ECE408/CS483/CSE408 Fall 2021

Applied Parallel Programming

Lecture 13: Optimizing Convolutional Layers & Final Project Kickoff

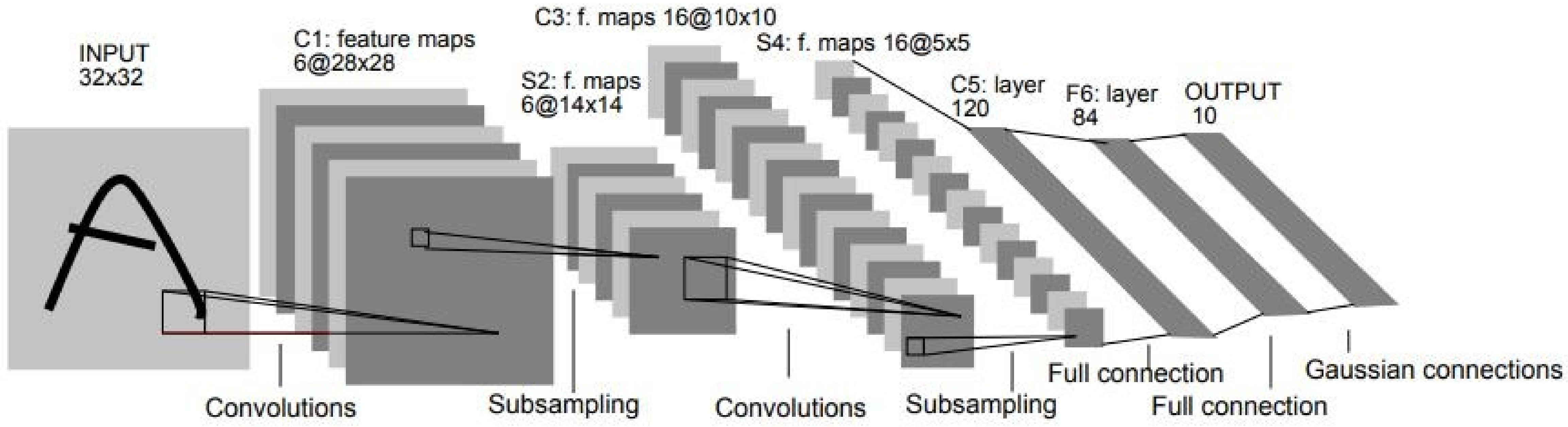
Course Reminders

- Midterm 1 is on this Thursday, October 7th
  - On-line, everybody will be taking it at the same time
    - Thursday, Oct. 7th 8:00pm-9:20pm US Central time
    - Friday, Oct. 9th 9:00am-10:20am Beijing time
  - Includes materials from Lecture 1 through Lecture 10
  - See **https://wiki.illinois.edu/wiki/display/ECE408/Exams** for details
- Project Milestone 1: Rai installation, CPU Convolution, Profiling
  - Due: Friday October 15th
  - Project details are posted on course wiki https://wiki.illinois.edu/wiki/display/ECE408/Project
  - In today's lecture, we will go over these details

Objective

- To learn to implement the different types of layers in a Convolutional Neural Network (CNN) in CUDA
- To understand how unrolling input X can improve performance for convolution layers on GPUs
- Review Final Project

LeNet-5:CNN for hand-written digit recognition



Sequential Code: Forward Convolutional Layer

```
void cL_f(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];
       }
}
```

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.

Sequential Code: Forward Pooling Layer

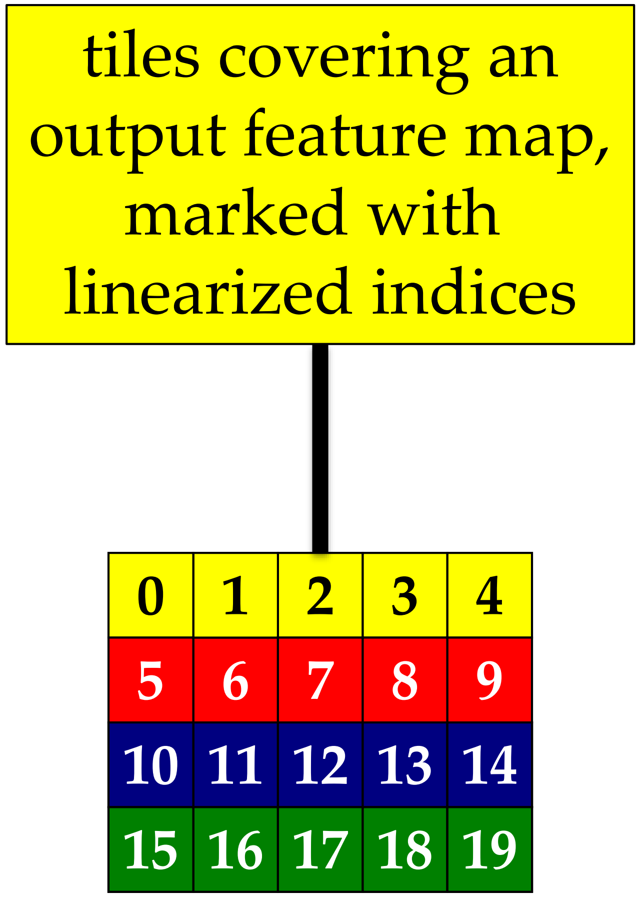
```
void pLayer_forward(int B, int M, int H_out, int W_out, int N, float* Y, float* S)
{
  for (int b = 0; b < B; ++b)                      // for each image
    for (int m = 0;  m < M; ++m)                   // for each output feature map
      for (int x = 0; x < H_out/N; ++x)            // for each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


tiles covering an output feature map, marked with linearized indices
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19

A Small Example

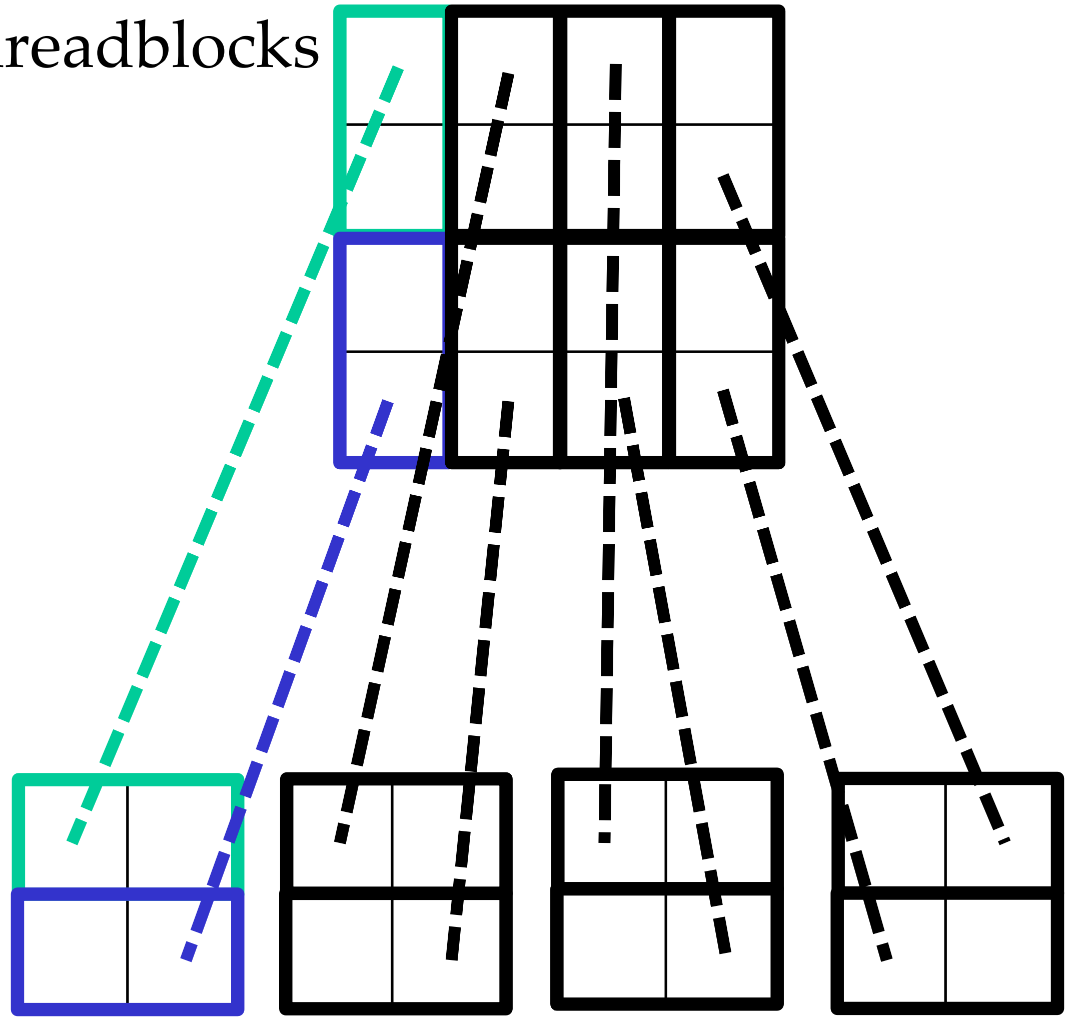
Assume

- **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.

CUDA Grid and Threadblocks

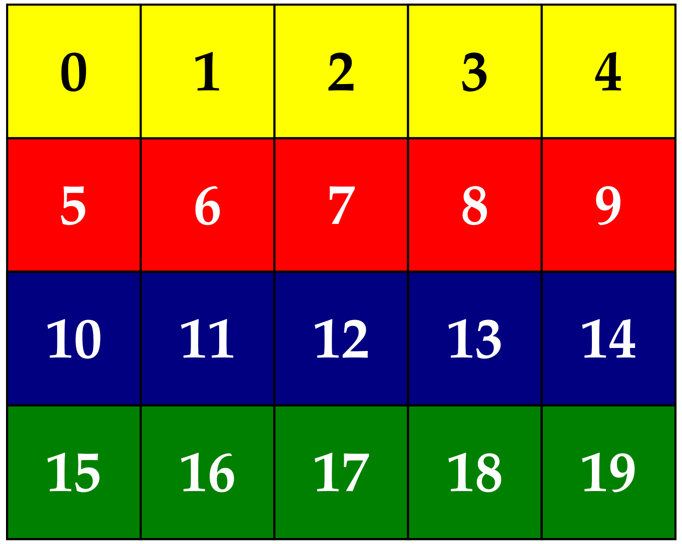


Output Feature Maps and Tiles

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**.



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 >>>(…);
```

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

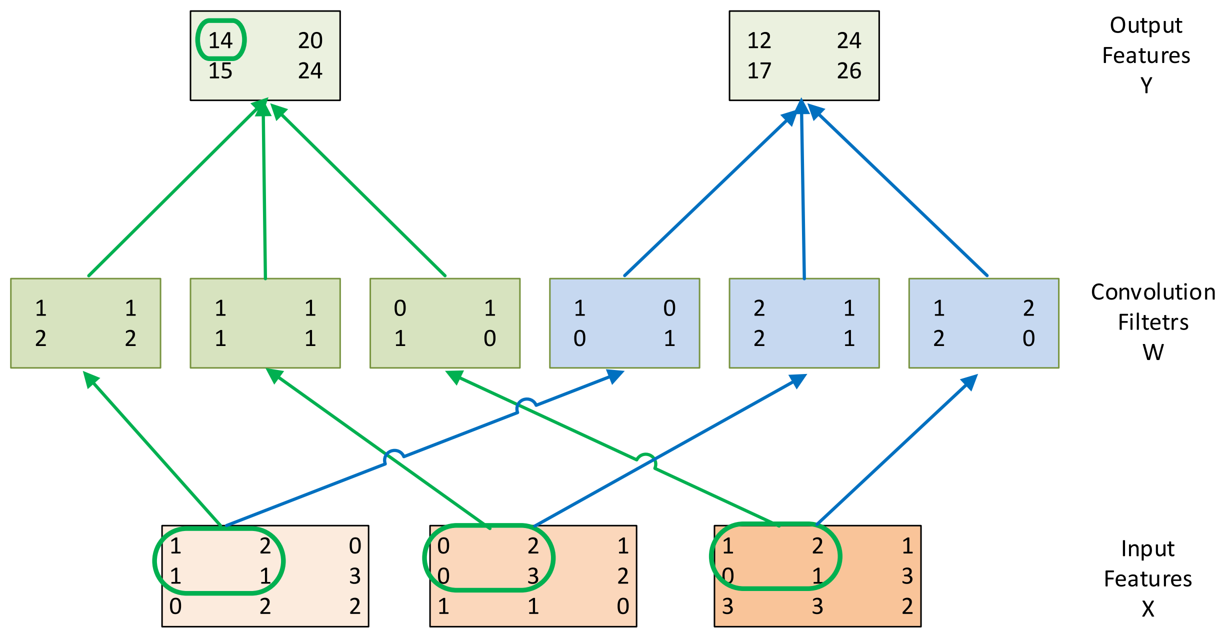
Enough parallelism

- 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

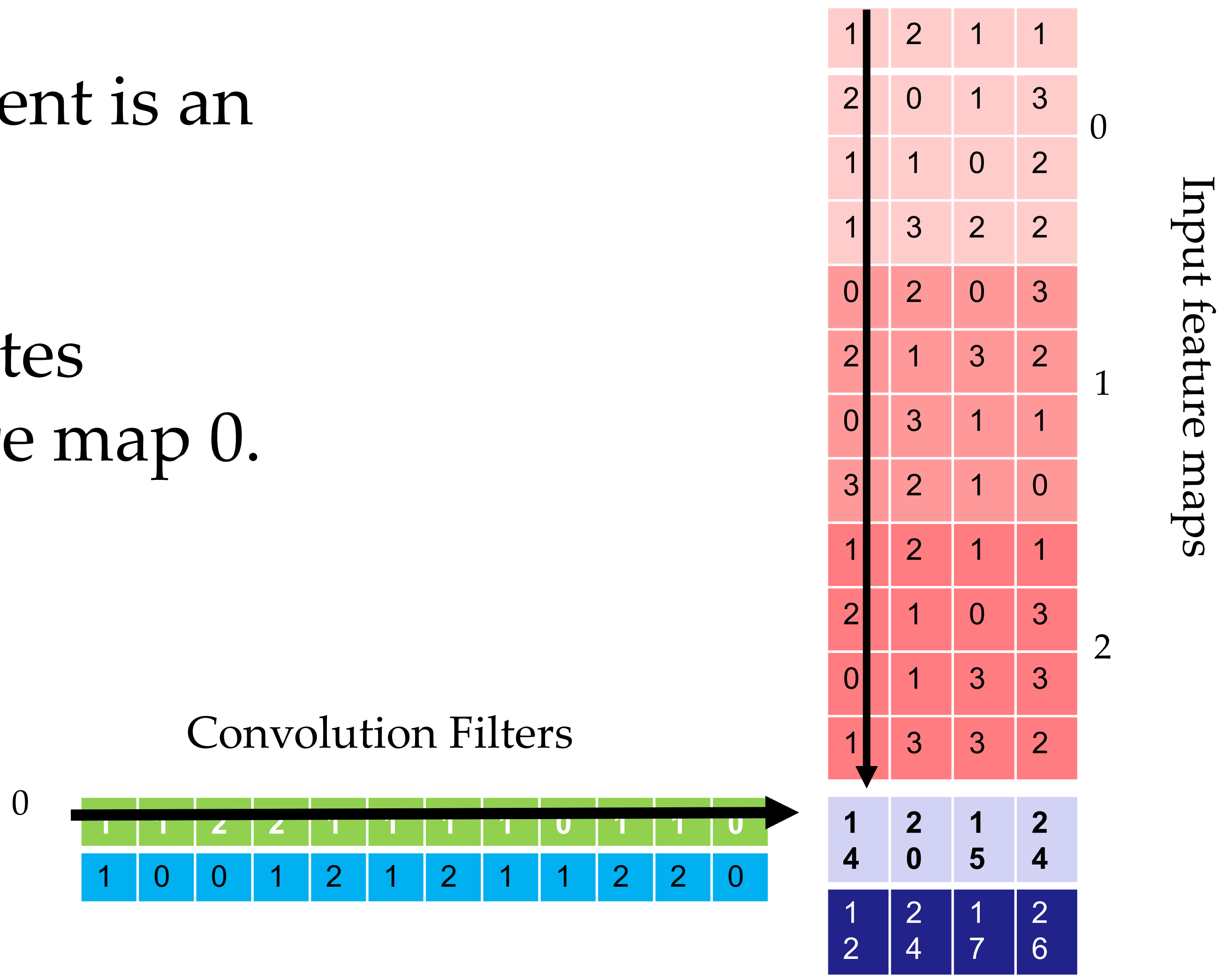


$$\underbrace{\left[\begin{array}{cccc|cccc|cccc} 1&1&2&2&1&1&1&1&0&1&1&0 \\ \hline 1&0&0&1&2&1&2&1&1&2&2&0 \end{array}\right]}_{\text{Convolution Filtetrs } W'} * \underbrace{\left[\begin{array}{cccc} 1&2&1&1 \\ 2&0&1&3 \\ 1&1&0&2 \\ 1&3&2&2 \\ \hline 0&2&0&3 \\ 2&1&3&2 \\ 0&3&1&1 \\ 3&2&1&0 \\ \hline 1&2&1&1 \\ 2&1&0&3 \\ 0&1&3&3 \\ 1&3&3&2 \end{array}\right]}_{\text{Input Features } X\_unrolled} = \underbrace{\left[\begin{array}{cccc} 14&20&15&24 \\ \hline 12&24&17&26 \end{array}\right]}_{\text{Output Features } Y}$$

Simple Matrix Multiplication

Each product matrix element is an output feature map pixel.

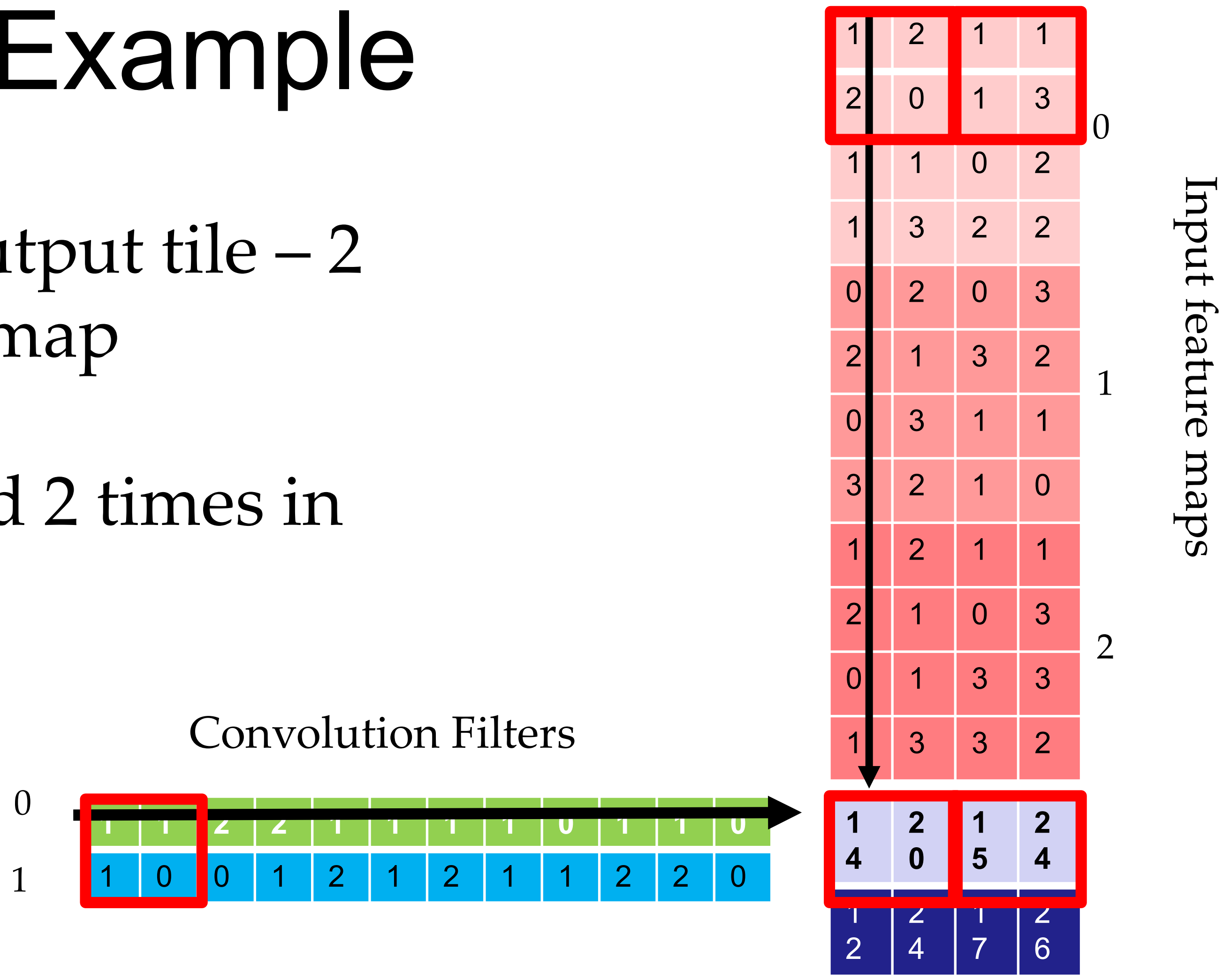
This inner product generates element 0 of output feature map 0.



Tiled Matrix Multiplication
2x2 Example

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

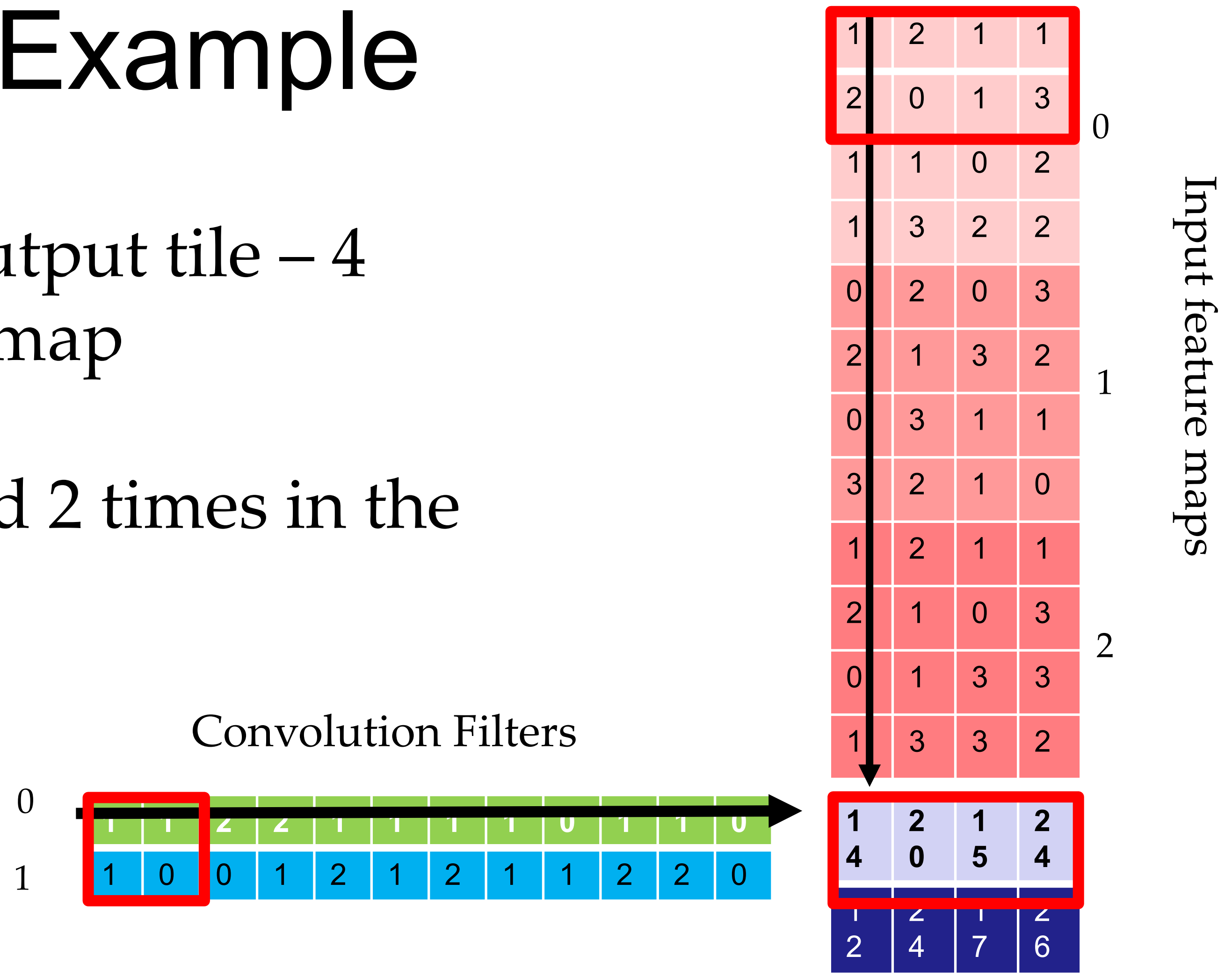
Each input element is reused 2 times in the shared memory



Tiled Matrix Multiplication 2x4 Example

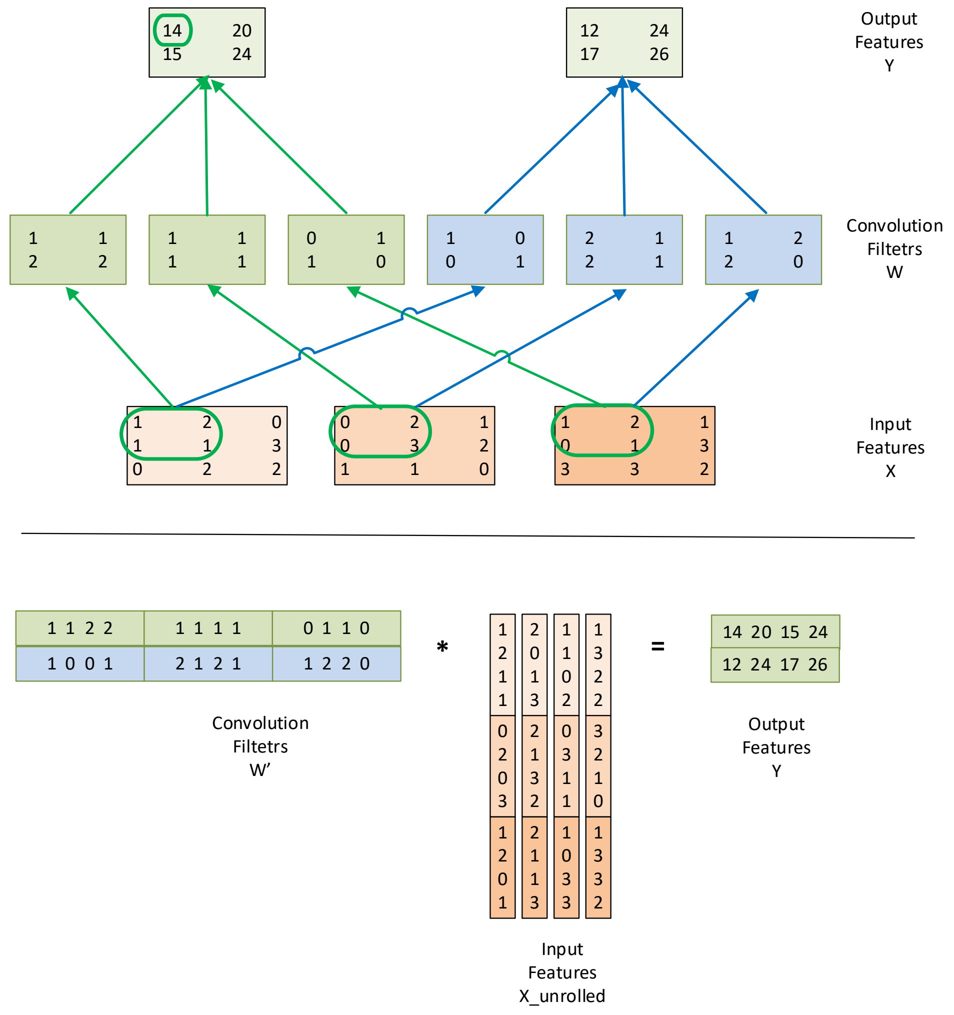
Each block calculates one output tile – 4 elements from each output map

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)



Analysis of a Small Example

H_out = 2

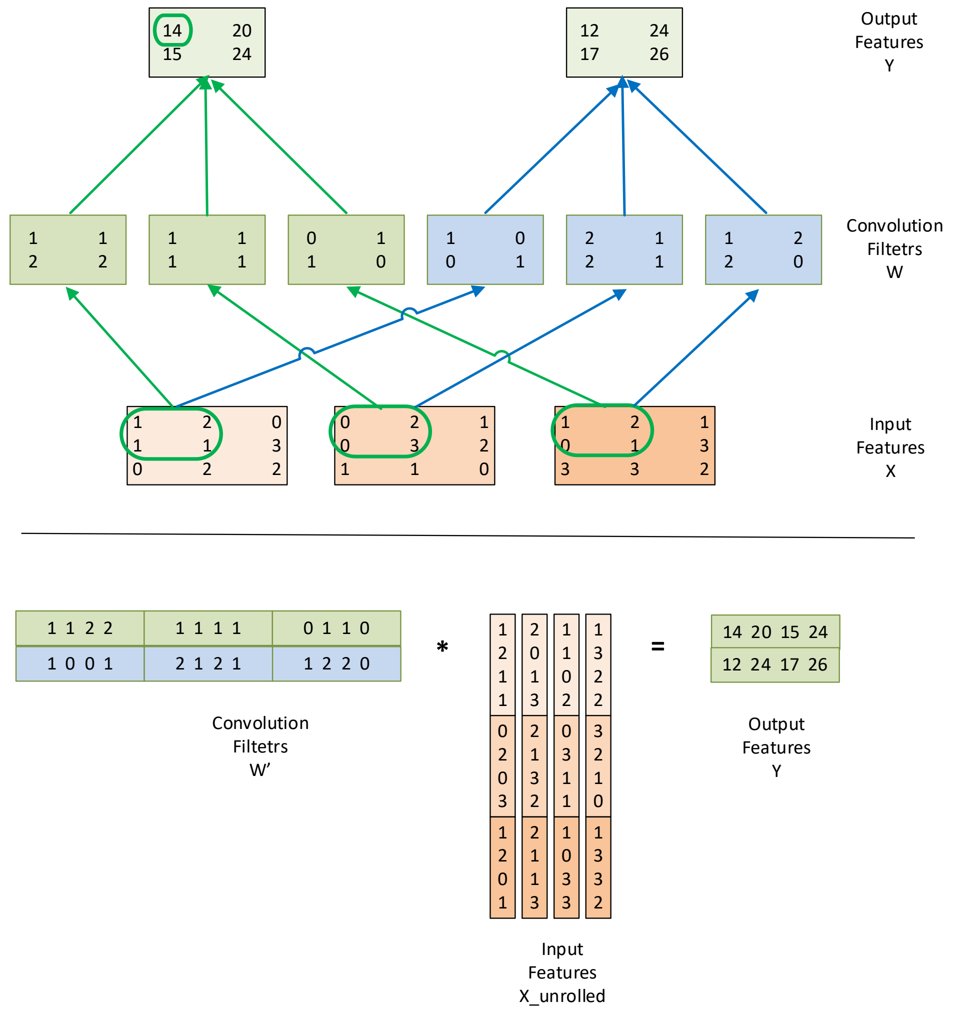
W_out = 2

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

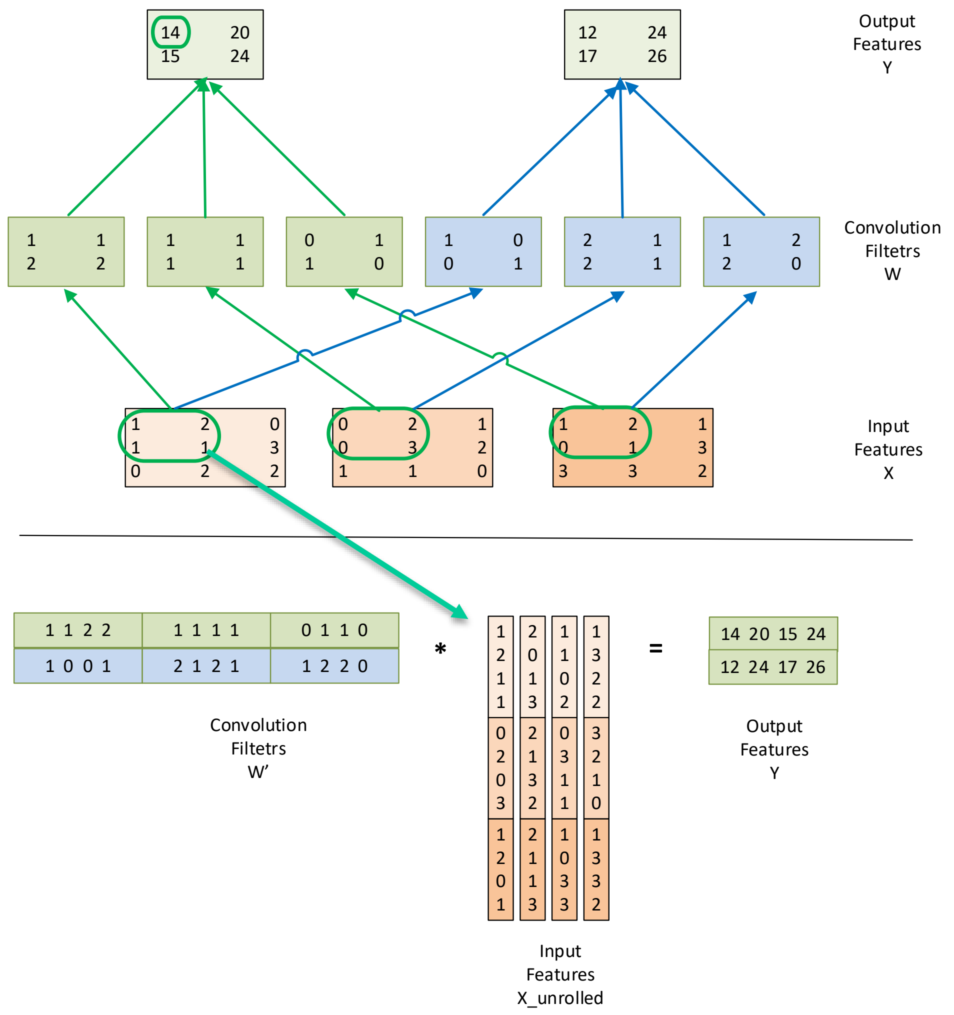


Memory Access Efficiency of Original Convolution Algorithm

- Assume that we use tiled 2D convolution
- For input elements
  - Each output tile has TILE_WIDTH^2 elements
  - Each input tile has $(\text{TILE\_WIDTH}+K-1)^2$
  - 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

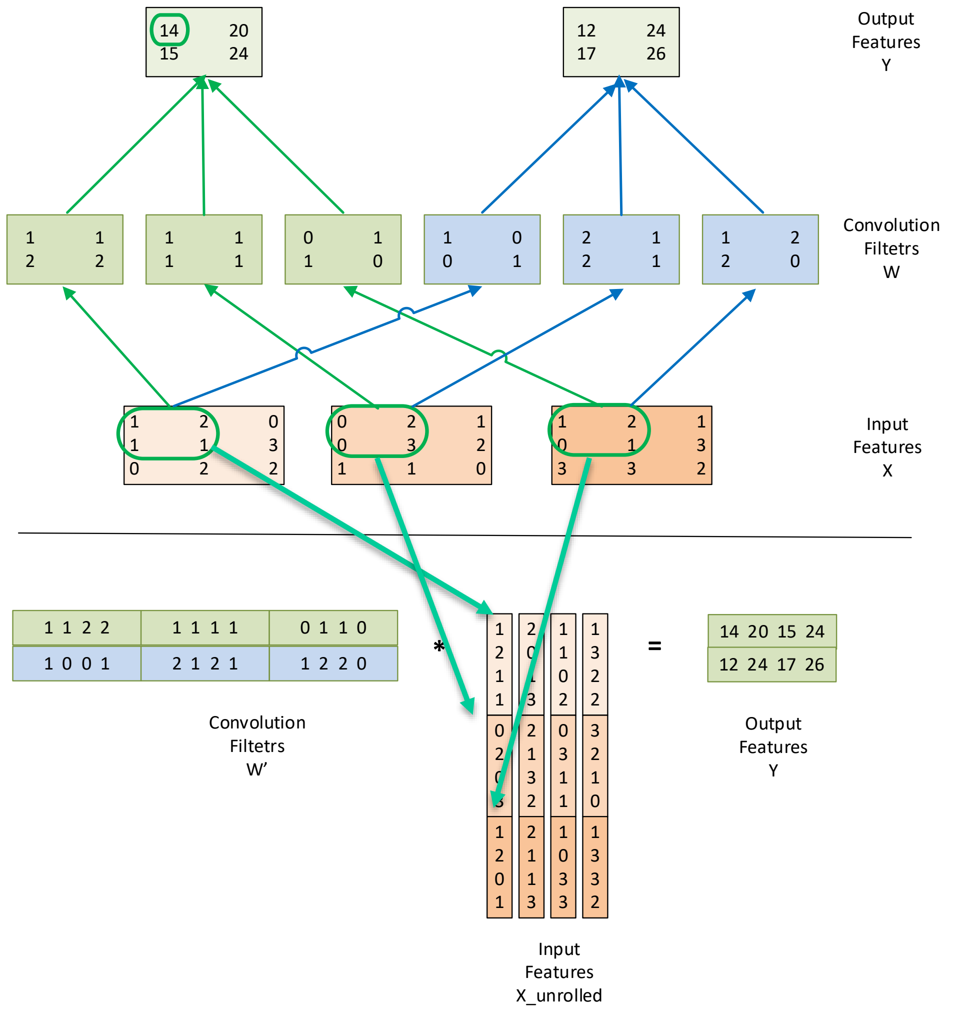
Properties of the Unrolled Matrix

- 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)



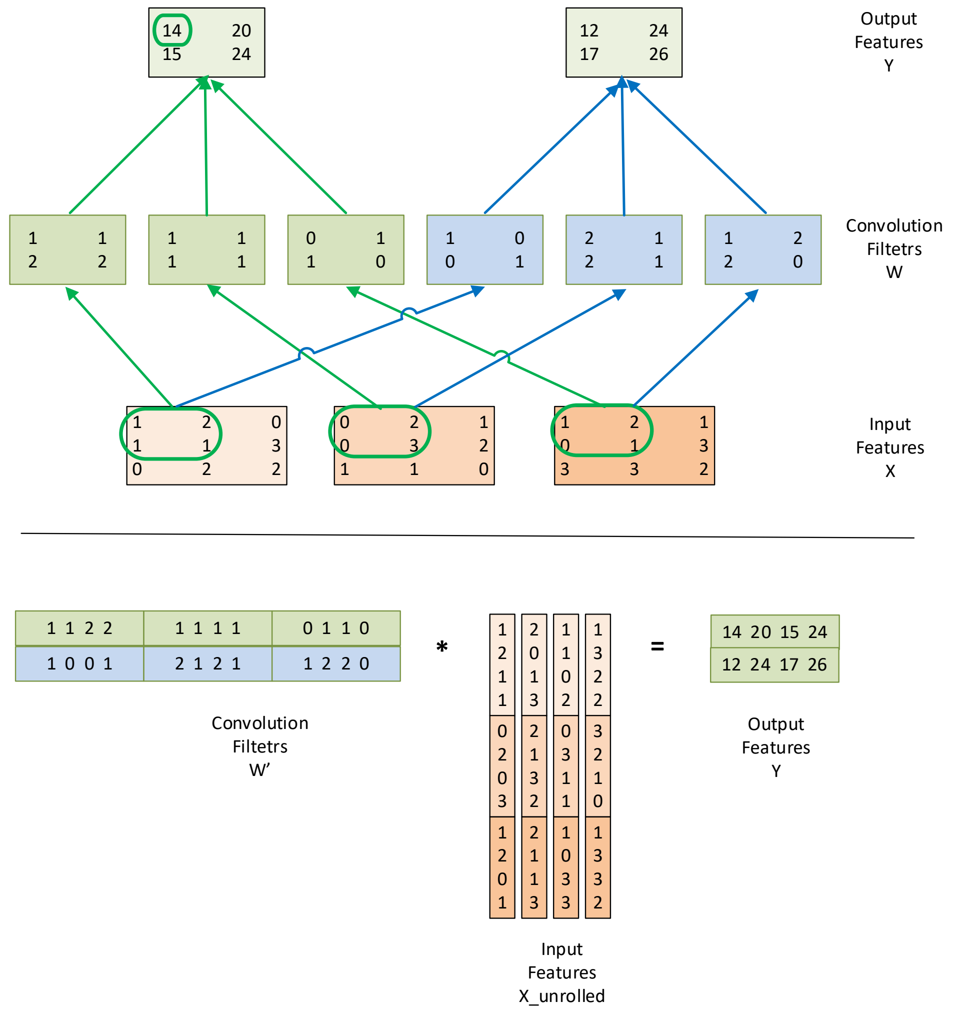
Properties of the Unrolled Matrix (cont.)

- 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

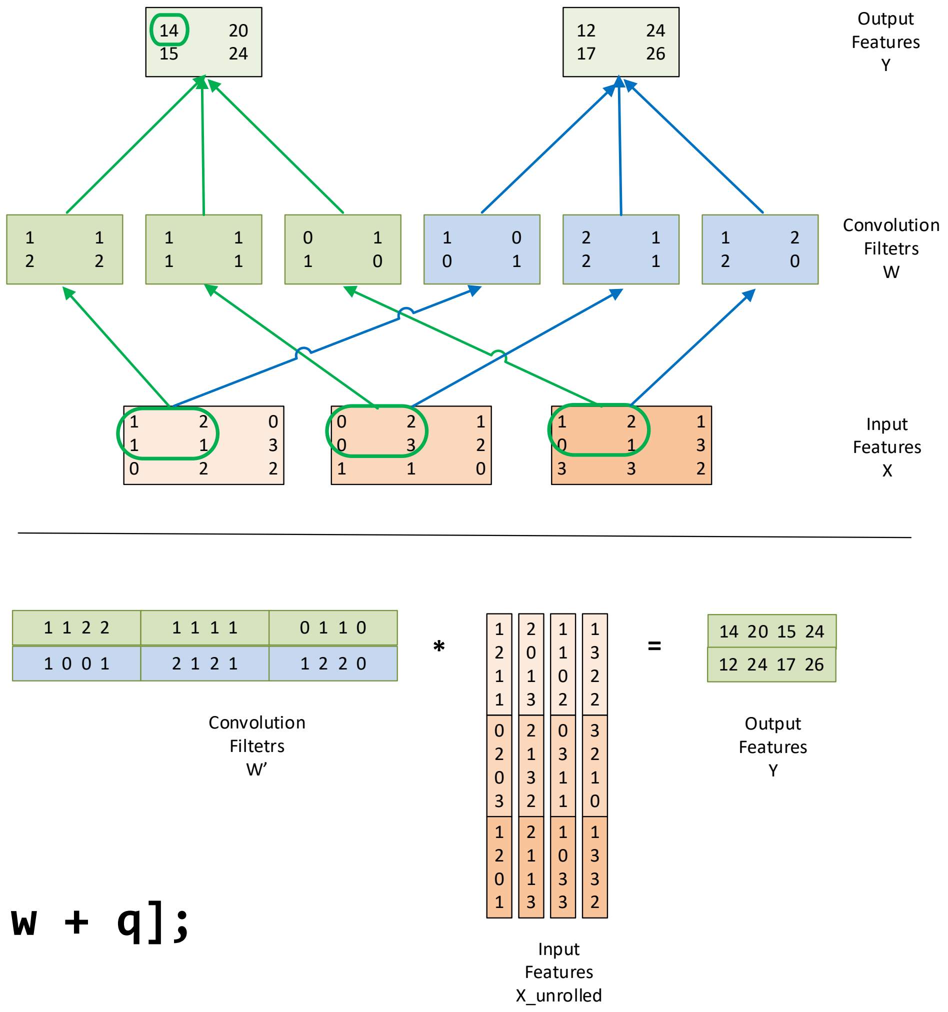
```
Output element (h, w)
Mask element (p, q)
Input feature map c

// 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

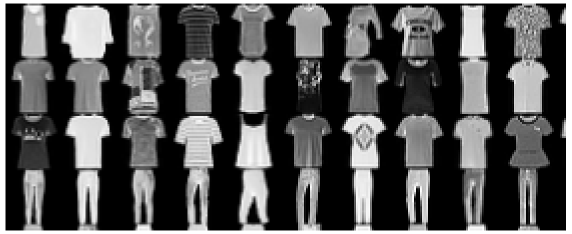
ANY MORE QUESTIONS?
READ CHAPTER 16

Project Overview

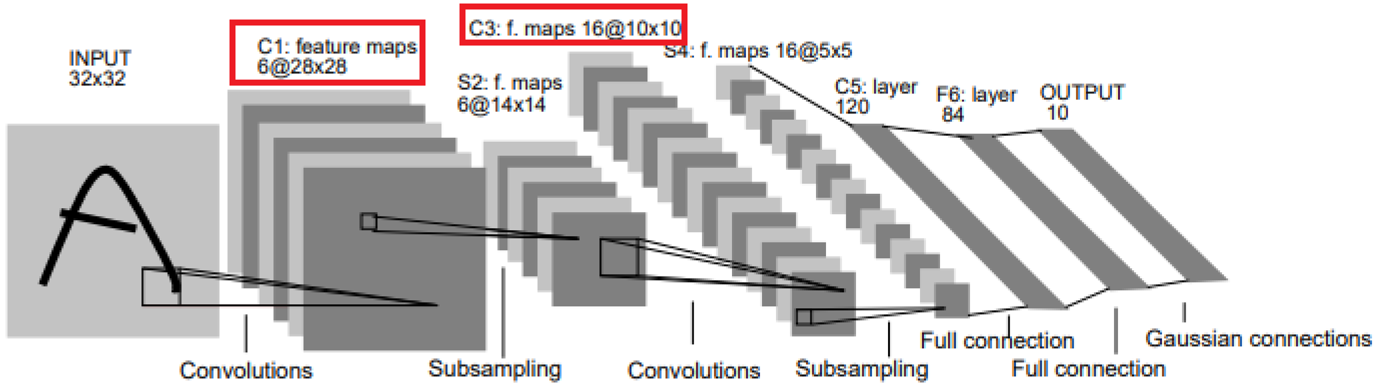
- Optimize the forward pass of the convolutional layers in a modified LeNet-5 CNN using CUDA. (CNN implemented using Mini-DNN, a C++ framework)

- The network will be classifying Fashion MNIST dataset

- Some network parameters to be aware of
  - Input Size: 86x86 pixels, batch of 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

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**

- October 15th: Project milestone 1:
  - Rai installation, CPU Convolution, profiling
- November 5th: Project milestone 2:
  - Baseline GPU Convolution Kernel
- December 3rd: Project milestone 3:
  - GPU Convolution Kernel Optimizations

Project Release

- Project is released now (only PM1 for now)
  - Check the course wiki page for the link to the github repository

- Project Landing Page:
  - https://wiki.illinois.edu/wiki/display/ECE408/Project

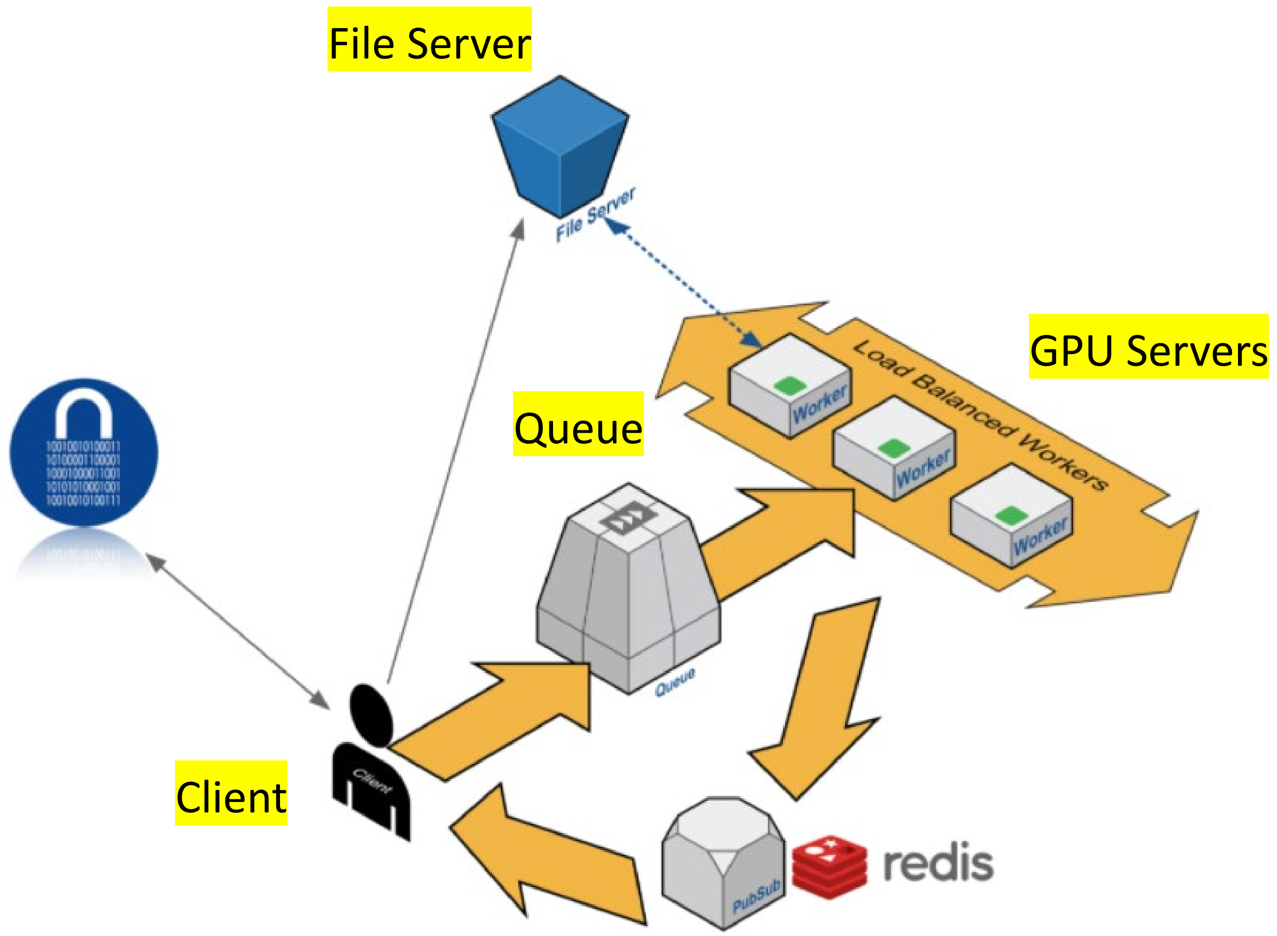
- Make sure that you clone the **2021fa** branch using git

```
git clone -b 2021fa https://github.com/illinois-impact/ece408_project.git
```

- The readme in the repository contains all the instructions and details to complete the project.

The RAI Submission System

- RAI is a scalable job submission system designed for CPU and GPU workloads



https://github.com/rai-project/raid

Setting up the RAI Client

- The link to download the client is in the project readme. Download the client version based on your OS.
- You will receive a `.rai_profile` file via email
  - Place this file in your home directory (i.e., `~/.rai_profile` on Linux/macOS)
  - This file will contain the credentials required for you to submit jobs via RAI

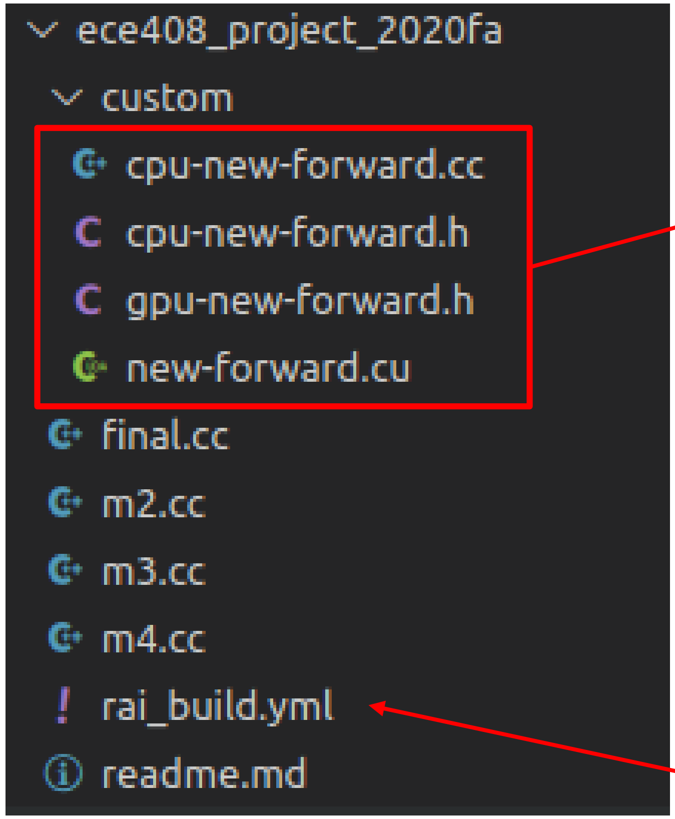
```
profile:
  id: 5f737b0788a5ec178e188f64
  firstname: James
  lastname: Cyriac
  username: jcyriac2
  email: jcyriac2@illinois.edu
  access_key: <snip>
  secret_key: <snip>
  affiliation: uiuc
```

Scheduling a job using RAI

```
rai -p <project folder> --queue rai_amd64_ece408
```

- Make sure the project folder points to the root of the Github repository
- Uploads your folder to the file server and queues your work request
- The code in `rai_build.yml` is executed on the server in the specified docker container
- The results are streamed back to you in real time

Mini-DNN Code Structure

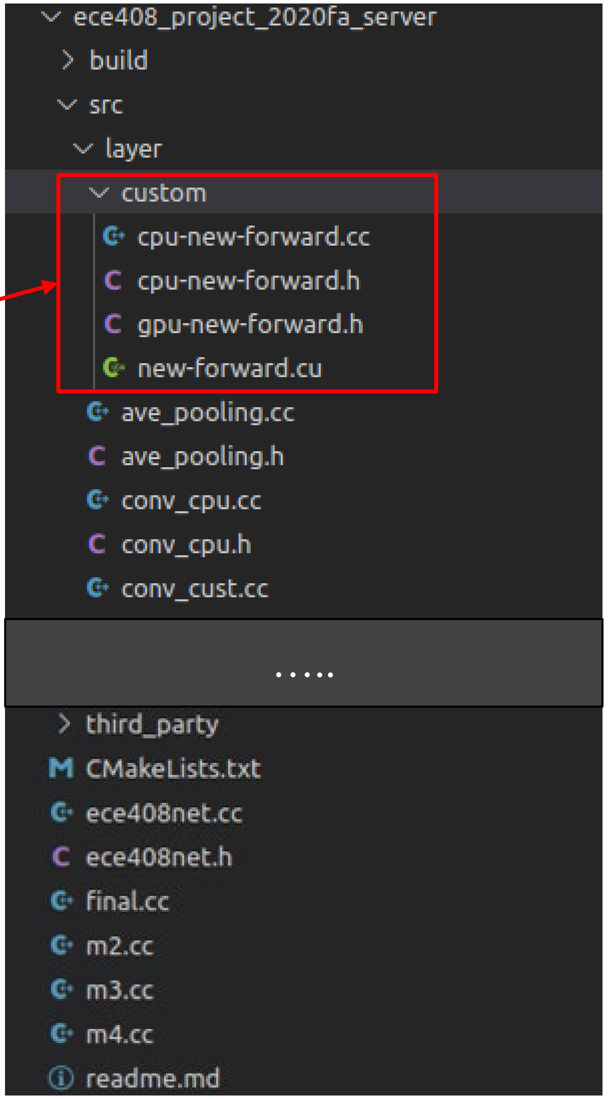


```
ece408_project_2020fa
  custom
    cpu-new-forward.cc
    cpu-new-forward.h
    gpu-new-forward.h
    new-forward.cu
  final.cc
  m2.cc
  m3.cc
  m4.cc
  rai_build.yml
  readme.md
```

Client Side

RAI replaces these 4 files on the server, recompiles the CNN and runs the inference

The commands RAI will run is specified in rai_build.yml. You will need to modify this file.



```
ece408_project_2020fa_server
  build
  src
    layer
      custom
        cpu-new-forward.cc
        cpu-new-forward.h
        gpu-new-forward.h
        new-forward.cu
      ave_pooling.cc
      ave_pooling.h
      conv_cpu.cc
      conv_cpu.h
      conv_cust.cc
      .....
  third_party
  CMakeLists.txt
  ece408net.cc
  ece408net.h
  final.cc
  m2.cc
  m3.cc
  m4.cc
  readme.md
```

Server Side

Submitting your code

To submit your code for grading, run

```
rai -p <project folder> --queue rai_amd64_ece408
--submit=[m1, m2, m3]
```

We will be using a specific version of `rai_build.yml` when grading. (We will not use the `rai_build.yml` in your project directory)

Make sure to include your `report.pdf` for the respective milestone in the project folder when you submit.

RAI tips and tricks

- `rai -p <project folder> --queue rai_amd64_ece408` **`history`**

  Shows your last 20 RAI runs and the associated file URL where your build directory is stored

- `rai -p <project folder> --queue rai_amd64_ece408` **`queued`**

  Shows the number of jobs in the queue and are waiting to start

- `rai -p <project folder> --queue rai_amd64_ece408` **`ranking`**

  Shows anonymized performance results comparing your team to other teams. Only applicable from Milestone 3

- `rai --help` – Lists all the other commands that can be run using RAI