Chapter 16
Application Case Study – Machine Learning

Boris Ginsburg

Keywords: Convolutional Neural Network, Machine Learning, Deep Learning, Matrix-matrix Multiplication, Forward Propagation, Gradient Back-Propagation, Training, cuDNN

CHAPTER OUTLINE

In this chapter we will describe a case study of accelerating machine learning algorithms with GPUs. Machine learning has been used in many applications to train or adapt the application logic according to the experience gleaned from data sets. To be effective, one often needs to conduct such training with a massive amount of data. While machine learning has existed as a subject of computer science for a long time, it has recently gained a great deal of practical industry acceptance due to the availability of inexpensive, massively parallel GPU computing systems that can effectively train application logic with massive data sets. We will start with brief introduction to deep learning, and then consider one of the most popular algorithms – convolutional neural networks in more details. Convolutional neural networks have high compute-to-bandwidth ratio, and high levels of parallelism, which makes them a perfect candidate for GPU acceleration. We will first implement a convolutional neural network with a very basic algorithm. Next we show how we can

improve this basic implementation with shared memory. We will then show how one can formulate the convolutional layers as matrix multiplication problems.

16.1. Background

Machine learning is a field of computer science which explores algorithms whose logic can be learned directly from data rather than ~~be~~ explicitly programmed. Machine learning is most successful in computing tasks where designing explicit algorithms is infeasible, mostly because there is not enough knowledge in the design of such explicit algorithms. For example, machine learning has ~~provided the~~contributed to the recent improvement in the core program logic in application areas such as automatic speech recognition, computer vision, natural language processing, and search engines.

Conventional machine-learning systems required humans with considerable domain expertise to define meaningful features for transforming the raw data (e.g. the pixels of an image or digital samples of a speech excerpt) into a curated representation, from which the machine-learning algorithms could detect important patterns that can be used for training the application logic. By contrast, deep-learning is a set of methods that allow~~s~~ a machine-learning system to automatically discover the complex features needed for detection directly from raw data [1]. This area of machine learning is called ‘deep’ because it is based on the idea of hierarchical, multi-level feature representation. The hierarchical features are obtained by composing simple non-linear modules that each transform the representation at one level (starting with the raw input) into a representation at a higher, slightly more abstract level. For example in computer vision, the first layer of representation typically detect edges at particular orientations and locations in the image. The second layer typically detects so called ‘motifs’ by spotting particular patterns of edges, regardless of small variations in the edge positions. The third layer assemble these motifs into larger parts. Such layered structures, as illustrated in Figure 16.1, are often referred to as ‘feed forward networks’ since the information flows from one layer to the next layer in one direction in these systems.

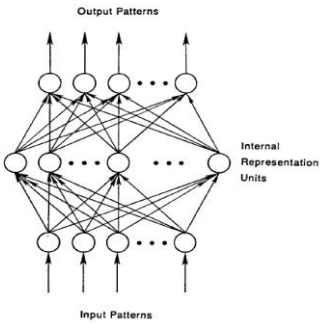

Figure 16.1: A multi-layer feed forward network

Deep learning procedures based on feed forward networks can learn very complex features which can achieve more accurate pattern recognition results ~~than~~ compared to features that are manually engineered by humans; however, this method requires that~~,~~ ~~as long as~~ there is enough data to allow the system to automatically discover ~~sufficient~~ an adequate number of relevant patterns. There is one ~~particular type~~category of deep learning procedures that are easier to train and that can be generalized much better than others. ~~This type of~~These deep-learning procedures ~~is~~ are based on a particular type of feed forward network called the convolutional neural network (CNN).

The ~~convolutional neural network (~~CNN~~s)~~ were invented in late 1980s [3]. By the early 1990s, ~~convolutional neural network~~CNNs had been successfully applied to automated speech recognition, optical character recognition (OCR), hand-writing recognition, and face recognition. However, until the late 1990s, the mainstream of computer vision and that of automated speech recognition had been based on carefully engineered features. ~~There was insufficient~~The amount of labeled data was insufficient for a deep-learning system to compete with recognition/classification functions crafted by human experts. It was a common belief that it was computationally infeasible to automatically build hierarchical feature extractors that have enough layers to perform better than human-defined application-specific feature extractors.

Interest in deep, feed ~~-~~forward networks was revived around 2006 by a group of researchers who introduced unsupervised learning methods that could create multi-layer, hierarchical feature detectors without requiring labelled data [5]. The first major application of this approach was in speech recognition. The breakthrough was made possible by GPUs that allowed researchers to train networks ten times faster than traditional CPUs. This advancement coupled with the massive amount of media data available online drastically elevated the position of deep-learning approaches. Despite their success in speech, convolutional neural networks were largely ignored in the field of computer vision until 2012.

In 2012, a group of researchers from University of Toronto trained a large, deep convolutional neural network to classify 1,000 different classes in the ILSVRC contest [7]. The network was huge by the norms of the time: it had approximately 60 million parameters and 650,000 neurons. It was trained on 1.2 million high-resolution images from the Imagenet database. The network was trained in only one week on two GPUs using ~~the~~ a very efficient ~~cuda-convnet~~CUDA-based convolutional neural network library ~~[8]~~ written by Alex Krizhevsky [8]. The network achieved break-through results with a winning test error rate of 15.3%. In comparison, the second place team that used the traditional computer vision algorithms had an error rate of 26.2%. This success triggered a revolution in computer vision, and convolutional neural networks (abbreviated as *ConvNet* for the rest of this chapter) became a mainstream tool in computer vision, natural language processing, reinforcement learning, and many other traditional machine learning areas.

16.2. Convolutional neural networks

To explain how ConvNets work, we will use LeNet-5, the network designed in the late 1980s for hand-writing digit recognition [3]. As shown in Figure 16.2, LeNet-5 is composed from 3 types of layers: convolutional layers, subsampling layers, and full connection layers. We will consider each type of layer in next section. The input to the network is shown as a gray image with a hand-written digit represented as 2D 32x32 pixel array. The last layer computes the output, which is the probability for the original image to belong to one of the 10 classes (digits) that the network is set up to recognize.

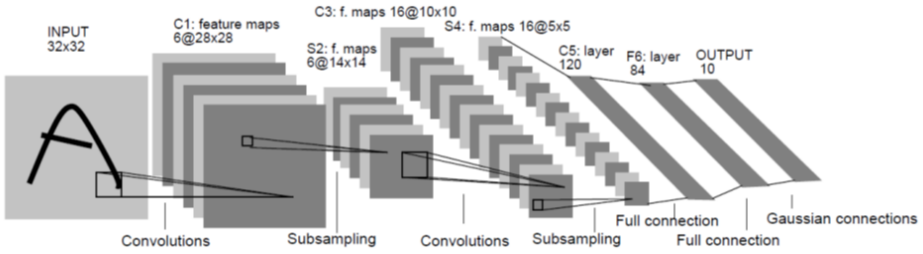

Figure 16.2 LeNet-5, a convolutional neural network for hand-written digit recognition.

ConvNets: basic layers

The computation in a convolutional network~~s~~ is organized as a sequence of layers. We will call inputs to and outputs ~~to~~ from layers 'feature maps'. For example in Figure 16.2, the computation of the C1 layer is organized to generate six output feature maps from the INPUT pixel array. The first type of layers are the convolution layers such as C1. The ~~computation result or~~ output to be produced for input feature maps consists of pixels, each of which is produced by performing a convolution between a small local patch of the feature map pixels ~~of~~ produced by the previous layer (INPUT in the case of C1) and a set of weights (i.e., a convolution mask as defined in Chapter 7) called a filter bank. Furthermore, each output feature mpa pixel is the sum of convolution results from all input feature maps.

All pixels in an input feature map are processed with the same filter bank when generating a particular output feature map. Different pairs of input and output feature map pairs in a layer use different filter banks. For example, there are 6 input feature maps and 16 output feature maps for layer C3. A total of 6*16=96 filter banks will be used in C3. Although not shown in Figure 16.2, all filter banks used in LeNet-5 are 5x5 convolutions. They differ in the 25 weights that are present in them. In general, if~~f~~ a convolution layer~~s~~ has *n* input feature maps and *m* output feature maps, *n***m* different filter banks will be used.

Formatted: Font: Italic

Formatted: Font: Italic

Formatted: Font: Italic

Recall from Chapter 7 that generating a 32x32 convolution image from a 32x32 input image and a 5x5 convolution mask requires one to make assumptions about the "ghost cells". Instead of making such assumptions, the LeNet-5 design simply uses two elements at the edge of each dimension as 'ghost cells.' This reduces the size of each dimension by four: two at the top, two at the bottom, two at the left, and two at the bottom. As a result, we see that ~~with convolution with each filter bank~~for C1, the 32x32 INPUT image results in an output feature map that is a 28x28 image. Figure 16.2 illustrates this computation by showing that a pixel in the C1 layer is generated from a square (5x5 although not explicitly shown) patch of INPUT pixels.

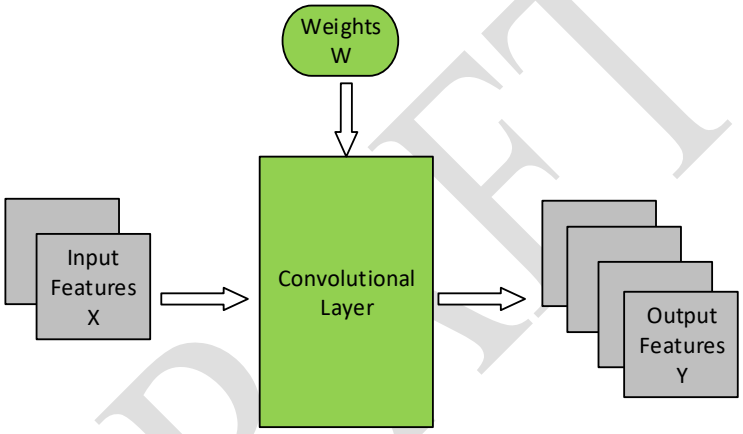

Figure 16.3 Overview of the forward propagation path of a convolution layer.

Figure 16.3 shows an overview of the forward propagation path of a convolution layer. We assume that the input feature maps are stored in a 3D array X[C, H, W], where 'C' is the number of input feature maps, 'H' is the height of each input map image, and 'W' is the width of each input map image. That is, the highest dimension index selects one of the feature maps and the lower two dimension indices selects one of the pixels in a feature map. For example, the input feature maps for the C1 layer is stored in X[1, 32, 32] since there is only one input ~~image~~ feature map (INPUT in Figure 16.2) that consists of 32 pixels in each of the x and y dimensions.

The output feature maps of a convolutional layer ~~is~~ are also stored in a 3D array Y[M, H-K+1, W-K+1], where "M" is the number of output feature maps, "H" is the height of each input map image, "W" is the width of each input map image, and "K" is the height (and width) of

each filter bank W[~~C~~M, ~~M~~C, K, K].[1] For example, the output feature maps for the C1 layer is stored in Y[6, 28, 28] since C1 generates six output feature maps and a 5x5 filter bank. Two elements are used at each edge of the image as halo cells when generating the convolved image. There are MxC filter banks. Filter bank W[m, c,_,_] is used when using input feature map X[c,_,_] to calculate output feature map Y[m,_,_]. Note that each output feature map is the sum of convolutions of all input feature maps. Therefore, we can consider the forward propagation path of a convolutional layer as set of M 3D convolutions, where each 3D convolution is specified by a 3D filter bank that is a C x K x K submatrix of W.

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

  for(int m = 0;  m < M;  m++)                  // for each output feature maps
    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] += Y[m, h, w] + X[c, h + p, w + q] * W[m, c, p, q];
      }
}
```

Figure 16.4 A sequential implementation of the forward propagation path of a convolution layer.

Figure 16.4 shows a sequential implementation of the forward propagation path of a convolution layer. Each iteration of the outermost (*m*) for-loop generates an output feature map. Each of the next two levels (h and w) of for-loops generates one pixel of the current output feature map. The innermost three levels performs the 3D convolution between the input feature maps and the 3D filter banks.

Formatted: Font: Italic

The output feature maps of a convolution layer typically go through a subsampling (also known as pooling) layer. A subsampling layer reduces the size of image maps by combining pixels. For example, in Figure 16.2, subsampling layer S2 takes 6 input feature maps of size

[1] Note that W is used for both the width of images and the name of the filter bank matrix. In each case, the usage should be clear from the context.

28x28 and generates 6 feature maps of size 14x14. Each pixel in a subsampling feature map is generated from a 2x2 neighborhood in the corresponding input feature map. The values of these four pixels are averaged to form one pixel in the output feature map. The output of a subsampling layer has the same number of output feature maps as the previous layer, but each map has half the number of rows and columns. For example, the number of output feature maps (6) of the subsampling layer S2 is the same as the number of its input feature maps, or the output feature maps of the convolutional layer C1.

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

Figure 16.5 A sequential C implementation of the forward propagation path of a subsampling layer.

Figure 16.5 shows a sequential C implementation of the forward propagation path of a subsampling layer. Each iteration of the outermost (m) for-loop generates an output feature map. The next two levels (h, w) of for-loops generates individual pixels of the current output map. The two innermost for-loops sum up the pixels in the neighborhood. K is equal to 2 in our LeNet-5 example in Figure 16.2. 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 non-linear function such as tanh, sigmoid or ReLU to give the output pixel values a more desirable distribution. ReLU is a very simple non-linear filter which passes only non-negative values:

$$Y = X, \text{ if } X \geq 0, \text{ and } 0 \text{ otherwise.}$$

To complete our example, convolutional layer C3 has 16 output feature maps, each of which is a 10x10 image. This layer has 6x16 filter banks and each filter bank has 5x5 weights. The output of C3 is passed through subsampling layer S4 which generates 16 5x5 output feature

maps. Finally, the last convolutional layer C5 which uses 16x120=1920 5x5 filter banks to generate 120 one-pixel output features from its 16 input feature maps.

These feature maps are passed through fully connected layer F6 which has 84 output units, where each output is fully connected to all inputs. The output is computed as a product of a weight matrix W with input vector X. For the F6 example, W is a 120x84 matrix then bias is added, and output is passed through sigmoid. In summary the output is an 84-element vector Y6 = sigmoid (W * X + b) assuming the implementation shown in Figure 16.2.

The final stage is an output layer that uses Gaussian filters to generate a vector of 10 elements, which correspond to the probability that input image contains one of 10 digits. It also computes *loss* functions which estimate the difference between true label and the prediction.

ConvNets: Back-propagation

Training of ConvNets is based on a procedure called gradient back-propagation. The training data set is labeled with the "correct answer." In the hand writing recognition example, the labels give the correct letter in the image. The label information can be used to generate the "correct" output of the last stage: the correct probability values of the 10-element vector.

For each training image, the final stage of the network calculates the loss function or the error as the difference between the generated output vector element values and the "correct" output vector element values. Given a sequence of training images, we can numerically calculate the gradient of the loss function with respect to the elements of the output vector. Intuitively, it gives the rate at which the loss function value changes when the values of the output vector elements change.

The back-propagation process starts by calculating the gradient of loss function dE/dY for the last layer. It then propagates the gradient from the last layer towards the first layer through all layers of network. Each layer receives as its input dE/dY – gradient with respect to its output feature maps and computes dE/dX – gradient with respect to its input feature maps.

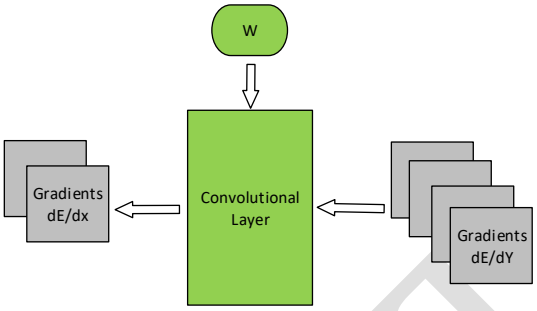

Figure 16.6: Convolutional layer: Back-propagation of dE/dX

If a layer has learned parameters ('weights') W, then the layer also computes dE/dW - gradient of loss with respect to weights:

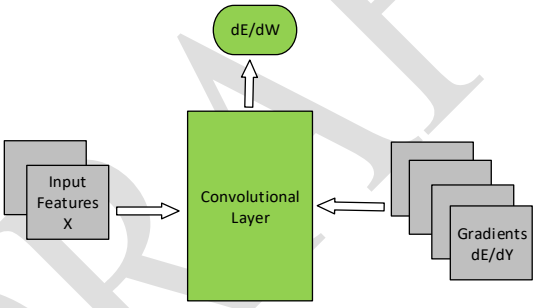

Figure 16.7 Convolutional layer: Back-propagation of dE/dw

For example the fully connected layer is given as: Y = W * X. The back-propagation of gradient dE/dY is given by two equations:

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

Let's describe now the back-propagation for a convolutional layer. We will start from the calculation of dE/dX.

Note that the calculation of dE/dX is important for propagation the gradient to the previous layer. The gradient dE/dX with respect to the channel *c* of input X is given as sum of "backward convolution" with corrsponfing $W^T(c,m)$ over all layer outputs *m:*

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

Figure 16.8 shows the calculation of the dE/dX function in the form of one matrix for each input feature map. Note that the code assumes that dE/dY has been calculated for all the output feature maps of the layer and passed in with a pointer argument dE_dY. It also assumes that the space of dE/dX has been allocated in the device memory whose handle is passed in as a pointer argument. The kernel will be generating the elements of dE/dX.

```
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];
}
```

Figure 16.8 dE/dX calculation of the backward path of a convolution layer.

The algorithm for calculating dE/dW for a convolution layer computation is very similar to that of dE/dX and is shown in Figure 16.9. Since each W(~~e~~m,~~m~~c) affects all elements of output Y(m), we should accumulate gradients over all pixels in the corresponding output feature map:

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

Note that while the calculation of dE/dX is important for propagating the gradient to the previous layer, the calculation of the dE/dW is key to the adjustments to the weight values of the current layer.

```
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];
}
```

Figure 16.9 dE/dW calculation of the backward path of a convolutional layer.

After the dE/dW values at all feature map element positions are computed, weights are updated iteratively to minimize the expected error: $W(t+1) = W(t) - \lambda * dE/dW$, where λ is a constant called the learning rate. The initial value of λ is set empirically and reduced through the iterations according to the rule defined by user The value of λ is reduced through the iterations to ensure that the convergence to a minimal error. The negative sign of the adjustment term makes the change opposite to the direction of the gradient so that the change will likely reduce the error. Recall that the weight values of the layers determine the behavior the network: they determine how the input is transformed through the network. This adjustment of these weight values of all the layers adapts the behavior of the network. That is, the network "learns" from a sequence of labeled training data and adapts its behavior by adjusting all weight values at all its layers.

The training data sets are usually large, so the training of ConvNets is typically done using Stochastic Gradient Descent: instead of doing forward-backward step to compute dE/dW for the whole training data set, one randomly selects a small subset ('mini-batch') of N images from training data set, and computes the gradient only for this subset. Next, one will select another subset etc[2]. This adds one additional dimension to all data arrays with n - the index of sample in the mini-batch. It also adds one additional loop over samples:

[2] If we would work by "optimization book" we should return samples back to the training set, and then build new mini-batch by randomly picking next samples. In practice we go sequentially over whole training set. In machine learning it's called epoch. Then we shuffle the whole training set, and start the next epoch.


```
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] = Y[n, m, h, w] + X[n, c, h + p, w + q] * W[m, c, p, q];
         }
}
```

Figure 16.10 Forward path of a convolutional layer with mini-batch training.

Figure 16.10 shows the revised forward path implementation of a convolutional layer. It generates the output feature maps for all the samples of a mini-batch. During back-propagation, one first computes the average gradient of the error with respect to the weights of last layer over all samples in mini-batch. The gradient is then propagated backward through the layers and used to adjust all the weights. Each iteration of the weight adjustment processes one mini-batch. The training typically measured in epochs, where one epoch is a sequential pass over all the samples in the training data set. The training data set is typically reshuffled between epochs.

16.3 Convolutional layer – a basic CUDA implementation of forward propagation

The computation pattern in training a convolutional network is very similar to matrix multiplication: it is both compute intensive and highly parallel. We can compute in parallel different samples in a mini-batch, different output feature maps for the same sample, and different elements for each output feature map. Figure 16.11 shows a conceptual parallel code for the forward path of a convolutional layer. Each parallel_for loop indicates that all its iterations can be executed in parallel.

```
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] = Y[n, m, h, w] + X[n, c, h + p, w + q] * W[m, c, p, q];
     }
}
```

Figure 16.11 Parallelization of the forward path of a convolutional layer with min-batch training.

As shown in Figure 16.11, there are four levels of parallelism. The total number of parallel iterations is the product N*M*H_out*W_out. This very high degree of available parallelism makes ConvNets an excellent candidate for GPU acceleration. As an example, let's implement the forward path for convolutional layer.

Let's refine the high-level parallel code into a kernel by making some high-level design decisions. Assume that we will have each thread to compute one element of one output feature map. We will use 2D thread blocks, where each thread block computes a tile of TILE_WIDTH x TILE_WIDTH elements in one output feature map. For example, if we set TILE_WIDTH = 16, we would have a total 256 threads per block. Blocks will be organized into a 3D grid:

1) the first dimension (X) in the grid corresponds to samples (N) in the batch
2) the second dimension (Y) corresponds to the (M) output features maps
3) the last dimension (Z) will define the location of the output tile inside the output feature map.

The last dim Z depends on the number of tiles in the horizontal and vertical dimensions of the output image. Assume for simplicity that H_out (height of the ouput image) and W_out (width of the output image) are multiples of 16, the tile width:

```
# define TILE_WIDTH 16
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
Z = H_grid * W_grid;
dim3 blockDim(TILE_WIDTH, TILE_WIDTH, 1);
dim3 gridDim(N, M, Z);
ConvLayerForward_Kernel<<< gridDim, blockDim>>>(…);
```

As we discussed before, each thread block is responsible for computing one 16x16 tile in the output Y(n, c, ., .)., and each thread will compute one element Y[n, m, h, w] where

```
n = blockIdx.x;
m = blockIdx.y;
h = blockIdx.z / W_grid + threadIdx.y;
w = blockIdx.z % W_grid + threadIdx.x;
```

This result is the kernel shown in Figure 16.12. Note that in the code above we use multi-dimensional index in arrays. We leave it to reader to translate this pseudo-code into regular C assuming that X, Y, and W must be accessed via linearized indexing based on row-major layout (Chapter 3).

```
__global__ void
ConvLayerForward_Kernel(int C, int W_grid, int K, float* X, float* W, float* Y)
{
  int n, m, h, w, c, p, q;
  n = blockIdx.x;
  m = blockIdx.y;
  h = blockIdx.z / W_grid + threadIdx.y;
  w = blockIdx.xz % W_grid + threadIdx.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;
}
```

Figure 16.12 Kernel for the forward path of a convolution layer.

The kernel in Figure 16.12 has a very high degree of parallelism but consumes too much global memory bandwidth. Like the based convolution pattern, the execution speed of the kernel will be limited by the global memory bandwidth. As we have seen in Chapter 7, we can use shared memory tiling to dramatically improve the execution speed of the kernel. Let's now modify the basic kernel to reduce traffic to global memory. The reader should review the tiling sections of Chapter 7 before proceeding. The kernel is shown in Figrure 16.13. The basic design is stated in the comments and outlined below:

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 partial sum of output Y_shared[n, m,.,.]
4. Move to the next input channel c

We need to allocate shared memory for the input block X_tile_width * X_tile_width, where X_tile_width = TILE_WIDTH + K-1. In addition we also need to allocate shared memory for K*K filter coefficients. So the total amount of shared memory will be (TILE_WIDTH + K-1)* (TILE_WIDTH + K-1)+ K*K . Since we do not know K at compile time we need to add it to the kernel definition as the third parameter.

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

We will divide the shared memory between input buffer and filter inside the kernel. The first X_tile_width* X_tile_width entries are allocated to the input tiles and the rest of the entries are allocated to the weight values.

```
__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;
   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, j, k, p, q;
   for (c = 0; c < C; c++) {                    // sum over all input channels
                                                // load weights for W [m, 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];
     __syncthreads();
                                        // load tile from X[n, c,…] into shared memory

     for (int i = h; i < h_base + X_tile_width; i += TILE_WIDTH) {
       for (int j = w; j < w_base + X_tile_width; j += TILE_WIDTH)
         X_shared[i - h_base, j - w_base] = X[n, c, h, w]
     }
     __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;
}
```

Figure 16.13 A kernel that uses shared memory tiling to reduce the global memory traffic of the forward path of the convolutional layer

The use of shared memory tiling results in a very high level of acceleration in the execution of the kernel. The analysis is similar to that discussed in Chapter 7 and is left as an exercise.

16.4 Reduction of convolutional layer to matrix multiplication

We can build an even faster convolutional layer by reducing it to matrix multiplication and then using highly efficient matrix multiplication GEMM from the CUDA linear algebra library (cuBLAS). This method was proposed by Chellapilla K., Puri S., Simard P. [9]. The central idea is unfolding and duplicating of the inputs to the convolutional kernel in such way that all elements needed to compute one output element will be stored as one sequential block. This will reduce the forward operation of the convolutional layer to one large GEMM matrix-matrix multiplication [3].

Consider for small example a convolutional layer which takes as input C=3 feature maps of 3x3 and produces M=2 output features 2x2. It uses M*C = 6 filter banks, where each filter bank is 2x2. The matrix version of this layer will be constructed in the following way:

First we will 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

[3] See also https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/ for very detailed explanation

of this matrix contains all the input values necessary to compute one element of an output feature. This means that each input element will be replicated multiple times. For example, the center of each 3x3 input feature is used four times to compute each element of an output feature, so it will be duplicated 4 times. The middle element on each edge is used two times so it will be duplicated two times. The four elements at corners of each input feature is used only one time and will not need to be duplicated. Therefore, the total number of elements in the expanded input feature matrix is 4*1 + 2*4 + 1*4 = 16.

In general, the size of the expanded (unrolled) input feature map matrix can be derived by thinking about the number of input feature map elements required to generate each output feature map element. In general, the height, or the number of rows, of the expanded matrix is the number of input feature elements contributing to each output feature map element. The number is C*K*K: each output element is the convolution of K*K elements from each input feature map and there are C input feature maps. In our example, the K is 2 since the filter bank is 2x2 and there are three input feature maps. Thus the height of the expanded matrix should be 3*2*2 = 12, which is exactly the height of the matrix shown in Figure 16.14.

The width, or the number columns, of the expanded matrix should be the number of elements in each output feature map. Assuming that the output feature maps are H_out x W_out matrices, the number of columns of the expanded matrix is H_out*W_out. In our example, note that the each output feature map is a 2x2 matrix so there are four columns in the expanded matrix. Note that the number of output feature maps M does not play into the duplication. This is because all output feature maps share the same expanded matrix.

The ratio of expansion for the input feature maps is the size of the expanded matrix over the total size of the original input feature maps. The reader should verify that the expansion ratio is (K*K*H_out*W_out)/(H_in*W_in), where H_in and W_in are the height and width of each input feature map. In our example, the ratio is (2*2*2*2)/(3*3)=16/9. In general, if the input feature maps and output feature maps are much larger than the filter banks, the ratio will approach K*K.

The filter banks are represented as a filter-bank matrix in a fully linearized layout, where each row contains all weight values that are needed to produce one output feature map. The height of the filter-bank matrix is the number of output feature maps (M). The height of the filter-bank matrix allows the output feature maps to share a signle expanded input matrix. The width of the filter-bank matrix is the number of weight values needed for generating each output feature map element, which is `C*K*K`. Note that there is no duplication when placing the weight values into the filter-banks matrix. In our example, the filter-bank matrix is simply a linearized arrangement of the six filter-banks.

When we multiply the filter-bank matrix W by the expanded input matrix X_unrolled the output features Y are computed as one large matrix of height M and width `H_out*W_out`.

Let's discuss now how we can implement this algorithm in CUDA. Let's first discuss the data layout. We can start from the layout of the input and output matrices.

- We assume that the input feature map samples in a mini-batch will be supplied in the same way as that for the basic CUDA kernel. It is organized 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 is the height of each input feature map, and W is the width of each input feature map.

- As we showed in Figure 16.14, the matrix multiplication will naturally produce an output Y stored as an M x H_out*W_out array. This is what the original basic CUDA kernel would produce.

- Since the filter-bank matrix does not involve duplication of weight values, we assume that it will be prepared as ahead of time and organized as an M x C x (K x K) array as illustrated in Figure 16.14.

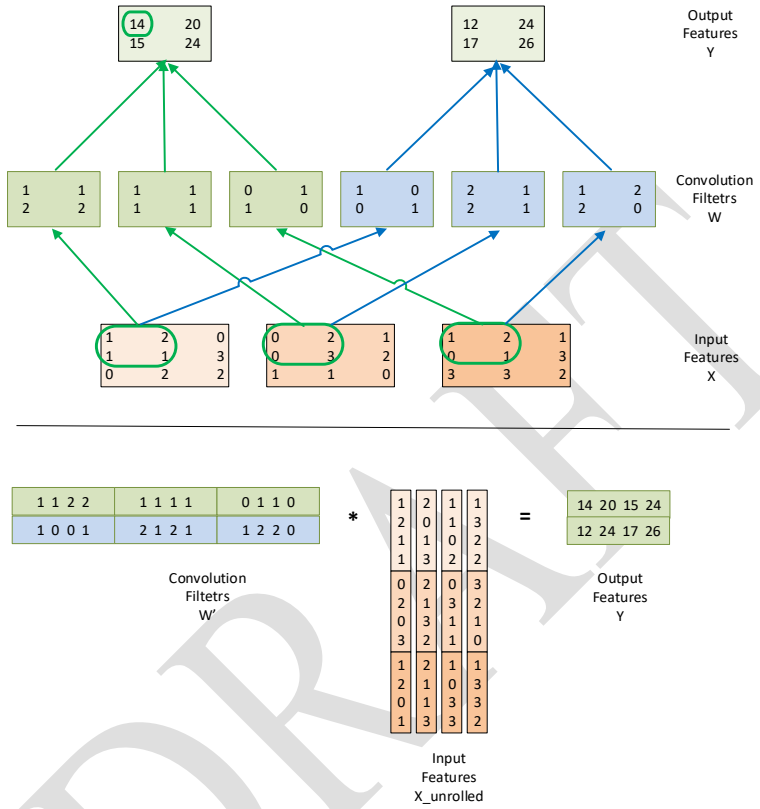

Figure 16.14: Reduction of convolutional layer to GEMM

The preparation of the expanded input feature map matrix X_unroll is more complex. Since each expansion increases the size of input by approximately up to K*K times, the expansion ratio can be very large for typical K values of 5 or larger. The memory footprint for keeping all sample input feature maps for a mini-batch can be prohibitively large. To reduce the memory footprint, we will allocate only one buffer for X_unrolled [C * K * K* H_out * W_out]. We will reuse this buffer by adding loop over samples in the batch. During each iteration, we convert the simple input feature map from its original form into the expanded matrix.

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]);
  }
}
```

Figure 16.15 Implementing the forward path of a convolutional layer with matrix multiplication.

Figure 16.15 shows the sequential implementation of the forward path of a convolutional layer with matrix multiplication. The code loops through all samples in the batch.

Figure 16.16 shows a sequential function that produces the X_unroll array by gathering and duplicating the elements of an input feature map X. The function uses five levels of loops. The innermost two levels of for-loop (w and h) place one input feature map element for each of the output feature map elements. The next two levels repeat the process for each of the K*K input feature map elements for the filtering operations. The outermost loop repeats the process of all input feature maps. This implementation is conceptually straightforward and can be quite easily parallelized since the loops do not impose dependencies among their iterations. Also, successive iterations of the innermost loop reads from a localized tile of one of the input feature maps in X and write into sequential locations in the expanded matrix X_unrolled. This should result in efficient memory bandwidth usage on a CPU.

```
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);
```

```
        }
    }
}
```

Figure 16.16 The function that generates the unrolled X matrix.

We are now ready to design a CUDA kernel which implements the input feature map unrolling. Each CUDA thread will be responsible for gathering (K*K) input elements from one input feature map for one element of output feature map. The total number of threads will be (C * H_out * W_out). We will use one-dimensional blocks. If we assume that a maximum number of threads per block is CUDA_MAX_NUM_THREADS (e.g. 1,024), the total number of blocks in the grid will be num_blocks = ceil((C*H_out*W_out) / CUDA_MAX_NUM_THREADS).

```
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>>>();
}
```

Figure 16.17 Host code for invoking the unroll kernel.

Figure 16.18 shows an implementation of the unroll kernel. Note that each thread will build a K*K section of a column, shown as a shaded box in the Input Features X_Unrolled array in Figure 16.14. Each such section contains all elements of input feature map X from channel c, required for convolution with corresponding filter to produce one element of output Y.

```
__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);
      }
   }
}
```

Figure 16.18 A high-performance implementation of the unroll kernel.

Comparing the loop structure of Figure 16.18 and Figure 16.16 shows that the innermost two loop levels in Figure 16.16 have been exchanged into outer level loops. Having each thread to collect all input feature map elements from an input feature map needed for generating an output generates a coalesced memory write pattern. As shown in Figure 16.16, adjacent threads will be writing adjacent X_unroll elements in a row as they all move vertically to complete their sections. The read access patterns to X are similar. We leave the analysis of the read access pattern as an exercise.

An important high-level assumption is that we keep the input feature maps, filter bank weights, and output feature maps in the device memory. The filter-bank matrix is prepared once and stored in the device global memory for use by all input feature maps. For each sample in the mini-batch, we launch the unroll_Kernel to prepare an expanded matrix and launch a matrix multiplication kernel, as outlined in Figure 16.15.

Implementing convolutions with matrix multiplication can be very efficient, since matrix multiplication is highly optimized on all hardware platforms. Matrix multiplication is especially fast on GPUs because it has a high ratio of floating-point operations per byte of global memory data access. This ratio increases as the matrices get larger, meaning that matrix multiplication is less efficient on small matrices. Accordingly, this approach to convolution is most effective when it creates large matrices for multiplication.

As we mentioned earlier, the filter-bank matrix is an M x C*K*K matrix and the expanded input feature map matrix is a C*K*K x H_out*W_out matrix. Note that except for the height of the filter-bank matrix, the sizes of all dimensions depend on products of the parameters to the convolution, not the parameters themselves. While individual parameters can be small, their products tend to be large. This means that size of the matrices tends to be consistently large and thus the performance using this approach can be very consistent. For example, it is often true that in early layers of a convolutional network, C is small, but H_out and W_out are large. On the other hand, at the end of the network, C is large, but H_out and W-out

attend to be small. However, the product C*H_out*W_out is usually fairly large for all layers, so performance can be consistently good.

The disadvantage of forming the expanded input feature map matrix is that it involves duplicating the input data up to K*K times, which can require a prohibitively large temporary allocation. To work around this, implementations such as the one shown in Figure 16.15 materialize X_unroll matrix piece by piece, for example, by forming the expanded input feature map matrix and calling matrix multiplication iteratively for each sample of the mini-batch. However, this limits the parallelism in the implementation, and can sometimes lead to cases where the matrix multiplications are too small to effectively utilize the GPU. This approach also lowers the computational intensity of the convolutions, because X_unroll must be written and read, in addition to reading X itself, requiring significantly more memory traffic as a more direct approach. Accordingly, the highest performance implementation has even more complex arrangements in realizing the unrolling algorithm to both maximize GPU utilization while keeping the reading from DRAM minimal. We will come back to this point when we present the CUDNN approach in next section.

16.5 CUDNN Library

CUDNN is a library of optimized routines for implementing deep learning primitives. It was designed to make it much easier for deep learning frameworks to take advantage of GPUs. It provides a flexible, easy-to-use C-language deep learning API that integrates neatly into existing frameworks (Caffe, Tensorflow, Theano, Torch,…). The library requires that input and output data be resident in the GPU device memory as we discussed in the previous section. This requirement is analogous to that of cuBLAS.

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 using arbitrary strides along each dimension. The most important computational primitive in convolutional neural networks is a special form of batched convolution. In this section, we describe the forward form of this convolution. The CUDNN parameters governing this convolution are listed Table 16.1.

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

Table 16.1: Convolution parameters for CUDNN. Note that the CUDNN naming convention is slightly different than what we have been using in previous sections.

There are two inputs to the convolution:

- D is a four-dimensional N x C x H x W tensor, which forms the input data, [4]
- F is a four-dimensional K x C x R x S tensor, which forms the convolutional filters.

The input data array (tensor) D ranges over N samples in a mini-batch, C input feature maps per sample, H rows per input feature map, and W columns per input feature map. The filters range over K output feature maps, C input feature maps, R rows per filter bank, and S columns per filter bank. The output is also a four-dimensional tensor O that ranges over N samples in the mini-batch, K output feature maps, P rows per output feature map, and Q columns per output feature map, where P = f(H;R; u; pad_h), Q = f(W; S; v; pad_w), meaning that the height and width of the output feature maps depend on the input feature map and filter bank height and width, along with padding and striding choices. The striding parameters u and v allow the user to reduce the computational load by computing only a subset of the output pixels. The padding parameters allow users to specify how many rows or columns of 0 entries are appended to each feature map for improved memory alignment and/or vectorized execution.

CUDNN supports multiple algorithms for implementing a convolutional layer: matrix-multiplication-based (GEMM and Winograd [13]), FFT-based [10], etc. The GEMM-based algorithm to implement the convolutions with a matrix multiplication, is similar to the approach presented in Section 16.4. As we discussed at the end of Section 16.4 materializing the expanded input feature matrix in global memory can be costly in both the global memory space and bandwidth consumption. CUDNN avoids this problem by lazily generating and loading the expanded input feature map matrix X_unroll into on-chip memory only, rather than by gathering it in off-chip memory before calling a matrix multiplication routine. NVIDIA provides a matrix multiplication based routine that achieves a high utilization of maximal theoretical floating point throughput on GPUs. The algorithm for this routine is similar to the algorithm described in [12] Fixed sized sub-matrices of the input matrices A and B are successively read into on-chip memory and are then used to compute a sub-matrix of the output matrix C. All indexing complexities imposed by the convolution are handled in the management of tiles in this routine. We compute on tiles of A and B while fetching the next tiles of A and B from off-chip memory into on-chip caches and other memories. This technique hides the memory latency associated with the data transfer, allowing the

[4] Tensor is a mathematical term for arrays that have more than two dimensions. In mathematics, matrices have only two dimensions. Arrays with three or more dimensions are called tensors. For the purpose of this book, one can simply treat a T-dimensional tensor as a T-dimensional array.

matrix multiplication computation to be limited only by the time it takes to perform the arithmetic

Since the tiling required for the matrix multiplication routine is independent of any parameters from the convolution, the mapping between the tile boundaries of X_unroll and the convolution problem is non-trivial. Accordingly, the CUDNN approach entails computing this mapping and using it to load the correct elements of A and B into on-chip memories. This happens dynamically as the computation proceeds, which allows the CUDNN convolution implementation to exploit optimized infrastructure for matrix multiplication. It requires additional indexing arithmetic compared to a matrix multiplication, but fully leverage the computational engine of matrix multiplication to perform the work. After the computation is complete, CUDNN performs the required tensor transposition to store the result in the user's desired data layout.

.

16.6 Exercises

1. Implement the forward path for the pooling layer described in the section 16.2.
2. We used an [N x C x H x W] layout for input and output features. Can we reduce the memory bandwidth by changing it to an [N x H x W x C]. What are potential benefits of [C x H x W x N] layout?
3. It is possible to implement the convolutional layer using FFT using the schema described in [10].
4. Implement the backward path for the convolutional layer described in the section 16.2.
5. Analyze the read access pattern to X in the unroll_kernel in Figure 16.18 and show whether the memory reads done by adjacent threads can be coalesced.

References

1. LeCun, Y., Bengio, Y., & Hinton, G. E., (2015) *Deep learning*, Nature 521, 436–444 (28 May 2015) http://www.nature.com/nature/journal/v521/n7553/full/nature14539.html
2. Rumelhart, D. E., Hinton, G. E., & Wiliams R. J. (1986). *"Chapter 8 : Learning Internal Representations by Error Propagation"*. In Rumelhart, D. E.; McClelland, J.L. *Parallel Distributed Processing*, Volume 1, MIT Press. pp. 319–362
3. LeCun, Y., Bottou L., Bengio,Y., & Haffner P., (1998). "*Gradient-based learning applied to document recognition*" . Proceedings of the IEEE 86 (11): 2278–2324, http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf

4. LeCun, Y. et al. *Handwritten digit recognition with a back-propagation network.* In Proc. Advances in Neural Information Processing Systems 396–404 (1990). http://yann.lecun.com/exdb/publis/pdf/lecun-90c.pdf
5. Hinton, G. E., Osindero, S. & Teh, Y.-W. *A fast learning algorithm for deep belief nets*. Neural Comp. 18, 1527–1554 (2006) https://www.cs.toronto.edu/~hinton/absps/fastnc.pdf
6. Raina, R., Madhavan, A. & Ng, A. Y. *Large-scale deep unsupervised learning using graphics processors*. In Proc. 26th ICML 873–880 (2009). http://www.andrewng.org/portfolio/large-scale-deep-unsupervised-learning-using-graphics-processors/
7. Krizhevsky, A., Sutskever, I. & Hinton, G. *ImageNet classification with deep convolutional neural networks*. In Proc. Advances in NIPS 25 1090–1098 (2012). https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf
8. Krizhevsky, A., cuda-convnet https://code.google.com/p/cuda-convnet/
9. Chellapilla K., Puri S., Simard P., High Performance Convolutional Neural Networks for Document Processing , 2006 https://hal.archives-ouvertes.fr/inria-00112631/document
10. Vasilache N., Johnson J., Mathieu M., Chintala S., Piantino S, LeCun Y. , *Fast Convolutional Nets With fbfft: A GPU Performance Evaluation, 2014* http://arxiv.org/pdf/1412.7580v3.pdf
11. Chetlur S., Woolley C., Vandermersch P., Cohen J., Tran J., *cuDNN: Efficient Primitives for Deep Learning NVIDIA,* 2014
12. Tan G., Li L., Treichler S., Phillips E., Bao Y., Sun N, *Fast implementation of DGEMM on Fermi GPU. In Supercomputing 2011*, SC '11, 2011
13. Lavin A., Gray S., Fast Algorithms for Convolutional Neural Networks http://arxiv.org/abs/1509.09308