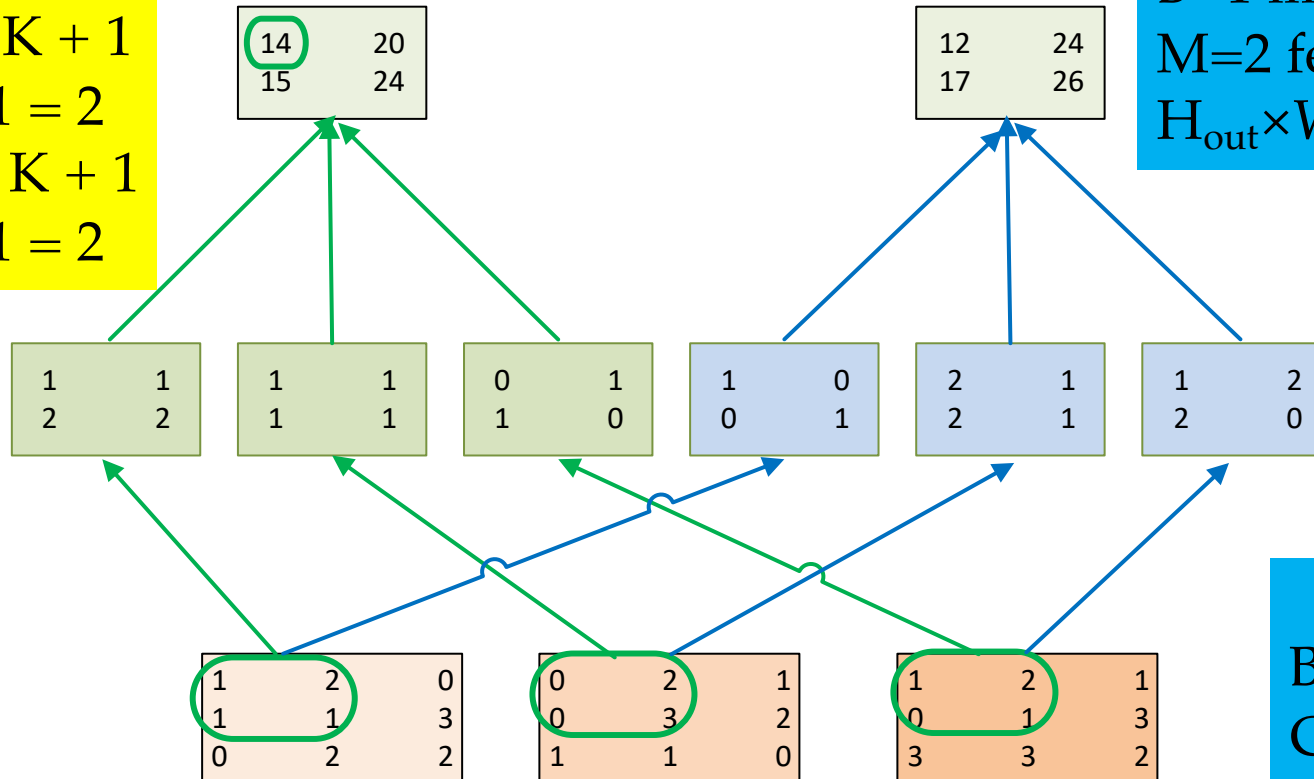
1	2	3	2	1
2	3	4	3	2
3	4	5	4	3
2	3	4	3	2
1	2	3	2	1

1	4	9	8	5
4	9	16	15	12
9	16	25	24	21
8	15	24	21	16
5	12	21	16	5

Example of the Forward Path of a Convolution Layer

Output Size

$$H_{\text{out}} = H - K + 1$$
$$= 3 - 2 + 1 = 2$$
$$W_{\text{out}} = W - K + 1$$
$$= 3 - 2 + 1 = 2$$



Convolution Output Y

B=1 image
M=2 features per image
 $H_{\text{out}} \times W_{\text{out}} = 2 \times 2$ values per feature

Weights W

M=2 feature maps
C=3 channels per map
 $K \times K = 2 \times 2$ pixels per channel

Input X

B=1 image
C=3 channels
 $H \times W = 3 \times 3$ pixels per channel

Sequential Code: Forward Convolutional Layer

```
void convLayer_forward(int B, int M, int C, int H, int W, int K, float* X, float* W, float* Y) {  
    int H_out = H - K + 1;           // calculate H_out, W_out  
    int W_out = W - K + 1;  
  
    for (int b = 0; b < B; ++b)      // for each image  
        for(int m = 0; m < M; m++)    // for each output feature map  
            for(int h = 0; h < H_out; h++) // for each output value (two loops)  
                for(int w = 0; w < W_out; w++) {  
                    Y[b, m, h, w] = 0.0f; // initialize sum to 0  
                    for(int c = 0; c < C; c++) // sum over all input channels  
                        for(int p = 0; p < K; p++) // KxK filter  
                            for(int q = 0; q < K; q++)  
                                Y[b, m, h, w] += X[b, c, h + p, w + q] * W[m, c, p, q];  
                }  
    }  
}
```

A Small Convolution Layer Example

Image b in mini batch

$x[b,0,_,_]$

1	2	0	1
1	1	3	2
0	2	2	0
2	1	0	3

1	1	1
2	2	3
2	1	0

$w[0,0,_,_]$

$x[b,1,_,_]$

0	2	1	0
0	3	2	1
1	1	0	2
2	1	0	3

1	2	3
1	1	0
3	0	1

$w[0,1,_,_]$

0	?
?	?

$y[b,0,_,_]$

$x[b, 1,_,_]$

$w[0,1,_,_]$

$y[b, 0,_,_]$

output map

$x[b,2,_,_]$

1	2	1	0
0	1	3	2
3	3	2	0
1	3	2	0

0	1	1
1	0	2
1	2	1

$w[0,2,_,_]$

A Small Convolution Layer Example

$c = 0$

$x[b,0,_,_]$

1	2	0	1
1	1	3	2
0	2	2	0
2	1	0	3

$x[b,1,_,_]$

0	2	1	0
0	3	2	1
1	1	0	2
2	1	0	3

$x[b,2,_,_]$

1	2	1	0
0	1	3	2
3	3	2	0
1	3	2	0

$w[0,0,_,_]$

1	1	1
2	2	3
2	1	0

$w[0,1,_,_]$

1	2	3
1	1	0
3	0	1

$w[0,2,_,_]$

0	1	1
1	0	2
1	2	1

$3+13+2$



18	?
?	?

$y[b,0,_,_]$

A Small Convolution Layer Example

$c = 1$

X[b,0,_,_]

1	2	0	1
1	1	3	2
0	2	2	0
2	1	0	3

X[b,1,_,_]

0	2	1	0
0	3	2	1
1	1	0	2
2	1	0	3

X[b,2,_,_]

1	2	1	0
0	1	3	2
3	3	2	0
1	3	2	0

1	1	1
2	2	3
2	1	0

w[0,0,_,_]

... 18+7+3+3

W[0,1,_,_]

1	2	3
1	1	0
3	0	1

$w[0,1,_,_]$

31	?
?	?

$y[b,0,_,_]$

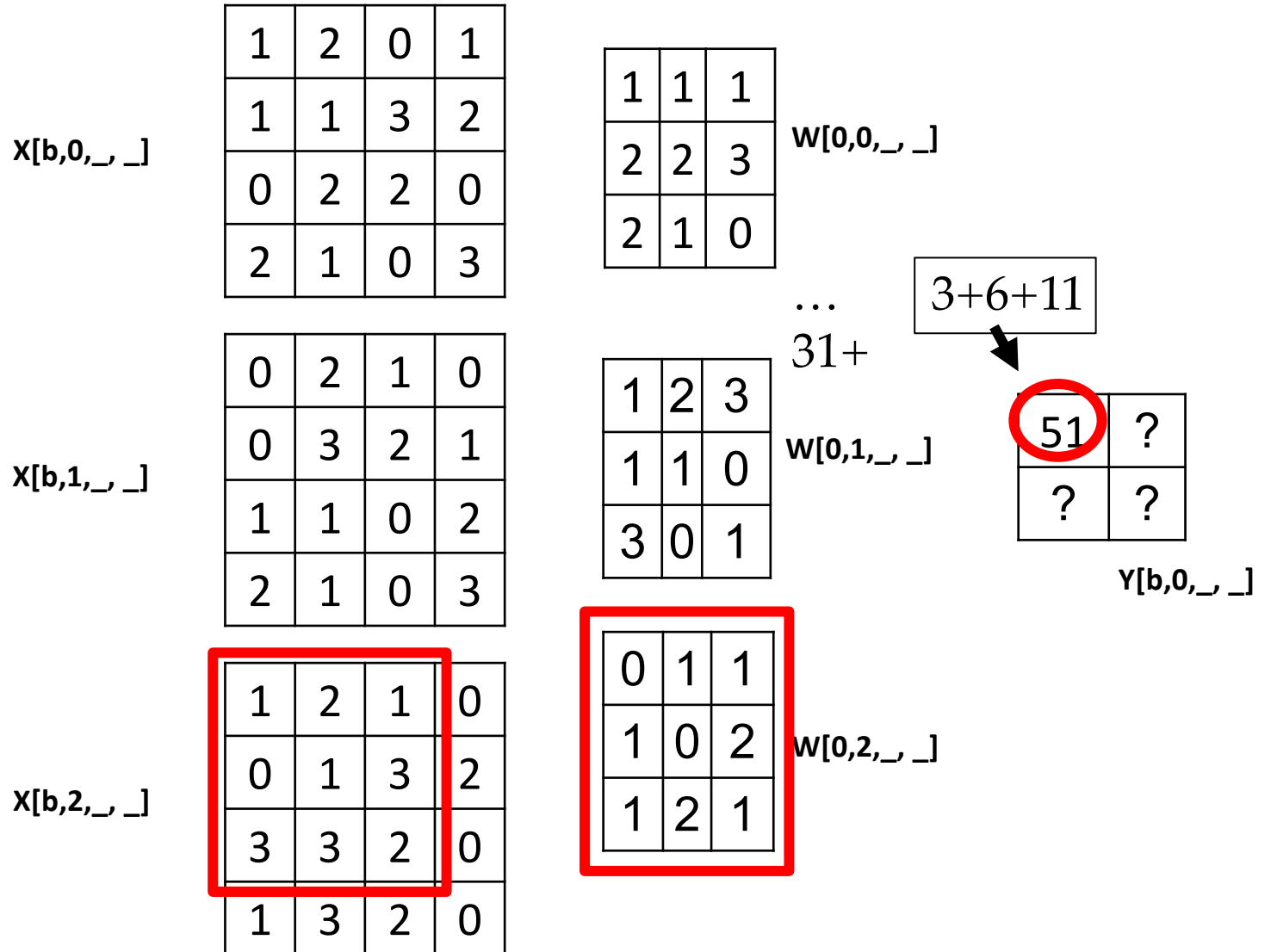
Y[b,0,_,_]

0	1	1
1	0	2
1	2	1

w[0,2,_,_]

A Small Convolution Layer Example

$c = 2$



Parallelism in a Convolution Layer

Feature maps can be calculated in parallel

- Usually a small number, not sufficient to fully utilize a GPU
- We'll need to handle tree reduction for features using channels (we'll learn this later)

All **output** feature map **pixels** can be calculated in parallel

- All rows can be done in parallel
- All pixels in each row can be done in parallel
- Large number but diminishes as we go into deeper layers

Different layers may demand different strategies.

Subsampling/Pooling by Scale N

Convolution Output Y

B images

M features per image

$H_{\text{out}} \times W_{\text{out}}$ values per feature

Average over $N \times N$
blocks,

then calculate sigmoid

Subsampling/Pooling Output S

B images

M features per image

$H_{S(N)} \times W_{S(N)}$ values per feature

Output Size

$$H_{S(N)} = \text{floor} (H_{\text{out}} / N)$$

$$W_{S(N)} = \text{floor} (W_{\text{out}} / N)$$

Sequential Code: Forward Pooling Layer

```
void poolingLayer_forward(int B, int M, int H_out, int W_out, int N, float* Y, float* S)
{
    for (int b = 0; b < B; ++b)                // for each image
        for (int m = 0; m < M; ++m)            // for each output feature map
            for (int x = 0; x < H_out/N; ++x)    // for each output value (two loops)
                for (int y = 0; y < W_out/N; ++y) {
                    float acc = 0.0f              // initialize sum to 0
                    for (int p = 0; p < N; ++p)    // loop over NxN block of Y (two loops)
                        for (int q = 0; q < N; ++q)
                            acc += Y[b, m, N*x + p, N*y + q];
                    acc /= N * N;                  // calculate average over block
                    S[b, m, x, y] = sigmoid(acc + bias[m]) // bias, non-linearity
                }
    }
```

Kernel Implementation of Subsampling Layer

- Straightforward mapping from grid to subsampled output feature map pixels
- in GPU kernel,
 - need to manipulate index mapping
 - for accessing the output feature map pixels
 - of the previous convolution layer.
- Often merged into the previous convolution layer to save memory bandwidth

Design of a Basic Kernel

- Each block computes
 - a tile of output pixels for one feature
 - TILE_WIDTH pixels in each dimension
- Grid's X dimension maps to M output feature maps.
- Grid's Y dimension maps to the tiles in each output feature map.
- (Grid's Z dimension is used for images in batch, which we omit from slides.)

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19

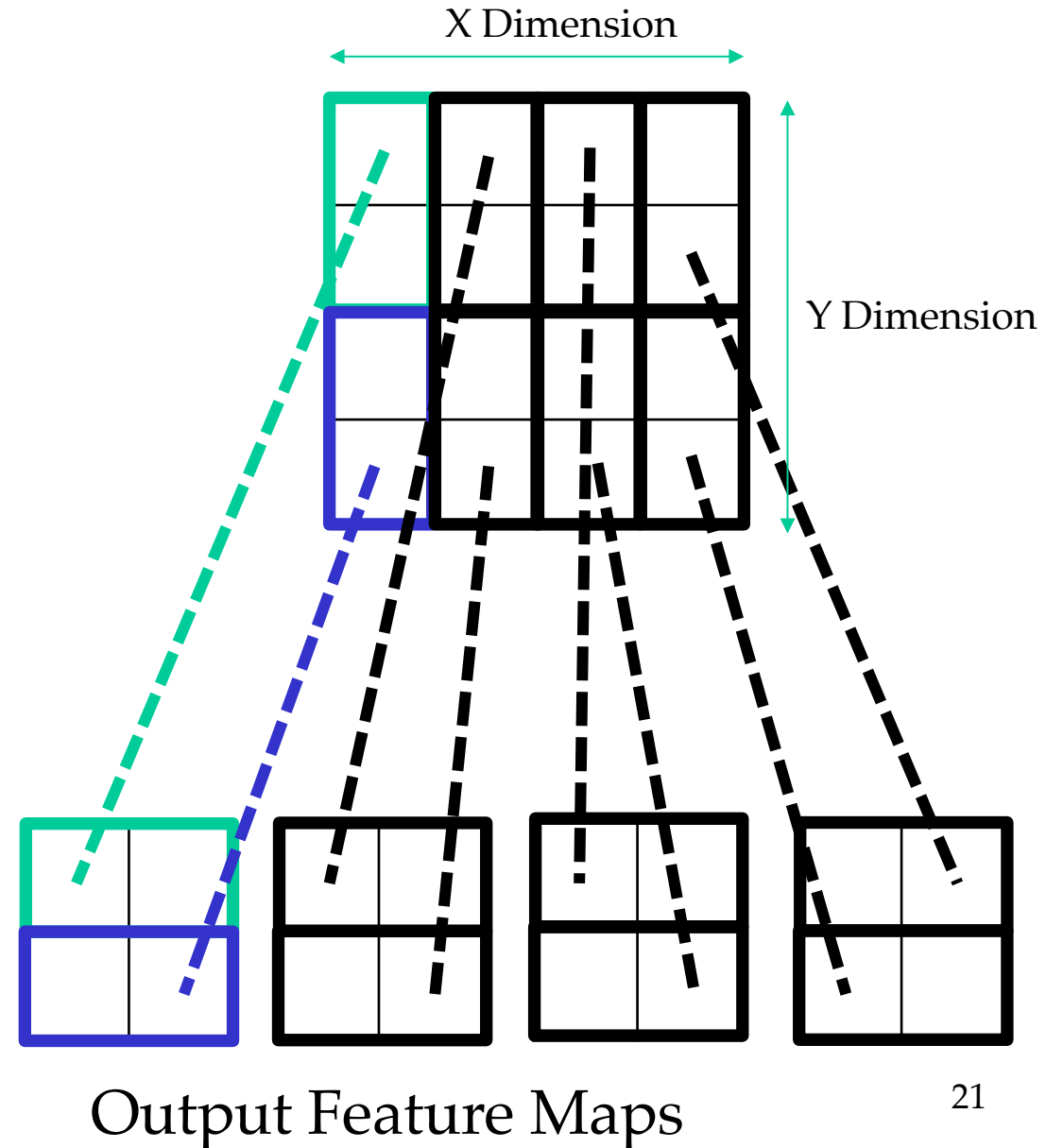
A Small Example

Assume

- **M = 4** (4 output feature maps),
- thus 4 blocks in the X dimension, and
- **W_out = H_out = 8** (8x8 output features).

If **TILE_WIDTH = 4**,
we also need 4 blocks in the Y dimension:

- for each output feature,
- top two blocks in each column calculates the top row of tiles, and
- bottom two calculate the bottom row.



Overall CUDA Approach

Consider an output feature map:

- width is **W_out**, and
- height is **H_out**.
- Assume these are multiples of **TILE_WIDTH**.

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19

Let **W_size** be the number of blocks needed in X dim (5 above).

Let **H_size** be the number of blocks needed in Y dim (4 above).

Host Code for a Basic Kernel

(Assuming W_{out} and H_{out} are multiples of $TILE_WIDTH$.)

```
#define TILE_WIDTH 16
W_size = W_out/TILE_WIDTH; // number of horizontal tiles per output map
H_size = H_out/TILE_WIDTH; // number of vertical tiles per output map
Y = H_size * W_size;      // total number of tiles per map

dim3 blockDim(TILE_WIDTH, TILE_WIDTH, 1); // output tile for untiled code
dim3 gridDim(M, Y, 1);

ConvLayerForward_Kernel<<< gridDim, blockDim >>>(...);
```

Partial Kernel Code for a Convolution Layer

```
__global__ void ConvLayerForward_Basic_Kernel
(int C, int W_size, int K, float* X, float* W, float* Y)
{
    int m = blockIdx.x;
    int h = (blockIdx.y / W_size) * TILE_WIDTH + threadIdx.y;
    int w = (blockIdx.y % W_size) * TILE_WIDTH + threadIdx.x;
    float acc = 0.0f;
    for (int c = 0; c < C; c++) {                // sum over all input channels
        for (int p = 0; p < K; p++)                // loop over KxK filter
            for (int q = 0; q < K; q++)
                acc += X[c, h + p, w + q] * W[m, c, p, q];
    }
    Y[m, h, w] = acc;
}
```


Memory Efficiency of Convolution Algorithm

- Assume that we use tiled 2D convolution
- For input elements
 - Each output tile has TILE_WIDTH^2 elements
 - Each input tile has $(\text{TILE_WIDTH}+K-1)^2$
 - The total number of input feature map element accesses was $\text{TILE_WIDTH}^2 * K^2$
 - The reduction factor of the tiled algorithm is $K^2 * \text{TILE_WIDTH}^2 / (\text{TILE_WIDTH}+K-1)^2$
- The convolution filter weight elements are reused within each output tile

Some Observations

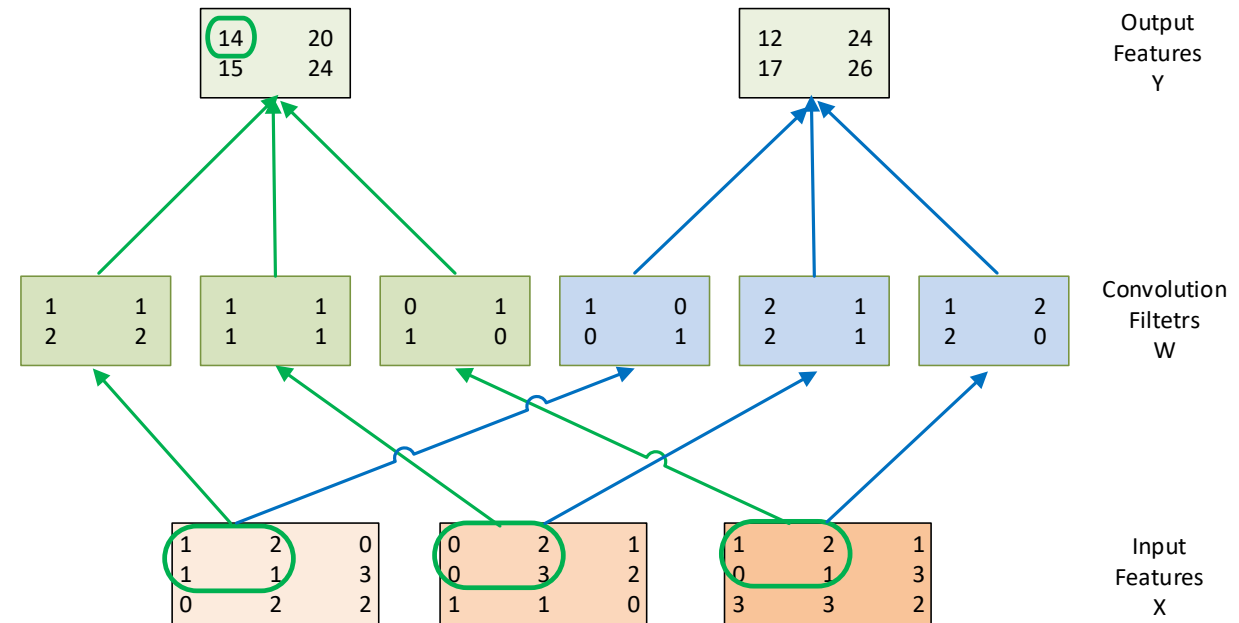
Enough parallelism

- if the total number of pixels across all output feature maps is large
- (often the case for CNN layers)

Memory Bandwidth

- We get reuse for the tiled convolution approach
- but, each tile loaded M times (number of output features), so
- **not efficient in global memory bandwidth,**

Implementing a Convolution Layer with Matrix Multiplication



1	1	2	2
1	0	0	1

1	1	1	1
2	1	2	1

0	1	1	0
1	2	2	0

*

1	2	1	1
2	0	1	3
1	1	0	2
1	3	2	2
0	2	0	3
2	1	3	2
0	3	1	1
3	2	1	0
1	2	1	1
2	1	0	3
0	1	3	3
1	3	3	2

=

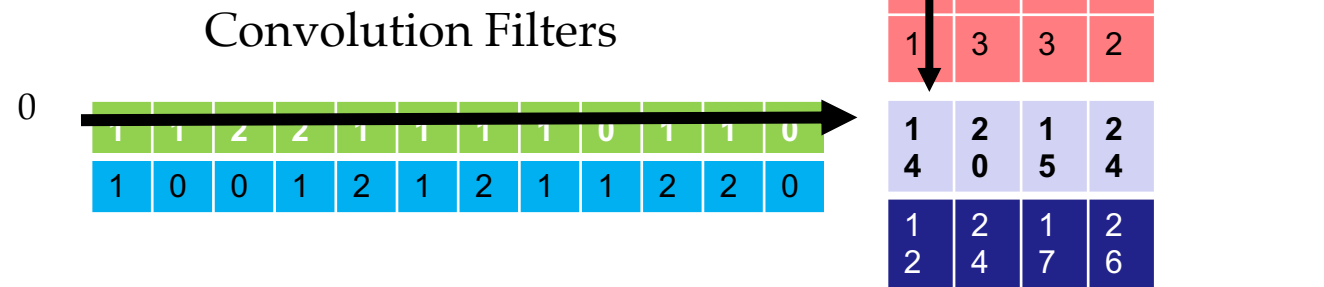
14	20	15	24
12	24	17	26

Convolution Filters W'
Input Features $X_{unrolled}$
Output Features Y

Simple Matrix Multiplication

Each product matrix element is an output feature map pixel.

This inner product generates element 0 of output feature map 0.

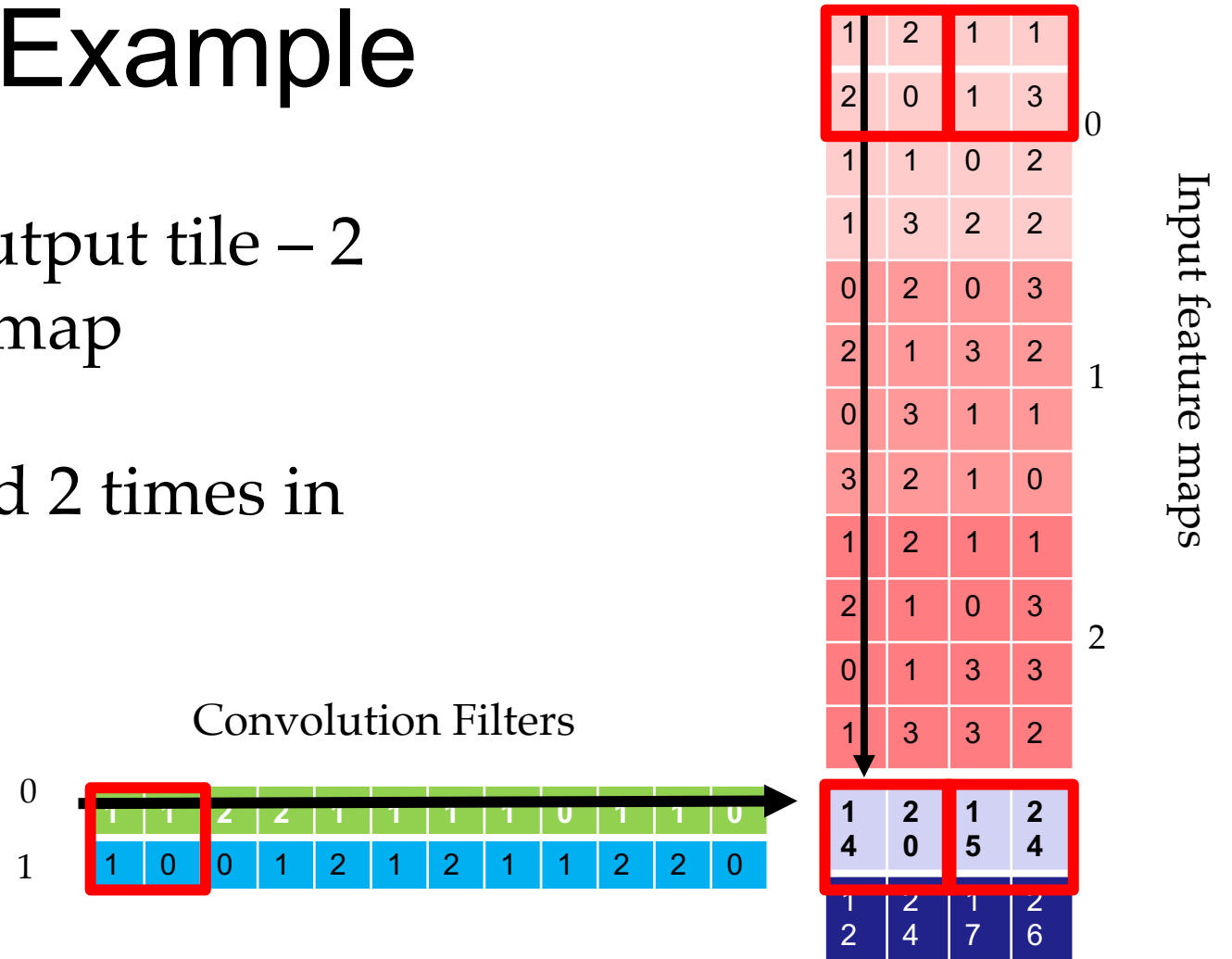


Tiled Matrix Multiplication

2x2 Example

Each block calculates one output tile – 2 elements from each output map

Each input element is reused 2 times in the shared memory

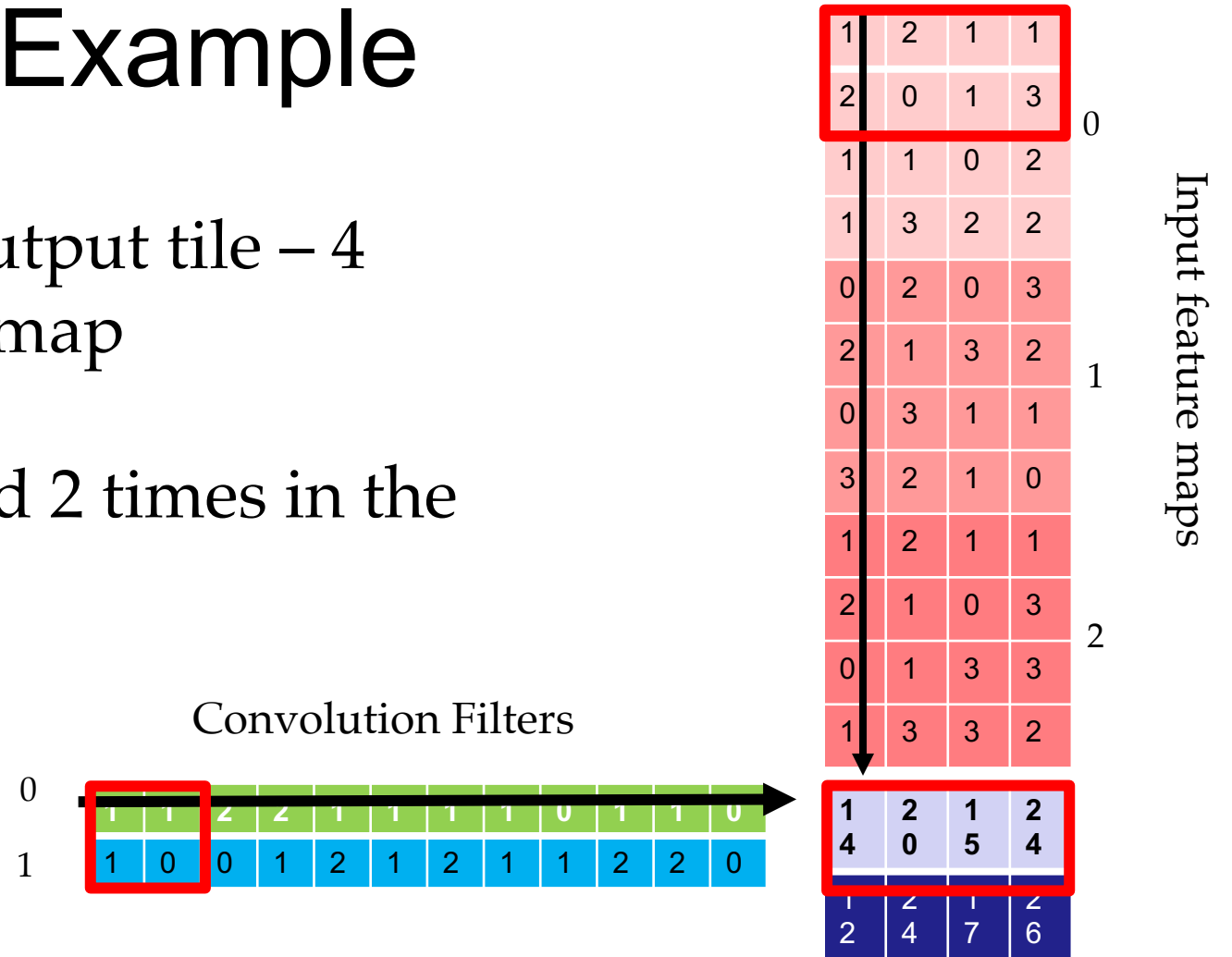


Tiled Matrix Multiplication

2x4 Example

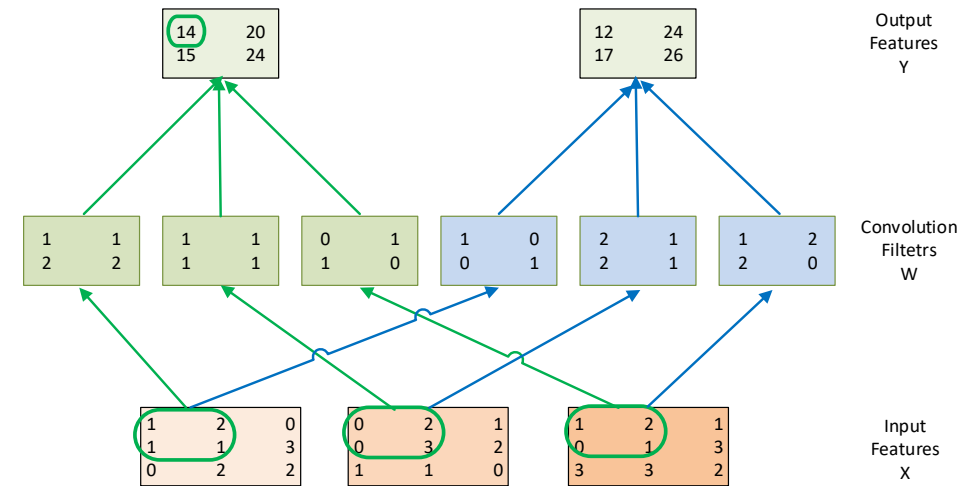
Each block calculates one output tile – 4 elements from each output map

Each input element is reused 2 times in the shared memory



Efficiency Analysis: Total Input Replication

- Replicated input features are shared among output maps
 - There are $H_{out} * W_{out}$ output feature map elements
 - Each requires $K*K$ elements from the input feature maps
 - So, the total number of input element after replication is $H_{out}*W_{out}*K*K$ times for each input feature map
 - The total number of elements in each original input feature map is $(H_{out}+K-1) * (W_{out}+K-1)$



1	1	2	2
1	0	0	1

Convolution
Filters
W'

*

1	2	1	1
2	0	1	3
1	1	0	2
1	3	2	2
0	2	0	3
2	1	3	2
0	3	1	1
3	2	1	0
1	2	1	1
2	1	0	3
0	1	3	3
1	3	3	2

Input
Features
X_unrolled

=

14	20	15	24
12	24	17	26

Output
Features
Y

Analysis of a Small Example

$$H_{\text{out}} = 2$$

$$W_{\text{out}} = 2$$

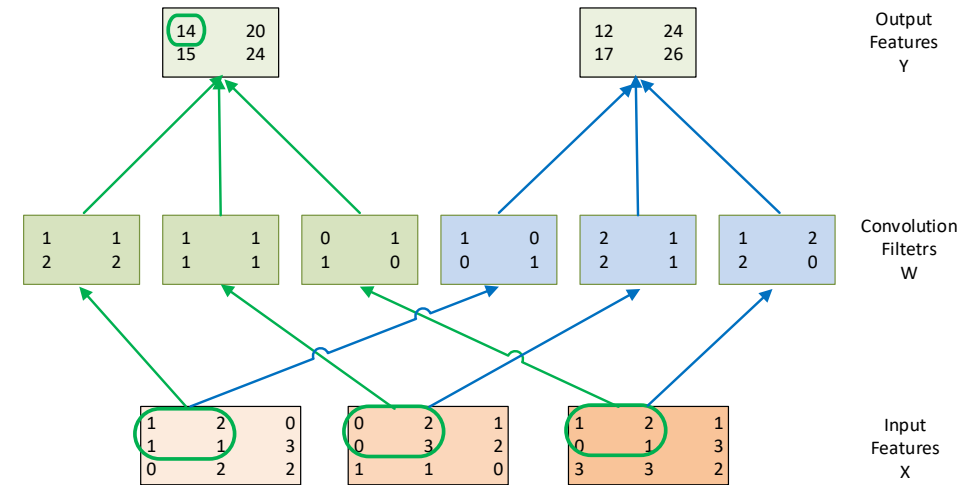
$$K = 2$$

There are 3 input maps (channels)

The total number of input elements in the replicated (“unrolled”) input matrix is $3 \times 2 \times 2 \times 2 \times 2$

The replicating factor is

$$(3 \times 2 \times 2 \times 2 \times 2) / (3 \times 3 \times 3) = 1.78$$



1	1	2	2
1	0	0	1

1	1	1	1
2	1	2	1

0	1	1	0
1	2	2	0

*

1	2	1	1
2	0	1	3
1	1	0	2
1	3	2	2
0	2	0	3
2	1	3	2
0	3	1	1
3	2	1	0
1	2	1	1
2	1	0	3
0	1	3	3
1	3	3	2

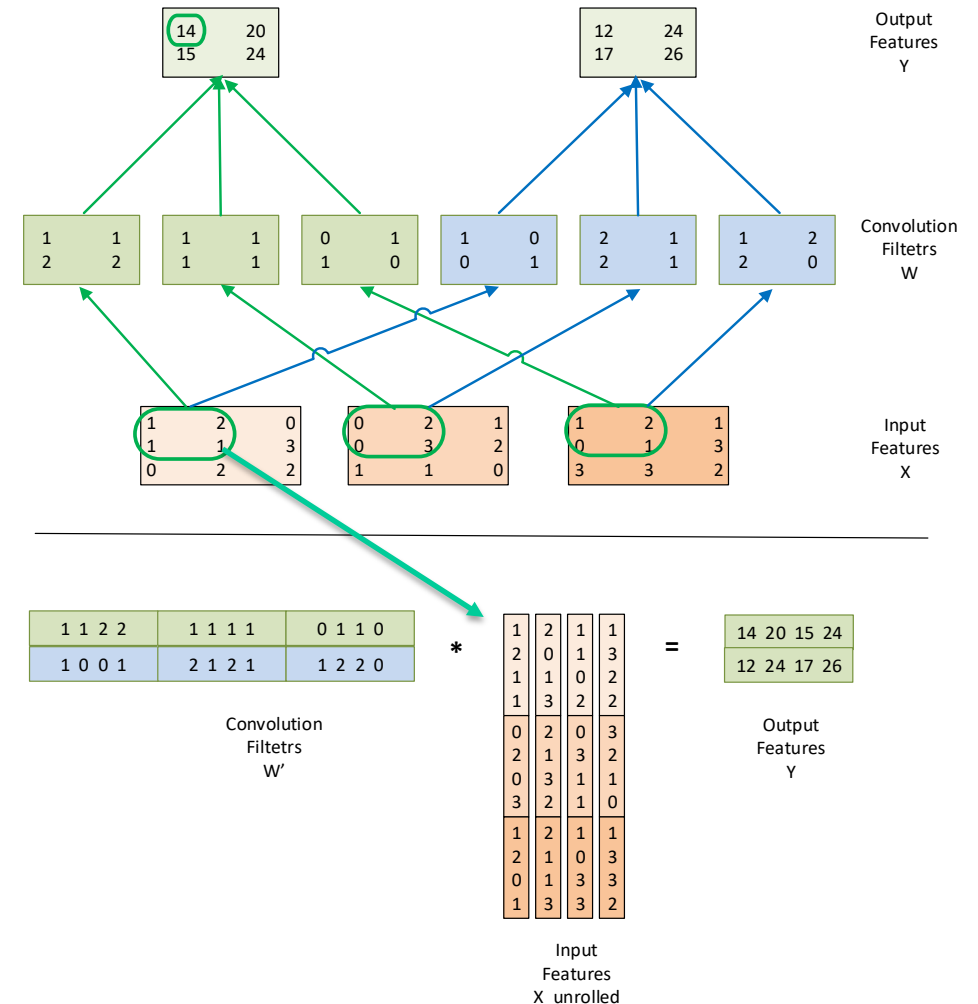
=

14	20	15	24
12	24	17	26

Convolution Filters W' Input Features X_{unrolled} Output Features Y

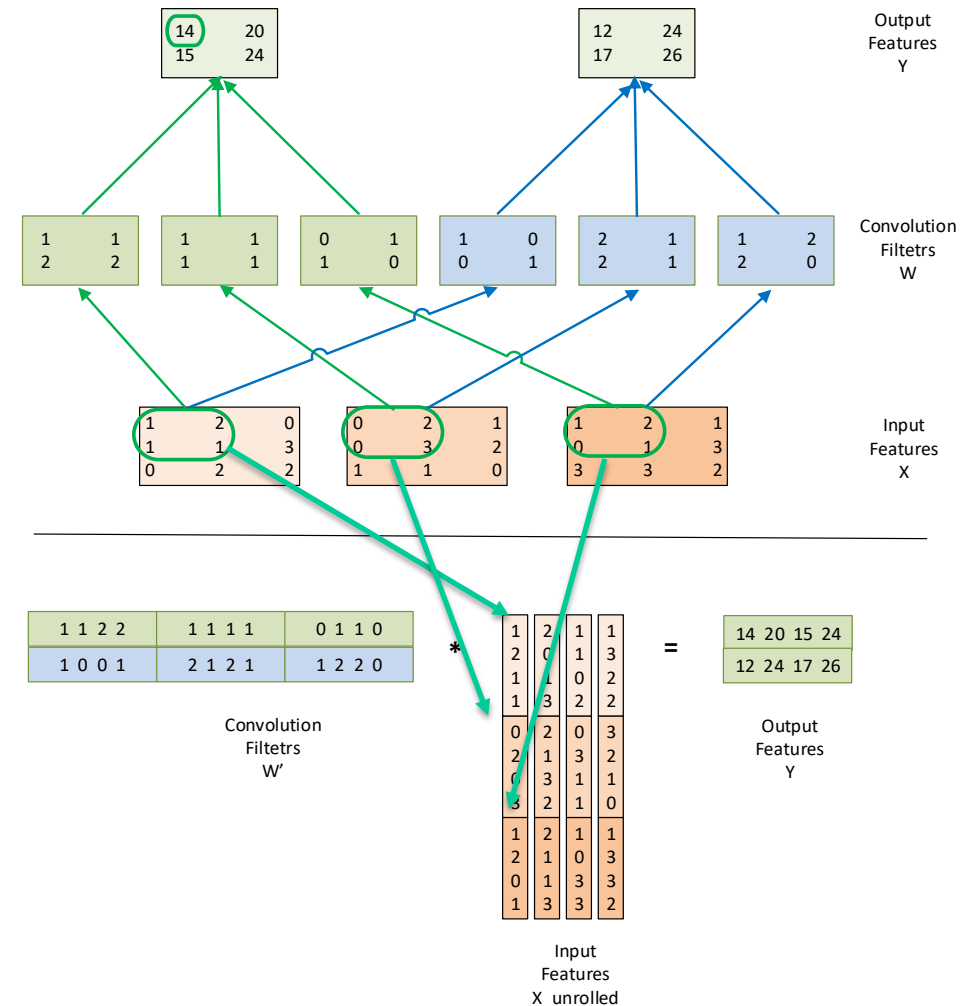
Properties of the Unrolled Matrix

- Each unrolled column corresponds to an output feature map element
- For an output feature element (h,w), the index for the unrolled column is $h * W_{out} + w$ (linearized index of the output feature map element)



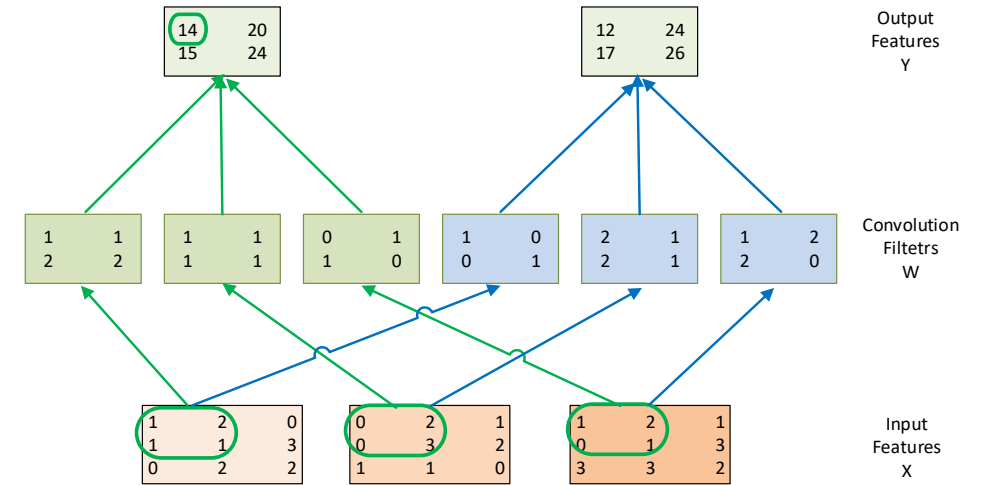
Properties of the Unrolled Matrix (cont.)

- Each section of the unrolled column corresponds to an input feature map
- Each section of the unrolled column has $k \times k$ elements (convolution mask size)
- For an input feature map c , the vertical index of its section in the unrolled column is $c \times k \times k$ (linearized index of the output feature map element)



To Find the Input Elements

- For output element (h,w) , the base index for the upper left corner of the input feature map c is (c, h, w)
- The input element index for multiplication with the convolution mask element (p, q) is $(c, h+p, w+q)$



1	1	2	2
1	0	0	1

1	1	1	1
2	1	2	1

0	1	1	0
1	2	2	0

*

1	2	1	1
2	0	1	3
1	1	0	2
1	3	2	2
0	2	0	3
2	1	3	2
0	3	1	1
3	2	1	0
1	2	1	1
2	1	0	3
0	1	3	3
1	3	3	2

=

14	20	15	24
12	24	17	26

Convolution Filters W' Input Features $X_{unrolled}$ Output Features Y

Input to Unrolled Matrix Mapping

Output element (h, w)

Mask element (p, q)

Input feature map c

```
// calculate the horizontal matrix index
```

```
int w_unroll = h * W_out + w;
```

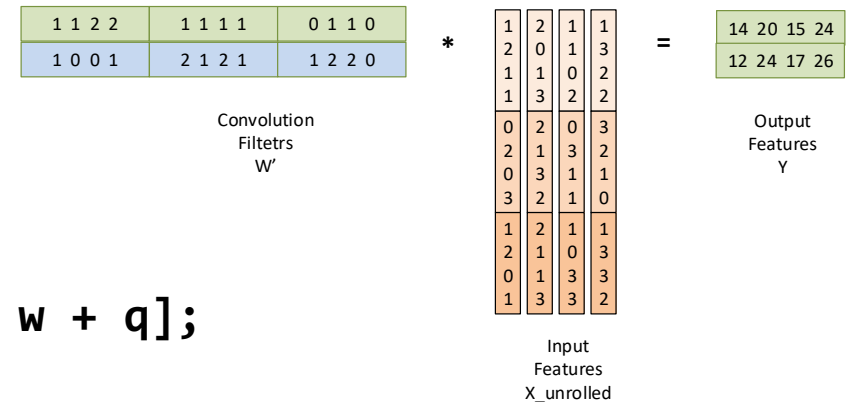
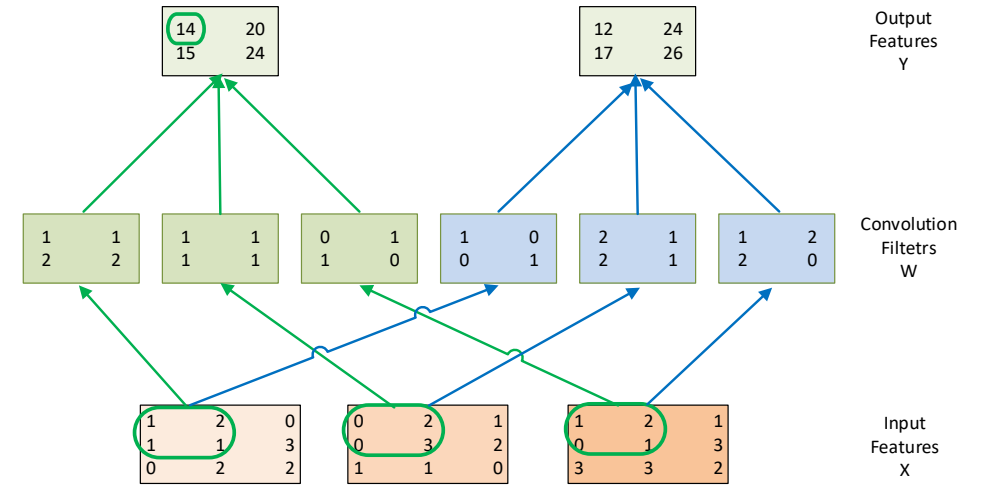
```
// find the beginning of the unrolled
```

```
int w_base = c * (K*K);
```

```
// calculate the vertical matrix index
```

```
int h_unroll = w_base + p * K + q;
```

```
X_unroll[b, h_unroll, w_unroll] = X[b, c, h + p, w + q];
```



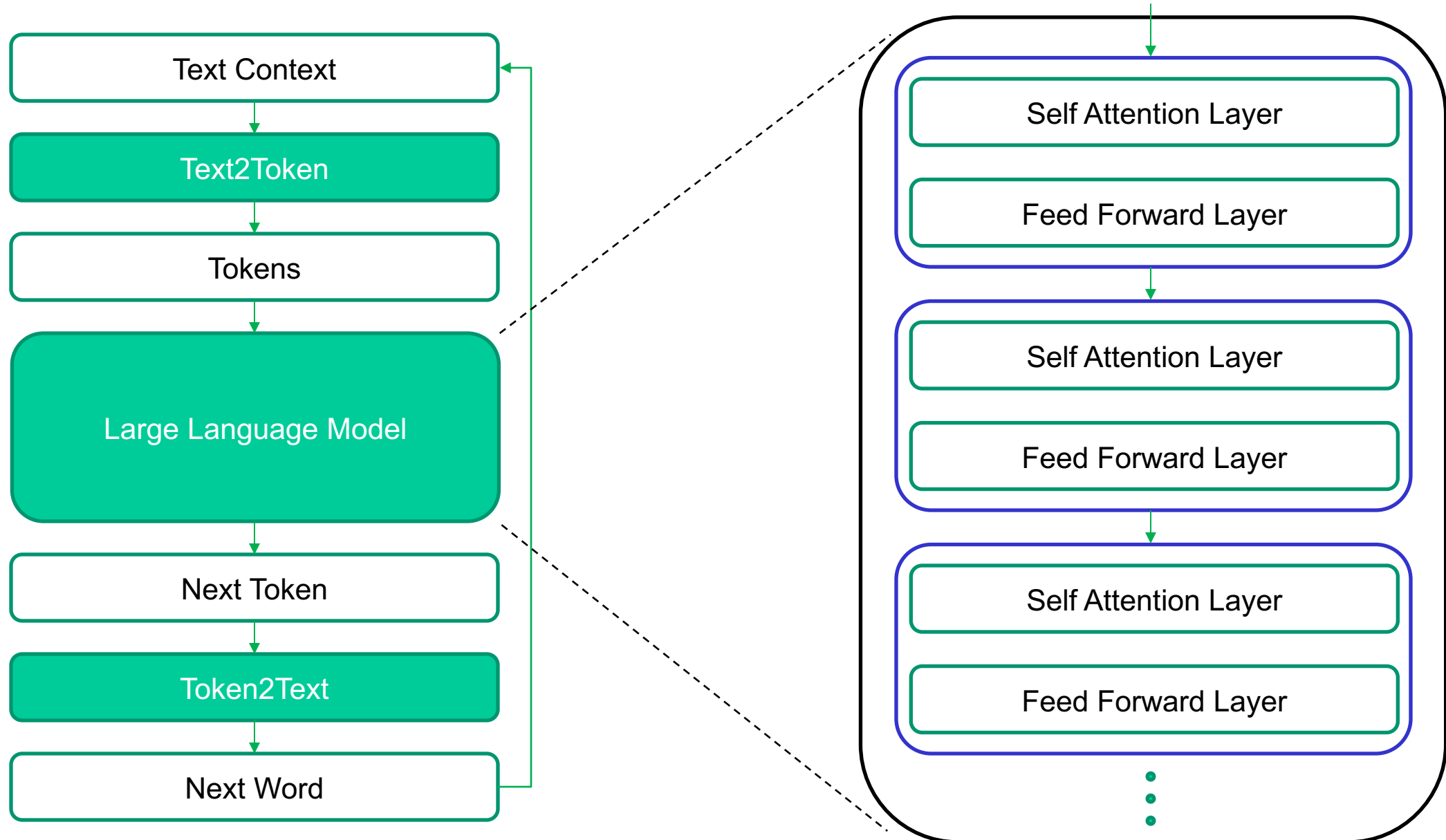
Function to generate “unrolled” X

```
void unroll(int B, int C, int H, int W, int K, float* X, float* X_unroll)
{
    int H_out = H - K + 1;                // calculate H_out, W_out
    int W_out = W - K + 1;
    for (int b = 0; b < B; ++b)           // for each image
        for (int c = 0; c < C; ++c) {     // for each input channel
            int w_base = c * (K*K);        // per-channel offset for smallest X_unroll index
            for (int p = 0; p < K; ++p)    // for each element of KxK filter (two loops)
                for (int q = 0; q < K; ++q) {
                    for (int h = 0; h < H_out; ++h) // for each thread (each output value, two loops)
                        for (int w = 0; w < W_out; ++w) {
                            int h_unroll = w_base + p * K + q; // data needed by one thread
                            int w_unroll = h * W_out + w;        // smallest index--across threads (output values)
                            X_unroll[b, h_unroll, w_unroll] = X[b, c, h + p, w + q]; // copy input pixels
                        }
                }
        }
}
```

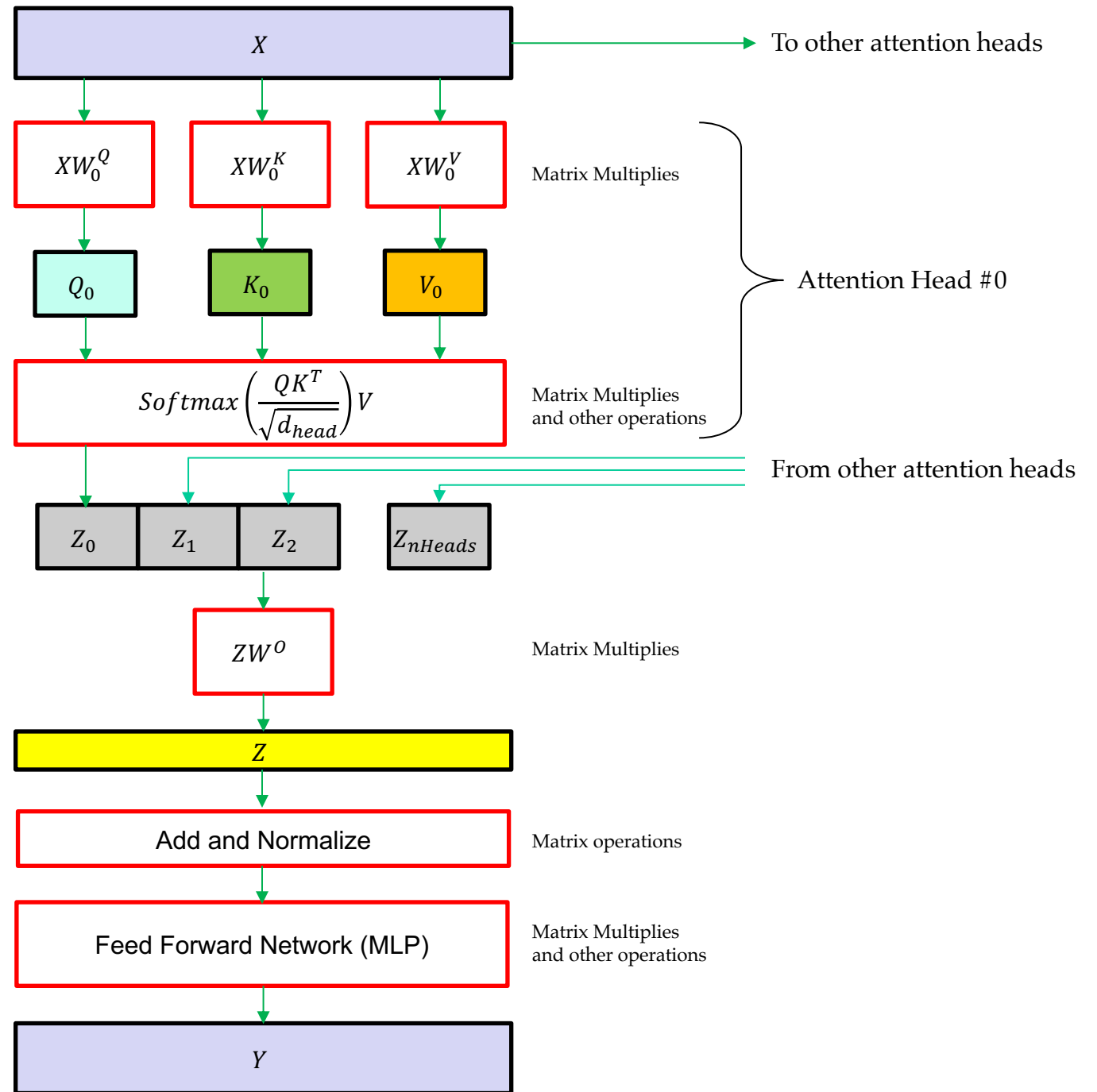
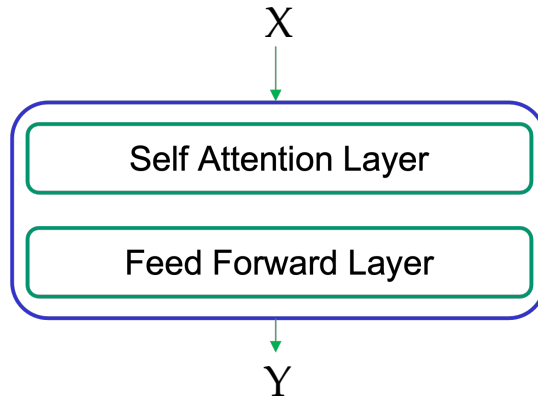
Implementation Strategies for a Convolution Layer

- **Baseline**
 - Tiled 2D convolution implementation, use constant memory for convolution masks
- **Matrix-Multiplication Baseline**
 - Input feature map unrolling kernel, constant memory for convolution masks as an optimization
 - Tiled matrix multiplication kernel
- **Matrix-Multiplication with built-in unrolling**
 - Perform unrolling only when loading a tile for matrix multiplication
 - The unrolled matrix is only conceptual
 - When loading a tile element of the conceptual unrolled matrix into the shared memory, use the properties in the lecture to load from the input feature map
- **More advanced Matrix-Multiplication**
 - Use joint register-shared memory tiling

Transformer-based Language Models



Single Layer Computational Flow



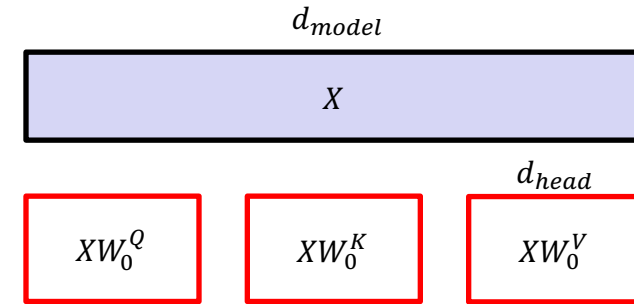
GPT-3, as an example

Language Models are Few-Shot Learners, Brown et al., OpenAI, July 2020

Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	350M	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	760M	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	1.3B	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	2.7B	32	2560	32	80	1M	1.6×10^{-4}
GPT-3 6.7B	6.7B	32	4096	32	128	2M	1.2×10^{-4}
GPT-3 13B	13.0B	40	5140	40	128	2M	1.0×10^{-4}
GPT-3 175B or "GPT-3"	175.0B	96	12288	96	128	3.2M	0.6×10^{-4}

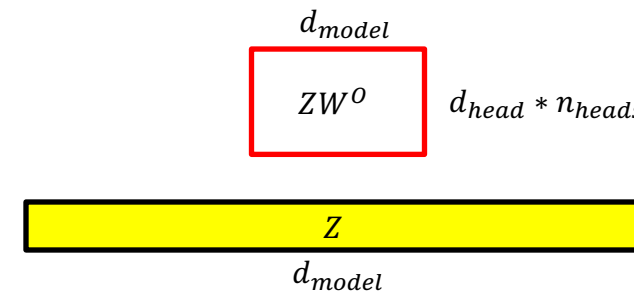
Table 2.1: Sizes, architectures, and learning hyper-parameters (batch size in tokens and learning rate) of the models which we trained. All models were trained for a total of 300 billion tokens.

GPT-3 has 96 Layers, 55B parameters just from Self Attention. 220 Gflop per X vector, per output token.



In GPT-3, each W is 12288x128 matrix, 1.5M parameters, 3MB (@ 16 bit floats), or 6 MFlop per X vector

There are 96x3 of these for a total of 432M parameters, 864MB (@ 16 bit floats), or 1.7 GFlop per X vector



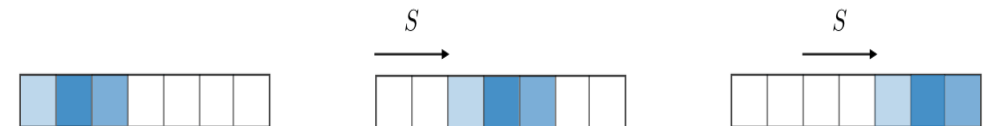
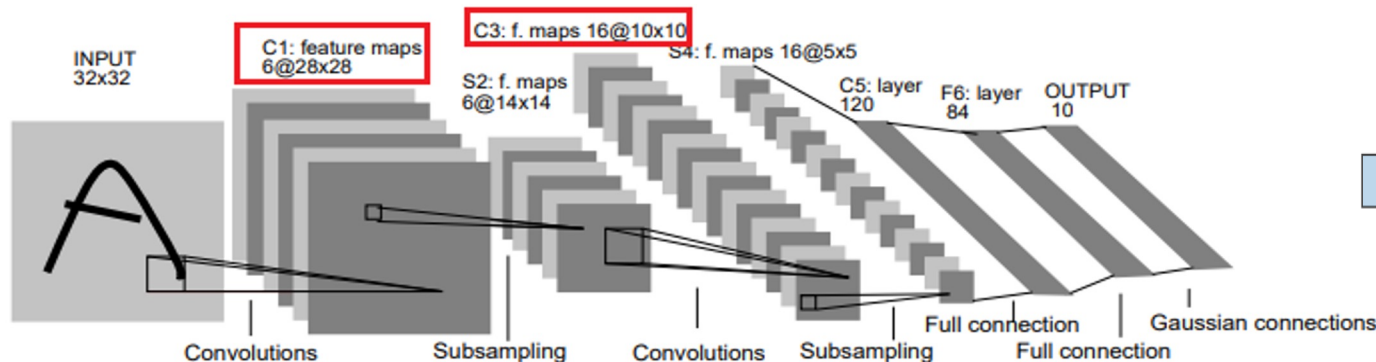
W^O is 12288x12288 matrix, 150M parameters, 300MB (@ 16 bit floats), or 600 MFlop per X vector

Project Overview

- Optimize the forward pass of the convolutional layers in a modified LeNet-5 CNN using CUDA. (CNN implemented using Mini-DNN, a C++ framework, **with Stride functionality**)
- The network will be classifying Fashion MNIST dataset
- Some network parameters to be aware of
 - Input Size: 86x86 pixels, batch of 10k images
 - Input Channels: 1
 - Convolutional kernel size: 7x7
 - Number of kernels: Variable (your code should support this)
 - Stride length: Variable (your code should support this)



<https://github.com/zalandoresearch/fashion-mnist>



<https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks#layer>

Project Timeline

- **All milestones are due on Fridays at 8 pm Central Time**
- Everyone must individually submit all milestones.
 - **No sharing of code is allowed**
- Project milestone 1:
 - CPU Convolution with stride, CPU code profiling
- Project milestone 2:
 - Baseline GPU Convolution Kernel with stride
- Project milestone 3:
 - GPU Convolution Kernel Optimizations

Project Release

- Project will be released soon (only PM1 for now)
 - Check the course wiki page for the link to the github repository
 - <https://github.com/aschuh703/ECE408/tree/main/Project>
- The Readme in the repository contains all the instructions and details to complete the project.
- The github repo will be updated with additional code and instructions for PM2 & PM3