中国科学院大学
University of Chinese Academy of Sciences

Introduction to CUDA

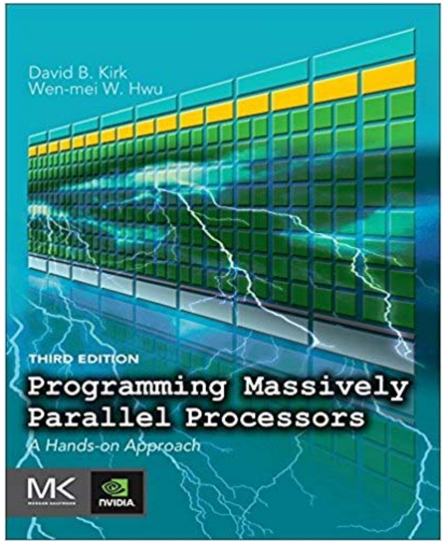
(7) Application: ML and CNN

Reference

- CUDA C Programming Guide,
  - https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

- **Programming Massively Parallel Processors,**
  - **A Hands-on Approach**
  - **Third Edition**

  - **Chapter 16**



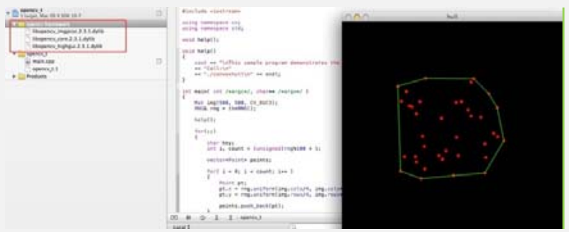
Content

- **Background**
- **Convolutional Neural Networks**
- **Convolutional Layer: A Basic CUDA Implementation of Forward Propagation**
- **Reduction of Convolutional Layer to Matrix Multiplication**

Machine Learning

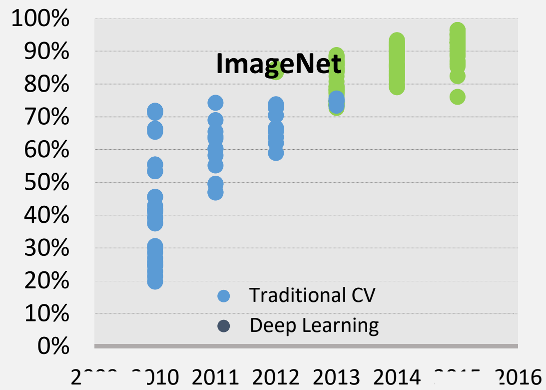
- An important way of building applications whose logic is not fully understood.
  - Use labeled data – data that come with the input values and their desired output values – to learn what the logic should be
  - Capture each labeled data item by adjusting the program logic
  - Learn by example!
- **Training Phase**
  - The system learns the logic for the application from labeled data.
- **Deployment (inference) Phase**
  - The system applies the learned program logic in processing data

Deep Learning in Computer Vision



Traditional Computer Vision
Experts + Time

Deep Learning Object Detection
DNN + Data + HPC

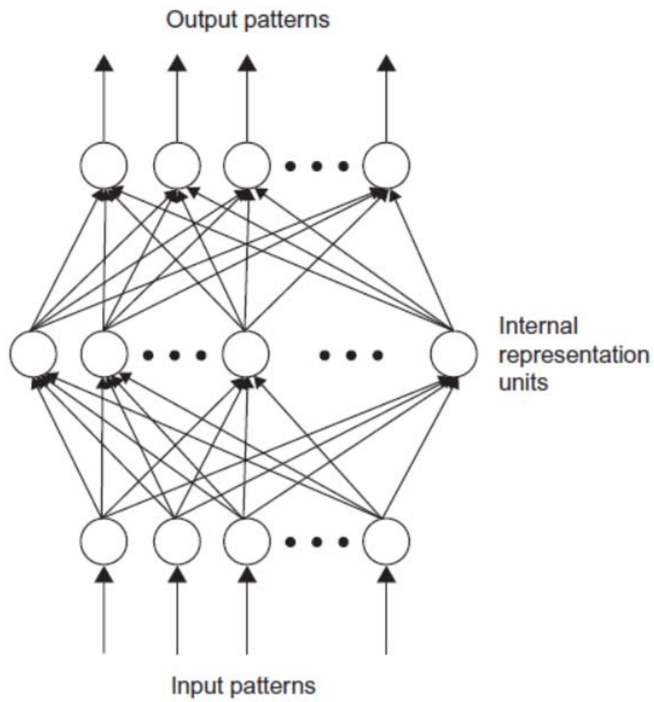


Deep Learning Achieves
“Superhuman” Results

ConvNet

- One type of deep learning procedure is based on **ConvNet :**
  - Easy to train
  - Better generalization

Multilayer
Feedforward
Network



ConvNet

- ConvNet was invented in late 1980s.

- By earlier 1990s, ConvNet has been successfully applied to:
  - Speech recognition
  - Optical character recognition
  - Handwriting recognition
  - Face recognition

- But data insufficient/computationally infeasible.

ConvNet

- Hinton, etc., 2006
  - Introduced unsupervised learning methods that could create multilayer, hierarchical feature detectors without requiring labeled data.
  - First use in speech recognition.

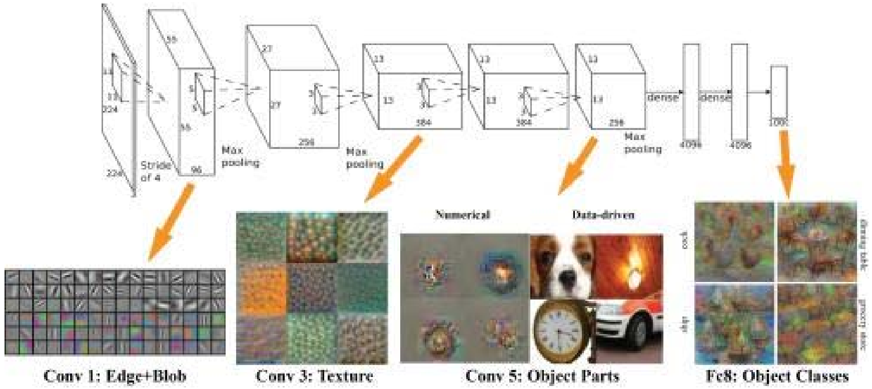
- But in computer vision ConvNets were largely ignored until 2012.
  - Driven by GPUs and massive online data.

Behind the Scenes

- In 2010 University of Toronto:
  - Programming Massively Parallel Programming
  - Prof. Andreas Moshovos
- Prof. Geoffrey Hinton’s students took the course.
  - Developed the GPU implementation of the DNN.
  - Trained 10 more times faster than CPU.

Deep ConvNet

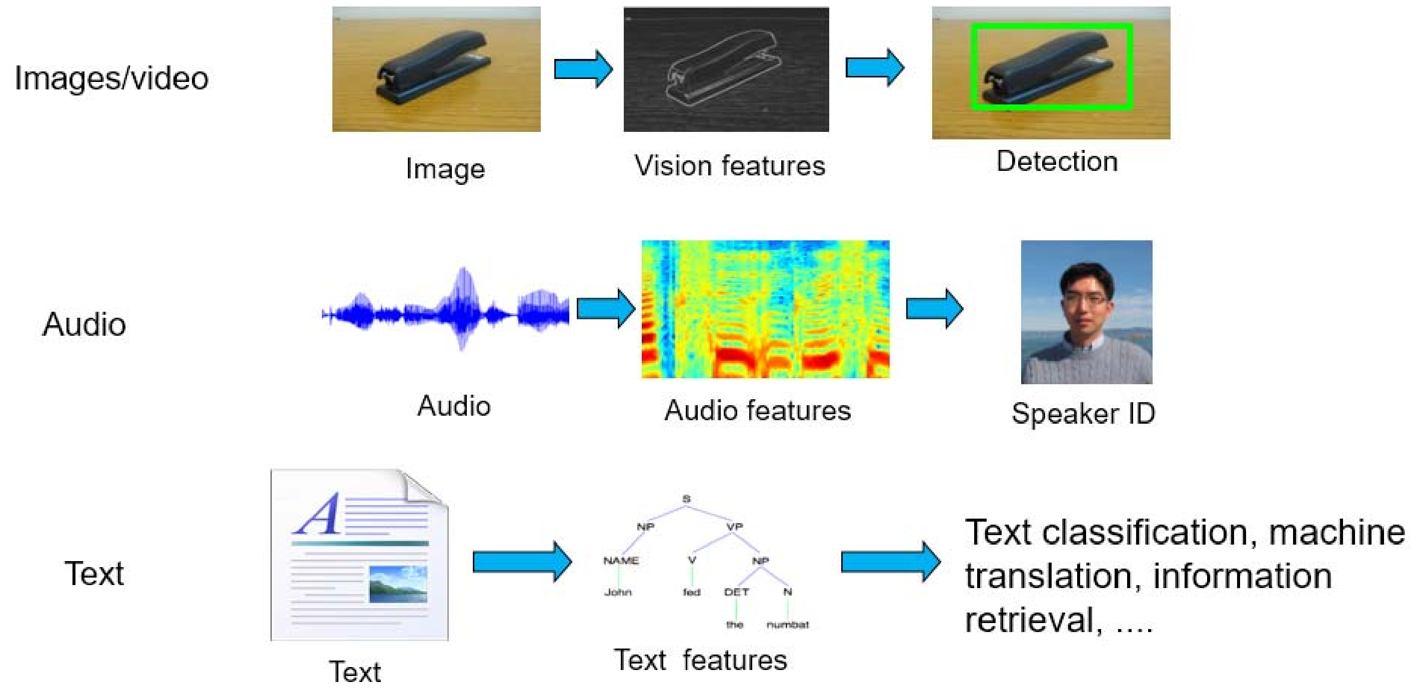
- Alex Krizhevsky, etc., 2012
    - AlexNet (extend the LeNet)
    - 60 million parameters
    - 650,000 neurons.



- Training :
    - Trained on 1.2 million images from ImageNet database.
    - One week on two GPUs. (GTX580, 3GB)
- Breakthrough results
    - Error of 15.3%
    - Second place 26.2%

Recent Explosion of DNN

- GPU with CUDA has enabled very fast research cycle of deep neural net training:

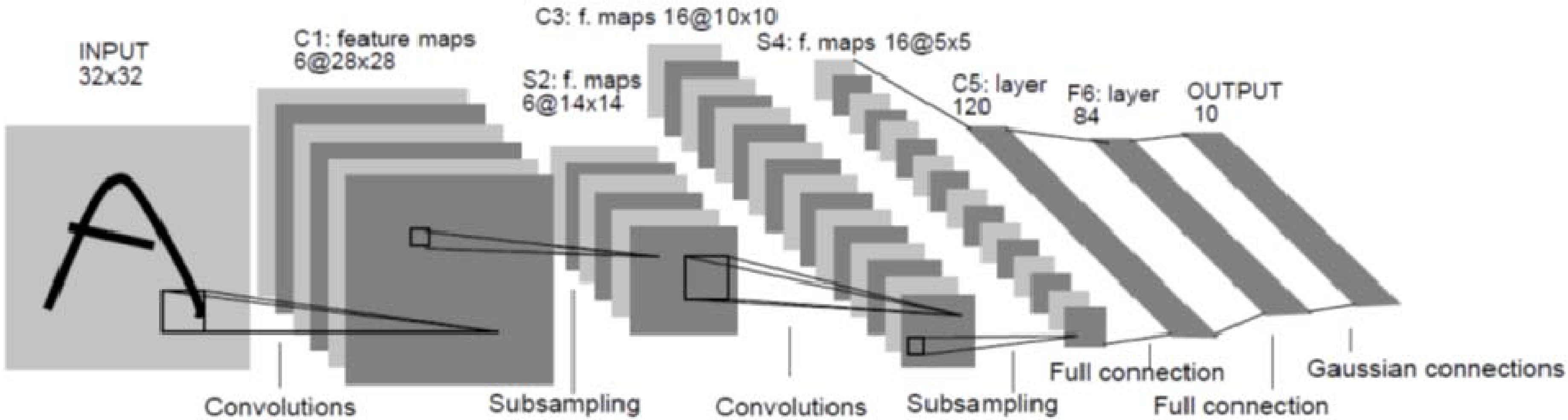


Content

- **Background**
- **Convolutional Neural Networks**
- **Convolutional Layer: A Basic CUDA Implementation of Forward Propagation**
- **Reduction of Convolutional Layer to Matrix Multiplication**

LeNet-5

- Yann LeCun, 1998, a convolutional neural network for hand-written digit recognition.

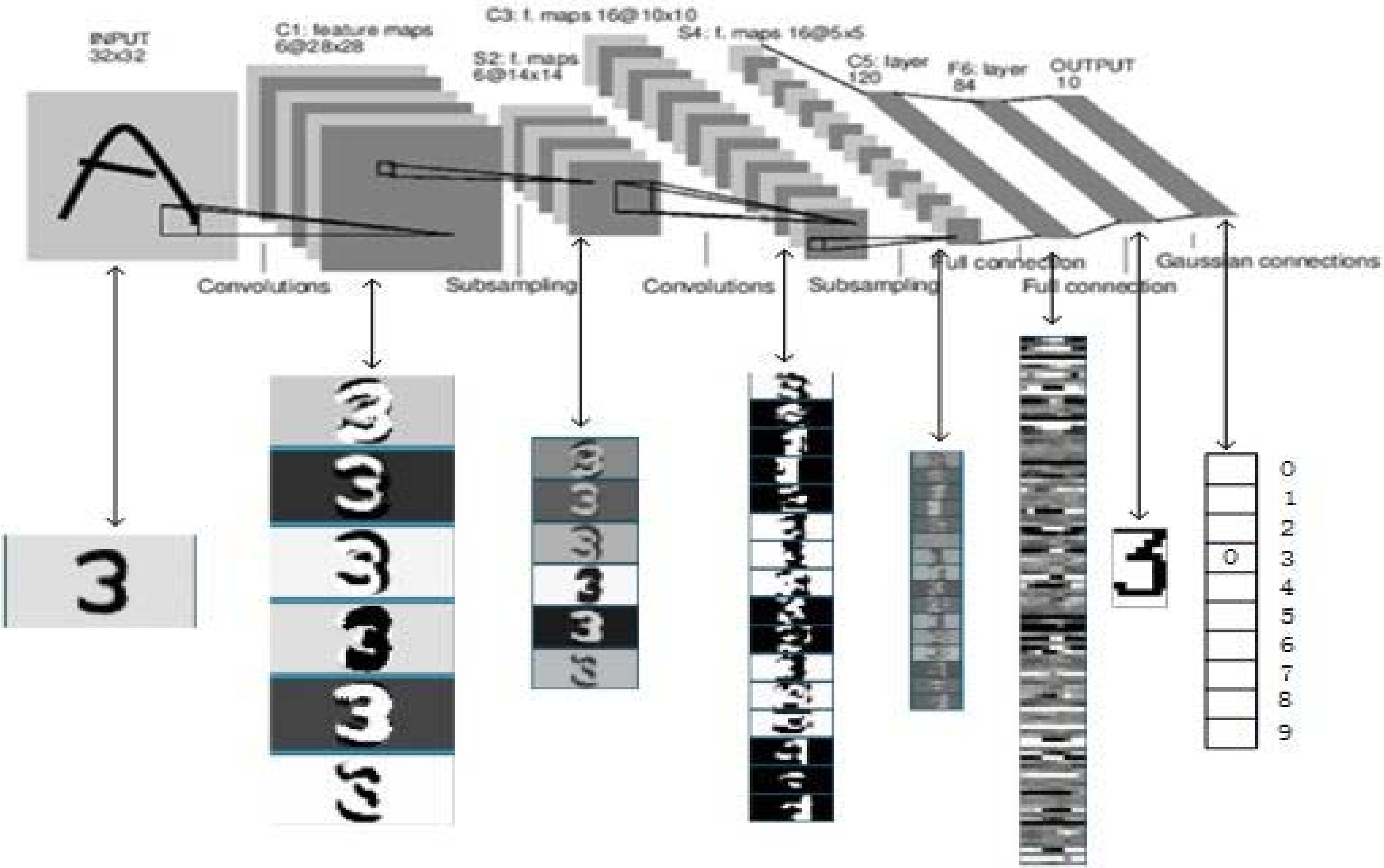


Three types of layers:

convolutional layers, subsampling layers, and full connection layers.

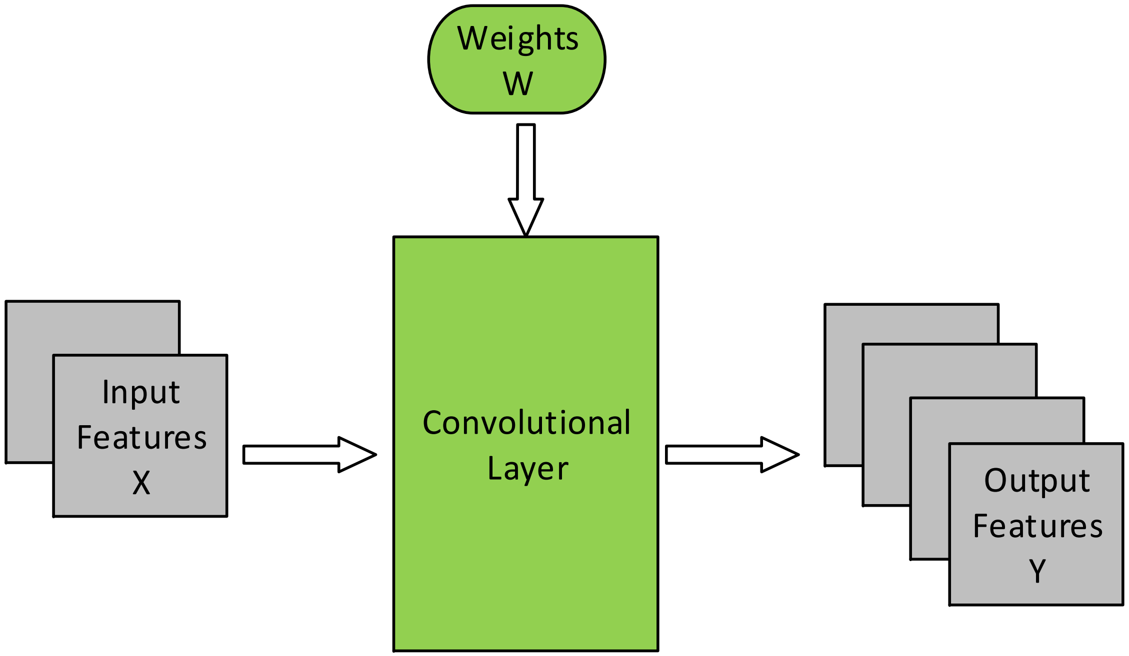
60,840 parameters and 340,908 connections

LeNet-5



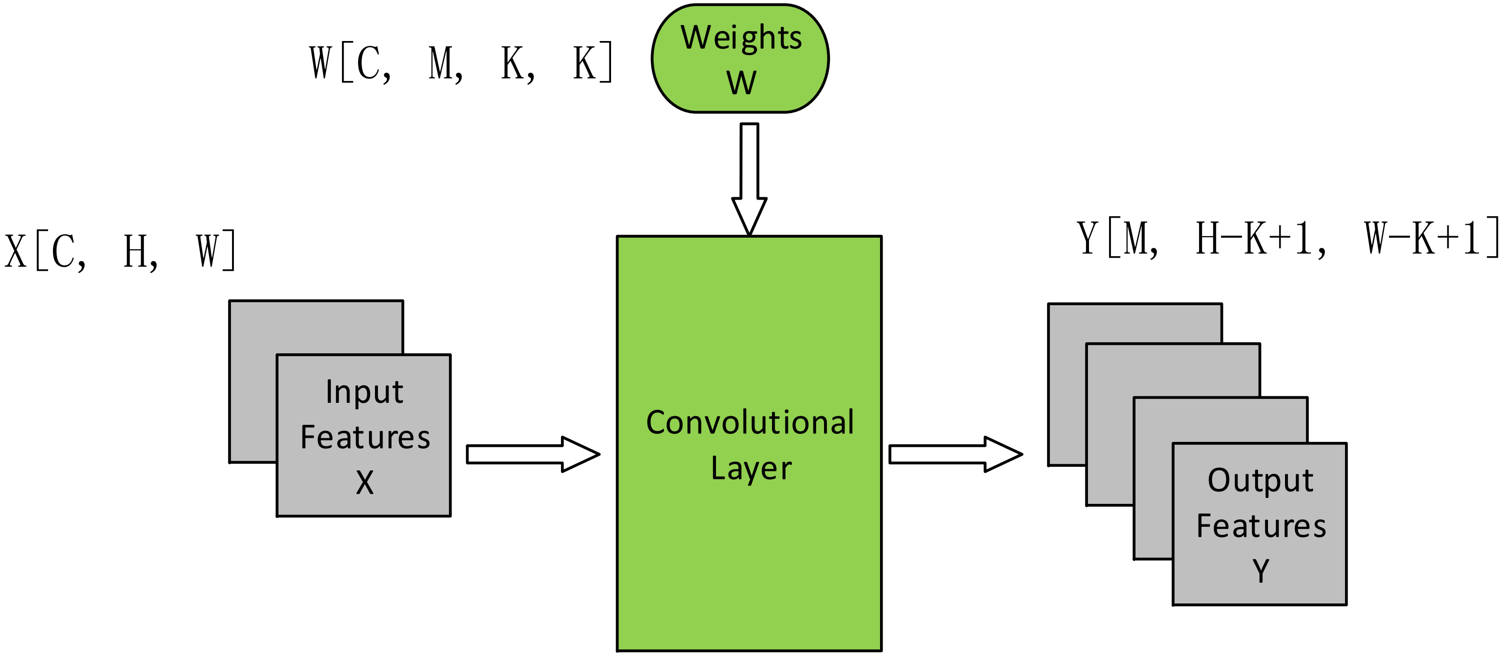
INPUT
32x32
C1: feature maps
6@28x28
S2: f. maps
6@14x14
C3: f. maps 16@10x10
S4: f. maps 16@5x5
C5: layer
120
F6: layer
84
OUTPUT
10
Convolutions
Subsampling
Convolutions
Subsampling
Full connection
Full connection
Gaussian connections
0
1
2
3
4
5
6
7
8
9

Forward: Convolution Layer



- All input feature maps contribute to all output feature maps.
- One convolution mask is provided for each input-output combination.

Forward: Convolution Layer



- LeNet C1
  - X[C, H, W] -- X[1, 32, 32]
  - Y[M, H-K+1, H-K+1] -- Y[6, 28, 28]
  - W[C, M, K, K] -- W[1, 6, 5, 5]

Sequential Code for the Forward Path of a Convolution Layer

```
void convLayer_forward(int M, int C, int H, int W, int K, float* X, float* W, float* Y)
{
  int H_out = H – K + 1;
  int W_out = W – K + 1;

  for(int m = 0;  m < M;  m++)                      // for each output feature map
    for(int h = 0; h < H_out; h++)                  // for each output element
      for(int w = 0; w < W_out; w++) {
        Y[m, h, w] = 0;
        for(int c = 0;  c < C; c++)                 // sum over all input feature maps
          for(int p = 0; p < K; p++)                // KxK  filter
            for(int q = 0; q < K; q++)
              Y[m, h, w] += X[c, h + p, w + q] * W[m, c, p, q];
      }
}
```

Subsampling Layer

- A subsampling layer reduces the size of image maps by combining pixels.

- LeNet S2:
  - Takes six input feature maps of size 28×28.
  - Generates six feature maps of size 14×14.
  - Each pixel in a subsampling feature map is the average from a **2×2** neighborhood.

  - A bias value b[m] that is specific to each output feature map is then added to each output feature map;
  - The sum goes through a nonlinear function such as the tanh, sigmoid, or ReLU functions.

Sequential code for the Forward Path of a Sub-sampling Layer

```
void poolingLayer_forward(int M, int H, int W, int K, float* Y, float* S)
{
  for(int m = 0;  m < M;  m++)                      // for each output feature maps
    for(int h = 0; h < H/K; h++)                    // for each output element
      for(int w = 0; w < W/K; w++) {
        S[m, x, y] = 0.;
        for(int p = 0; p < K; p++) {                 // loop over KxK input samples
          for(int q = 0; q < K; q++)
            S[m, h, w] += Y[m, K*h + p, K*w + q] /(K*K);
        }
        // add bias and apply non-linear activation
        S[m, h, w] = sigmoid(S[m, h, w] + b[m])
      }
}
```

K=2 for LeNet S2

LeNet layers

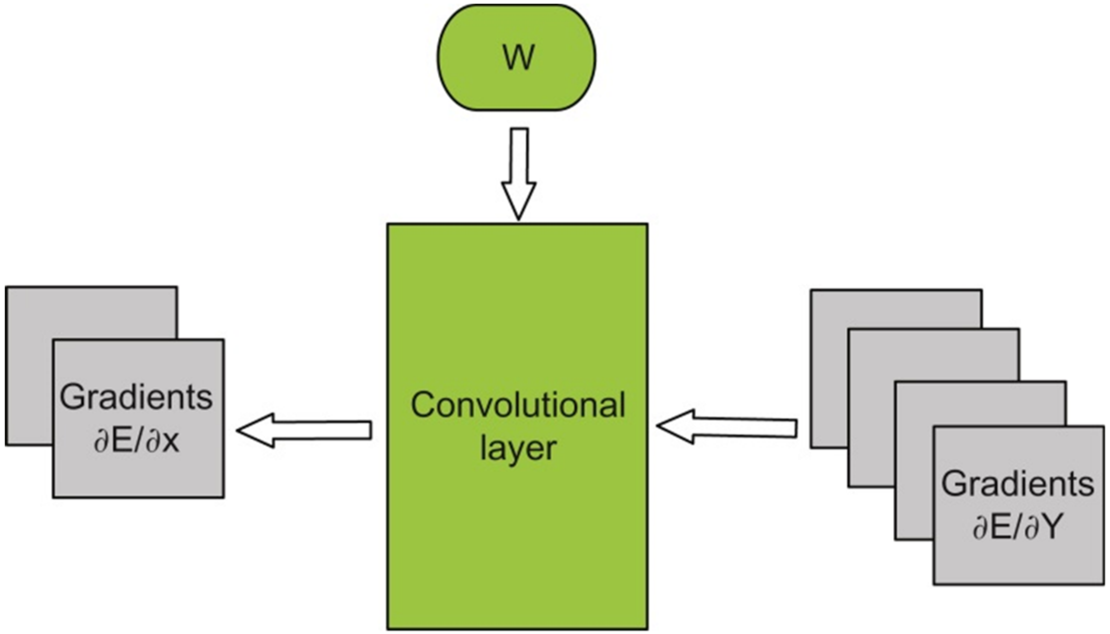
- C3:
  - 16 output feature maps (10×10 image for each)
  - 6×16 filter banks (5×5 weights for each)
- S4:
  - 16 output feature maps (5×5 image for each)
- C5:
  - 120 one-pixel output
  - $16 \times 120 = 1920$ filter banks (5×5 weights for each)
- F6:
  - 84 output units $Y6 = \text{sigmoid}(W*X + b)$
  - fully connected from C5
- Final Output
  - 10 elements generated from Gaussian filters
  - Compute loss function

Back-Propagation

- ConvNets training:
  - Labeled data or correct
  - Loss function or error

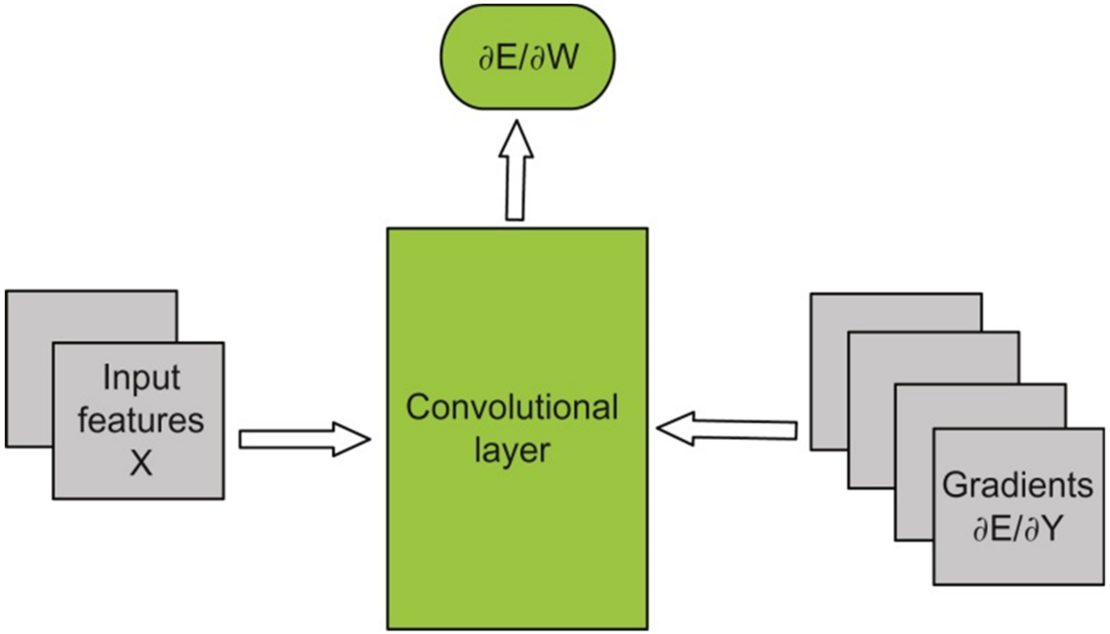
  - Through a procedure gradient backpropagation

Back-Propagation of ∂E/∂X



- Each layer receives as its input ∂E/∂Y
  - —gradient with respect to its output feature maps
- and calculates ∂E/∂X
  - — gradient with respect to its input feature maps

Back-Propagation of ∂E/∂W



- If a layer has learned parameters ("weights") W,
- then it also calculates ∂E/∂W
    - —gradient of loss with respect to weights

Back-Propagation

- For the fully connected layer is given as:

$$Y = W*X$$

- The backpropagation of gradient $\partial E/\partial Y$ is expressed by two equations:

$$\frac{\partial E}{\partial X} = W^T * \frac{\partial E}{\partial Y} \qquad \frac{\partial E}{\partial W} = \frac{\partial E}{\partial Y} * X^T$$

Calculation of ∂E/∂X

- The gradient ∂E/∂X：
  - with respect to the channel c of input X is given as
  - sum of "*backward convolution*" with corresponding $W^T(c,m)$ over all layer outputs *m*:

$$\frac{\partial E}{\partial X}(c,h,w)$$

$$= \sum_{m=1}^{M} \sum_{p=1}^{k} \sum_{q=1}^{k} \left(W(p,q) * \frac{\partial E}{\partial Y}(h-p, w-q) \right)$$

Calculation of ∂E/∂X

```
void convLayer_backward_xgrad(int M, int C, int H_in, int W_in, int K,
                                                          float* dE_dY, float* W, float* dE_dX) {
    int m, c, h, w, p, q;
    int H_out = H_in – K + 1;
    int W_out = W_in – K + 1;
    for(c = 0; c < C; c++)
        for(h = 0; h < H_in; h++)
            for(w = 0; w < W_in; w++)
                dE_dX[c, h, w] = 0.;
    for(m = 0; m < M; m++)
        for(h = 0; h < H_out; h++)
            for(w = 0; w < W_out; w++)
                for(c = 0; c < C; c++)
                    for(p = 0; p < K; p++)
                        for(q = 0; q < K; q++)
                            dE_dX[c, h + p, w + q] += dE_dY[m, h, w] * W[m, c, p, q];
}
```

assumes ∂E/∂Y has been calculated;
dE_dX has been allocated in device.

Calculation of ∂E/∂W

- The gradient ∂E/∂W：
  - Since each W(c,m) affects all elements of the output Y(m),
  - we accumulate gradients over all pixels in the corresponding output feature map:

$$\frac{\partial E}{\partial W}(c,m;p,q)$$

$$= \sum_{h=1}^{H_{out}} \sum_{w=1}^{W_{out}} \left(X(h+p, w+q) * \frac{\partial E}{\partial Y}(h,w) \right)$$

Calculation of ∂E/∂W

```
void convLayer_backward_wgrad(int M, int C, int H, int W, int K,
                              float* dE_dY, float* X, float* dE_dW) {
   int m, c, h, w, p, q;
   int H_out = H – K + 1;
   int W_out = W – K + 1;
   for(m = 0; m < M; m++)
      for(c = 0; c < C; c++)
         for(p = 0; p < K; p++)
            for(q = 0; q < K; q++)
               dE_dW[m, c, p, q] = 0.;
   for(m = 0; m < M; m++)
      for(h = 0; h < H_out; h++)
         for(w = 0; w < W_out; w++)
            for(c = 0; c < C; c++)
               for(p = 0; p < K; p++)
                  for(q = 0; q < K; q++)
                     dE_dW[m, c, p, q] += X[c, h + p, w + q] * dE_dY[m, c, h, w];
}
```

assumes ∂E/∂Y has been calculated;
dE_dW has been allocated in device.

$W(t+1) = W(t) - \lambda * \partial E/\partial W$

Stochastic Gradient Descent

- Training：
  - The training data sets are usually large.
- Instead of forward–backward for the whole training data set:
  - one randomly selects a small subset ("mini-batch") of N images;
  - computes the gradient only for this subset;
  - subsequently selects another subset and so on.
- This procedure adds one additional dimension ***n***
  - —the index of the sample in the mini-batch

Training with mini-batch

```
void convLayer_forward(int N, int M, int C, int H, int W, int K, float* X, float* W, float* Y)
{
  int n, m, c, h, w, p, q;
  int H_out = H– K + 1;
  int W_out = W – K + 1;
  for(n = 0;  n < N;  n++)                 // for each sample in the mini-batch
   for(m = 0;  m < M;  m++)                // for each output feature maps
    for(h = 0; h < H_out; h++)             // for each output element
     for(w = 0; w < W_out; w++) {
      Y[n, m, h, w] = 0;
      for (c = 0;  c < C; c++)             // sum over all input feature maps
       for (p = 0; p < K; p++)             // KxK  filter
        for (q = 0; q < K; q++)
          Y[n, m, h, w]  += X[n, c, h + p, w + q] * W[m, c, p, q];
     }
}
```

Content

- **Background**
- **Convolutional Neural Networks**
- **Convolutional Layer: A Basic CUDA Implementation of Forward Propagation**
- **Reduction of Convolutional Layer to Matrix Multiplication**

Parallel forward path

```
void convLayer_forward(int N, int M, int C, int H, int W, int K, float* X, float* W, float* Y)
{
  int n, m, c, h, w, p, q;
  int H_out = H - K + 1;
  int W_out = W - K + 1;
  parallel_for(n = 0;  n < N;  n++)
   parallel_for (m = 0;  m < M;  m++)
    parallel_for(h = 0; h < H_out; h++)
     parallel_for(w = 0; w < W_out; w++) {
      Y[n, m, h, w] = 0;
      for (c = 0;  c < C; c++)
       for (p = 0; p < K; p++)
        for (q = 0; q < K; q++)
         Y[n, m, h, w] += X[n, c, h + p, w + q] * W[m, c, p, q];
     }
}
```

N*M*H_out*W_out

Different parallel samples in a mini-batch, different output feature maps for the same sample, and different elements for each output feature map. In Parallel.

Parallel forward path

- Threads Organization:
  - Assume that each thread will compute one element of one output feature map.
  - use 2D thread blocks and each block for a tile of (TILE_WIDTH x TILE_WIDTH) elements.
  - e.g. TILE_WIDTH=16, then 256 threads per block.

- Blocks are organized into a 3D grid:
  - **1.** The first dimension (X) of the grid corresponds to samples (N) in the batch;
  - **2.** The second dimension (Y) corresponds to the (M) output features maps; and
  - **3.** The last dimension (Z) will define the location of the output tile inside the output feature map.

Parallel forward path

- Assume for simplicity that H_out (height of the output image) and W_out (width of the output image) are multiples of the tile width (set to 16 below):

```
# define TILE_WIDTH 16
// number of horizontal tiles per output map
W_grid = W_out/TILE_WIDTH;
// number of vertical tiles per output map
H_grid = H_out/TILE_WIDTH;
Z = H_grid * W_grid;

dim3 blockDim(TILE_WIDTH, TILE_WIDTH, 1);
dim3 gridDim(N, M, Z);
ConvLayerForward_Kernel<<< gridDim, blockDim>>>(…);
```

Parallel forward path -- Kernel

```
__global__ void
ConvLayerForward_Kernel(int C, int W_grid, intK, float* X, float* W, float* Y)
{
  int n, m, h, w, c, p, q;
  n = blockId.x;
  m = blockId.y;
  h = blockId.z / W_grid + threadId.y;
  w = blockId.z % W_grid + threadId.x;
  float acc = 0.;
  for (c = 0;  c < C; c++) {          // sum over all input channels
   for (p = 0; p < K; p++)            // loop over KxK  filter
    for (q = 0; q < K; q++)
     acc = acc + X[n, c, h + p, w + q] * W[m, c, p, q];
  }
  Y[n, m, h, w] = acc;
}
```

pseudo-code

? High degree of parallelism; but excessive global memory bandwidth.

Parallel forward -- improvement

- use shared memory tiling:
  - **1.** Load the filter W[m, c] into the shared memory.
  - **2.** All threads collaborate to copy the portion of the input X[n,c,.,.] that is required to compute the output tile into the shared memory array X_shared.
  - **3.** Compute for the partial sum of output Y_shared[n, m,.,.].
  - **4.** Move to the next input channel c.
- shared memory allocation:
  - input block X_tile_width * X_tile_width, where X_tile_width = TILE_WIDTH + K-1.
  - K*K filter coefficients.

Kernel using shared memory

```
__global__ void
ConvLayerForward_Kernel(int C, int W_grid, int K, float* X, float* W, float* Y)
{
  int n, m, h0, w0, h_base, w_base, h, w;
  int X_tile_width = TILE_WIDTH + K-1;
  extern __shared__ float shmem[];
  float* X_shared = &shmem[0];
  float* W_shared = &shmem[X_tile_width * X_tile_width];
  n = blockIdx.x;
  m = blockIdx.y;
  h0 = threadIdx.x;  // h0 and w0 used as shorthand for threadIdx.x and threadIdx.y
  w0 = threadIdx.y;
  h_base = (blockIdx.z / W_grid) * TILE_SIZE; // vertical base out data index for the block
  w_base = (blockIdx.z % W_grid) * TILE_SIZE; // horizontal base out data index for the block
  h = h_base+ h0;
  w = w_base+ w0;

  float acc = 0.;
  int c, i, j, p, q;
  for (c = 0; c < C; c++) {                          // sum over all input channels

    if (( h0 < K) && ( w0 < K))
     W_shared[h0, w0]= W [m, c, h0, w0];  // load weights for W [m, c,..],
    __syncthreads()                        // h0 and w0 used as shorthand for threadIdx.x
                                           // and threadIdx.y


    for (i = h; i < h_base+ X_tile_width; i += TILE_WIDTH) {
     for (j = w; j < w_base + X_tile_width; j += TILE_WIDTH)
      X_shared[i -h_base, j -w_base] = X[n, c, h, w]
    }                                      // load tile from X[n, c,…]into shared memory
    __syncthreads();
    for (p = 0; p < K; p++) {
     for (q = 0; q < K; q++)
      acc = acc + X_shared[h + p, w + q] * W_shared[p, q];
    }
    __syncthreads();
  }
  Y[n, m, h, w] = acc;
}
```

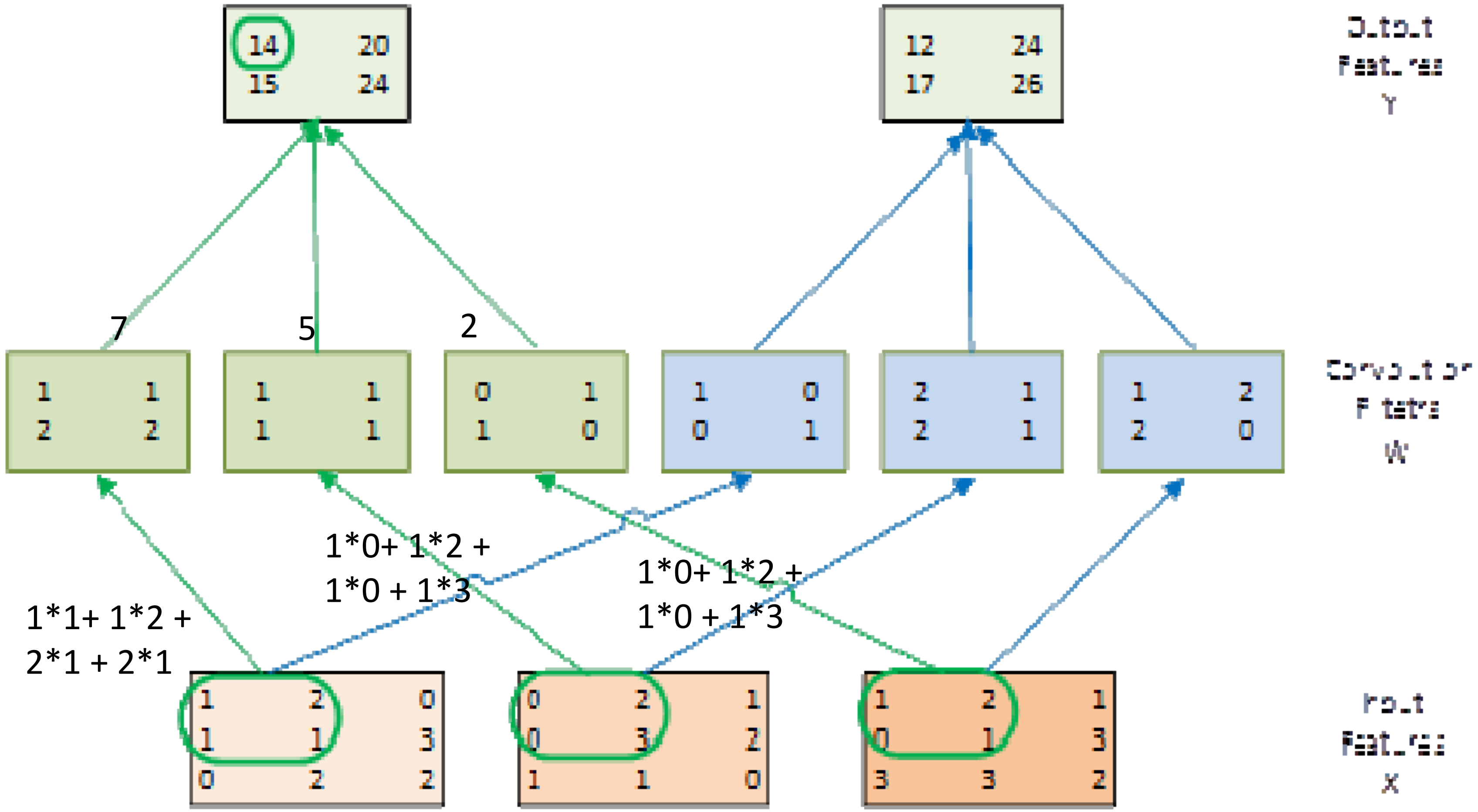
Content

- **Background**
- **Convolutional Neural Networks**
- **Convolutional Layer: A Basic CUDA Implementation of Forward Propagation**
- **Reduction of Convolutional Layer to Matrix Multiplication**

GEMM

- Build an even faster convolutional layer:
  - Reducing to highly efficient matrix multiplication;
  - Using **GE**neral **M**atrix to Matrix **M**ultiplication (GEMM), from CUDA linear algebra library (cuBLAS).
  - Introduced in 2006 by CPS.
- Main idea:
  - unfolding and replicating the inputs to the convolutional kernel such that all elements needed to compute one output element will be stored as one sequential block.
  - reduce the forward operation of the convolutional layer to one large matrix–matrix multiplication.
  - https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/

Example of the Forward Path of a Convolution Layer



GEMM-example

- Rearrange all input elements :
  - Since the results of the convolutions are summed across input features, the input features can be concatenated into one large matrix.
  - Each row of this matrix contains all input values necessary to compute one element of an output feature.

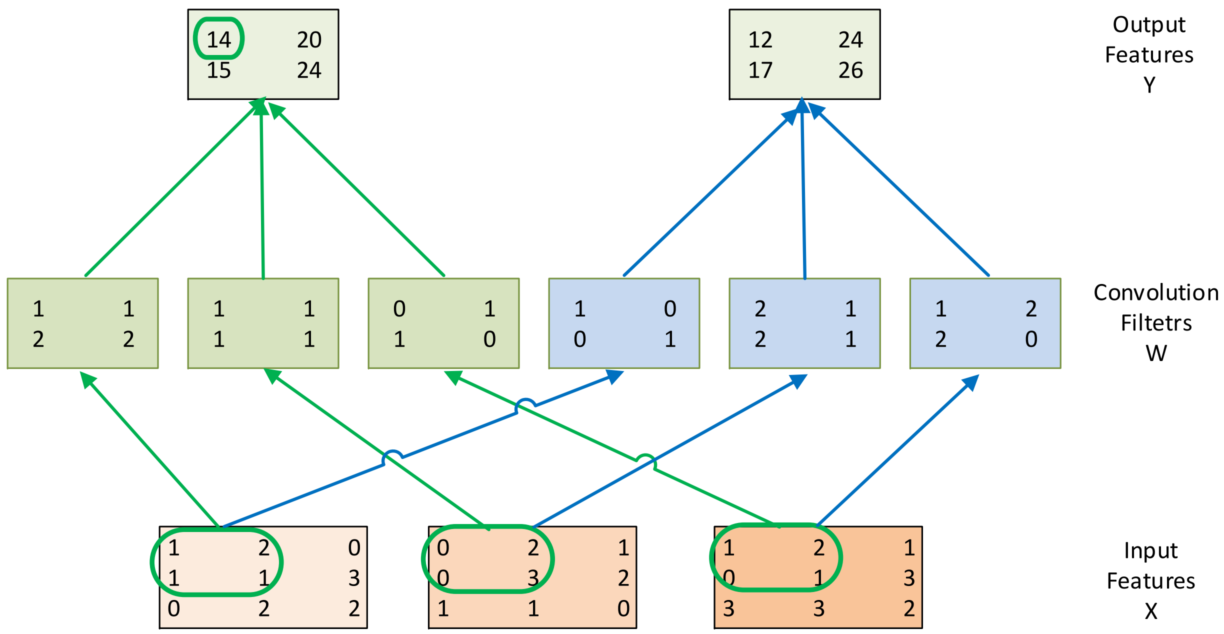
- This process means that each input element will be replicated multiple times.
  - In example 4*1 + 2*4 + 1*4 = 16.

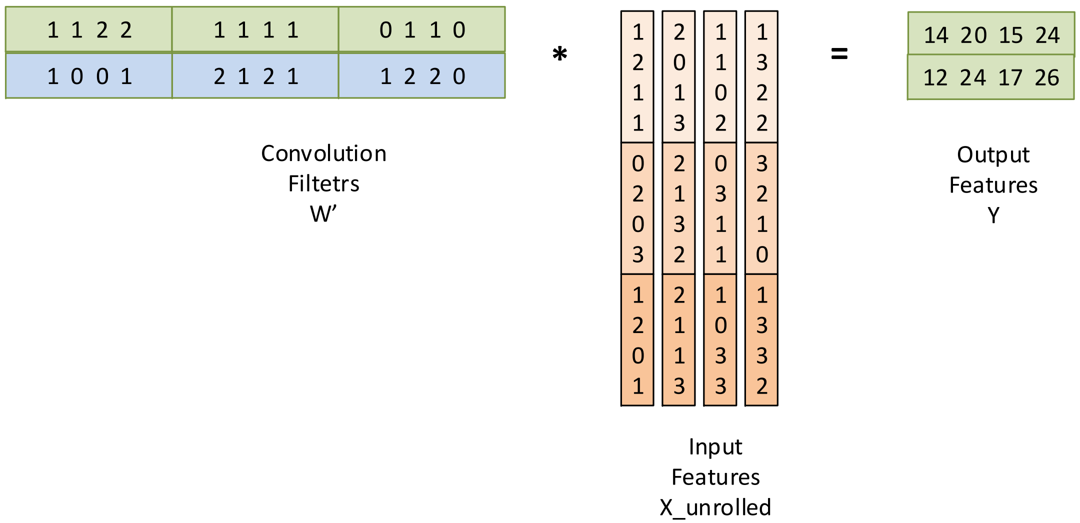
Size of the unrolled <u>input</u> matrix

- The height:
  - Is the number of input feature elements contributing to each output feature map element.
  - The number is C*K*K
- The width:
  - Is the number of elements in each output feature map.
  - The number is H_out*W_out
- Expansion ratio:
  - (K*K*H_out*W_out)/(H_in*W_in) -- > K*K

Size of the unrolled filter-bank matrix

- The height:
  - is the number of output feature maps (M).
- The width:
  - Is the number of weight values needed to generate each output feature map element.
  - The number is C*K*K
- Expansion ratio:
  - no duplication occurs


14 20
15 24
12 24
17 26
Output Features Y
1 1
2 2
1 1
1 1
0 1
1 0
1 0
0 1
2 1
2 1
1 2
2 0
Convolution Filtetrs W
1 2 0
1 1 3
0 2 2
0 2 1
0 3 2
1 1 0
1 2 1
0 1 3
3 3 2
Input Features X


1 1 2 2
1 1 1 1
0 1 1 0
1 0 0 1
2 1 2 1
1 2 2 0
*
1 2 1 1
0 2 0 3
1 2 0 1
2 0 1 3
2 1 3 2
2 1 1 3
1 1 0 2
0 3 1 1
1 0 3 3
1 3 2 2
3 2 1 0
1 3 3 2
=
14 20 15 24
12 24 17 26
Convolution Filtetrs W'
Input Features X_unrolled
Output Features Y

CUDA Implementation

```
void convLayer_forward(int N, int M, int C, int H, int W, int K, float* X, float* W_unroll, float* Y)
{
  int W_out = W– K + 1;
  int H_out = H– K + 1;
  int W_unroll = C * K * K;
  int H_unroll = H_out * W_out;
  float* X_unrolled = malloc(W_unroll * H_unroll * sizeof(float));
  for (int n=0; n < N; n++) {
    unroll(C, H, W, K, n,X, X_unrolled);
    gemm(H_unroll, M, W_unroll, X_unrolled, W, Y[n]);
  }
}
```

the sequential implementation of the forward path of a convolutional layer with matrix multiplication.

CUDA Implementation

```
void unroll(int C, int H, int W, int K, float* X, float* X_unroll)
{
   int c, h, w, p, q, w_base, w_unroll, h_unroll;
   int H_out = H– K + 1;
   int W_out = W– K + 1;
   for(c = 0; c < C; c++) {
    w_base = c * (K*K);
    for(p = 0; p < K; p++)
     for(q = 0; q < K; q++) {
      for(h = 0; h <  H_out; h++)
       for(w = 0; w < W_out; w++){
         w_unroll = w_base + p * K + q;
         h_unroll = h * W_out + w;
         X_unroll(h_unroll, w_unroll) = X(c, h + p, w + q);
       }
    }
}
```

sequential function that produces the X_unroll array.

CUDA Implementation

```
void unroll_gpu(int C, int H, int W, int K, float* X, float* X_unroll)
{
  int H_out = H – K + 1;
  int W_out = W– K + 1;
  int num_threads = C * H_out * W_out;
  int num_blocks = ceil((C * H_out * W_out) / CUDA MAX_NUM_THREADS);
  unroll_Kernel<<<num_blocks, CUDA MAX_NUM_THREADS>>>();
}
```

Host code for invoking the unroll kernel.

CUDA Implementation

```
__global__ void unroll_Kernel(int C, int H, int W, int K, float* X, float* X_unroll)
{
   int c, s, h_out, w-out, h_unroll, w_base, p, q;
   int t = blockId.x * CUDA MAX_NUM_THREADS + threadId.x;
   int H_out = H– K + 1;
   int W_out = W– K + 1;
   int W_unroll = H_out * W_out;

   if (t < C * W_unroll) {
     c = t / W_unroll;
     s = t % W_unroll;
     h_out = s / W_out;
     w_out = s % W_out;
     h_unroll = h_out * W_out + w_out;
     w_base = c * K * K;
     for(p = 0; p < K; p++)
       for(q = 0; q < K; q++) {
         w_unroll = w_base + p * K + q;
         X_unroll(h_unroll, w_unroll) = X(c, h_out + p, w_out + q);
       }
   }
}
```

Some Observations

- The amount of parallelism is quite high as long as the total number of pixels across all output feature maps is large
  - This matches the CNN architecture well
  - C*H_out*W_out is usually fairly large for all layers

- Each input tile is loaded multiple times, once for each block that calculates the output tile that requires the input tile
  - Not very efficient in global memory bandwith

CuDNN

- C-language deep learning API for implementing deep learning primitives routines:
  - D is a four-dimensional N x C x H x W tensor which forms the input data;
  - F is a four-dimensional K x C x R x S tensor, which forms the convolutional filters;
- cuDNN supports multiple algorithms:
  - matrix multiplication-based(GEMM & Winograd)
  - fast-Fouriertransform-based