14

Architectures for Machine Learning

Traditional paradigms of computing are undergoing a tectonic shift. This is primarily because many conventional methods including the algorithms that we study in our early years of college have proven to be inadequate to solve today's problems. For example, we still do not have good algorithms for complex tasks such as face recognition, feature recognition in images, and speech synthesis. These tasks are traditionally associated with the human brain; training machines to do them using current techniques is very difficult. Such tasks were almost impossible to successfully complete in the 2010-2015 time frame with old school artificial intelligence technologies. However, off late the scenario has changed. With the advent of technologies like deep learning that mimic the processes in the human brain like never before, it is possible to solve many of these complex problems to some degree. We are far from getting the accuracies that even a 2-year-old can achieve; however, many steps have been taken in this direction in the last 5 years (as of 2020). In fact the way your author has written most of this book is by using a speech to text translation software! Typing for long hours places a strain on the wrists and shoulders; hence, to reduce the associated risk of injury and to significantly increase the speed and pleasure of writing this book, your author simply spoke in to a microphone, and a computer software did the rest!

All of these interesting things are possible because instead of using traditional algorithms based on data structures and graphs, such speech recognition software use methods known as *deep learning* or *deep neural networks*. A deep learning system is nothing but a hierarchy of simple learners that increasingly learn more and more complex concepts – this is inspired by the way the human brain learns new concepts. It is thus possible to transcribe an entire piece of speech with the right spellings and punctuation. Deep learning technologies are the norm now when it comes to image recognition, image analysis, speech recognition, natural language processing, self-driving cars, and robotic systems. In fact last night your author went for dinner and to book a table he used a chatbot that asked him for his preferences including the time and the number of people, and then dutifully booked the table in the right restaurant. What technology did the chatbot use? **Answer: Deep learning**.

Any deep learning system is divided into a set of layers. Each layer processes the inputs by computing a function over the set of inputs. They can either be linear or nonlinear functions. Designers typically alternate the linear and nonlinear layers to learn extremely complex functions. The layers learn increasingly complex concepts, and ultimately they are able to recognize faces or transcribe speech into text. Note that this kind of computation is very different from the kind of computation that happens in normal integer or floating point programs. Furthermore, the computation is massively parallel and requires a very high memory bandwidth. This is one reason why deep learning methods were not pop-

ular before 2010 primarily because we did not have the hardware to run these algorithms. But with the advent of large-scale parallel processing frameworks such as FPGAs and GPUs, large memories and storage devices, it is now possible to run deep learning algorithms on massive amounts of data. Just consider the vast amounts of data that social networking sites process on a daily basis. To analyze all this data, we need very large data centers that essentially run deep learning algorithms.

Let us make it clear that this chapter is not about teaching the fundamentals of deep learning or discussing popular software implementations. Readers can refer to the book by Goodfellow, Bengio, and Courville [Goodfellow et al., 2016] to get a thorough understanding of deep learning technologies. Some popular deep learning frameworks such as Caffe [Jia et al., 2014], TensorFlow [Abadi et al., 2016], and Keras [Gulli and Pal, 2017] are extensively documented and readers can go through them. This chapter is devoted to novel computer architectures that are designed exclusively for accelerating deep learning algorithms. We shall first provide a very brief introduction to some popular deep learning architectures, then discuss the process of mapping the code to an architecture, and finally discuss the design of custom deep learning hardware.

14.1 Basics of Deep Learning

The aim of any learning system is to learn a function that is hidden. The learner is given a set of inputs and their corresponding outputs. Based on them, it needs to estimate the function that computes the outputs given the inputs. The process of trying to learn this function is known as *training*. Once the learner (also known as the *model*) has been trained, it can be used to predict the output of a hitherto unseen input. This process is known as *testing*. Since we do not know the actual process that converts the inputs to the outputs, the function that is estimated will be an approximation of the real function. Hence, it is expected that there will be some error in the output. Better learners minimize this error over a set of test inputs. In the learning literature there are several ways to measure the error: absolute value of the difference, mean square error, and so on. Regardless of the way that the error is measured, the main aim of any learning process is to minimize the error for unseen test inputs.

14.1.1 Formal Model of the Learning Problem

A learning system typically takes an n-element vector $\mathbf{x}$ as input and returns an output y, which can be a floating point value or an integer. We shall use the bold font for vectors and matrices, and use the regular font for scalars. In addition, let the real output be $\hat{y}$ in this case. The error is thus a function of y and $\hat{y}$. Furthermore, following convention let $\mathbf{x}$ be a column vector. We traditionally write $\mathbf{x} \in \mathcal{R}^n$, which means that $\mathbf{x}$ is a vector that contains n real numbers. In general, the training algorithm is given a set of inputs and a set of outputs. Let us represent the set of inputs as $\mathbf{X}$, where the i^{th} column is the i^{th} input vector, and the set of outputs as a column vector $\mathbf{y}$ (the i^{th} entry is the i^{th} output). We want to find the relationship – function f^* – between the set of inputs and the set of outputs. Given that we will never get to know what the real relationship actually is, we have to make an intelligent guess from the training data that has been provided to us: $\mathbf{X}$ and $\mathbf{y}$. It is important to note here that the hidden function f^* does not take into account the order of inputs. It is in a sense memoryless – does not remember the last input.

The main aim of the learning problem is to find a good estimate for f^*. The number of possible functions is very large, and unless we simplify the problem, we will not arrive at a good function. The simplifying assumption is that in this case we desire a *universal approximator*. A universal approximator is an algorithm that does not rely on any a priori estimate of f^*. It can be used to approximate any continuous function with inputs in $\mathcal{R}^n$. Every such approximator takes in a list of parameters that completely specify its behavior; it is possible to approximate different hidden functions just by changing the parameters. This method simplifies the learning problem significantly; all that we need to do is simply estimate the parameters of a universal approximator.

Definition 104
A universal approximator is an algorithm that does not rely on any a priori estimate of the function to be learned (f^* in this case). Moreover, it can be used to approximate any continuous function with inputs in $\mathcal{R}^n$, and its behavior can be fully controlled by a list of parameters.

Most of the initial learning algorithms were not universal approximators; hence, they failed for many classes of learning problems. Let us outline the journey from simple linear models to deep neural networks.

Linear Regression

The simplest approach is to assume that f^* is a linear function. We can thus estimate y as follows:

$$y = \mathbf{w}^T\mathbf{x} + b \tag{14.1}$$

Here, $\mathbf{w}$ is a *weight vector* and b is a bias parameter. Even though this approach is very simple, however its main shortcoming is that if f^* is not linear then the estimate can turn horribly wrong. For example if f^* consists of *sine*, *cos*, *tan*, and other transcendental functions then a linear estimate will give us a very low accuracy. Such a linear approach is not a universal approximator because of this issue.

ML using Nonlinear Models

Given that linear models are associated with significant limitations, much of the research in machine learning has moved towards nonlinear models. Here we consider complex nonlinear functions that are parameterized by a set of constants. The task of the learning algorithm is to find an appropriate set of constants that minimize the error.

In this space, support vector machines and support vector regression are very popular. A support vector machine (SVM) is used for binary classification. Here it is assumed that each input data point is associated with a binary bit: 0 or 1 (its class). We can assume that each input vector is a point in an n-dimensional space. We can increase the number of dimensions by mapping each point to an even higher dimensional space where it is easy to segregate the points labeled 0 and 1 respectively. SVM based approaches try to fit an imaginary hyperplane (surface in a high dimensional space) that separates both the classes of points. It is not necessary to compute the equation of this plane directly, we can instead describe its behavior by considering a set of points that are the closest to it (known as *support vectors*). This basic approach can be extended for classification with multiple classes, and can even be used to estimate real values instead of discrete class labels. The main problem of SVMs are the difficulty in finding functions to map points to higher dimensional spaces and their limited accuracy as compared to neural networks.

14.1.2 Neural Networks

Modeling Nonlinearity with Neural Networks

The main issue that constrained prior approaches was that there was no effective way to model nonlinearity. Given that the space of functions that we want to approximate is potentially infinite, the learning model should be general enough such that we can achieve a low error with almost any data set. Neural networks (inspired by the human brain) are universal approximators and also generalize very well.

The key idea here is to introduce nonlinear transformations along with linear transformations such that the relationship between the input and the output can be captured accurately. We introduce a

function g such that the output can be represented as $g(\mathbf{w}^T\mathbf{x} + b)$. The function g is typically one of the following functions.

Sigmoid function This was one of the earliest functions used in the design of neural networks. It is defined as,

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{14.2}$$

tanh function This is the hyperbolic tangent function.

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{14.3}$$

ReLU activation function "ReLU" stands for *Rectified Linear Unit*. The function associated with it, also known as the activation function, is as follows:

$$f(x) = \begin{cases} 0, & (x < 0) \\ x, & (x \geq 0) \end{cases} \tag{14.4}$$

Let us visualize all the three functions in Figure 14.1.

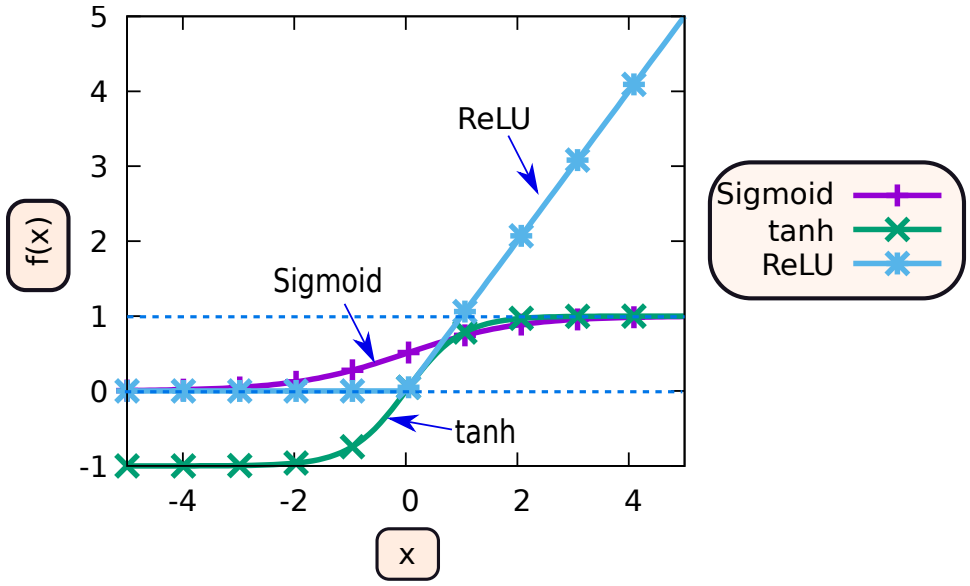

Figure 14.1: The Sigmoid, *tanh*, and ReLU activation functions

Let us now look at an example that uses such nonlinear units. Let us solve a problem that is not possible to solve with purely linear approaches. We wish to create a network that evaluates the XOR function. Consider the input to be a column vector $[a, b]$, where the output is $a \oplus b$. The reader needs to convince herself that simply by multiplying weights with a and b and adding the results, it is not possible to realize a XOR function. Note that for the sake of readability we will be writing the column vectors horizontally. For example, as per our representation $[a, b]$ is a column vector and $[a, b]^T$ is a row vector.

Let us focus on the Karnaugh map in Figure 14.2(a). The aim is to identify and "somehow nullify" the inputs when $a = b$. Let us compute the vector product $[1, -1]^T[a, b]$ (dot product of $[1, -1]$ and $[a, b]$)

where $[1, -1]$ is the weight vector. This is arithmetically the same as computing $a - b$. The results are shown in Figure 14.2(b). For the inputs $(0, 0)$ and $(1, 1)$, the result of this operation is 0. For the inputs where $a \neq b$, the result is non-zero (1 and -1). The final output needs to be the modulus of this result. Computing $|x|$ is easy. It is equal to $ReLU(x) + ReLU(-x)$. The resulting neural network is shown in Figure 14.2(c). Note that we have two functional units in the first linear layer. Each unit computes a dot product between a weight vector and the input vector. For the first functional unit the weight vector is $[1, -1]$, and for the second functional unit it is $[-1, 1]$. The second weight vector is generated by multiplying $[1, -1]$ with -1 because we wish to compute $ReLU(-x)$ in the next nonlinear layer. The rest is self-explanatory.

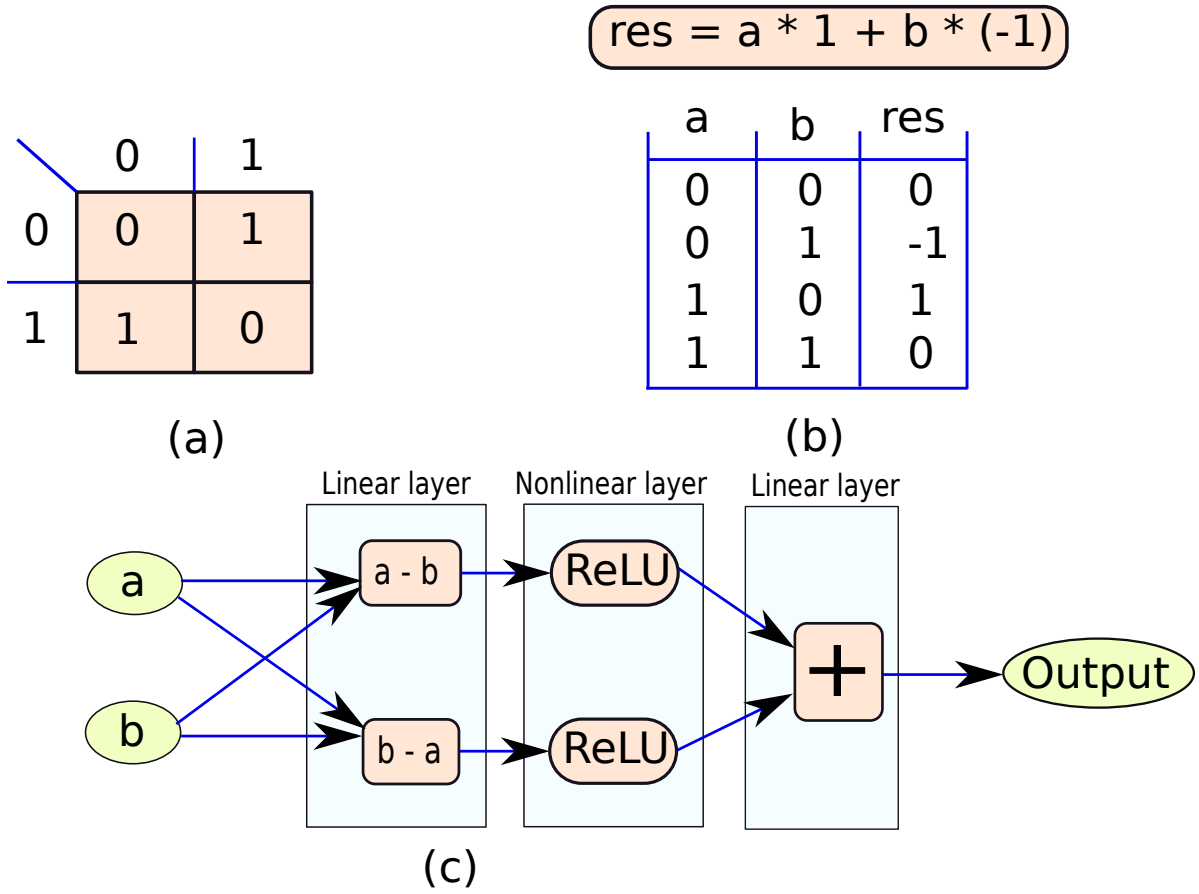

Figure 14.2: Computation of the XOR function. (a) Karnaugh map, (b) Outputs after computing a dot product with [1,-1], (c) Structure of the network

The structure of the network has an interesting property: it has alternating linear and nonlinear layers. The inputs are fed into the first linear layer, which computes dot products with different weight vectors. Subsequently, the outputs of this layer are passed to a nonlinear layer that uses the ReLU function. Finally, the outputs of the nonlinear layer are passed to a linear layer that generates the final output of the neural network. In this case, this network implements a XOR function. Even though this network appears to be very simple, readers will be surprised to know that for a very long time it was not possible to come up with such a network! This had stalled the development of neural networks for decades. Gradually, neural networks increased in terms of complexity: this led to an increasing number of layers and weights within a layer.

Note that this was a specific example for a simple function. In general, it is possible to take such neural networks and train the weights to compute any given function. The architecture is fairly generic, and we can learn functions by simply changing the weight vectors. There are two important terms that we need to introduce here: training and inferencing. While *training* a neural network we are provided a set of known inputs and outputs, and then we try to compute the weights such that the outputs of the network match the given outputs as far as possible. Almost all neural networks use the *backpropagation algorithm* [Goodfellow et al., 2016] for computing the weights in the training phase. It is important to note that the architecture of a neural network in terms of the number and type of layers, and the nature of functional units within the layers is decided a priori. Subsequently, these parameters do not change. The training phase is used to only compute the weights in the network. This phase is carried out offline and there is typically no need to accelerate this phase in hardware.

However, for hardware designers, the *inferencing part* where given an unknown input we try to predict the output, is far more important. Almost all neural architectures as of 2020 focus exclusively on accelerating the inferencing.

Note that the expressive power of a neural network is dependent on the number of layers and the number of functional units within each layer. Hence, for learning complex functions it is typically necessary to have deeper neural networks with more layers. Previously, neural networks used to be fairly small with 3 to 4 layers as we showed in Figure 14.2. With an increase in compute power, the rise of GPUs and FPGAs, and also a concomitant increase in memory capacity, we can now afford to have large neural networks with a few hundred layers. This has spawned the revolution in the design of large deep neural networks.

Deep Neural Networks

The simple example that we saw in Figure 14.2(c) had three layers: two linear layers and one nonlinear layer. Such small networks are good for learning small and simple functions. However, to identify far more complex patterns such as the number of faces in an image or perform face recognition, we need many more layers. Neural networks with a large number of layers are known as *deep neural networks* or simply *DNNs*. It is not uncommon for a modern neural network to have 100+ layers with millions of weights. Such deep neural networks have a series of layers that progressively learn more and more complex concepts. For example, if we would like to classify the images of different animals, the first few layers serve to identify local features such as the paws, the head, the body and the tail. Subsequent layers try to aggregate all this information and identify global features. For example, it is hard for a computer to differentiate between a dog and cat by just looking at the tail or the legs. However, when we consider the image in entirety, the difference between a dog and a cat is very obvious. This is the job of the later layers that aggregate all the information that has been learned by the earlier layers and then try to infer high-level concepts.

Before proceeding forward, let us define some terms.

Input Feature Map or *ifmap* The input feature map is an input to a functional unit in a layer. For DNNs processing images, it is a 2D matrix. It can also be a 1D vector or a 3D matrix (if we are considering a set of images).

Output Feature Map or *ofmap* It is an output of a functional unit in a layer. Note that a layer typically has a multitude of functional units. It thus takes several *ifmaps* as inputs, and produces several *ofmaps* as outputs. For a given layer, all the *ifmaps* (or *ofmaps*) typically have the same dimensions; however, the dimensions of an *ifmap* and an *ofmap* need not be the same.

Pixels We refer to each entry of an *ifmap* or *ofmap* as a *pixel*.

In such neural networks we never have two linear layers adjacent to each other. This is because two adjacent linear layers are equivalent to a single linear layer. DNNs typically have alternating linear and nonlinear layers. As we have seen, the linear layer simply computes a dot product between the *ifmaps*

and a vector of weights. Additionally, DNNs also use different types of nonlinear layers that are either ReLU, Sigmoid or pooling layers. A *pooling layer* takes a region of $K \times K$ pixels in an *ifmap*, and replaces it with a single value. This can either be the mean of the values or the maximum. The latter is known as *max pooling*. The advantage of pooling is twofold: we reduce the size of data by K^2, and secondly, if there is some translation in the feature (displacement by a few pixels) then this operation successfully mitigates its effect.

Before an astute reader asks how the layers are connected, let us answer the question. The most common category of networks is the deep feed-forward network. Here, the layers are organized as a linked list, one after the other. The output of the i^{th} layer is the input of the $(i+1)^{th}$ layer. This however need not always be the case. We can have back edges, where a later layer feeds its outputs to an earlier layer. Such cyclic connections make the process of training and inferencing harder. However, they also increase the expressive power of the neural network. Let us now look at the most popular variants of DNNs in use today, which are known as Convolutional Neural Networks (CNNs).

14.1.3 Convolutional Neural Networks (CNNs)

Let us reconsider the linear layers in Figure 14.2 once again. In each functional unit of a linear layer we compute a dot product between the *ifmaps* and weight vectors. In most modern neural networks that process complex images or analyze speech, the *ifmaps* are very large. In each linear layer in particular, we need to store large weight vectors and then compute the dot products. If we have millions of pixels in an *ifmap*, which is incidentally not that uncommon, we also need to store millions of weights just for each layer. The storage complexity, memory access overheads, and compute time will make the process of training and inferencing intractable. Additionally, for training we will need a prohibitive number of examples, which in most cases will prove to be impossible. We thus need to simplify the problem. Hence, we typically create two kinds of linear layers: one that has a lot of weights, and a layer that uses a small set of weights.

Fully Connected Layer This is a traditional linear layer where we simply multiply each element in an *ifmap* with a weight. If the *ifmap* has N elements, then we also need N weights. We cannot afford many such layers given the amount of computation that is needed to generate a single output value. Typically, the last layer in a DNN is a fully connected layer. This layer is presented with a lot of high-level concepts identified by earlier layers, and it simply needs to make the final decision. It has very high memory requirements as well. Given that there are a very few such layers, their total contribution to the computational time is small ($\approx 10\%$).

Convolutional Layer For intermediate linear layers we do not store large weight vectors. Instead, we store a very small set of weights known as a *filter*. We compute a convolution between typically two-dimensional *ifmaps* and the filter to compute an *ofmap*.

Before going into the details, let us explain the high level idea. Consider a neural network that needs to classify the image of an animal. We need to first identify small features such as the mouth, horns, and paws. Even before identifying them we need to identify the edges and make a distinction between points within the image and outside it. For detecting edges, and simple shapes, we do not need to compute a dot product with a weight vector that is as large as the *ifmap*. Conceptually, this is a local operation, and computing a localized dot product with a small weight vector should suffice. This is precisely the idea. We consider a small filter with R rows and S columns and a portion of the *ifmap* with the same dimensions and just compute a dot product. This is known as the *convolution operation*. We can extend our definition of a dot product of two vectors to a dot product of two n-dimensional matrices. Here, we multiply corresponding elements, and the value of the final dot product is a sum of the individual element-wise products.

The Convolution Operation

Consider Figure 14.3. It shows an *ifmap* with H rows and W columns, and a given position within it (h, w). The convention that we adopt is that we list the row number first and then the column number (similar to addressing a matrix). We draw an $R \times S$-element shaded rectangle whose left top is at (h, w). We then compute a dot product between the elements of the filter and the shaded rectangle in the *ifmap*. Let us represent the *ifmap* by the matrix $\mathbf{I}$, the *ofmap* by the matrix $\mathbf{O}$, and the filter by the matrix $\mathbf{F}$. We can then write this formally as

$$\mathbf{O}[h][w] = \sum_{r=0}^{R-1} \sum_{s=0}^{S-1} \mathbf{I}[h+r][w+s] \times \mathbf{F}[r][s] \tag{14.5}$$

This is not a traditional convolution operation that we learn in a signal processing course – this is an element-wise dot product. However, we can modify the classical convolution equations to become equivalent to Equation 14.5 by changing the sign of some variables. Basic insight: note that $\sum f(x-y)g(y) = \sum h(z+y)g(y)$, if we consider $z = -x$ and $h(x) = f(-x)$. We will thus henceforth refer to expressions listed in Equation 14.5 as convolutions.

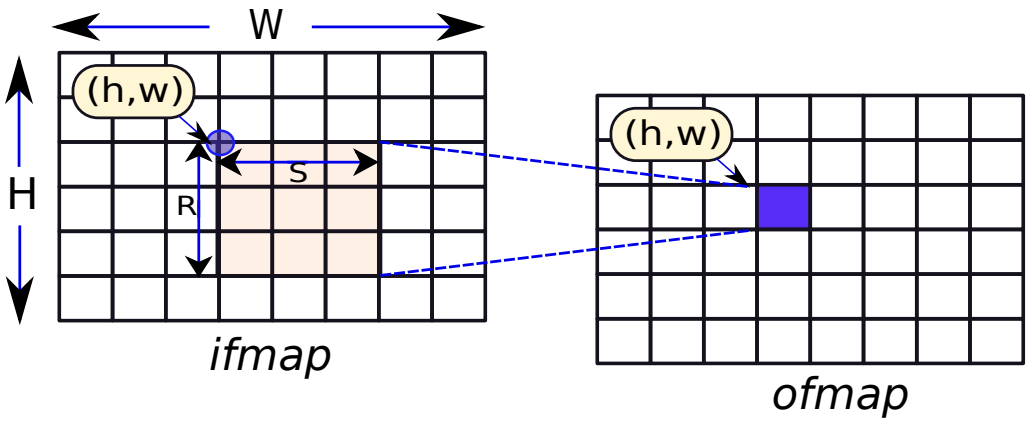

Figure 14.3: A convolution operation

The main advantage here is that we need not store very large weight vectors. We can just store small filters. Let us now complicate Equation 14.5 and make it more realistic. Here are some of the crucial insights. First, an *ofmap* is typically dependent on multiple *ifmaps*, and there is a unique filter for each *ifmap-ofmap* pair. Second, we typically compute a set of *ofmaps* in each layer, and finally to maximize the reuse of inputs and filter weights, we process a *batch* of input images in one go. Let us thus introduce some additional terminology. Consider a layer that takes as input C *ifmaps* (each *ifmap* is called a *channel*), and produces K output *ofmaps*. Additionally, the entire neural network processes N input images in a batch. For the sake of simplicity, let us assume that all the *ifmaps* and *ofmaps* have the same dimensions: $H \times W$ (row-column format). The terminology is summarized in Table 14.1. Please thoroughly memorize the terms. We shall be using them repeatedly in the next few sections.

Variable	Symbol	Iterator
Number of input images	N	n
Number of input channels	C	c
Number of *ofmaps* produced	K	k
Dimensions of the *ifmaps*/*ofmaps*	$H \times W$	h and w
Dimensions of the filter	$R \times S$	r and s

Table 14.1: Terminology used to describe a convolution operation

We can thus represent a convolution operation as follows:

$$\mathbf{O}[n][k][h][w] = \sum_{c} \sum_{r} \sum_{s} \mathbf{I}[n][c][h+r][w+s] \times \mathbf{F}[k][c][r][s] \tag{14.6}$$

We make several simplifications in this equation. We omit the ranges of the iterators, and secondly we assume that the operation is defined for pixels at the edges of the image. Consider the pixel at the bottom right $\mathbf{O}[H-1][W-1]$. The convolution operation is not defined for this pixel because most of the pixels that need to be considered for the convolution do not exist. In this case, a simplifying assumption is typically made where we assume the existence of additional elements beyond the bottom and right edges that contain zeros. This is known as *zero padding*.

Moreover, we observe that for each input image, we compute a convolution. Each pixel of an *ofmap* is dependent on all the input *ifmaps* (general case), and for each *ifmap-ofmap* pair we have a filter. This is the basic equation for a convolution, which allows us to avoid heavy computations with large weight vectors. Such convolutional neural networks are known as CNNs, and they have proved to be extremely useful in very diverse fields.

Design of a CNN

We have four kinds of layers in a CNN: convolutional layer, fully connected layer, ReLU layer, and pooling layer. The latter two layers are nonlinear layers. An illustration of a CNN's design is shown in Figure 14.4 that shows all these layers.

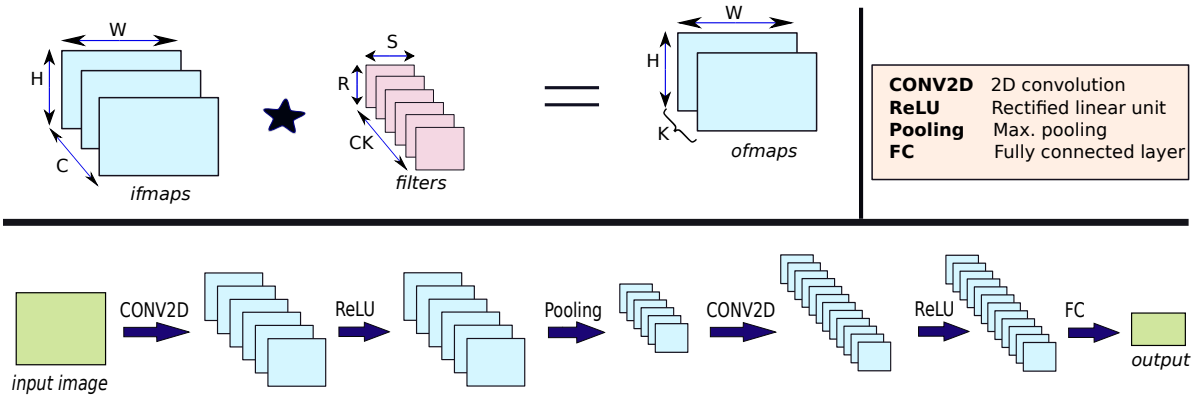

Figure 14.4: Design of a CNN

From the point of view of computation, the ReLU and pooling layers are very easy to handle. We require very little logic to realize their functionality, and their execution is a very small fraction of the total execution. The fully connected layer is heavy in terms of its memory footprint and computational overhead. However, since we have only one such layer in a deep neural network and its generate a few outputs, its execution time is not particularly concerning. Almost all the research in CNNs has been devoted to accelerating the convolutional layers that account for more than 90% of the total execution time. We shall thus henceforth focus on the convolutional layers and discuss various methods to optimize their execution.

14.2 Design of a CNN

14.2.1 Overview

As far as we are concerned, the only computation that we need to do is compute all the outputs as per Equation 14.6. Let us reproduce it again for the sake of readability.

$$\mathbf{O}[n][k][h][w] = \sum_c \sum_r \sum_s \mathbf{I}[n][c][h+r][w+s] \times \mathbf{F}[k][c][r][s] \tag{14.7}$$

We can alternatively write this equation as a piece of code with 7 nested loops (see Listing 14.1).

Listing 14.1: 2D Convolution

```
for (n=0; n<N; n++) {  /* input images */
  for (k=0; k<K; k++) { /* outputs */
    for (c=0; c<C; c++) { /* input channels */
      for (h=0; h<H; h++) { /* rows of the ifmap/ofmap */
        for (w=0; w<W; w++) { /* cols of the ifmap/ofmap */
          for (r=0; r<R; r++) { /* rows of the filter */
            for (s=0; s<S; s++){ /* cols of the filter */
              O[n][k][h][w] += I[n][c][h+r][w+s]
                             * F[k][c][r][s];
}}}}}}}
```

The code in Listing 14.1 has several important features, which are as follows.

1. The order of the loops does not matter. Since there are no dependences between the loop variables, we can reorder the loops in any way we wish. We will see that this has important implications when it comes to cache locality.

2. For each output pixel, we perform $C \times R \times S$ multiplications and the same number of additions. We thus perform a total of $2C \times R \times S$ operations. Most ISAs provide a multiply-and-accumulate operation (MAC) that performs a computation of the form $a \mathrel{+}= b \times c$ similar to what we are doing. The number of operations in such algorithms is typically mentioned in terms of the number of MAC operations. We thus observe that per output pixel we perform $C \times R \times S$ MAC operations where we add partial sums to the output pixel (initialized to 0). Here, the product $I[n][c][h+r][w+s] \times F[k][c][r][s]$ is referred to as a *partial sum*.

3. We are essentially defining a 7-dimensional space where the dimensions are independent. This space can be *tiled* – broken down into subspaces. Let us explain with an example. Assume that we change the increment for the loop iterators w and h from 1 to 3. It means that we are considering 3×3 tiles of output pixels. Then we need to add two inner loops that traverse each tile (3×3 space) and compute the corresponding partial sums and add them to obtain the *ofmaps*.

4. Given that the computations are independent, we have a great opportunity to run this code on a set of parallel processors, where each processor is given a fixed amount of work. This also naturally fits with our notion of tiling the loop where a small amount of work can be given to each processor. Furthermore, since these processors only have to perform MAC operations and iterate through a loop, we do not need regular processors. Akin to a GPU, we can create an array of very small and simple processors. Let us call such a small and simple processor as a processing element or a PE.

A Reference Architecture

Let us think about this problem from the point of view of software first. We can reorder and tile loops, and moreover also embed directives in the code to run all the iterations of a loop in parallel – map each iteration to a separate PE or a separate group of PEs.

A high level reference architecture is presented in Figure 14.5 that will allow us to achieve these objectives. Such an architecture typically has a 1D array or a 2D matrix of PEs, some local storage in each PE (akin to the L1 cache), a large L2 cache, and an off-chip main memory (see Figure 14.5). In the figure, the local buffer (LB) in each PE is analogous to the L1 cache and the global buffer (GB) is analogous to the L2 cache. The PEs are interconnected with an NoC. Note that the small, filled circles represent the connections between wires. The horizontal and vertical links are not connected in this figure (no filled circles at the intersections).

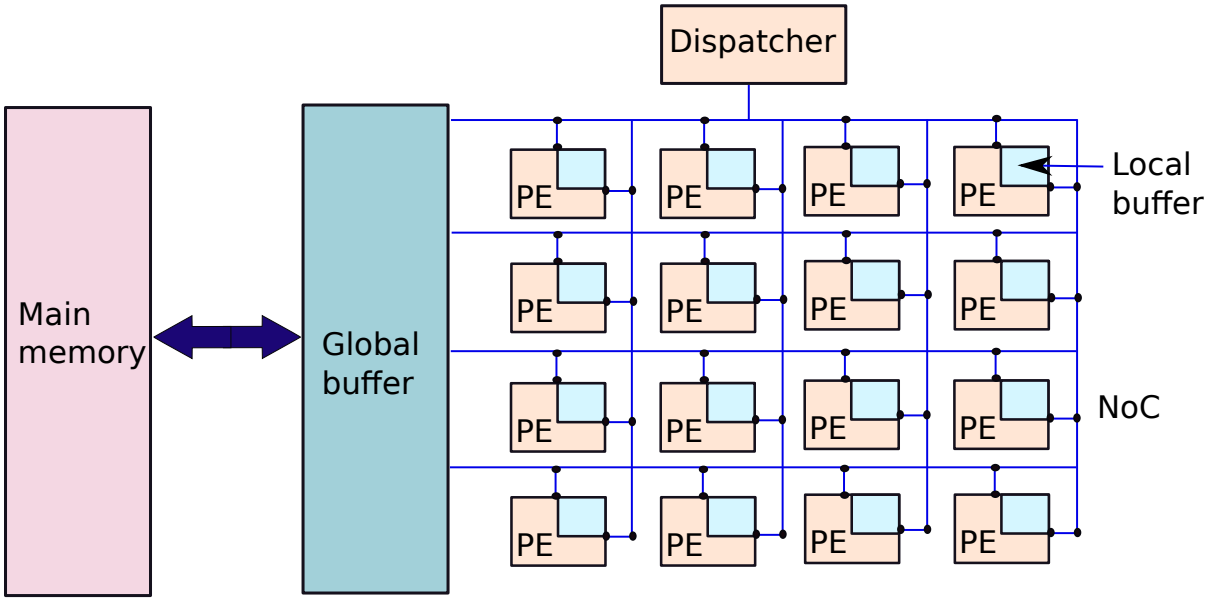

Figure 14.5: Reference architecture with a matrix of PEs

From the point of view of software we need at least one complex processor that we refer to as the *dispatcher*. It controls and orchestrates the entire computation. This includes dividing the work among the PEs, sending fetch and prefetch commands to memory, and moving outputs from the global buffer to the main memory. We can reorder, tile, and parallelize the loops in numerous ways. There are implications in terms of data locality in the GB and LBs. Additionally, we need to consider the overhead of moving data over the NoC, sending multicast messages, and computing the final output value for a pixel after adding the partial sums. The latter operation is also known as *reduction*.

To understand the space of loop transformations, let us describe a formal representation to represent a transformation. It will be easy to understand different optimizations subsequently.

Formal Representation of the Nested Loops

For the loops shown in Listing 14.1, let us represent them by the notation $n \triangleright k \triangleright c \triangleright h \triangleright w \triangleright r \triangleright s$, which represents the temporal order of loops. Let us refer to this as a *mapping* because we are in essence mapping the computations to PEs. In this case it is a single PE; however, very soon we shall introduce directives to parallelize this computation. Here, the operator $\triangleright$ indicates a temporal relationship. The loops on the right-hand side are strictly nested within the loops on the left-hand side. In the context of

this notation, it means that we process one input image at a time. Then for an input image, we process the *ofmaps* one after the other, and so on.

Let us introduce another operator to denote the possible parallel execution of a loop (distributed across the PEs) with the symbol $\|$, where the notation $n\|$ means that we can process all the N input images in parallel. It does not mean that we necessarily have adequate hardware to actually run all the iterations in parallel; it just says that it is possible to do so if we have enough hardware. If we do not have enough hardware then each PE needs to run several iterations. Furthermore, if we have two consecutive $\|$ symbols then it means that both the corresponding loops run in parallel. For example $h\|w\|$ means that each (row,column) pair is processed in parallel. We can also enclose it in square brackets for readability such as $[h\|w\|]$. This notation denotes a single level of parallelism. We create $H \times W$ parallel copies of loops and map them to the PEs.

We might however wish to implement hierarchical parallelism. This means that we might first want to parallelize the loop with iterator h, map each iteration to a group of PEs, and then assign one PE in each group to an iteration of w. This would be represented by $h\| \triangleright w\|$. This provides a structure to the parallel execution. The rule of thumb here is that $\triangleright$ represents a sequential order and $\|$ represents parallelism.

There is an important point to note here. The $\|$ operator indicates that we "can parallelize a loop" by distributing its iterations among PEs. Assume a case where the loop has 1024 iterations, and we just have 256 PEs. In this case, we can run the first 256 iterations in parallel, then schedule the next 256 iterations and so on. The way that we interpret the $\|$ operator is that the loop corresponding to the loop iterator preceding the operator can be executed in parallel if we have enough PEs. However, if we do not have enough PEs then the execution will be a hybrid of parallel and sequential execution. In this case, we say that the execution is *folded* (as we just described). Our representation is conceptual and conveys the perspective of a software writer.

Example 15
What is the difference between $[h\| \triangleright n \triangleright w\|]$ *(*S_1*) and* $[h\| \triangleright w\|n\triangleright]$ *(*S_2*)?*

Answer: *In both* S_1 *and* S_2*, we partition the iteration space for* h*, and assign each iteration to a group of PEs. However, the difference arises after that. In* S_1 *we consider an input image and process its pixels in parallel. An image is loaded only once. After that we process its pixels by dividing the values of* w *across the PEs in each group. Even if the execution is folded, the image is nevertheless loaded only once. However, in* S_2 *if* W *is more than the number of PEs in a group, then we need to cycle through the images. This means that if the execution is folded, then we need to load an image into memory many times. Note that if the execution is not folded, then both the representations are equivalent from the point of view of loading the input images (iterator:* n*).*

Now consider loop tiling (see Section 7.4.5). Let's say we decide to tile the loops with the iterators h and w, then we have to create two loops per iterator – one where we increment h by the tile size T_h, and an inner loop where we iterate in the range $[h, h + T_h - 1]$. We introduce two variables h' and h'': h' is incremented by the tile size T_h (outer loop), and h'' (inner loop) is a temporary variable that is used to iterate through the tile – it is incremented by 1 in each iteration. Now if we decide to tile the loops corresponding to the iterators h and w, a possible mapping can be $n \triangleright k \triangleright c \triangleright h' \triangleright w' \triangleright h'' \triangleright w'' \triangleright r \triangleright s$. One advantage of doing this is that we can have a multitude of parallel processing elements (PEs), where each PE can compute the results for a tile represented by the computation $h'' \triangleright w'' \triangleright r \triangleright s$. To realize this, we can run the outer loops for the iterators h' and w' in parallel. This will leverage the effect of locality in the LBs (local buffers). The mapping will thus be $n \triangleright k \triangleright c \triangleright h'\|w'\|h'' \triangleright w'' \triangleright r \triangleright s$. Note the parallel execution of the tiles.

In this case, we run H/T_h parallel instances of the loop for h' and W/T_w parallel instances for the

loop for w'. We can visualize this as running $\frac{HW}{T_h T_w}$ parallel copies of the tile.

Way Point 15

- *We can reorder loop iterations. Different reorderings have different implications in terms of the temporal locality of data in the LBs and GB.*
- *The $\|$ operator indicates that we can parallelize the iterations of a loop across the PEs. If the number of iterations is more than the number of PEs, then we need to fold the loop – each PE runs multiple iterations.*
- *The $\triangleright$ operator determines the order of nesting of the loops. If we write $n \triangleright k$, then it means that for each value of n, we consider all the values of k.*

Software Model

Our formalism for describing the nested loops predominantly captures the control flow of the CNN program, and the nature of parallelism. For the same control flow we can have different types of data flow. For example, we can cache some data in the local buffer (LB) of the PE, move some data from the GB to the PEs, and also move some data values between PEs. To capture the intricacies of the data flow we need to unnecessarily complicate our model. Hence, to a large extent we shall avoid doing so.

The only extension that we propose is to encompass some loops in a shaded box to indicate that the corresponding data is cached locally within a PE. For example the mapping – $n \triangleright k \triangleright c \| h \triangleright w \triangleright$ $r \triangleright s$ – indicates that the entire filter ($R \times S$ elements) is cached locally within a PE. Think of this as a software *hint* given to the hardware asking it to cache certain types of data. This is an example of *temporal reuse* where a given piece of data is cached within a PE.

The way that we shall interpret such mappings is as follows. Each PE runs a thread that computes partial sums according to the mapping. We split the iteration space among the threads, and each PE executes a set of threads (iterations) *mapped* to it. Secondly, we assume very coarse grained synchronization among the threads. For the mapping $n \triangleright k \triangleright c \| h \triangleright w \triangleright$ $r \triangleright s$ where we parallelize the loops on the basis of input channels, after every iteration of k, we insert a barrier (see Section 6.3.2 for a definition). This means that after we have processed all the channels for a given value of k, we encounter a barrier (a synchronization point for all the threads). The threads otherwise need not run in lockstep, and we assume that they do not suffer from data races while computing the values of output pixels.

14.2.2 Design Space of Loop Transformations

In this section, we shall look at different methods of reordering and tiling the loops to maximize locality at each PE. Moreover, we shall also like to leverage the fact that we have a parallel array of PEs. The aim is to consider different types of architectures where we keep different types of data cached in each PE. Caching data in local buffers is known as *stationarity*. We shall discuss a few reference designs in this section, and it should be kept in mind that the designs are suggestive in nature. More efficient designs are possible.

The key idea behind stationarity of any data is to use it as **frequently as possible** when it is resident in the local cache, and then **minimize** the number of times the data needs to be reloaded into the cache. This means that the stationary data should be reused as much as possible and then **replaced**.

Note that it is possible to design architectures where there is no stationarity. Such architectures are known as No Local Reuse (NLR) architectures.

Weight Stationary (WS) Architecture

Let us now look at a few of the common ways in which the basic 7-loop structure can be transformed. Consider our example from the previous section: $n \triangleright k \triangleright c \triangleright h' \| w' \| h'' \triangleright w'' \triangleright r \triangleright s$. This means that we compute the results for each *ifmap-ofmap* pair and image sequentially. Next, we divide an *ofmap* (and corresponding *ifmap*) into small tiles and distribute them across the 2D PE array. Each PE computes the output pixels for each tile. It reads the corresponding inputs, and repeatedly accesses the filter weights. In this case, it makes sense to cache the filter weights for a given *ifmap-ofmap* pair in each PE. This is an example of *filter reuse* or *weight reuse*. We can keep the filter weights *stationary* at each PE and this will ensure that we only need to stream the inputs and outputs to and from each PE. This is an example of a *weight stationary* or *WS* architecture.

Let us make the mapping more efficient. The aim is to reuse the filter weights as much as possible. This means that we need to load them once, finish all the computations that require them, and then load the next filter. This is equivalent to maximizing the distance between the iterations that change the filter; this also maximizes the number of iterations that use the filter in the mapping. One such mapping is as follows.

$$k \triangleright c \triangleright n \triangleright h' \| w' \| h'' \triangleright w'' \triangleright \boxed{r \triangleright s} \tag{14.8}$$

The parameters k and c change the filter, and thus they are the iterators of the outermost loops. The loops traversing the filter are the innermost. Note the shaded box $\boxed{r \triangleright s}$. In this case it indicates that the entire filter (dimensions $R \times S$) is cached within a PE.

Let us analyses the execution described in Equation 14.8. We cache the entire filter (dimensions: $R \times S$) in the LBs of the PEs. The filter depends on k and c, which are incremented in the two outermost loops. Each PE is assigned a tile ($h'' \triangleright w''$). At runtime, each PE can read the part of the *ifmap* that corresponds to its tile, and keep computing the convolution. Note that to compute the partial sums for a tile with dimensions $T_h \times T_w$, we actually need to read $(T_h + R - 1) \times (T_w + S - 1)$ input pixels. We assume that all of these reads are performed by the system. For the sake of simplicity, we shall not go into the details now. Finally, when there is a change in the iterators c or k, each PE reads the filters corresponding to the new values of c and k.

This mapping is inefficient. This is because even though we are maximizing filter reuse, computing the outputs is slow. For the same (c, k) combination we first compute all the partial sums for all the images, and then move to a new value of (c, k). Recall that every time we compute a partial sum we need to read the relevant entry from the *ofmap*, add the partial sum to it, and write it back. The next time we reuse the output is after we have processed entries from all the images. Processing *ofmaps* thus has very little temporal locality. Furthermore, we also need a lot of space to store all of these partial sums (across all the images).

Let us instead create a different mapping (see Equation 14.9).

$$n \triangleright k \| c \| h \triangleright w \triangleright \boxed{r \triangleright s} \tag{14.9}$$

In this case we also cache the filter weights. However, we allocate work to the PEs differently. Each PE is responsible for a (c, k) (*ifmap-ofmap*) combination. For this combination, the filter remains the same. Then we process all the elements of an *ifmap* to generate all the partial sums. Given that we aggregate the partial sums, it is best to process an entire image in one go before loading the next image. Note that in general, it is a good idea to finish the computations for one large image before loading the next one from the point of view of locality and the space required to store the partial sums. Hence, we set the loop that increments n to be the outermost loop. This design is frequently used. We learned an important lesson.

> Trying to maximize the locality of one block of data can adversely impact the locality of another block of data and the space required to store temporary results. Often there is a

trade-off. We would like to opt for a balanced choice that maximizes the performance of the system as a whole.

Input Stationary (IS) Architecture

Consider another kind of mapping where we distribute parts of the *ifmaps* (inputs) to every PE, and keep them stationary in the LB there. We tile the loops with iterators h and w. We thus break the *ifmap* into tiles of dimensions $(T_h + R - 1) \times (T_w + S - 1)$. Given that there are C input channels, we can store C such tiles in every PE assuming we have the space for them. Note that in this case, the tiles stored across the PEs have an overlap. This had to be done to ensure that we can efficiently compute the convolutions for pixels at the right and bottom edges of each tile. Otherwise, we need to communicate values between the tiles. Such pixels at the bottom and right edges of the tile for which we need to store or communicate extra information are known as *halo pixels*.

In this case, the mapping is as follows for such an *input stationary* or *IS* architecture.

$$n \triangleright h' \| w' \| k \triangleright r \triangleright s \triangleright \boxed{c \triangleright h'' \triangleright w''} \tag{14.10}$$

The input *ifmaps* are stationary. In each PE we store C *ifmap* tiles. Each PE reads the relevant filter weights and computes the corresponding partial sums. Finally, it adds the partial sums for the corresponding output tile and computes the output; then it moves to the next image. This limits the number of partial sums that need to be stored. Given that a PE stores tiles for all the C channels, it can compute all the convolutions locally. There is no need to handle halo pixels in a special manner.

Output Stationary (OS) Architecture

On similar lines we can define an *output stationary* or *OS* architecture. Here we distribute the output pixels across the PEs. They read the relevant inputs and filter weights, and then compute the partial sums.

$$n \triangleright h' \| w' \| c \triangleright r \triangleright s \triangleright \boxed{k \triangleright h'' \triangleright w''} \tag{14.11}$$

Row Stationary (RS) Architecture

We can alternatively distribute rows of the *ifmap* and the filter across the PEs. They can compute the relevant partial sums. For a given *ofmap* row one of the PEs can take up the role of aggregating the partial sums. The rest of the PEs that have computed partial sums for that *ofmap* row, albeit with different filter weights and *ifmap* rows, can send the partial sums to the aggregating PE. A possible mapping is as follows.

$$n \triangleright k \triangleright c \triangleright h' \| r' \| w \triangleright s \triangleright \boxed{h'' \triangleright r''} \tag{14.12}$$

Important Point 22
The important point to note here is that there are many ways of creating stationary architectures. Even for a given type of architecture, there are many ways of distributing the computations and organizing the data flow. The examples given in this section were of a generic and simplistic nature. Let us now create data flow mechanisms for such architectures.

14.2.3 Hardware Architectures

In Section 14.2.2 we described loop transformations from the point of view of a generic software model. We only described temporal reuse, where we keep some data stationary in the PEs such that it can be used by later computations. However, for building an efficient hardware implementation, we also need to consider efficient *spatial reuse*, which means that we read a block of data once from memory and try to reuse it as much as possible. For example, this can be done by multicasting it to a set of PEs. The PEs do not have to issue separate reads to memory. Alternatively, data values can flow from one PE to the next along the same row or column. Again this reduces the memory bandwidth requirement. Any hardware implementation has to consider such kinds of spatial reuse to reduce the pressure on the global buffer.

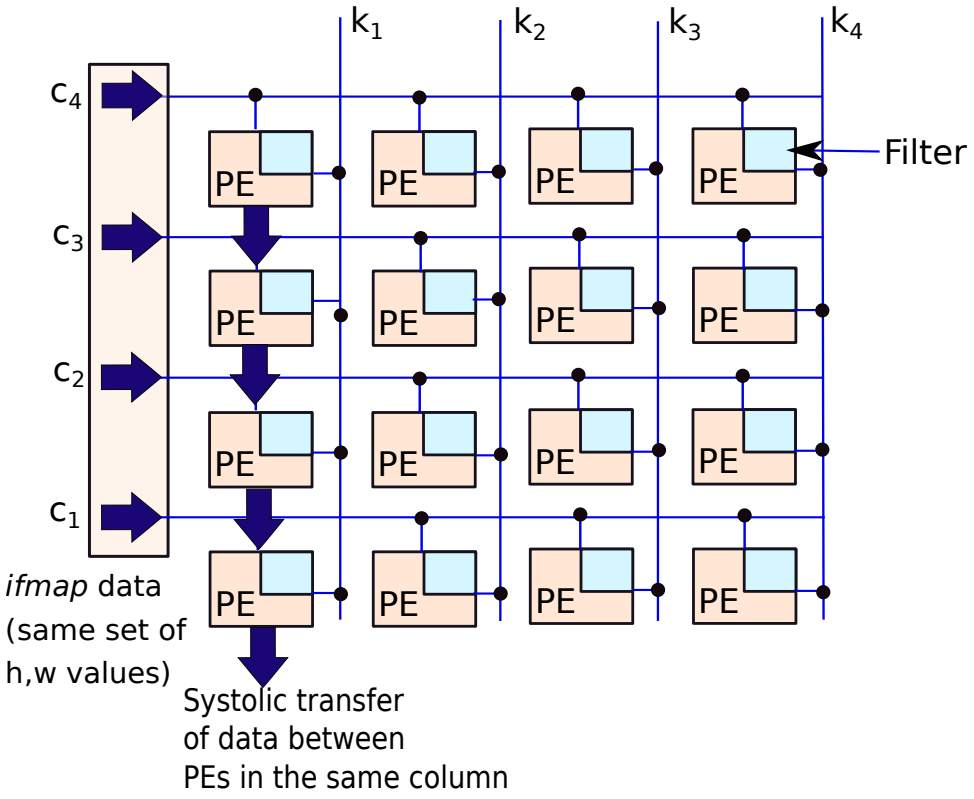

Figure 14.6: A weight stationary architecture

Weight Stationary Architecture

Consider the following mapping that we derived in Section 14.2.2. We need to realize this on a 2D array of PEs, which is the most generic architecture in this space. Each PE has a filter stored for an *ofmap-ifmap* (k, c) pair.

$$n \triangleright k \| c \| h \triangleright w \triangleright r \triangleright s$$

For a 2D-matrix of PEs, we need to structure the matrix in such a way that it allows us to aggregate the outputs. We thus assign the rows to channels, and the columns to *ofmaps*. The mapping thus

becomes,

$$n \triangleright k\| \triangleright c\| \triangleright h \triangleright w \triangleright \boxed{r \triangleright s} \tag{14.13}$$

The steps are as follows (keep referring to Figure 14.6). In Figure 14.6, there are no connections between the vertical and horizontal wires. Note that there are no connection symbols (filled dark circles) at their intersections. For systolic transfer between the PEs, the vertical links are used. Note that the arrows between PEs (denoting systolic transfer) are conceptual: they are only showing the direction of the flow of data.

Phase I First for a given (k, c) pair we need to load the filter weights in the PEs. Each PE can issue reads to the GB via the NoC. We arrange the filters as follows. Each row of the 2D-array corresponds to one channel $(c_1 \ldots c_4)$, and each column of this array corresponds to a given *ofmap* (total of K $(= 4)$ such *ofmaps*). We thus have C rows and K columns.

Phase II For each channel, we send a block of values from the corresponding *ifmap* along the rows. We have a choice here, either we can send data byte by byte, pixel by pixel, or as a tile of pixels. Normally the last approach (a tile of pixels) is preferred. In this case, we send a tile of pixels for each channel. The important point to note is that all the tiles have the same coordinates (same h and w values).

Phase III Each PE computes the convolutions between the input data and the filter data.

Phase IV Note that all the partial sums computed in each column need to be added to get the value of the corresponding output pixels. We need to sum up the values column-wise. This can be done in two ways. We can either have a tree of adders at the end of each column. All the values can be sent to the adder tree via the NoC. This is known as *parallel reduction*. The other option is to opt for a *systolic transfer*. A PE in the highest row transfers its partial sums to the PE below it (in the same column). This PE adds the received partial sums with the partial sums it has computed, and transfers the result to the PE below. This process continues, and finally the result leaves the PE array via the last row of PEs (bottom row in Figure 14.6).

Input Stationary Architecture

We derived the following mapping in Section 14.2.2.

$$n \triangleright h'\|w'\|k \triangleright r \triangleright s \triangleright \boxed{c \triangleright h'' \triangleright w''}$$

The aim was to map C input tiles with the same coordinates to a PE. We just need to stream all the filters and the PE can compute all the K outputs. These outputs can then be streamed out via the vertical links. This is shown in Figure 14.7.

Note that we are running the PEs independently in this case. We can do better. We can parallelize the computations differently. Consider the following mapping where we split the channels for each *ifmap* tile across the rows.

$$n \triangleright h'\|w'\| \triangleright c\|k \triangleright r \triangleright s \triangleright \boxed{h'' \triangleright w''} \tag{14.14}$$

The corresponding data flow is shown in Figure 14.8. In each column we store the inputs for the same *ofmap* tile: one *ifmap* tile per channel. Then we send the filter weights down the rows. For channel c and *ofmap* k, we send $\mathbf{F}[k][c][\ldots]$ on the row corresponding to channel c. The PEs compute a convolution between the filter weights and the stored input tile to get a set of partial sums. Note that all the PEs in each column compute partial sums for the same *ofmap* tile; hence, all that we have to do is send the partial sums down the column. Each PE adds its computed partial sums to the values it receives from its neighbor above it. It passes the result to the PE below it. This is an example of a *systolic transfer*.

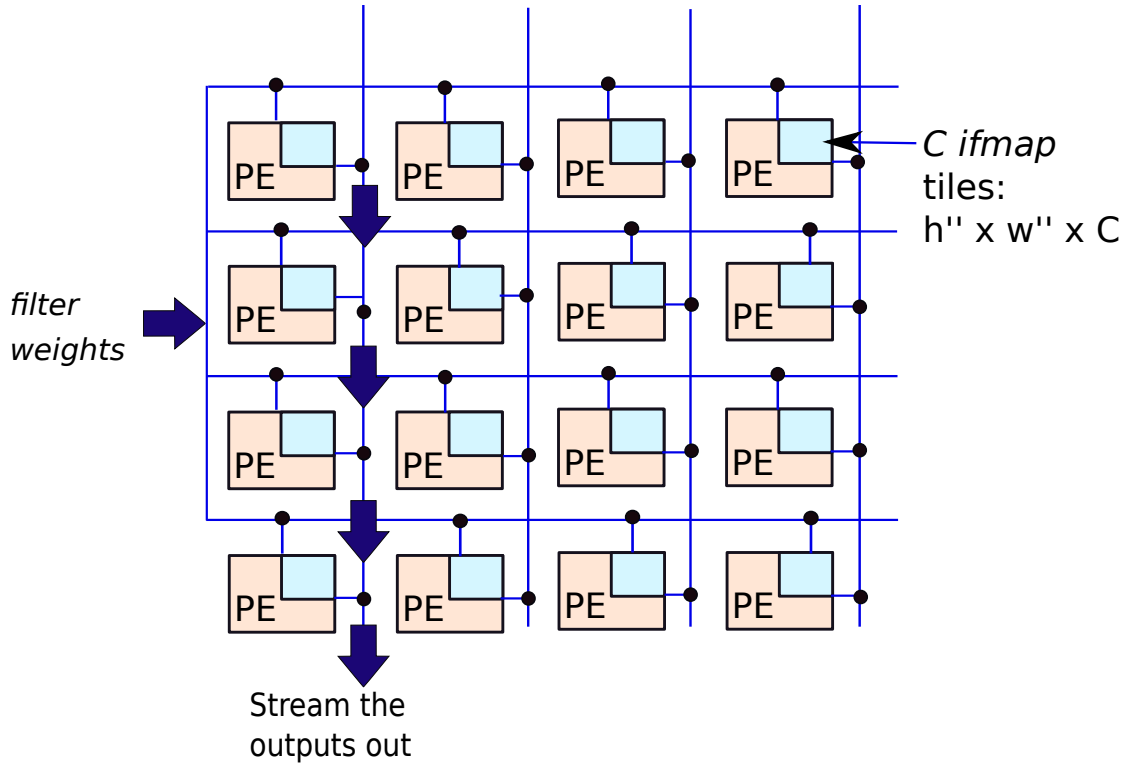

Figure 14.7: An input stationary architecture. The thick dark arrows represent the conceptual flow of data from one PE to the one below it (via vertical links).

Output Stationary Architecture

Let us consider the order of loops that we derived for a software-only implementation. It was as follows.

$$n \triangleright h' \| w' \| c \triangleright r \triangleright s \triangleright \boxed{k \triangleright h'' \triangleright w''} \tag{14.15}$$

The idea here was simple, we simply parallelize the computation by assigning a set of output pixels to each PE. The PE computes the output pixels for a $T_h \times T_w$ tile for all the *ofmaps*. It does so by reading the corresponding input pixels and filters. This is indeed an effective mechanism; however implementing this scheme in hardware is difficult. The memory bandwidth requirements will be high. We ideally want to read some data and reuse it as much as possible by multicasting it to a set of PEs. This is what we have done in the WS and IS architectures. Let us change the mapping to enable more spatial reuse.

If we compute $k \triangleright h'' \triangleright w''$ in a PE, it means that only that PE will get the inputs corresponding to the $h'' \triangleright w''$ tile. This cuts the opportunities for spatial reuse of *ifmap* input data. Let us thus organize the computation differently. Let us assign all the PEs in a column to an *ofmap* (refer to Figure 14.9).

Consider a single row first. In the new organization, the order of computation will look like $k \| \boxed{h'' \triangleright w''}$ for a single row. All the PEs in the row share the input tile but produce output pixels for different *ofmaps*. We can cache a tile for a single *ofmap* in each PE in the column. Next, we stream the inputs corresponding to the tile $h'' \triangleright w''$ along the row for all the input channels one after the other (horizontal direction). At the same time, we stream the filters along the columns. The mapping thus becomes $k \| c \triangleright r \triangleright s \triangleright \boxed{h'' \triangleright w''}$.

Each PE reads the corresponding input tile for channel c and the filter for the (k, c) pair, and then computes the partial sums. Finally, across the rows we need to stream a set of input tiles. For example,

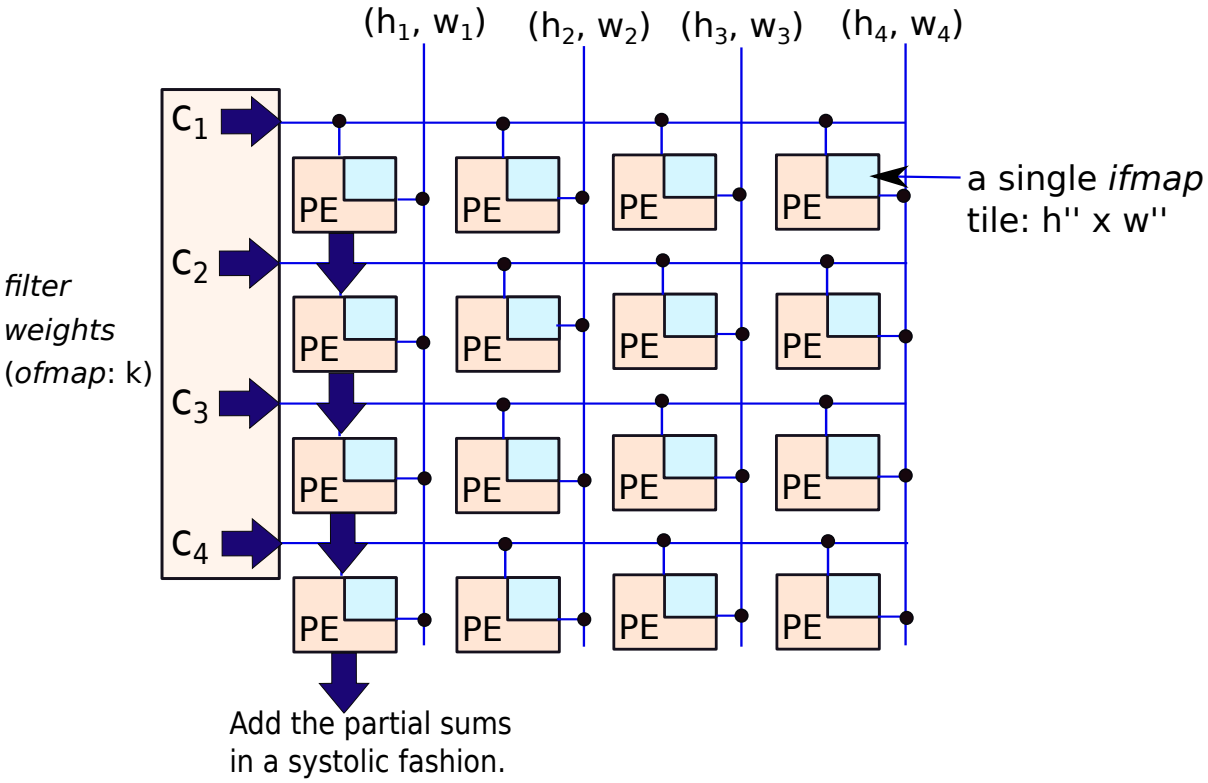

Figure 14.8: An input stationary architecture that performs a column-wise reduction. The thick dark arrows represent the conceptual flow of data from one PE to the one below it (via vertical links).

if we have 10 rows, then we can stream 10 different input tiles along these rows. The final mapping is thus as follows.

$$n \triangleright h' \| w' \| \triangleright k \| c \triangleright r \triangleright s \triangleright \boxed{h'' \triangleright w''} \tag{14.16}$$

Figure 14.9 realizes this mapping. The key idea can be summarized as follows. Each column corresponds to a specific output channel (*ofmap*). Each row corresponds to an input/output tile. We transmit input tiles along the rows and filter data along the columns corresponding to the respective *ofmaps*. At each PE assigned to *ofmap* k, *ifmap* data for channel c and filter data for the (k, c) pair arrive simultaneously; it subsequently computes the output pixels for a single tile in an *ofmap*. At the end of a round of computations, the output pixels are read out through the columns.

Row Stationary Architecture

Let us create a row stationary or RS architecture where we keep rows of the inputs and the filters stationary. We derived the following mapping in Section 14.2.2.

$$n \triangleright k \triangleright c \triangleright h' \| r' \| w \triangleright s \triangleright \boxed{h'' \triangleright r''}$$

Consider a 3×3 filter, and a tile size of 1 for the *ifmap* and filter rows. It means that for computing an output pixel in the context of a single channel, we need to consider three rows. These rows are cached in different PEs. We need to read the partial sums that they have computed and aggregate them. We

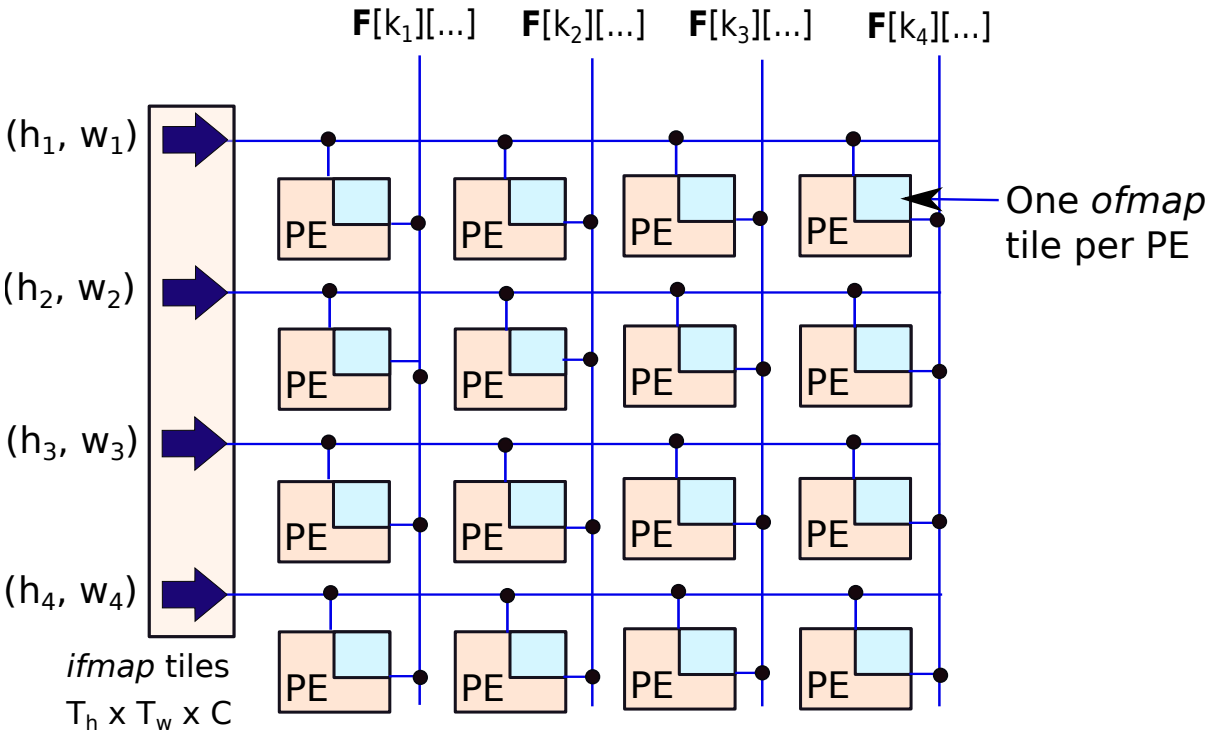

Figure 14.9: An output stationary architecture

thus need to organize pairs of *ifmap* and filter rows such that it is possible to do so very easily. One of the representative organizations (refer to [Chen et al., 2016]) is as follows.

Let us number the rows in the *ifmap* $h_1, h_2, h_3, \ldots$, and the rows in a filter $r_1, r_2, r_3, \ldots$. Figure 14.10 shows a simplified representation of a row stationary (RS) data flow for a filter with 3 rows. All the PEs in each row keep a row of the filter stationary. We then distribute rows of the input among the PEs such that each column computes the partial sums corresponding to the top row. For example, in the first column we store the *ifmap* rows h_1, h_2, and h_3. We compute the convolutions $h_1 \star r_1$, $h_2 \star r_2$, and $h_3 \star r_3$ in the first column. We need to aggregate the partial sums by transferring them down each column to compute the aggregated partial sums for all the output pixels that correspond to row h_1.

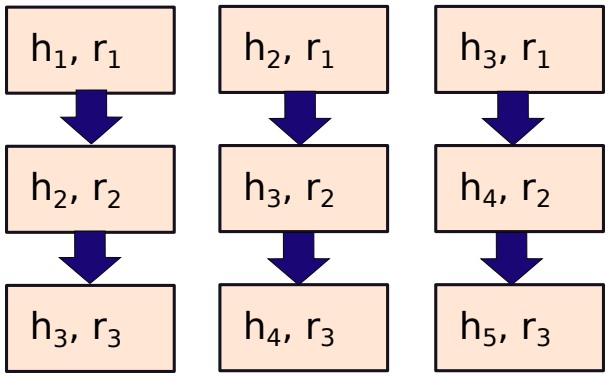

Figure 14.10: A row stationary data flow

The stored *ifmap* rows can be represented by a matrix as follows.

$$\begin{bmatrix} h_1 & h_2 & h_3 \\ h_2 & h_3 & h_4 \\ h_3 & h_4 & h_5 \end{bmatrix}$$

In the next round, the filter weights remain the same, we just change the *ifmap* rows to the following matrix.

$$\begin{bmatrix} h_4 & h_5 & h_6 \\ h_5 & h_6 & h_7 \\ h_6 & h_7 & h_8 \end{bmatrix}$$

Note that this computation is happening for only one input channel. To compute the final values of the output pixels, we need to add the partial sums across all the channels. An issue with this design is that the filter, which is kept stationary, also determines the *ifmap-ofmap* pair. We cannot change the input channel without changing the cached filter rows. Hence, we propose to partition the 2D matrix of PEs into 2D blocks. The partitioning will be based on the input channel c and the output *ofmap* k. For each (c, k) pair, we implement an RS data flow. Then for each row of output pixels, we simply need to add the partial sums computed by each group of PEs. This can be done either by storing the partial sums in a dedicated set of buffers, or by reading them from memory. Hence, the mapping for a full RS system will look something like this.

$$n \triangleright k \| c \| \triangleright h' \| r' \| w \triangleright s \triangleright h'' \triangleright r'' \tag{14.17}$$

Let us critique this design. There is clearly a degree of redundancy. We store multiple copies of filters, and multiple copies of input rows. Additionally, for multiple *ifmap-ofmap* pairs, we need to create 2D blocks of PEs. The IS, WS, and OS architectures were comparatively more elegant. However, in those architectures, we were computing 2D convolutions within each PE, and in this RS architecture we need to compute only a 1D convolution, which is comparatively far easier to compute. Secondly, whenever we have more redundancy, we have more flexibility. In this case, there is no information exchange between the columns of PEs; hence, they need not run in lockstep. Secondly we can achieve an overlap between adding partial sums and loading rows of PEs with new data. When the bottom row is adding partial sums, the top row of PEs can be getting loaded with new data (*ifmap*-filter row pairs).

No Local Reuse (NLR) Architecture

The NLR architecture does not allocate any local storage to a PE for caching data; all the inputs, filters, and partial sums are either read from the global buffers or passed between PEs in a systolic fashion. An advantage is that we can create architectures with a large number of PEs; however, to compensate for this we need a large memory that can quickly supply data to the PEs via a very efficient NoC. An example of such an architecture with a 1D array of PEs is shown in Figure 14.11. In this case, the parallelization is across the channels. Each PE computes the partial sums for each channel; each such partial sum is sent to the neighboring PE on the right. We can parallelize along other axes also, and also create a 2D array of PEs by juxtaposing such 1D arrays.

14.3 Intra-PE Parallelism

Up till now we have been discussing parallelism at the level of PEs. We observed that it is a good idea to divide large chunks of computations amongst the PEs such that they can work independently. We created rather loose synchronization mechanisms across the PEs. This allowed us to scale the design without relying on very tight clock synchrony. As we shall see later, such design choices will allow us to make use of different optimizations such as using 0-valued pixels to reduce the number of computations

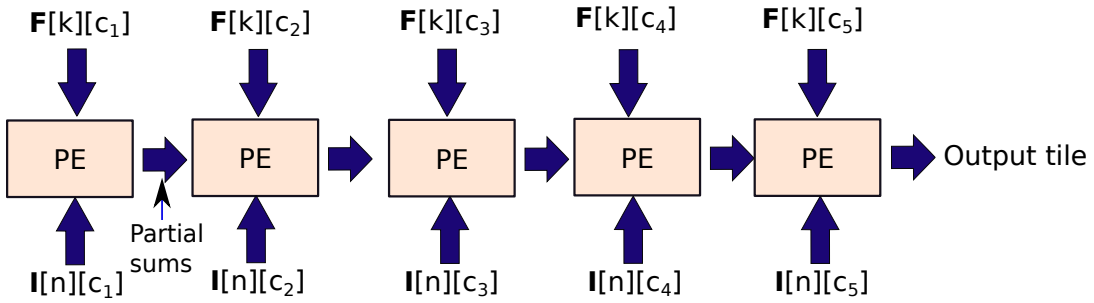

Figure 14.11: An NLR data flow

within the PEs. Let us now focus on the design of a PE. As far as a PE is concerned, it receives a set of inputs and filter weights. It needs to compute a 2D-convolution in most architectures, and a 1D convolution in the row stationary architecture.

Let us first focus on 1D convolution, which is simpler.

14.3.1 1D Convolution

Let us first introduce simpler terminology. We shall use the variables $\mathbf{w}$, $\mathbf{x}$, and $\mathbf{y}$ to represent the weight vector, a row of inputs, and a row of outputs respectively. Let the length of the weight vector be n. For referring to individual elements we shall use a subscript. For example, w_i refers to the i^{th} element of $\mathbf{w}$. The convolution operation that we wish to compute is of the form,

$$\begin{aligned} &\mathbf{y} = \mathbf{w} \star \mathbf{x} \\ \Rightarrow &\forall i,\ y_i = \sum_{j=0}^{n-1} w_j \times x_{i+j} \end{aligned} \tag{14.18}$$

Such circuits contain only two kinds of basic elements: logic elements for computing the result of addition and MAC operations, and registers. Let us explain with some examples. We shall first introduce semi-systolic arrays in this context. Please note that in the diagrams shown in the subsequent sections, we start the count from 1 for the sake of simplicity. All the definitions remain the same; there is a slight abuse of notation.

Semi-systolic Arrays

Stationary Weights

Consider the design shown in Figure 14.12. In this case, we have four weights: w_1, w_2, w_3, and w_4. They are kept stationary in four registers respectively. Then inputs start flowing in from the left. The inputs are numbered $x_1, x_2, \ldots$. Consider time $t = 1$ (see the timeline on the right side). Each of the combinational elements computes a product. The products are $w_1 \times x_1$, $w_2 \times x_2$, $w_3 \times x_3$, and $w_4 \times x_4$. All of these products are computed in parallel. We then assume the existence of an adder tree that adds all these *partial sums* in a single cycle. If we now consider the timeline we see that in the first cycle we compute y_1, then we compute y_2, and so on. The computation can be visualized as a sewing machine where the cloth (representing the inputs) passes through the needle.

Every cycle each input moves one step to the right. Note the placement of a register between two multiply units. We assume that it takes an input one cycle to pass through a register. This pattern of

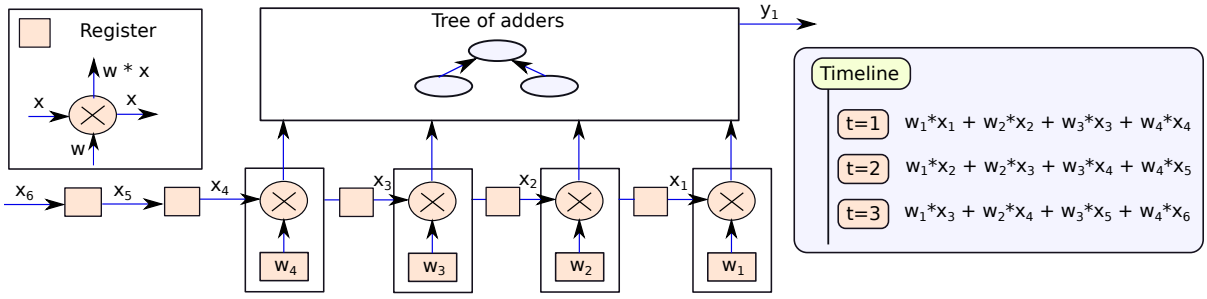

Figure 14.12: The weights remain stationary, the inputs pass through them, and the output is computed by an adder tree.

computation where in every cycle the inputs or partial sums move to a neighboring register is known as *systolic computation*. Such systolic circuits typically use an array of registers, MAC/multiply elements, and adders. We can have many kinds of systolic machines. This is an example of a semi-systolic array. Whenever two combinational elements are directly connected without an intervening register then we call this structure a semi-systolic array. In this case, there are no registers between each multiply element and the adder. However, if we have a register between every two combinational elements, then the structure is known as a *systolic array*.

Definition 105

- *An array is defined as a structured network of registers and combinational elements. In a systolic computation, values flow from one register to adjacent registers every cycle.*
- *In a semi-systolic array there need not be any intervening registers between adjacent combinational elements. Combinational elements A and B are said to be adjacent if from the output of one to the input of the other, there is a path comprising only of wires and registers.*
- *In a systolic array we always have a register between any two adjacent combinational elements.*

Semi-systolic arrays have their fair share of shortcomings. Let's say we have 25 weights, then we need to add all the 25 partial sums in a single cycle. This may not be possible. We need to unnecessarily slow down the cycle time. In such semi-systolic architectures a single cycle needs to accommodate a lot of Boolean computations, which may result in a slowdown. To convert this architecture into a systolic architecture, we can pipeline the adder tree by adding an array of registers at each level.

Now instead of keeping the weights stationary, let us create another semi-systolic architecture where we broadcast the weights. This avoids having the costly adder tree.

Broadcasting the Weights

In this case, we keep the outputs stationary, and broadcast the weights one after the other in successive cycles (refer to Figure 14.13). We compute a set of outputs in parallel ($y_1 \ldots y_6$). Consider the computation from the point of view of output y_1. In the first cycle, we broadcast w_1 to all the MAC units. We set $y_1 = w_1 \times x_1$. In the next cycle, the input x_2 arrives from the left, and simultaneously we broadcast w_2. The MAC unit then computes $y_1 = y_1 + w_2 \times x_2$. Likewise, in the next few cycles we compute and add the rest of the partial sums to y_1. At the end of 4 cycles, y_1 is correctly computed. In

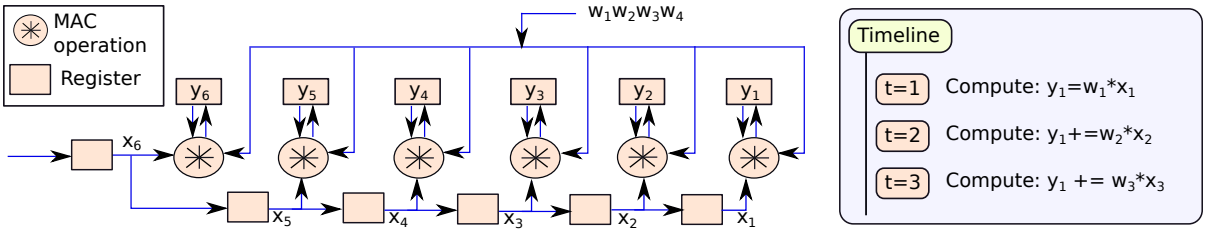

Figure 14.13: The weights are broadcast to all the MAC units

parallel, we would have computed the rest of the partial sums: $y_1 \ldots y_6$. All of them can be written to the output memory. We can then compute the next 6 outputs and so on. This is also an example of a systolic computation, but the array is semi-systolic because we are broadcasting the inputs – there are no registers between the inputs and the MAC units. Here again, the broadcast operation can be slow if we have a large number of weights. We need to take into account the RC delay of the wires and the input capacitance values of the registers.

We looked at two examples of performing 1D convolution using semi-systolic arrays in this section. Many more architectures are possible. In fact there is a lot of theoretical work on automatically converting a set of loops into a systolic computation [Lavenier et al., 1999, Lam, 2012]. Discussing this is beyond the scope of this book.

It is possible to automatically convert a semi-systolic array into a systolic array using the Retiming Lemma [Leighton, 2014]. Hence, many designers initially create a semi-systolic array for their problem because it is often easier to design it. Then they use the Retiming Lemma to automatically convert the architecture to one that uses a systolic array, which effectively bounds the amount of computation that needs to be done in a single clock cycle. This is because of the mandatory placement of registers between combinational units.

The Retiming Lemma

We will not explain this in great detail in this chapter. We will just present the basic idea. We represent the array as a graph $\mathcal{G}$, where each combinational element is a node, and if there is a connection between two combinational elements (even via some registers), we add an edge between them. The weight of each edge is equal to the number of registers it contains. If there are no registers on an edge, its weight is zero. We next compute $\mathcal{G} - 1$ by subtracting the weight of each edge by 1. Now, we present a few results without proofs.

1. For any node, we can move k registers from each of its input edges and add them to each of the output edges. The results will remain the same and the rest of the nodes will not perceive this change. This is known as *retiming*. This is a purely local operation.

2. If $\mathcal{G} - 1$ does not have any negative weight cycles, we can successfully *retime* it to produce an equivalent systolic array. If there are negative weight cycles, we typically slow down the array by a factor of k by multiplying the number of registers on each edge with k (we produce the graph $k\mathcal{G}$). For small values of k, $k\mathcal{G} - 1$ typically does not have a cycle.

3. We set the *lag* of a vertex/node as the weight of the shortest path from it to the output in $k\mathcal{G} - 1$. For an edge from u to v, we set the new weight as follows. $weight_{new} = weight_{old} + lag(v) - lag(u)$. The result of this operation is that we add extra registers to different edges – we delay different values flowing in the array to varying extents.

We introduced multiplicative slowdowns (to avoid negative weight cycles), and additive delays by adding more registers to edges. This additional timing overhead may be prohibitive in some cases,

and thus we may prefer a semi-systolic array if it is faster. This determination needs to be done on a case-by-case basis.

Systolic Array

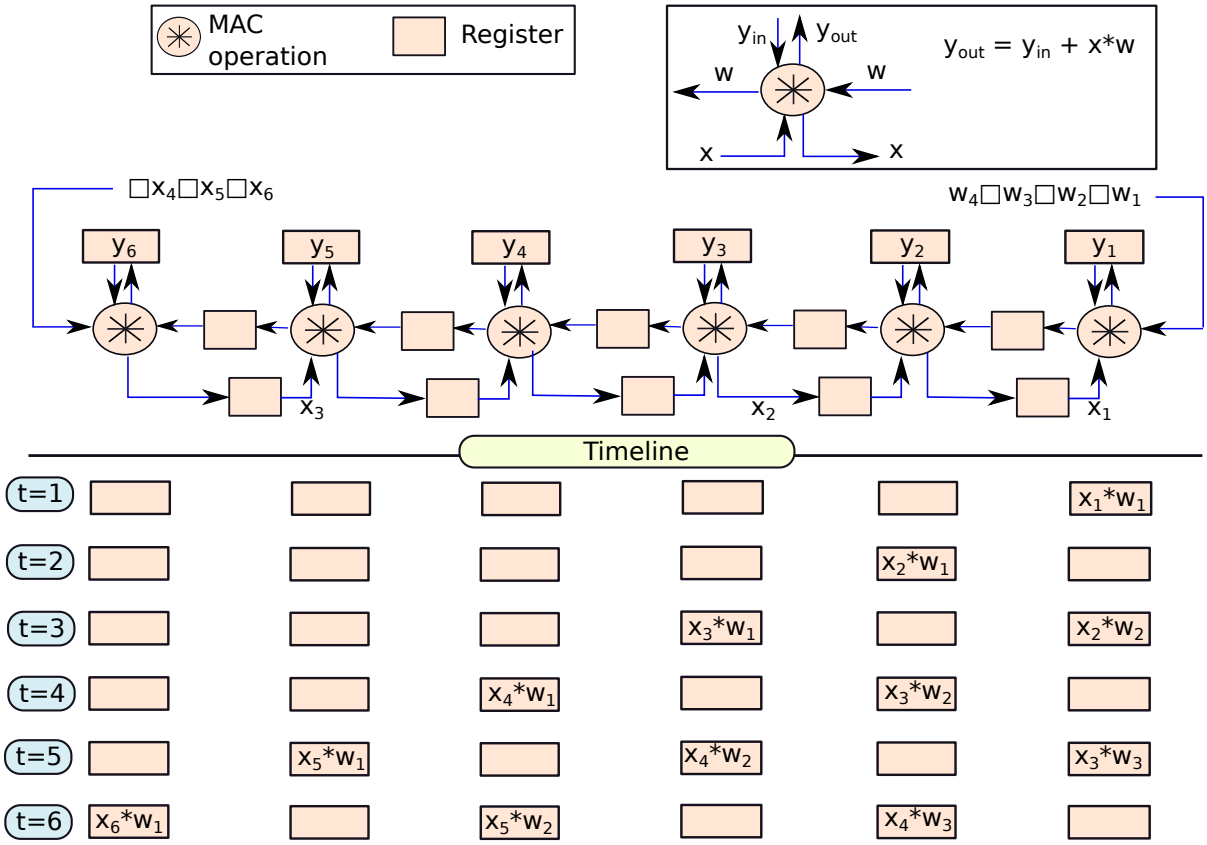

Figure 14.14: A systolic array for 1D convolution

Let us take the semi-systolic array presented in Figure 14.13 (architecture that broadcasts weights) and use the Retiming Lemma to create a systolic version of it. This is shown in Figure 14.14. First, note that it is indeed a systolic array because there is always a register between any two adjacent combinational elements. Second, we have slowed down the array by a factor of 2 in accordance with the Retiming Lemma (to eliminate negative weight cycles). Hence, the inputs and the weights are provided to the array every alternate cycle. In this context, the symbol □ represents an empty cycle – no inputs or weights are provided to the array in that cycle.

Now consider the structure of the systolic array. The inputs and weights traverse the array from opposite sides. The array is designed in such a way that whenever an input *collides* with a weight, it produces a partial sum and this sum is added to the value in output register y_i. The pictorial description at the bottom of the figure shows the timeline, where each rectangle corresponds to an output register (registers storing $y_1 \ldots y_6$). Here, we can see that a separation of one cycle between the inputs or weights is necessary. This is because if the previous input is two columns away, in the next cycle it will only be one column away, which is where it needs to be to compute the partial sum correctly. Using a similar argument we can reason about the spacing between the weights. The reader should convince herself about the correctness of this design.

Now, an important criticism of this design can be that we are halving the throughput. On the flip side, we are gaining regularity, we can most likely afford a much higher clock speed, and we are avoiding costly broadcasts. However, for all of these advantages, sacrificing 50% of the throughput does not seem to be justified. This problem can be solved very easily. We can compute two convolutions at the same time. The second convolution can be scheduled in the empty cycles. The different convolutions will not interfere with each other as long as we have separate output registers for each convolution. This is the conventional technique that is used to ensure that we do not sacrifice on throughput. This can easily be done in a CNN because we typically need to compute the results of a lot of convolutions.

Direct Product

If we are willing to use a lot of resources, then there is a simple and direct method. Instead of a traditional systolic style computing, we can instead directly compute the convolution as follows. Let us say that we want to convolve the 1D input vector x with the filter vector w that contains n elements. We load $x_1 \dots x_n$ into a buffer B_1, and simultaneously load $w_1 \dots w_n$ into another weight buffer W. At the same time, we load the elements $x_2, \dots, x_n, x_{n+1}$ into buffer B_2. We can use combinational logic to select the elements $x_2 \dots x_n$ while they are being loaded into B_1, shift them by one position, and separately load x_{n+1} from memory. We can do the same for more buffers such as B_3 and B_4 that start with x_3 and x_4, respectively. For this design we can have 4 PEs; each PE reads the contents of one input buffer and the weight buffer, computes the convolution, and writes the output to an output array. All the convolutions can be computed in parallel. Furthermore, each PE can have an array of MAC units and an adder tree.

This is a very simple algorithm and can easily be implemented on an FPGA. It does not require the sophistication or coordination that a systolic array requires. In terms of the resources required, we need more storage space because now a single input element is stored in multiple buffers, and we also need more hardware to handle data movement.

14.3.2 2D Convolution

Systolic Approaches

Let us now extend our results to perform a 2D convolution. The simplest approach is to divide it into a sequence of 1D convolutions, compute each convolution separately, and add the partial sums. Given that we have a fairly robust architecture for computing 1D convolutions, we can easily reuse it in a 2D scenario.

To describe this, let us use similar terminology where $\mathbf{X}$ represents the input. In this case, $\mathbf{X}$ is a 2D matrix. We shall use the term $row(x_i)$ to refer to the i^{th} row of $\mathbf{X}$. Similarly, we shall use the notation $row(w_i)$ to refer to the i^{th} row of the filter, and $row(y_i)$ to refer to the i^{th} row of the output.

To compute all the output pixels for the i^{th} row of the output *ofmap*, we need to compute the following convolutions: $row(w_i) \star row(x_i)$, $row(w_{i+1}) \star row(x_{i+1})$, ..., $row(w_{i+R-1}) \star row(x_{i+R-1})$. We are assuming that the filter has R rows. We need to add these partial sums column-wise to compute all the output pixels for the i^{th} row. We then need to start from the next row and again compute a similar set of partial sums. This is a very classical design that was originally proposed in [Kung and Song, 1981].

Figure 14.15 shows a representative scheme. We assume that the filter has 3 rows. We first compute the output pixels for $row(x_1)$ by sending three rows as input to the first convolutional block. We simultaneously feed in the filter weights (not shown in the figure) and compute the three 1D convolutions. We then need to add the three partial sums (computed by each 1D convolution) to get the final result $row(y_1)$. Then we stream the input rows $row(x_2)$ and $row(x_3)$ to the second convolutional block. Additionally, we stream $row(x_4)$ to this block. This convolutional block computes $row(y_2)$. We can similarly have many more cascaded convolutional blocks. It is also possible to reuse a block to perform multiple convolutions one after the other. This can be done by feeding two input rows back into the block once the convolution is done. Simultaneously, we stream in a new input row.

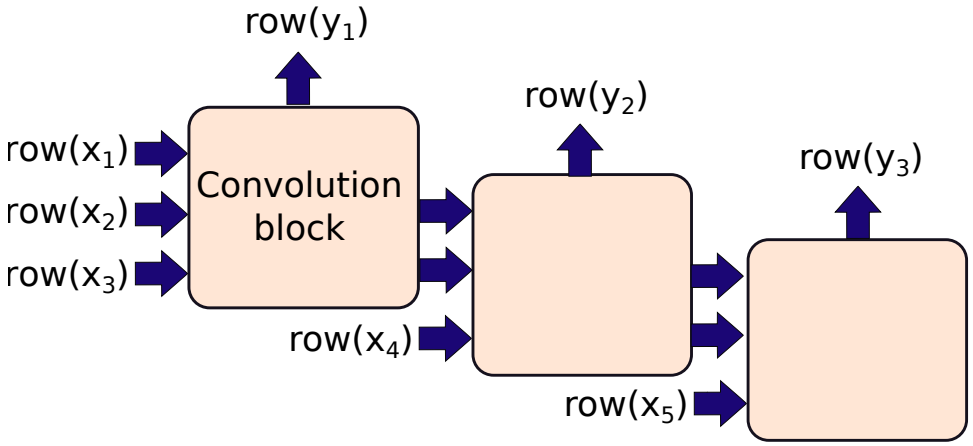

Figure 14.15: A set of systolic arrays for 2D convolution

Matrix-Vector Product

Convolution can also be represented as a matrix-vector product. Let us explain with a small example. Assume we want to compute the following convolution. We have a 2D weight matrix $\mathbf{W}$ with elements w_{ij}, and a 2D input matrix $\mathbf{X}$ with elements x_{ij}.

$$\mathbf{W} = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} \qquad \mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix}$$

The convolution $\mathbf{W} \star \mathbf{X}$ can be converted into a matrix-vector product as follows.

$$\mathbf{Y} = \underbrace{\begin{bmatrix} w_{11} & w_{12} & 0 & w_{21} & w_{22} & 0 & 0 & 0 & 0 \\ 0 & w_{11} & w_{12} & 0 & w_{21} & w_{22} & 0 & 0 & 0 \\ 0 & 0 & 0 & w_{11} & w_{12} & 0 & w_{21} & w_{22} & 0 \\ 0 & 0 & 0 & 0 & w_{11} & w_{12} & 0 & w_{21} & w_{22} \end{bmatrix}}_{\hat{\mathbf{W}}} \times \begin{bmatrix} x_{11} \\ x_{12} \\ x_{13} \\ x_{21} \\ x_{22} \\ x_{23} \\ x_{31} \\ x_{32} \\ x_{33} \end{bmatrix}$$

$\hat{\mathbf{W}}$ is known as a *doubly block circulant matrix*. Each row is generated by flattening the weight matrix and inserting an adequate number of zeros. Each subsequent row is generated by shifting the previous row by either 1 position or by 2 positions (relevant only for this example). The reader needs to convince herself that the matrix-vector product is indeed equal to the convolution of the matrices $\mathbf{X}$ and $\mathbf{W}$. We can extend this framework to compute a matrix-matrix product for calculating the respective convolutions of several input blocks in one go.

One of the advantages of this approach is simplicity. Moreover, matrix multiplication is a classical problem, and there are a lot of highly optimized hardware circuits to compute the product of two matrices. Such circuits can be incorporated into commodity processors that contain CNN accelerators. As a matter of fact, modern CPUs and GPUs have already started incorporating matrix and tensor processing units as of 2020.

These units can be used to compute such convolutions by converting them into standard linear algebra operations such as matrix-vector or matrix-matrix products. Even though we need to create additional

redundancy in this scheme by storing multiple copies of the weights, and we need to flatten matrices into vectors, sometimes using standard matrix multiplication hardware that has been rigorously verified and optimized is a worthwhile design decision.

14.4 Optimizations

We can perform a wide variety of optimizations while designing CNN accelerators. For example, we may have 0-valued weights or inputs. In this case, we need not compute any MAC operations that involve them. Such operations are known as *ineffectual operations*. In some other cases we have repeated weights or repeated input values. We can leverage such patterns to reduce the number of arithmetic operations that we perform.

Typically, a systolic array based system is rather rigid. It does not allow us to take advantage of such patterns in the inputs or in the weights. However, if we are using reconfigurable hardware such as FPGAs or if there is some scope for reconfigurability within our ASIC circuits, then we can leverage many such patterns. Let us look at some of the common methods that are used to reduce the computation time, and then we shall move on to techniques that optimize the usage of the memory system.

14.4.1 Reduction of the Computing Time

Common Subexpression Elimination

One of the common techniques in this space is common subexpression elimination. Assume that w, w' and w'' are filter weights. While computing the convolution we might encounter an expression of the form: $(wx_1 + wx_2 + w'x_3 + w'x_4 + w''x_5 + w''x_6 + w''x_7)$. In this case, we can group the elements as follows: $w(x_1 + x_2) + w'(x_3 + x_4) + w''(x_5 + x_6 + x_7)$. In this case, we can create circuits to compute the sum of the first two inputs elements (x_1 and x_2), the next two input elements (x_3 and x_4), and then the input elements from positions 5 to 7. By doing this we save a few multiplications. To compute $wx_1 + wx_2$, we need one addition and two multiplications. Whereas, to compute $w(x_1 + x_2)$, we need only one addition and one multiplication. We thus save a multiplication over here. The savings are even more if we have more repeated weights. Furthermore, the sum of the inputs that we compute can be used across filters if they also have a similar pattern of weight reuse.

Zero and Negative Values

Next, we can consider zero-valued weights or inputs. We can add additional circuits to detect those weights or inputs whose value is below a certain threshold. Subsequently, we can generate a mask that is essentially a bit vector, which contains a '1' for those positions that have a non-zero weight or input. We can only fetch those values into the PEs and then compute a convolution. Alternatively, we can fetch all the values into the PEs, and only compute convolutions for the non-zero values. Performing such optimizations using direct products is easier than other methods that rely on the fixed structure of hardware such as systolic arrays.

Most convolution layers feed their output to ReLU layers, which filter out all the negative values. If we are aware of this, then in the process of the convolution itself, we can terminate the computation and set the value of the output pixel to 0, if we are sure that the final output is going to be negative. To get a quick estimate of the final sign of the output, we can compute an approximate value only using the more significant bits of the inputs and the weights. This will give us an estimate of the final value. If we find it to be negative, then we can skip the entire convolution process.

One of the main criticisms of such approaches has been that the time taken to compute a convolution becomes nondeterministic. Since we need to wait for all the blocks to finish their convolutions, the slowest block determines the total computation time. This is not desirable. Hence, in this case we need

to dynamically apportion the work among the PEs such that all the parallel computations roughly finish at the same time.

14.4.2 Reduction of the Memory Access Time

Compression

In general, it is very easy to compress images because they have a fair amount of redundancy. For example, a large scenery might have a lot of blue pixels because they represent the blue sky. We can take advantage of such patterns and optimize this process. We can use traditional image compression techniques to compress the *ifmaps*, and then load compressed versions of these *ifmaps* into higher levels of memory such as the global buffer or the local buffers. When the input data is ready to be processed, we can uncompress the *ifmaps*. Such techniques effectively increase the memory bandwidth and reduce the time it takes to load a set of *ifmaps*.

One of the common techniques that is used in such compression strategies is base-delta compression [Wang et al., 2016]. Here, we compute the mean value for a block of filter weights or inputs. This is known as the *base*. Then we compute the offset of each value (the *delta*) about the mean. If all the values are clustered roughly in the same zone, then we need far fewer bits to represent all the offsets. For example, if the values are 10, 11, 12, and 13, the delta values will be -2, -1, 0, and 1 assuming the base is equal to 12. These values will only require 2 bits to be represented. We thus need a total of 8 (2×4) bits to store all the deltas and 4 bits to store the base, whereas we would have needed 4×4 (16) bits to store all the original values. We observe a space savings of 25%. Of course, this scheme does make the implicit assumption that all the deltas are in the range of [-2,1]. If the possible values of delta were more spaced apart from the mean, we might not have seen any savings in storage space.

Dynamic Detection of Zero-Valued Weights and Inputs

We can detect zero-valued weights and inputs either statically or dynamically. Almost always, the latter is preferable if we can find an efficient method. We can scan the weights or inputs and create a bitmap or a bit vector that stores a 1 if the corresponding weight or input element is non-zero. Such bitmaps can be stored in a separate region in memory and treated like an indexing structure. Whenever we need to read in an *ifmap*, we first access the index (in the bitmap or bit-vector) and only fetch those elements that have non-zero values.

Reduction of Precision

Different layers of the neural network do not need the same level of precision. We can reduce the precision of values in some layers if the resulting error is insignificant. Of course, this depends on the nature of the input, the nature of the features we are trying to detect, and the architecture of the network. Sometimes by studying the interaction of all three, it is possible to identify layers where the precision of the stored values can be reduced significantly. In this case, we can reduce both the computation time and the memory access overheads because the data values now are significantly narrower. One of the most important criticisms of such architectures is that the functional units are typically designed for a fixed operand width in mind. If the operand width is changed, then the functional units need to be changed as well. This is not possible in an ASIC architecture, and is also hard to implement in reconfigurable architectures because they typically incorporate fixed-width adders and multipliers. Hence, reducing the computation time with reduced precision is hard.

Bit-serial Multipliers

It is possible to use bit-serial multipliers that read in the input bit by bit to implement this functionality. The key insight here is that each of the two numbers that we wish to multiply (input and weight) can be represented as bit vectors. We claim that the product of two numbers is equal to the convolution

of the bit vectors. This is very easy to visualize if we think about how we actually multiply the numbers using the standard primary-school multiplication algorithm. Figuring out a proof for this is left as an exercise for the reader.

We can use a standard systolic architecture to compute the convolution. Since it is independent of the number of elements (operand width in this case), we can use it to implement a bit-serial multiplier that multiplies two operands bit by bit completely disregarding the width of the operands. The multiplier can thus scale to any precision. Even though we gain a tremendous degree of flexibility by using such bit-serial multipliers, the latency suffers. Such multipliers take $O(N)$ time, whereas a conventional parallel multiplier takes $O(log(N))$ time. Here, N is the operand width. This can be offset by the fact that such multipliers have a very low area and power footprint.

Bit-serial architectures can additionally make optimizations based on zero-valued bits. The key idea is to break a large n-bit number into several smaller m-bit numbers. For each of these numbers we compute the length of the prefix and suffix that contain all zeros. These are ineffectual bits, and need not be considered in the multiplication. We then multiply these m-bit numbers with the filter weight using a bit-serial multiplier, and then left-shift the result according to the length of the zero-valued suffix and the position of the m-bit number within the larger number. We can then use a traditional adder tree to add all of these partial sums.

14.5 Memory System Organization

Up till now we have only discussed the design of the compute engines of CNN accelerators. However, the design of the memory system is also equally important. This is because a large array of PEs also requires a very high memory bandwidth. We need to supply all the inputs, partial sums, and weights to the large array of PEs. This requires a very high-bandwidth connection to memory. Latency is not particularly important here. The most important metric that we are concerned with is the achieved throughput.

14.5.1 DRAM+SRAM based Organization

The traditional approach uses DRAM banks for off-chip memory, and SRAM technology for the global buffer and the on-chip local buffers. Consider the output stationary (OS) architecture presented in Figure 14.9. In such architectures we typically stream the inputs from one side, and stream in the filter weights from another side of the 2D array. Consider a 16×16 PE array. This means that we typically need to read 16 different input streams, and 16 different filter streams in parallel. The only way of sustaining such a high degree of parallelism is to have a 32-banked global buffer. We dedicate 16 banks to store the inputs, and we dedicate 16 banks to store the filter weights. Every cycle we access all the banks and stream in data from them.

There are two ways to manage the process of loading the local buffers and the global buffers. The first method is to use the traditional cache eviction and replacement mechanisms to fill any data that the local buffer or global buffer may require. This is the traditional *pull*-based mechanism. However, in the case of a CNN, since the entire computational process is deterministic, we can have a *push*-based mechanism where we have a dedicated finite state machine that loads data from the lower levels of memory at appropriate times and effectively acts as a data prefetching engine. This process can be automated where the FSM figures out the data access pattern automatically, or it can use hints embedded in the code. The latter approach is preferred in high-performance designs.

Let us show an example for an OS architecture. Its mapping was as follows.

$$n \triangleright h' \| w' \| \triangleright k \| c \triangleright r \triangleright s \triangleright h'' \triangleright w''$$

Listing 14.2 shows a piece of code (using a traditional software paradigm) that represents this mapping. The code does not capture the finer aspects of the data flow; we use it for explaining the role of

prefetching. It introduces the *parallel for* statement to represent the $\|$ operator. h' is represented by $h1$ and h'' is represented by $h2$. We have omitted the code to prefetch data. For the ease of readability, we have put comments in its place.

Listing 14.2: 2D Convolution with directives to load data from memory

```
for (n=0; n<N; n++) {  /* input images */
// load the nth image I[n] in the DRAM

 parallel for (h1=0; h1<H; h1+= Th) {/*rows: ifmap/ofmap */
  parallel for (w1=0; w1<W; w1+= Tw) {/*cols: ifmap/ofmap */
// load the output tile at O[n][k][h1][w1] (for each k) in a separate LB

   parallel for (k=0; k<K; k++) { /* outputs */
    for (c=0; c<C; c++) { /* input channels */
// load the filter F[k][c] in the GB
// load the input ifmap at I[n][c] in the GB

     for (r=0; r<R; r++) { /* rows of the filter */
      for (s=0; s<S; s++) { /* cols of the filter */

       for (h2=h1; h2<h1+Th; h2++) {
        for (w2=w1; w2<w1+Tw; w2++) {
         O[n][k][h2][w2] += I[n][c][h2+r][w2+s]
                          * F[k][c][r][s];
}}}}}}}
```

The key point to note is the comments that we have added for loading data into different memory structures. We can at the beginning load an entire image into the DRAM. Since we keep the output pixels stationary in the local buffers (LBs), we can immediately load them once we have partitioned the pixel space. They remain stationary until we move to a new set of output pixels. Then, when we change the channel, c, we load the corresponding input pixels and filters into the GB. Such hints can be made more sophisticated, and we can prefetch values into the GB, and also add a streaming component – as old data leaves, new data moves in to take its place. Something similar can be implemented in hardware as well. It needs to capture the finer aspects of the data flow and accurately synchronize the data transfer with the computation.

Traditional DRAM memory typically proves to be a bottleneck in terms of bandwidth. Hence, for implementing CNNs we typically prefer high-bandwidth memory technologies such as embedded DRAM, HBM memory, or Hybrid Memory Cubes (see Section 10.5.6). We typically assign a PE to each vault. We can think of a vault as a high-bandwidth bank in 3D memory.

14.5.2 Processing in Memory

Till now, our approach has been to stream values to the array of PEs, compute the results, and write them back. Processing in memory (PIM) approaches propose to take the computation to the memory system. This means that we make the memory *smarter* such that we can easily perform addition or multiplication operations in situ. We can use traditional memory technologies such as SRAMs, and DRAMs, or even use modern NVM based technologies.

Overview

The basic idea is the same for all the PIM designs. To compute a dot product between two vectors, we need to first perform an element-wise multiplication, and then add the partial sums. Let us consider a typical array-based memory design, where each row of memory cells is activated via the word line, and

each column of memory cells is connected to at least one bit line. Let the voltage on each word line correspond to the value of an input. The assumption is that the weight is embedded within the memory cell, and we use a column of cells to compute a single dot product between a vector of inputs and a vector of weights.

There are two broad paradigms in this space: *charge sharing* and *current summing*. In the charge sharing approach each memory cell has a capacitor, whose stored charge is proportional to the product of the input and the weight. This is done for all input-weight pairs, and it is further assumed that all such capacitors are connected to the same bit line via switches (see Figure 14.16(a)). Then, to add the values we just need to connect all the capacitors to the bit line by closing all the switches. The stored charge will redistribute. Since for each capacitor $Q = VC$ [1], we can think of this as a multiplication operation. There are two design choices here. Either we can keep C the same and use an analog multiplier to generate V or think of C as the weight and V as the input. Regardless of the design choice, the charge Q_i for memory cell i represents the product of an input pixel and a weight. When we connect all the switches we shall have $\sum Q_i = V_{bitline} C_{lumped}$. Here, $V_{bitline}$ is the voltage of the bit line, and C_{lumped} is the lumped capacitance of the entire set of memory cells and the bit line – this is known a priori. Hence, the voltage on the bit line can be a very good estimate of the dot product. This can be measured with an ADC (analog to digital converter).

Next, let us consider the current summing approach. If we look at the basic Ohm's law equation, $V/R = I$, here also we are performing a multiplication between the voltage and the conductance $1/R$. If there is some way to configure the conductance then we can realize a multiplication operation between an input (voltage V) and a weight (the conductance). Furthermore, if we add the resultant current values, then we effectively realize an addition operation. The magnitude of this current can again be detected by measuring the voltage across a register using an ADC (see Figure 14.16(b)).

In both cases, we perform an approximate computation where we get an estimate of the dot product. In such cases, we are embedding the weight into the memory cell either as a capacitance or a conductance. It is possible to use modern nonvolatile memory technologies to also dynamically configure these values. There are criticisms for both these approaches. Any analog computation of this nature is associated with a certain degree of error caused by noise and process variation. Hence, for such architectures designers typically use either binary weights, or significantly reduce the precision of the weight values. Over the years many optimizations have been proposed to get acceptable accuracies with such neural networks that use such reduced weights. Furthermore, many advances have been made particularly over the last five years (as of 2020) to increase the noise tolerance. Charge sharing approaches are in general more tolerant to noise than current summing approaches.

Implementations

We typically use a voltage multiplier circuit for charge sharing based schemes and a variable resistance in a current summing based scheme.

Let us first look at options in traditional CMOS logic. For binary-valued inputs and weights, we can use an AND gate to effect a voltage multiplication. Another way of creating such a multiplier in traditional CMOS logic is to operate a transistor in the linear mode of operation. In this case, the current is proportional to the drain-source voltage. This mechanism can be used to create a configurable current source. We can alternatively keep the drain to source voltage constant, and instead vary the gate voltage. This will change the drain current, which we can approximate as a linear relationship. This approach is typically preferred while using regular CMOS transistors where the word line voltage is set to be proportional to the input. We use a DAC (digital to analog converter) to generate these voltages for each word line. We can then use any of the approaches – charge sharing or current summing – with appropriate modifications.

Nonvolatile memory technologies such as resistive RAMs (ReRAMs) are ideally suited for this purpose. Here, we can easily vary the resistance as discussed in Section 10.5.5. Then we can use the current

[1] Q is the charge, V is the voltage, and C is the capacitance

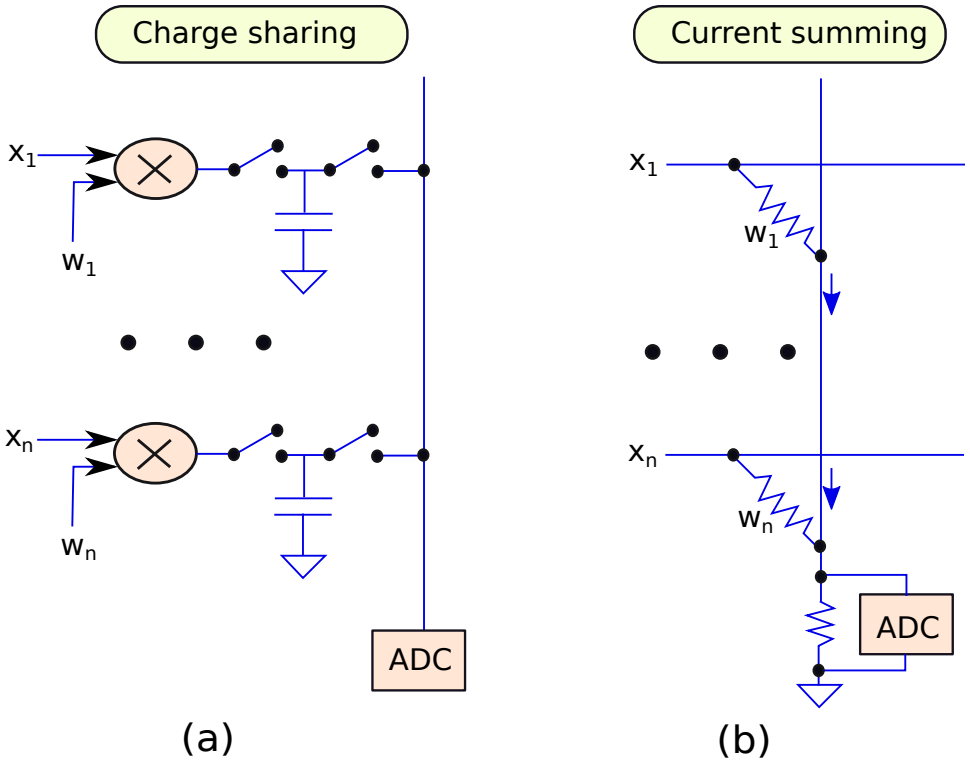

Figure 14.16: (a) Charge sharing, (b) Current summing

summing approach. Given that such devices are very easily configurable, we can change the weights at run time. Researchers have created variable resistance states with almost all known NVM devices and used them to realize such computations. Such circuits are also known as *neuromorphic circuits*.

In all such architectures, dealing with negative weights is not possible. However, they can be easily incorporated into such designs by computing two dot products: one with positive weights (negative weights are zeroed), and one with negative weights (positive weights are zeroed). Subsequently, we can subtract the second dot product (one with negative weights) from the first one (one with positive weights).

14.6 Summary and Further Reading

14.6.1 Summary

Summary 13

1. *In any learning problem, we try to figure out the relationship between a set of inputs and the corresponding outputs. We can either assume a linear relationship or a nonlinear relationship.*
2. *Since the relationship is not known a priori, we typically use a class of learners known as universal approximators, where we simply need to change the parameters to realize different functions. The aim of the learning problem is to learn these parameters.*

3. *Early approaches were based on linear and nonlinear regression, where it was assumed that the relationship is a polynomial curve and the main aim was to learn the coefficients.*

4. *Neural networks are one of the most popular universal approximators that are composed of a set of layers. The input to a layer is called an* ifmap*, and the output is called an* ofmap*. A layer typically takes in multiple* ifmaps *as inputs and generates multiple* ofmaps *as outputs. Each element of an* ifmap *or an* ofmap *is called a* pixel*. The layers either compute a linear function over the inputs (outputs of the previous layer) by multiplying the input vector with a dedicated weight vector, or by computing a nonlinear function over the inputs such as the Sigmoid function, the tanh function, or the ReLU function. In modern deep neural networks we can have hundreds of such layers with millions of weights.*

5. *In convolutional neural networks (CNNs) we typically consider 2D* ifmaps *and* ofmaps*. We avoid costly vector or matrix products, and instead compute the convolution of a set of* ifmaps *with a very small weight matrix known as the* filter*. To reduce the size of the inputs, we perform an operation known as* max pooling*, where we replace a set of pixels with its maximum value for translational invariance. A typical CNN consists of four types of layers: convolutional, max pooling, ReLU, and a traditional fully connected linear layer that computes a dot product between an input vector and an equal-sized weight vector.*

6. *The operation of the CNN can be represented as a nested loop with seven iterations. Some of these iterations can further be tiled, and also be parallelized across a set of functional units known as processing elements or PEs. We designed a custom notation to represent such computations.*

7. *In this notation, the symbol $\|$ indicates that the loops of the iterator preceding it can be parallelized, and the operator $\triangleright$ refers to sequential execution.*

8. *We initially proposed a software model where we model each PE as a separate thread that has some local storage space. We can decide to keep some data stationary within the local storage space. In this space we proposed all kinds of architectures: input stationary (IS), weight stationary (WS), output stationary (OS), and row stationary (RS).*

9. *We can simply take the software abstraction and map it to hardware, where each thread is a separate processing element. In this case, we need to consider the connection between the PEs. The PEs are typically arranged as a 2D matrix interconnected via an on-chip network. Depending upon the type of the architecture we stream in one kind of data from one side (inputs, filters, etc.) and another kind of data from another side. Additionally, we also have the option of storing some data within the local buffers of each PE. We can realize all the four kinds of architectures using such hardware designs.*

10. *We typically distribute work at a coarse grain among the PEs because it is very hard to make all the PEs work in lockstep. To further achieve the benefits of parallel execution we can leverage intra-PE parallelism. In general, to compute a 2D convolution, we break it into a series of 1D convolutions.*

11. *We first discussed semi-systolic arrays, where we have a set of combinational units (CUs) arranged in a linear sequence. In every cycle we can either broadcast a value to all the CUs, or make a set of values flow between adjacent CUs. Because of the lockstep nature of the execution such execution patterns are known as* systolic execution *patterns.*

12. *In a semi-systolic architecture we at least have one pair of adjacent CUs without registers in between. This makes it hard to maintain the timing, manage clock skew, and also increases*

the cycle time. Hence, we may prefer systolic architectures, where we always have intervening registers between a pair of adjacent CUs. We can use the Retiming Lemma to convert a semi-systolic architecture to a systolic architecture. However, while doing so, it is typically necessary to introduce k stall cycles every time we stream in one input. This slows down the computation by a factor of k, and additionally we need to add many extra registers throughout the array. The problem of stall cycles can be solved by solving multiple problems concurrently. We can then clock such systolic networks at a very high speed without wasting cycles.

13. *In many reconfigurable architectures we use a direct method, where we load large overlapping sections of the* ifmaps *into different memory arrays at the same time. For each array, we have a dedicated set of MAC units and a tree of adders to compute a dot product with the filter weights. This approach does require more resources because of the added redundancy, however, it is fast and simple to implement if we have the required hardware.*

14. *We can also convert a convolution into a matrix-vector product. The* ifmap *can be flattened into a vector, and we can convert the filter matrix into a* doubly block circulant matrix*. Their product is equal to the 2D convolution.*

15. *For designing a memory system we typically use a large multi-banked global buffer. We can alternatively use modern 3D stacked memory technologies such as High Bandwidth Memory (HBM) or Hybrid Memory Cubes (HMC). They are integrated into the same package and connected using an interposer.*

16. *We can make memory cells smarter and use them to compute dot products. Two common approaches to compute dot products using an array of memory cells (traditional or NVM) are the charge sharing and current summing techniques. This is an analog computation that yields an estimate of the dot product of an input vector (typically expressed as word line voltages) and the weight vector (typically embedded as a conductance or capacitance within each memory cell).*

14.6.2 Further Reading

To understand all the concepts presented in this chapter and to pursue research in this area, it is necessary to first get a good idea of machine learning. Readers should first consult a standard textbook on machine learning such as [Bishop, 2006]. Before moving on to deep learning, it is important to thoroughly understand linear algebra [Cooperstein, 2015], and also understand matrix calculus [Turkington, 2013]. Subsequently, readers can move on to the classic text on deep learning by Goodfellow, Bengio, and Courville [Goodfellow et al., 2016].

After appreciating deep learning, the next step is to cover some basic literature on systolic arrays. Most of these papers were published in the early 80s. Some early papers in this area are as follows: [Kung and Song, 1981,Kung, 1982,Kung and Picard, 1984,Ersoy, 1985,Kwan and Okullo-Oballa, 1990]. A large number of semi-systolic and systolic architectures are covered in the book by Leighton [Leighton, 2014]. Please refer to this book to learn more about the Retiming Lemma. The paper by Lam discusses algorithms for designing a compiler for systolic architectures [Lam, 2012].

Now, coming to architectures for CNNs, the reader should first take a look at the survey paper by Moolchandani et al. [Moolchandani et al., 2020] and follow up with the e-books by Reagen [Reagen et al., 2017] and Sze et al. [Sze et al., 2020]. Then interested readers can read a few highly cited papers in this area such as the papers on Neuflow [Farabet et al., 2011], Diannao [Chen et al., 2014], Cnvlutin [Albericio et al., 2016], Flexflow [Lu et al., 2017], Google TPU [Jouppi et al., 2017] and Eyeriss [Chen et al., 2016].

Some recent progress has been made in designing software simulators for hardware neural networks

(the Scale-Sim project [Samajdar et al., 2020]), and in modeling the power and performance of CNNs [Wu et al., 2019, Parashar et al., 2019, Kwon et al., 2018]. These tools can be used to quickly estimate the overheads of implementing different CNN architectures in hardware.

Exercises

Ex. 1 — Implement a CNN accelerator using a hardware description language such as Verilog or VHDL.

Ex. 2 — Create an architecture for an RNN or LSTM using the techniques that we have learned in this chapter.

Ex. 3 — In this chapter, we described the architecture for inferencing. Create an architecture for neural network training.

Ex. 4 — Design and simulate a ReRAM based convolution layer.