CHAPTER

16 Application case study—machine learning

Boris Ginsburg

CHAPTER OUTLINE

In this chapter, we will describe a case study of accelerating machine learning algorithms with GPUs. Machine learning has been used in numerous applications to train or adapt the application logic in accordance with 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 in computer science for a considerable time, it has recently gained significant practical industry acceptance because of the availability of inexpensive, massively parallel GPU computing systems that can effectively train application logic with massive data sets. We will start with a brief introduction to deep learning and then consider one of the most widely used algorithms, convolutional neural networks (ConvNets), in more detail. ConvNets are characterized by high compute-to-bandwidth ratio and high levels of parallelism, which are perfect attributes for GPU acceleration. We will first implement a ConvNet with a basic algorithm. We will then show how we can improve this basic implementation with shared memory. Finally, we will demonstrate how one can formulate the convolutional layers as matrix–matrix multiplication problems.

Programming Massively Parallel Processors. DOI: http://dx.doi.org/10.1016/B978-0-12-811986-0.00016-9

16.1 BACKGROUND

Machine learning is a field of computer science that 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 knowledge in the design of such explicit algorithms is inadequate. Machine learning is the foundation of automatic speech recognition, computer vision, natural language processing, and search engines.

Conventional machine learning systems require humans with considerable domain expertise to define meaningful features for transforming raw data (e.g., the pixels of an image or speech signal) into a curated representation. From this representation, 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 allows a machine learning system to automatically discover the complex features needed for detection directly from raw data [LBH 2015]. This area of machine learning is described as "deep" because it is based on the idea of hierarchical, multilevel feature representation. The hierarchical features are obtained by composing simple nonlinear modules that each transforms a representation at one level (starting with the raw input) into another at a higher, slightly more abstract level. For example, in computer vision, the first layer of representation typically detects edges at particular orientations and locations in the image. The second layer typically detects the so-called "motifs" by spotting particular patterns of edges, regardless of small variations in the edge positions. The third layer assembles these motifs into larger parts. Such layered structures, as illustrated in Fig. 16.1, are often referred to as "feed forward networks" because the information flows in one direction from one layer to the next in these systems.

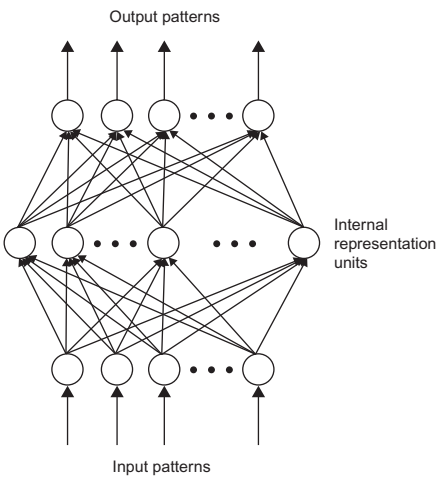

FIGURE 16.1

A multilayer feedforward network.

Deep learning procedures based on feedforward networks can learn highly complex features that can achieve more accurate pattern recognition results compared with features that are manually engineered by humans; however, this method requires that sufficient information is accessible in order to allow the system to automatically discover an adequate number of relevant patterns. One type of a deep learning procedure is based on a particular type of feedforward network called the ConvNet. These procedures are easier to train and can be generalized much better than others.

The ConvNet was invented in late 1980s [LBB 1998]. By the early 1990s, ConvNet had been successfully applied to automated speech recognition, optical character recognition, handwriting recognition, and face recognition. However, the mainstream of computer vision and that of automated speech recognition had been based on carefully engineered features until the late 1990s. The amount of labeled data was insufficient for a deep learning system to compete with recognition/classification functions crafted by human experts. The common notion was 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 feedforward networks was revived around 2006 by a group of researchers who introduced unsupervised learning methods that could create multilayer, hierarchical feature detectors without requiring labeled data [HOT 2006]. The first major application of this approach was in speech recognition. The breakthrough was made possible by GPUs that allowed researchers to train networks 10 times faster than traditional CPUs [RMN 2009]. This advancement, coupled with a massive amount of media data available online, drastically elevated the position of deep learning approaches. Despite their success in speech, ConvNets 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 ConvNet to classify 1000 different classes in the ImageNet Large Scale Visual Recognition Competition contest [KSH 2012]. 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 very efficient cuda-convnet library [Krizhevsky] written by Alex Krizhevsky. The network achieved breakthrough results with a winning top-5 test error of 15.3%. By comparison, the second place team that used the traditional computer vision algorithms achieved an error rate of 26.2%. This success triggered a revolution in computer vision, and ConvNet 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 handwritten digit recognition [LBB 1998]. As shown in Fig. 16.2, LeNet-5 is composed of three types of layers: convolutional layers, subsampling

layers, and full connection layers. We will consider each type of layer in the next section. The input to the network appears as a gray image with a handwritten digit represented as 2D 32 × 32 pixel array. The last layer computes the output, the vector which contains the probabilities for the original image to belong to each of the 10 classes (digits) that the network is set up to recognize.

CONVNETS: BASIC LAYERS

The computation in a convolutional network is organized as a sequence of layers. Inputs and outputs to layers will be referred to as "feature maps." In Fig. 16.2, the computation of the C1 layer is organized to generate six output feature maps from the INPUT pixel array. The computation result or output to be generated for input feature maps consists of pixels, each of which is produced by performing a convolution between a small local patch of feature map pixels of the previous layer (INPUT in the case of C1) and a set of weights (i.e., a convolution mask as defined in Chapter 7, Parallel patterns: convolution) called "filter bank."

All pixels in an input feature map are processed with the same filter bank when generating a particular output feature map. Different feature maps in a layer use different filter banks. Although not shown in Fig. 16.2, all filter banks used in LeNet-5 are 5 × 5 convolutions. They differ in the 25 weights that are present in them. If a convolution layers has n input feature maps and m output feature maps, n*m different filter banks will be used.

Recall from Chapter 7, Parallel patterns: convolution that generating a 32 × 32 convolution image from a 32 × 32 input image and a 5 × 5 convolution mask requires making assumptions regarding "ghost cells." However, instead of making such assumption, the LeNet-5 design simply uses two elements at the edge of each dimension as ghost cells. By so doing, the size of each dimension is reduced by four: two at the top, two at the bottom, two at the left, and two at the right. We see that by performing convolution with each filter bank, the 32 × 32 image results in a feature map that is a 28 × 28 image. Fig. 16.2 illustrates this computation by showing that a pixel in the C1 layer is generated from a square (5× 5 although not explicitly shown) patch of INPUT pixels.

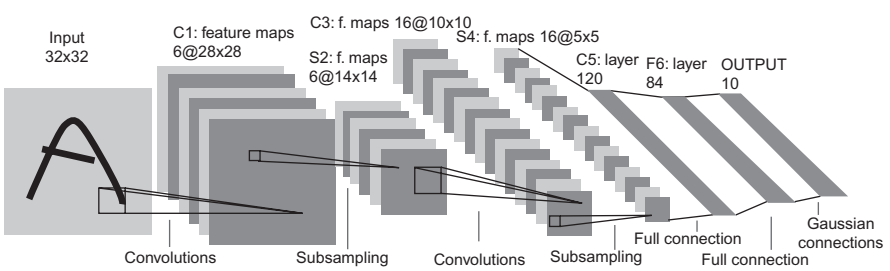

FIGURE 16.2

LeNet-5, a convolutional neural network for handwritten digit recognition.

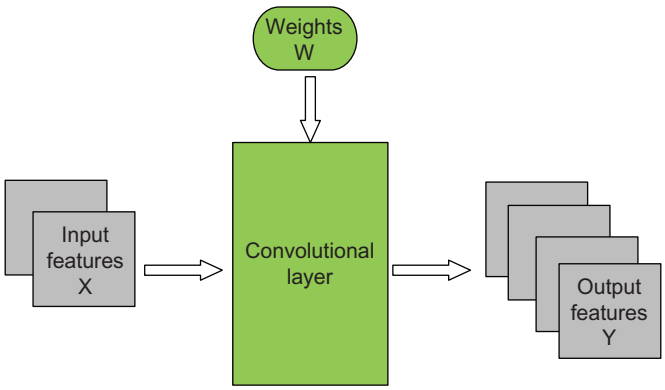

FIGURE 16.3

Overview of the forward propagation path of a convolution layer.

Fig. 16.3 provides 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. The highest dimension index selects one of the feature maps and the lower two dimension indexes selects one of the pixels in a feature map. To illustrate, the input feature maps for the C1 layer is stored in `X[1, 32, 32]` because only one input image (INPUT in Fig. 16.2) consists of 32 pixels in each of the `x` and `y` dimensions.

The output feature maps of a convolutional layer is 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, K, K]`. For instance, the output feature maps for the C1 layer are stored in `Y[6, 28, 28]` because C1 generates six output feature maps and a 5×5 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 for the input feature map `X[c,_,_]` to calculate the 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 a 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`. Note that W is used for both the width of the images and the name of the filter bank matrix. In each case, the usage should be clear from the contex.

Fig. 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 three innermost levels perform the 3D convolution between the input feature maps and the 3D filter banks.

```
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(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[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[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.

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 Fig. 16.2, the subsampling layer S2 takes six input feature maps of size 28×28 and generates six 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; however, each map has half the number of rows and columns. To illustrate, 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`.

Fig. 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 generate 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 Fig. 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 nonlinear function such as the tanh, sigmoid, or ReLU functions to provide the output pixel values a more desirable distribution. ReLU [JKL 2009], a very simple nonlinear filter, passes only nonnegative values, as follows:

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

To complete our example, the convolutional layer C3 has 16 output feature maps, each of which is a 10×10 image. This layer contains 6×16 filter banks, with each filter bank having 5×5 weights. The output of C3 is passed through the subsampling layer S4, which generates 16 5×5 output feature maps. The last convolutional layer

```
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/K; h++)         // for each output element
    for(w = 0; y < W/K; y++) {
     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 + p, K*y + 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.

C5, which uses $16 \times 120 = 1920$ 5×5 filter banks to generate 120 one-pixel output features from its 16 input feature maps.

These feature maps are passed through the Fully Connected layer F6 with 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 an input vector X. For the F6 example, W is a 120×84 matrix. Then bias is added, and output is passed through the sigmoid function. In summary, the output is an 84-element vector Y6 = sigmoid (W*X + b), assuming the implementation presented in Fig. 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 1 of 10 digits. It also computes *loss* functions, which estimate the difference between the true label and the prediction.

CONVNETS: BACKPROPAGATION

Training of ConvNets is based on a procedure called gradient backpropagation [RHW 1986]. The training data set is labeled with the "correct answer." In the handwriting recognition example, the labels give the correct digit 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 would be all "0" except the right digit which should have probability "1".

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 calculate the gradient of 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.

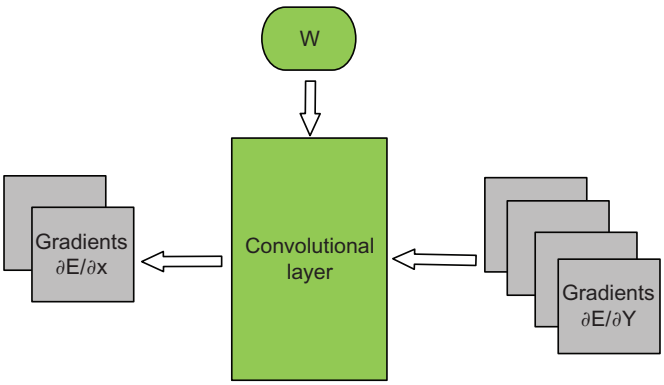

FIGURE 16.6

Convolutional layer: Backpropagation of ∂E/∂X.

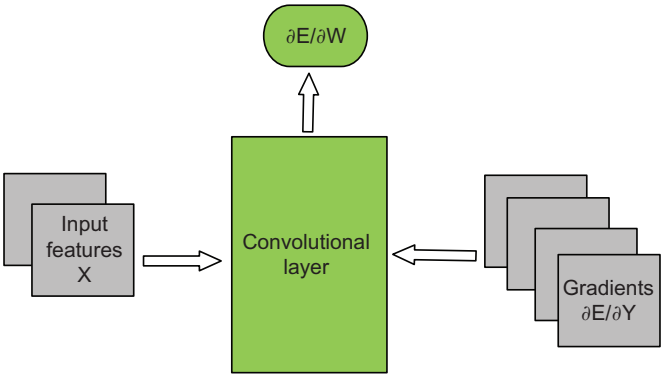

FIGURE 16.7

Convolutional layer: Backpropagation of ∂E/∂w.

Backpropagation starts by calculating the gradient of loss function ∂E/∂Y for the last layer. This process then propagates the gradient from the last layer toward the first layer through all layers of the network. 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 (Fig. 16.6).

If a layer has learned parameters ("weights") W, then the layer also calculates ∂E/∂W—gradient of loss with respect to weights (Fig. 16.7).

For instance, the fully connected layer is given as Y = W*X. The backpropagation of gradient ∂E/∂Y is expressed by two equations:

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

```
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.

We will now describe backpropagation for a convolutional layer, starting with the calculation of ∂E/∂X.

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

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

Fig. 16.8 demonstrates the calculation of the ∂E/∂X function in the form of one matrix for each input feature map. The code assumes that ∂E/∂Y has been calculated for all the output feature maps of the layer and passed 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 as a pointer argument. The kernel will be generating the elements of `dE_dX`.

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

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

Note that while the calculation of ∂E/∂X is important for propagating the gradient to the previous layer, the calculation of ∂E/∂W is key to the weight value adjustments 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 $\partial E/\partial W$ values at all positions of feature map elements are evaluated, weights are updated iteratively to minimize the expected error: $W(t + 1) = W(t) - \lambda * \partial E/\partial W$, where λ is a constant called the learning rate. The initial value of λ is set empirically and reduced through the iterations in accordance with the rule defined by the user. The value of λ is reduced through the iterations to ensure 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 how the input is transformed through the network. This adjustment of the weight values of all the layers adapts the behavior of the network, i.e. the network "learns" from a sequence of labeled training data and adapts its behavior by adjusting weight values at all layers.

The training data sets are usually large; thus, the training of ConvNets is typically accomplished using Stochastic Gradient Descent. Instead of performing a forward–backward step to determine $\partial E/\partial W$ for the whole training data set, one randomly selects a small subset ("mini-batch") of N images from the training data set and computes the gradient only for this subset. Subsequently, one selects another subset, and so on. If we would work by the "optimization book," we should return samples to the training set and then build a new mini-batch by randomly picking subsequent samples. In practice, we go sequentially over the entire training set. We then shuffle the entire training set and start the subsequent epoch. This procedure adds one additional dimension to all data arrays with n—the index of the sample in the mini-batch. It also adds another loop over samples.

Fig. 16.10 shows the revised forward path implementation of a convolutional layer. It generates the output feature maps for all samples of a mini-batch. During

```
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];
     }
}
```

FIGURE 16.10

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

backpropagation, one first computes for the average gradient of the error with respect to the weights of the last layer over all samples in a 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 is 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 highly similar to matrix multiplication: compute-intensive and highly parallel. We can compute in different parallel samples in a mini-batch, different output feature maps for the same sample, and different elements for each output feature map. Fig. 16.11 presents 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.

As shown in Fig. 16.11, the parallelism in the forward-path convolutional layer has four levels. The total number of parallel iterations is the product `N*M*H_out*W_out`. This high degree of available parallelism makes ConvNets an excellent candidate for GPU acceleration. To illustrate, forward path for a convolutional layer is implemented.

We will refine the high-level parallel code into a kernel by making some high-level design decisions. Assume that each thread will compute one element of one output feature map. We will use 2D thread blocks, with each thread block computing a tile of `TILE_WIDTH x TILE_WIDTH` elements in one output feature map. For instance,

```
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];
    }
}
```

FIGURE 16.11

Parallelization of the forward path of a convolutional layer with mini-batch training.

if we set `TILE_WIDTH=16`, we would have a total of 256 threads per block. Blocks will be 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.

The last dimension 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 output image) and `W_out` (width of the output image) are multiples of the tile width (set to 16 below):

```
# 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 previously discussed, each thread block is responsible for computing one 16×16 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;
```

```
__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;
}
```

FIGURE 16.12

Kernel for the forward path of a convolution layer.

This result is the kernel shown in Fig. 16.12. Note that in the code above, we use a multidimensional index in arrays. We leave it to the reader to translate this pseudo-code into a regular `C`, with the assumption that `X`, `Y`, and `W` must be accessed via linearized indexing based on a row-major layout (see Chapter 3: Scalable parallel execution).

The kernel in Fig. 16.12 exhibits a considerably high degree of parallelism but consumes excessive global memory bandwidth. Like in the convolution-based pattern, the execution speed of the kernel will be limited by the global memory bandwidth. We will now modify the basic kernel to reduce traffic to global memory. We can use shared memory tiling to dramatically improve the execution speed of the kernel as in Chapter 7, Parallel patterns: convolution. The kernel appears in Fig. 16.13. The basic design is stated in the comments and outlined below, as follows:

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`.

We need to allocate shared memory for the input block `X_tile_width * X_tile_width`, where `X_tile_width = TILE_WIDTH + K-1`. We also need to allocate shared memory for `K*K` filter coefficients. Thus, 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>>>(…);
    …
```

```
__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;
}
```

FIGURE 16.13

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

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

The use of shared memory tiling leads to a considerably high level of acceleration in the execution of the kernel. The analysis is similar to that discussed in Chapter 7, Parallel patterns: convolution, and is left as an exercise to the reader.

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, GEneral Matrix to Matrix Multiplication (GEMM), from CUDA linear algebra library (cuBLAS). This method was proposed by Chellapilla, Puri, and Simard [CPS 2006]. The central idea is 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. This technique will reduce the forward operation of the convolutional layer to one large matrix–matrix multiplication. See also https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/ for an elaborate explanation.

Consider a convolutional layer that takes as input C = 3 feature maps of size 3 × 3 and produces the M = 2 output features 2 × 2. It uses `M*C = 6` filter banks, with each filter bank of size 2 × 2. The matrix version of this layer will be constructed as follows:

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 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. To illustrate, the center of each 3 × 3 input feature is used four times to compute for each element of an output feature for it to be reproduced four times. The central element on each edge is used two times so that it will be duplicated. The four elements at the corners of each input feature are used only once and will not need to be reproduced. 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 considering 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, K is two since the filter bank is 2 × 2, 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 in Fig. 16.14.

The width, or the number column, 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, each output feature map is a 2 × 2 matrix so that the expanded matrix consists of four columns. The number of output feature maps M does not affect the duplication as 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

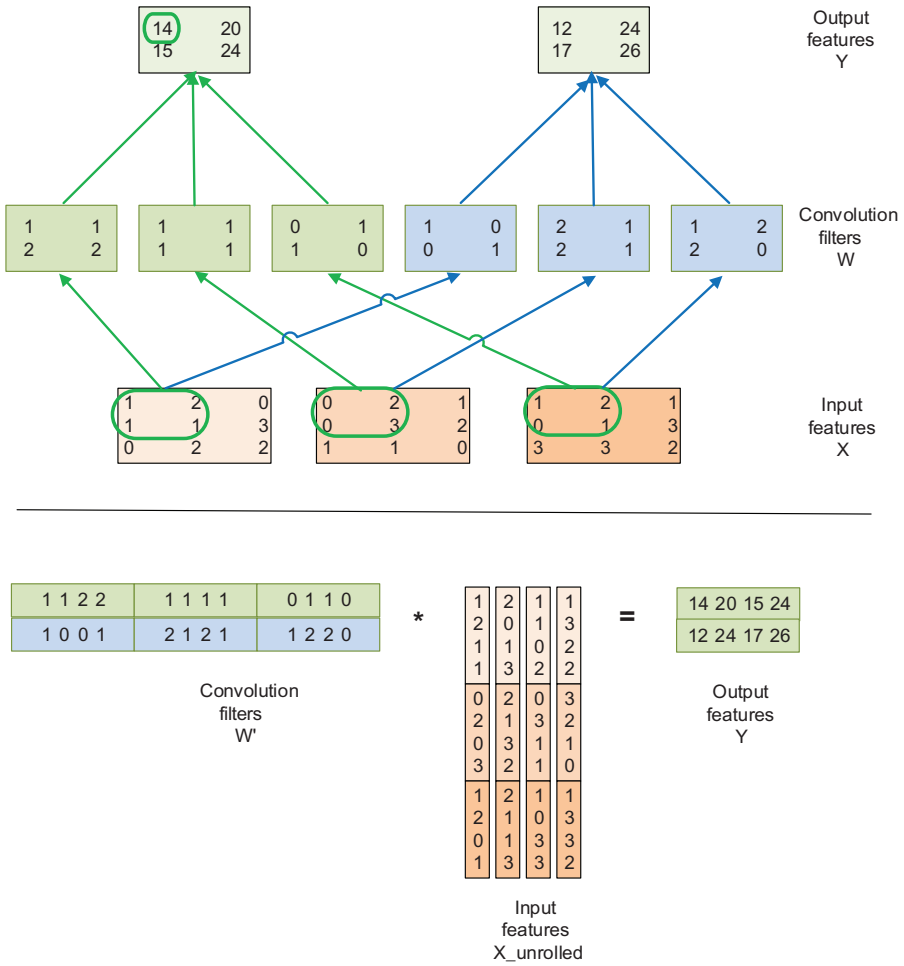

FIGURE 16.14

Reduction of a convolutional layer to GEMM.

that the expansion ratio is `(K*K*H_out*W_out)/(H_in*W_in)`, where `H_in` and `W_in` denote the height and width of each input feature map, respectively. 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 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 single expanded input matrix. Meanwhile, the width of the filter-bank matrix is

the number of weight values needed to generate each output feature map element, which is C*K*K. Note that no duplication occurs when placing the weight values into the filter-banks matrix. In our example, the filter-bank matrix is simply a linearized arrangement of six filter banks.

When the filter-bank matrix W is multiplied 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.

The discussion that follows is on the method of implementing this algorithm in CUDA. We will first discuss the data layout, starting with 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 shown in Fig. 16.14, the matrix multiplication will naturally generate an output Y stored as an M x H_out*W_out array. This output is what the original basic CUDA kernel would generate.
- Since the filter-bank matrix does not involve duplication of weight values, we assume that it will be prepared as early and organized as an M x C x (K x K) array, as illustrated in Fig. 16.14.

The preparation of the expanded input feature map matrix X_unroll involves greater complexity. Since each expansion increases the size of the 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 a loop over samples in the batch. During each iteration, we will convert the simple input feature map from its original form into the expanded matrix.

Fig. 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.

Fig. 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 two innermost levels of the for-loop (w and h) place one element of the input feature map for each of the output feature map elements. The next two levels repeat the process for each of the K*K element of the input feature map 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. In addition, successive iterations of the innermost loop read 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 process should result in efficient usage of memory bandwidth on a CPU.

```
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.

```
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 that 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 an 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., 1024), the total number of blocks in the grid will be `num_blocks = ceil((C*H_out*W_out) / CUDA_MAX_NUM_THREADS)` (Fig. 16.17).

Fig. 16.18 illustrates an implementation of the unroll kernel. Each thread will build a `K*K` section of a column, shown as a shaded box in the Input Features `X_Unrolled` array in Fig. 16.14. Each such section contains all elements of the input

```
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.

```
__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

High-performance implementation of the unroll kernel.

feature map X from channel c, which is required for convolution with a corresponding filter to produce one element of output Y.

Comparison of the loop structure in Figs. 16.16 and 16.18 indicates that the two innermost loop levels in Fig. 16.16 have been exchanged into outer level loops. 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. As shown in Fig. 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 Fig. 16.15.

Implementing convolutions with matrix multiplication can be highly efficient because 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 become larger, implying 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 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 the products of the parameters to the convolution, not the parameters themselves. While individual parameters can be small, their products tend to be large. The implication is that the matrices tend to be consistently large; thus, this approach can exhibit a consistent performance. For instance, C is often small in the early layers of a convolutional network, whereas `H_out` and `W_out` are large. At the end of the network, C is large, whereas `H_out` and `W_out` tend 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 Fig. 16.15 materialize the `X_unroll` matrix piece by piece, e.g. by forming the expanded input feature map matrix and calling matrix multiplication iteratively for each sample of the mini-batch. However, this process limits the parallelism in the implementation and can lead to matrix multiplication being too small to effectively use the GPU. This approach also lowers the computational intensity of the convolutions. The reason is that `X_unroll` must be written and read, in addition to X reading itself, requiring significantly more memory traffic as a more direct approach. Accordingly, the highest performance implementation involves even more complex arrangements in realizing the unrolling algorithm to both maximize GPU utilization while keeping the reading from DRAM minimal. We will return to this point when we present the cuDNN approach in the subsequent section.

16.5 CUDNN LIBRARY

cuDNN is a library of optimized routines for implementing deep learning primitives [CWVCT-2014]. 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 (Caffe, Tensorflow, Theano, Torch, etc.).

Table 16.1 Convolution Parameters for cuDNN

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

Note that the cuDNN naming convention is slightly different than what we have been using in previous sections.

The library requires that input and output data reside in the GPU device memory, as 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 by using arbitrary strides along each dimension. The most important computational primitive in ConvNets 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 in Table 16.1.

There are two inputs to convolution:

- D is a four-dimensional `N x C x H x W` tensor which forms the input data. 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.
- 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) and Q = f(W; S; v; pad_w). 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 computational load by computing only a subset of the output pixels. The padding parameters allow users to specify the number of 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 [Lavin & Gray]), fast-Fourier-transform-based [VJM 2014], etc. The GEMM-based algorithm used to implement the convolutions with a matrix multiplication is similar to the approach presented in Section 16.4. As 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 prevents 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 into 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 that described in [TLT 2011]. Fixed-sized submatrices of the input matrices A and B are successively read into on-chip memory and are then used to compute a submatrix 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 subsequent 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; consequently, the matrix multiplication computation is 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 nontrivial. 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 process occurs dynamically as the computation proceeds, which allows the cuDNN convolution implementation to exploit optimized infrastructure for matrix multiplication. It requires additional indexing arithmetic compared with matrix multiplication but fully leverages the computational engine of matrix multiplication to perform the work. Once the computation is complete, cuDNN performs the required tensor transposition to store the result in the desired data layout of the user.

16.6 EXERCISES

1. Implement the forward path for the pooling layer described in Section 16.2.
2. We used an `[N x C x H x W]` layout for input and output features. Can we reduce the required memory bandwidth by changing it to an `[N x H x W x C]`? What are the potential benefits of the `[C x H x W x N]` layout?

3. Implement the convolutional layer employing fast Fourier transform by using the schema described in [VJM 2014].

4. Implement the backward path for the convolutional layer described in Section 16.2.

5. Analyze the read access pattern to X in `unroll_kernel` in Fig. 16.18 and determine whether the memory reads done by adjacent threads can be coalesced.

REFERENCES

Chellapilla, K., Puri, S., & Simard, P. (2006). *High performance convolutional neural networks for document processing*. <https://hal.archives-ouvertes.fr/inria-00112631/document>.

Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., & Tran, J. (2014). *cuDNN: efficient primitives for deep learning*. NVIDIA.

Hinton, G. E., Osindero, S., & Teh, Y. -W. (2006). A fast learning algorithm for deep belief nets. *Neural Comp*, *18*, 1527–1554. <https://www.cs.toronto.edu/~hinton/absps/fastnc.pdf>.

Jarrett, K., Kavukcuoglu, K., & Lecun Y. (2009). *What is the best multi-stage architecture for object recognition*? In Proc. IEEE 12th ICCV, 2146–2153. <http://ieeexplore.ieee.org/document/5459469/?arnumber=5459469&tag=1>.

Krizhevsky, A. *Cuda-convnet*. <https://code.google.com/p/cuda-convnet/>.

Krizhevsky, A., Sutskever, I., & Hinton, G. (2012). ImageNet classification with deep convolutional neural networks. In *Proc. advances in NIPS 25 1090–1098*. <https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf>.

Lavin, A., & Gray, S. *Fast algorithms for convolutional neural networks*. <http://arxiv.org/abs/1509.09308>.

LeCun, Y. et al. (1990). Handwritten digit recognition with a back-propagation network. In *Proc. advances in neural information processing systems 396–404*. <http://yann.lecun.com/exdb/publis/pdf/lecun-90c.pdf>.

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>.

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>.

Raina, R., Madhavan, A., & Ng, A. Y. (2009). Large-scale deep unsupervised learning using graphics processors. In *Proc. 26th ICML 873–880*. <http://www.andrewng.org/portfolio/large-scale-deep-unsupervised-learning-using-graphics-processors/>.

Rumelhart, D. E., Hinton, G. E., & Wiliams, R. J. (1986). Chapter 8: Learning internal representations by error propagation. In D. E. Rumelhart & J. L. McClelland (Eds.), *Parallel distributed processing* (volume 1, pp. 319–362.). MIT Press.

Tan, G., Li, L., Treichler, S., Phillips, E., Bao, Y., & Sun, N. (2011). Fast implementation of DGEMM on Fermi GPU. In *Supercomputing 2011, SC '11*.

Vasilache, N., Johnson, J., Mathieu, M., Chintala, S., Piantino, S., & LeCun, Y. (2014). *Fast convolutional nets with fbfft: a GPU performance evaluation*. <http://arxiv.org/pdf/1412.7580v3.pdf>.