UNIVERSITÀ
DEGLI STUDI
FIRENZE

# Parallel Programming

Prof. Marco Bertini

UNIVERSITÀ
DEGLI STUDI
FIRENZE

# Data parallelism: GPU computing

UNIVERSITÀ
DEGLI STUDI
FIRENZE

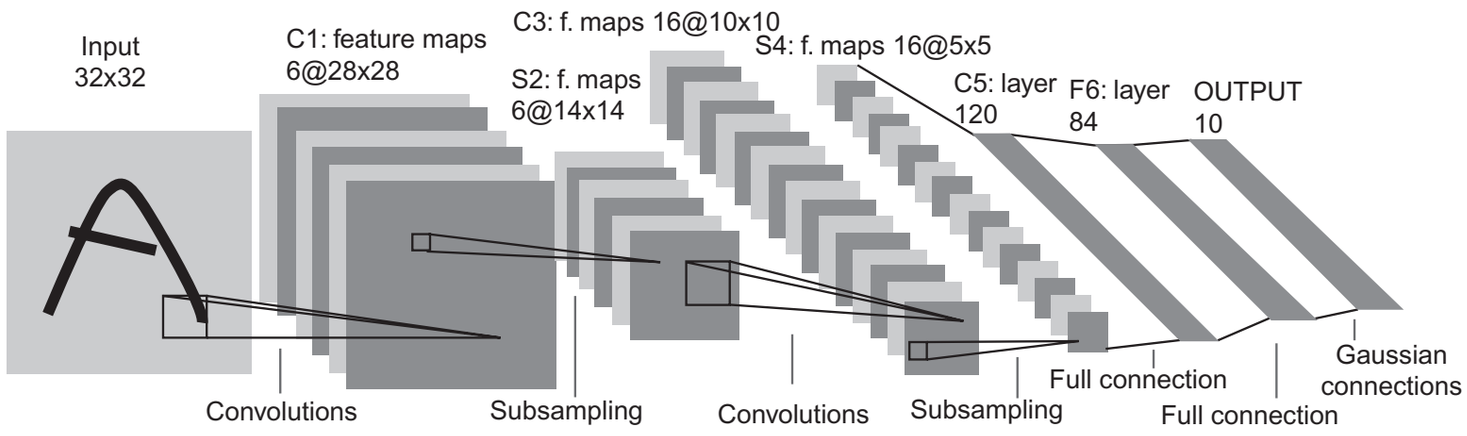
# CUDA and Convolutional Neural Networks

# ConvNets

- Since the victory of AlexNet in the ImageNet competition, convolutional neural networks (ConvNets) have become a mainstream tool in computer vision, natural language processing, reinforcement learning, and many other traditional machine learning areas.

- ConvNets are characterized by high compute-to-bandwidth ratio and high levels of parallelism, which are perfect attributes for GPU acceleration

# CNN layers



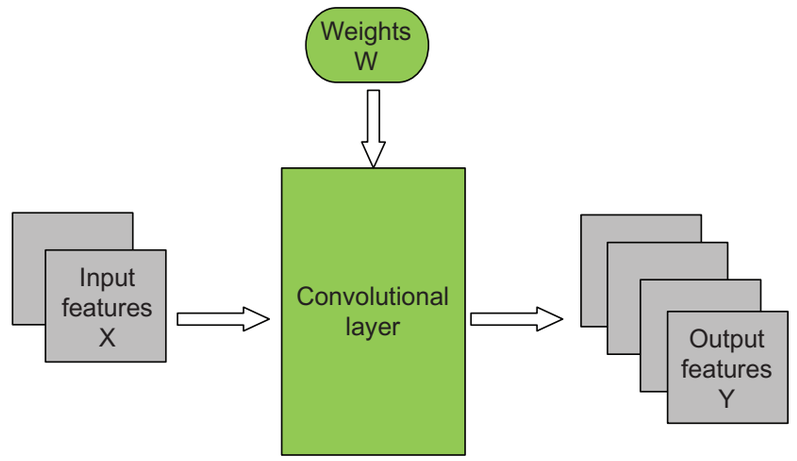

- For simplicity consider LeNet-5 CNN, composed of convolutional layers, subsampling layers, and full connection layers.

- Inputs and outputs to layers will be referred to as “feature maps”, composed of pixels, each of which computed using a convolution (each conv. mask is a filter - 5×5 and no ghost cells — i.e. “valid” — in LeNet-5).

- Different feature maps in a layer use different filter banks.

- If a convolution layers has *n* input feature maps and *m* output feature maps, *n*×*m* different filter banks will be used.

# Conv layer




- Let's consider input feature maps are stored as 3D array `X[C, H, W]`, where `C` is the # maps

- Output feature maps are stored as 3D array `Y[M, H0, W0]` where output width and height depend on the convolution (if uses ghost cells they are the same otherwise they become `H-K+1`, `W-K+1` where `K` is convolution mask size, where `M` is # maps

- Filter banks are stored as `W[C, M, K, K]`

- Computation is a set of `M` 3D convolutions

- Y = W * X

# Conv layer

- A sequential implementation consists of a series of nested loops:
  - Each iteration of the outermost (`m`) for-loop generates an output feature map using…
    - the next two levels (`height`, `width`) of for-loops that generate individual pixels of the current output map.
      - The next loop over the (`c`) input features sums…
        - the two innermost for-loops sum up the pixels in the neighborhood using the convolution mask
- A bias value `b[m]` that is specific to each output feature map is then added to each output feature map, and the sum goes through a nonlinear function such as the tanh, sigmoid, or ReLU functions.

# Pooling layer

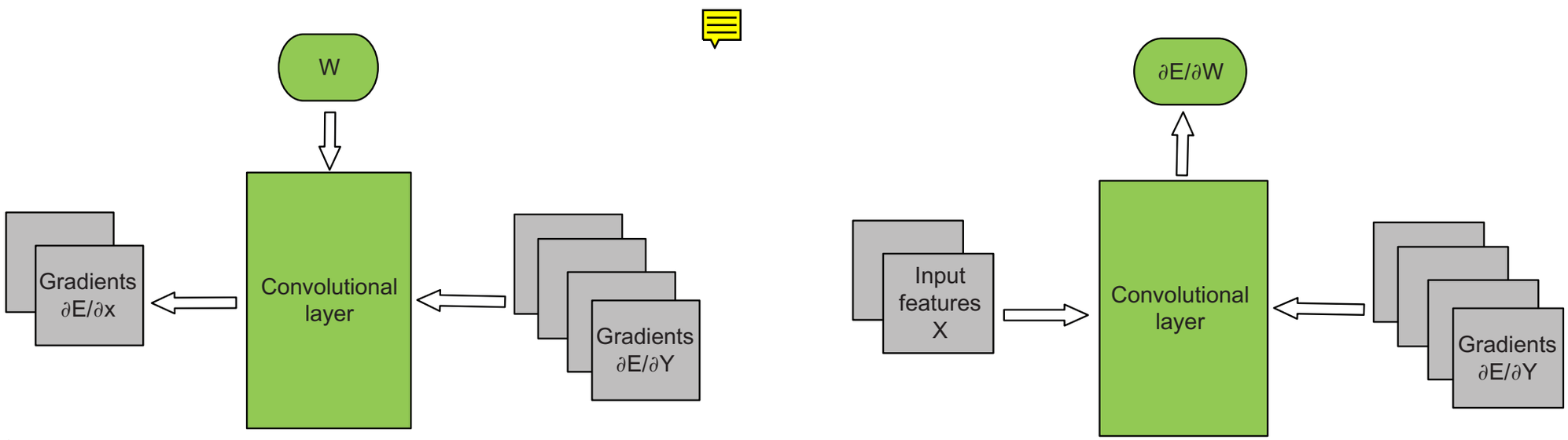
- It is a subsampling layer reduces the size of image maps by combining pixels (e.g. avg. or max).
- The output of a subsampling layer has the same number of output feature maps as the previous layer, but maps are smaller.

# Pooling layer

- A sequential implementation consists of a series of nested loops:
  - Each iteration of the outermost (m) for-loop generates an output feature map using…
    - the next two levels (height, width) of for-loops that generate individual pixels of the current output map.
      - The next two levels iterate over the pooling neighborhood computing max or average

# Training

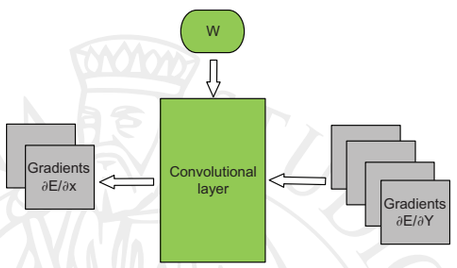
- The weights of the convolutions are learned, adjusting them during the backpropagation, calculating the gradient of loss function with respect to the elements of the output vector and propagating it toward the first layers
- We must update the weights and propagate the gradient



# Training - dE/dX

- A sequential implementation consists of a series of nested loops:
  - Each iteration of the outermost (m) for-loop of the output feature map using…
    - the next two levels (height, width) of the pixels of the current output map…
      - looping over the (c) input features…
        - over the two innermost for-loops of the convolution mask…
          - Computing the backward convolution of dE/dY with $W^T$

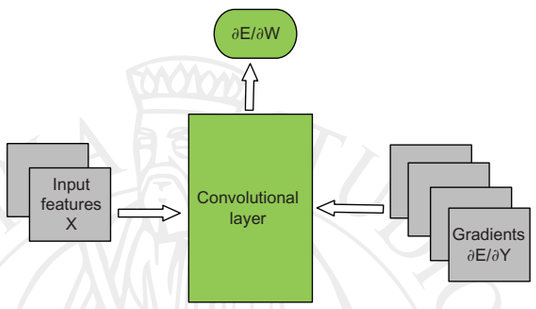
$$\frac{\partial E}{\partial X} = W^T * \frac{\partial E}{\partial Y}$$



# Training - dE/dW

- A sequential implementation consists of a series of nested loops:
  - Each iteration of the outermost (m) for-loop of the output feature map using…
    - the next two levels (height, width) of the pixels of the current output map…
      - looping over the (c) input features…
        - over the two innermost for-loops of the convolution mask…
          - Accumulating gradients over all pixels in the output map, since each filter bank affects them

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



# Mini batches

- Since training data sets are large we use Stochastic Gradient Descent, training over randomly selected small subsets (mini batches) of the dataset
- Add one more external loop over the N images of the mini batch
  - And add one more dimension to the X and Y arrays

# Parallelizing the forward path

- The loops over
  - N - mini batch images
  - M - input filters
  - H - height of input filters
  - W - width of input filters
- Can be parallelized since they are independent

# CUDA parallelization

- Assume that each thread will compute one element of one output feature map.
- Use 2D thread blocks, with each thread block computing a tile of `TILE_WIDTH` x `TILE_WIDTH` elements in one output feature map.
- Blocks are organized into a 3D grid:
  - The first dimension (X) of the grid corresponds to samples (N) in the batch;
  - The second dimension (Y) corresponds to the (M) output features maps;
  - The last dimension (Z) will define the location of the output tile inside the output feature map.
    - For simplicity assume that height and width are multiples of tile width. Note: typically squared input images are used !

# CUDA parallelization

```
# 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>>>(...);
```

```
// output coordinates Y[n, m, h, w]
n = blockIdx.x;
m = blockIdx.y;
h = blockIdx.z / W_grid + threadIdx.y;
w = blockIdx.z % W_grid + threadIdx.x;
```

# CUDA parallelization

- We can use tiling to speedup the computation of the convolution

- Keep convolution weights in shared memory

- Use dynamic allocation:

```
size_t shmem_size = sizeof(float) *
 ( (TILE_WIDTH + K-1)*(TILE_ WIDTH + K-1) +
   K*K );
ConvLayerForward_Kernel<<< gridDim, blockDim,
                           shmem_size>>>(...);
```

# CUDA Parallelization - 1

```
__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; // shorthand for threadIdx.x
w0 = threadIdx.y; // shorthand for threadIdx.y


h_base = (blockIdx.z / W_grid) * TILE_SIZE; // vert. base out data index for the block
w_base = (blockIdx.z % W_grid) * TILE_SIZE; // hor. base out data index for the block


h = h_base + h0;
w = w_base + w0;
```

# CUDA Parallelization - 1

#input maps  Convolution mask

```
__global__ void
ConvLayerForward_Kernel(int C, int W_grid, int K, float* X, float* W, float* Y) {
```

#horizontal tiles per output map  Input, mask, output

```
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; // shorthand for threadIdx.x
w0 = threadIdx.y; // shorthand for threadIdx.y


h_base = (blockIdx.z / W_grid) * TILE_SIZE; // vert. base out data index for the block
w_base = (blockIdx.z % W_grid) * TILE_SIZE; // hor. base out data index for the block


h = h_base + h0;
w = w_base + w0;
```

# CUDA Parallelization - 1

#input maps  Convolution mask

```
__global__ void
ConvLayerForward_Kernel(int C, int W_grid, int K, float* X, float* W, float* Y) {
```

#horizontal tiles per output map  Input, mask, output

```
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];
```

# input sample
# output map

```
n = blockIdx.x;
m = blockIdx.y;
h0 = threadIdx.x; // shorthand for threadIdx.x
w0 = threadIdx.y; // shorthand for threadIdx.y


h_base = (blockIdx.z / W_grid) * TILE_SIZE; // vert. base out data index for the block
w_base = (blockIdx.z % W_grid) * TILE_SIZE; // hor. base out data index for the block


h = h_base + h0;
w = w_base + w0;
```

# CUDA Parallelization - 1

#input maps Convolution mask

```
__global__ void
ConvLayerForward_Kernel(int C, int W_grid, int K, float* X, float* W, float* Y) {
```

#horizontal tiles per output map Input, mask, output

```
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;
```

# input sample

# output map

```
h0 = threadIdx.x; // shorthand for threadIdx.x
w0 = threadIdx.y; // shorthand for threadIdx.y


h_base = (blockIdx.z / W_grid) * TILE_SIZE; // vert. base out data index for the block
w_base = (blockIdx.z % W_grid) * TILE_SIZE; // hor. base out data index for the block
```

```
h = h_base + h0;
w = w_base + w0;
```

output coordinates

# CUDA Parallelization - 1

#input maps  Convolution mask

```
__global__ void
ConvLayerForward_Kernel(int C, int W_grid, int K, float* X, float* W, float* Y) {
```

#horizontal tiles per output map  Input, mask, output

```
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];
```

# input sample

# output map

```
n = blockIdx.x;
m = blockIdx.y;
h0 = threadIdx.x; // shorthand for threadIdx.x
w0 = threadIdx.y; // shorthand for threadIdx.y
```

```
h_base = (blockIdx.z / W_grid) * TILE_SIZE; // vert. base out data index for the block
w_base = (blockIdx.z % W_grid) * TILE_SIZE; // hor. base out data index for the block
```

output coordinates

```
h = h_base + h0;
w = w_base + w0;
```

```
// output coordinates Y[n, m, h, w]
n = blockIdx.x;
m = blockIdx.y;
h = blockIdx.z / W_grid + threadIdx.y;
w = blockIdx.z % W_grid + threadIdx.x;
```

# CUDA Parallelization - 2

```
float acc = 0.;

// sum over all input channels
for (int c = 0; c < C; c++) {

// h0 and w0 used as shorthand for threadIdx.x and threadIdx.y

if (( h0 < K) && ( w0 < K))
  W_shared[h0, w0]= W [m, c, h0, w0]; // load weights for W [m, c,..],
__syncthreads();

for (int 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();
```

# CUDA Parallelization - 2

```
float acc = 0.;

// sum over all input channels
for (int c = 0; c < C; c++) {

// h0 and w0 used as shorthand for threadIdx.x and threadIdx.y

if (( h0 < K) && ( w0 < K))
  W_shared[h0, w0]= W [m, c, h0, w0]; // load weights for W [m, c,..],
__syncthreads();
```

Adjust C/C++ array index syntax !!

```
for (int 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();
```

# CUDA Parallelization - 2

```
float acc = 0.;

// sum over all input channels
for (int c = 0; c < C; c++) {

// h0 and w0 used as shorthand for threadIdx.x and threadIdx.y

if (( h0 < K) && ( w0 < K))
  W_shared[h0, w0]= W [m, c, h0, w0]; // load weights for W [m, c,..],
__syncthreads();
```

Adjust C/C++ array index syntax !!

```
for (int 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();
```

Adjust C/C++ array index syntax !!

# CUDA Parallelization - 2

```
float acc = 0.;

// sum over all input channels
for (int c = 0; c < C; c++) {

// h0 and w0 used as shorthand for threadIdx.x and threadIdx.y

if (( h0 < K) && ( w0 < K))
  W_shared[h0, w0]= W [m, c, h0, w0]; // load weights for W [m, c,..],
__syncthreads();

for (int 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();
```

Preload shared memory

Adjust C/C++ array index syntax !!

Adjust C/C++ array index syntax !!

# CUDA Parallelization - 3

```
for (p = 0; p < K; p++) {

  for (q = 0; q < K; q++)

    acc = acc + X_shared[h + p, w + q] * W_shared[p, q];
  } // end p loop

  __syncthreads();

 } // end c loop

Y[n, m, h, w] = acc;

} // end CUDA kernel
```

# CUDA Parallelization - 3

```
for (p = 0; p < K; p++) {

  for (q = 0; q < K; q++)

    acc = acc + X_shared[h + p, w + q] * W_shared[p, q];

  } // end p loop

  __syncthreads();

 } // end c loop

Y[n, m, h, w] = acc;

} // end CUDA kernel
```

Adjust C/C++ array index syntax !!

Adjust C/C++ array index syntax !!

# CUDA Parallelization - 3

```
for (p = 0; p < K; p++) {

  for (q = 0; q < K; q++)

    acc = acc + X_shared[h + p, w + q] * W_shared[p, q];

  } // end p loop

  __syncthreads();

 } // end c loop

Y[n, m, h, w] = acc;

} // end CUDA kernel
```

Adjust C/C++ array index syntax !!

Compute convolution

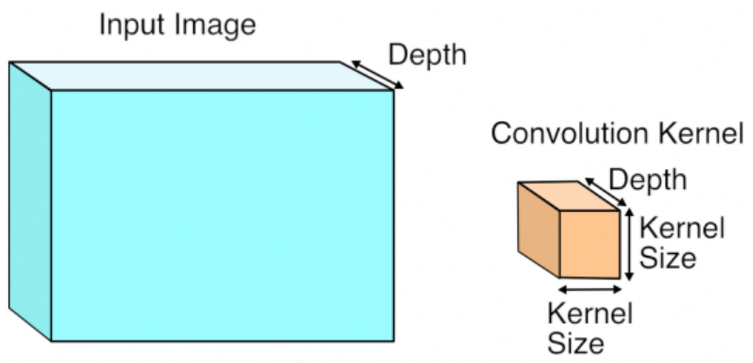
Adjust C/C++ array index syntax !!

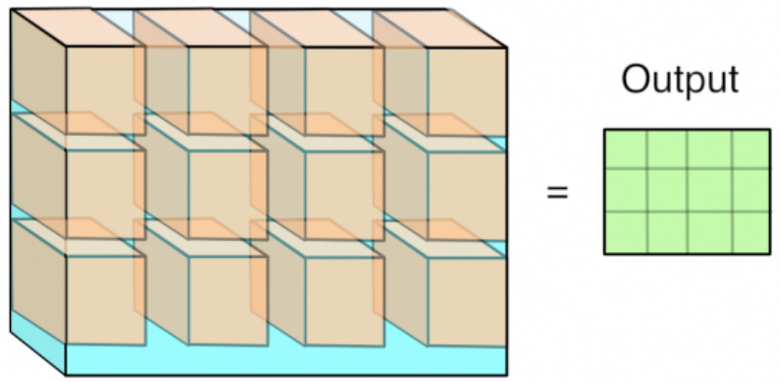
UNIVERSITÀ
DEGLI STUDI
FIRENZE

# From Convolutions to Matrix Multiplication

# Convolutions using GEMM

- Faster convolutional layers can be implemented transforming the convolution in a matrix multiplication using GEneral Matrix to Matrix Multiplication (GEMM), from CUDA linear algebra library (cuBLAS)

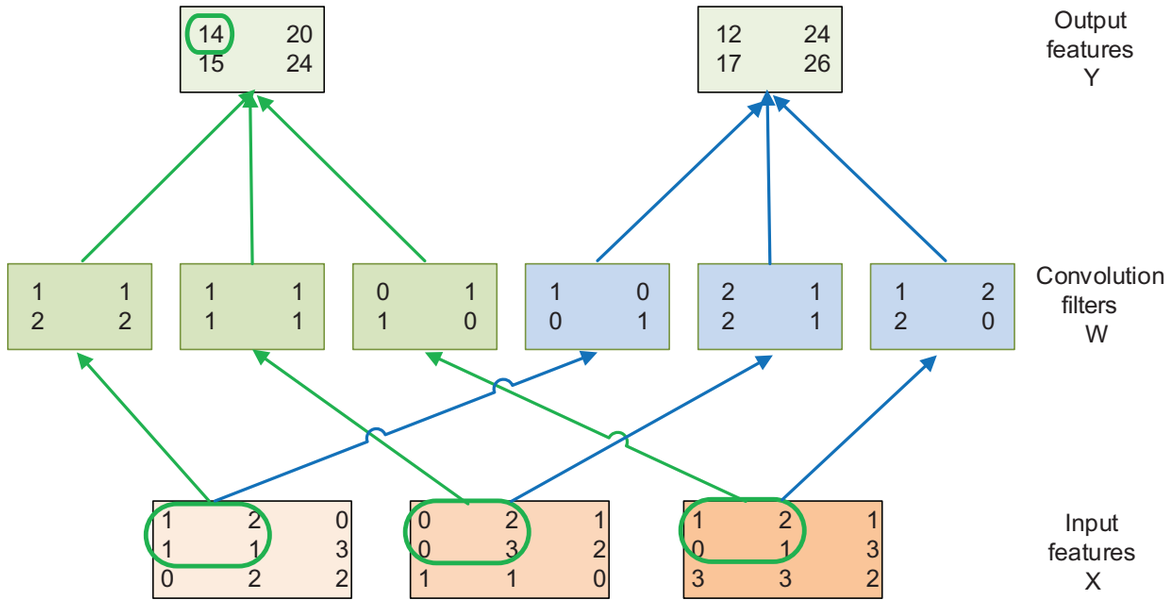


# Images to columns

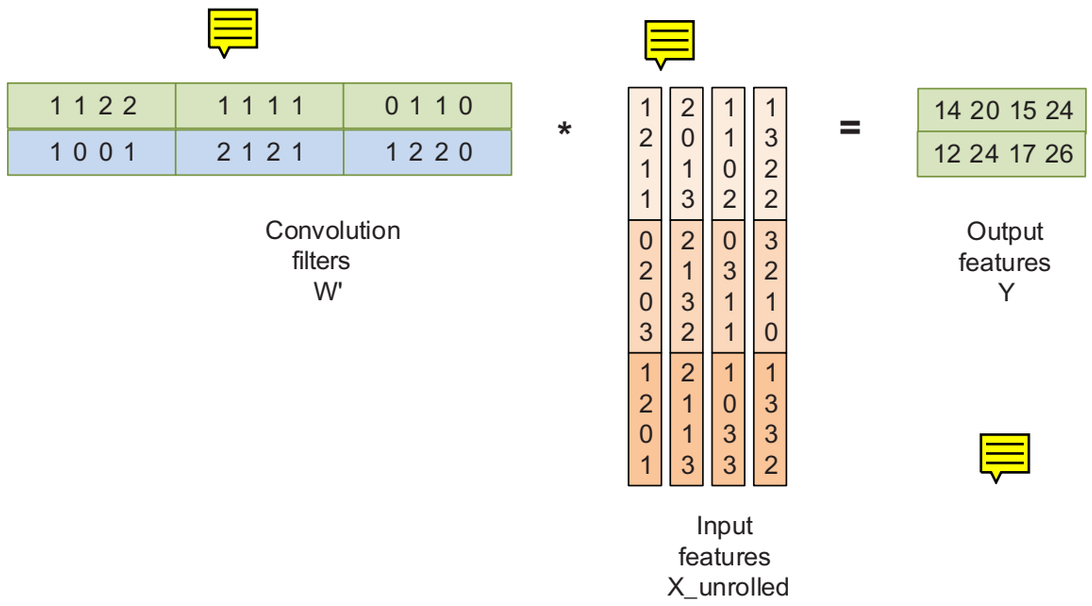
- Images are transformed into columns and convolutions into rows so that it’s possible to multiply them
- An expansion in memory size happens when we do this conversion, if the stride is less than the kernel size.
  Pixels that are included in overlapping kernel sites will be duplicated in the matrix, an inefficiency that provides benefits and that may be skipped with some lazy processing.

# Images to columns




Input feature patches become columns
Kernels become rows

Input features were 3×3, after the transformations they are 4×4 !
Filter banks remain the same.




In general, if the input feature maps and output feature maps are much larger than the filter banks, the expansion ratio of the input features will approach K*K.

# CUDA parallelization

- The input feature map samples in a mini-batch will be supplied in the same way as that for the basic CUDA kernel, as an `N x C x H x W` array, where N is the number of samples in a mini-batch, C is the number of input feature maps, H and `W` are the height and width of each input feature map, respectively.

- Filter banks are `M x C x (K x K)` arrays

- Output `Y` is stored as an `M x` $H_{out}$`*`$W_{out}$ array

- Unrolling input requires K*K times more memory, we should reuse the same buffer
  `X_unrolled[C * K * K*` $H_{out}$ `*` $W_{out}$`]`
  looping over the samples in the mini batch

- Multiplying matrices in GPU is efficient (e.g. using tiling).

# CUDA unrolling

- A CUDA kernel can be used to unroll the input features.

- The kernel contains two loops over mask indices.

- Having each thread collect all input feature map elements from an input feature map needed to generate an output generates a coalesced memory write pattern.

- It’s quite hard to manage the additional memory

UNIVERSITÀ
DEGLI STUDI
FIRENZE

# cuDNN Library

# cuDNN library

- cuDNN is a library of optimized routines for implementing deep learning primitives.

- It was designed to help deep learning frameworks take advantage of GPUs. The library provides a flexible, easy-to-use C-language deep learning API that integrates neatly into existing frameworks (e.g. Tensorflow, PyTorch, etc.)

- The library is thread-safe, and its routines can be called from different host threads.

- Convolutional routines for the forward and backward paths use a common descriptor that encapsulates the attributes of the layer. Tensors and filters are accessed through opaque descriptors, with the flexibility to specify the tensor layout by using arbitrary strides along each dimension.

# cuDNN naming convention

| Parameter | Meaning |
|---|---|
| N | Number of images in mini-batch |
| C | Number of input feature maps |
| H | Height of input image |
| W | Width of input image |
| K | Number of output feature maps |
| R | Height of filter |
| S | Width of filter |
| u | Vertical stride |
| v | Horizontal stride |
| pad_h | Height of zero padding |
| pad_w | Width of zero padding |

# cuDNN convolution

- The parameters are:
  - D is a four-dimensional $N \times C \times H \times W$ tensor which forms the input data.
  - F is a four-dimensional $K \times C \times R \times S$ tensor, which forms the convolutional filters.
  - O is a four-dimensional $N \times K \times H_o \times W_o$ tensor which forms the output and the output height and width depend (H, R, u, pad_h) and (W, S, v, pad_w) respectively.

# cuDNN convolution

```
cudnnCreateTensorDescriptor(&in_desc);
cudnnSetTensor4dDescriptor(in_desc,
    CUDNN_TENSOR_NCHW,CUDNN_DATA_FLOAT,
    in_n, in_c, in_h, in_w);
// cudaMalloc in_data

cudnnCreateFilterDescriptor(&filt_desc);
cudnnSetFilter4dDescriptor(filt_desc,
    CUDNN_DATA_FLOAT, CUDNN_TENSOR_NCHW,
    filt_k, filt_c, filt_h, filt_w);
// cudaMalloc filter_data

cudnnCreateConvolutionDescriptor(&conv_desc);
cudnnSetConvolution2dDescriptor(conv_desc, pad_h, pad_w, str_h, str_w,
    dil_h, dil_w, CUDNN_CONVOLUTION, CUDNN_DATA_FLOAT);

cudnnGetConvolution2dForwardOutputDim(conv_desc, in_desc, filt_desc,
    &out_n, &out_c, &out_h, &out_w);

cudnnCreateTensorDescriptor(&out_desc);
cudnnSetTensor4dDescriptor(out_desc, CUDNN_TENSOR_NCHW, CUDNN_DATA_FLOAT,
    out_n, out_c, out_h, out_w);
// cudaMalloc out_data

cudnnGetConvolutionForwardWorkspaceSize(cudnn, in_desc, filt_desc, conv_desc,
    out_desc, algo, &ws_size);
// cudaMalloc workspace data

// perform cudnnConvolutionForward using input, filter, output descriptors and data and workspace
```

# cuDNN convolution

- cuDNN lazily generates and loads the expanded input feature map matrix `X_unroll` into on-chip memory only

- cuDNN provides a matrix multiplication-based routine that achieves a high utilization of maximal theoretical floating-point throughput on GPUs.
  - Uses tiling
  - Fetches input data while computing matrix multiplications, hiding memory latency

- Once the computation is complete, cuDNN performs the required tensor transposition to store the result in the desired data layout of the user.

# Credits

- These slides report material from:
  - Pete Warden, JetPac Inc.

UNIVERSITÀ DEGLI STUDI FIRENZE

# Books

- Programming Massively Parallel Processors: A Hands-on Approach, D. B. Kirk and W-M. W. Hwu, Morgan Kaufman - 3rd edition - Chapt. 16