ECE408 / CS483 / CSE408
Summer 2024

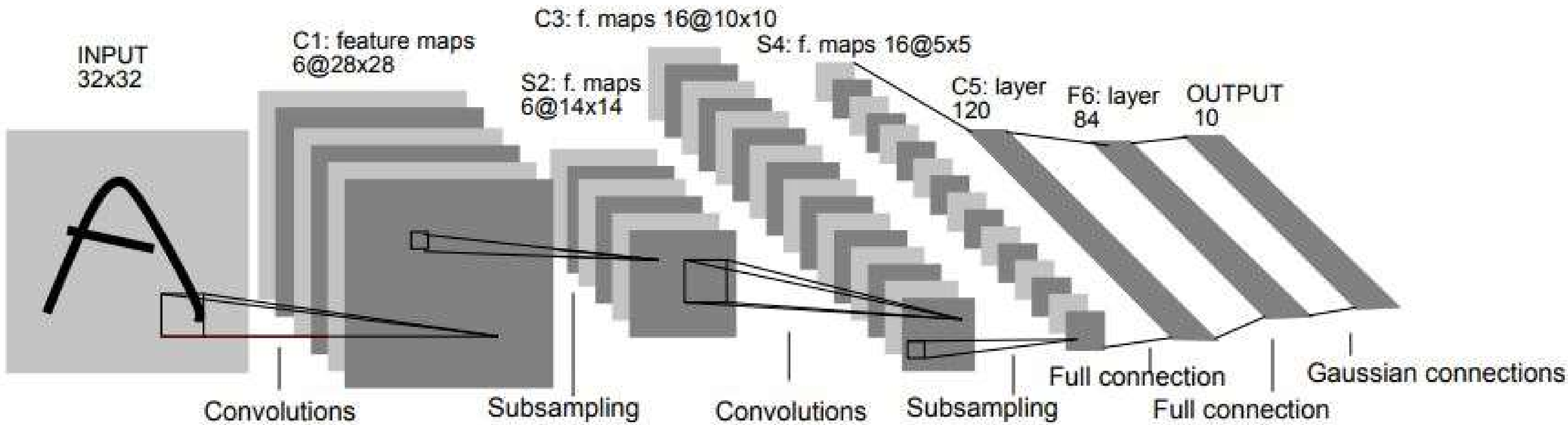
Applied Parallel Programming

Lecture 18: Computation in Deep Neural Networks

What Will You Learn Today?

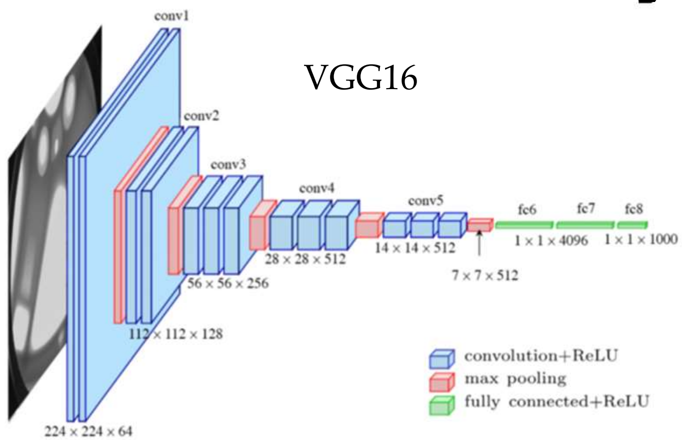
to implement the different types of layers
in a Convolutional Neural Network (CNN)

LeNet-5:CNN for hand-written digit recognition

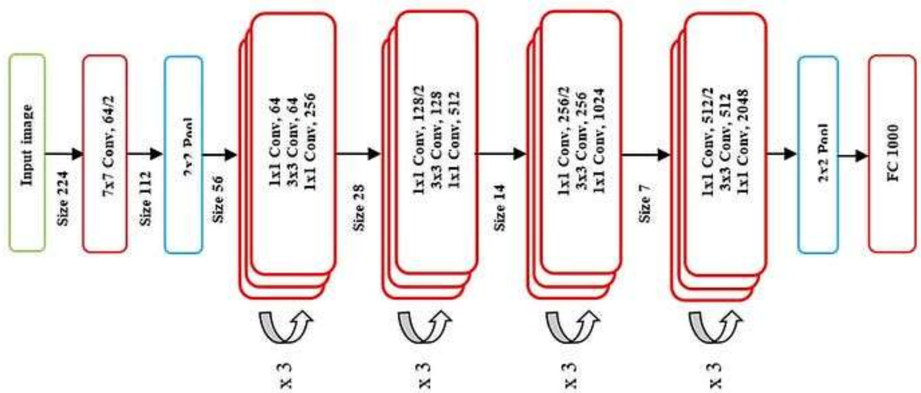


Many Types of CNNs

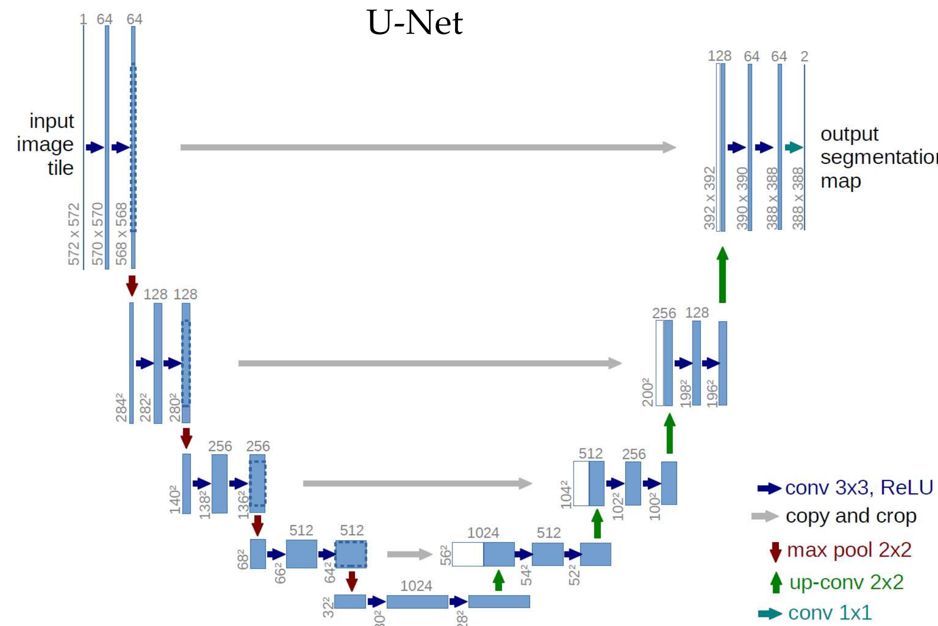
VGG16



ResNet50



U-Net

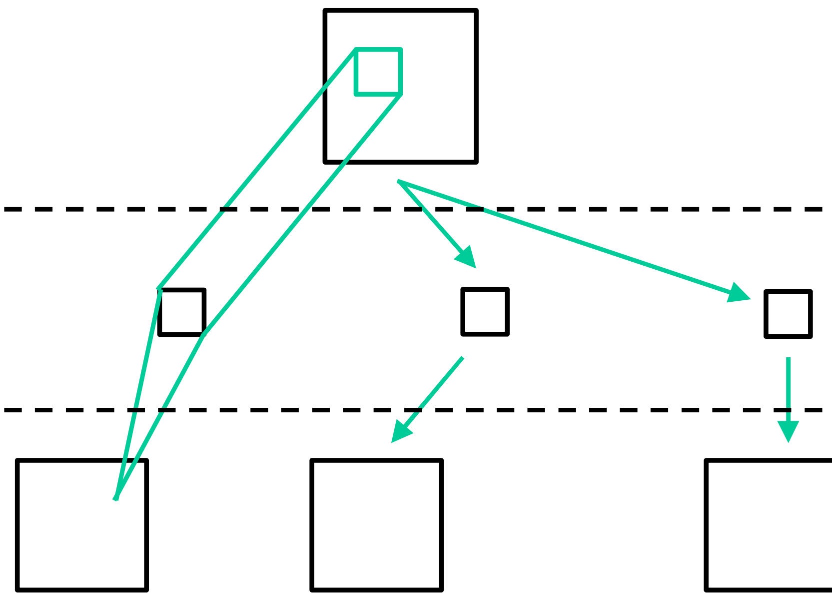


Anatomy of a Convolution Layer

Input features
- A inputs each $N_1 \times N_2$

Convolution Layer
- B convolution kernels each $K_1 \times K_2$

Output Features (total of B)
- $A \times B$ outputs each $(N_1 - K_1+1) \times (N_2 - K_2+1)$

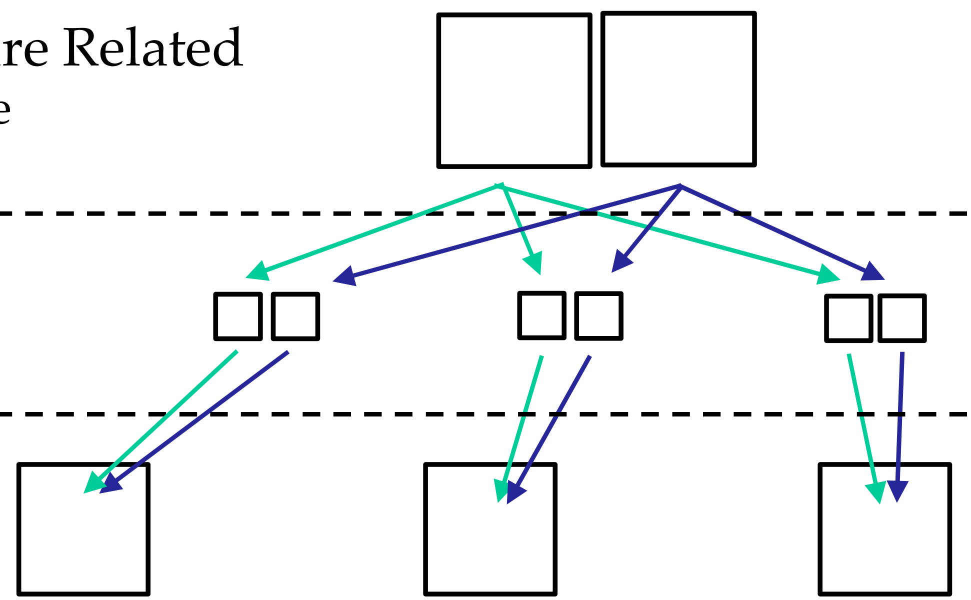


Notion of a Channel in Input Layer

Some Set of Input Features are Related
- For example: Red, Green, Blue

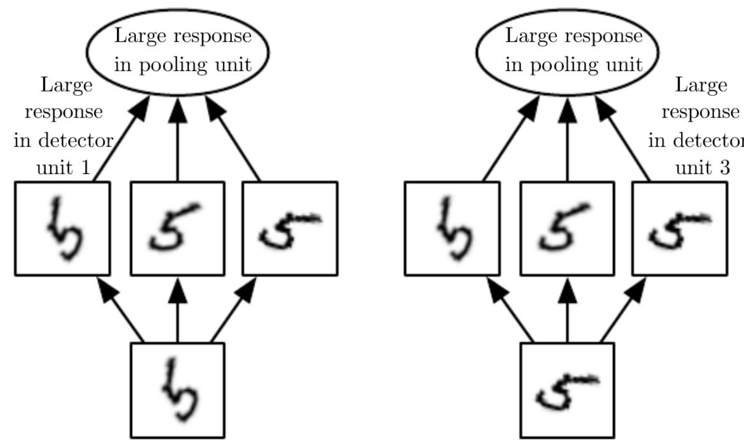
Convolution Layer
- Different kernels per channel

Output Features
- Channels combine per output feature



2-D Pooling (Subsampling)

- A 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

Output Size

$H_{out} = H - K + 1$

$W_{out} = W - K + 1$

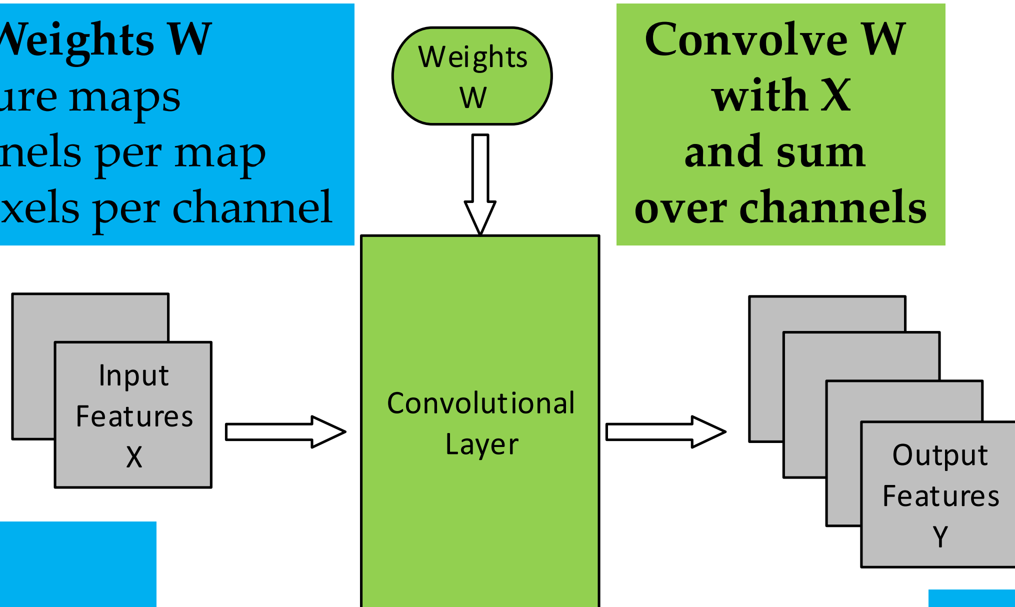
Weights W

M feature maps

C channels per map

K×K pixels per channel

Convolve W with X and sum over channels



Input X

B images

C channels per image

H×W pixels per channel

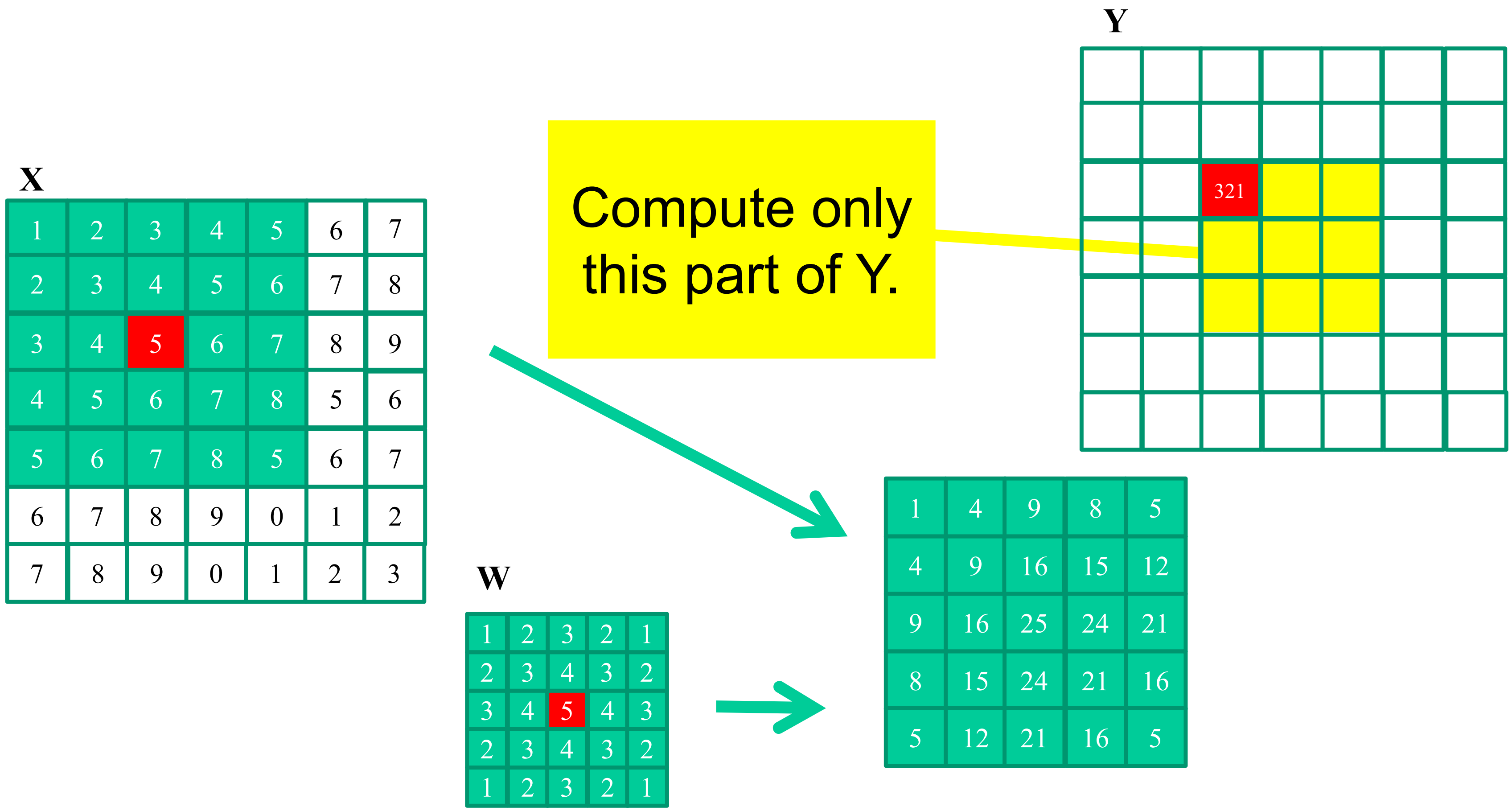
Convolution Output Y

B images

M features per image

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

Outputs Must Use Full Mask/Kernel

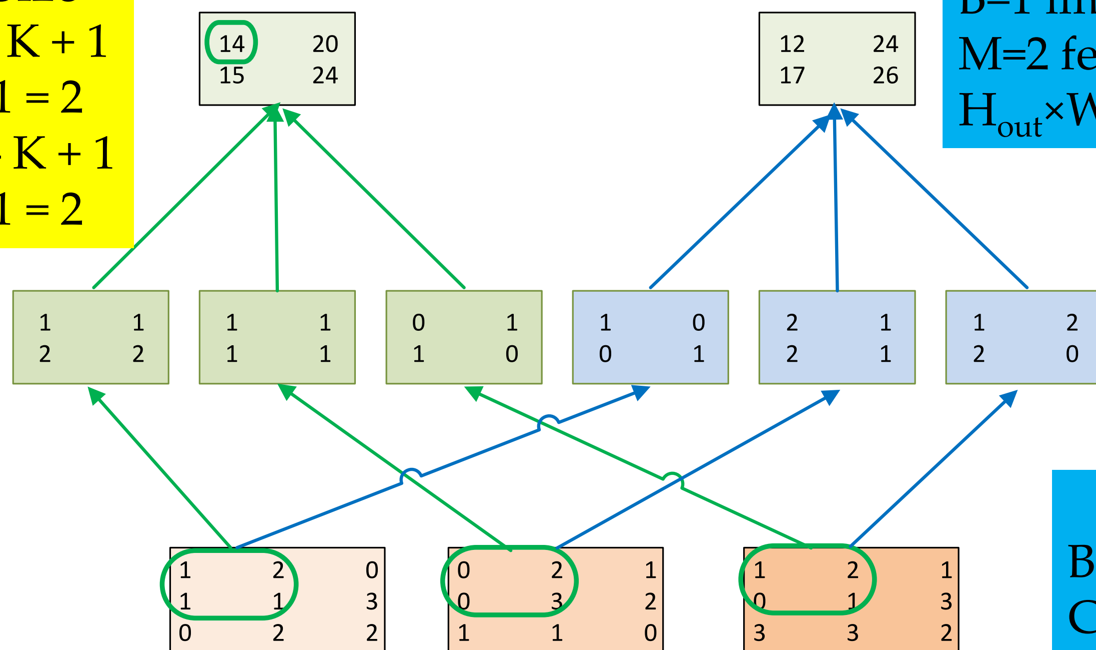


Example of the Forward Path of a Convolution Layer

Output Size

$H_{out} = H - K + 1$
$= 3 - 2 + 1 = 2$

$W_{out} = W - K + 1$
$= 3 - 2 + 1 = 2$



Convolution Output Y

B=1 image

M=2 features per image

$H_{out} \times W_{out} = 2 \times 2$ values per feature

Weights W

M=2 feature maps

C=3 channels per map

K×K=2×2 pixels per channel

Input X

B=1 image

C=3 channels

H×W=3×3 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

input channel

X[b, 1,_,_]

W[0,1,_, _]

Y[b, 0,_, _]

output map

X[b,0,_, _]

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

W[0,0,_, _]

1	1	1
2	2	3
2	1	0

X[b,1,_, _]

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

W[0,1,_, _]

1	2	3
1	1	0
3	0	1

Y[b,0,_, _]

0	?
?	?

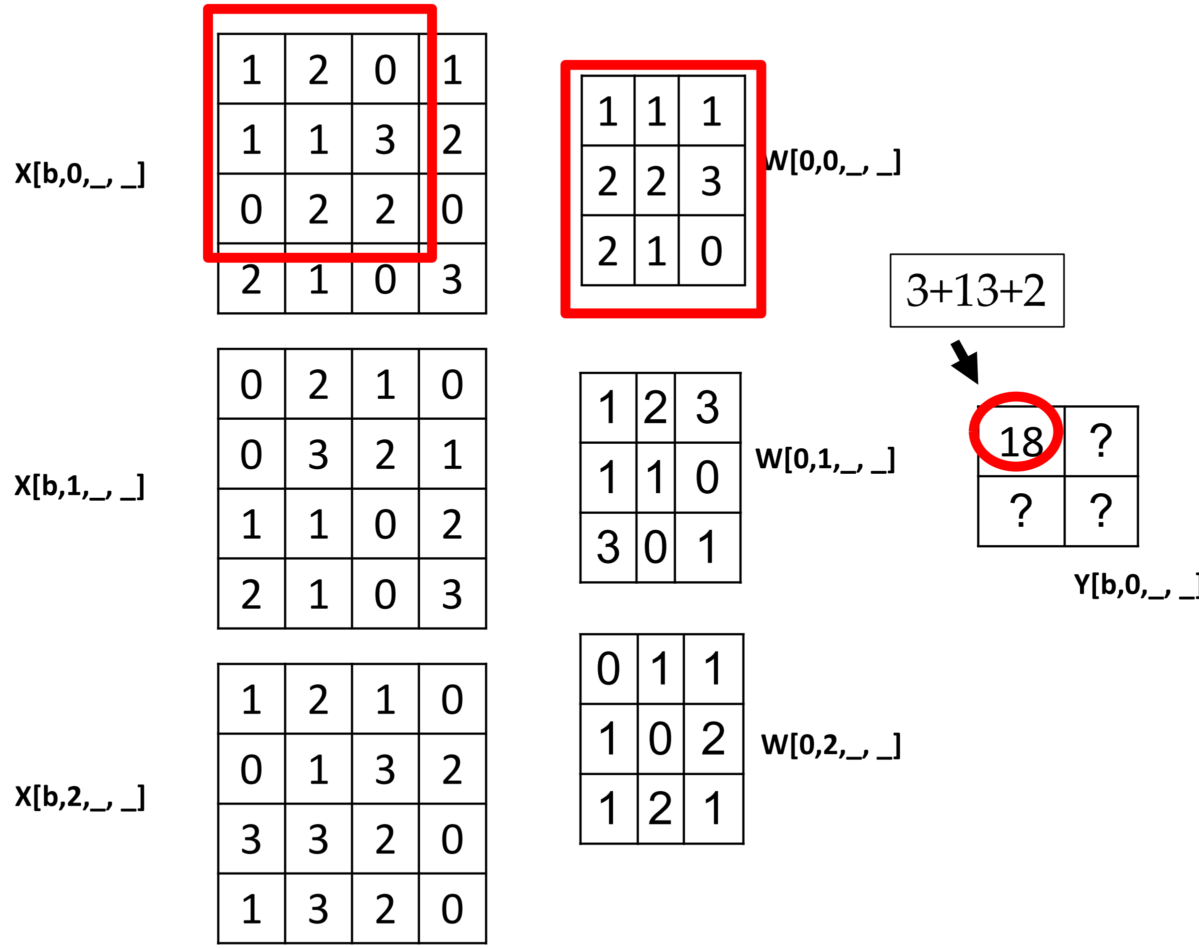
X[b,2,_, _]

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

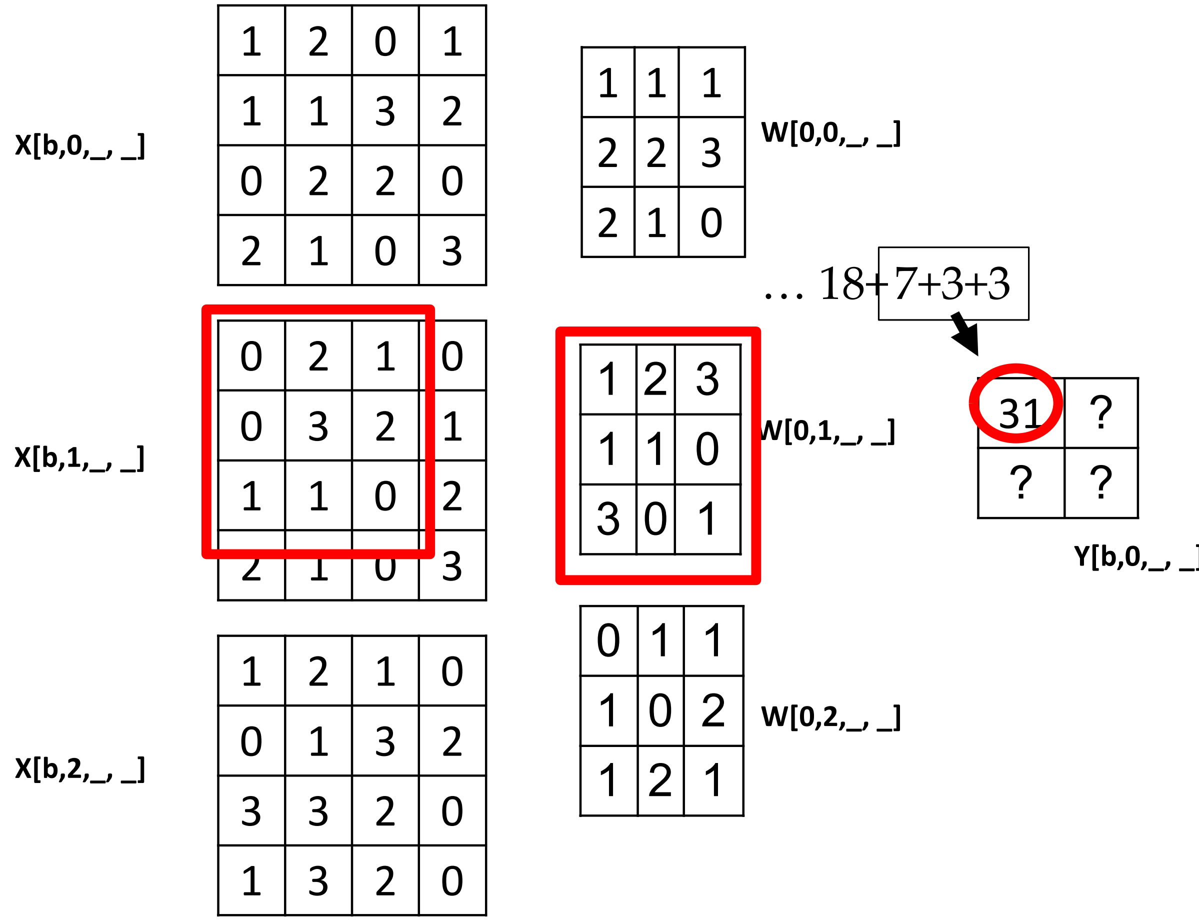
W[0,2,_, _]

0	1	1
1	0	2
1	2	1

A Small Convolution Layer Example c = 0



A Small Convolution Layer Example c = 1



A Small Convolution Layer Example c = 2

X[b,0,_, _]

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

W[0,0,_, _]

1	1	1
2	2	3
2	1	0

X[b,1,_, _]

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

W[0,1,_, _]

1	2	3
1	1	0
3	0	1

X[b,2,_, _]

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

W[0,2,_, _]

0	1	1
1	0	2
1	2	1

…

31+

3+6+11

Y[b,0,_, _]

51	?
?	?

Parallelism in a Convolution Layer

Output feature maps can be calculated in parallel

- Usually a small number, not sufficient to fully utilize a GPU

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

All **input feature maps** can be processed in parallel,
but need atomic operation or tree reduction (we'll learn later)

Different layers may demand different strategies.

Subsampling (Pooling) by Scale N

Convolution Output Y
B images
M features per image
$H_{out} \times W_{out}$ values per feature

Average over N×N blocks,
then calculate sigmoid

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

Subsampling/Pooling Output S
B images
M features per image
$H_{S(N)} \times W_{S(N)}$ values per feature

Sequential Code: Forward Pooling Layer

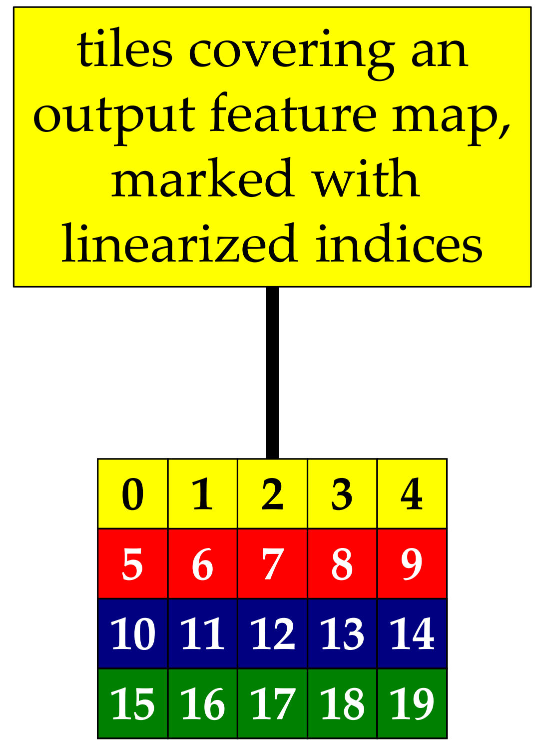
```
void poolingLayer_forward(int B, int M, int H_out, int W_out, int N, float* Y, float* S)
{
  for (int b = 0; b < B; ++b)                    // for each image
    for (int m = 0;  m < M; ++m)                  // for each output feature map
      for (int x = 0; x < H_out/N; ++x)            // for 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 the output feature maps (linearized order).
- (Grid's Z dimension is used for images in batch, which we omit from slides.)



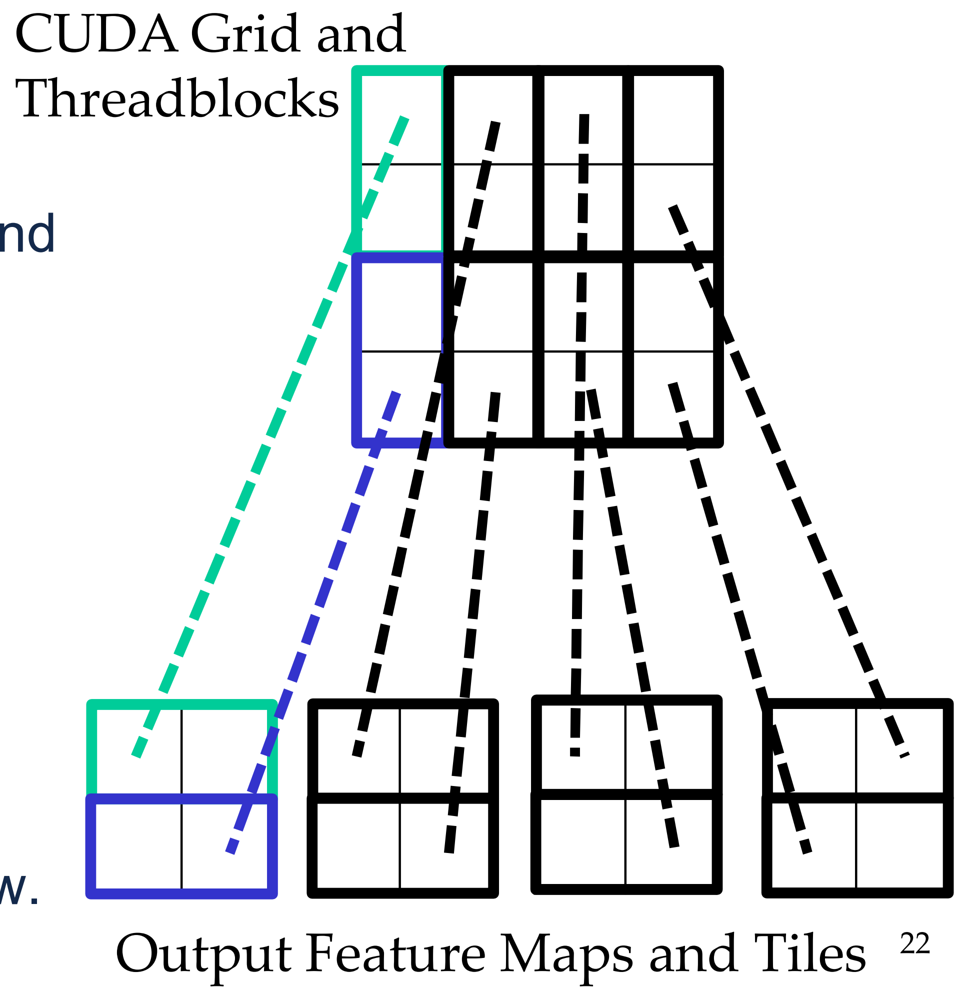
A Small Example

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.



Host Code for a Basic Kernel: CUDA Grid

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

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

Host Code for a Basic Kernel: CUDA Grid

(Assuming W_out and H_out are multiples of TILE_WIDTH.)

```
#define TILE_WIDTH 16        // We will use 4 for small examples.
W_grid = W_out/TILE_WIDTH;   // number of horizontal tiles per output map
H_grid = H_out/TILE_WIDTH;   // number of vertical tiles per output map
Y = H_grid * W_grid;

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

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

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];
        }
}
```

Computed by the grid

Computed by a thread

Partial Kernel Code for a Convolution Layer

```
__global__ void ConvLayerForward_Basic_Kernel
    (int C, int W_grid, int K, float* X, float* W, float* Y)
{
    int m = blockIdx.x;
    int h = (blockIdx.y / W_grid) * TILE_WIDTH + threadIdx.y;
    int w = (blockIdx.y % W_grid) * 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;
}
```

Some Observations

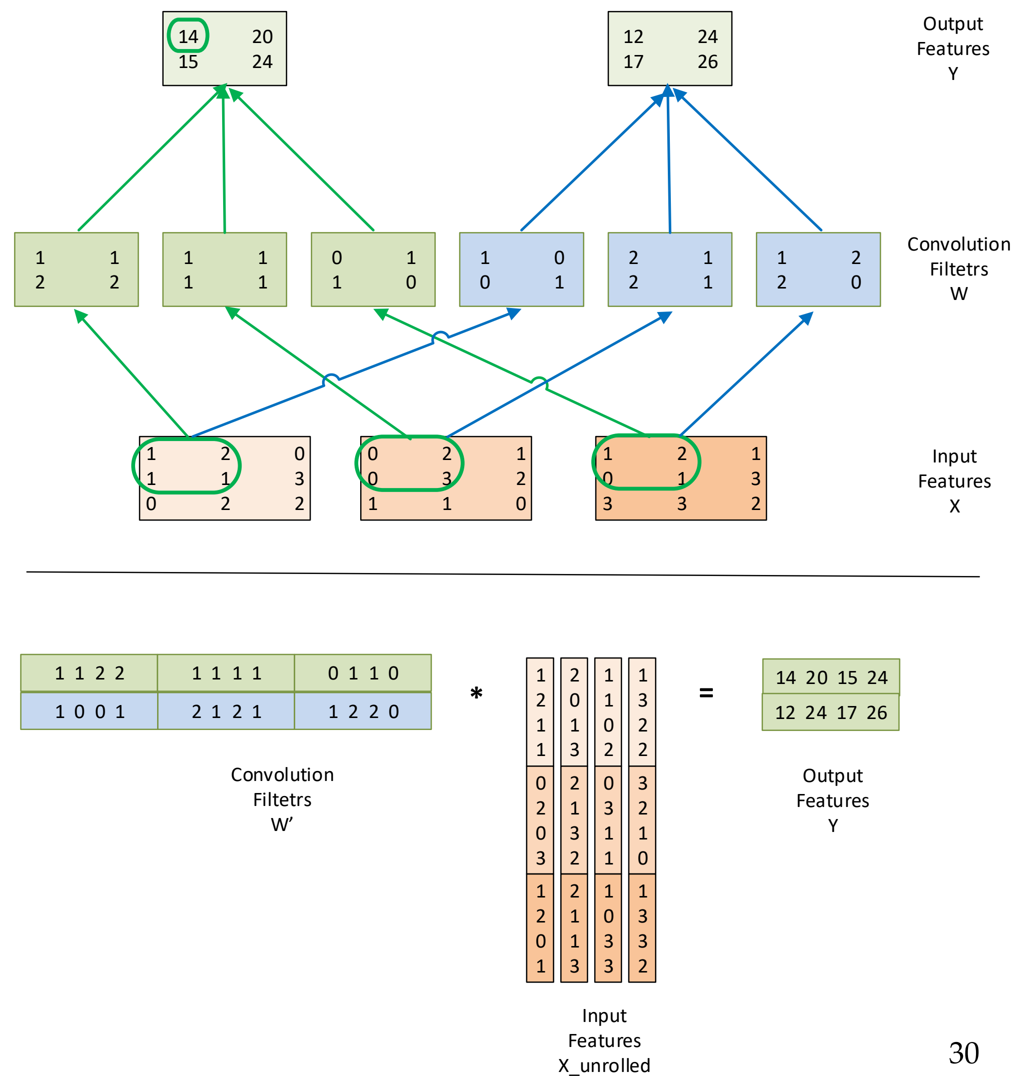
Enough parallelism

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

Each input tile

- loaded M times (number of output features), so
- **not efficient in global memory bandwidth,**
- but block scheduling in X dimension should give cache benefits.

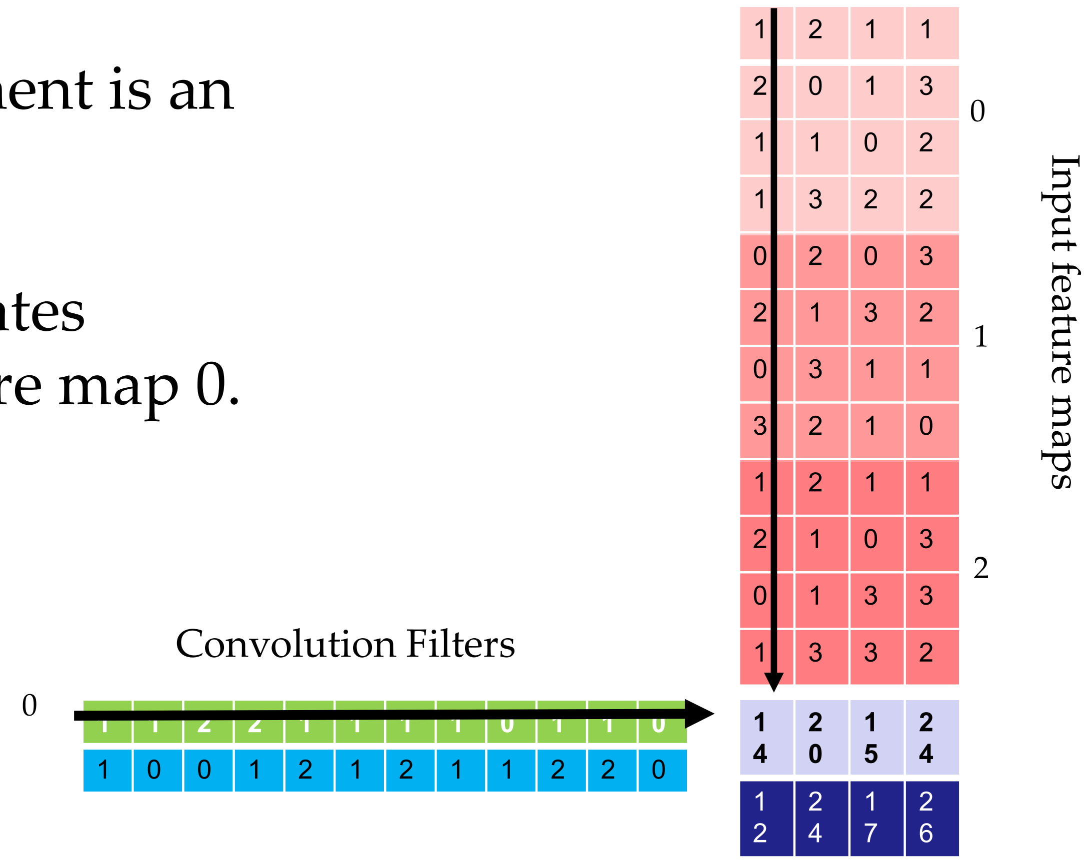
Implementing a Convolution Layer with Matrix Multiplication



Simple Matrix Multiplication

Each product matrix element is an output feature map pixel.

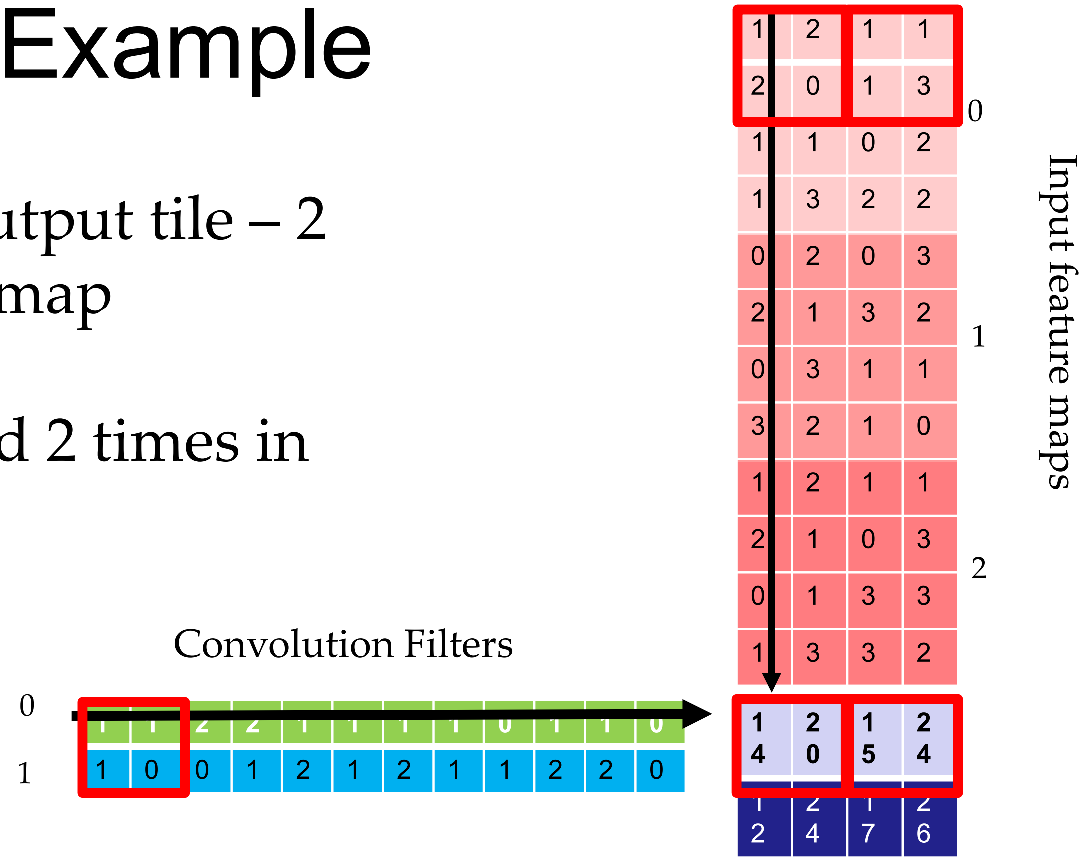
This inner product generates element 0 of output feature map 0.



Tiled Matrix Multiplication
2x2 Example

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

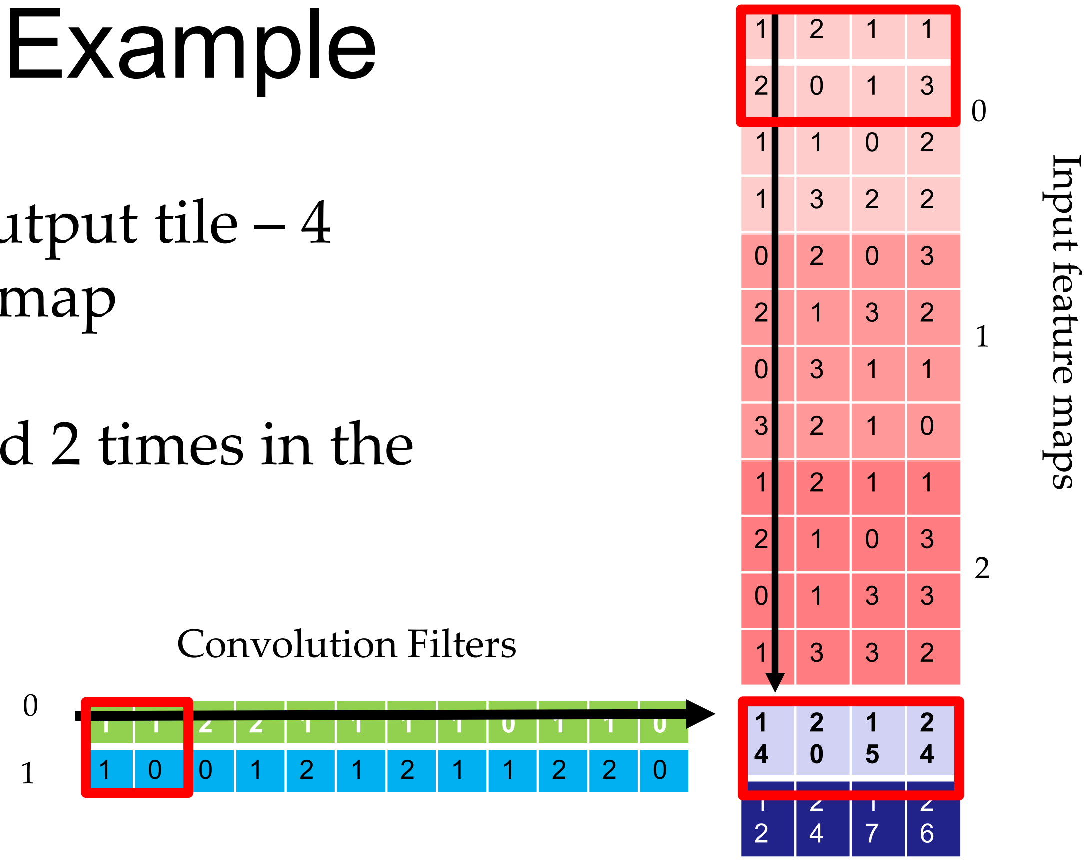
Each input element is reused 2 times in the shared memory



Tiled Matrix Multiplication
2x4 Example

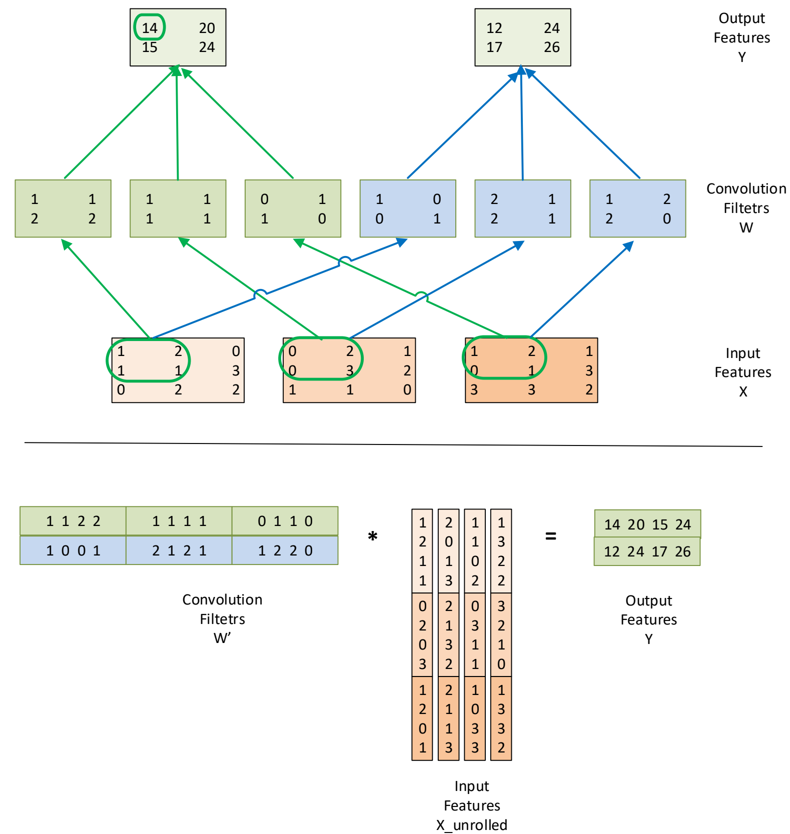
Each block calculates one output tile – 4 elements from each output map

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)



Analysis of a Small Example

H_out = 2

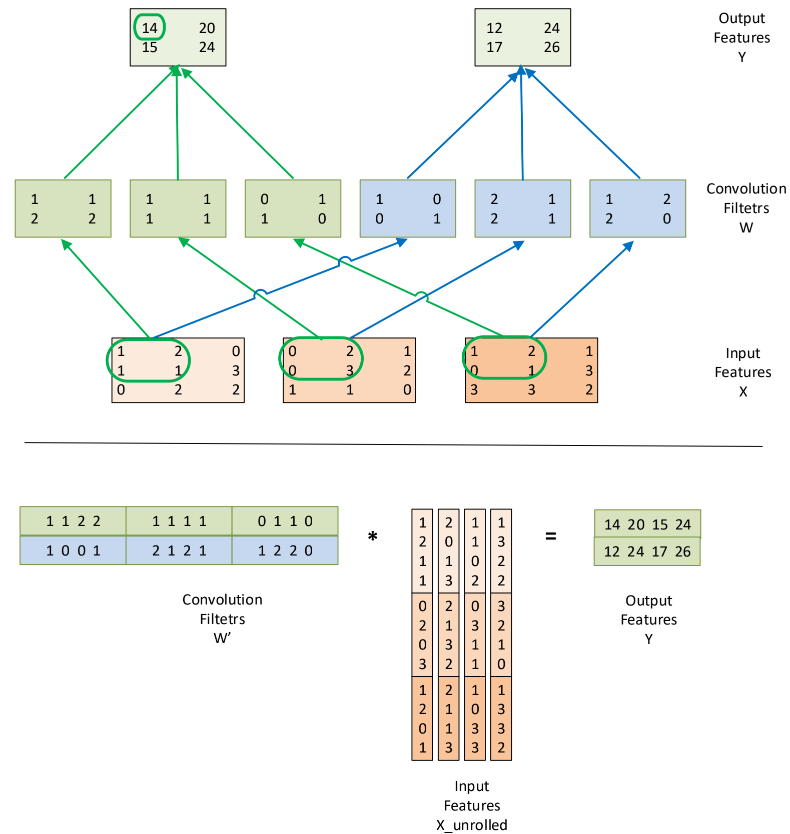
W_out = 2

K = 2

There are 3 input maps (channels)

The total number of input elements in the replicated ("unrolled") input matrix is 3*2*2*2*2

The replicating factor is (3*2*2*2*2)/(3*3*3) = 1.78

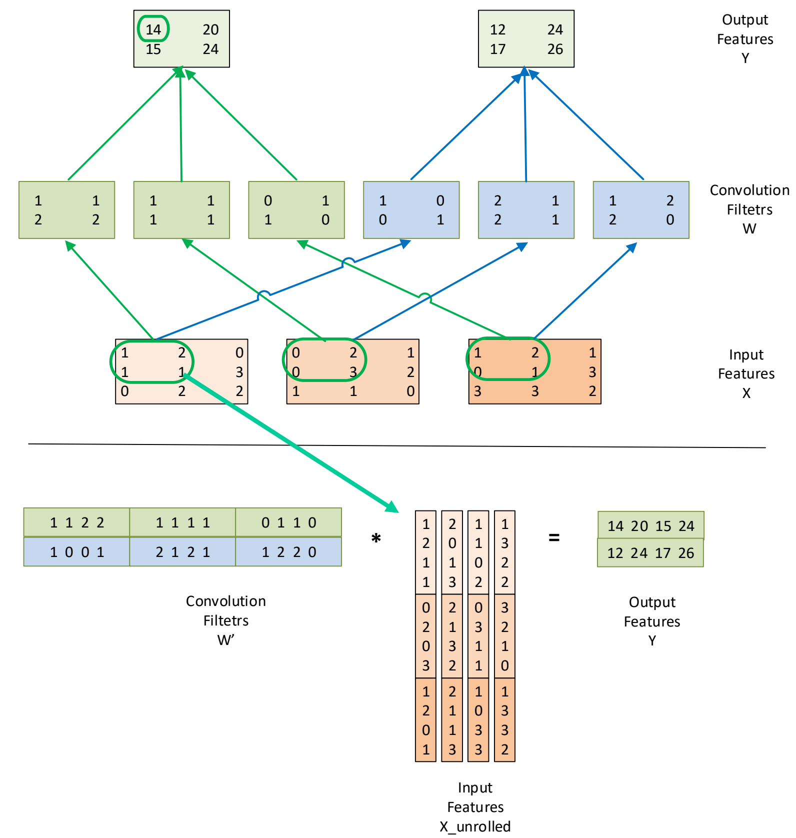


Memory Access Efficiency of Original Convolution Algorithm

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

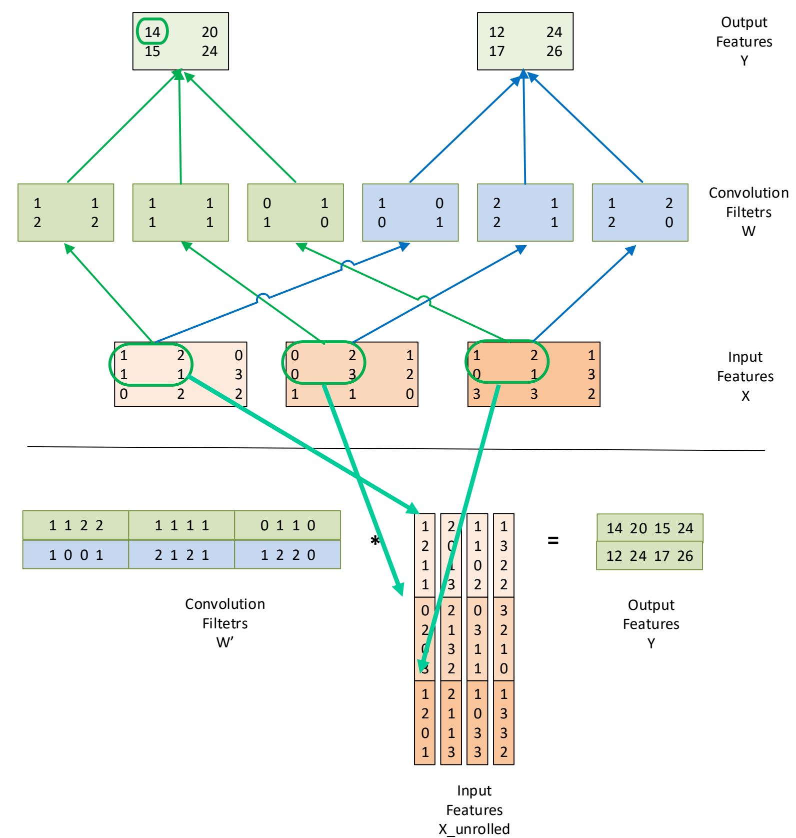
Properties of the Unrolled Matrix

- 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)



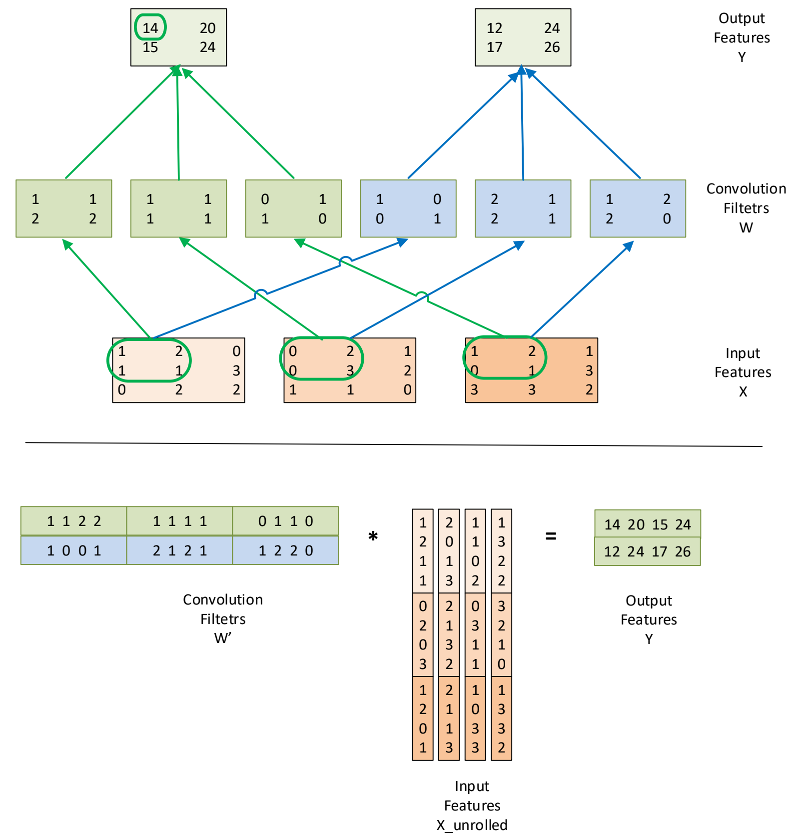
Properties of the Unrolled Matrix (cont.)

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



Input to Unrolled Matrix Mapping

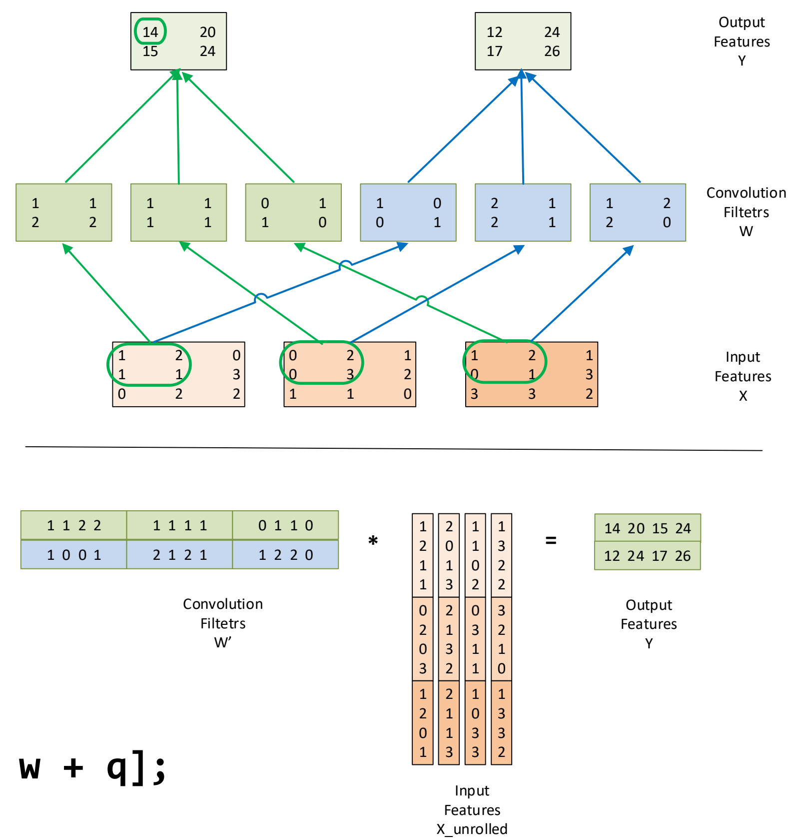
```
Output element (h, w)
Mask element (p, q)
Input feature map c

// 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];
```



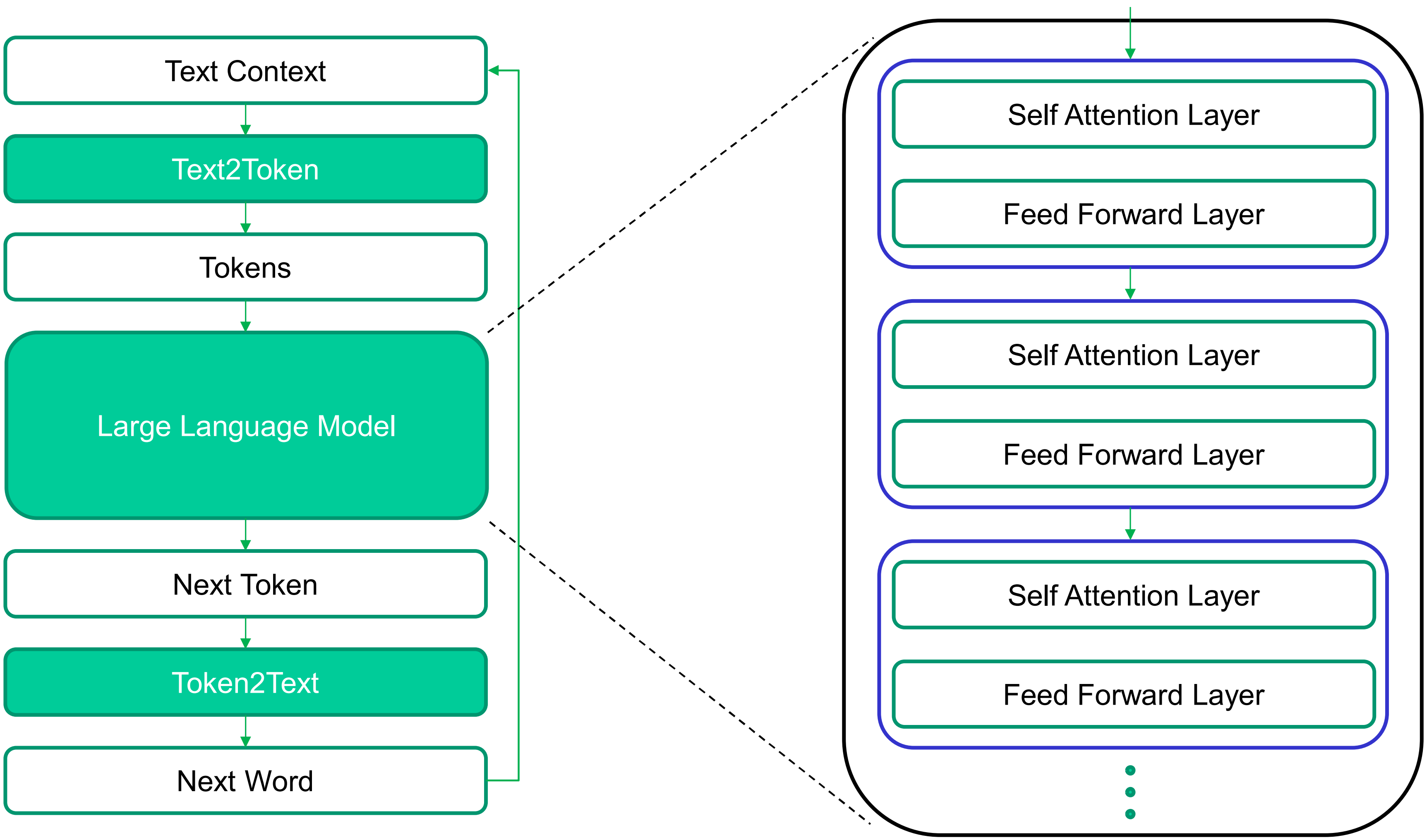
Function to generate “unrolled” X

```
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
            }
    }
}
```

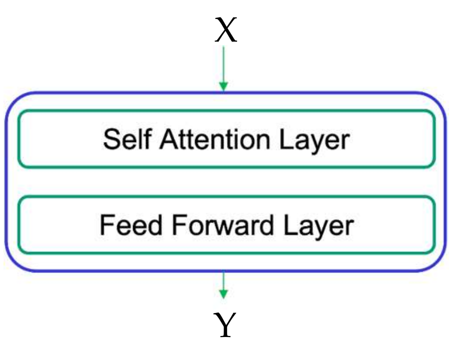
Implementation Strategies for a Convolution Layer

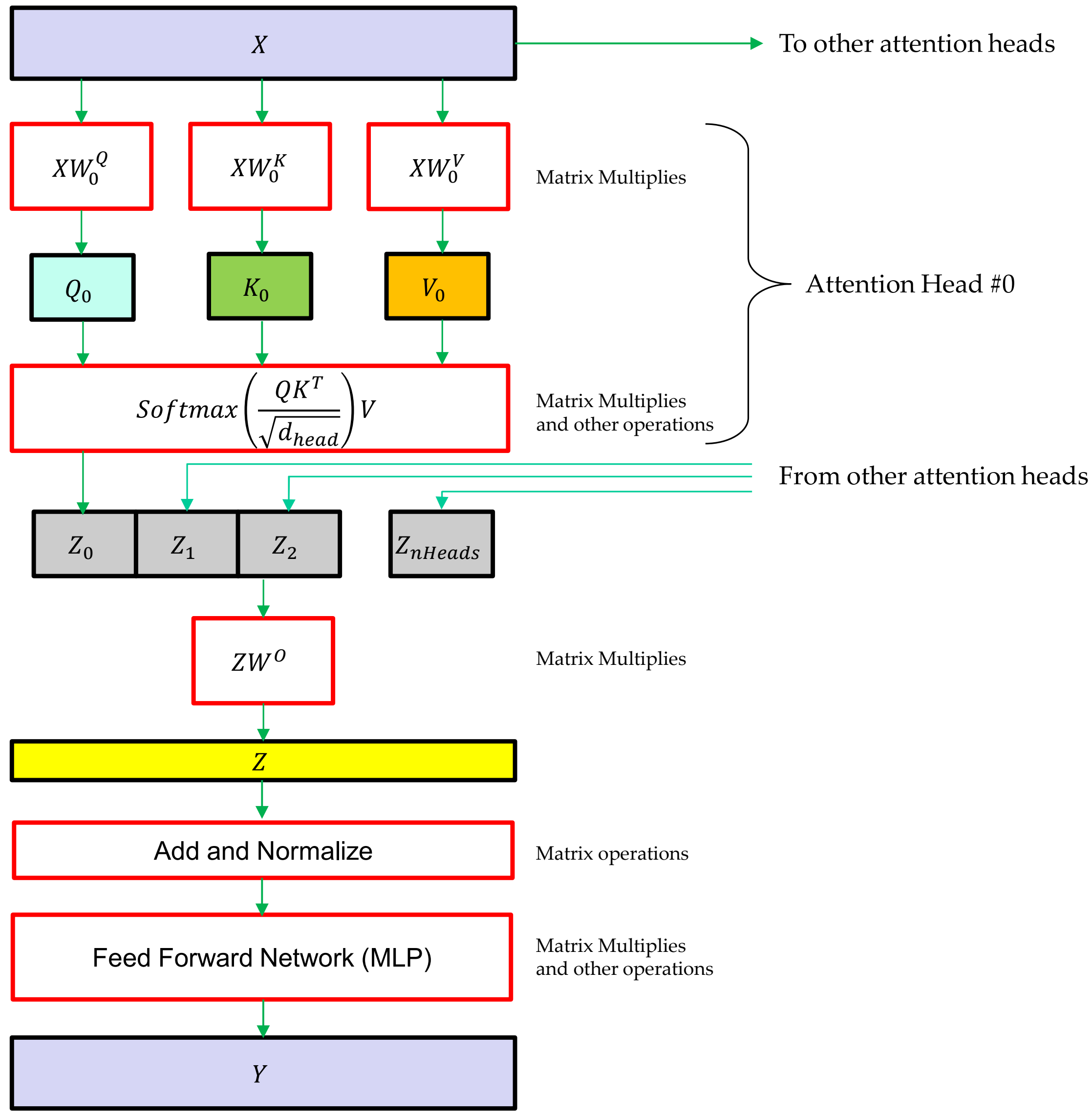
- Baseline
  - Tiled 2D convolution implementation, use constant memory for convolution masks
- Matrix-Multiplication Baseline
  - Input feature map unrolling kernel, constant memory for convolution masks as an optimization
  - Tiled matrix multiplication kernel
- Matrix-Multiplication with built-in 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





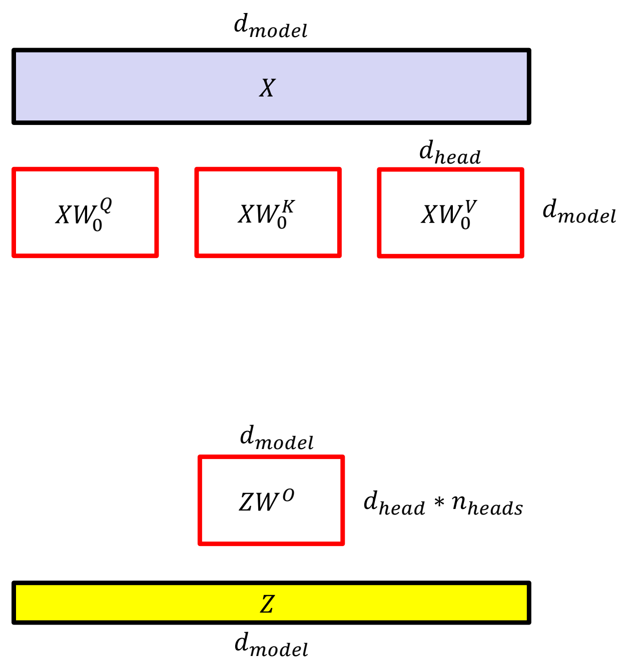
GPT-3, as an example

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

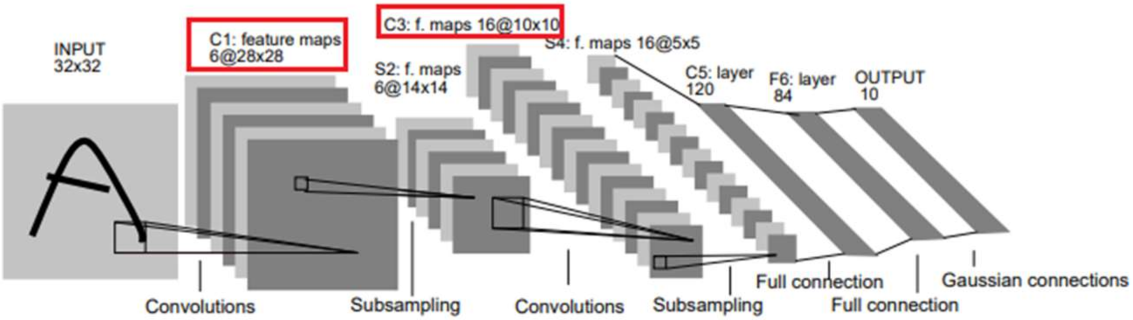
There are 96x3 of these for a total of 432M parameters, 864MB (@ 16 bit floats), or 1.7 GFlop per X vector

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)
- The network will be classifying Fashion MNIST dataset
- Some network parameters to be aware of
  - Input Size: 86x86 pixels, batch of 1k-10k images
  - Input Channels: 1
  - Convolutional kernel size: 7x7
  - Number of kernels: Variable (your code should support this)

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



http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf

QUESTIONS?

READ CHAPTER 16!