ECE408/CS483/CSE408 Spring 2025

Applied Parallel Programming

Lecture 12: Computation in Deep Neural Networks

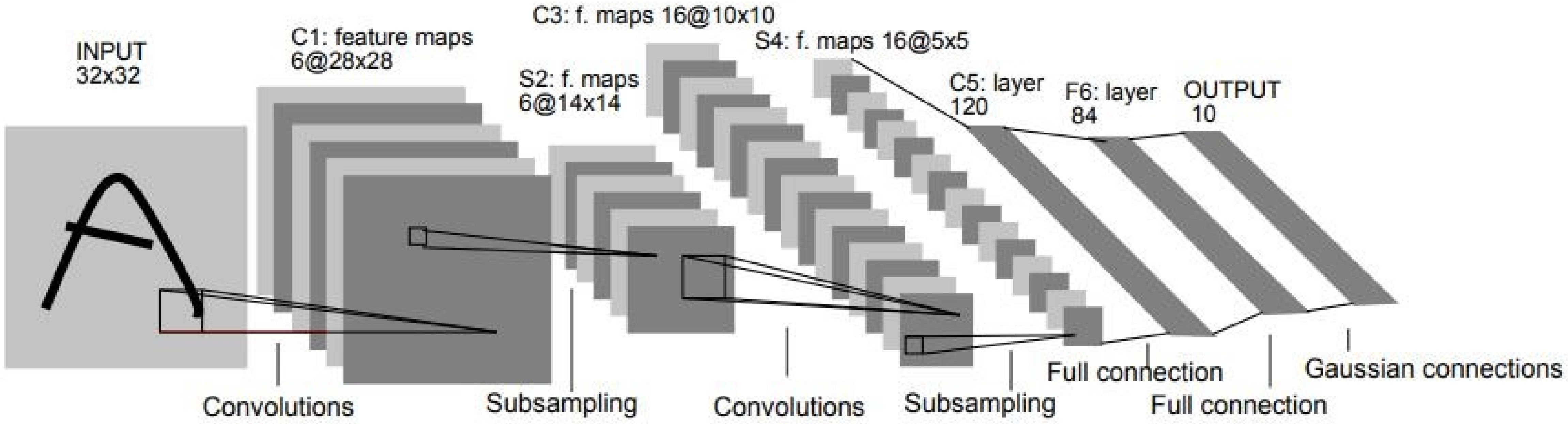
Course Reminders

- Midterm 1 is on Tuesday, March 4th
  - See Canvas for details, including exact time and topics
  - Please email your lecture instructor if you have a conflict
- Project Milestone 1

Objective

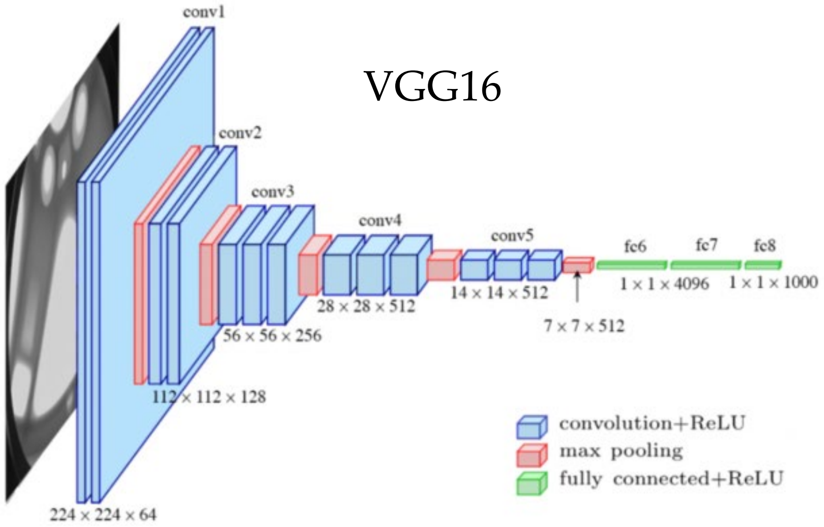
- To learn to implement the different types of layers in a Convolutional Neural Network (CNN)
- Brief intro to Project Milestone 1

LeNet-5:CNN for hand-written digit recognition

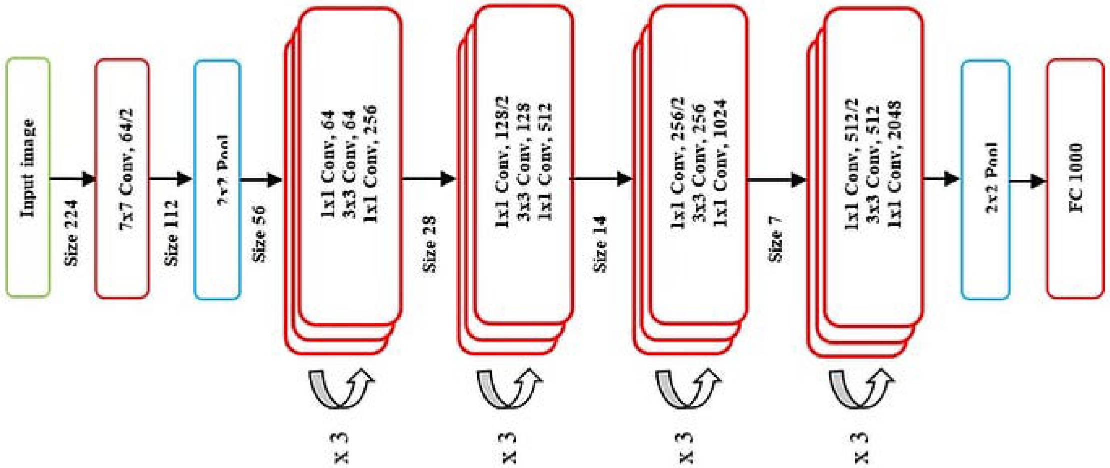


Many Types of CNNs

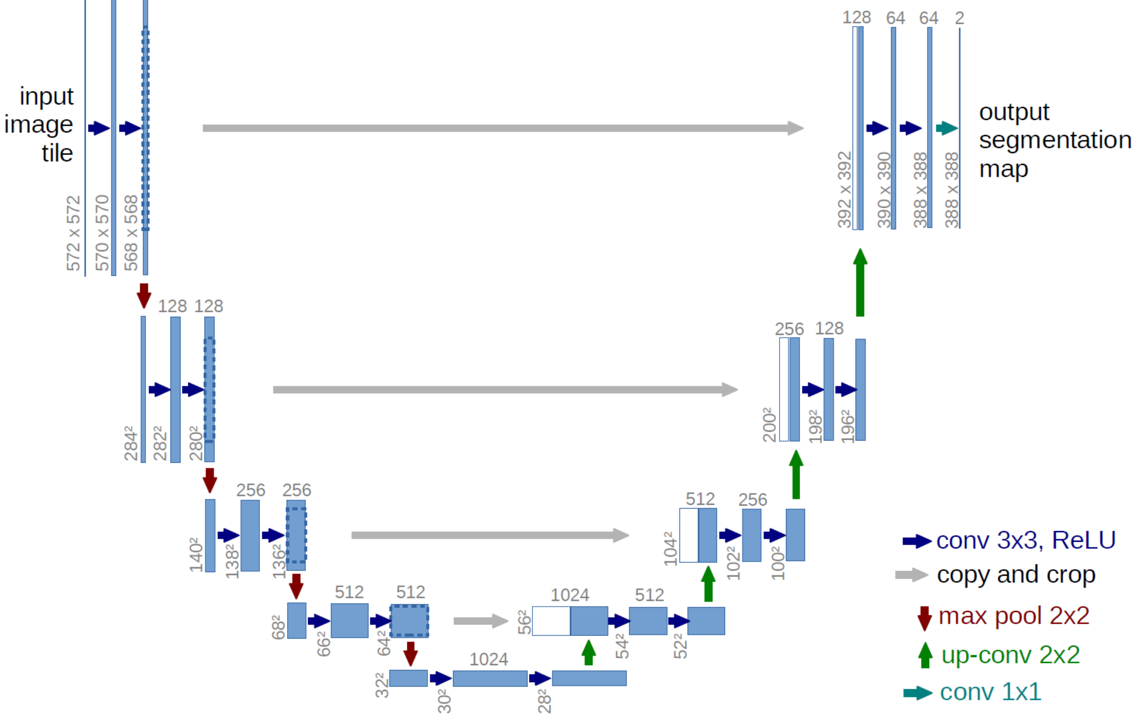
VGG16



ResNet50



U-Net



Anatomy of a Convolution Layer

Input features

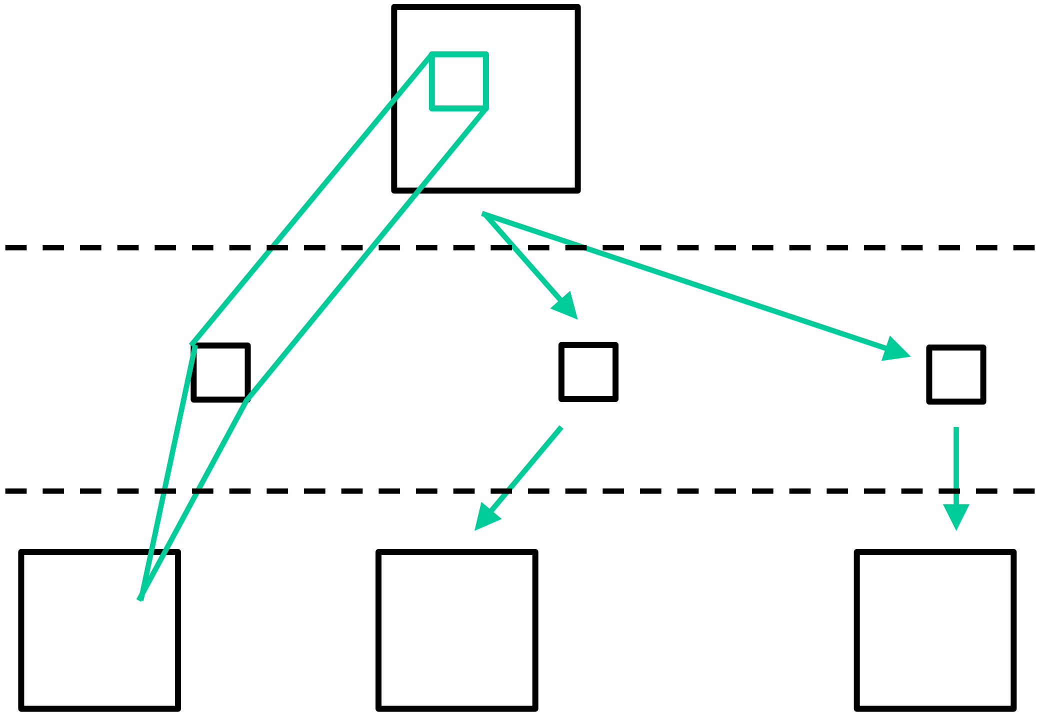
- A inputs each $N_1 \times N_2$

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

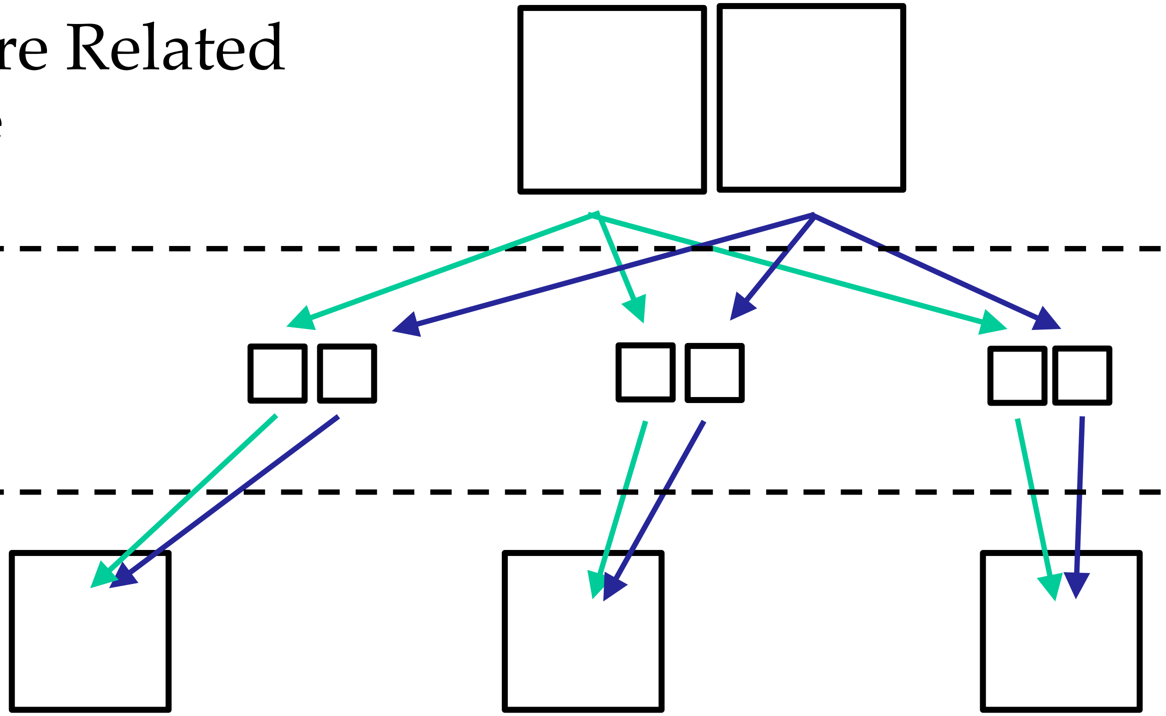
- For example: Red, Green, Blue

Convolution Layer

- Different kernels per channel

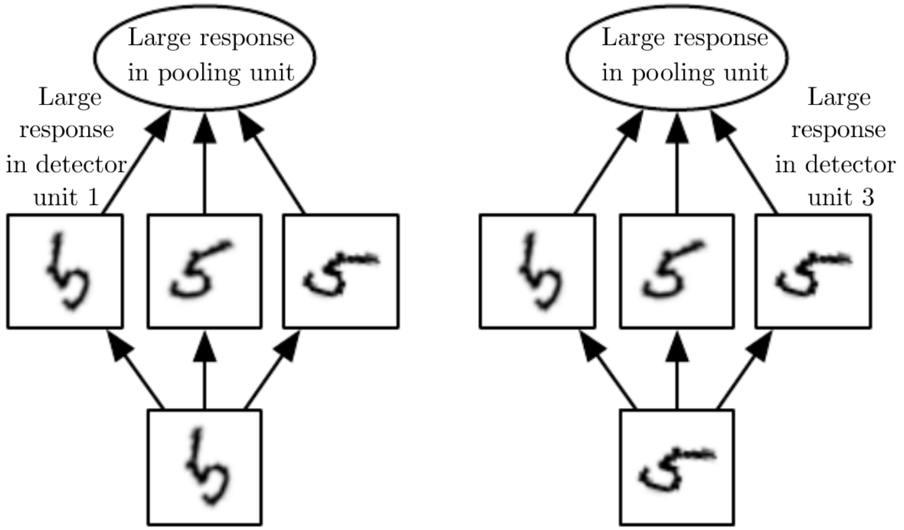
Output Features

- 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

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

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

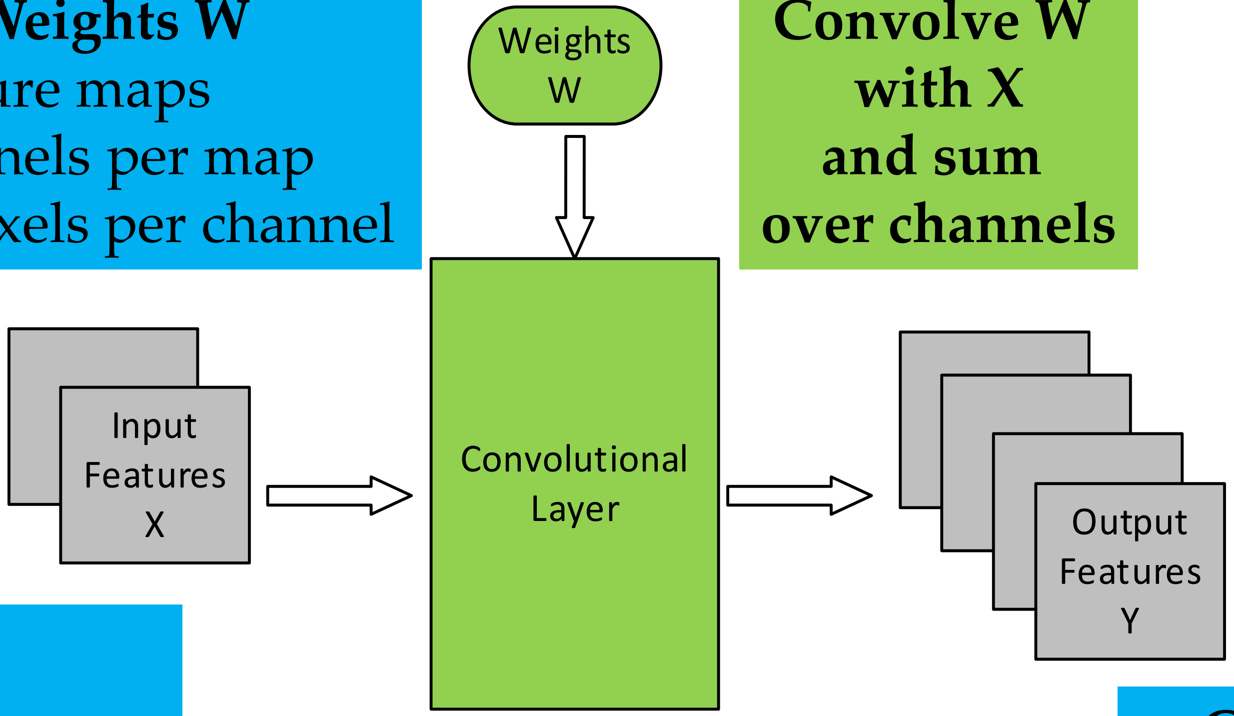
Weights W

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

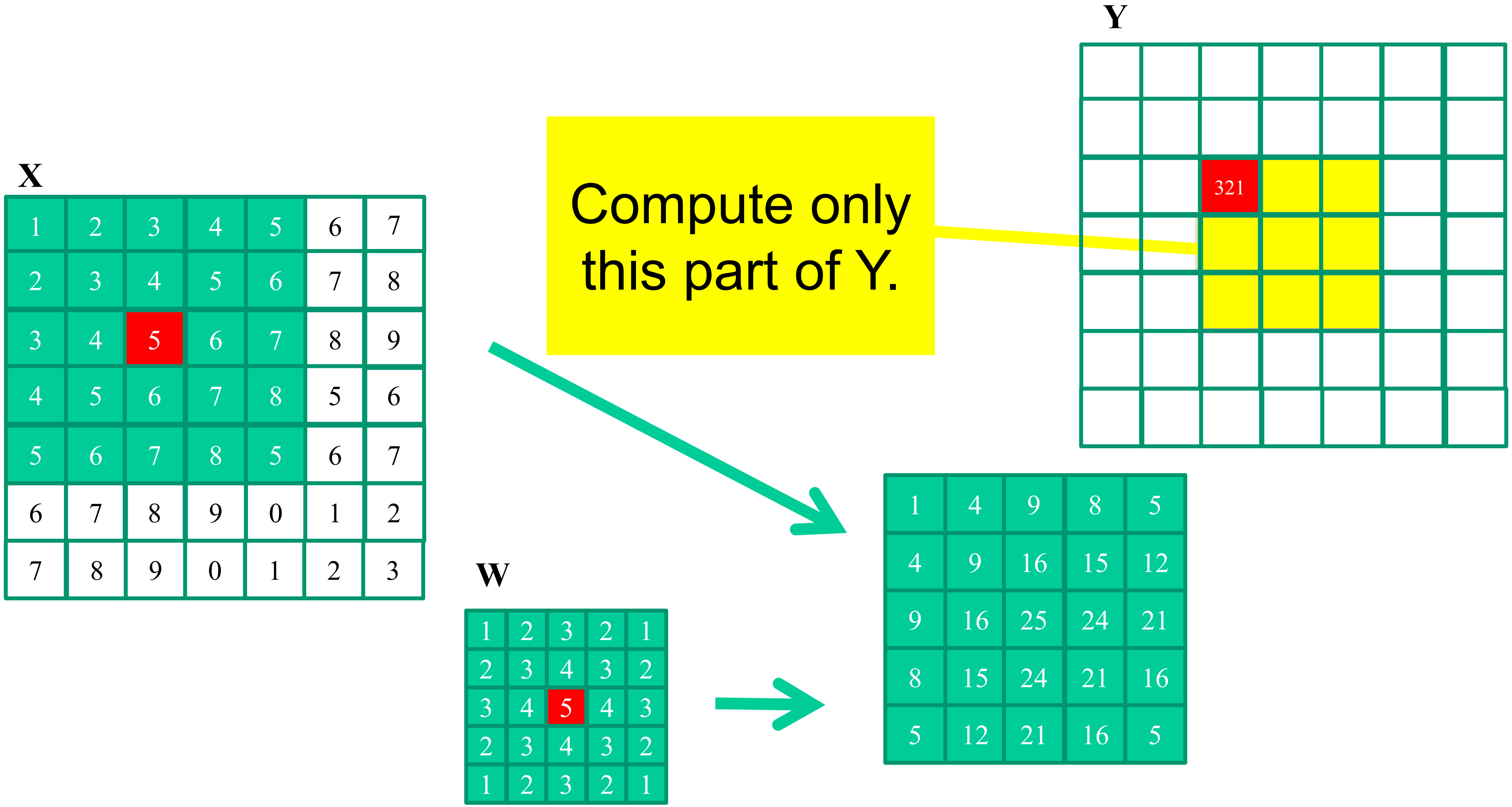
Convolution Output Y

B images

M features per image

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

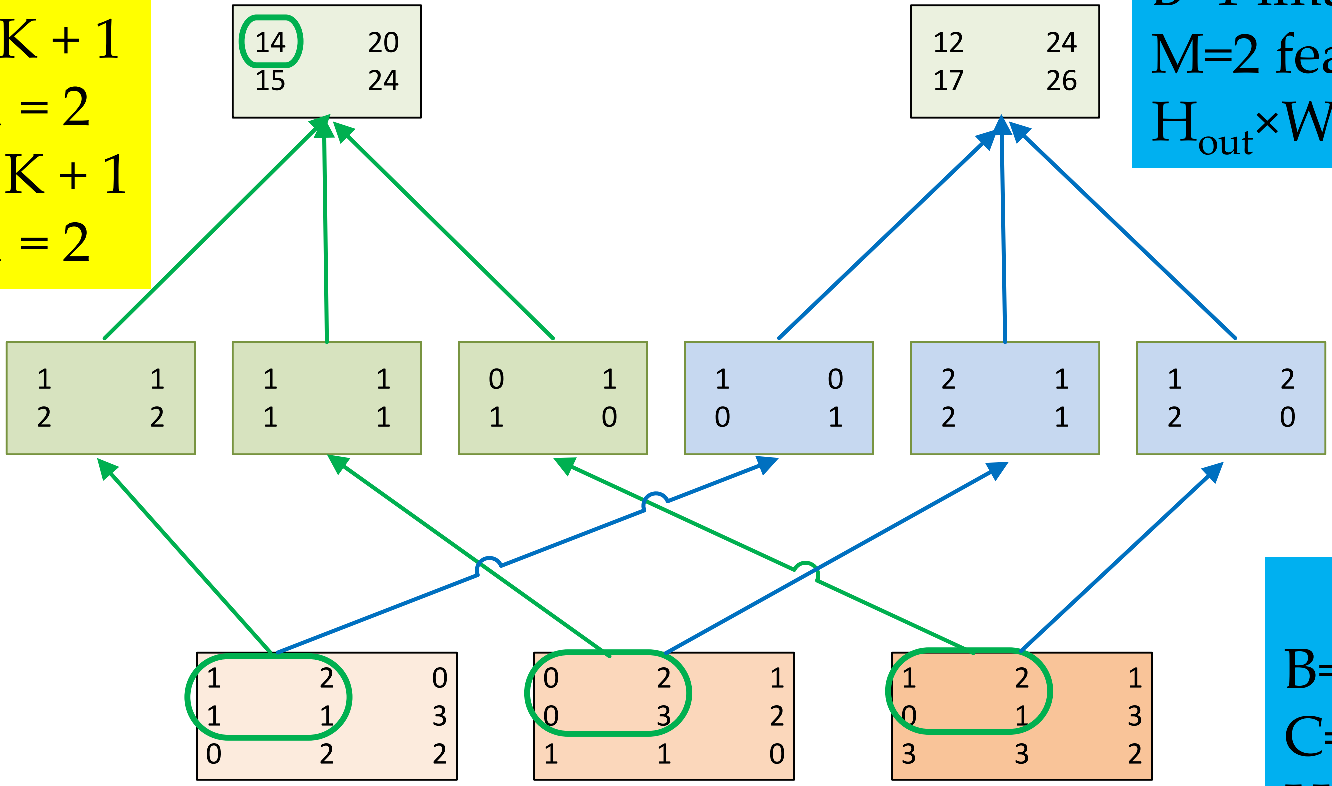
Outputs Must Use Full Mask/Kernel



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
$H_{out}\times W_{out}$=2×2 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

X[b,0,_, _]

$$\begin{bmatrix} 1 & 2 & 0 & 1 \\ 1 & 1 & 3 & 2 \\ 0 & 2 & 2 & 0 \\ 2 & 1 & 0 & 3 \end{bmatrix}$$

W[0,0,_, _]

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 3 \\ 2 & 1 & 0 \end{bmatrix}$$

X[b,1,_, _]

$$\begin{bmatrix} 0 & 2 & 1 & 0 \\ 0 & 3 & 2 & 1 \\ 1 & 1 & 0 & 2 \\ 2 & 1 & 0 & 3 \end{bmatrix}$$

W[0,1,_, _]

$$\begin{bmatrix} 1 & 2 & 3 \\ 1 & 1 & 0 \\ 3 & 0 & 1 \end{bmatrix}$$

X[b,2,_, _]

$$\begin{bmatrix} 1 & 2 & 1 & 0 \\ 0 & 1 & 3 & 2 \\ 3 & 3 & 2 & 0 \\ 1 & 3 & 2 & 0 \end{bmatrix}$$

W[0,2,_, _]

$$\begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

$\boxed{3+13+2}$

Y[b,0,_, _]

$$\begin{bmatrix} 18 & ? \\ ? & ? \end{bmatrix}$$

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

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

… 18+7+3+3

Y[b,0,_, _]

31	?
?	?

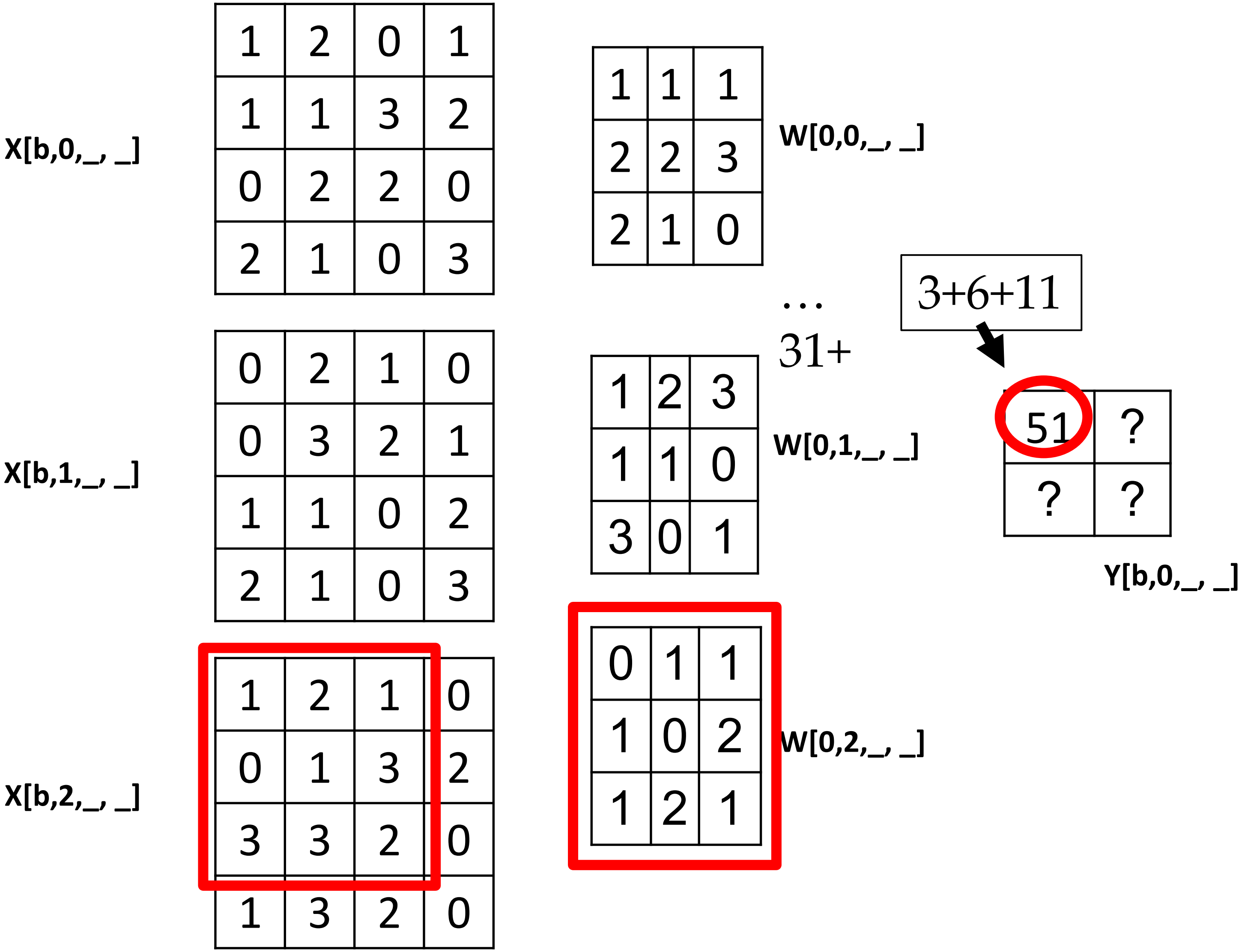
X[b,2,_, _]

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 = 2



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

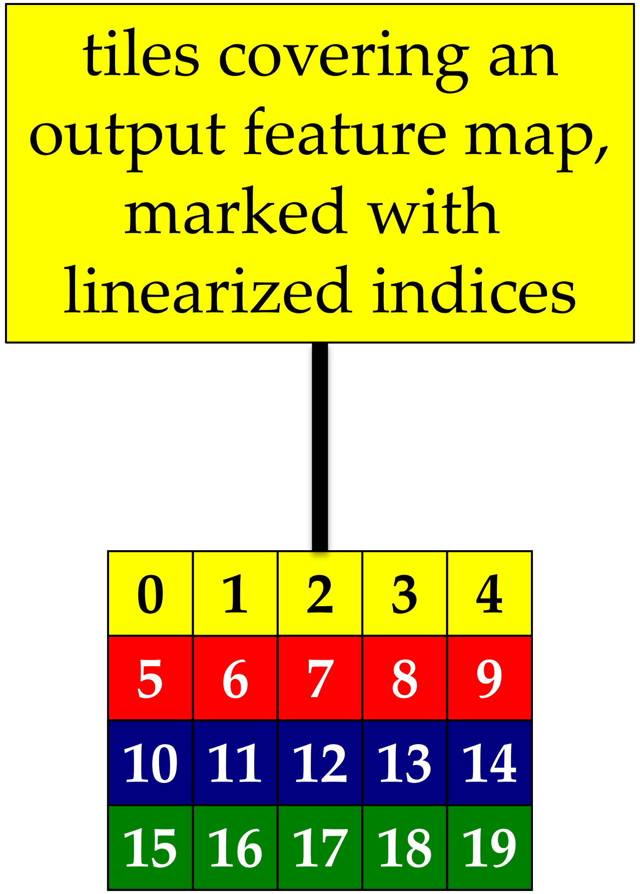
```
void poolingLayer_forward(int B, int M, int H_out, int W_out, int N, float* Y, float* S)
{
  for (int b = 0; b < B; ++b)                     // for each image
    for (int m = 0;  m < M; ++m)                   // for each output feature map
      for (int x = 0; x < H_out/N; ++x)           // for 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

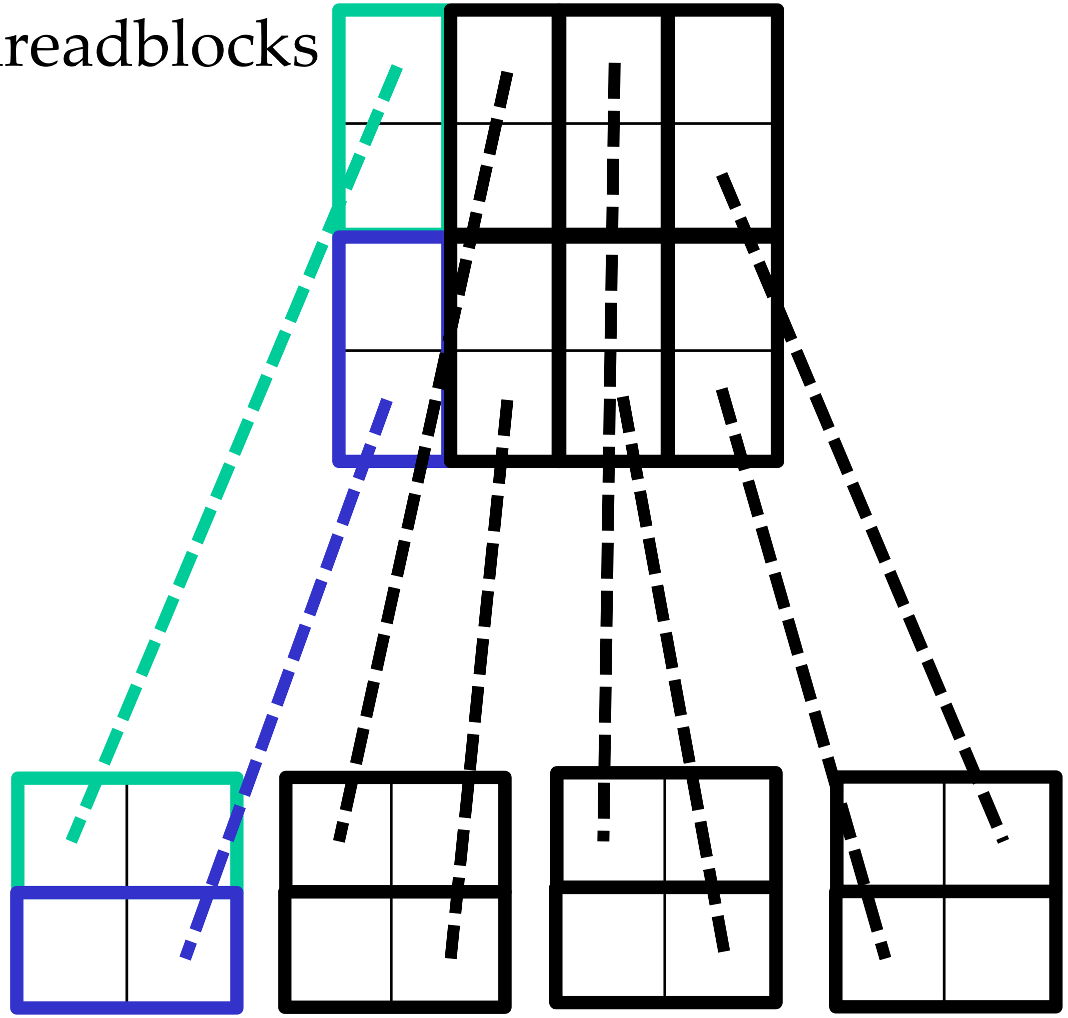
Assume

- **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.

CUDA Grid and Threadblocks



Output Feature Maps and Tiles

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

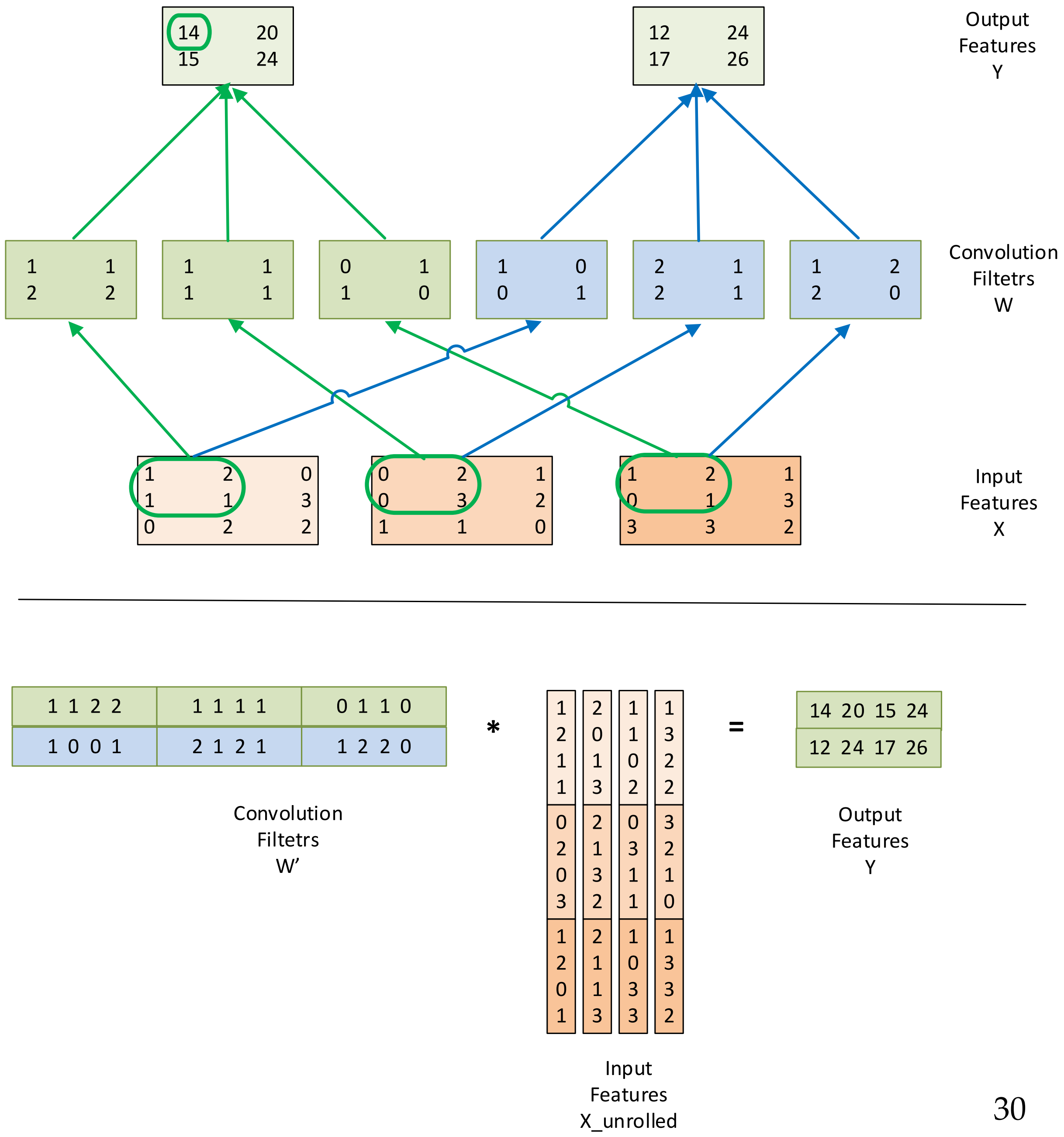
Enough parallelism

- 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.

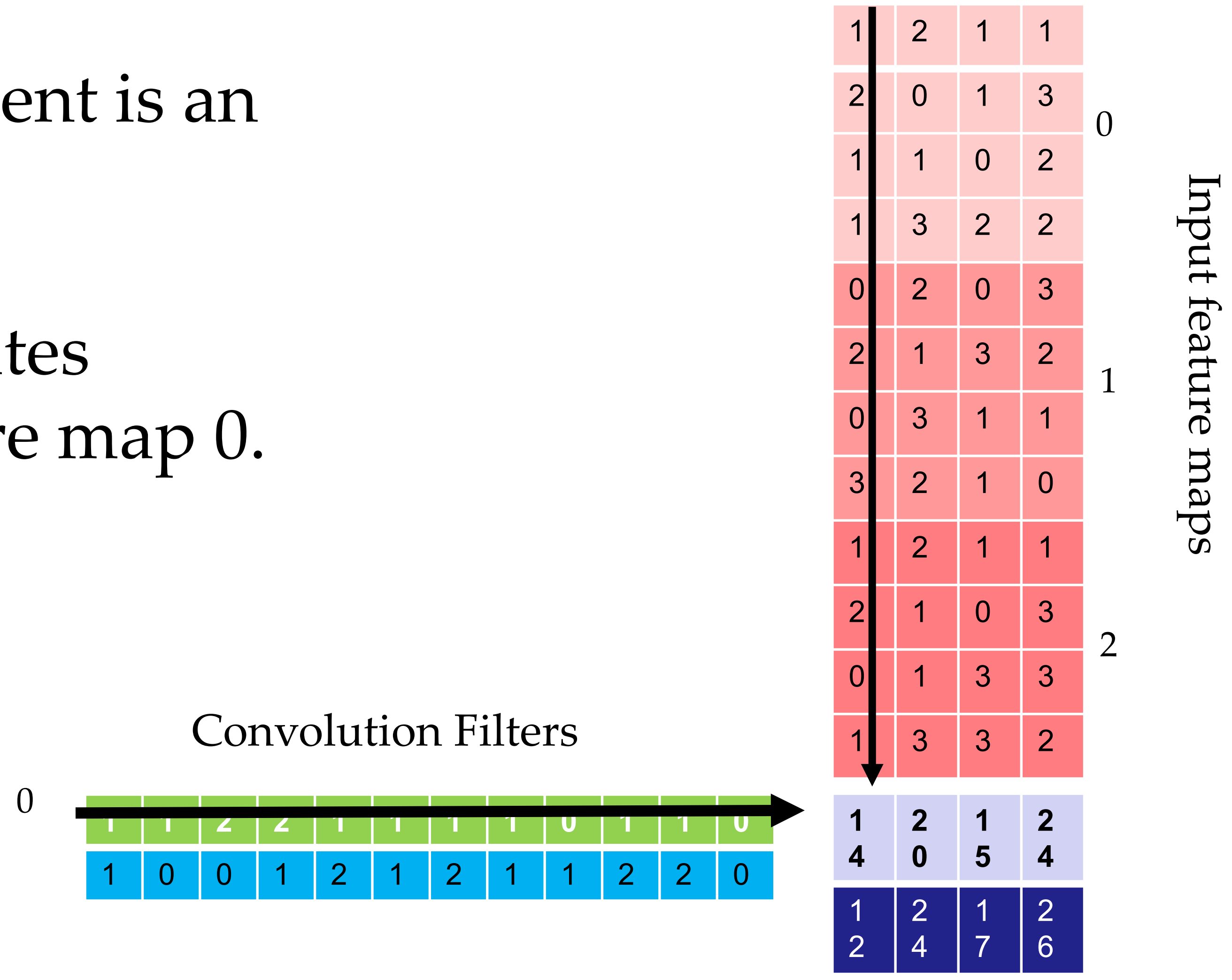
Implementing a Convolution Layer with Matrix Multiplication



Simple Matrix Multiplication

Each product matrix element is an output feature map pixel.

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

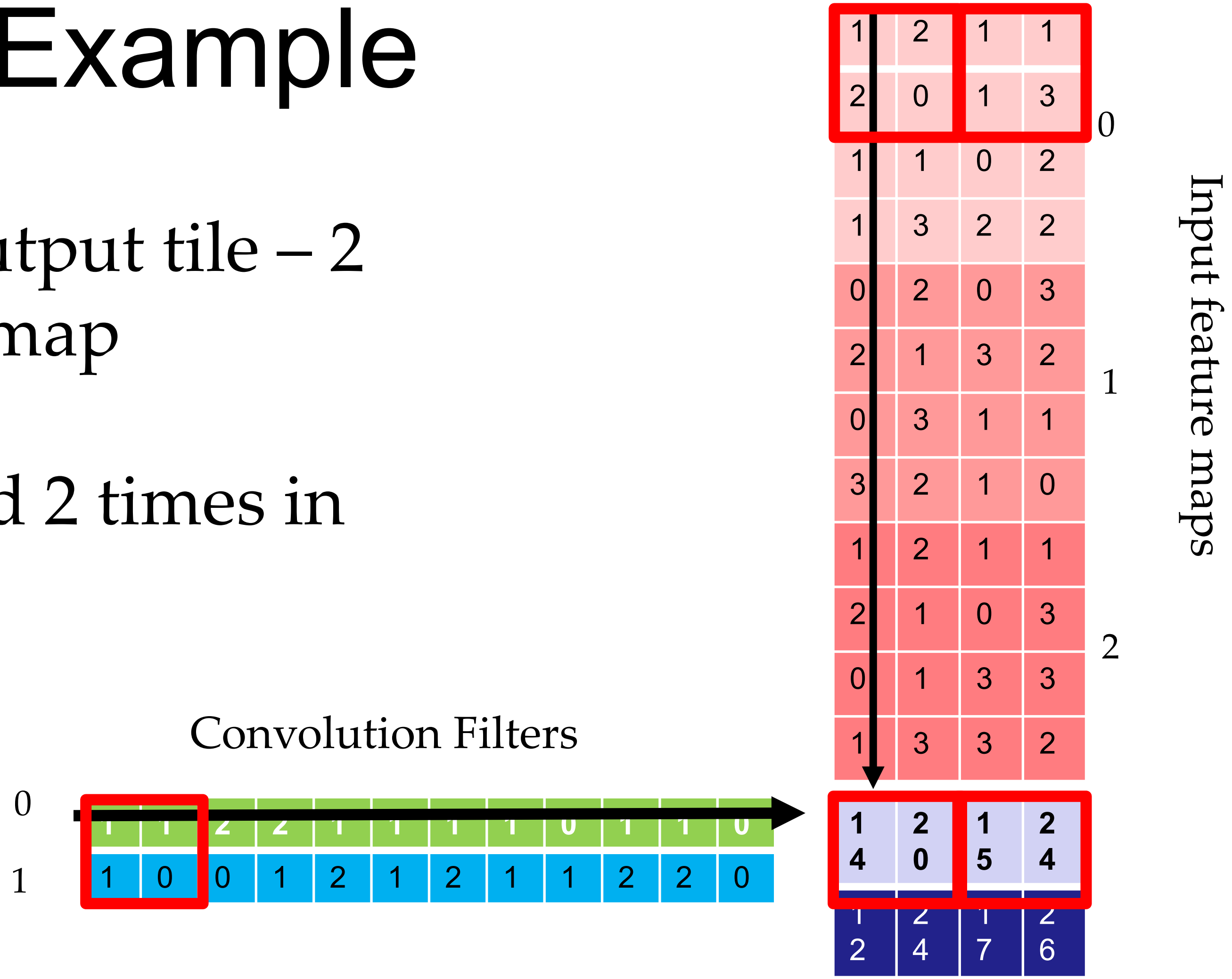


Tiled Matrix Multiplication

2x2 Example

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

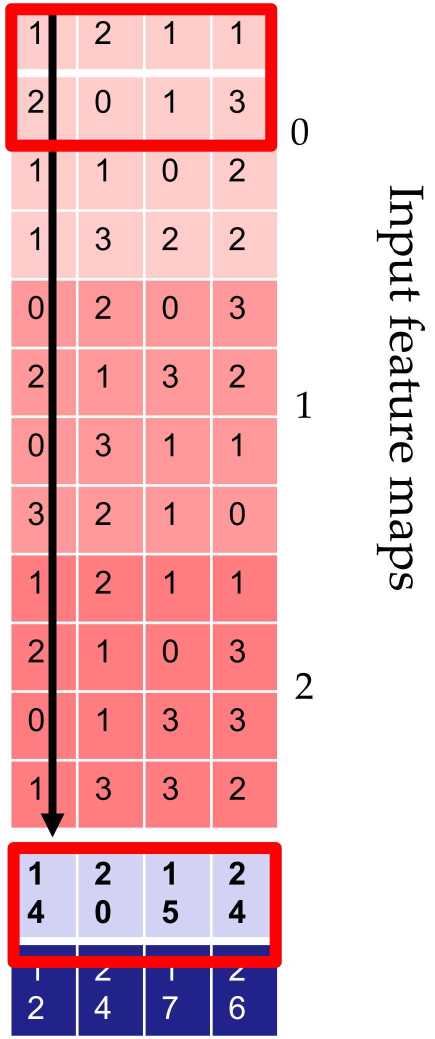
Each input element is reused 2 times in the shared memory

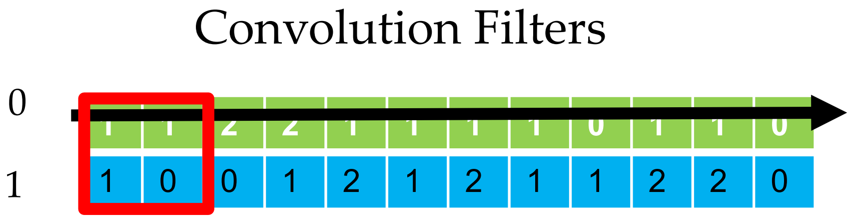


Tiled Matrix Multiplication
2x4 Example

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

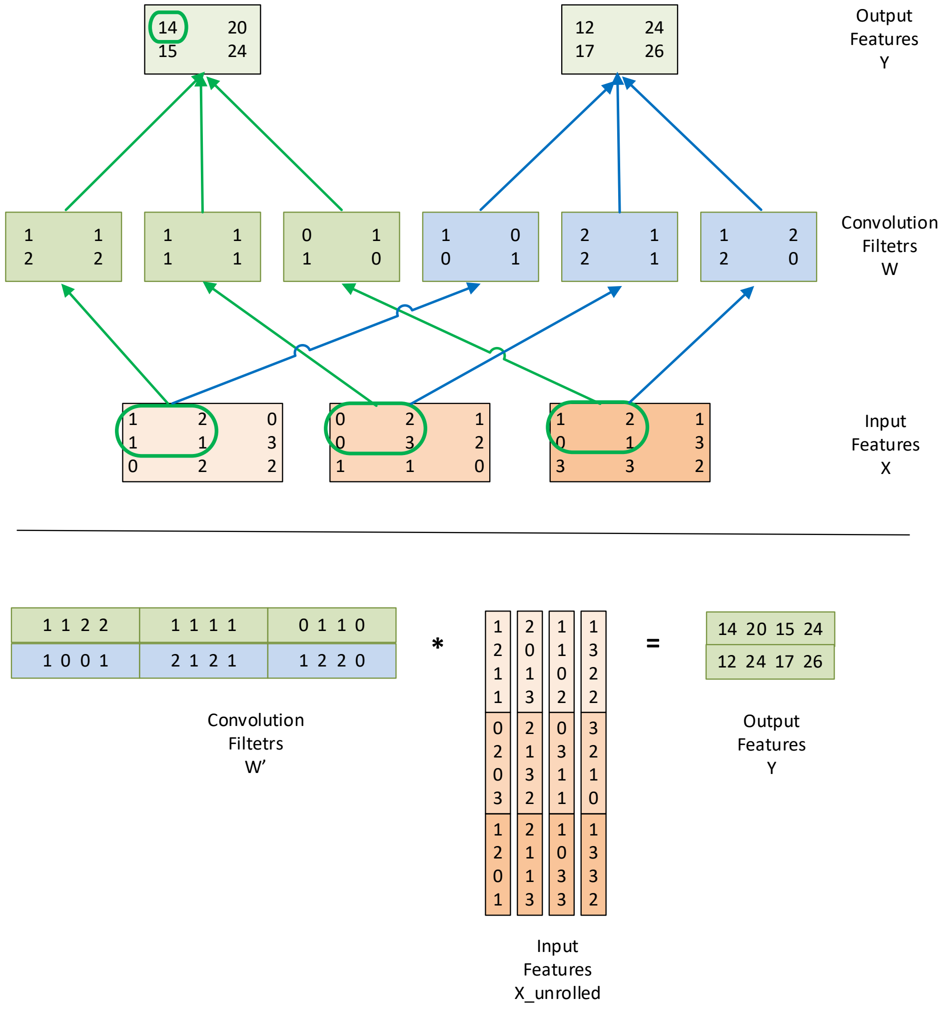
Each input element is reused 2 times in the shared memory





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

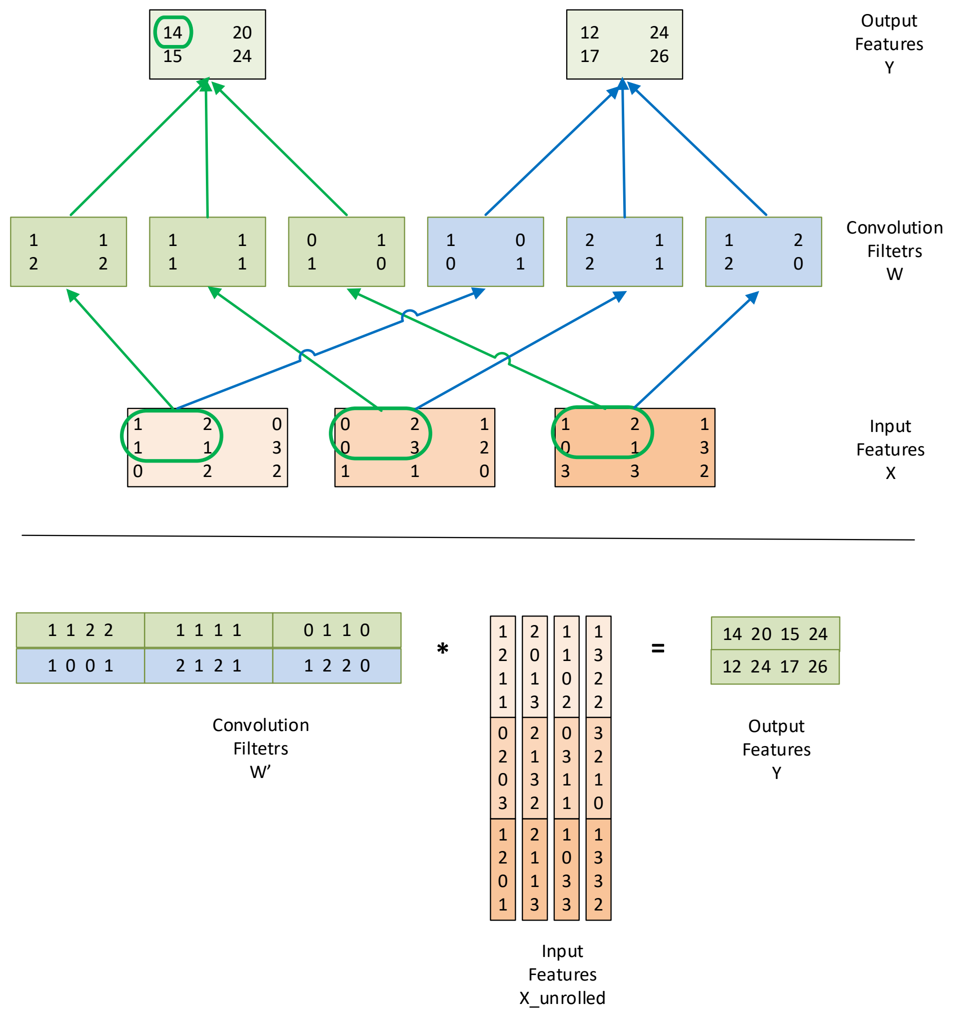
W_out = 2

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

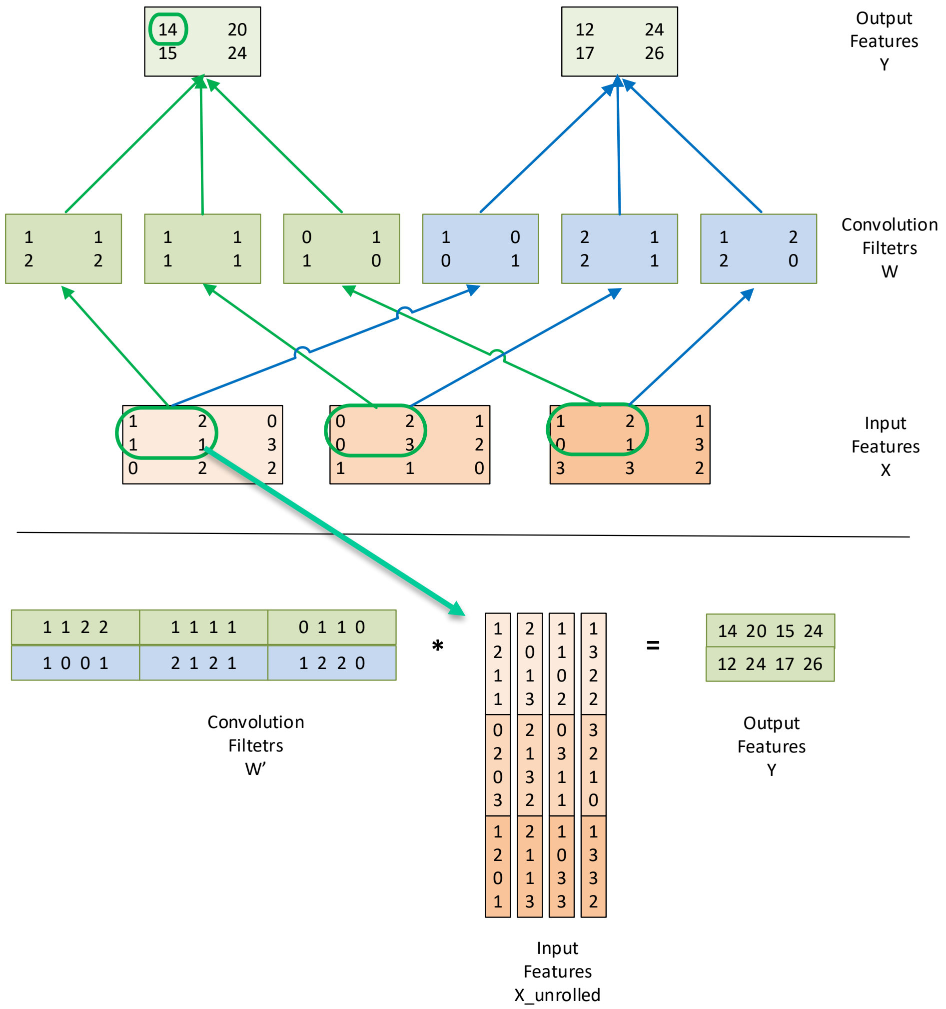


Memory Access Efficiency of Original Convolution Algorithm

- 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

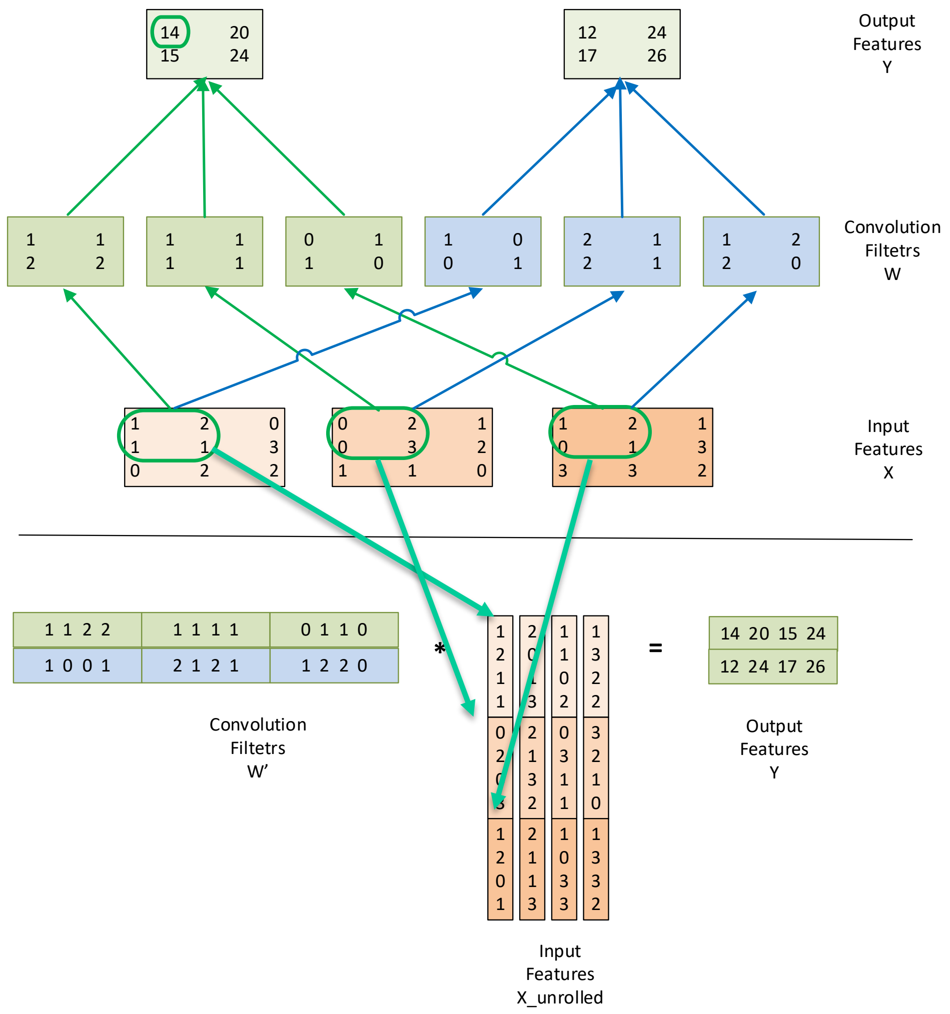
Properties of the Unrolled Matrix

- 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)



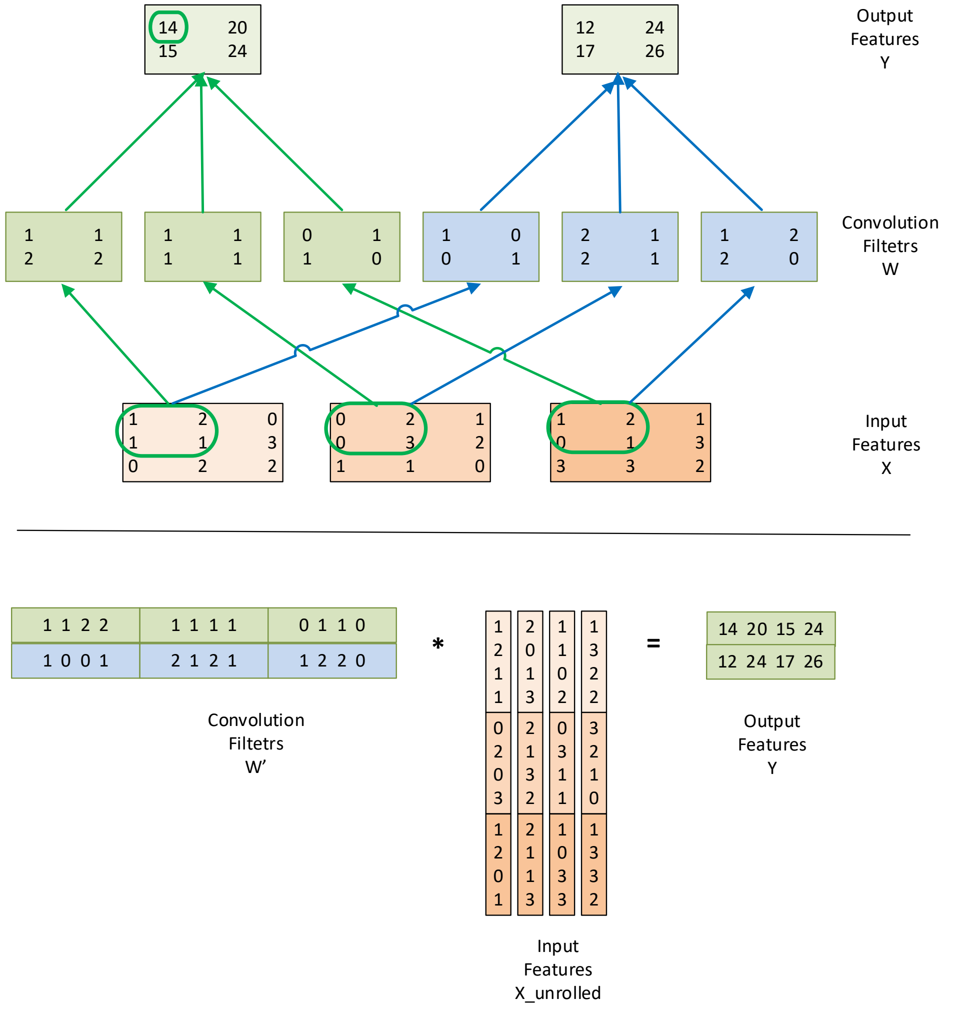
Properties of the Unrolled Matrix (cont.)

- 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

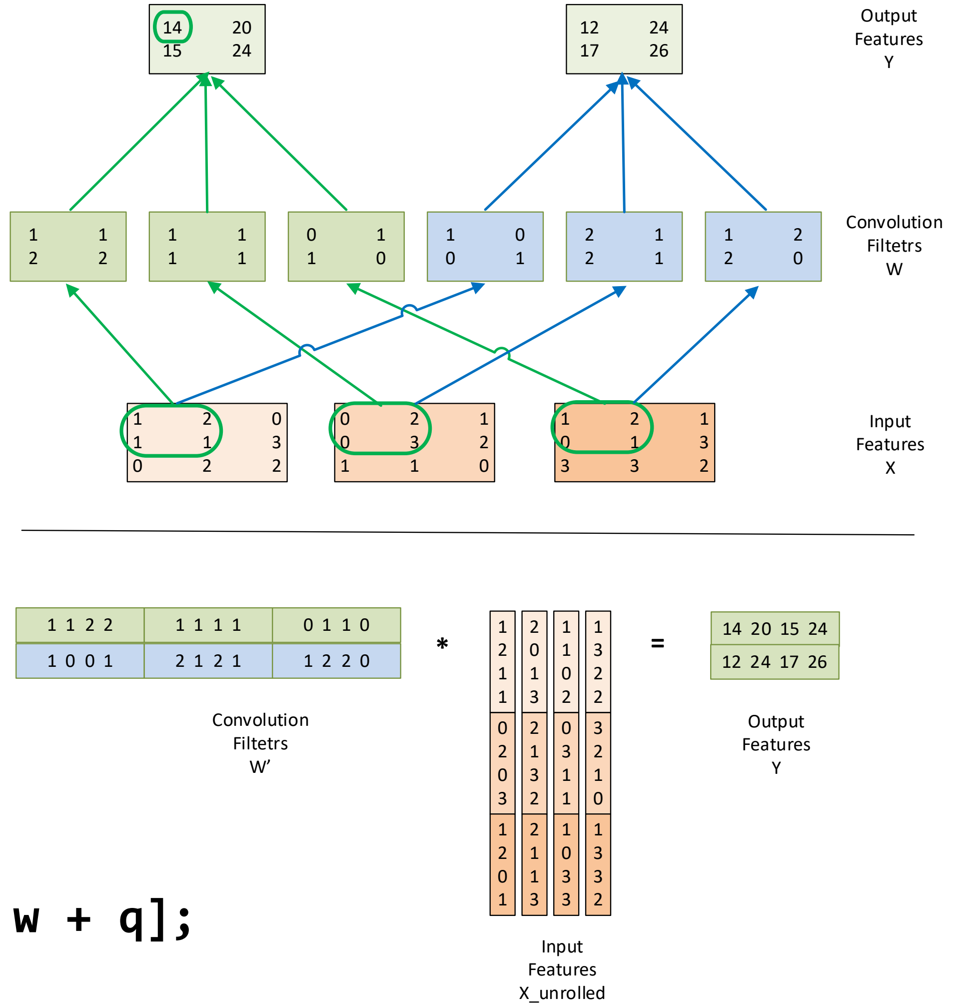
```
Output element (h, w)
Mask element (p, q)
Input feature map c

// 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

ANY MORE QUESTIONS? READ CHAPTER 16

Problem Solving

- Q: Consider a 3D video filtering (convolution) code in CUDA with a 5x5x7 mask, which is stored in constant memory. Shared memory is used to fully store the input tile required for an 16x16x16 output tile (for example, using strategy 2). What is the ratio of total global memory read operations to shared memory accesses for one output tile? For this question, only consider interior tiles with no ghost elements.
- A:
  - 20*20*22 to 5*5*7*16*16*16

Problem Solving

- Q: Consider a convolutional neural network that takes 100x200 images with 3 color channels (red, green, blue). The first layer of this network generates 10 output feature maps using 9x9 filters, where all channels are combined in each output feature map. Assuming all convolutions are performed in floating point, and considering only the convolutional layer (e.g., no pooling, thresholding, non-linearity, etc.), how many floating-point operations (both multiplications and additions) are required to generate all the output feature maps in a single forward pass? Remember: output feature maps are smaller than input maps because only pixels without ghost elements are generated.
- A:

  10 (output feature maps)

  * 92*192 (total # of pixels to compute in each output feature map)

  * 9*9 (conv. filter size)

  * 3 (color channels)

  * 2 (multiply-add operations)